ON-LINE RANDOM FOREST

A DISSERTATION SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF MASTER OF SCIENCE
IN THE FACULTY OF ENGINEERING AND PHYSICAL SCIENCES

2014

By
Germán Alfonso Chaparro Alvarez
School of Computer Science

Contents

Word Count: 17,250

List of Tables

List of Figures

List of Algorithms

Abstract

ON-LINE RANDOM FOREST
Germán Alfonso Chaparro Alvarez
A dissertation submitted to the University of Manchester
for the degree of Master of Science, 2014

This dissertation presents the complete process of design, implementation and test of machine learning model based on on-line decision trees which main goal consists in learning from data streams. This is a supervised learning model that changes the paradigm of having training and testing phases and proposes an on-line learning method in which the model learns while the data continuously arrives. The selected model is called On-line Random Forest, and the final algorithm was applied over several machine learning datasets and the results show how the error rate decreases while more learning data is being processed. Moreover a software application was built over the model in order to detect faces from different sources such as videos taken from traditional webcams or bank of images stored in folders in a file system. The application can be used as learning tool for future students in machine learning and computer vision fields. The on-line learning models have many real world applications: since the traditional email categorization problem, to the real time trading algorithms in financial sector, and surveillance systems in security companies are just a few examples.

Declaration

No portion of the work referred to in this dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

I would like to thank my supervisor and mentor, Dr. Gavin Brown for giving me the opportunity to work in my favourite field of Computer Sciences: Machine Learning. His remarkable teaching and coaching strategies challenged me to give my best since the first day I started working on this project. Thank you Gavin for teaching me how to love and enjoy your proper work.

Special thanks to the School of Computer Science of the University of Manchester, especially to my teachers Dr. Uli Sattler, Dr. Mikel Lujan and Dr. Ke Chen for offering me their help and support in the most difficult moment of my life. Without them this dream would not have met.

I would also like to express my gratitude to the Colombian Securities Exchange and to COLFUTURO for supporting me with the scholarships which financed my tuition fees and living expenses during this year.

For all the unwavering support and encouragement, thanks to my friends, my beloved family and my dear Helena.

Dedication

From my heart...
to my father, you are in heaven but I know you are taking care of me...
to my mother, all I am is because of you...
to Gustavo, Tatiana and Juli, I will always take care of you...
to Helena, my present, my future and my source of inspiration...

Chapter 1

Introduction

Off-line decision trees algorithms have proved high accuracy in the classification of data in several types of applications: data recognition, clustering, segmentation, organ detection, etc. These types of trees demand to have training and testing phases to be able to learn before their final version could be released to production environments.

A new generation of decision trees, On-line decision trees, have been studied in order to satisfy the need of learn and process data in real time. On-line decision trees change the paradigm of having training and testing phases into to a unique phase in which the information "arrives" on-line (or streaming) and the model must learn and process it as fast as possible to be ready to process the next arriving data. In order to maintain the accurate behaviour of the off-line decision trees, the on-line decision trees have based their algorithms in their predecessors but with some modifications that allow them to learn and process on-line data.

The ensemble methods are a good technique to improve the performance of traditional models, so this research bases the work in building an ensemble model: On-line Random Forests. The On-line Random Forest is applied by [25] to demonstrate their good performance by solving visual tracking and interactive segmentation problems. The final motivation of this dissertation is oriented to find a new way to solve traditional machine learning classification problems including the face detection issue by using on-line random forests.

Moreover, I have the intention of building a software that could be used as a learning tool by the students of the School of Computer Science of the University of Manchester and in general to those that want to emphasize their knowledge in Machine Learning techniques. This tool could be used to learn and apply the concept of on-line decision trees and could be improved in time by applying brand new techniques to solve

this and several related problems.

1.1 Aim

Develop and evaluate a machine learning model based on On-line Random Forest technique able to solve classification problems, including the issue of face detection in images taken from a bank of images and from videos.

1.2 Objectives

- **Learning Objectives**
  - Investigate and understand the concept of On-line decision trees.
  - Review how On-line decision trees can be applied to actual Machine Learning related problems such as classification, object segmentation, clustering, detection, etc.
  - Comprehend and go deeper into the topic of On-line Random Forest and its relation with On-line decision trees.
  - Investigate and understand the procedure of feature extraction of faces from an image.
- **Deliverable Objectives**
  - Develop and implement in a software application an On-line Random Forest algorithm.
  - Understand and implement statistical functions to be used as splitting criteria once the data is being treated.
  - Measure the application results in several scenarios such as well known publicly machine learning datasets, faces datasets and videos taken from traditional webcams.
  - Analyse previous results and give a concept of the performance of the application based on the classification error and the performance.

1.3 Report Outline

This report is structured as follows: chapter 2 shows an overview of the history of works related to the on-line learning models and makes an approach of decision trees in off-line and on-line learning environments. Then chapter 3 exposes the background related to face detection by referencing several techniques used for that issue and finishes with the theoretical framework related to the use of image processing techniques to detect faces features. Chapter 4 develops the on-line random forest algorithm which is the core of this research and exposes how this algorithm is integrated with the face detection techniques. The details of the design and implementation of the software solution is explained in chapter 5. The results of the experiments, some screenshots of the final software program and the evaluation of the algorithm follow in chapter 6. The conclusions and future work are exposed in chapter 7. The report concludes with appendix A which has the user manual of the program.

Chapter 2

Machine learning background

This chapter shows the background related to the on-line random forest model used as algorithm to solve classification problems.

2.1 Decision tree models

The algorithm is based on the supervised learning protocol [12] in which the model should evolve according to patterns of known labelled samples in order to predict the labels of unknown incoming records. Mathematically a sample can be defined as tuple (x,y) where x is a n-dimensional vector or *feature vector* that represents all the features of an object or sample and y is the class or *label* of the sample. A dataset is collection or set of samples, so it can be represented as $D = (x_1,y_1),...,(x_n,y_n)$ where (x_i,y_i) is the i-th sample of the dataset.

2.1.1 Off-line learning models

Off-line machine learning models are characterized mainly because there is a clear separation between the training and testing stages in the whole process. So in its most basic representation the model should be built or *trained* based on training algorithm with one training dataset D_{TRAIN} and is *tested* with the testing dataset D_{TEST}. Is relevant in this point to say that D_{TRAIN} and D_{TEST} should not share samples in order to guarantee a good quality testing process. Mathematically this can be expressed as $D_{TRAIN} \cap D_{TEST} = \varnothing$. The objective of the training phase consists in fitting the model according to D_{TRAIN} and the test phase sends D_{TEST} to the model in order to compare each predicted label with its corresponding label in the testing sample. In figure 2.1

the labels of the arrows show the flow sequence of the complete training and testing process with a supervised learning methodology.

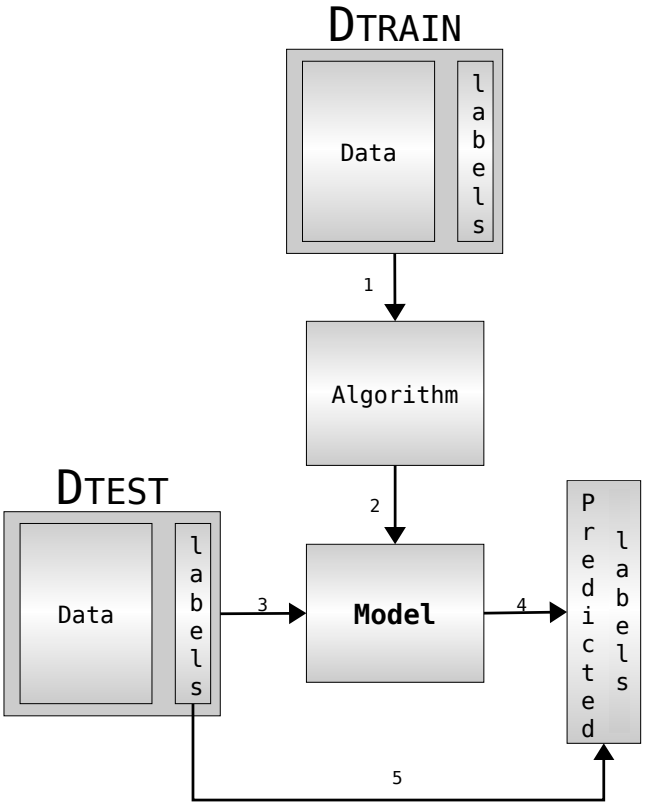


Figure 2.1: Supervised learning pipeline: arrows 1 and 2 show the flow of the training phase, arrows 3 and 4 show the flow of testing phase, arrow 5 shows the comparison of predicted and true labels to evaluate the performance of the model

The ratio between the number of misclassified samples over the total number of samples is called the classification error or *error rate* of the model. The execution of cross validation techniques, as explained in [12], helps the scientist to check the stability of the model. Other very informative measurements such as true positive rates (also called sensitivity, hit rate or recall) and false positive rates (well known as false alarm rate or 1-specificity), could be done in order to perform a Receiver Operating Characteristics (ROC) analysis [16] and visualize the performance of the model.

For this models, if new data for training arrives after the model is already trained, tested or deployed in production environments, the complete process must be done again to include the old and the new training data in a new model.

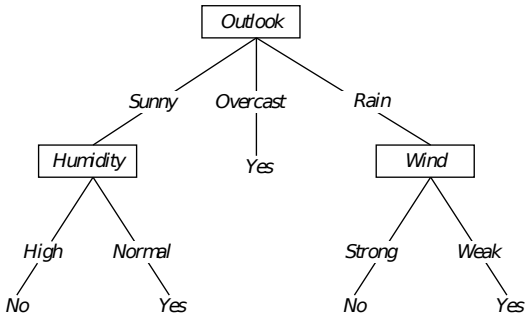


Figure 2.2: Example of decision tree using categorical values as features

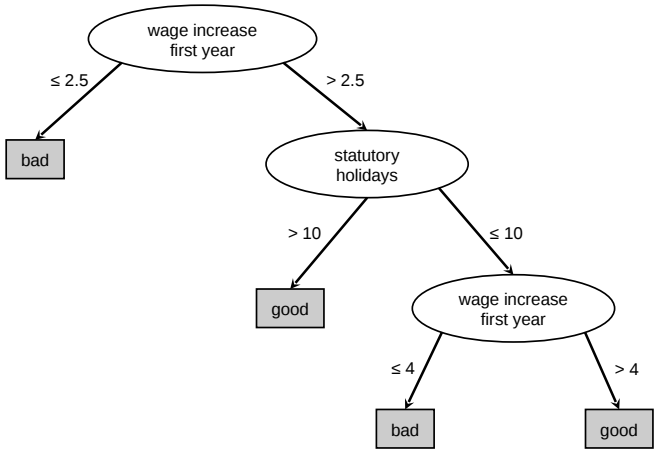


Figure 2.3: Example of decision tree using numerical values as features

2.1.1.1 Decision Trees

Decision trees are very powerful machine learning models able to solve classification problems, including those that are non-linearly separable [12]. The internal nodes of the tree (those nodes that have children) have a test function that is applied over the features of the arriving samples, and the terminal nodes (nodes without children) or *leaves* of the tree have the labels that are assigned as category or prediction of the model in a classification problem. The model is flexible in the sense that is able to work with categorical or numerical as values for the features. Figure 2.2 shows a tree made of categorical values, figure 2.3 has a tree of numerical values and figure 2.4 shows a tree combining numerical and categorical values in the features[1].

A very important characteristic of these trees is its *maximum depth*, this means the number of levels a tree has counting from the root node to its deepest leaf. In

[1] Based on The Weather problem and Labour negotiation example in [19]

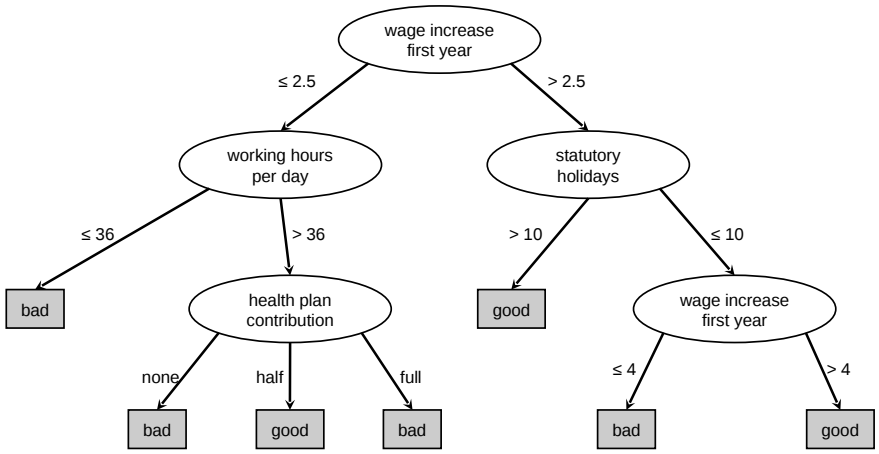


Figure 2.4: Example of decision tree using numerical and categorical values as features

machine learning the maximum depth is used as a parameter of the algorithm and help us to control overfitting (big depth selected) or underfitting (a small value for depth is chosen). So several experiments should be done in order to get its optimal value.

The training process of this type of models is summarised in the execution of algorithms able to build decision trees using the data of D_{TRAIN}. *ID3* exposed in [23] and *C4.5* in [24] are two of the most well known algorithms and have been used as base to build new more sophisticated tree classifiers. These algorithms propose building the tree by using a recursive function in which in each iteration D_{TRAIN} is evaluated among all the features by applying a quality function (an entropy function in the case of ID3 and normalized information gain in the case of C4.5). The feature with the best value of quality function is selected as winner and will be used to create a node that contains the feature with the split value. Based on the split value, D_{TRAIN} is divided into two smaller datasets that will be used to create the new two children nodes. The recursive process finishes when all the elements in the subset have the same class, or when the maximum depth is reached. In that case the predicted label will be the most common label of the samples that reach this final node.

Once the training process has finished, this means the tree is built, the testing process begins by sending each sample of D_{TEST} to the tree. In this process each sample should look for a leaf of the tree by comparing the values of its features with the values of the features that were set up in the nodes of the tree. Once the sample reaches one leaf of the tree, the most common label of that leaf is thrown as the predicted value for that sample. Finally the comparison between the predicted values against the labels of the samples is done to count the number of errors for that D_{TEST}. The ratio between

that number of errors and the size of D_{TEST} measures the test error rate.

2.1.1.2 Random Forest

The first definition of a Random Forest is done in [11] and fits into the category of *Ensemble* models. Basically and ensemble model is the one that base its result on the combination of outputs of other models. As is said in [12] "The principle is that the committee decision, with individual predictions combined appropriately, should have better overall accuracy, on average, than any individual committee member".

With that idea in mind a Random Forest could be abstracted as an ensemble model formed by a set of decision trees. So, training the forest consists in training the each tree in the set, and testing the forest consists in testing each tree, collecting all their answers and apply a function that combines those answers to give a final result. Some example of the final combination function could be a voting process in which each tree returns its solution and the final decision of the forest is decided by the majority vote.

The complete process of training a forest involves adding some modifications in the training process of the trees with the main goal of having completely different trees in the ensemble. It does not make sense to have a set in which all the elements are exactly the same tree. So, two modifications are done in the training phase:

- The first one is known as *Bagging* [10], and in a very rough way it consists in changing slightly the training dataset for each tree in the forest. These changes are based in choosing randomly samples of the dataset to replace some other samples of the same dataset. In this process a sample could be chosen more than once or *with replacement*. Mathematically is explain as follows: if we have D_{TRAIN} of size S, and is required to build a forest of N trees, N *bootstraps* of size S are build by picking randomly for each bootstrap S samples from D_{TRAIN}. Finally B is generated where $B = \{D_{1_{TRAIN}}, ..., D_{N_{TRAIN}}\}$ in which $D_{i_{TRAIN}}$ is the bootstrap that will be send to the $i-th$ tree of the ensemble. Moreover the sizes of all the bootstraps are the same, that means $size(D_{i_{TRAIN}}) = size(D_{j_{TRAIN}}) = size(D_{TRAIN}) = S, \forall i, j \in [1, N]$. The basic steps of this process are shown in algorithm 1. A very good explanation of this process can be found also in [12].

- The second change consists in selecting randomly K features at each split point instead of working with all the features of the sample. The selection of the best feature could be done as suggested in the algorithms ID3 or C4.5 as was

Algorithm 1: Bagging algorithm

input: D_{TRAIN}, S
return $B = \{D_{1_{TRAIN}}, ..., D_{N_{TRAIN}}\}$
for *i = 1 to N* **do**
    D_{temp} = CreateBootstrapWithReplacement(D_{TRAIN}, S)
    $B_i = D_{temp}$
end
return B

explained in section 2.1.1.1. The figure 2.5 shows different configuration of decision trees in a random forest.

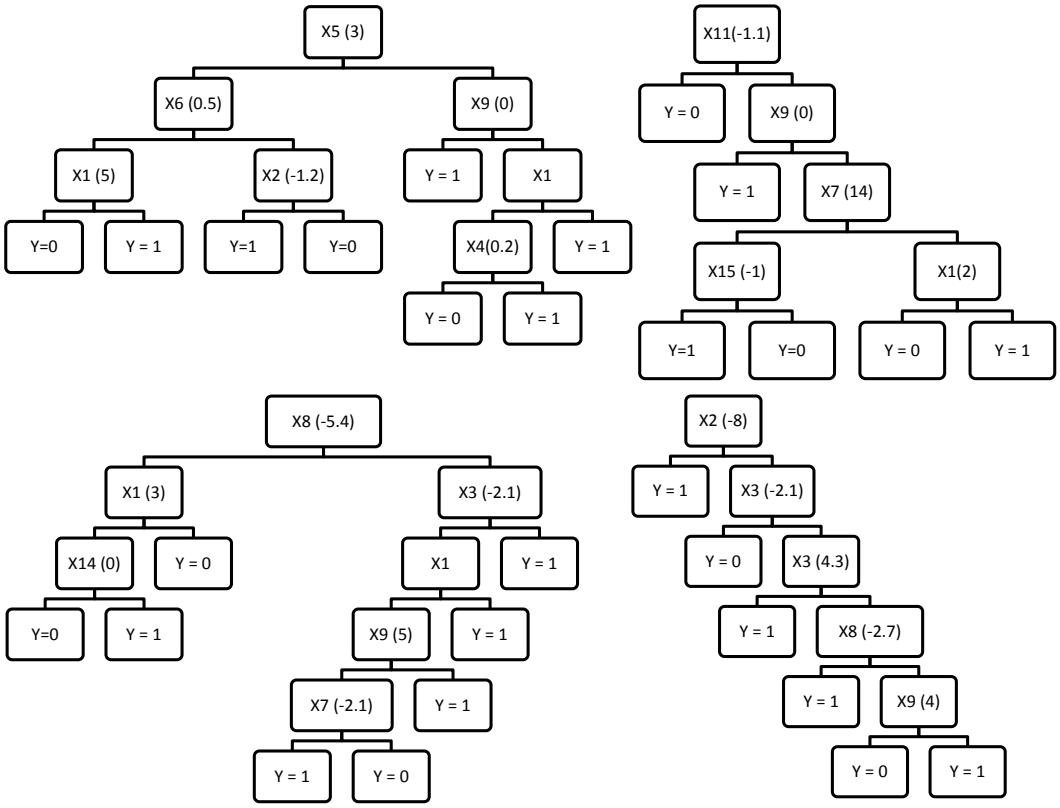


Figure 2.5: Possible configuration of Random Forest with 4 decision trees

2.1.2 On-line learning models

The on-line learning definition we used for this research, is related to the way a machine learning model classifies or predicts the class of the samples with better accuracy while more learning samples arrive to the model. In this case, the training data arrives

to the model in any stage of the process, so the model should be able to fit its internal configuration when processing the new information without "forgetting" the information learned from previous samples. In this type of models is expected that the error rate decreases while more training data arrives to the model. In this case the model is able to learn from datasets of huge proportions (big data) or theoretically datasets with infinite size.

The main disadvantage of these type of models consists basically in the low accuracy of the the results in the early stages of the learning process. Because of the model becomes *smarter* while more data arrives to it, the results are comparable against its off-line version only until a reasonable amount of data is processed.

2.1.2.1 On-line Decision Trees

The on-line decision trees are those decision trees able to learn from data that arrives sequentially, this means there is no way to have the complete training dataset in a given moment of time to start officially the training process. Because of that generally these trees use interfaces such queues to get the training data. So the main difference with its off-line version lies on the impossibility to load the complete dataset in memory to scan all the possible values of the features and decide which is the best one to split a node. This scenario obligates to build techniques able to determine a good moment to split the nodes based on the information that have passed through them.

[15] reaches this goal by doing a learning system based on the Hoeffding tree algorithm. In this model the decision of splitting is done by calculating the true statistical average x of a random variable with probability $1-p$ is $\widehat{x}-e$ with $e=\sqrt{\frac{ln\frac{1}{p}}{2n}}$ where n is the number of samples that have reached the node and $\widehat{x}$ is the calculated average of the random variable. In this research is shown that this technique determines with a very high probability with only few samples, that the selected value of feature to split a node is the same value of feature when having all the dataset in memory.

[25] uses a split criterion based on two parameters: the first one, α, indicates the minimum number of samples should visit a node; and β indicates the minimum gain over the randomly chosen features according to quality measures such as Entropy or Gini Index. The gain of the final selected feature should be the greatest between the gains of all the selected features and, of course, greater than β. So, only until the process reaches the values of α and β the node is split.

[33] considers the case in which the arriving data has very few labelled samples.

So the strategy to split consists in saving those samples temporally in the leafs of the tree until the quality function (in this case is Gini) applied over those samples returns a value higher than a parameter Δ. Once this condition is met, the new nodes are created as child of the current node and the previously saved samples are deleted.

2.1.2.2 On-line Random Forest

In the same way an ensemble of off-line decision trees are combined together to build the off-line random forest and produce a more accurate prediction or classification results, a ensemble of on-line decision trees can be combined to have a much more reliable model: the On-line Random Forest.

The on-line decision tree model already solved how the nodes of the trees must be split according to the sequentially arriving data, so here the main issue lies on the fact of how to send slightly different amount of data to each tree of the ensemble to fit the definition of Random Forest. In other words, how to replace the bagging process (explained in section 2.1.1.2) that is executed in the off-line version of the random forest.

For this issue, [20] and [21] created a strategy in which sequential arriving data is modelled as a Poisson distribution. The first step to reach this point consists in modelling the traditional bagging process as a binomial distribution of equation (2.1), in which K is the number of times the same sample is in the bootstrap.

$$P(K=k) = \binom{S}{k}\left(\frac{1}{S}\right)^{k}\left(1-\frac{1}{S}\right)^{S-k} \tag{2.1}$$

Let's remember that in a on-line learning model the size of the dataset is unknown and theoretically could be infinity, this means $S \to \infty$, so finally the equation (2.2) shows how K can be approximated as a discrete probability function with Poisson distribution $Poisson(\lambda)$ with $\lambda = 1$.

$$\begin{aligned} K &\sim \frac{exp(-1)}{k!} \\ K &\sim Poisson(1) \end{aligned} \tag{2.2}$$

So the algorithm 2 shows the steps to calculate the number of times a sample A must be sent to each tree in the ensemble.

Algorithm 2: On-line Bagging algorithm

```
Data: Sample A
for each tree t in the ensemble do
    k = Poisson(1)
    for i = 1 to k do
        sendSampleToTree(A, t)
        i++
    end
end
```

2.2 Previous work

In the original version of the random forest the bagging process is done by re-sampling randomly the examples from the original dataset to produce several training datasets or bootstraps, and each of those new datasets will be the input for each tree in the forest. So each tree will be trained by a bootstrap of the original dataset. In the on-line version of the random forest, the bagging process is not used because of the of the meaningless of knowing the size of the dataset (now the input is a stream of data so we don't how many samples will arrive). [21] and [20] propose the on-line version of the bagging process. The authors of those articles prove that if the number of samples tends to infinite (that is the case of a data streaming) the bagging process could be replaced by calculating how many times each single example must be sent to each tree in the forest. This number of times is calculated based on a Poisson distribution. Their results show that the on-line model behaves similar (and identical when the dataset is large) to the original off-line version. The papers use the Incremental Tree Inducer (ITI) algorithm to test and evaluate the results.

Another on-line model based on decision trees is explained in [5]. The technique is called Online Adaptive Decision Trees which is a neural network with the topology of a binary tree. The authors develop this algorithm taking advantage of the properties of the neural networks related to process information in on-line mode. Some characteristics about this model are: it uses the gradient descent learning algorithm due to is one of the traditional algorithms used for training neural networks, each of its nodes stores a decision function and an activation function, and the depth of the tree is the only parameter that affects the performance of the model. Their results show that underfitting could be present if the depth of the tree is low, but overfitting is not present while the depth of the tree is increase.

The article cited in [33] uses a Extremely Random Forest model to commit on-line

learning and execute tracking tasks over videos. The authors of this article also explain the need of treat data streams as source samples for the learning process, and their technique is based on the idea of expanding trees with very few examples. To reach this goal they save the samples in the leaf nodes of the trees, and using the information of the samples in combination with the Gini Index they will decide to split the node into two new ones. The results exposed in this paper are quite impressive by showing how tracking activities are done with very few samples.

Finally [25] shows a novel algorithm to create an On-line Random Forest and compares its performance against an On-line Adaboost algorithm. As in [21], the authors proposed to replace the bagging process used in the off-line version of random forest by calculating the number of times each sample should be sent using a Poisson distribution. They use an extremely randomized forest with a growing procedure based on the statistics that are taken when the stream of data is arriving to the model and consequently to some nodes in the tree. The authors also make difference between the terms *on-line* and *incremental* arguing that in the on-line mode the samples are not stored in the tree while the incremental mode does, so this is one of the most important differences with [33]. The code was implemented in c++ and executed in a GPU. The model was executed to solve traditional machine learning classification problems, tracking tasks and interactive segmentation in well known datasets with very interesting results. This new technique converges to the performance of the off-line version of Random Forest while the more information arrives in the data stream. For the tracking tasks, the authors executed experiments to detect faces using haar-like features.

Chapter 3

Face detection background

The second part of the research is related to the goal of detect faces. This is a well known theme in the computer vision area and many techniques have been developed to solve this problem.

3.1 Previous work

[28] shows the Eigenfaces algorithm, a well known technique based on PCA to detect and recognise faces. The algorithm consists basically in calculate the covariance matrix for a dataset of faces and calculate the eigenvalues and eigenvectors of that matrix. The eigenvectors are sorted in descending order according to their corresponding eigenvalues and the eigenvectors in the top of the list are called the eigenfaces of the solution that will represent all the faces of the original dataset.

Another interesting algorithm is the Fisherfaces model explained in [7]. It creates a method based on Fisher's Linear Discriminant that produce well separated classes with strong changes in illumination or in the face expression, characteristics that traditional eigenfaces model fails to classify.

Later [30] proposes a new model called TensorFaces that considers faces expressions, lighting and head poses to detect faces in a scene. It is based on representing the face image in a tensor that basically consists in a N-dimensional matrix, where each dimension corresponds to a one of the previous three features in the image.

Finally, based on the random forest researches in [25] and [27], the face detection technique proposed in [31] and [32] is reviewed. These are well known papers in the computer vision field and they exposed how by combining properly several sets of simple rectangular features and calculating their value over an integral image, the face

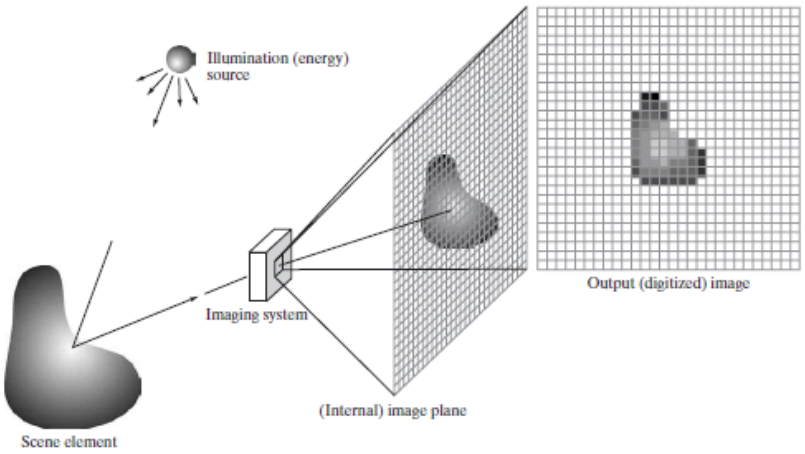


Figure 3.1: Digital Image Representation [1]

detection happened 15 times faster than previous studies of that time. [34] showed how by using the same concept of [31] they are able to detect facial features such as eyes, nose and mouth. Later [8] also based their work in [31] by implementing the face detection algorithm in a GPU.

[26] collected and extended the use of haar-like features by showing how this technique can be applied to detect different types of objects such as pedestrians in a street or cardiac structures in medical images.

3.2 Obtaining faces features

3.2.1 Integral Image

Based on [18], a grey-scale *digital image* (or monochrome image) is one way to represent a scene based on the combination of the illumination and the reflectance over the objects in the scene. This representation is done by a 2-D array (or digitalised image) in which each cell or *pixel*, has an integer value, or intensity l, between 0 (total absorption of light or black) and 255 (total reflection of light or white). The image 3.1 represents a digitalisation process over a scene. The digital image can be treated as a Cartesian plane which the origin is the top left corner, so each pixel Q has a coordinate over the columns of the array (or horizontal movement over the image) and a coordinate over the rows (or vertical movement over the image). This convention from now on will be represented as $Q(col, row)$ where col and row are the column and the row of Q respectively in the image. The equation (3.1) resumes previous definition.

$$l = Q(col, row) \in [0, 255] \tag{3.1}$$

The *integral image* (also known as *summed area table*) can be described as other way to represent an image and it is based in the digitalised image. Similarly to the digital image it is a 2-D array in which the value of each pixel, from now on Q_{ii}, corresponds to the sum of values of the pixels with lower values of *col* and *row* of the digital image. In other words each pixel in the integral image has the sum of values of the pixels above and to the left of it. Because of that, generally the pixel with lowest value is the one in the top left corner of the image and the pixel with highest value is in the bottom right corner of the integral image. The seed condition for this definition is that the value of the top left pixel of the integral image has the same value of the top left pixel the digital image. The integral image and the digital image have the same size, that means the number of columns and rows of the integral image correspond to the number of columns and rows of the digital image. The equations (3.2) and (3.3) show the mathematical definition of the integral image.

$$Q_{ii}(0,0) = Q(0,0) \tag{3.2}$$

$$Q_{ii}(col, row) = \sum_{row'=0}^{row-1} \sum_{col'=0}^{col-1} Q(col', row') \tag{3.3}$$

Another way to calculate the integral image is by calculating the value of each Q_{ii} based on the values of Q and the calculus of previous positions of Q_{ii}. With this trick only one scan is needed over the digital image to be able to calculate the integral image [4]. Mathematically this behaviour is shown in equation (3.4).

$$Q_{ii}(col, row) = Q(col, row) + Q_{ii}(col-1, row) + Q_{ii}(col, row-1) - Q_{ii}(col-1, row-1) \tag{3.4}$$

If the values of Q_{ii} are normalised to the interval $[0, 255]$, in which 0 represents the darkest pixel and 255 the lightest, the integral image of digital images with several values of l will look like the image in figure 3.2

The generation of the integral image is done in order to calculate in a very fast way the sum of the values of the pixels of a digital image in a region of the image.

[1] Image taken from [18]

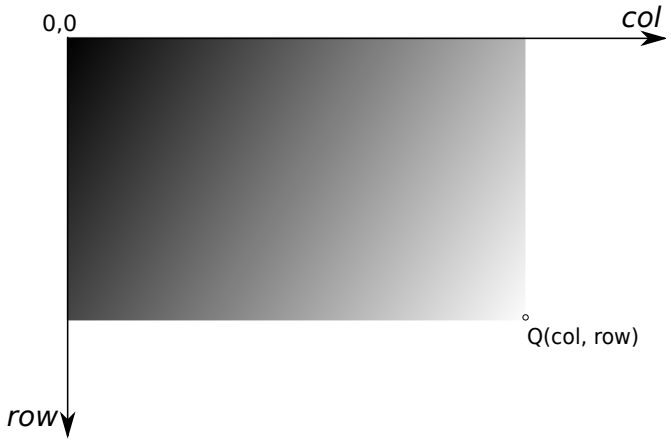


Figure 3.2: Integral Image representation normalising the value of the pixels to the interval $[0, 255]$

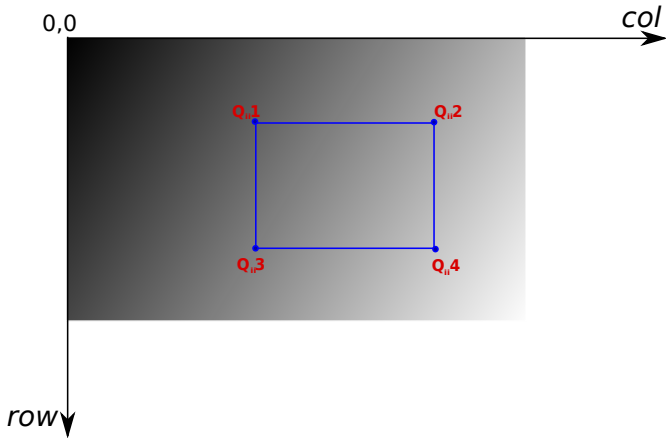


Figure 3.3: Region in an integral image

In a digital image, this process is accomplished by scanning the complete region to accumulate the values of each pixel. Therefore the sum of values of a pixels in a big region will take longer than doing the same task in a small region. The complexity of this process is On^2. With the integral image the execution time of this procedure is reduced dramatically with an arithmetic operation of 4 values. If we think that the value of a pixel $Q_{ii}(col, row)$ in an integral image is actually the sum of the values of the region determined by $Q(0,0)$ and $Q(col, row)$ in the digital image, is not difficult to conclude that is possible to calculate the value of the sum if the values of any region by taking into account the values Q_{ii} of the corners of that region. The image 3.3 shows the corners $Q_{ii}1$, $Q_{ii}2$, $Q_{ii}3$ and $Q_{ii}4$ of a region the integral image.

Basically these 4 pixels have the values of four regions between $Q(0,0)$ and each of them. So the final value of the marked region in figure 3.3 is calculated by adding and

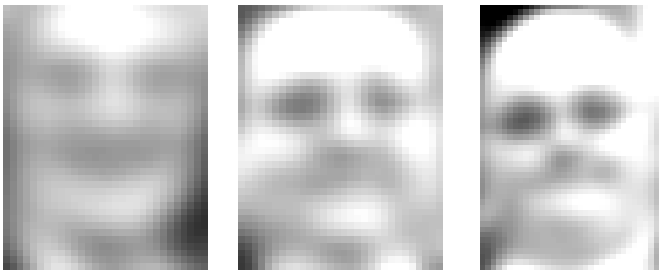


Figure 3.4: Dark and light regions in faces [2]

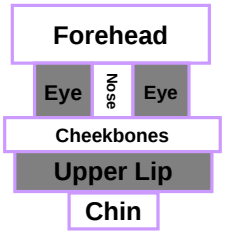


Figure 3.5: Possible combination of rectangles to detect features in a face [3]

subtracting the values of the regions determined by those pixels in the way is showed in the equation (3.5). This guarantees that this process has complexity $O(1)$.

$$S = Q_{ii}1 + Q_{ii}4 - Q_{ii}2 - Q_{ii}3 \tag{3.5}$$

3.2.2 Haar-like Features

The use of rectangular features is a concept introduced by [31] and basically it consists in emulate, by combining the rectangles, the dark and light regions in faces according to the position of the real characteristics of faces such as eyes, nose, mouth, cheeks, etc. The figure 3.4 shows how the light and dark regions are common in some grey-scale digital images of faces. For example, is easy to notice that the region of the eyes is darker than the region of the forehead in all images.

The figure 3.5 shows how by combining the positions and sizes of several rectangles is in some way possible to detect the faces features in grey-scale digital images.

This combination of rectangles is called a *Haar-Feature*, so by changing the sizes and positions of the rectangles is possible to build a huge set of Haar-Features in a very small region of the window.

[31] propose the use of features of two, three and four rectangles that are vertical or

[2] Image taken from [29]
[3] Image taken from [29]

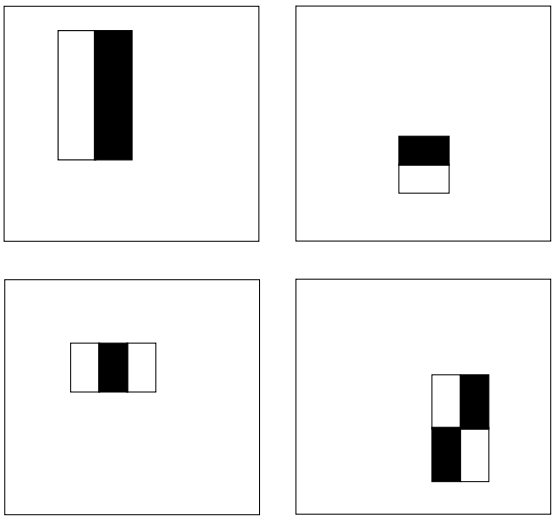


Figure 3.6: Simple Haar-Features enclosing in a rectangle window

horizontal adjacent and have the same size. The figure 3.6 shows those different kind of features could be enclosed in a detection window.

Now, each rectangle has a weight that indicates if the sum of values of the pixels inside the rectangle should be multiplied -1 or $+1$ according to the colour of the rectangle (generally -1 for black and $+1$ for white). In this point is when the integral image comes into action, because the sum of values of the pixels inside each rectangle is done executing the process explained in section 3.2.1. Finally the sum of the values of all the rectangles will result in the value of the feature. Mathematically this can be expressed as shown in equation (3.6) in which H is the value of the feature, R_{white} are all the white rectangles in a feature and R_{black} are all the black rectangles in the same feature.

$$H = \sum R_{white} - \sum R_{black} \tag{3.6}$$

So if a window containing a Haar-feature is over an image and the pixels in the image have a similar pattern of colours to the feature, the value of the feature will be high. On the other hand, if the pattern of colours of the image does not match with the feature, the value of the feature will be low.

Most recent researches such as [34] or [22] propose improvements over [31] by calculating optimal weights over the rectangles in a feature, changing the size of the rectangles or even more rotating the feature.

Chapter 4

Algorithm design

The process to develop the algorithm in this research takes into account the concepts explained previously in sections 2.1.2.1 and 2.1.2.2 and combines the characteristics of the forests developed in [25], [33] and [20]. So finally an On-line Random Forest (ORF) model is designed by following the steps below.

Let t be a on-line decision tree in the ensemble. Let ORF be the ensemble that represents the on-line random forest. N will be the number of decision trees in ORF so let's denote the ORF as $ORF = \{t_1, ..., t_N\}$. Following previous notation let A be an arriving sample defined as (x, y) in which x is a n-dimensional vector that represents the features of the sample and y depicts the label or class of that sample. Let $MaxMin$ be two global lists independent of the structure of the forest in which will be stored the maximum and the minimum values of the features according to the arriving samples.

4.1 Parameters

The parameters used to create an ORF are the following:

- N: Is the number of trees in the forest.
- D: Sets up the maximum depth of the trees.
- F: Is the number of features that are selected randomly in each node of the tree to apply the quality function.
- α: Is an integer that indicates the minimum number of samples that must pass through a node before trying to apply the split routine.

- β: Is a decimal value and shows the minimum value of quality measurement of the information that a node has collected before trying to split the node.

4.2 On-line Random Forest characteristics

- **Bagging**: The bagging process is done in the same way it was explained in section 2.1.2.2. For each t in ORF a value $k = Poisson(1)$ is calculated. k corresponds to the number of times A is sent to t.

- **A Node**: Each $node$ in t could be a leaf node $node_l$ or a split node $node_s$. $node_l$ could become $node_s$ but not vice versa.

  When a $node_l$ is created, a histogram H_{node} for that node will keep track of the quantity of samples per label that have reached that node. So, each label of the samples that reaches the node is registered in H_{node}. In the same way that an off-line random forest performs, F number of features are randomly selected to work with them as input to perform the split criterion function. Each feature is used to create a *random test* that could be defined as a tuple (f, θ) in which f is a function applied over the selected feature and θ is random value between the maximum and minimum values of that feature. Therefore each $node_l$ has a set of random tests $\{(f_1, \theta_1), ..., (f_F, \theta_F)\}$. In addition a random test holds 2 histograms H_l and H_r.

  Because of the impossibility of knowing the real maximum and minimum values of the features (in an infinite data-set new samples could modify the maximum or minimum values of the features), θ is calculated (using $MaxMin$) just when $alpha$ samples have reach the node. Because of that we need to maintain the samples in the node. Once each θ is calculated, each f is applied for each of the samples in the node and for new arriving samples. If the result of applying f is lower than its respective θ the sample is registered in H_l and if its greater or equal is registered and H_r (the samples are treated in H_l and H_r in the same they were in in H_{node}).

- **Split Criterion**: As was told before, the first condition to split a node is that the number of samples that meet a node should be greater than α. The second condition implies that the gain G respect to a random test must be greater than β.

There are two main functions to calculate G. The first one is considering the number of samples registered in a histogram: $|H|$. The second consists in selecting a quality function Q to measure the amount of information in a histogram. In this case, *Entropy* or *Gini Index* are valid Q. So with a histogram H storing the frequency of the labels in a set of samples, is easy to calculate whatever of this quality functions as $Entropy = Q(H) = \sum_y p(y)\log(p(y))$ or $Gini = Q(H) = \sum_y p(y)(1-p(y))$.

With previous definitions, the calculus of G (as define in equation (4.1)) respect to a random test involves the three histograms a node has.

$$G = Q(H_{node}) - \frac{|H_l|}{|H_{node}|}Q(H_l) - \frac{|H_r|}{|H_{node}|}Q(H_r) \tag{4.1}$$

So in each $node_l$, for each sample the algorithm calculates F different values for G. If the biggest G is greater than β several things happen:

- The correspondent test (f, θ) of biggest G is selected as official test of the node, so new arriving samples will only apply this test over the correspondent feature to decide the sample must go down to the left or to the right children of the node.
- The stored samples of $node_l$ are deleted.
- Two new child leaf nodes are created: $node_{l_{right}}$ and $node_{l_{left}}$
- H_r becomes H_{node} of $node_{l_{right}}$ and H_l becomes H_{node} of $node_{l_{left}}$
- $node_l$ becomes $node_s$

Figure 4.1 emulates the split criterion process. The blue shadow inside $node_l$ corresponds to the number of samples that have reach the node. Once the samples in the node reach α (the blue shadow fills the node) the gain function is applied and if the result is greater than β the node change to state the $node_s$ and two new $node_l$ are created as children of the current node.

- **End of growing**: Each node in t knows its the level. At the beginning the root node have its depth equal to D, when its split happens the new child nodes are created with depth $D-1$. This process continues until the depth of the node is 0. In this point just H_{node} is updated with the information of the new arriving samples.

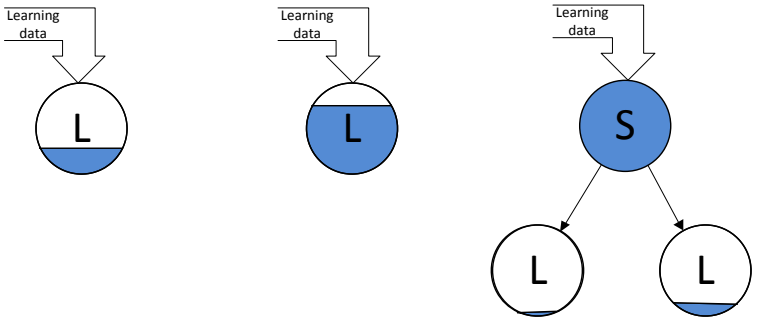


Figure 4.1: Split node process

- **Classification result**: When a new sample A arrives for evaluation purposes, it will fall into a single $node_l$ for each tree. To predict its label, each tree returns the most common label of H_{node} for that $node_l$. All the results are collected in a general histogram H_{ORF}. The most common label of H_{ORF} is the predicted label for A.

4.3 Pseudo algorithms

The algorithm 3 shows the details of the learning process explained in sections 4.1 and 4.2 of the on-line random forest designed as core of this research.

The algorithm 4 shows the process designed to test an example.

4.4 Merge with the face detection tools

The face detection process implies the selection of features able to describe clearly the characteristics of faces, and treat them as features of traditional machine learning data-sets. Moreover, the execution time of processing the features must be very low in order to be compatible with the process describe in chapter 2. For this reason, based on the research exposed in chapter 3 the algorithm proposed by [31] and [32] was selected as base process for the face detection phase of this work. Specifically, the concepts *Integral Image*, and the *Haar-like features* developed in [31] and explained in sections 3.2.1 and 3.2.2 are taken to apply the set of faces features over the images used in this research.

The figure 4.2 shows the procedure when the algorithm uses a window holding haar-like features to scan and Integral Image. First a small window starts the process

by looking for patterns in the image that match the holding haar-like features in the top-left corner of the image. As soon as it finishes, it will move a given number of steps to the right to repeat the same process. When the window reaches the end of that row, it will move to the next row to repeat the previous process. When that window ends the scanning process over all the integral image, a new bigger window starts the process to scan the complete image again. This algorithm finishes when the biggest possible window (one side of the window is as big as the smallest side of the image) scans the image.

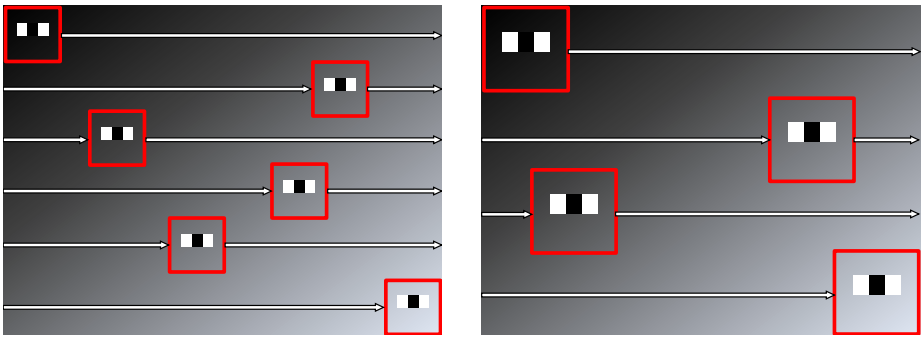


Figure 4.2: Process of scanning and Integral Image with subwindows holding haar-like features

Given previous scanning strategy, the modifications done to algorithm 4 for detecting faces are shown in algorithm 5.

Algorithm 3: Learning algorithm of the On-line Random Forest

```
input: N → number of trees
input: D → maximum depth of trees
input: α → minimum number of Samples
input: β → minimum Gain
input: F → Number of features to select for testing
input: A = (x, y) → an arriving sample

MaxMin = SaveMaxAndMinValuesOfFeatures(x)
for i from 1 to N do                                          // for each tree

    k = Poisson(1)                                            // OnlineBagging
    for j from 1 to k do                                  // send k times A to each tree

        findNodeInTree(x)                          // go down into the tree to find a node_l
        if A is first Sample in node then
            createRandomTests(F)
        end
        H_node(y)                                      // update histogram of this node
        saveSampleInStore(A)
        if α > number of samples in store then
            if haven't calculated thresholds then
                for each random test do
                    calculateRandomThreshold(MaxMin)
                    // flush samples into histograms of random tests
                    H_l(storeSamples)
                    H_r(storeSamples)
                end
            end
            else
                for each random test do
                    // update histograms of random test with this y
                    H_l(y)
                    H_r(y)
                end
            end
            // Calculate the gain for each random test and get the
               maximum
            maximumGain = calculateMaximumGain()
            // if maximumGain is grater than β the node must be split
            if maximumGain.value > β then
                // create child nodes with D-1 and with the selected histograms
                createLeftChildNode(D − 1, maximumGain.H_l)
                createRightChildNode(D − 1, maximumGain.H_r)
                deleteSamplesFromStore()          // delete temporary stored samples
                testOfNode = maximumGain          // the test of the node is set up
                change node_l to node_s                    // now this node is split
            end
        end
    end
end
```

Algorithm 4: Testing algorithm of the On-line Random Forest

input: $A = (x, ?) \rightarrow$ testing sample
return $\widehat{y} \rightarrow$ *predicted label*

for *i from 1 to N* **do** // for each tree

    $node_l$ = findNodeInTree(x) // go down into the tree to find a $node_l$
    $y[i]$ = getMostCommonLabel($node_l.H_{node}$) // find the most common label of the histogram of that node
end
$\widehat{y}$ = getMostCommonLabel(y) // find the most common label across all the trees
return $\widehat{y}$

Algorithm 5: Modifications on testing algorithm of the On-line Random Forest for detecting faces

input: $A = (x, ?) \rightarrow$ image testing sample
return *subwindowsFaces* $\rightarrow$ *all the windows that have faces*

iM = calculateIntegralImage(A)
for *each subwindow in iM* **do** // Scan *iM* with subwindows of different sizes

    for *i from 1 to N* **do** // for each tree

        $node_l$ = findNodeInTree(x) // go down into the tree to find a $node_l$
        $y[i]$ = getMostCommonLabel($node_l.H_{node}$) // find the most common label of the histogram of that node
    end
    $\widehat{y}$ = getMostCommonLabel(y) // find the most common label across all the trees
    if $\widehat{y}$ *is label for face* **then** // if this subwindow has a face

        subWindowsFaces.add(*subwindow*) // save this subwindow
    end
end
return *subwindowsFaces*

Chapter 5

Software design and implementation

This chapter shows explicit details of the process of design and implementation of the software based on the theoretical mark developed in chapters 2 and 3 and using the algorithm designed in chapter 4.

5.1 Design

5.1.1 Context Diagram

The first step in the design process consists in modelling the context diagram (or level 0 diagram) which final goal is to identify the boundaries of the system, and the way it interacts with external components. The figure 5.1 depicts the concept in which the *On-line Random Forest* (our system) is connected to two sources of data (external components): the *Learning Data* and the *Testing Data*. Let's remember that the bank of learning data could have infinite information. This is the model we will use to built the random forest that will work with the traditional machine learning datasets. Then, the figure 5.2 exposes the context of the forest that will work in the face detection problem. This diagram extends the case of the context diagram of figure 5.1 by adding the images taken from video as new source of testing data, and also adds the *Bank of faces features* as new component. This last bank of information could be explained as follows: in one hand for traditional machine learning datasets each column of each sample corresponds to a feature, on the other hand the face detection model uses samples of images with faces and non-faces but from these samples is not easy do determine which are the features of the faces, so is necessary to have a new bank of information containing the faces' features to be used in the learning and testing phases. Basically this repository

contains the haar-like features explained in section 3.2.2.

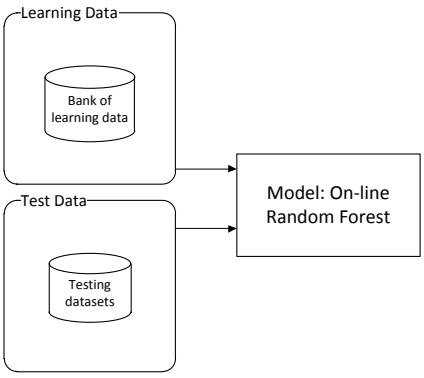


Figure 5.1: Context Diagram

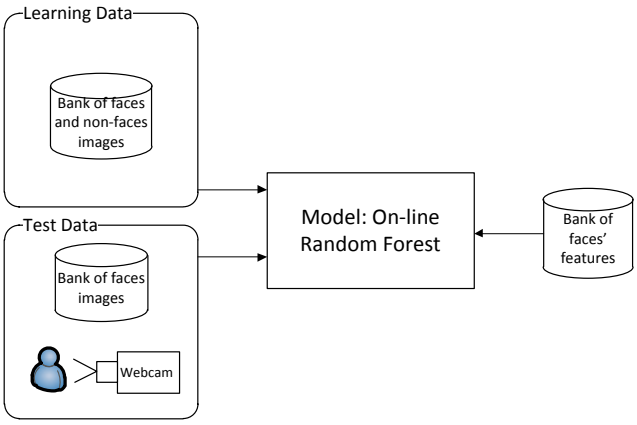


Figure 5.2: Context Diagram for face detection

5.1.2 Components Diagram

The figure 5.3 shows a level 1 diagram having more details about the components that interact in the software. The first part of the diagram shows the *DataSources layer* that basically holds components that will extract the information from all the data sources. In the case of the *Learning datasets*, components able to handle images in PGM format will read the information from the files holding faces and non faces images. For the *Testing datasets* all the components related to the administration of the webcam driver to control the camera features must be built into the *Video Source* component, the *Snapshoter* will be in charge of taking continuously frames from the video source to send them as images to the next component, and the components that will read the

information of the files holding the faces that will be used for test and evaluate the program must be built in the *Bank of faces images* component. This layer finishes with the component that will read the xml file that holds the information related to the features of the faces. All previous components have communication interfaces that will be used by the *Adapters layer*.

The *Adapters layer* holds the components that will transform the information that was read in previous layer into the objects the on-line random forest needs. The *Adapter Images* component is an adapter able to transform each digital image received from the learning/testing datasets into an integral image which is the required image format. It also sends the information to the forest through the *iLearning* or *iTesting* interfaces according to the data source (learning information through iLearning interface and testing data through iTesting interface). The *Adapter Features* component obtains the information from the bank of features and sends all these features in the right format to the forest through the *iFeatures* interface.

Finally the base of the design has the *OnlineRandomForest*. Its three interfaces allow the communication with the rest of the system. It holds components called $Tree_i$ that correspond to each on-line decision tree of the ensemble. Those trees have the same three interfaces due to each of them have to process the information in the same way the forest does. The *MaxMin Handler* is the component that stores the maximum and minimum values of each feature and it has communication with all the trees due to the this information is necessary to create each random test inside each node for all the trees.

5.1.3 Class Diagram

This is a level 2 diagram that shows the main classes that articulate the component *OnlineRandomForest* of figure 5.3. In figure 5.4 we can see the *Model* interface has the methods *addSample* and *evalSample* that correspond to the communication interfaces related to the learning and testing process respectively for the *OnlineRandomForest*, *OnlineTree* and *OnlineNode* classes (iLearning and iTesting interfaces in figure 5.3). The basic structure of the forest explained in chapters 2 and 4 can be visualized as well: the *OnlineForest* has a collection of *OnlineDecisionTrees*, each of those trees has a root *OnlineNode* and each node has two references to new nodes that are their left and right children. Each *OnlineNode* has one *SetRandomTests* that corresponds to the features that are selected randomly once the node is created. The *SetRandomTests* has a list of *RandomTest* classes that are in charge of apply each test to its correspondent

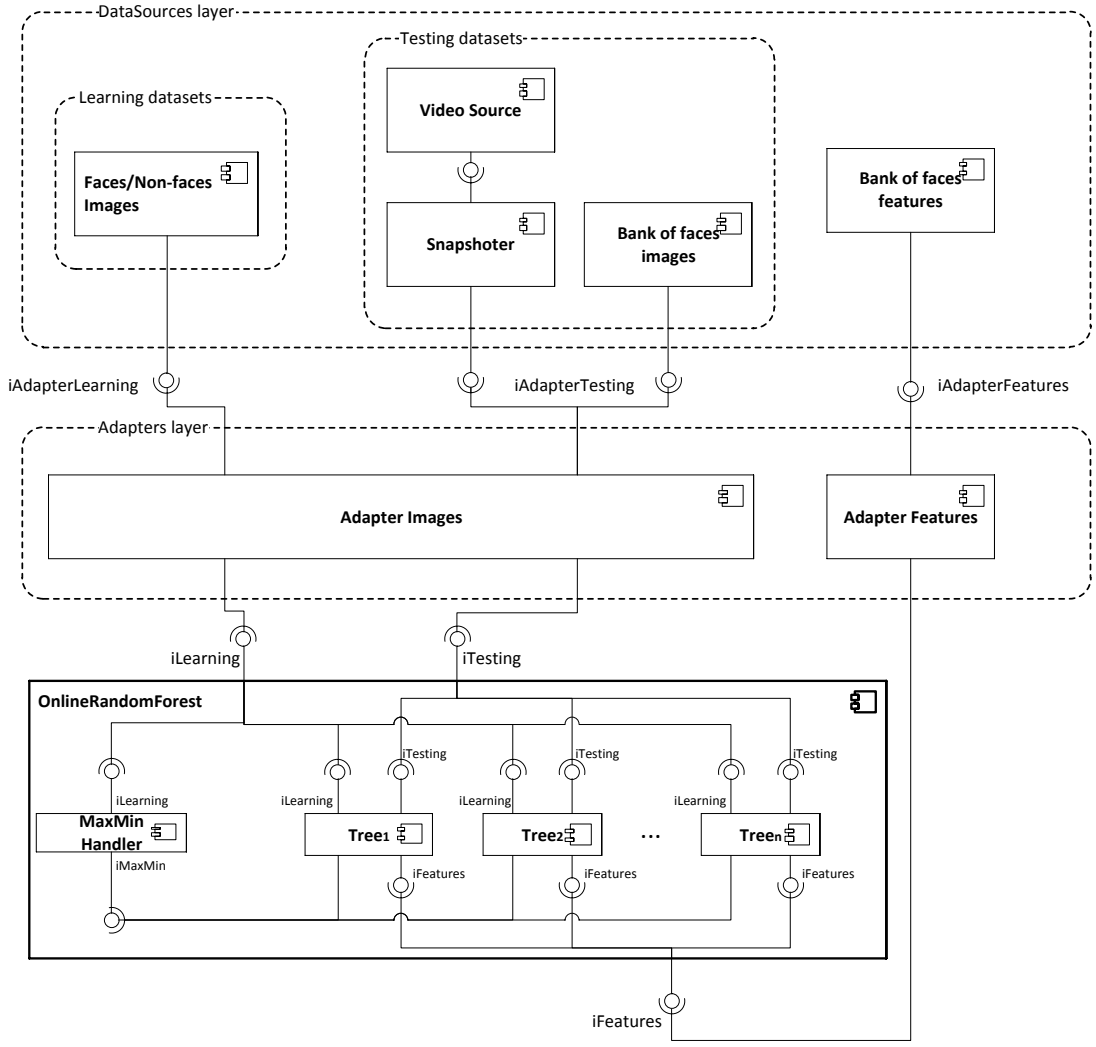


Figure 5.3: Components diagram

feature. The *FeaturesMaxiMin* class is the one that have the maximum and minimum values of each feature, so it must have a reference from the classes that need those values.

5.1.4 Quality attributes

The utility tree in figure 5.5 shows the quality attributes the system must have:

- **Performance - Execution time**: The nature of the model related to the on-line learning process makes this quality attribute the most relevant to the project. The model should be able to work with streams of data that continuously will arrive waiting to be processed. So the model should be fast enough to deal with this

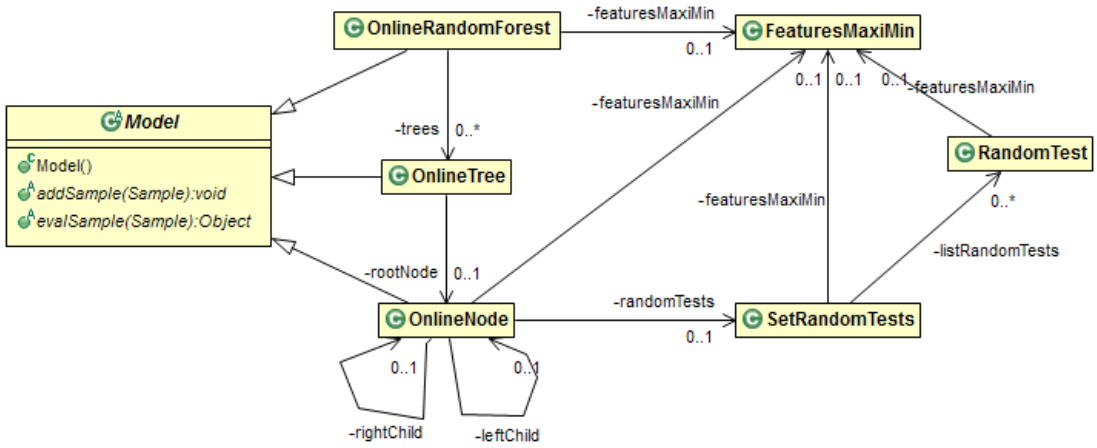


Figure 5.4: Class diagram

kind of characteristic. For testing videos, 15 frames per second will be generated from the video source with resolutions up to 320×240 pixels.

- **Portability**: Based on one of the objectives of the project related to the software could be used as a teaching tool, the final product will be installed in many types of machines with different operative systems. So is necessary to build a program easy to install and execute in several platforms.

- **Traceability**: Is important to leave a trace of failures in case some unexpected data or situation occurs during the execution of the software in order to trace the error an recover the execution of the program.

- **Security**: No relevant to this project in terms of confidentiality of data, data integrity and data availability.

- **Fault tolerance**: This is not a relevant issue for the final product in this project. It is not necessary that the system could recover automatically in case of a fail, and a manual procedure should be done in order to detect the error that caused the failure and to start again the process.

5.1.5 Design decisions

Based on previous quality attributes, the programming language selected to build the software is Java. Due to its Oriented Object Programming paradigm, many architectural patterns, design patterns and best practices rules can be applied to reach the performance and efficiency attributes. Moreover actual hardware architectures based

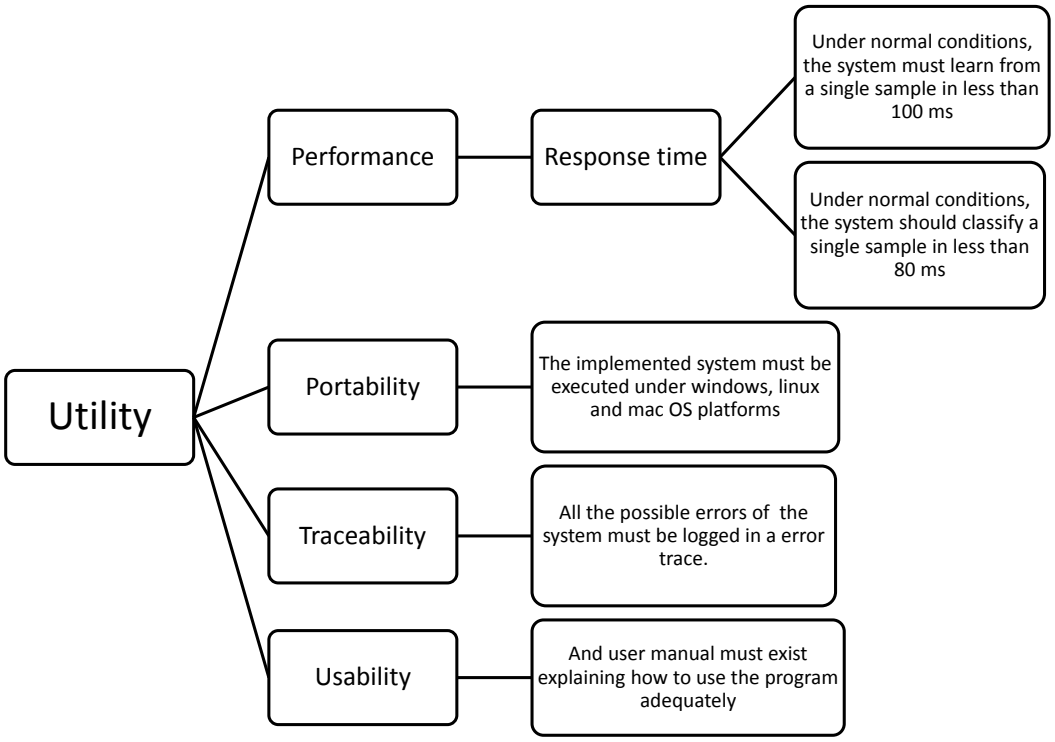


Figure 5.5: Utility tree

on multi-core CPUs allow Java to take advantage of its multi-threading and concurrency tools. The portability attribute is in some way "natural" for Java codes due to that responsibility is already solved by the Java Virtual Machine or in general by the Java Runtime Environment installed in each platform. So the same binary class executable files will run in a machine with Java installed. The traceability is solved by doing a right exception handling in order to leave a trace of errors with enough information to find what is going wrong in any given moment. Because of the popularity of the programming language there are many available APIs related to many of the tasks that should be accomplished according to the design such as managing images, dealing with webcam, building user interfaces, solving communications issues between components, managing data in memory, dealing with files, etc.

Talking about the datasets, there are several options of machine learning datasets available on the internet that could be use for research purposes. The datasets collected by [13] will be used as source learning and testing sets. In the face detection environment the research led us to choose [9] as learning set because of the huge database of faces and non-faces images that are already cropped to 19×19 pixels in grey-scale

PGM format. The database of faces in [6] has 165 grey-scale images in GIF format mixing 11 face expressions for 15 multi ethnic subjects, so we consider this database as a good option to for testing set. Moreover the size of each image is 320×243 pixels which is a very similar size of an image capture from a traditional webcam.

The Haar-like features technique proposed in [31] was the chosen one due to its performance in execution time. Moreover, this technique can be modified to detect several types of objects by changing the features. So in theory, the on-line decision tree model would be able to learn to detect whatever object in the image according to set of features it uses.

The features of faces could be built by ourselves following the patterns suggested in [31]. However we decided to use and already proved set of features of faces using the file *haarcascade_frontalface_default.xml* of openCV [2]. The file contains 2043 haar-features inside a window which size is 24×24.

The Performance Quality Attribute obligates us to think in a multithread solution in order to process the highest number of samples possible once the stream of data is arriving to the model. The design decision for this case consists in applying a dynamic scheduling multithread strategy to process the maximum number of samples possible. In this case, we will take advantage of the nature of the algorithm in the sense that each tree can process the sample independently of the other trees of the assemble. So, each tree will work with one thread of the pool. A barrier at the end of the process will guarantee that the results of each sample are collected to calculate the result of the complete forest for that sample. This will work both learning and testing processes.

5.2 Implementation

5.2.1 External Resources and Libraries

Some libraries and external resources were used in the implementation of this project:

- **Machine learning datasets**: The datasets used as learning and testing data are found in [13] and [14]. Their format is shown in figure 5.6. Each record of the file corresponds to a learning/testing sample and the features are separated by *;*. The final column of each record is the label or class of each sample.

- **Webcam driver handler**: This is a java library able to detect and already installed webcam in the platform and perform video operations such as open and

winequality-red.txt

```
7.4;0.7;0;1.9;0.076;11;34;0.9978;3.51;0.56;9.4;5
7.8;0.88;0;2.6;0.098;25;67;0.9968;3.2;0.68;9.8;5
7.8;0.76;0.04;2.3;0.092;15;54;0.997;3.26;0.65;9.8;5
11.2;0.28;0.56;1.9;0.075;17;60;0.998;3.16;0.58;9.8;6
7.4;0.7;0;1.9;0.076;11;34;0.9978;3.51;0.56;9.4;5
7.4;0.66;0;1.8;0.075;13;40;0.9978;3.51;0.56;9.4;5
7.9;0.6;0.06;1.6;0.069;15;59;0.9964;3.3;0.46;9.4;5
7.3;0.65;0;1.2;0.065;15;21;0.9946;3.39;0.47;10;7
7.8;0.58;0.02;2;0.073;9;18;0.9968;3.36;0.57;9.5;7
7.5;0.5;0.36;6.1;0.071;17;102;0.9978;3.35;0.8;10.5;5
6.7;0.58;0.08;1.8;0.097;15;65;0.9959;3.28;0.54;9.2;5
7.5;0.5;0.36;6.1;0.071;17;102;0.9978;3.35;0.8;10.5;5
5.6;0.615;0;1.6;0.089;16;59;0.9943;3.58;0.52;9.9;5
7.8;0.61;0.29;1.6;0.114;9;29;0.9974;3.26;1.56;9.1;5
8.9;0.62;0.18;3.8;0.176;52;145;0.9986;3.16;0.88;9.2;5
8.9;0.62;0.19;3.9;0.17;51;148;0.9986;3.17;0.93;9.2;5
8.5;0.28;0.56;1.8;0.092;35;103;0.9969;3.3;0.75;10.5;7
```

Figure 5.6: Format of the datasets used for the Machine Learning tests

close the webcam driver, take a snapshot picture, take video, select the resolution, etc. All the information about this projects is found in [17]. The figure 5.7 show a screenshot of an image taken with the webcam

Figure 5.7: Test of webcam using [17]

- **OpenCV faces' features**: The file *haarcascade_frontalface_default.xml* contains the faces' features obtained by using the adaboost algorithm for face detection developed by [31] and developed by openCV. The header of the xml file is shown below:

```
<haarcascade_frontalface_default
type_id="opencv-haar-classifier">
<size>24 24</size>
<stages>
<_>
 <!-- stage 0 -->
 <trees>
  <_>
```

```
<!-- tree 0 -->
<_>
 <!-- root node -->
  <feature>
   <rects>
    <_>6 4 12 9 -1.</_>
    <_>6 7 12 3 3.</_></rects>
   <tilted>0</tilted></feature>
   <threshold>-0.0315119996666908</threshold>
   <left_val>2.0875380039215088</left_val>
   <right_val>-2.2172100543975830</right_val></_></_>
<_>
...
```

For this case we care about the information of the tags *size*, *feature* and *rects* (our algorithm does not need information about stages, threshold, tilted, left_val and right_val.) So, in this case all the features are in a window of size 24×24 (information in the tag size) and there is a feature consisting of 2 rectangles. The first rectangle is in the position $(6,4)$ of the window, has a width of 12 units and a height 9 units, and the sum of the values of the pixels inside that rectangle must be multiplied by -1. The second rectangle is in the position $(6,7)$ of the window, its dimensions are 12 units of width and 3 of height, and the sum ov the values of the pixels in the rectangle should be multiplied by 3. This feature is shown in figure 5.8. The xml file has 2043 features in total. The information of this project can be found in [2].

- **MIT center for biological and computational learning**: This is the database of faces used as learning dataset. All the information of this database is found in [9]. Examples of the faces and non-faces images used in the learning set are shown in figures 5.9 and 5.10.

- **Yale faces**: The information of this project can be found in [6]. This is the database of faces that was used as testing dataset. Samples of the 15 subjects of the database with neutral light conditions and without any face expression are shown in figure 5.11.

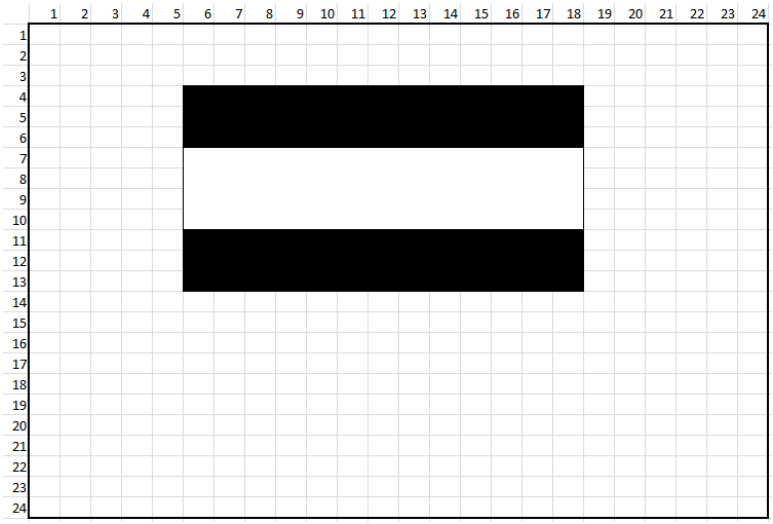


Figure 5.8: First haar-feature in *haarcascade_frontalface_default.xml*

Figure 5.9: Learning faces

Figure 5.10: Learning non-faces

Figure 5.11: Test faces

5.2.2 Packages

The following are the packages built during the implementation process:

- **adapters**: This package has the classes that transform the input data (learning and testing datasets) into the object the forest needs to start the process.
- **exceptionHandler**: It has the handler classes in charge of capture and register the program exceptions.
- **gui**: This package holds the classes that model all the components of frames and panels of the user interface of the program. The design of the user interface is shown in figure 5.12

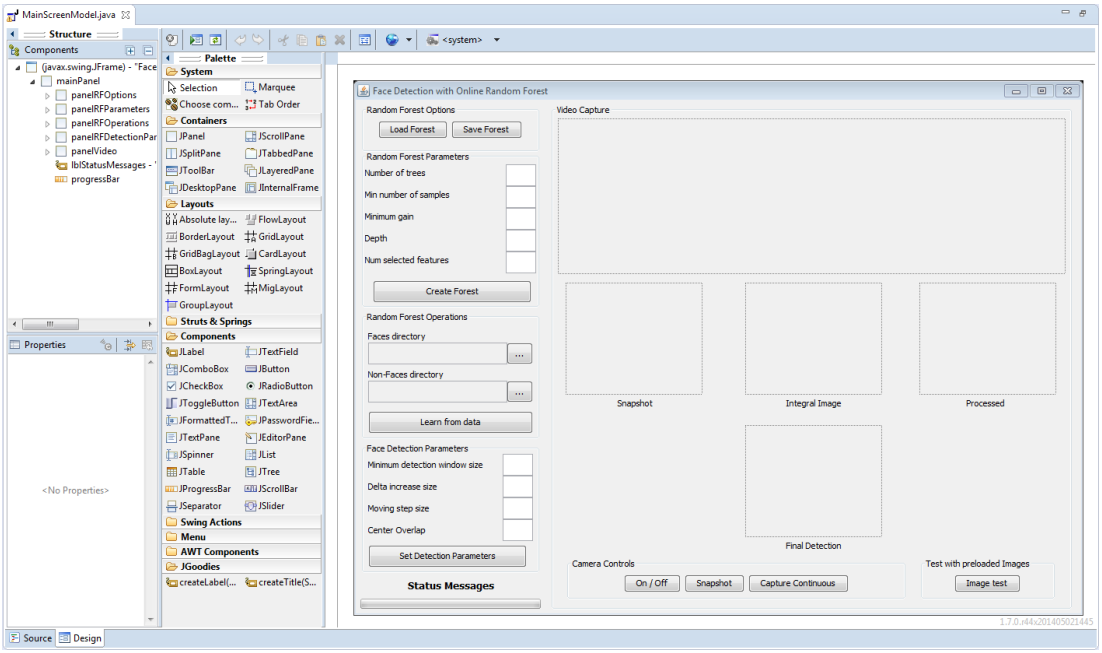


Figure 5.12: Design of the User Interface

- **helper**: Contains helpers classes able to perform general purpose functions such as file handling, calculus of random numbers with different techniques, lists managing, etc.
- **image**: The classes in this package are the ones that are able to handle all the operations with images with different formats such as PGM and GIF. The class that calculates the Integral Image of the grey-scale digital image is also in this package.

- **image.haarlikefeatures**: It has the base classes with the necessary structures to work with haarlike features such as points, areas and combination of areas. It also has the class that reads the information from the file *haarcascade_frontalface_default.xml* and stores the information in the haar feature class that will be used by the model.

- **onlineRandomForest**: This package contain the classes that have the core algorithm of the on-line random forest. All the classes related to the structure of the forest such as node, trees and forest belong to the package. The classes related to calculate the random test and calculate the threshold for each of these test are also here. Moreover it has the class that calculates and selects the best value of gain.

- **valueObjects**: Finally in this package there are the classes that have the structure of the learning and testing datasets.

There are test packages that hold the classes that were designed as unit test for each module of the software.

The figure 5.13 shows this distribution of packages in the IDE eclipse.

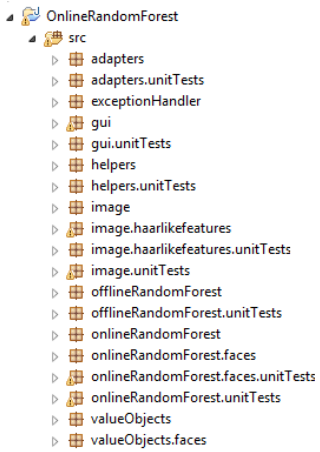


Figure 5.13: Packages in the eclipse IDE

5.3 Repository

All the code of the project was uploaded into a development repository using Apache Subversion (SVN) for software versioning and revision control. The repository was created in the site [1]. During the development and testing phases 53 commit operations were executed.

Chapter 6

Testing and Evaluation

The tests of this research were divided into two blocks: The first block of tests is related to the scenarios executed over the base algorithm of the on-line random forest using traditional machine learning datasets. The second one uses the face detection program to show the accuracy of the model embedded in the designed program. All the parameters, configuration, execution, results and evaluation of the tests are explained in detail in each scenario.

6.1 Tests with Machine Learning Datasets

Several tests were executed over the base algorithm of the on-line random forest. Each scenario was designed and executed with a specific goal in order to test an particular characteristic of the model. The datasets were obtained from [13], [14] and [3]. Traditionally the machine learning datasets are divided into training data and testing data, for these tests we will call the training data as *learning data* and the testing data preserves its name. The learning data and the testing data do not have registers in common. The table 6.1 shows the most relevant characteristics of datasets used during the tests.

Dataset	Learning data size	Testing data size	Features	Classes
dna	1400	1186	180	3
usps	7291	2007	256	10
redwine	1000	599	11	11
australian	400	190	14	2

Table 6.1: Datasets used in the machine learning tests

6.1.1 Accuracy of the model vs Stream of data

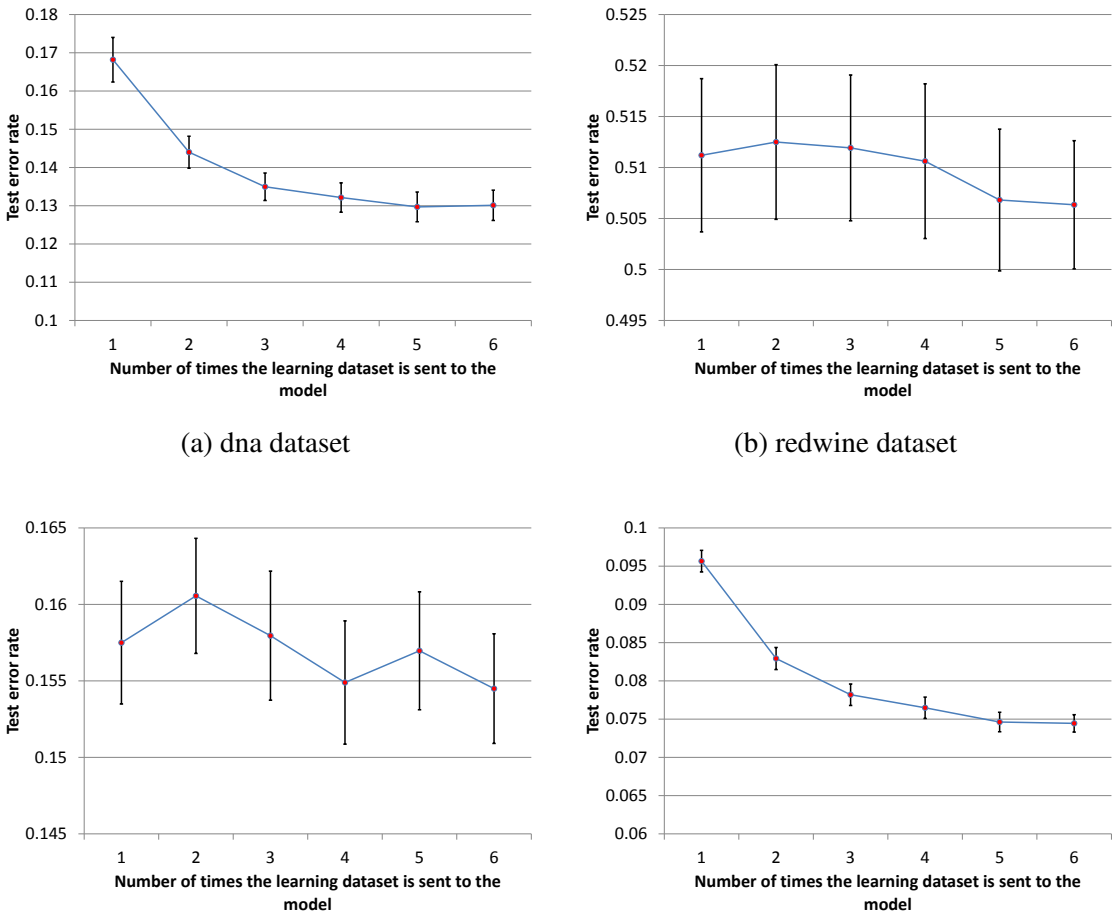
This test was done in order to check the accuracy (or test error rate) of the model while more learning data is arriving, in other words to test the on-line learning process. The data streaming was emulated by sending 6 times the complete learning dataset to the same forest. The same testing set was evaluated each time a learning dataset was processed by the on-line random forest. Each experiment was executed 50 times, and the results were averaged to get more reliable numbers. The configuration of the parameters for creating each forest for all the datasets is shown in table 6.2. The figure 6.1 shows the results of this experiment.

	Forest Configuration				
Datset	**N**	**F**	**D**	**alpha**	**beta**
dna	37	13	10	20	0.05
winred	31	3	10	20	0.05
australian	20	3	10	20	0.05
usps	85	16	10	20	0.05

Table 6.2: Forest's parameters for test Accuracy vs Stream

This is one of the most important tests done to the model due to it shows the behaviour when new streaming data arrives. Each graph shows different scales to do a better analysis for each case (for all the cases 1 corresponds to maximum error and 0 to no error). Both figures, 6.1a and 6.1d, show consistently a decreasing trend of the error rate each time new learning data arrives. The rate of error does not fall as quick in figure 6.1b, first a slight increase of the error rate is shown when the learning set is sent by second time, but after this point a slow monotonic fall of the error rate is shown each time a learning set arrives. The behaviour of model's error rate in figure 6.1c is not as uniform as happened with the rest of datasets. Here the value of error rate after sending the second and the fifth learning set is higher than its previous value (first and fourth respectively). However the error rate after sending the sixth dataset is lower than the rest. The very small values of the confidence intervals plotted in the figures guarantees the reliability of the test.

Discussion: The results show how the models were learning while more data arrived. Here the stream of data was emulated by sending multiple times the same dataset. Because each learning dataset had different sizes, the graphs of the biggest datasets showed perfectly the expected behaviour (dna and usps dataset) while the dataset with few samples on it showed a fluctuation in the value of the test error rate.



(a) dna dataset

(b) redwine dataset

(c) australian dataset

(d) usps dataset

Figure 6.1: Data stream vs testing error with 95% of confidence intervals for all datasets

This could be explained as: the more information is flowing in the trees, more splits occur and more leaf nodes with a dominating class appear. Moreover, the set up configuration for each forest in this experiment could affect the behaviour of how fast/slow the error rate decreased. N was set according to the size of the learning dataset and F according to the number of features but D, α and β were constant for all the datasets. An independent tuning process could be done for each dataset in order to find an a suitable configuration for each experiment. However the general trend of the error rate was to decrease in all the datasets while more learning data was processed. With this test is being demonstrated properly the concept of on-line learning explained in section 2.1.2.

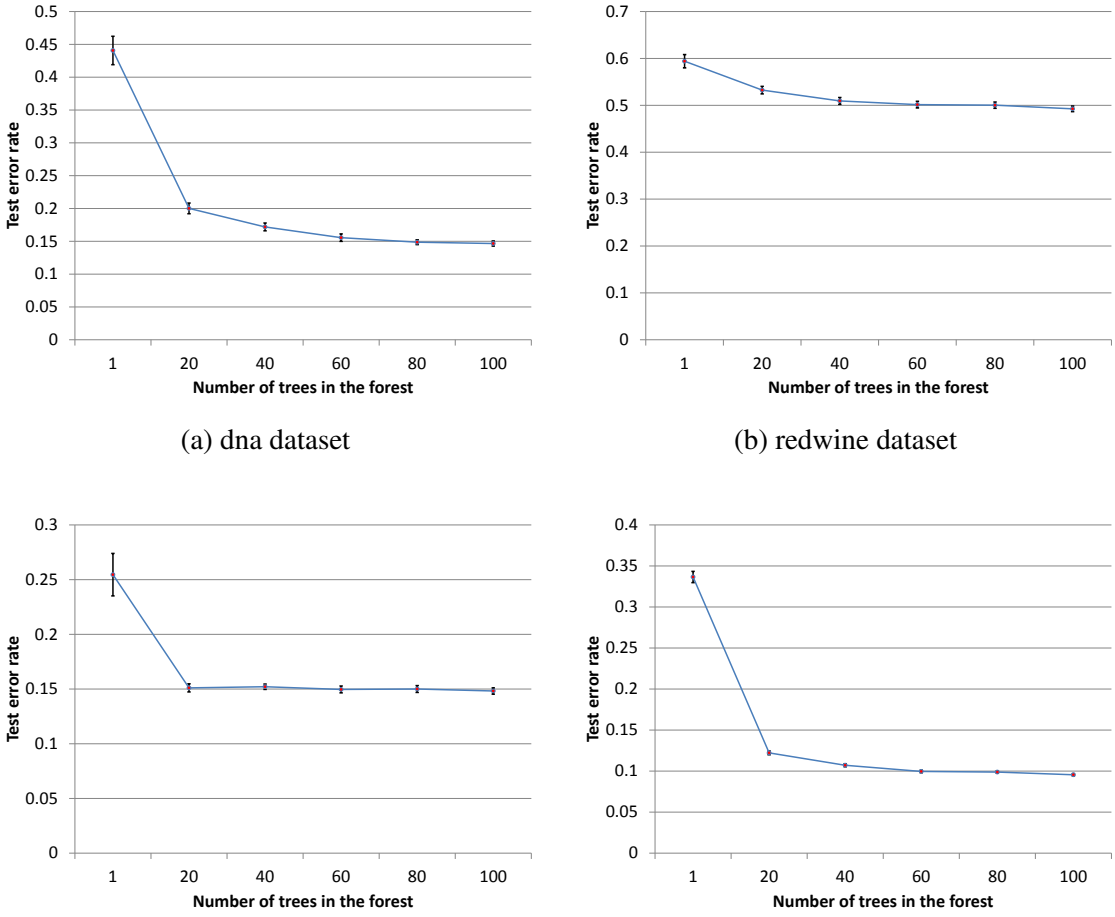
6.1.2 Accuracy of the model vs Number of trees

This test was done to validate the behaviour of the testing error while the number of the on-line decision trees into the on-line random forest are increased. Basically the test consisted in sending the same learning set to 5 on-line random forest with sizes 1, 20, 40, 60, 80 and 100 (let's remember that the size of the forest corresponds to the parameter N), and then the same testing set was processed for all the forests to measure the each test error. All the datasets were used for this test. The table of parameters 6.3 has the configuration of forest used for each dataset. The experiment was executed 50 times to get more reliable numbers. The results are shown in figure 6.2.

	Forest Configuration				
Datset	**N**	**F**	**D**	**alpha**	**beta**
dna	{1, 20, 40, 60, 80, 100}	13	10	20	0.05
winred	{1, 20, 40, 60, 80, 100}	3	10	20	0.05
australian	{1, 20, 40, 60, 80, 100}	3	10	20	0.05
usps	{1, 20, 40, 60, 80, 100}	16	10	20	0.05

Table 6.3: Forest's parameters for test Accuracy vs Number of trees

The results confirm clearly the concept of how an ensemble of models perform better than a single model. For all the datasets the most remarkable change in the performance is shown between the forests of sizes 1 and 20 (19 trees of difference). In the case of the dna dataset this improve of the performance passes from 44% to 20%, for the winered dataset from 59% to 53%, the case of the australian dataset shows an improvement from close to the 25% to the 15% and similarly occurs for the usps dataset showing how the error rate decreases from 30% to 13%. Then, for figures 6.2a, 6.2b and 6.2d a slight performance occurs between 20 and 60 trees. The improvement in this range is much more lower than in previous range even when the forest grew by 40 trees. Finally for these three graphs after 60 trees are reached there is almost none improvement in the error rate. For figure 6.2c this plateau is reached from 20 trees. This suggest that no more improvement in the error rate is reached after a determined number of trees. In this case a wise decision consists in finding the minimum number of trees that give us the best performance of the model. If we select a big number of trees to ensure that we are finding the minimum error rate, we will be increasing unnecessarily the execution time of the algorithm. Another conclusion of the test lies on the fact that the reliability of the test increases while the number of trees increases. In the graph this is shown as a decrease of the size of the confidence bars when the size of the forest grows. The most relevant case happened with the dna dataset in which the confident interval for a forest of 1 tree is in the range ± 0.02 and with 100 trees is in



(a) dna dataset

(b) redwine dataset

(c) australian dataset

(d) usps dataset

Figure 6.2: Number of trees vs testing error with 95% of confidence intervals for all datasets

± 0.004

Discussion: In the first range of data (N between 1 and 20) is demonstrated that an ensemble model performs better than a single model. In all the datasets the fall in the test error rate had the greatest negative slope in this range than in any other. Morever is interesting to notice that in all figures there is a point in which increasing the number of trees does not make any significant difference in the testing error rate. So we conclude from these tests the N has a great impact in the accuracy of the model so is necessary to choose a right value to have good performance without affecting negatively the execution time of the model (greater values for N, greater is the execution time and in consequence slower is the classification task).

6.1.3 Measuring the performance while changing the parameters of an on-line random forest

Due to the on-line random forest algorithm we used has many parameters, we consider to execute a Systematic Test process to measure how the performance of the forest is affected while its parameters change. This test was done by following the steps of algorithm 6:

Algorithm 6: Algorithm to execute the test for measuring the performance while the parameters are changed

input: The dataset
input: first configuration of parameters
input: variation of each parameter
input: The order in which the parameters are going to be changed
input: $nExperiments \rightarrow$ the number of times a experiment is going to be executed

for *each parameter* **do**
  for *each variation of this parameter* **do**
    for *i = 1 to nExperiments* **do**
      create the forest using the parameters
      send learning set to the forest
      test the forest with the testing set
      save the True Positive Rate and False Positive Rate of this experiment
    end
    take the average of True Positives and False Positives of all experiments
    plot the result in a ROC Space (with confidence intervals)
  end
  Analyse the result and keep the best variation of the actual parameter according to the ROC curves
end
return a tuned On-line Random Forest

The dataset chosen for this test was *australian* due to it reduces problem to a YES/NO classification problem (it only has two labels) but it can be extended to whatever dataset. Then the table 6.4 was built according to the inputs of the algorithm 6. It shows the configuration of the order in which the parameters are going to change and all the values for each parameter. As seen in the table, we change first the value of N, then D, after that F, then α and finally β. The first column of the *Values Variations* section of the table shows the first configuration of forest, that for us is a pretty *bad* configuration (a forest with 1 tree, with maximum depth 2, choosing randomly 1 feature among all, holding 1 sample per node and with gain value of 0.5). We decided to run each experiment 20 times to have reliable numbers and plot the results in ROC

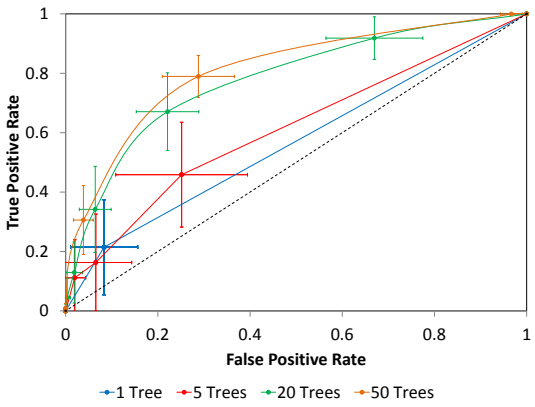
curves [16] with confidence intervals for true positive rates and false positive rates.

Feature	Order	Values Variations			
N	1	1	5	20	50
D	2	2	10	20	50
F	3	1	2	7	14
alpha	4	1	10	20	50
beta	5	0.5	0.1	0.05	

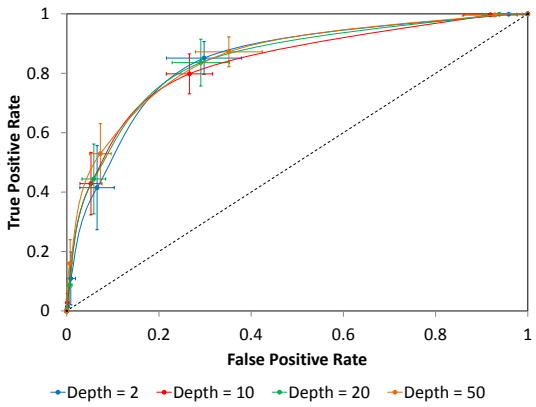
Table 6.4: Input parameters for measure the on-line random forest performance

The sum of the amount of variations per parameter times the number of parameters times 20 executions, results in total of 380 experiments. The figure 6.3 shows the ROC curves with the result of the execution of all the tests.

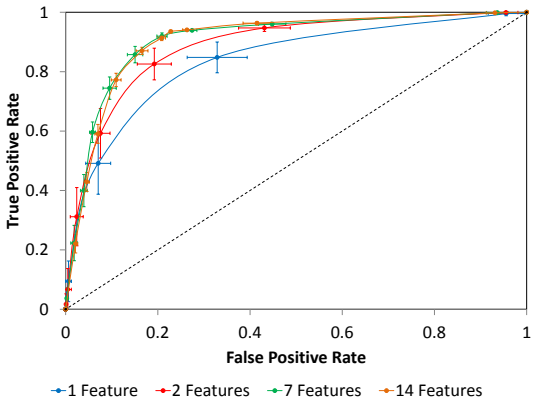
Figure 6.3a shows how the performance of the model gets better while the number of trees increases (we confirmed the results we got in section 6.1.2). Here the configuration of the forest with 1 tree performs slightly better than the random guessing and the best performance is got with a configuration of 50 trees, so for the next test the parameter N is fixed to **50**. For next test in figure 6.3b, is not very clear the trend about how the performance is affected while the value of depth varies. Is interesting to check that our worst value of depth $D = 2$ has higher values of performance than other values in some parts of the graph. There is no a single line in the graph that is always over the rest of the lines. The consistency of $D = 20$ was this time the criteria to fixed it for the next test. In figure 6.3c there is an evident difference between the amount of features that are randomly selected when a new node is created. The worst performance of this case occurs when $F = 1$ and a considerably better performance happens when this value is increased to $F = 2$. Very similar higher values of performance happen with $F = 7$ and $F = 14$, so we chose $F = 7$ due to the model will take less execution time for evaluating **7** features than 14 in each node. The figure 6.3d shows the tuning process of the parameter α. In this case our thought worst value of alpha performed much better than the rest of the values so we chose **1** as best value of alpha. Finally, the figure 6.3e show the changes in the performance while β varies. In this point of the test, the three chosen values expose extremely low differences in the ROC graph so it makes more difficult to make a decision. The decision we took in this point was to pick $\beta = 0.05$ due to this value never was the worst between all the possible values. The figure 6.3f shows the difference in performance of the first model (with $N = 1$, $D = 2$, $F = 1$, $\alpha = 1$ and $\beta = 0.5$) against the performance of the final model ($N = 50$, $D = 20$, $F = 7$, $\alpha = 1$ and $\beta = 0.05$).



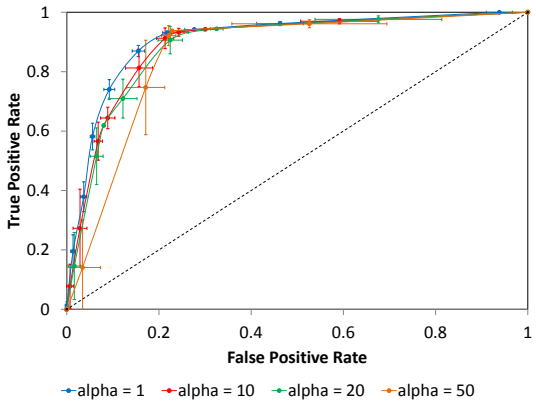
(a) Changing amount of trees



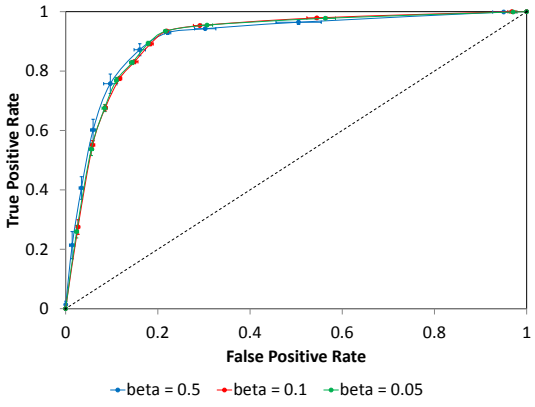
(b) Changing depth of the trees



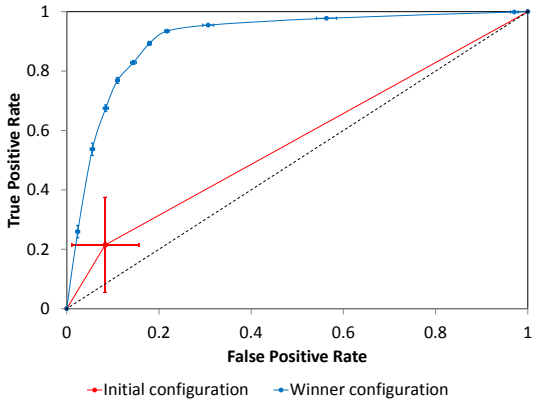
(c) Changing number of features



(d) Changing value of α



(e) Changing value of β



(f) Initial and final configurations of the forest

Figure 6.3: Performance ROC curves for australian dataset with 95% confidence intervals changing systematically the parameters of the on-line random forest

Discussion: This test strategy does not guarantee that the final result is the best possible configuration of an on-line random forest. This strategy finds a set of parameters that falls into a local minima among the 5-dimensional configuration space. The order in which the parameters are tested, the range of the values, and the step size between the range of the values are variables that affect the final result. This dissertation does not solve this issue, so it is proposed as future work. With many other tests we did, we proved that once a good configuration of parameters is reached, trying to find another good configuration starting from this point (with this we mean trying to go away from the local minimum we found) requires changing the values of the parameters dramatically so it turns on a very difficult task. The conclusion here lies on that each parameter of the forest affects in a very different way the performance of the model, so a tuning process is recommended before trying deciding the value of the parameters.

6.2 Tests for Face Detection

The on-line random forest created for face detection is built by using the MIT center for biological computational learning faces database [9]. This database contains grey-scale images in PGM format with a size of 19×19 pixels which has datasets of faces and non faces images. So in all the experiments this database was used as learning dataset for the model.

The testing data is taken from other two sources of faces: The first one is the Yale Face Database [6] which contains 165 grey-scale images in GIF format with a size of 320×243 pixels of 15 different persons, each with the same 11 facial expressions. The second source is the video taken from a traditional webcam. Therefore, the tests for face detection were divided into two (explained in sections 6.2.1 and 6.2.2) to check visually the accuracy of the model.

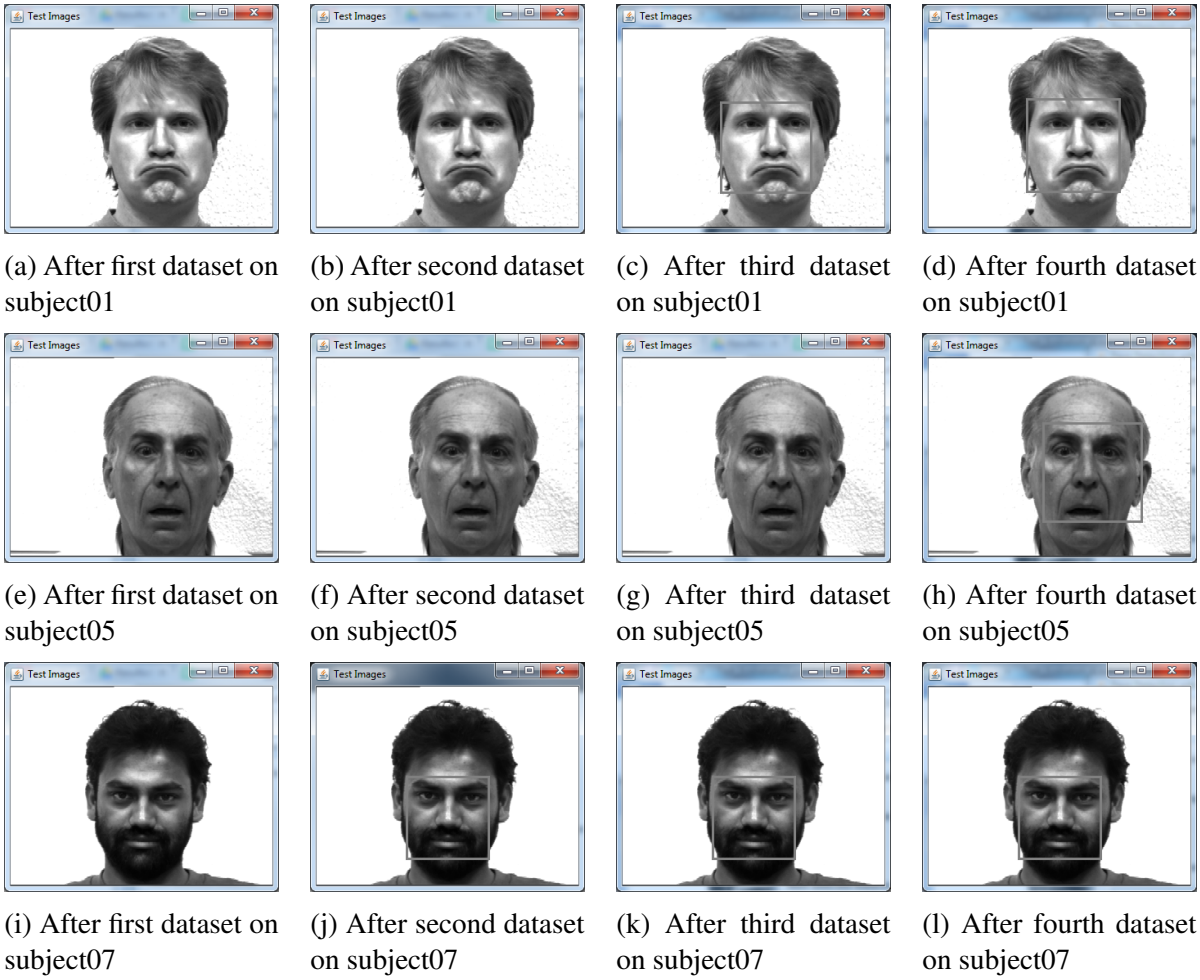
All the tests done in this section were executed with the face detector program developed for this research.

6.2.1 Faces files test

Three scenarios were executed in this test: Checking the performance of the model while the number of samples in the learning dataset increases, same person with different facial expressions, and same expression for different person.

6.2.1.1 Accuracy of the model vs Stream of data

This test was done in order to check how the performance of the model improves while more learning data arrives. The configuration for the test consisted of having a dataset of [9] with 6977 images with 2429 files of faces and 4548 files of non faces and divided into 5 smaller datasets each of them with 485 images of faces and 909 files of non faces. Moreover three images (each image corresponds to a person: subject01, subject05 and subject07) from [6] were chosen as test images. Then an on-line random forest was set up with $N = 100$, $D = 10$, $F = 50$, $\alpha = 20$ and $\beta = 0.05$. The execution of the test consisted of, always over the same forest, sending the three testing faces after each small learning dataset was applied to the model. The output of the test is shown in 6.4.



(a) After first dataset on subject01 (b) After second dataset on subject01 (c) After third dataset on subject01 (d) After fourth dataset on subject01

(e) After first dataset on subject05 (f) After second dataset on subject05 (g) After third dataset on subject05 (h) After fourth dataset on subject05

(i) After first dataset on subject07 (j) After second dataset on subject07 (k) After third dataset on subject07 (l) After fourth dataset on subject07

Figure 6.4: Visual test of performance of the On-line Random Forest vs Amount of arriving data

The results of this test show that after the first learning dataset was applied to

the model, no faces were detected for the three testing images as shown in figures 6.4a, 6.4e and 6.4i. When the second dataset was applied and again the three subject faces were tested, only the face of subject07 was detected as shown in figure 6.4j and the faces of subject05 and subject07 were not exposed as shown in figures 6.4b and 6.4f. After the third dataset was sent, the faces of subject01 and subject07 were detected, but no detection occurred in the image of subject05 (figures 6.4c, 6.4k and 6.4g respectively). In this point is relevant to notice that the model was able to detected the face of subject07 in the same way it did it after the second dataset was sent, this means that no alteration in previous detections happened after a new dataset is sent. Finally, after submitting the fourth learning dataset all the faces were detected correctly as shown in figures 6.4d, 6.4h and 6.4l.

Discussion: Is interesting to notice that the on-line learning behaviour was proved perfectly during the experiment. The performance of the model (in this case the face detection rate) improved while new data streaming was arriving. Some interesting facts can be explained according to the theoretical mark. The no face detection after the first learning dataset was submitted can be explained as, for most of the trees, an underfitting circumstance in which the learning information was not enough to fill the leaf nodes with data about images with faces and most of them had a non face as most common label. So, the depth of the trees in that moment was not good enough to separate properly the information of faces vs non-faces. After second, third and fourth learning datasets were sent, the trees of the forest divided more precisely the faces and non-faces information in different leaf nodes, so the accuracy was getting better. Moreover is interesting to notice that the model was still able to detect the faces that there were detected previously. The explanation of this behaviour can be supported as after the second dataset was sent to the testing face fell, for most of the trees, in a leaf node that clearly had more labels with faces than labels with no faces (or only faces labels), and after new learning data arrived, more faces information was accumulated in that node, so the same testing face fell again in the same leaf node confirming the previous detection. This test scenario is comparable with the test executed in section 6.1.1.

6.2.1.2 Same person with different facial expressions and light conditions

This test was done in order to check visually the accuracy of the model by detecting the same person under different conditions of facial expressions and light conditions. For this test an on-line random forest was created with the following parameters: $N = 100$,

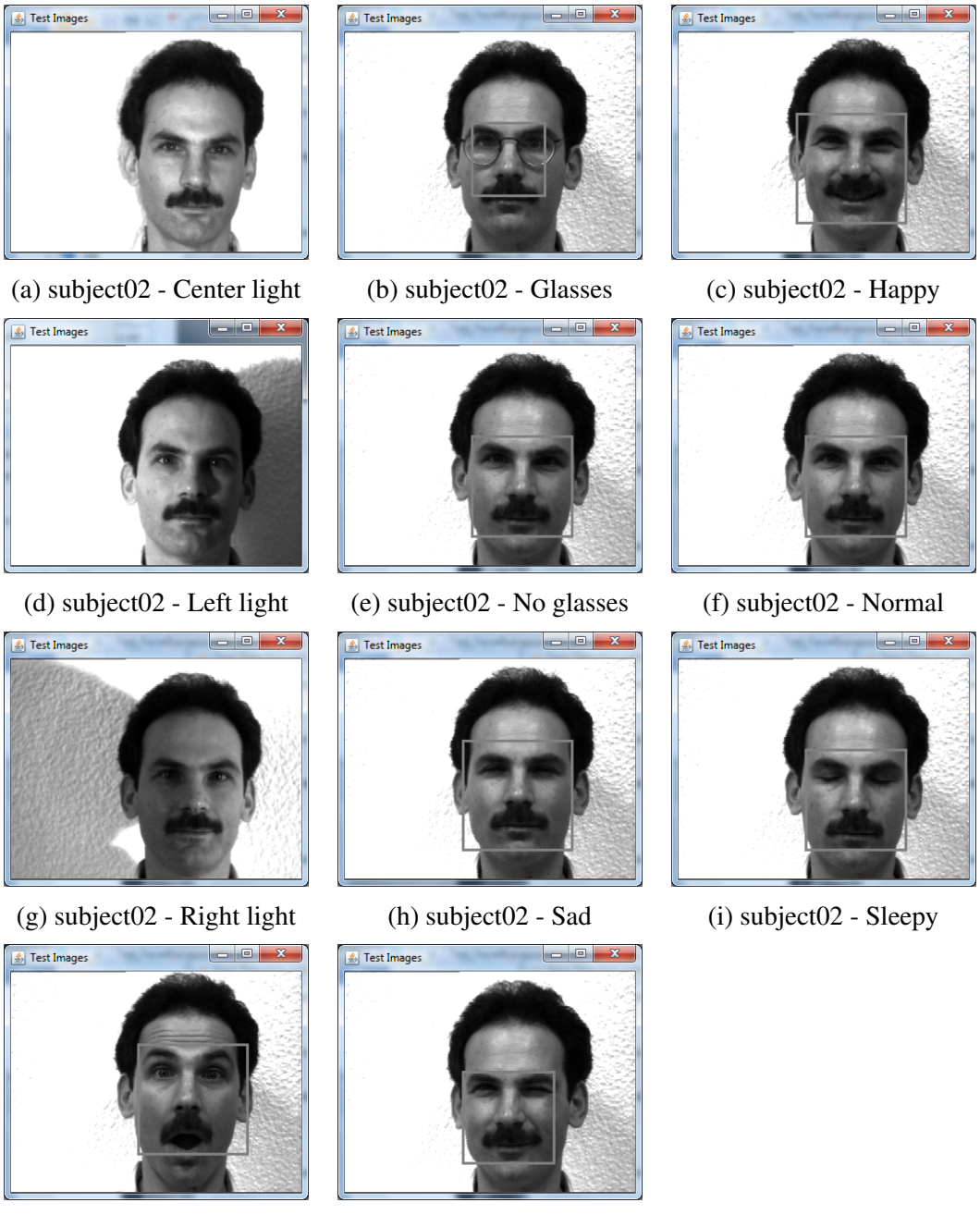
$D = 10$, $F = 50$, $\alpha = 20$ and $\beta = 0.05$. Then the learning process was executed by sending a dataset of [9] with 2429 images of faces and 4548 non-faces images. The test phase is executed by using 11 files of [6] with the image of the same person (named as subject02) with different face expressions and light conditions. The results of the face detection are shown in figure 6.5.

This experiment shows how the model is able to detect faces with different expressions. Figure 6.5e and 6.5f show the base case of the test in which a neutral expression of the face is and detected. The modifications in the expression of the face are shown in figures 6.5c, 6.5h, 6.5i and 6.5j, and in all of them the accuracy of the face detection is pretty decent being well managed by the model. This suggest that there were enough information in the learning dataset in order to detect faces showing sentiments such as happiness, sadness, laziness and surprise. An accurate and interesting detection is done in figure 6.5b in which the actor is hiding his eyes behind some big the spectacles. Less symmetry in the face expression is shown in figure 6.5k in which subject02 is doing a wink and even with that characteristic the model detects clearly the face. However, we can verify that there is no face detection in the images of figures 6.5a, 6.5d and 6.5g.

Discussion: The common factor in the failed face detection in these images involves a strong change in the scene illumination (a source of light is being used to illuminate the actor from different angles: centre, left and right), so we can conclude that the accuracy of the model is impacted because of the different light conditions of the scenes. There are several solutions for this issue, the first is related to add more information of faces with different light conditions in the learning data, and the second one is related to perform an image preprocessing task over the testing faces before they enter to the model to reduce the impact of excessive light and/or shadows on the faces. However the fact that the model is able to detect faces with different expressions suggest that there were information in the learning dataset of faces with different expressions and that for sure helped the model to detect the faces in the test. Moreover, the haar-like features are not fitted according to the expression of the faces so this shows how good can be generalized the features of the faces using the haar-like features strategy.

6.2.1.3 Same expression for different person

The accuracy of the model was tested by detecting the face of several images of different subjects, all of them doing the same face expression. For this test, the wink expression was selected due to the low level of symmetry a face shows when one eye



(a) subject02 - Center light (b) subject02 - Glasses (c) subject02 - Happy

(d) subject02 - Left light (e) subject02 - No glasses (f) subject02 - Normal

(g) subject02 - Right light (h) subject02 - Sad (i) subject02 - Sleepy

(j) subject02 - Surprised (k) subject02 - Wink

Figure 6.5: Tests over image of subject02 with different facial expressions and light conditions

is opened and the other one is closed. An on-line random forest was created with the following configuration: $N = 100$, $D = 10$, $F = 50$, $\alpha = 20$ and $\beta = 0.05$. Then the same dataset used in 6.2.1.2 was used as learning data, so the model received all the 6977 images of faces and non-faces. After that all the wink faces from [6] were used

as test images. The result of this experiment is shown in figure 6.6.

From the results of the test we can conclude: All the images have the same neutral conditions of illumination, so the face detection is not being impacted by this factor. The performance of the model is not impacted negatively because of subjects with facial hair, as shown in figures 6.6g and 6.6i. The faces of the actors in figures 6.6f and 6.6n are not centred in the image, however the model detects the face in a proper way. Figure 6.6k shows that feminine faces are detectable by the on-line random forest. The subjects in the images have different ethnicities and the model is able to detect their faces properly. For this test the images of subjects that combine spectacles plus the wink affects negatively the model as shown in figures 6.6h and 6.6n.

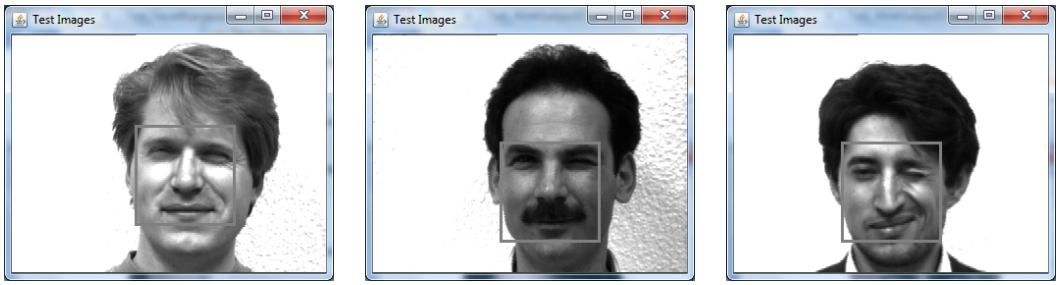
Discussion: There are three failed detections in this experiments: subject04, subject08 and subject13. The cases of subject08 and subject13 can be considered as a common fail because the characteristic of the combination of the weird face expression plus the spectacles accessory. We have the hypothesis that there were not enough faces (or there were not at all) in the learning dataset with those characteristics, so these faces always fell in leaf nodes with predominant non-face label. The case of subject 04 was a very hard case due to this image was always the most complicated face to detect among all the experiments we performed during this project (after adding more learning data and tuning the parameters of the forest many times this face was not detected). There are two hypothesis in which the next researcher could work: firstly verify that how this face matches with the haar-like features and secondly add more learning faces with similar subject04 facial's characteristics. The rest of the faces were detected accurately across all the experiments we did.

6.2.2 Webcam test

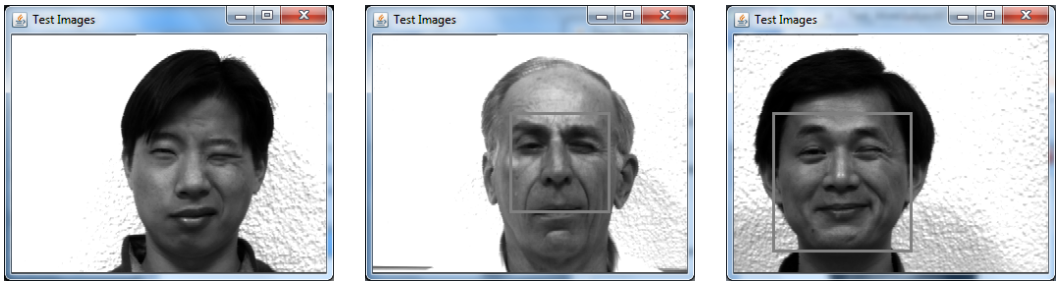
The final test is done by checking how the program performs in the process of face detection in images taken from the webcam.

6.2.2.1 Snapshots

Finding a good configuration of parameters is a process that requires continuously changing/testing tasks, as was proved in section 6.1.3. So the first task was to find a good set of parameters that falls into a acceptable local minima. The tests suggest the following configuration as good test: $N = 10$, $D = 20$, $F = 50$, $\alpha = 20$ and $\beta = 0.05$. We used a learning set of 1404 faces and non-faces images and the snapshots from

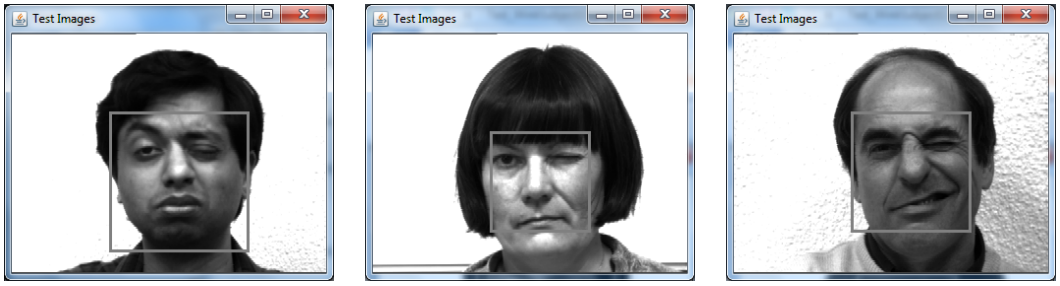


(a) Wink of subject01 (b) Wink of subject02 (c) Wink of subject03

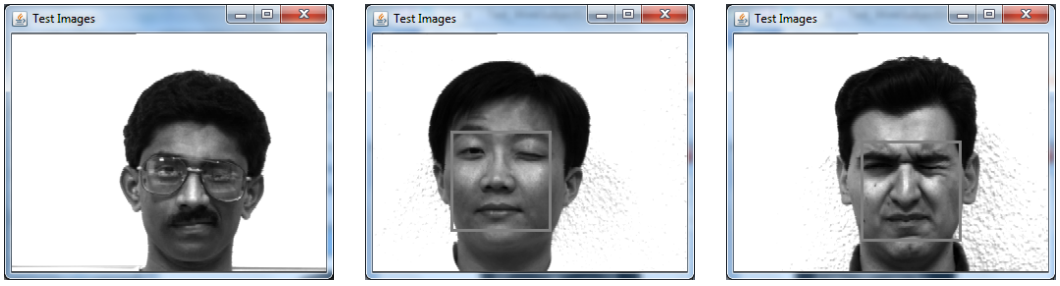


(d) Wink of subject04 (e) Wink of subject05 (f) Wink of subject06

(g) Wink of subject07 (h) Wink of subject08 (i) Wink of subject09



(j) Wink of subject10 (k) Wink of subject11 (l) Wink of subject12



(m) Wink of subject13 (n) Wink of subject14 (o) Wink of subject15

Figure 6.6: Tests over images of 15 subjects doing a wink

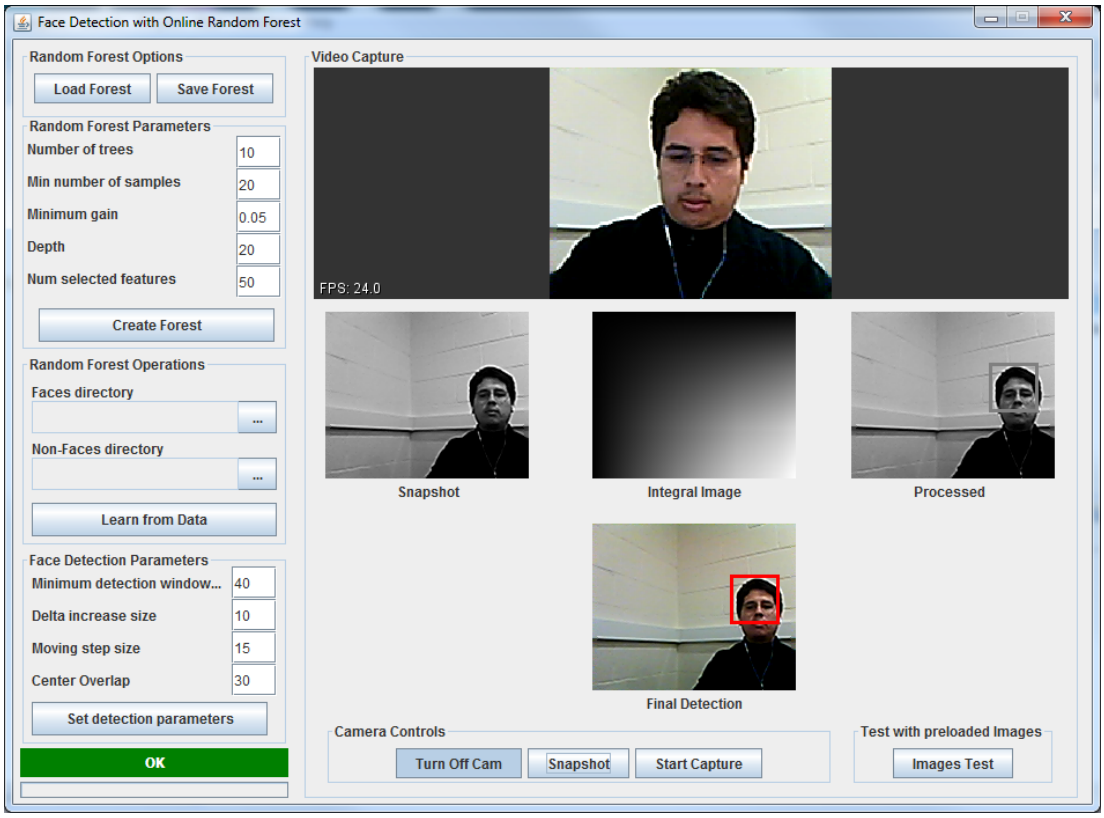
the webcam were taken using the face detection program. The results of the face detection are shown in figure 6.7. The red square surrounding the face in the *Final Detection* panel shows accurate results in the process. From figure 6.7a we can see a face detection of a small face that is not centred in the image. The same forest was used to take the picture in figure 6.7b in which there is a big face in a different position of the image. So the forest was able to learn from the learning faces and the scanning process ensures finding as many faces as possible in the whole image, independently of their size and position.

6.2.2.2 Video

The final test consists in detecting faces from a video. We use the same forest used for the previous snapshot test in section 6.2.2.1, but this time the continuous capture mode of the image processing was activated. The results we found were very interesting, the program shows it is able to detect faces continuously from the stream of video acting as face tracking program. This result shows that the program fits with the non-functional performance requirements presented in section 5.1.4. Let's remember that in one single frame the forest is tested for each position and size the window has during the scanning process.

The figure 6.8 shows screenshots of the execution of the face detection program while detecting faces using the video from the webcam.

One interesting test was done in order to detect multiple faces in the same video frame. This time several subjects were capture by the webcam. The results in figure 6.9 show that the scanning process solves this issue perfectly by detecting two and three persons in the image at the same time.



(a) Small face

(b) Big face

Figure 6.7: Test of face detection taking a snapshot from webcam

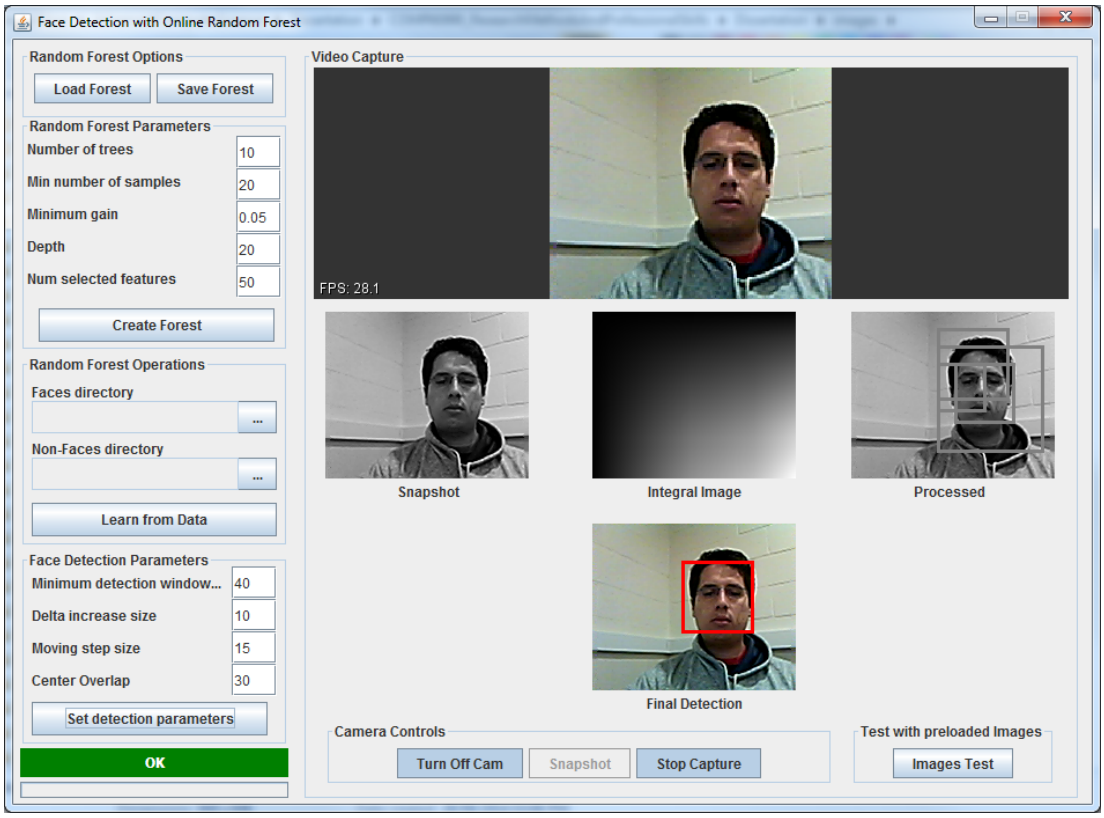


Figure 6.8: Test of face detection using video from webcam

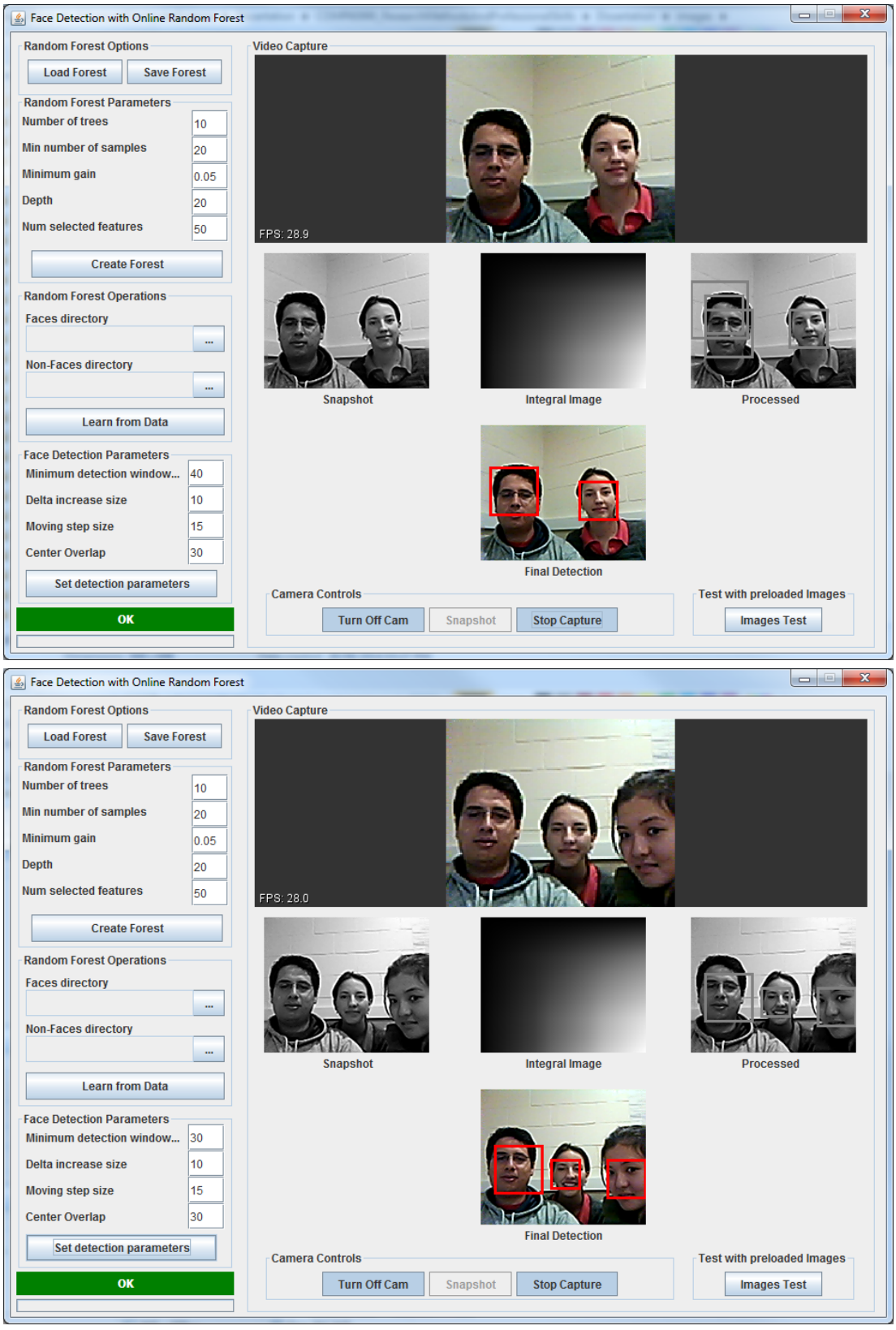


Figure 6.9: Test of face detection using video from webcam to detect multiple faces

Chapter 7

Conclusions and future work

7.1 Conclusions

The project started with the idea of developing an on-line learning model based on decision trees, and go beyond there by building and interactive program that showed a real application of the algorithm by solving a classification problem. So we decided to solve a face detection problem obtaining images from different data sources including a stream of video.

Learning and deliverable objectives were established, and they were used as trace route for developing the project. The learning objectives were reached because we completed the following activities:

- The background process was carried out in order to understand previous studies in this subject and to know the actual state of this topic. Several strategies and techniques were studied and analysed and the concept of on-line learning was understood.

- A complete review of the random forest concept was done, starting from the traditional decision trees, going through the random forest model and extending their concepts to the on-line models. We conclude from here that in the off-line mode a training phase occurs using the complete set of learning data, and after that testing phase is done to check how good was the previous training phase. In the on-line mode the training phase is replaced by a learning phase in which the model could receive learning data in any time, even after a testing phase was applied. This models are characterized mainly because their accuracy improves while more learning data is processed.

- Talking about the decision trees, the main difference between the off-line and on-line concepts lies on the split criterion of the nodes inside the tree. In one hand, the nodes of the trees in off-line models process the complete the training data in order to find the best possible value for a feature among all the dataset to split it into new subsets that will be passed to its new children nodes. On the other hand, because of the way the data arrives to the model (a streaming way), the nodes of trees in the on-line learning algorithm have to take the split decision based on the learning data that has passed through the node in a given moment.

- When we analyse the ensemble models (the forest), in the off-line process a bagging algorithm is done over the training dataset to send slightly different sets of data to each tree in the forest, in the on-line mode this process is replaced by calculating a randomly the number of times a sample must be sent to each on-line tree of the ensemble.

- The background also carried out doing a wide research based on several techniques to solve the face detection issue. The study led to work with the mathematical tools used in [31], so the extraction of faces features was made by working with haar-like features and calculating in a fast way their value applying the concept of the integral image. By using this technique any other object could be detectable as shown in other projects.

The deliverable objectives were reached because with the project we are delivering the following artefacts:

- An algorithm of an on-line random forest was designed and developed taking into account the theoretical mark related to the on-line random forest machine learning models. The implemented split criterion and the on-line bagging process met the requirements of the these type of models.

- The implemented algorithm was tested over several machine learning datasets measuring its error rate and its performance. For all the datasets it was concluded that the error rate decreases while new stream of data arrives to the algorithm, so it met the principal function related to the on-line learning process. The performance of the algorithm is affected by changing all the parameters of the forest. This concept was verified by doing a ROC analysis over an on-line

random forests created with different configurations of parameters. This test finished by suggesting a method to find a good combination of parameters to create a forest.

- A face detection program was built over the main algorithm of the on-line random forest. Modules of image processing were added in order to treat images taken from files, and photos and video taken with a webcam. The concepts of Integral Image and Haar-like features were taken from [31] to detect the features in faces. The error rate of the face detection process is comparable with the results obtained with the machine learning datasets taking into account that the face detection process was improving while more learning faces and non-faces images were processed by the model. The model was accurate enough to pass the tests over databases with faces images that have persons of different gender, ages and ethnicity with facial expressions (neutral, happy, sad, sleepy, wink, etc). It also detected faces of people with facial hair and with glasses. The program also detected the faces from pictures and videos taken from traditional webcams. It also showed that it is able to track multiple faces in a video streaming.

- The face detection application meets with the requirements of performance and portability. It is able to track faces processing continuously images from a video stream, and is a program that could be installed in whatever platform with a java runtime environment. It can be used as tool in Machine Learning classes in order to show interactively the behaviour and characteristics of on-line learning models and specifically of on-line random forest. This software also shows some image processing concepts that are the base of Computer Vision classes.

In all the research process we faced several challenges. Because of the random nature of the model it was impossible to execute exactly the same test each time we required, so all the tests were executed several times and we took measures statistically to be able to analyse and compare results with previous executions. We also noticed that each parameter of the on-line random forest affects the learning process, so finding a good combination of them in the process of tuning was not an easy task, it and requires experience and good knowledge of the concepts. Another interesting challenge was the one related to join the faces in [9] and [6] with the features in [2], for this task a normalization process over the features was developed in order to guarantee that the sizes of the images was not a fail factor in the process of learning and testing.

Even though the face detection problem is an already solved issue, traditional face detector implemented in photographic cameras and cellphones are not able to learn because they are based on off-line learning mechanisms. The on-line learning approach can be used as new feature of these devices in order improve their accuracy while more pictures or videos are being taken.

7.2 Future Work

Next stages of the research could deal with some of problems that were found during the execution of the project. One of the most important questions that arise during the project was related to the maximum and minimum values of the features in the data stream. We asked ourselves if exists the chance of knowing in advance (before the first learning sample arrives to the model) the maximum and minimum values of all the features in the stream of data. If we decided to answer yes to that question, the calculus of the random threshold for each feature could have been done before the first sample arrives to the model. Because we decided to answer "no" (we make the assumption from the beginning that we do not known the range of values for the features) in the model that we developed some samples are store temporary in the nodes to calculate the random threshold between the maximum and minimum values known until that moment. In order to avoid storing samples in the nodes, the future work we propose is related to calculate statistically and with high degree of certainty the maximum and minimum values of the features according to very few the arriving samples. After that the random threshold could be calculated in early stages of the process.

The research also shows that is possible for the model to evolve according to how good or bad it is predicting the values of the labels of the samples. With that in mind a further step consists in managing individually the error rate of the members of the ensemble and replace the ones with high error rate by new decision trees. This will become in a stronger model due to it would change slowly according to changes in the incoming data.

In the faces context new improvements over the program could be done. The strategy we use to detect faces is based on the haar-like features exposed in [31]. This features are called weak classifiers and a combination of them become in a more reliable classifier called strong classifier or stages. An idea we have is to manage the features of the input samples as a strong classifier, so it is possible that the performance of our model could improve by doing this change. Moreover we realized that

this technique is very sensitive to the light of the environment so an image preprocessing step is suggested (for example equalize the input sample) in order to enhance the image before sending it to the model. Finally the execution time of the model could be improved by doing a code review and by implementing the algorithm in a GPU. This will decrease the execution time of the face detector and would improve the user experience by showing a smoother transition between the video taken from the webcam and the final processed image.

An strategy for selecting the best combination of input parameters can be done as future work. We developed an strategy to find a good combination of parameters but not necessarily the best.

The face detection issue solved in this research is a preamble in the objects detection problem. Future work over this research could be done to detect and recognize whatever object in the scenes.

Finally, an evaluation of the user interface of the face detector could be done in order improve the usability of the program.

Bibliography

[1] Assembla repository. www.assembla.com.

[2] The opencv reference manual. http://opencv.org/.

[3] K. Bache and M. Lichman. Uci machine learning repository, 2013.

[4] Badgerati. Computer Vision - The Integral Image. http://computersciencesource.wordpress.com/2010/09/03/computer-vision-the-integral-image/, Sep 2010.

[5] J. Basak. Online adaptive decision trees. *Neural Computation, Massachusetts Institute of Technology*, (16):1959–1981, 2004.

[6] P. Belhumeur, J. Hespanha, and D. Kriegman. Eigenfaces vs. fisherfaces: Recognition using class specific linear projection.

[7] P. N. Belhumeur, J. P. Hespanha, and D. J. Kriegman. Eigenfaces vs. fisherfaces: Recognition using class specic linear projection. *IEEE Trans. on PAMI*, 1997.

[8] K. Berggren and P. Gregersson. *Camera focus controlled by face detection on GPU*. PhD thesis, Lund University, 2008.

[9] M. C. F. Biological and C. Learning. Cbcl face database 1. http://cbcl.mit.edu/software-datasets/FaceData2.html.

[10] L. Breiman. Bagging predictors. *Mach. Learn.*, 24(2):123–140, Aug. 1996.

[11] L. Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, Oct. 2001.

[12] G. Brown. The comp61011 guide to machine learning, 2013.

[13] C.-C. Chang and C.-J. Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[14] P. Cortez, A. Cerdeira, F. Almeida, T. Matos, and J. Reis. Modeling wine preferences by data mining from physicochemical properties. *Decis. Support Syst.*, 47(4):547–553, Nov. 2009.

[15] P. Domingos and G. Hulten. Mining high-speed data streams. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 71–80. ACM, 2000.

[16] T. Fawcett. An introduction to roc analysis. *Pattern recognition letters*, 27(8):861–874, 2006.

[17] B. Firyn. Java webcam capture api. `https://github.com/sarxos/webcam-capture`, 2012 - 2013.

[18] R. C. Gonzalez and R. E. Woods. *Digital Image Processing (3rd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[19] T. M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997.

[20] N. C. Oza. Online bagging and boosting. Technical report, Intelligent Systems Division - NASA Ames Research Center, 2005.

[21] N. C. Oza and S. Russell. Online bagging and boosting. Technical report, Computer Science Division, University of California, 2001.

[22] S.-K. Pavania, DavidDelgado, and AlejandroF.Frangi. Haar-like features with optimally weighted rectangles for rapid object detection. 2007.

[23] J. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[24] J. R. Quinlan. *C4. 5: programs for machine learning*, volume 1. Morgan kaufmann, 1993.

[25] A. Saffari, C. Leistner, J. Santner, and M. Godec. On-line random forests. Technical report, Institute for Computer Graphics and Vision - Graz University of Technology, 2009.

[26] S. Schmitt. Real-time object detection with haar-like features. 2010.

[27] J. Shotton, T.-K. Kim, and B. Stenger. Boosting and randomized forests for visual recognition, 2009.

[28] M. Turk and A. Pentland. Eigenfaces for recognition. *Journal of Cognitive Neuroscience*, 3(1), 1991.

[29] C. Twining. Short lecture: Face detection, msc. advanced computer science, 2014.

[30] M. A. O. Vasilescu and D. Terzopoulos. Multilinear analysis of image ensembles: Tensorfaces. *A. Heyden et al. (Eds.): ECCV*, pages 447–460, 2002.

[31] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. *CONFERENCE ON COMPUTER VISION AND PATTERN RECOGNITION*, 2001.

[32] P. Viola and M. J. Jones. Robust real-time face detection. *International Journal of Computer Vision*, 57(2):137–154, 2004.

[33] A. Wang, G. Wan, Z. Cheng, and S. Li. An incremental extremely random forest classifier for online learning and tracking. 2009.

[34] P. I. Wilson and J. Fernandez. Facial feature detection using haar classifiers. 2006.

Appendix A

User Manual

This section describes in detail the functionality of the face detector program built upon the On-line Random Forest. The install and uninstall instructions are explained, the hardware and software requirements are specified, the way to execute it and a detailed explanation of how to use it is shown.

A.1 Requirements, Install and Uninstall

A.1.1 Hardware requirements

We recommend the following minimum hardware configuration to execute the program:

- Processor: Intel Core i3 2.27GHz
- Installed Ram Memory: 4.0 GB
- Webcam: Integrated or preinstalled webcam

A.1.2 Software Requirements

The software configuration should meet the following requirements:

- Operative System: The program was tested in the following systems:
  - Windows XP/Vista/7/8
  - Mac OS X

  - Linux

- Java Runtime Execution (JRE) 1.7 or superior
- Driver of the webcam properly installed and working.

A.1.3 Install

A traditional zip file contains a folder with the binary file to be executed. The installation process must be done by extracting the content of the zip file into a folder of your preference (please be sure you have all the privileges over that folder). From now on, we will call that directory as *HOME_ORF_FaceDetector*

A.1.4 Uninstall

To uninstall the program simply look for the directory *HOME_ORF_FaceDetector* in your file system and delete the folder.

A.2 How to execute the program

- Open a terminal console
  - In a windows machine, enter to a command line console by executing the *cmd* in the execute/search box of the start button.
  - In a mac/linux system, open a terminal console.
- Move to the *HOME_ORF_FaceDetector* by typing *cd HOME_ORF_FaceDetector*
- Verify you are in the right folder
  - In a windows machine, type the command *dir* and verify that in the list of files exists the entry OnlineRandomForest.jar
  - In a mac system, type the command *ls* open and verify that in the list of files exists the entry OnlineRandomForest.jar
- Type the command *java -jar OnlineRandomForest.jar*. The system should launch the screen showed in A.3

A.3 How to use it...

Once the program is launched the screen in figure A.1 will appear. It contains several panels each of them with its very specific functionality.

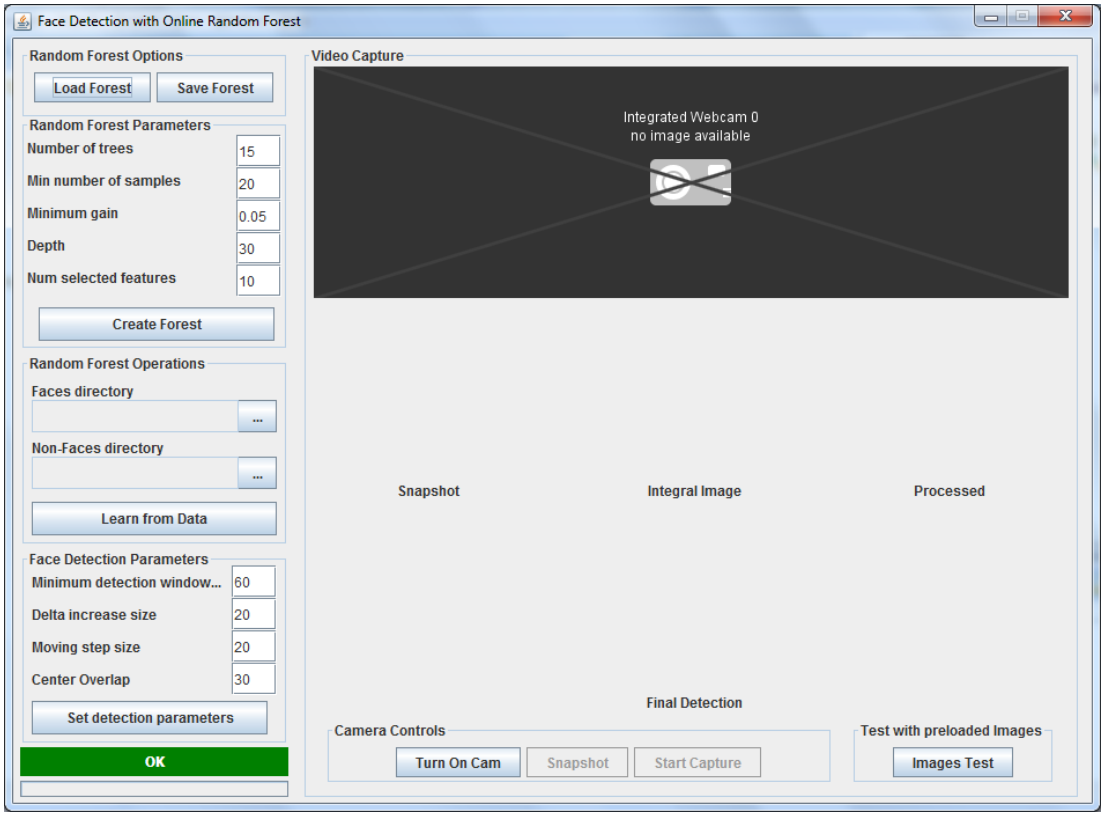


Figure A.1: Main screen of the face detection program

- **Random Forest Options**: This panel has 2 buttons: The *Save Forest* button allows to save the current state of a forest in a directory of the file system. The name of the file will have the following format: *orf_Faces_YYYY-MM-DD_hh-mm-ss_nElements_N_α_β_D_N.orf*. YYYY-MM-DD_hh-mm-ss is the date and time when the forest was saved. *nElements* is the number of samples that were used in the learning process, N is the number of trees created for that forest, α is the minimum number of samples that must pass through a node before try to split, β is the minimum value of gain the information in the node should have before trying to split, D is the maximum depth of the trees, and N is the number of features that will be selected each time a node is created to apply the random tests over them. The file is saved with extension *.orf*. Then the *Load Forest* button allows to load previously saved forest. The only files the program can

load are those saved previously with the Save Forest button. An image of this panel is shown in figure A.2.

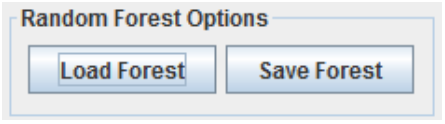


Figure A.2: Panel Random Forest Options

- **Random Forest Parameters**: This panel has the parameters for creating an online random forest. The meaning theses parameters is explained in 4.1. Once all the values are selected the Create Forest should be pressed to create a new online random forest. Every time this button is pressed a new forest is created and if the previous forest was not saved, that configuration with is learning process will be lost. The figure A.3 shows this panel.

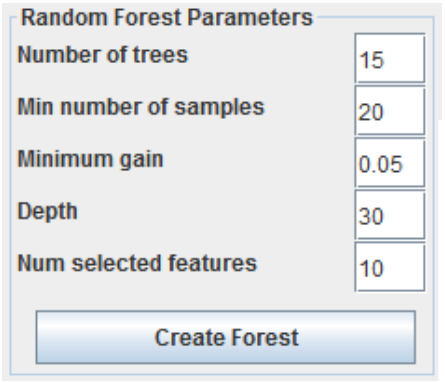


Figure A.3: Panel Random Forest Parameters

- **Random Forest Operations**: The fields of this panel are related to choose the learning sets of faces and non-faces images. Once the buttons with three dashes are pressed a dialogue window will appear asking for the directory of the files. Is responsibility of the user to choose the right directory because the program does not validate the information that is going to be processed. In the installation directory are included datasets with faces and non faces images in PGM format. If new files want to be included is suggested to create new folders for the new faces and non-faces images. The program will take all the files in the directory only if all of them have the same format. Once the directories are selected the button *Learn from Data* must be pressed in order to send the information for to the forest created with the Random Forest Panel. This panel handles the learning phase of the process. The figure A.4 shows this panel.

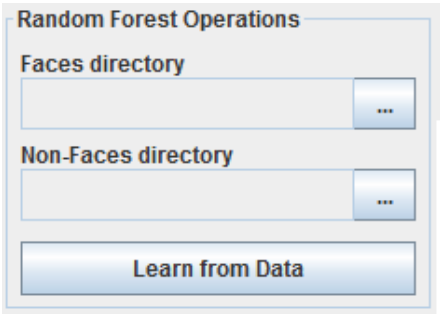


Figure A.4: Panel Random Forest Operations

- **Face Detection Parameters**: This panel contains the configuration of how the program is going to scan the images in the testing phase. The field emphMinimum detection window size has the minimum value of the window's side size in pixels. This means for example if 60 is the selected value, the program will start scanning the image that is going to be tested with a window of 60×60 pixels trying to find faces that fit in that window. The field *Delta increase size* has the value in pixels of how fast the scanning window is going to grow. Following the previous sample if this value is 20 after the 60×60 window finishes scanning the image a new window of 80×80 will start the process again to find faces fitting in the new window size. The *Moving step size* says how fast the windows are going to move while scanning the images. Again with previous sample if 25 is selected in this field the 60×60 will look for faces each 25 pixels starting from point $(0,0)$, this means the first scan will be done in the coordinate $(0,0)$ and the following scan will be done in the coordinate $(25,0)$. After all that row is scanned the next next position of the window will be the coordinate $(0,25)$. Finally the *Center Overlap* fits how far should be the windows if some of them find a face in the same position. Let's say a window of 60×60 is scanning the image, and found a face in the position $(100,110)$. Then when the window grows and its new size is 80×80 scans the image and found a face in the position $(105,108)$. Because the Manhattan distance of the points where faces where found is less than 30 the program assumes that the two windows found the same face in the picture. So an average of the windows is done to determine the final window size to show the face detection. All this parameters are sent to the program once the button *Set detection parameters* is clicked. The figure A.5 shows this panel.

- **Status and progress bars**: The status bar indicates to the user if the program is ready to receive any command or if it is performing a task that takes some time

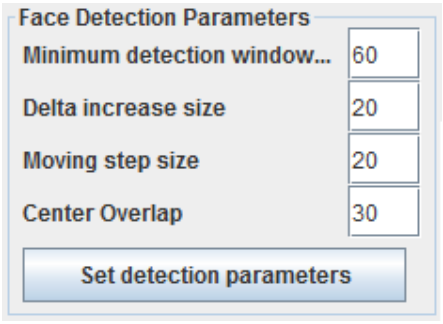
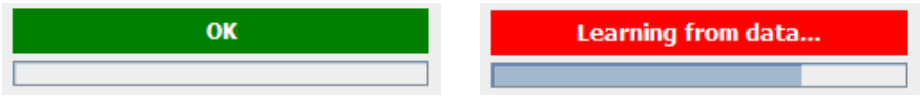


Figure A.5: Panel Face Detection Parameters

to be finished. In the first case the progress bar has a green colour with the label *OK* and in the second case its color is red and the label indicates the type of task is doing in that moment. The progress bar indicates visually the remaining time to complete a task. If the progress bar is red the user should not give more instructions to the program until it returns to green colour again. Figures A.6a and A.6b show the status and progress bars.



(a) Program ready to work (b) Program working

Figure A.6: Status and progress bars

- **Video Capture**: The video capture section has 4 panels: the first one with label *Snapshot* show the image taken from the webcam in grey-scale format that is going to be processed by the on-line random forest; the second panel *Integral Image* has the normalized Integral Image of the previous panel snapshot; the third panel labelled as *Processed* shows the face detection done by the forest; and the final panel with *Final Detection* shows the final detection after applying the values of the *Face Detection Parameters* to the image of *Processed* panel. All this panels are activated once the buttons *Snapshot* or *Start Capture* are clicked. Figures A.1, A.8, A.9 and A.10 show the possible status of all of these panels.

- **Camera Controls**: The figure A.7 shows this panel. It has the buttons in charge of doing the operations with the webcam. The button *Turn On/Off Cam* turns-on and turns-off the webcam. By clicking this button only the Video Capture panel displays the video from the webcam without doing any image processing with the forest as shown in A.8. Once a forest has been created and if it has learn

from some data, the *Snapshot* button captures a picture from the webcam and sends it to the forest. The figure A.9 shows the result of a face detection with the snapshot button. Finally in this panel the button *Start Capture* allows to send concurrently images taken from the video to the forest in order to detect the faces in the video as shown in A.10

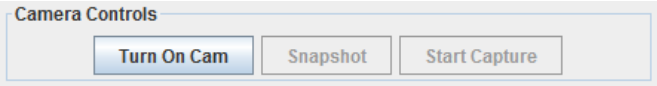


Figure A.7: Panel Camera Controls

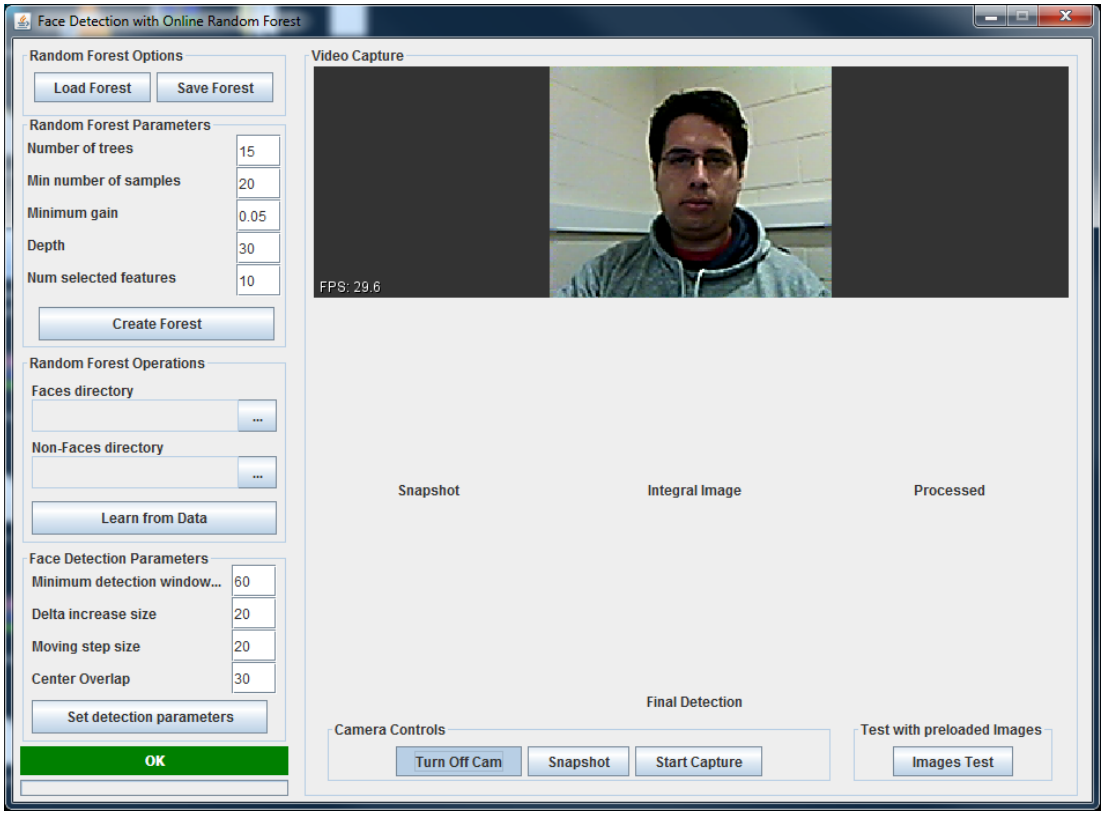


Figure A.8: Panel Camera Controls

- **Test with preloaded images**: This panel allows to test the model by sending images from files. Once the button is pressed, a directory holding the files must be selected. The program will show the detection result for each file in the folder. All the formats of the files in the selected directory must be the same. The figure A.11 shows this panel, and the figure A.12 shows a result of processing a face image.

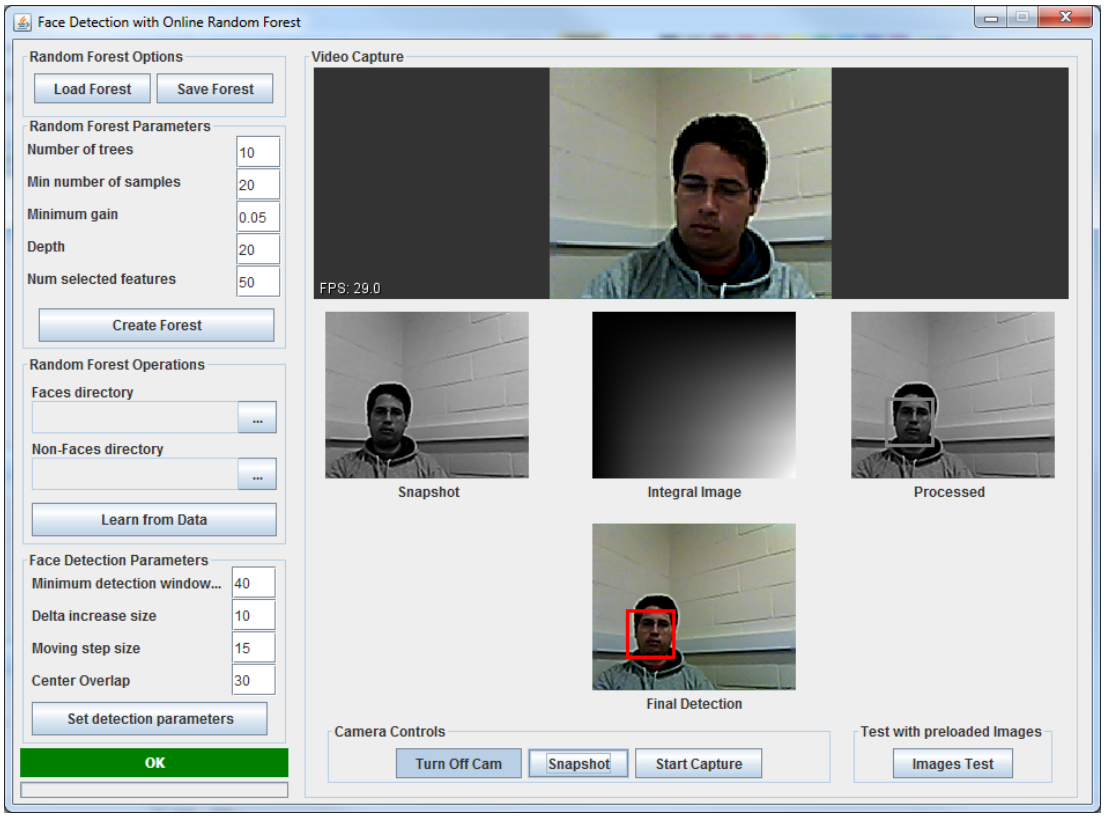


Figure A.9: Panel Camera Controls

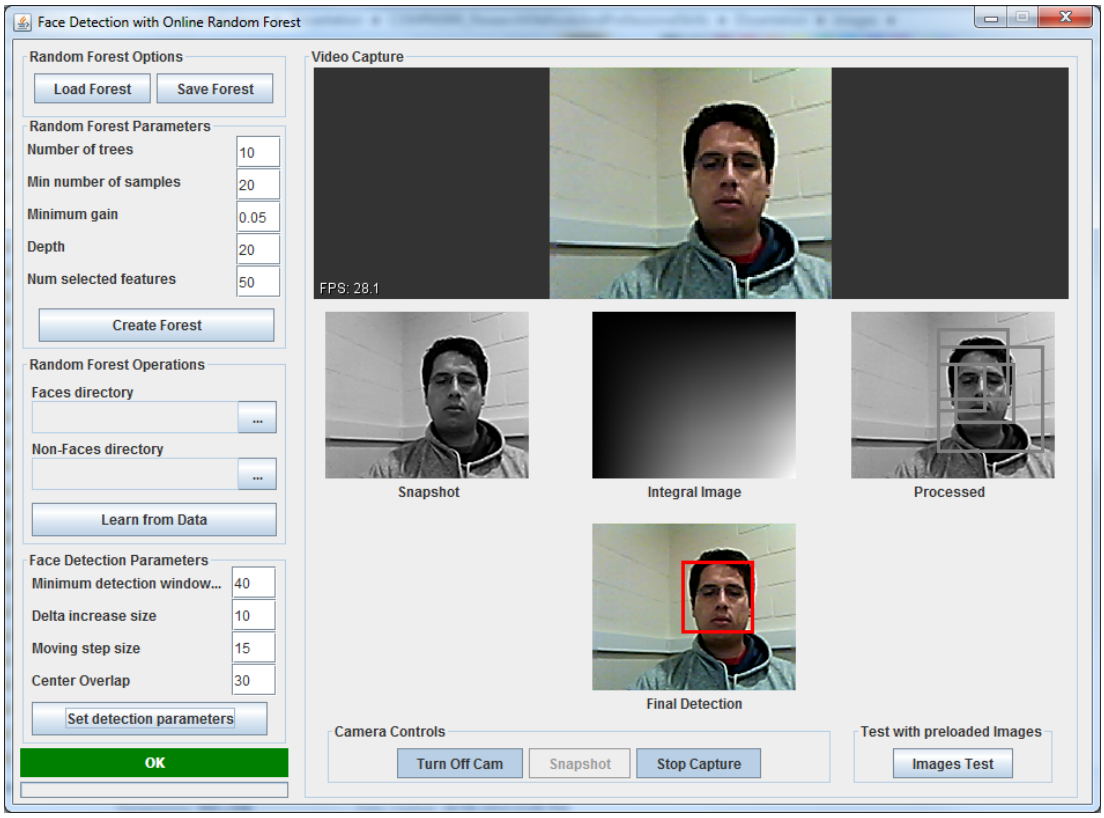


Figure A.10: Panel Camera Controls

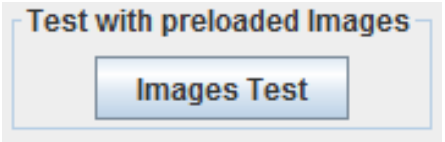


Figure A.11: Panel Test

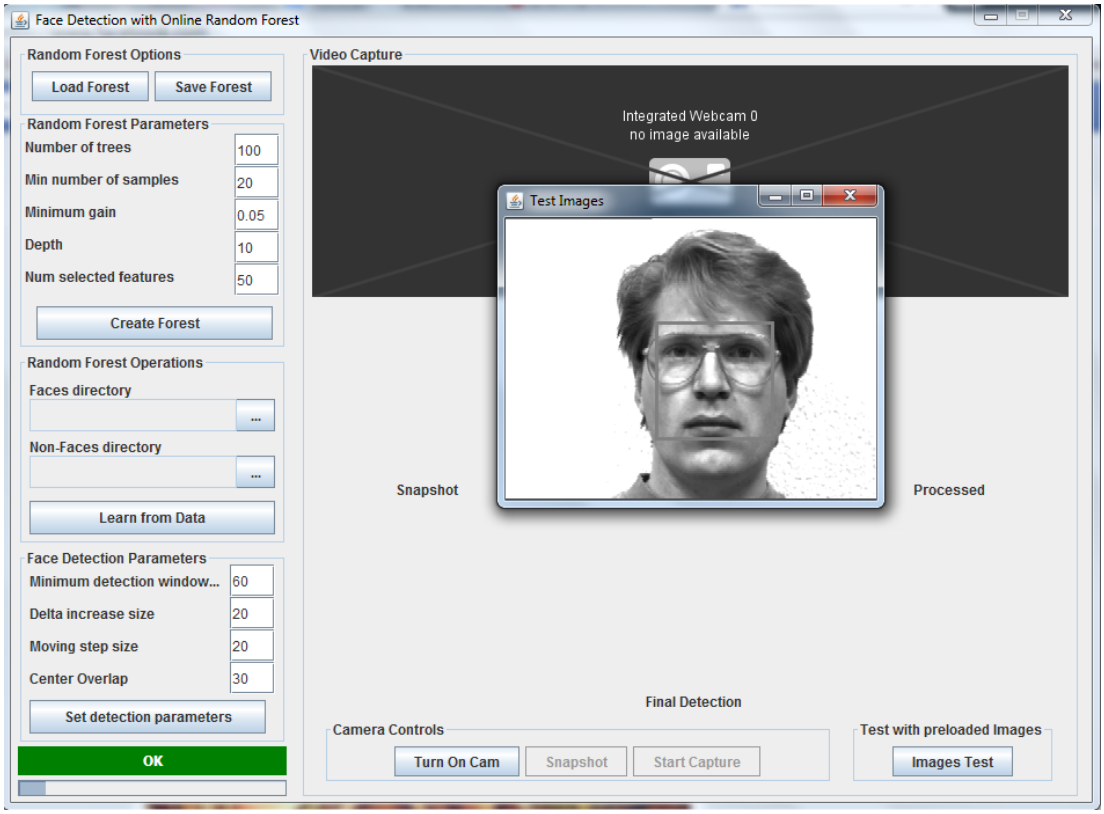


Figure A.12: Face detection in file image