Analyzing low light adaption methods for pre-trained cascade of boosted MB-LBP classifiers

Måns Jarlskog

May 2016

Abstract

In applications of real time gesture detection on devices which have cameras and limited power supply it is important to consider both robustness and maintaining a low power consumption. A well explored approach is the use of cascades of boosted classifiers. As retraining is a very cumbersome process, online environment adaption would be an appealing approach to counter situations where the boosted cascade detector underperforms.

The purpose of this thesis is to investigate ways to adapt a pre-trained boosted cascade detector, with regression trees as weak classifiers, to an elusive environment in test-time.

The approach is unsupervised collection of data in real time on highly probable false negatives coming from temporal series of frames. The data is then used to find features associated with missed detections and increase their influence on the classifier while keeping the unwanted impact bounded. Adaption is accomplished by reclassifying any new low confidence data with a set of adapted leaves.

In this thesis, adaption to a low light environment is investigated. Methods to unsupervised collect data and output adapted leaf values are presented. The results suggest convergence of the increased detection rates up to 13%, with respect to both the number of target stages and the size of the reclassification area, can be achieved with a few percent increase of feature evaluations.

In order to draw further conclusions, about the general performance as well as the environment biased performance, more data is needed together with appropriate tools for analyzing results, such as ROC-curves.

List of abbreviations

The following table describes some acronyms and abbreviations used throughout the thesis.

Abbreviation	Meaning
GPR	Gaussian Process Regression
RGB	Red, Green and Blue light (color model)
LBP	Local Binary Patterns
MB-LBP	Multi scale Block Local Binary Patterns
OpenCV	Open Source Computer Vision
PAC	Probably Approximately Correct
SVM	Support Vector Machines
TP	True Positive
FP	False Positive
FN	False Negative
TN	True Negative
ERM	Empirical risk minimization
VC-dimension	Vapnik–Chervonenkis dimension
AdaBoost	Adaptive Boosting
FPS	frames per second
G	Graph; A set of vertices and a set of edges
ROC	Receiver Operating Characteristic
f_evals	feature evaluations
ROI	Regions of interest; Set of image patches used to approximate the target domain
LP	Linear programming (also called linear optimization)

Table 1: List of acronyms

Contents

Acknowledgement

I would like to thank my supervisor at Crunchfish, Paul Cronholm, for all the help, input and feedback I received during this thesis. My sincere thanks go to my supervisor Magnus Oskarsson at Centre for Mathematical Sciences at Lund University. I wish to express my gratitude to Johan Metzner and all other developers at Crunchfish for their friendly and useful support. Last but not least, it is a great pleasure for me to express my heartfelt thanks to the members of my family. They have been an invaluable source of encouragement during my studies.

1 Introduction

Boosted cascade classifiers used for gesture recognition are trained with supervised machine learning algorithms. In supervised learning the training set is used to approximate the application domain. Whenever the training set fails to represent a target domain, poor performance may follow as a result. These problems are typically first discovered in test time and the standard approach would be to gather more training data associated with the particular situation and perform a retraining. As retraining is a cumbersome process (and the application domains numerous), collecting data for every situation the classifier underperforms in may be infeasible.

A hypothesis is that if a target domain is only partly approximated, the classifiers' performance within the domain may still be improved, possibly at the expense of the other domains.

In this thesis the focus is to investigate the hypothesis above in the case with Viola-Jones based cascade classifiers with regression trees as weak hypotheses.

The approach chosen is to conduct the adaption by altering the leaf values of the weak trees. An advantage with this approach is that any adaption result obtained is easy to store, apply and reverse. By storing earlier obtained adaption results, one can focus on recognizing the elusive environment rather than redo the entire adaption procedure.

Contribution of this thesis concerns methods to identify and unsupervised collect data from partly approximated domains as well as methods to produce new leaf values.

1.1 Related Work

The use of image object recognition for real-time applications can be traced back to the paper "Robust Real Time Face Detection" by Paul Viola and Michael Jones. The paper was published in 2001 and described the first object detection framework that was able to provide feasible results in real time. The original motivation was to solve the problem of face detection but it can be trained for many other types of objects.

Since then, enormous amounts of research and development have been done in the field and many popular object recognition approaches still use an underlying Viola-Jones based framework.

In 2011 the paper "Online Domain Adaption of a Pre-Trained Cascade of Classifiers" by Vidit Jain and Erik Learned-Miller was published. The paper introduced a method to soften the issues related to asymmetry between the training sets and test data on a pre-trained classifier, without retraining. Their work focused on the case where no labeled data were available and every image was seen as a new target domain. By assuming the original classifier outputted a continuous number, the approach was to encourage smoothness, at stage level, by reclassifying low candidate regions near the threshold such that similar data receive similar score-values. Their work was implemented and tested on a Viola-Jones based Haar-cascade.

The above paper mentioned two problems with the two-stage naive approach to perform the same as above: "to scan for high confident faces, and then retrain with the detection model according to the high-confident face and non-face regions". The first problem mentioned was the computational aspect of retraining in a real time environment. The second was that the retraining may lead to overfitting of classifier to the detected regions [36].

To bypass these problems, Jain and Learned-Miller used a Gaussian process regression (GPR) scheme as its parameters can be efficiently computed. Furthermore, the GPR included a term for the prior probability that prevented overfitting. The GPR, trained on high confident data, was used to reclassify the low confident data. The procedure was repeated for every image under the assumption that every image represented a new domain. This approach yielded improved performance for the face detection problem on a standard data set

[36].

1.2 Goal of the thesis

The main objective is to explore the possibility to adapt pre-trained Viola-Jones based cascade detectors that are using regression trees as weak classifier, to a target environment where the detector underperforms. Unlike the earlier work "Online Domain Adaption of a Pre-Trained Cascade of Classifiers" where every test-image represented a new domain, this thesis will focus on adaption to a single domain where there are many test images available. The test images are furthermore temporal ordered in sequences (videos).

The adaption will be implemented and executed in a two stage procedure. The first stage is to collect data from the new domain by counting leaf evaluations for each leaf, by each predicted class (positive and negative). The second stage is to produce a new set of adapted leaf values for the regression trees that are weighted in favour of the new domain.

The idea is to bypass the two issues mentioned earlier in section 1.1 by first focusing the adaption on the leaves of the regression trees, as they are in general computationally easy to handle. The possible overfitting to the earlier predicted classes is then supposed to be avoided by gathering enough data from the target domain in conjunction with limiting the maximum altered leaf value.

Possible overfitting of the detector to the target environment is not seen as an issue in this thesis. The reason being that the adaption only concerns the leaf values and therefore it is assumed that the adapted cascade could easily be reversed by the user, e.g. when the detector is about to leave the target environment. However it is still very interesting to measure to which extent such an event occurs.

1.2.1 Research questions

- How is the performance affected in the target environment in comparison to the more general domain earlier trained for (including the training error)?
- Is the intended approach computationally feasible in real time?

1.3 Limitations

The main limitations in this thesis are primarily due to circumstances under which adaption is investigated. Limitations in this thesis concern the object recognition task, the device with camera used, the features used internally by the regression trees as well as the elusive environment to adapt for.

Detection Algorithms Adaption is investigated for four different pre-trained cascade classifiers developed for hand-gesture recognition. The gesture in question is chosen as the "open hand" gesture. These hand-detectors are trained with weak regression trees that are using multi-scale block local binary patterns as features (MB-LBP).

Target environment - Low light The target domain to adapt the classifiers to is a low light environment that has proven to contain elusive detections. The data-set representing the domain is gathered with the camera from a Nexus 9 device running in a constant frame rate of 30 frames per second, without consideration for camera calibration.

2 Theory

2.1 Computer Vision

2.1.1 RGB color space

A color model is a mathematical representation of the physical phenomena of colors. Different models have different advantages and drawbacks. One goal is to minimize formulation complexity and the number of variables or dimensions.

Historically, it has been found that three variables seem to be enough to describe all colors perceived by the human eye [43]

This observation is used as a backbone for many color models. The theory, also called the trichromatic theory, is old. Names like Isaac Newton, Thomas Young, John Dalton can be found associated with it and they tend to date as far back as the seventeenth or eighteenth century [43].

More recent experiments suggest that there are three known kinds of photo-receptors present in the human eye [1]. The types called cone-cells or cones are responsible for the mental perception of colors. Cones are activated when hit by light of different wavelengths. There are three different types of cones in the eye; the 'reddish', 'blueish' and 'greenish'.

Figure 1 shows an example of how these would be activated when hit by light of different wavelengths. If light of the wavelength 500 nm were to hit the eye, cones of all the types would be activated in different proportions, as suggested in figure 1. The differences (among other things) could then be used by the brain to further process and mentally represent the color. Figure 1 is just an abstract illustration, and is not really correct at the detailed level.

There are many problems and limitations with expressing colors with the trichromatic models.
One of the drawbacks when using a tri-chromatic model is that not every visual color can be represented. To represent a monochromatic light of the wavelength 580 nm, a mixture of 'reddish' and 'greenish' is stored in the model but no 'blueish'. When later displayed on the device/screen, the green light would trigger some of the blue cones in the eye making it impossible to replicate the stimulus from the light of wavelength 580 nm. This example can be found in [43].

Furthermore, the eye has special sensors that are activated in low light condition, so called rods that are completely saturated in the amount of light needed for color distinction.
Another interesting concept that further highlights the complexity of color models is a concept called γ as follows.

The human eye is more sensitive to relative variations in light than absolute variation. This is often used in the models to save space by storing less data-points in their color representation of the high light areas since they would anyway be more insignificant to the brain. This fact generates non-linear scaling artifacts in the color models when representing color with illuminations. Gamma is introduced as a remedy. Commonly it is utilized in color models by storing x^{γ}, $\gamma < 1$, instead of x when the detector/camera detects an illumination of x. By storing x^{γ} in the model, more data-points are sampled in the low intensity area. At the output device, colors are displayed at their original illuminations (x not x^{γ}). A common choice is to keep γ around one-half, i.e. 0.5 [2, 43].

2.1.2 Histogram equalization

The goal of histogram equalization is to produce a new image where the usable data has increased contrast. This is especially achieved when the usable data values are clustered in the color model. The result is often a new image with higher global contrast. This is done by spreading out the most frequently used intensities

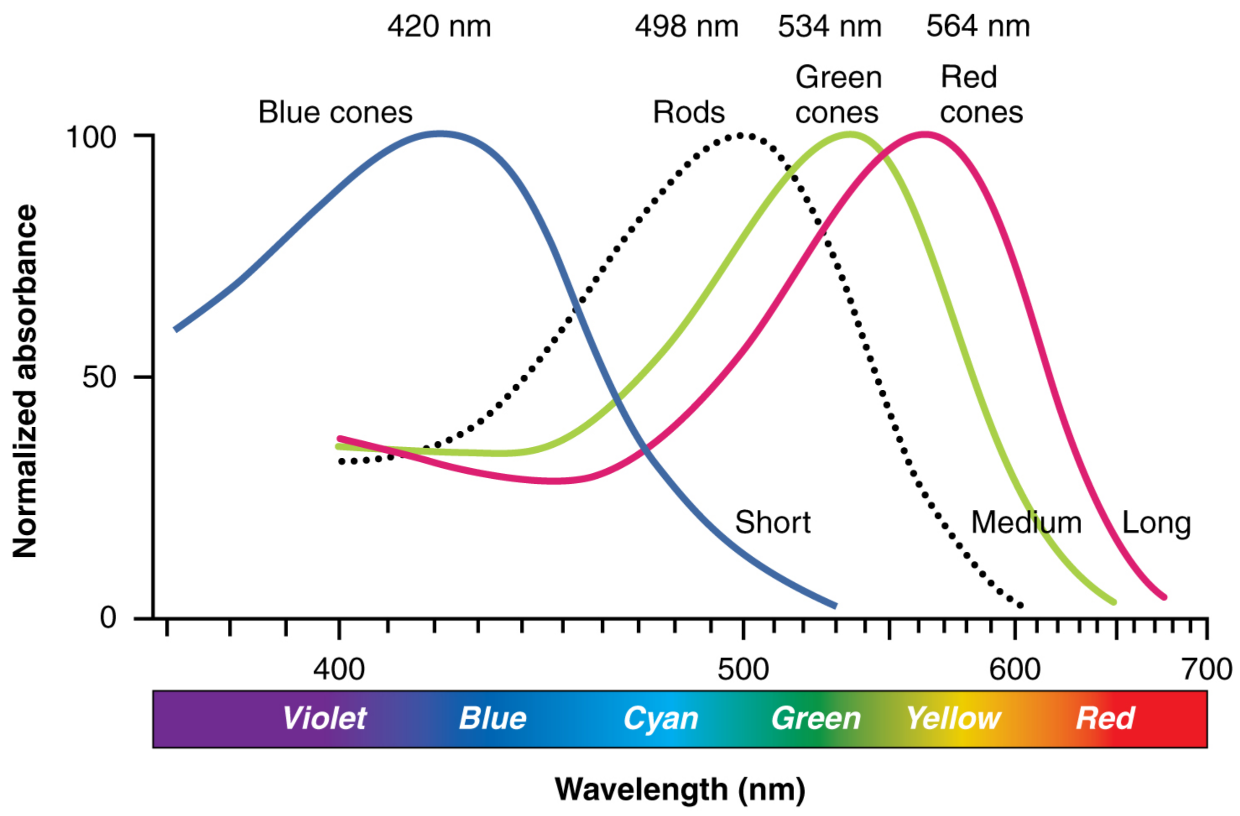

Figure 1: Illustrates the problem with the trichromatic models

in the image. Often, the result is best in images with background and foreground being both very bright or both very dark [3].

For further explanations of the concept of histogram equalization, the histogram of a single channel image is defined in definition 1 [4].

Definition 1. *For a single channel image X of dimension $M \times N$ and intensity values $x_{i,j} \in \{0, ..., L-1\}$, $(i,j) \in M \times N$,*
the ***intensity histogram*** *is the function that maps each possible pixel intensity to its relative frequency in the image X.*
In this thesis $p_X(k) \in [0,1]$, $k \in \{0, ..., L-1\}$ denotes the relative frequency.

Histogram equalization of a single channel image X with intensity/pixel-values $x_{i,j}$ is a transformation of the image X to a new image Y with a more equalized intensity histogram. According to [3, 4], it can be defined as in definition 2.

Definition 2. *For an image X with intensities $x_{i,j} \in \{0, ..., L-1\}$,*
and a (normalized) intensity histogram $p_X(k)$,
the ***equalized histogram image*** *Y is the*
image of the same dimension as X,
with intensities $y_{i,j} = floor(\sum_{k=0}^{x_{i,j}} p_X(k))$.

Here floor denotes rounding of the numbers downward to the nearest integer.

The histogram of the transformed image will effectively be more flat/equalized. See figure 2.

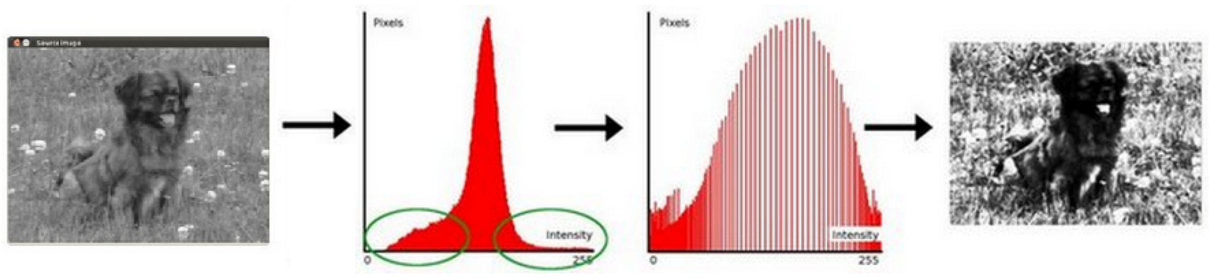

Figure 2: A sketch of how histogram equalization is done. See [5]

If an image is transformed a second time with histogram equalization the last image will not differ from the previous case. This is due to the fact that the histogram has already been equalized as much as it can [4].

2.1.3 Integral Image

The integral image, a concept introduced by Frank Crow in 1984 [6] in computer-graphics, is equivalent to the 'cumulative distribution functions' of multidimensional probability distribution functions in statistics.

In object detection applications of computer-vision/machine-learning, the integral image can be applied to implement the feature evaluator (the procedure that evaluates features) with dynamic programming for Haar and LBP-features.

The idea is to first compute the integral image of the whole image. Then use it as a look-up data structure to compute the pixel intensity sum within a block. It is used by OpenCV in both the case of Haar- and LBP features (see section 2.2).

Definition 3. *For an image X with intensities $x_{i,j}$, the integral image I_Σ is defined by its intensities;*

$$I_\Sigma(x,y) = \sum_{i\le x, j\le y} x_{i,j} \tag{1}$$

To compute the sum of pixel intensities within a block (or a rectangle) $r = ABDC$ in an image, (where A, B, D, C denote the corners of the rectangle r), the formula (2) is used, see figure 3. $I_\Sigma(ABDC)$ or $I_\Sigma(r)$ denotes the sum of pixel intensities and is sometimes referred to as the integral block of r,

$$I_\Sigma(ABDC) = I_\Sigma(A) - I_\Sigma(B) - I_\Sigma(C) + I_\Sigma(D). \tag{2}$$

2.2 Features

At some point, it is essential for a classifier to be able to extract measurable information from the target data in order to beat random guessing. An individual instance of a measurement is referred to as a feature. There are many different types of features suitable for image object recognition. The procedure to evaluate or find a feature is called feature extraction. The feature is then represented in some model. An instance of a feature representation is often called feature descriptor or just descriptor [7].

Example An image feature detection could be the procedure of finding a corner in the image by applying some algorithm. Carving out an area surrounding the corner, computing gradients and placing them in a histogram is the feature extraction. The feature is now represented in the histogram. The histogram of

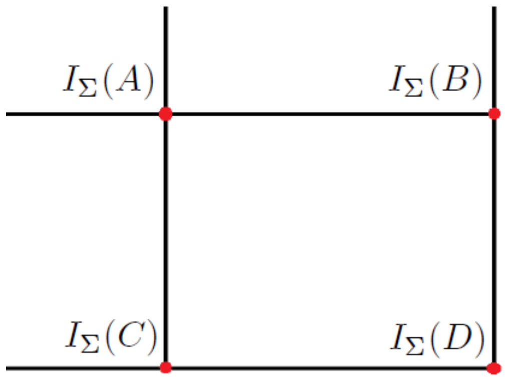

Figure 3: A sketch of an integral image and the four points used to calculate the integral block of $ABDC$
.

gradients is the feature descriptor.

The features mentioned in this report are the standard ones that are available in the part of OpenCV concerning the task of object recognition.

Histogram of oriented gradients (HoG)
A scale invariant feature that counts occurrences of oriented gradients within a region of the image. Especially useful for human detection [8].

Haar-features
A feature developed by Viola & Jones, based on the Haar-wavelet idea, especially used for object recognition on devices with limited resources due to its relatively cost efficient computational scheme [9].

LBP-features
Local binary patterns, originally developed for texture description and first described by T. Ojala, M. Pietikäinen and D. Harwood [35]. Suitable for object recognition on devices and, in general, less computationally heavy than the Haar features [10, 31]. LBP features are invariant under monotonic gray-scale changes [35].

2.2.1 Haar

In many cases the Haar features can be described as the sum of weighted integral-blocks extracted from the target image with equation (2). These blocks are placed relative to the resolution of the detector [49].

Given some rectangular blocks $\{r_1, r_2, ..., r_n\}$ and corresponding weights,
$\{\omega_1, \omega_2, ..., \omega_n\}$, the feature output would be the sum of the products, see equation (3) [38, 49],

$$\sum_i \omega_i I_\Sigma(r_i) \tag{3}$$

Further generalizations are proposed by some, as the 45 degree rotated Haar-features [38] or ordered decision trees of Haar-features. [41].

Example The proposed Haar-features seen in in figure 4 could be defined by the procedure of summing the black pixel values and removing the white.

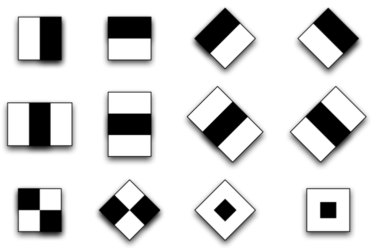

Figure 4: Some different Haar-features. The black areas are weighted with $\omega = 1$ and the light areas with $\omega = -1$. See [11]

2.2.2 LBP

LBP-features or local binary patterns, as mentioned before, are features originally developed for texture recognition, but later proven suitable for a broader range of object detection applications such as face detection. The idea is to extract information from the image by simply comparing pixel values surrounding a single target pixel in a specific manner.

For a pixel-intensity $x_{i,j}$ in a single channel image X, an LBP feature-descriptor can be computed at $x_{i,j}$ by comparing the numerical value of $x_{i,j}$ with intensities of predefined, surrounding pixels. The procedure is performed in a specific pattern (usually/always clockwise or anti-clockwise), see figure 5 [31].

The feature can also be used in conjunction with the integral image by comparing integral blocks (equation 2) of the same sizes instead of single pixel values.
The feature is partly defined by the location and size of a target pixel/block and its analogous surrounding.

Example One (possibly the original) LBP-feature-descriptor for image recognition can be constructed and computed for a pixel-intensity $x(i,j)$ by comparing each of the eight surrounding pixels intensities $x_{(i+k,j+l)}$, where $k,l \in \{-1,0,1\}$ (not both zero), with the one at the middle, $x(i,j)$. For each comparison either a zero for "the surrounding pixel has lower intensity" or a one, for the "opposite statement" is stored [10, 31, 35].

Eight comparisons are thus made and the result is stored in an ordered vector of size 8, represented by a byte. The order is defined by the image topology, going clock-wise starting from the top pixel, see figure 5. This idea can be further generalized by applying the same rule for a different set of surrounding pixels $p_{(i+k,j+l)}$, defined for other choices of values of k,l.

The notation LBP$_{(n,r)}$ can be used to express that the surrounding pixels lie equally "spaced" in a circle of "radius" r and their number is n.
Here the radius r is defined by the metric in $d(k,l) = \sup(k,l)$,
For example the feature LBP$_{(16,2)}$ is shown in figure (6).

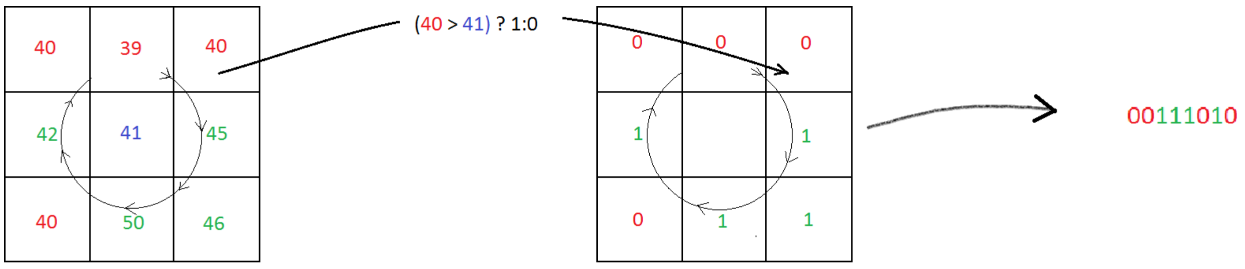

Figure 5: Shows a sketch of the computation of a single (three by three) LBP feature value.

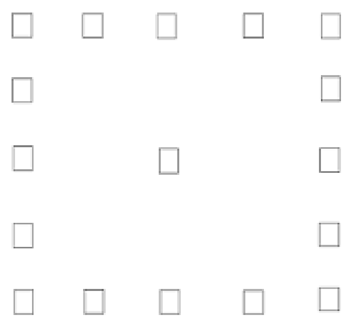

Figure 6: The set of surrounding pixels for the LBP feature $\text{LPB}_{16,2}$, each pixel with the distance of $\sup(l, k) = 2$ from middle

Local Binary Patterns in OpenCV Local binary pattern features are available in OpenCV:s implementation of a cascade detector. These features are defined on single channeled image patches X with fixed sizes, $M \times N$ (usually carved out and/or downscaled from the original camera feed), accompanied by a point $P = (x, y) \in X$, a block size (w, h) and a bipartition of the same size as the number of possible outcomes.

Introducing a coordinate system for X with *origin* placed in the top left corner, having the first axis (abscissa) point to the right and second (ordinate) downwards, nine blocks of size (w, h) are placed in a three by three grid. The position of the grid is defined by its top left corner at point P. Every small block is given its integral value. Now the three by three grid can be interpreted as an ordinary $\text{LBP}_{(8,1)}$ feature. The bit field is then extracted in a similar manner as in (5) by setting the start count at the top left block moving clockwise.

To assign a feature value to the X, the bit field is categorically placed into one of two different possible subsets from the bipartition and the class is outputted. The subsets make up a bipartition of the 256 different classes that the bit-field could belong to and they are referred to as either **left** or **right** class.

OpenCV:s detection algorithm then constructs weak classifiers by assigning numerical values (and sometime also other features) to classes.

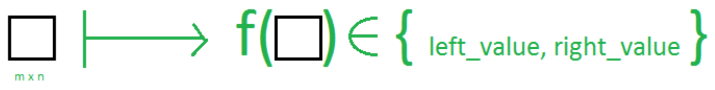

Figure 7: A sketch of a bit field extraction on a $m \times n$ image patch.

LBP Weak-trees in OpenCV OpenCV offers the use of LBP regression-trees as weak classifiers also called weak trees. The weak-tree consists of LBP features having other LBP features as right and left nodes and numerical values as leaves. This construction will generate a binary tree. As an image patch X is assigned to a new node, the feature at the new node evaluates X and sends it further down until X hits a leaf. The numerical value at the leaf is outputted.

An image patch will choose exactly one leaf for every weak classifier.

2.3 Classifiers

In machine-learning, one often refers to the procedure of mapping observed instances to predicted classes as a classification model or classifier. The learning process for a classifier needs data with known true class-

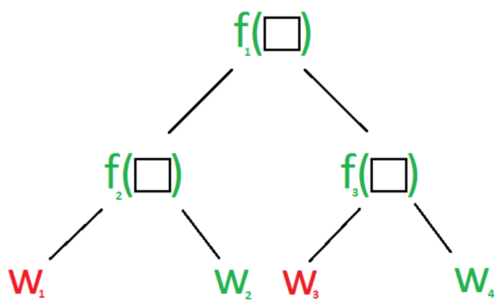

Figure 8: A sketch of how a regression tree of LBP features works in OpenCV.

belongings, also called training data. Models that do statistical classification without training data are rather known as clustering models [32, 12]. In the following chapter only supervised machine-learning models are considered.

A model is instantiated when it is trained on data. Its error on the training data is commonly called the training data error or simply the training error. However, when used on new observations or more general data it is possible that the previously instantiated classifier behaves differently. The error on new observation/non-training data is called the generalization error.

Overfitting If a classification model is tuned in a way that the training error is reduced at the expense of higher generalization error, it is referred to as over-fitting. The model becomes biased on its training data. A classification model that has a lower training bias often requires more training data to converge.

According to Edwin Chen [13], classifiers that have high bias (low variance) often have advantages over the low bias (high variance) classifiers when the training data set is smaller. However, when the training data set grows in size, the low bias classifiers usually start to out-preform the biased ones as they tend to overfit and are in general not powerful enough [13].

Probabilistic Classifiers A classifier that outputs ordered quantities that measure how much a model believes an instance belongs to a certain class is called a probabilistic classifier. Classifiers that output estimated probabilities (such as Naive Bayes') are probabilistic classifiers however the converse statement is not true. Probabilistic classifiers are sometimes referred to as ranking classifiers [32].

Furthermore, any probabilistic classifier can be discretized into a classification model by simply applying a threshold. Probabilistic classifiers are especially suitable for ROC-analysis [32].
In computer-vision/ machine-learning some common types of classification models are:

Naive-Bayesian Classifiers
: Probabilistic classifiers using Bayes' theorem to model probabilities. Trained by estimations of conditional probabilities under the strong assumption that each feature is independent of any other. Naive Bayesian classifiers do not require much training time or data to converge [14].

Nearest Neighbour classification
: Classification model provided with some metric (or distance function) defined on the data. Uses the distances between the new observations and the training data to classify [15].

Decision Trees
: Model that uses features or other classifiers arranged in tree structures to divide data into different classes [16]. Decision trees are used in other ensemble methods such as random forest or boosting [34, 41].

Support Vector Machines

A binary classification model that is trained by finding a suitable separating hyperplane in the feature space that divides the training data [17]. Usually the models are extended to handle non-separable and/or noisy data. SVM:s have nice theoretical properties (for instance the training-optimization problem is convex), but usually because of memory and other performance aspects, a multitude of optimizations and other ad-hoc approaches are needed in order for them to be competitive [29].

Boosting

An ensemble method (commonly iterative) that constructs a single classifier by combining a subset of existing ones (usually in a linear combination). For each round of boosting the classifier that minimizes the so called weighted error is added and the weights are then updated before the subsequent round. For further information, see section 2.6.

2.3.1 Naive-Bayesian Classifiers

Naive-Bayesian classification models are models that use Bayes theorem (4) on the feature data to predict different class labels.

The model works by calculating the different class (labels) probabilities given a feature vector with the assumption that all features are independent of each other.

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}. \tag{4}$$

Given l different classes and a feature space of size n the model estimates all l posterior probabilities $P(C_k|\hat{x})$. $P(C_k|\hat{x})$ is the probability that an observation $\hat{x} = (x_1, x_2, ..., x_n)$ belongs to a certain class $C_k \in \{C_1, ..., C_l\}$.

The independence assumption will then generate a way, see equation (5), to simply evaluate the posterior probability $P(C_k|\hat{x})$ by only knowing all the 'feature class probabilities' $P(x_i|C_k)$ and by using Bayes theorem (4) repeatedly.

When all the posterior probabilities are estimated, they are used to make a decision (a simple rule is to output the largest)

$$P(C_j|\hat{x}) \propto P(C_j) \prod_{i}^{n} P(x_i|C_j). \tag{5}$$

The training of the model is performed by estimating the $P(x_i|C_k)$ on the training data. If the data is entirely categorical, a frequency table is sufficient. With continuous feature data, it is assumed they come from some known distribution, such as normal, lognormal, poisson or gamma, etc., [44, 18, 14].

Advantages of Bayesian classification models are the simplicity of both the implementational and theoretical aspects. The training converges quite quickly (and even quicker if the feature-independence assumption holds) with relatively little training data. The model is also robust against irrelevant features and has a linear time-complexity for both evaluation (feature space size) and training (training set size).

A disadvantage of the model is its independence assumption, as the assumption also implies that the model will not treat dependencies between features correctly. It also tends to be more of a high bias model [13, 19].

2.3.2 Nearest Neighbour

Nearest neighbour classifier will classify a new observation $\hat{x}$ by measuring its distances from the training data of size n in the feature space (the space of possible feature vectors). The simplest way is of course to consider the closest neighbours category and place the new observation into that same category.

More complex classifiers can also be built by considering more than one neighbour. One example is the k-nearest neighbour classifier that classifies a new observation by letting the $k-$closest neighbours each vote with the same weight $w_i = \frac{1}{k}$. The concept can easily be generalized by letting the $i-th$ nearest neighbour from the training set have the weight w_i and letting the sum of all weights be equal to one, $\sum_i^n w_i = 1$.

One advantage of nearest neighbour classifiers is the simplicity of both their underlying concept and implementation.

A disadvantage of the nearest neighbor method is that the model needs to be provided with a norm or metric to measure distances. It is not always obvious how to introduce a simple metric for all features, for example if the data is partly or entirely categorical [15]. Nearest neighbor classifiers also tend to be more of the high bias type [13].

2.3.3 Classification and Regression Trees

A discrete classifier in the form of a decision tree is called a classification tree. Classification trees classify new observations by letting the data flow through a tree of different sub-classifiers. The sub-classifiers decide where to send the data next down the tree by classification. In the binary case the tree outputs either "yes" or "no". In the more general case, the outcome of the classification is a class-label, no more than one for each leaf. A similar type of classifier is the regression-tree. The main difference between regression and classification trees is that the regression tree outputs a continuous measure instead of labels [16].

There are many different algorithms to construct these classifications trees using the training data.

Many of the learning algorithms use the property of entropy of a set, as a crucial part in their construction.

For a set of training data, the entropy H can be used to determine a "quality quantity" of a test.
Advantages of decision trees are their simplicity to interpret and their natural ability to handle feature interactions [13].

Ensemble Methods with Trees A common drawback with the decision tree approach is that the models have bad habits to overfit the classifiers to the training data.

The random forest classifier is an ensemble method which utilizes many decision trees (forest) while classifying data, where each tree 'votes' on a class label. Another type of ensemble approach is to use boosted trees. Both random forest and boosted trees tend to be more robust (low bias) against generalization errors [20, 21, 13].

2.3.4 Support Vector Machines

The earliest Support Vector Machines (also called linear classifiers) were classification models that classify n-dimensional feature-data by separating them with a $n-1$ dimensional hyperplane.

If such a separating hyperplane on the training-data exists the training set is said to be linear-separable. Furthermore if the training set is linear-separable there must exist infinitely many distinct hyperplanes. A quadratic convex optimization problem (P) can be formulated to find the hyperplane that maximizes the margin (the minimal distance between the hyperplane and one of the classes). A dual formulation of the problem (P), $(\hat{P})$, is often/always preferred over the original, since it tends to be easier to solve ($(\hat{P})$ constraints are nicer than the ones from (P)). There are many extensions to this classification model, two well known examples are the Kernel- and the Soft Margin-approach [22, 29]. The dual problem is (by using

Lagrangian duality theory) often preferred over the original, since it tends to be easier to solve [30].

Kernel Approach As the linear classifier uses separating hyperplanes described with the n-dimensional dot product, separation cannot always be done. By exchanging the dot product in the feature space with an higher dimensional one, represented as a kernel function, any set of data that is not linear separable can be implicitly projected into the higher (even infinitely) dimensional space where it is separable. In the original feature-space, this can be seen as a nonlinear decision boundary is separating the data. Any continuous decision boundary can be implemented by using the kernel trick. There is no need to compute the feature vectors if the kernel function can be computed, thus avoiding some curse of dimension [29].

Soft Margins Although the kernel approach always offers a way to separate the training data, using it is not always a good idea when for example the data is noisy. The soft margin approach is all about penalizing the non separating errors by introducing more slack variables into the optimization problem thereby further increasing the complexity [29, 22].

2.4 Performance measures for discrete classifiers

2.4.1 Basic measures

Discrete classifiers map observed instances to predicted classes. For the binary case each instance is either positive $\mathbf{p}$ or negative $\mathbf{n}$. To not confuse the actual classes with the predicted ones the notation $\mathbf{Y}$ (positive) and $\mathbf{N}$ (negative) is here used to describe the labels outputted from a classification model.

When a discrete classifier is used to predict the class of an instance, there are four different outcomes.
A positive instance that is correctly classified as a positive is called a **true positive** ($\mathbf{p}$ predicted as $\mathbf{Y}$).
A negative instance that is correctly classified as a negative is called a **true negative** ($\mathbf{n}$ predicted as $\mathbf{N}$).
A positive instance that is incorrectly classified as a negative is called a **false negative** ($\mathbf{p}$ predicted as $\mathbf{N}$).
A negative instance that is incorrectly classified as a positive is called a **false positive** ($\mathbf{n}$ predicted as $\mathbf{Y}$).

These outcomes are usually visualized in a confusion matrix, see figure 9.
Let $\mathcal{S} \subset X \times Y$ be a test-set with data from X and true belongings $Y = \{\mathbf{p}, \mathbf{n}\}$. By applying a discrete classifier H,

$$H : X \to \{\mathbf{p}, \mathbf{n}\},$$

on the test set $\mathcal{S}$ the number of occurrences in each class within the confusion matrix can be counted. Each quantity from the confusion matrix is typically denoted by the first letter in each word of its class labels, TP for **true positives** etc. Note the notation used for the other quantities (FP, TN, FN).
Usually N and P are used to denote the number of negative and positive instances from some test set i.e. the column sums from the confusion matrix. That is, by using earlier notations,

$$\begin{cases} P \triangleq |\mathbf{p}| = TP + FN \\ N \triangleq |\mathbf{n}| = FP + TN. \end{cases} \tag{6}$$

However the number instances in each predicted/hypothesized class ($|\mathbf{Y}|$ and $\mathbf{N}|$) are more commonly denoted by their expressions for the corresponding row sums in the confusion matrix.

$$\begin{cases} |\mathbf{Y}| = TP + FP \\ |\mathbf{N}| = FN + TN. \end{cases}$$

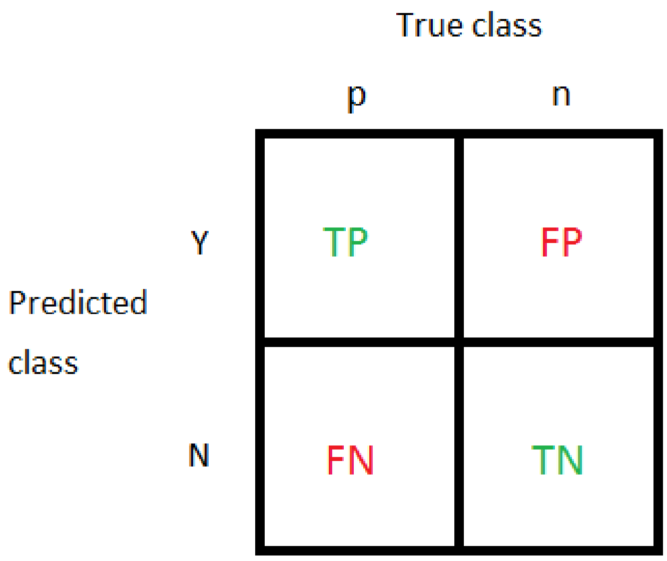

Figure 9: The confusion matrix. Note that the correct decisions reside in the diagonal.

2.4.2 Common performance metrics

From the basic performance measures defined in the previous section, some common performance metrics can be derived.

True Positive rate:
Denoted d, the **True Positive rate** measures the proportion of positives that are correctly classified and can be estimated with,

$$\text{TP rate} = \frac{TP}{P}. \tag{7}$$

True Positive rate is sometimes referred to as "hit-rate", "recall", "detection rate" or "sensitivity" [32, 49].

False Positive rate:
Denoted f, the **False Positive rate** measures the proportion of negatives that is incorrectly classified and can be estimated with

$$\text{FP rate} = \frac{FP}{N}. \tag{8}$$

Another name for the **False Positive rate** is "false alarm rate" [32, 49].

Positive rate:
Denoted p, the **positive rate** measures the amount of data that is classified as positive and can be estimated with,

$$\text{Positive rate} = \frac{TP + FP}{P + N}. \tag{9}$$

Precision:
Measures the amount of detections that are correct. Can be estimated with,

$$\text{Precision} = \frac{TP}{TP + FP}. \tag{10}$$

Accuracy:

Measures the amount of predictions that are correct:

$$\text{Accuracy} = \frac{TP + TN}{N + P}. \tag{11}$$

Combined metrics:

Metrics that combine other metrics into a single "score" function. One example is the F_β score:

$$F_\beta = (1 + \beta^2)\frac{\text{precision} \cdot \text{recall}}{(\beta^2 \cdot \text{precision}) + \text{recall}}.$$

Note that $F_0 = \text{precision}$ and "$F_\infty = \text{recall}$".

The **positive** and **negative rates** are measures insensitive to changes in the proportions of negative and positive test-data. This can sometime be an advantage. However, there are situations where the class-dependent measures such as **precision** and **accuracy** give a better picture of the general performance. This is especially the case in applications with large class skewing that change with time (or some other variable) [32]

2.4.3 Receiver operating characteristic

Receiver operating characteristic is a graphical tool commonly used to visualize and organize classifiers based on their performance. The ROC curve depicts classifier's trade off between **false positive rate** and **true positive rate** for different settings. For applications where an underlying ranking classifier is used, its ROC curve can be generated efficiently (after evaluating all test data) in linear time in terms of data size.

The ROC space is the two dimensional space with *FP* rates on the abscissa and *TP* rates on the ordinate. Each confusion matrix can be mapped to a point in the ROC space by calculating the positive and false positive rates. Thus each discrete classifier corresponds to a point in the ROC space.

The ROC points of six different discrete classifiers (labeled 1 to 6) are shown in figure 10. Classifiers residing in the top left corner (classifier 1) have high *TP* rates in conjunction with low *FP* rates. They are the ultimate classifiers. Classifier 2 is obviously inferior to 1 as it has a lower *TP* rate and higher *FP* rate. The anti diagonal symbolizes the random guessing, hence the worst classifiers, as number 4, reside on it. As any classifier below the random guessing line can be inverted (by always doing the opposite of what it suggests) the area is in general not interesting. Thus classifier 6 is considered equivalent to (and just as good as) 1. The left bottom area symbolizes the region of conservative classifiers, they do not sound alarm unless very sure but also tend to miss more positive instances. Conservative classifiers are commonly used in applications where a false alarm also has a high cost. The top right area is the liberal region. Liberal classifiers act on the "weaker signals". These are more suited for applications having a high cost for missing a positive, such as surveillance or medical decision making [32].

ROC-curves for probabilistic classifiers The ROC curve serves as one of the standard methods for performance measuring of classifiers.

Many discrete classifiers, such as the ones constructed with boosting methods used within the Viola-Jones framework, have underlying probabilistic classifiers that are discretized by a threshold, see section 2.3. If the threshold θ is set high enough (theoretically at $\theta = \infty$) the discretized classifier becomes super conservative, never classifying anything as a positive, hence yielding the ROC-point $(0, 0)$. At $\theta = -\infty$ the classifier is super liberal, believing all instances are positive which corresponds to the point $(1, 1)$ in the ROC-space.

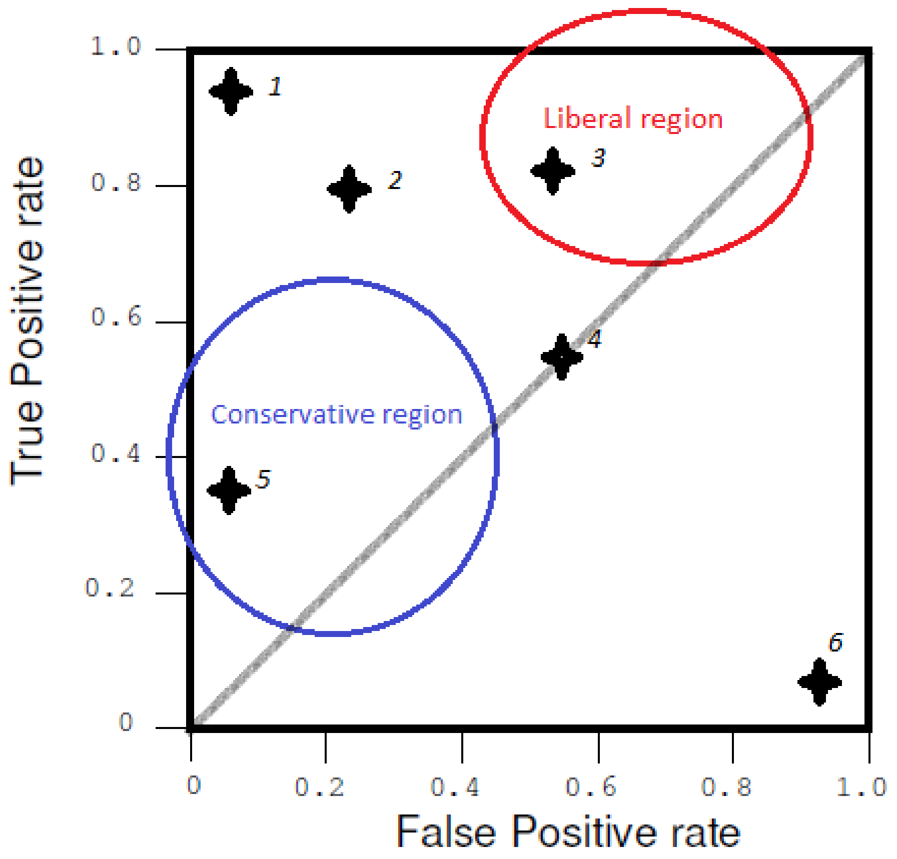

Figure 10: Six (discrete) classifiers in the ROC-spaec.

In practice two thresholds that generate identical confusion matrices are equivalent (within a domain), as they produce the same discrete classifier. For any ranking classifier, sliding the threshold from ∞ to $-\infty$, different possible discrete classifiers can be visualized as different points in the ROC space. The ROC-curve can then be estimated (unbiased) with linearly interpolating between consecutive points (parametrized by θ).

The procedure above can be done relatively fast for a ranking classifier, i.e. by sorting classified test data after score followed by a linear sweep over them. For further reading see [32].

2.5 Learning as function approximation

The task of (machine) learning can be described as to output a hypothesis H (classifier) that maps observations from an instance space $\mathbf{X}$ to an output space $\mathbf{Y}$, usually labels or some other continuous measure in $\mathbb{R}$,

$$H : \mathbf{X} \ni \mathbf{x} \mapsto H(\mathbf{x}) \in \mathbf{Y}.$$

For the sake of simplicity, the following description only considers discrete classifiers of the binary case, i.e the target domain is $\mathbf{Y} = \{-1, 1\}$. Furthermore, one can easily generalize the binary case by applying the principle of ”one against all”.

The general (real world) domain of the classifier can be described with a generator of random vectors with an underlying unknown probability distribution $P(\mathbf{x}, y)$. With this formalization, all possible observations are generated from $P(\mathbf{x}, y)$. The **generalization error** (or **risk**) of the classifier H, $R(H)$ is given by risk functional:

$$R(H) = \int_{\mathbf{x},y} L(y, H(\mathbf{x}))dP(\mathbf{x}, y). \tag{12}$$

Here L denotes the function that measures the loss (or discrepancy) which further depends on the context (some errors may be worse than others, hence more important to avoid). Commonly the loss function L is set to the indicator function of the event of a miss-classification. I.e. L is producing 1 at $\mathbf{x}$ if H misclassified $\mathbf{x}$ and zero otherwise,

$$L(y, H(\mathbf{x})) = \mathbb{I}(H(\mathbf{x}) \neq y). \tag{13}$$

This is also called the 0 - 1 loss function. In (13) $\mathbb{I}$ denotes the **indicator function** that is producing 1 if the argument (event inside) occurs and zero otherwise, i.e. for an **event** A,

$$\mathbb{I}(A) = \begin{cases} 1 & \text{if } A \text{ occurs} \\ 0 & \text{otherwise.} \end{cases} \tag{14}$$

The goal (in machine learning) is naturally to find a hypothesis $H^\star$ that minimizes R over a set of available functions $\mathbf{F}$. Since the P is unknown the **risk** cannot be evaluated, and therefore the optimal hypothesis can only be estimated with training data.

A set of training data $\mathcal{S}$ is a set of data points in the feature space X accompanied with its true belongings, drawn independently at random from P,

$$\mathcal{S} = \{(x_1, y_1), ..., (x_n, y_n)\} \subset X \times Y. \tag{15}$$

The **training error** of a hypothesis H is given by the **empirical risk** E of its training set,

$$E(H) = \frac{1}{n}\sum_{i=1}^{n} L(y_i, H(\mathbf{x}_i)). \tag{16}$$

The principle of replacing the **risk** with the **empirical risk** and furthermore assuming that the minimum to E over $\mathbf{F}$, $\hat{H}$, results in a close to minimum **risk** is called the ”induction principle of ERM” (Empirical risk minimization). According to [48] a necessary and sufficient condition for the principle to be consistent (over the set of functions $\mathbf{F}$) is uniform convergence in probability of R to E (over $\mathbf{F}$). By consistent is meant that the infimum of E over $\mathbf{F}$ converges in probability to the infimum value of R (over $\mathbf{F}$) [48].

If the problem is not consistent a hypothesis H which otherwise achieves good empirical performance may lead to large generalization errors. The definitions used to state the above depend heavily on the set of available functions $\mathbf{F}$. Therefore the function class is a crucial parameter in ERM learning. Insufficient training data sizes and/or very complex function classes $\mathbf{F}$ can result in a classifier being able to fit the training data extremely well without being able to generalize. By restricting the size of $\mathbf{F}$, the earlier mentioned overfitting can possibly be avoided. The approach of limiting the complexity of $\mathbf{F}$ is called regularization [42].

2.5.1 VC-dimension

The standard metric, introduced by Vapnik and Chervonenkis, used to measure the complexity of a set of hypotheses is the **VC-dimension**. First, we need to define what is meant by a set of hypotheses F being able to **shatter** a set of data points $\mathbf{X}$.

Definition 4. *A class of hypotheses* $\mathbf{F}$ *is said to* ***shatter*** *the set of points* $\mathbf{X} = \{\mathbf{x}_1, ..., \mathbf{x}_n\}$ *if* $\forall$ *labelings* $\{y_1, ..., y_n\}$, $\exists$ *an* $f \in \mathbf{F}$ *s.t.* $\forall$ i, $f(\mathbf{x}_i) = y_i$.

Definition 5. *For a class of hypotheses* $\mathbf{F}$*, the cardinality of a set with the largest cardinality that* $\mathbf{F}$ *shatters is called the* ***VC-dimension*** *of* $\mathbf{F}$.

2.6 Boosting

In machine learning boosting is an ensemble method to construct classification models. The practical boosting models can be viewed as proposal solutions to the **Hypothesis Boosting Problem** that, according to Michael Kearns, informally asks "whether an efficient learning algorithm that outputs a hypothesis whose performance is only slightly better than random guessing implies the existence of an efficient algorithm that outputs an hypothesis of arbitrary accuracy" [37].

More specifically the problem asked whether **weak learner** (the algorithm that outputs hypothesis) implies the existence of a **strong PAC learning algorithm**. Here **PAC** is an abbreviation for **Probably Approximately Correct** [42].

In 1996 Robert Schapire and Yoav Freund invented the first practical boosting algorithm, AdaBoost (or Adaptive Boosting). They both received the Gödel Prize for their work on AdaBoost in 2003 [46, 23]. Since then, other practical boosting algorithms have been proposed. Usually their main variations lie within their method of weighting training data and their hypotheses [24].

Since many boosting algorithms make use of similar framework as AdaBoost, basic understanding of the AdaBoost algorithm will do a lot for the understanding of a large number of boosting algorithms.

2.6.1 General Properties

AdaBoost and many of its sibling boosting algorithms share similar properties.

Weak Learning Condition
Boosting algorithm usually uses the assumption that the weak learning condition is fulfilled, see section 2.6.3. Furthermore, an assumption that the set of weak classifiers have sufficient richness is often used.

Adaptive Learners
Before AdaBoost was invented, other AdaBoost resembling solutions were proposed [47], however they were not adaptive. Adaptive boosting algorithms will adapt to the error rates of each weak classifier, which will let the user utilize more of each hypothesis added (training will therefore be done in an iterative procedure). Adaptivity was one of the main reasons AdaBoost was so successful. However, in modern times, non-adaptive boosting algorithms have been proposed to combat, among other things, noise (such as brown-boost) [46].

Training and Generalization error
Many boosting algorithms (such as AdaBoost) have great resilience against generalization errors. Different approaches used to explain boosting yield upper bounds on the training and generalization error, see [46, 33].

Minimizing a loss function
Many boosting algorithms can be shown to minimize a suitable cost functional on the linear hull of the space of weak-hypotheses. This approach to explain boosting helps with the understanding of convergence and the goal of the algorithm, see [46].

Relation to SVM:s
Using the margin approach to explain AdaBoost an obvious connection between SVM:s and boosting emerges as they both solve a similar optimization problem. SVM:s solve a quadratic programming problem and boosting a linear programming problem. SVM:s and boosting both operate in very high

dimensional spaces but use totally different methods to tackle the curse of dimension. For instance, compare SVM:s use of the kernel trick with the greedy search approach from AdaBoost [33].

Susceptible to Noise

A boosting algorithm that is adaptive also tends to be extra susceptible to noise and/or bad training data [39, 46]. Regularization of AdaBoost-similar boosting procedures seems to be mandatory in practice to avoid overfitting [42].

2.6.2 Strong PAC learners

Strong PAC learners can approximately be defined for the binary case as:

Given a set of n data points, $\{(\mathbf{x}_1, y_1), ..., (\mathbf{x}_n, y_n)\}$ that are generated independent and at random from a distribution $P(\mathbf{x})$, with their true belongings y_i given by a supervisor (binary function) $c(\mathbf{x}) = y$ from a known class of concepts, $\mathbf{C}$, a **strong PAC learner** is an algorithm that for every distribution P, every $c \in \mathbf{C}$, and every $\epsilon \geq 0$, $\delta \geq \frac{1}{2}$ with probability $1 - \delta$ returns a hypothesis h s.t. $\Pr[h(\mathbf{x}) \neq c(\mathbf{x})]$. The algorithm should be polynomial in $\frac{1}{\epsilon}, \frac{1}{\delta}, n$ and the dimension of the input space [42].

A concept class is said to be **strong PAC learnable** if a **strong PAC learning algorithm** exists for it. A concept is used to define the learning task, i.e. a concept is a problem that we want to teach the machine to solve.

2.6.3 Weak learner

Definition 6. *Let $\mathcal{S} = \{(x_1, y_1), ..., (x_n, y_n)\}$ be a set of data and $D(i) = w_i, i \in \{1, ..., n\}$ a probability weighting of $\mathcal{S}$. For a hypothesis h the **weighted error** on $\mathcal{S}$ with respect to D is the sum of all weights w_i associated with misclassified data. In this thesis $e(h)$ denotes the **weighted error** of h.*

Informally (and sufficient within the scope of this thesis) one can state that a **learner** that for some $\gamma > 0$, for every weighting D on $\mathcal{S}$, is able to efficiently find a hypothesis that can achieve a **weighted error** less than $\frac{1}{2} - \gamma$, is a **weak learner** on $\mathcal{S}$. Analogously a concept class is weak learnable if this condition holds for every concept in it [42, 47].

Each weak classifier (hypothesis) h returned by the **weak learner** is guaranteed to have a **weighted error** $e(h) < \frac{1}{2} - \gamma$ where $\gamma > 0$. γ is commonly referred to as the **edge** over random guessing. Especially the weak classifiers have to perform better than the strategy of always guessing on the class having the largest proportions of the weights [42].

While studying boosting algorithms, the assumption that the **learner** is a **weak learner** is often referred to as **Weak Learining Condition** [46].

In 1987 a paper [47] by Robert E. Schapire proved equivalence between a concept class being weak and strong learnable (naturally every **strong learner** is also a **weak learner**, Schapire proved the reverse). The goal with Boosting algorithms (such as AdaBoost) is to turn **weak learners** into **strong (PAC) learners**.

2.6.4 AdaBoost

AdaBoost was the first practical realization of how to turn a **weak learner** into a **strong PAC learner**.

Given a set of training data $\mathcal{S}$ of size N, and a **weak learner**, $\mathcal{L}$, that outputs hypothesis $h : \mathbf{X} \rightarrow \mathbf{Y}$, **AdaBoost** iteratively uses the weak learner to find binary hypotheses and combining them into a linear combination with coefficients from $\mathbb{R}$, thus producing a soft classifier (ranking or probabilistic classifier, see section 2.3). After T number of rounds the resulting soft classifier will be of the form,

$$S_T = \sum_{t=1}^{T} \alpha_t h_t \tag{17}$$

where h_t is a weak classifier chosen at round t and $\alpha_t \in \mathbb{R}$ its coefficient. The final hypothesis is then retrieved by thresholding the soft classifier S_T with $\theta = 0$ in case of $\mathbf{Y} = \{-1, 1\}$ being used as target labels (sometimes the labels are represented with $\mathbf{Y} = \{0, 1\}$ which means that $\theta = \frac{\sum_{t=1}^{T} \alpha_t}{2}$ is used instead).

While the description above is quite general what is particular with AdaBoost is its way of (re)weighting data and selecting hypotheses. See algorithm 1.

AdaBoost Algorithm

The AdaBoost algorithm originally proposed by Yoav Freund and Robert E. Schapire [33], described with $\mathbf{Y} = \{-1, 1\}$ is as follows.

Algorithm 1 AdaBoost, sources [45, 46, 33, 25, 26]

```
1: initialize $S_0 = 0$, $D^1(i) = \frac{1}{N}$
2: for $t = 1, ..., T$ do
3:     $h_t \approx \underset{h}{\operatorname{argmin}} \, [e_t(h) = \sum_{h(x_i) \neq y_i} w_i]$.
4:     define the coefficient $\alpha_t = \frac{1}{2} \ln(\frac{1 - e_t}{e_t})$.
5:     update the weights $w_i^{t+1} = \frac{w_i^t e^{-\alpha_t y_i h_t(x_i)}}{Z_t}$.
6:     $S_t = \alpha_t h_t + S_{t-1}$
7: return $S_T$
```

The user selects the number of rounds to boost T. Starting from $S_0 = 0$ and a rectangular error weighting, $D^1(i) = \frac{1}{N}$ which corresponds to all training data being equally important to classify correctly, the weak learner finds at each round t, a hypothesis h_t with a small weighted error,

$$e_t(h_t) := \sum_{i \in \{i | h_t(x_i) \neq y_i\}} D^t(i). \tag{18}$$

The selected hypothesis h_t is then added on to the previous result

$$S_t = \alpha_t h_t + S_{t-1}, \tag{19}$$

with the coefficient

$$\alpha_t = \frac{1}{2} \ln \left(\frac{1 - e_t}{e_t} \right). \tag{20}$$

Note that if the weak learning condition holds, $\frac{1}{2} \ln \left(\frac{1 - e_t}{e_t} \right)$ is an injective mapping of e_t to $(0, \infty)$ ($\alpha_t = 0$ if and only if $e_t = \frac{1}{2}$), with no theoretical upper limit for a good enough classifier.

In preparation for the next weak classifier to be selected, the last weak classifier h_t is used to reweight the **weighted error distribution** on the training data. The reweighting is done such that the errors of the previously selected hypothesis are amplified at the expense of the other (by h_t correctly classified) data. Thus making it more important for the next weak classifier to "fix" the errors of its predecessor.

The new weights are calculated with

$$w_i^{t+1} = \frac{w_i^t e^{-\alpha_t y_i h_t(x_i)}}{Z_t}. \tag{21}$$

Note that the expression $-\alpha_t y_i h_t(x_i)$ in equation (21) is just a compact way of writing:

$$\begin{cases} -\alpha_t & \text{if } h_t \text{ classified } x_i \text{ correctly,} \\ \alpha_t & \text{if } h_t \text{ classified} x_i \text{ incorrectly.} \end{cases}$$

This fact combined with equation (21) clarifies the previously mentioned reduction and amplification of the weights. Z_t is used to normalize D, keeping it as a probability distribution.

A practical improvement for AdaBoost

An interesting fact about AdaBoost is that the choice of the coefficient $\alpha_t = \frac{1}{2}\ln\left(\frac{1-e_t}{e_t}\right)$ makes it possible to do some computational simplifications [28].
The simplification can be seen as a corollary of the next theorem.

Theorem 2.1. *In algorithm 1 the choice of* $\alpha_t = \frac{1}{2}\ln\left(\frac{1-e_t}{e_t}\right)$ *will result in the sum of all new weights associated with the previous misclassified data being equal to one half. Furthermore the result also holds for new weights associated with the previous correctly classified data.*

$$\sum_{h_t(x_i)=y_i} w^{t+1}(i) = \sum_{correct} w^{t+1}(i) = \frac{1}{2}.$$

$$\sum_{h_t(x_i)\neq y_i} w^{t+1}(i) = \sum_{wrong} w^{t+1}(i) = \frac{1}{2}.$$

The proof is fairly simple yet it took some time, after the AdaBoost invention, before this result was discovered [28].

Algorithm 2 AdaBoost utilizing renormalization of the weights, source [28]

```
1: initialize $S_0 = 0$, $D^1(i) = \frac{1}{n}$
2: **for** $t = 1, ..., T$ **do**
3:     find $h_t \approx \underset{h}{\text{argmin}}\ [e_t(h) = \sum_{h(x_i)\neq y_i} w_i]$.
4:     let $N_c$ be the number of correctly guessed data and $N_w$ the corresponding number for the wrongly classified data.
5:     $w_i^{t+1} = \frac{1}{2N_c}$ for the the correct classified data.
6:     $w_i^{t+1} = \frac{1}{2N_w}$ for the wrongly classified data.
7:     $S_t = \alpha_t h_t + S_{t-1}$
8: return $S_T$
```

Proof.

$$w_i^{t+1} = \begin{cases} \dfrac{w_i^t}{Z_t} e^{-\alpha_t} & \text{if } h_t \text{ was correct at } x_i, \\ \dfrac{w_i^t}{Z_t} e^{\alpha_t} & \text{if } h_t \text{ was wrong at } x_i \end{cases} = \tag{22}$$

$$= \left[\alpha_t = \frac{1}{2} \ln\left(\frac{1-e_t}{e_t}\right)\right] = \begin{cases} \dfrac{w_i^t}{Z_t} \sqrt{\dfrac{e_t}{1-e_t}} & \text{, correct} \\ \dfrac{w_i^t}{Z_t} \sqrt{\dfrac{1-e_t}{e_t}} & \text{, wrong} \end{cases}$$

Since D is also probability distribution the sum of all new weights is equal to one. Splitting up the sum of w_i^{t+1} in the two parts such that one part consists of weights w_i^{t+1} associated with, by the previous weak classifier h_t, correctly classified data (and the other part with the wrong) using (22) followed by multiplying both sides with Z_t yields;

$$\sum_i w_i^{t+1} = 1 \iff Z_t = \sqrt{\frac{e_t}{1-e_t}} \sum_{correct} w_i^t + \sqrt{\frac{1-e_t}{e_t}} \sum_{wrong} w_i^t. \tag{23}$$

Using the fact that the error is defined as

$$e_t = \sum_{wrong} w_i^t,$$

and its immediate consequence

$$1 - e_t = \sum_{correct} w_i^t,$$

(23) implies that Z_t can be written as

$$Z_t = 2\sqrt{e_t(1-e_t)}.$$

Taking the sum over all wrong w_i^{t+1} and replacing Z_t with $2\sqrt{e_t(1-e_t)}$ results in

$$\sum_{wrong} w_i^{t+1} = \sum_{wrong} \frac{w_i^t \sqrt{\dfrac{1-e_t}{e_t}}}{Z_t} = \frac{1}{2e_t} \sum_{wrong} w_i^t = \frac{1}{2}.$$

Hence

$$\sum_{correct} w_i^{t+1} = 1 - \frac{1}{2} = \frac{1}{2}.$$

□

Relying on the theorem above, it is possible to compute all new weights w_i^{t+1} by simply re-normalizing the wrong and correct guessed weights for the selected weak classifier h_t.

The algorithms 1 and 2 are equivalent, however algorithm 2 is computationally faster to execute.

2.6.5 Properties

Training and generalization error

Given that the weak learning condition holds, AdaBoost is good at reducing the training error and in practice it usually has good guarantees against generalization errors (sometimes the generalization error continues to drop with additional rounds of boosting even after the training error has reached 0). As with most classification models, it is also possible to overfit a classifier with AdaBoost. The risk of overfitting is particularly increased when the boosting is done with too complex (such as large weak-trees) or too weak, hypotheses/weak classifiers [33, 26].

To understand how and why AdaBoost behaves like it does, different approaches have been proposed by various people.

Vapnik–Chervonenkis theory

VC-theory or Vapnik-Chervonenkis-theory is a theory that attempts to explain general statistical learning processes and has been applied to boosting and especially to AdaBoost. In section 2.5, basic concepts of VC-theory are introduced.

By applying VC-theory on AdaBoost, it can been shown that:

1. If the **weak learning condition** holds, the training error will very probably decrease to zero in only $\mathcal{O}(\log N)$ rounds, were N is again the size of the training data set [46].

2. It is possible to obtain a very probable upper bound (see expression (24)) on the generalization error [33],

$$P\left(H(\mathbf{x}) \neq y\right) \leq P\left(H(\mathbf{x}) \neq y|\mathcal{S}\right) + \mathcal{O}\left(\sqrt{\frac{Td}{N}}\right). \tag{24}$$

Here d denotes the **VC-dimension** of the weak hypotheses, T the number of rounds of boosting, $P(H(\mathbf{x}) \neq y|\mathcal{S})$ is the probability on the training sample $\mathcal{S}$ of size N.

VC-theory predicted a possible overfitting tendency when the amount of rounds and/or the complexity of the weak hypotheses increase. Empirically, overfitting due to a large number of rounds T has been shown not to be the common case [26] rendering the estimation (24) from VC-theory a bit inept. The latter led to other approaches being developed in order to analyze and explain the behaviour of AdaBoost (and some other boosting algorithms).

2.6.6 Greedy coordinate descent

The AdaBoost can be shown to be equivalent to greedily minimizing a convex upper bound, called the **exponential loss**, on the empirical risk (16) that is equipped with a 0-1 loss function (13). The following section is inspired by [50]. For simplicity, the set of hypotheses $\mathcal{H}$ is assumed to be finite.

Definition 7. *For a set of hypotheses $\mathcal{H}$ of size M, a training sample $\mathcal{S}$ of size N, the function, $C : \mathbb{R}^M \to \mathbb{R}$,*

$$C(\lambda_1, ..., \lambda_M) = \frac{1}{N}\sum_{i=1}^{N} e^{\sum_{j=1}^{M} -\lambda_j y_i h_j(\mathbf{x}_i)}, \lambda_j \geq 0,$$

*is called the **exponential loss**.*

Some reformulation of the returned output from the AdaBoost algorithm (algorithm 1) is needed in order to continue,

$$S_T(\mathbf{x}) = \sum_{t=1}^{T} \alpha_t h_{j(t)}(\mathbf{x}) = \sum_{j=1}^{M} \lambda_j h_j(\mathbf{x}). \tag{25}$$

In practice M is a much larger number compared to the number of rounds T from the AdaBoost algorithm, i.e the sum is greatly ”zero-padded” (a lot of zeros added). It is somewhat confusing to reformulate the output with an index function, $j(t)$, but in the following argument it appears to be necessary.

In the definition of the **exponential loss** every argument is associated to a unique weak classifier from $\mathcal{H}$. For the AdaBoost however, h_t refers to the solution of the minimization problem consisting of finding the weak classifier that minimizes the **weighted error** at round t,

$$h_t = \operatorname*{argmin}_{h \in \mathcal{H}} e_t(h).$$

Thus there is no requirement that classifiers from two (not consecutive) rounds to be different. Although this is painful to write and assuming the reader knows this, I will continue to write "h_t" instead of $h_{j(t)}$.

All information about the soft classifier S_T outputted from AdaBoost can be stored in the coordinate vector $\lambda = (\lambda_1, ..., \lambda_M)$ for the **exponential loss**. For the hard classifier $H = \text{sgn}(S_T)$ it is easy to confirm that $E(H) \leq C(H)$ for all (by AdaBoost produced) H when comparing the functional sum term-wise,

$$\mathbb{I}(H(\mathbf{x}) \neq y) = \mathbb{I}(y_i S_T(\mathbf{x}_i) < 0) \leq e^{-y_i S_T(\mathbf{x}_i)}.$$

Here $\mathbb{I}$ denotes the indicator function (14). The inequality follows since $e^x \geq \theta(x)$ where $\theta(x)$ is the Heaviside stepfunction.

Every term in C is convex due to the fact that every term can be expressed as a convex transformation, e^z, of affine functions of λ, $z = \sum_{j=1}^{M} \lambda_j h_j(\mathbf{x}_i)$. The convexity of C then follows from that every term in C is convex.

By preforming minimization of C over $\mathbb{R}^M$ one implicitly minimizes the empirical risk E. The convexity implies that performing a line-search from a point $\hat{\lambda}$ to find the minimum along $\hat{\mathbf{e}}_j$:s direction can be done by finding the stationary point. That is, finding the ${\Delta_\lambda}^*$ that minimizes

$$C_{\hat{\lambda}, \hat{e}_j}(\Delta_\lambda) = C(\hat{\lambda} + \Delta_\lambda \hat{\mathbf{e}}_j) \tag{26}$$

which can be done by solving

$$\frac{dC_{\hat{\lambda}, \hat{e}_j}}{d\Delta_\lambda} = 0. \tag{27}$$

The greedy coordinate descent algorithm 3 is equivalent to AdaBoost,

Algorithm 3 Greedy Coordinate Descent

```
1: initialize $\lambda_0 = \mathbf{0}$
2: for $t = 1, ..., T$ do
3:     $\forall$ coordinate $\hat{e}_j$ perform line-search in $\hat{e}_j$:s direction by solving (27).
4:     Choose the coordinate, $\hat{e}_t$, which attains the maximum reduction of $C$ while line-searching.
5:     Update $\lambda_t = \lambda_{t-1} + {\Delta_\lambda}^* \hat{e}_t$.
6: return $S_T(\mathbf{x}_i) = \sum_{j=1}^{M} \lambda_j h_j(\mathbf{x}_i) = \sum_{t=1}^{T} \alpha_t h_{l(t)}(\mathbf{x}_i)$
```

Theorem 2.2. *The algorithms 1 and 3 are equivalent, i.e. they will find the same hypothesis H.*

Proof. Relying on equation (25) that describes the relation between the different representations of the soft classifiers, to prove equivalency it is sufficient to conclude that the coordinate $\hat{e}_t$ (weak classifier h_t) and stepsize Δ_λ (coefficient α_t) the algorithms choose at a round t represent the same choices.

It's obvious that the setups are the same for $t = 0$ because $S_0 = 0$ and $\lambda = 0$. To construct a proof by induction, assume that the algorithms output, S_T and λ_{S_T}, represent the same soft classifier for $t = T$.

First I will show that for a fixed direction, the two algorithms will update with the same coefficient α_{T+1}, then finishing the proof by concluding that the algorithms will choose the same direction/weak classifier.

By assumption, equality must hold for the two sums

$$S_T(\mathbf{x}_i) = \sum_{t=1}^{T} \alpha_t h_{j(t)}(\mathbf{x}_i) = \sum_{j=1}^{M} \lambda_j h_j(\mathbf{x}_i).$$

Assume now that the direction of choice is $\hat{e}_k$.

Differentiation of the exponential loss (26) with respect to the stepsize Δ_k in the direction $\hat{e}_k$, and demanding it to be zero, gives

$$\frac{\partial C(\lambda_{S_T} + \Delta_k \hat{e}_k)}{\partial \Delta_k} = 0 \iff$$

$$\iff \sum_{i=1}^{N} -y_i h_k(\mathbf{x}_i) e^{-\Delta_k y_i h_k(\mathbf{x}_i)} \; e^{\sum_{j=1}^{M} -\lambda_j y_i h_j(\mathbf{x}_i)} = 0 \iff$$

$$\iff \sum_{i=1}^{N} -y_i h_k(\mathbf{x}_i) e^{-\Delta_k y_i h_k(\mathbf{x}_i)} e^{-y_i S_T(\mathbf{x}_i)} = 0$$

Recall that the expression $-y_i h_k(\mathbf{x}_i)$ is the compact way of arranging a sign, depending on the correctness of the weak classifier h_k. This gives,

$$0 = \sum_{h_k(\mathbf{x}_i) \text{ correct}} e^{-y_i S_T(\mathbf{x}_i)} e^{\Delta_k} - \sum_{h_k(\mathbf{x}_i) \text{ wrong}} e^{-y_i S_T(\mathbf{x}_i)} e^{-\Delta_k} \tag{28}$$

Using the recursive formulation for the weights w_i^T from algorithm 1, and the fact that they sum up to one gives

$$w_i^{T+1} = \frac{e^{-y_i S_T(\mathbf{x}_i)}}{\sum_i^N e^{-y_i S_T(\mathbf{x}_i)}} = \frac{e^{\sum_{t=1}^{T} -y_i \alpha_t h_t(\mathbf{x}_i)}}{\sum_{i=1}^{N} e^{\sum_{t=1}^{T} -y_i \alpha_t h_t(\mathbf{x}_i)}} \tag{29}$$

Dividing both sides in (28) with $\sum_{i=1}^{N} e^{-y_i S_T(\mathbf{x}_i)}$ yields, by using equation (29), for every term i, the weight w_i^{T+1}. Solving the equation for Δ_k gives

$$\sum_{h_k(\mathbf{x}_i) \text{ correct}} e^{-\Delta_k} w_i^{T+1} = \sum_{h_k(\mathbf{x}_i) \text{ wrong}} w_i^{T+1} e^{\Delta_k} \iff$$

$$\iff \Delta_k = \frac{1}{2} \ln \left(\frac{\sum_{\text{correct}} w_i^{T+1}}{\sum_{\text{wrong}} w_i^{T+1}} \right) = \frac{1}{2} \ln \left(\frac{1 - e_{T+1}}{e_{T+1}} \right). \tag{30}$$

The above equation shows that $\Delta_k = \alpha_{t+1}$ and it remains only to prove that the direction is the same.

By inserting (30) into the exponential loss (26) and dividing it with $\sum_{i=1}^{N} e^{-y_i S_T(\mathbf{x}_i)}$, the weights can once again be identified term-wise.

$$C(\lambda + \Delta_k e_k) = \frac{\sum_i e^{-y_i S_T(\mathbf{x}_i)}}{\sum_i e^{-y_i S_T(\mathbf{x}_i)}} C(\lambda + \Delta_k e_k) =$$

$$= \sum_i e^{-y_i S_T(\mathbf{x}_i)} \left(\sum_{correct} w_i^{T+1} e^{-\Delta_k} + \sum_{wrong} w_i^{T+1} e^{\Delta_k} \right) = [t = T] = C(\lambda) 2\sqrt{e_{t+1}(1 - e_{t+1})} \iff$$

$$\iff C(\lambda + \Delta_k e_k) = 2\sqrt{e_{t+1}(1-e_{t+1})}C(\lambda). \quad (31)$$

As $C(\lambda)$ is independent of the direction $\hat{e}_k$, the greedy algorithm 3 is searching for the direction which minimizes the dependent part $\sqrt{e_{t+1}(1-e_{t+1})}$. The AdaBoost algorithm 1 at round t chooses the h_k that minimizes the error e_t which is equivalent to minimizing $\sqrt{e_t(1-e_t)}$. If both algorithms are assumed to choose the same hypothesis, given that two, or more, equally good are presented, they are equivalent.

□

2.6.7 Other Boosting Algorithms

Some other boosting algorithms (that turn **weak learners** into **strong PAC algorithms**) can be constructed by simply replacing the **exponential loss** (definition 7) with another loss function and/or using different line-search methods. Different loss functions yields different choices of coefficients and hypotheses by inducing different weighting and updating rules. Some boosting algorithms are:

GentleBoost
Uses the same loss function as AdaBoost but with a restricted stepsize. This will limit the complexity of the set of possible hypotheses outputted and is a regularized version of AdaBoost [40].

LogitBoost
Uses a loss function with the expression $ln(1 + e^{-yS_T})$ instead of e^{-yS_T}, which is good on noisy data, as it tends to "give up" on the harder examples [42, 40].

Arcing-X4
A boosting algorithm similar to AdaBoost that minimizes a loss function that replaced the expression e^{-yS_T} with $(1 - yS_T)^5$ [40].

2.7 Cascade of Boosted Classifiers

As mentioned before, in 2001 (revised in 2003) Paul Viola and Micheal Jones published a paper "Robust Real Time Face Detection" describing a, at the time, new object detection framework that turned out to be the first object detection framework that produced feasible object detection rates for real time applications.

Their approach was to use a cascade of classifiers, each layer trained with AdaBoost over a set of hypotheses they introduced, decision stumps of Haar features [49].

Even though the AdaBoost algorithm enables the construction of a strong classifier with a weak learner it is not computationally feasible to let all of the weak classifiers evaluate every candidate region in a larger image. This is due to the fact that within an image there are lots of possible positions and sizes to search through in order to find all possible faces.

Example A typical approach to search through all candidate regions (sub-windows) in a single scale is to extract an image patch every second pixel in both width and height. With a classifier that operates on image-patches of the size 24×24 from a larger image of size 240×160, this single scale search procedure would yield 7344 different candidate regions to evaluate. In practice however a multiscale search procedure is used to search for objects in different sizes thus further increasing the number of candidate regions to evaluate.

Viola & Jones found that it is possible to adjust smaller, by AdaBoost, boosted classifiers such that they still reject many negative samples/patches while keeping a high proportion of the positives (instead of scoring a low training error) [49].

The cascade is then constructed as a gauntlet of boosted classifiers where each consecutive classifier is more advanced and computationally heavier than the previous one. Each stage classifier has the ability to immediately reject a patch. If a patch is not rejected it is sent for further investigation at the next stage. The idea is to let the earlier stages with simpler classifiers first weed out most negative samples before the heavier ones begin to distinguish the remaining and thus harder examples. See figure 11.

The power of the cascade approach comes mainly from its ability to capture the fact that in most object detection applications, larger parts of the data are negative, i.e. large amount of data can be rejected with little computation at the early stages [49].

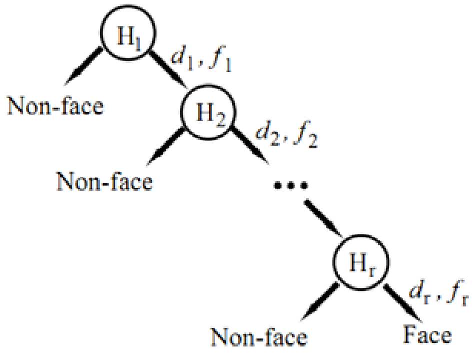

Figure 11: Shows a sketch of a cascade classifier.

Another important contribution from their work was the **integral image** that allowed very fast evaluation of the Haar-features (see section 2.1.3). The **integral image** was prior to its introduction in the Viola-Jones framework known as ”summed area table” [6].

2.7.1 Training Viola-Jones

In the Viola-Jones framework a cascade classifier is trained until it meets its target **false positive rate** $F_{\text{target}} > 0$. Each layer/stage is trained with data that has survived all previous layers, giving the next layer a harder task to solve. These layers were, by Viola-Jones, originally trained with AdaBoost over a set of overcomplete Haar features. However, other boosting algorithms and features may be used. For example OpenCV supports both Haar, rotated Haar and MB-LBP features as well as a variety of boosting algorithms [27].

With each layer i trained such that the **false positive rate** on the data reaching to it is f_i, the overall false positive rate F of a k-layer cascade becomes

$$F_k = \prod_{i}^{K} f_i. \tag{32}$$

Analogously the **detection rate** for the entire cascade (TP-rate) D is the product of the stage detection rates d_i,

$$D_k = \prod_{i}^{K} d_i. \tag{33}$$

To train the stage classifiers, AdaBoost is used. The user inputs the desired stage properties:

- Maximum false alarm rate $f^{\max}$.
- Minimum detection rate $d^{\min}$.

- Set of positive instances P.
- Set of negative instances N.

Here the set N and P consists of instances that have survived the previous layers.

Recall that the original AdaBoost first constructs a ranking classifier $S_T = \sum_{t=1}^{T} \alpha_t h_t$ later to be discretized with the threshold

$$\theta = \frac{\sum_t \alpha_t}{2}, \tag{34}$$

or $\theta = 0$ if the target set is $Y = \{-1, 1\}$, in order to score a low error rate.

Viola-Jones approach was to train the stage classifier with AdaBoost but after each round search for a new threshold, fulfilling the specified *FP*- and *TP* rates, rather than using the original threshold (34). As soon as it is possible to find a threshold that fulfills the requirements the boosting stops and a new layer is trained instead. See algorithm 4 [49].

Algorithm 4 Viola-Jones Train Cascade, source [49]

```
1: User inputs $f_i^{\max}$, $d_i^{\min}$, $F_{target}$
2: $i = 0, F_0 = 1.0, D_0 = 1.0$
3: while $F_i > F_{target}$ do
4:     $i := i + 1$
5:     $F_i = F_{i-1}$
6:     while $F_i > f_i^{\max} F_{i-1}$ do
7:         Do a round of boosting.
8:         Evaluate $D_i$ and $F_i$ with the current cascade on the evaluation set.
9:         Move threshold to fullfill minimum detection rate $D_i = d_i^{min} D_{i-1}$.
10:        Update $F_i$.
```

The constraints $f < f^{\max}, d > d^{\min}$ corresponds to the rectangular region left above $(f^{\max}, d^{\min})$ in the ROC-space. The stage training can be visualized as the ROC curve of the stage classifier is pushed towards the top left corner with every round of boosting. The process stops as soon at the curve intersects the feasible region. Viola-Jones stated in their paper that it was unclear at the moment (2003) how the generalization guarantees otherwise provided with AdaBoost were affected by moving the threshold [49].

While the Viola-Jones cascade approach was able to speed up the execution time drastically, it also enabled the training set to (implicitly) be very large and focused on harder examples. This is due to that the previous stages are efficiently selecting harder training data from a very much larger set of examples, in order to train the current layer [49].

2.7.2 Execution speed

The execution time of the cascade classifier depends on the number of features evaluated. This is a probabilistic measure since it is impossible to know in advance whether a stage is going to label a patch negative. According to the paper of Viola-Jones, the expected number of feature evaluations needed to classify an example depends on each stage's number of features and **positive rate**, see equation (9) in section 2.4.2. Their result can be confirmed by using the definition and linearity of the expected value.

Theorem 2.3. *For each layer i in a K layer cascade, let p_i be the* ***positive rate*** *on the data set X. Then the expected number of features evaluations used to classify an instance from X drawn at random is,*

$$N = n_0 + \sum_{i=1}^{K} n_i (\prod_{i<j} p_j) = n_0 + n_1 p_0 + n_2 p_1 p_0 + \ldots + n_K p_{k-1} p_{k-2} \ldots p_1 p_0. \tag{35}$$

Here n_i is the fixed number of features any examples reaching stage i evaluates at stage i, (if decision stumps are used, n_i is equal to the number of weak classifiers).

Proof. Let Y_i be the number of features evaluated at stage i by an image-patch that was drawn at random from X. There are two possible outcomes; either the image-patch reached stage i and evaluates all its n_i features or was rejected by some earlier stage thus evaluating zero features. Therefore $E[Y_i] = P(\text{survived})n_i + P(\text{died})0 = P(\text{survived}) = 1 \cdot p_0 p_2 ... p_{i-1}$.

The total number of feature evaluations Y is equal to the sum of the stage contributions, $Y = Y_0 + ... + Y_K$. The linearity of the expected value gives,

$$E[Y] = E[\sum_{i=0}^{K} Y_i] = \sum_{i}^{K} E[Y_i] = \sum_{i=0}^{K} n_i P(\text{survived}) = n_0 \cdot 1 + \sum_{i=1}^{K} n_i (\prod_{i<j} p_j).$$

□

3 Algorithms and implementation

3.1 Object detection

The object detection in question works by inputting an illumination image X of one or more channels, and outputs a set of rectangles $\{r_i\}$ defined in X, called objects.

Definition 8. *In this section a rectangle is defined as two points in a Cartesian system of coordinates with origin at the top left corner. See fig (12)*
A rectangle with the top left corner $P_1 = (x, y)$ and the width w and height h can be defined by its corners $(P_1, P_2) \in \mathbb{R}^4$ where $P_2 = (x + w, y + h)$.

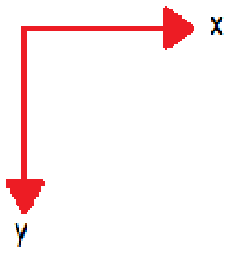

Figure 12: The coordinate system used to oriente the rectangles.

3.1.1 Overview

The detection algorithm involves several steps. Firstly there is the preprocessing of the input image X_{raw}. For example if edges are blurry one can apply edge detection and so on.
After all the preprocessing, the resulting image X is transformed into a set of downsized images $\{X^{(i)}\}$ where each member resembles X. Every rescaled image X^i is used as input to a single-scale detector.

The single-scale detector will have a sliding window with a width and height equal to the one specified in the cascade detector. This sliding window carves out regions from $X^{(i)}$ (usually called images patches) which are then fed into the actual cascade detector. The sliding window will move in a specific pattern extracting image-patches at different overlapping areas. Each image-patch is classified by the cascade as either positive or negative.

When, for every scale, all image-patches have been extracted and classified, the positive classified image-patches are collected. The next step is to compare them to each other in the original preprocessed image. In order to compare patches to each other, patches from downsized images are scaled to match the original image X.

Shapes and positions of the image-patches are compared to see if patches are similar. A clustering algorithm, see section 3.1.6, that outputs groups of image-patches (rectangles) is then applied. Every cluster (group) is merged into a single representative rectangle by taking the mean values of its members' corner positions and dimensions. The set of representative rectangles is then returned.

The main purpose of the clustering procedure above is to get rid of some false positives and to eliminate the artifact that many detections overlap around the actual object, see figure 13.

Algorithm 5 Object Detection from OpenCV

```
1: $X \leftarrow X_{raw}$ from preprocessing procedures.
2: Specify the cascade classifier, and its window size $(w, h)$.
3: Specify the scale factor $f > 1$ and interpolation method $\mathcal{I}$
4: Set $X^{(i=0)} = X$ and define its dimension $(M_i \times N_i)$.
5: Define the set of objects $O$.
6: for $f^{(0)} = 1$ do
7:     $Q^{(i)} = \frac{1}{f^{(i)}}$.
8:     if $M_i < w \vee N_i < h$ then
9:         break.
10:    compute $X^{(i)} \leftarrow cv :: resize(X, Q^{(i)}, \mathcal{I})$.
11:    insert, into $O$ the candidate regions from $detectSingleScale(X^{(i)}, Q^{(i)})$.
12:    $f^{(i)} = f^{(i-1)} f$.
13: $O \leftarrow rectangleGrouping(O)$.
14: return $O$.
```

3.1.2 Preprocessing

The preprocessing usually consists of different transformations of the input image X_{raw}. Common procedures are resizing and transformations between color spaces (such as RBG to gray scale). Other things that can be applied to the input image X_{raw} are transformations such as histogram equalization and edge detection, binaryzation etc.

3.1.3 Multiscale Search

In practice the distance from the camera and the size of objects vary. To handle this, the detection algorithm searches for objects of different sizes. However, the underlying cascade classifier will only classify image patches of a single (from training specified) pre-defined size (w, h) .

The detection algorithm implicitly searches for objects at varying distances by looking for objects in varying sizes. The last is done by transforming the input image X into a set of downsized images,

$$\left\{X^{(i)}\right\}_{i=0}^{q} = \left\{X^{(0)}, ..., X^{(q)}\right\}, \tag{36}$$

where $X = X^{(0)}$ resembles the original image, and the rest are downsized with a factor of Q^i with the "resize" function available in OpenCV. Parameters set in the procedure include the choice of interpolation method (a bilinear interpolation is set as the default) and the scale factor $f > 1$ used to calculate $Q = \frac{1}{f}$ (per default $f = 1.1$).

The construction of $\{X^{(i)}\}_{i=0}^{q}$ is done in an iterative procedure. After each iteration, the new image $X^{(i)}$ (together with Q^i) is mounted into the single scale detector which then returns a set of regions (rectangles). Each region is defined in relation to the original image X.

The loop stops when the factor $Q^i = \frac{1}{f^i}$ would generate a target image with some dimension less than corresponding dimensions specified by the cascade detector (w, h). See algorithm 5.

3.1.4 Single Scale Search

The single scale search is executed by a function available in OpenCV called **detecSingleScale**. **detecSingleScale** first computes the integral image of $X^{(i)}$ and stores it for later use by the feature evaluator (the procedure that evaluates features).

A sliding window with a width and height equal to the one specified in the cascade detector (w, h) is then used to extract image-patches from $X^{(i)}$. Right after a region is carved out, it is fed into the actual cascade detector.

First the window carves out a patch in the top left corner. The sliding window will move in a specific pattern from left to right jumping 2 or more pixels (in $X^{(i)}$) to the right for every new patch. When the sliding window is about to slide out to the right of the image it is reset two pixels down at the left-most edge of the image.

If an image patch that has a possible neighbour patch to the right is rejected at the first stage in the cascade the patch to the right is excluded from extraction, hence also rejected. It is however unclear which effect this has on the actual performance since no documentation or comment is available in OpenCV and no further investigation was done in this thesis.

A hypothesis is that this is an optimization to save the algorithm some unnecessary processing power. The first stages of the cascade are just designed to rule out the easy cases and if so is done one usually expects that the next patch is also a bad one. No such optimization exists for neighbouring patches downward or any other pattern.

This algorithm is per default parallelized in OpenCV.

3.1.5 Cascade Classifier

The work of the cascade classifier is to classify image patches of a fixed size (w, h). The classifier is applied in several stages each associated with a stage classifier. The type can be any, in this thesis the classifier investigated is a boosted classifier with regression trees of Multi scale Block LBP features as weak classifier (as in section 2.2.2). These are already trained by Crunchfish AB.

3.1.6 Rectangle grouping

The typical output of the cascade classifier will consist of many overlapping rectangles. This is due to the overlapping image patches that are extracted from the multi-scale search. In figure 13 we can see a typical output. Similar rectangles can be grouped by a clustering algorithm and merged into a single non-overlapping output to localize the object with higher confidence. Furthermore the grouping (clustering algorithm) eliminates some false positives.

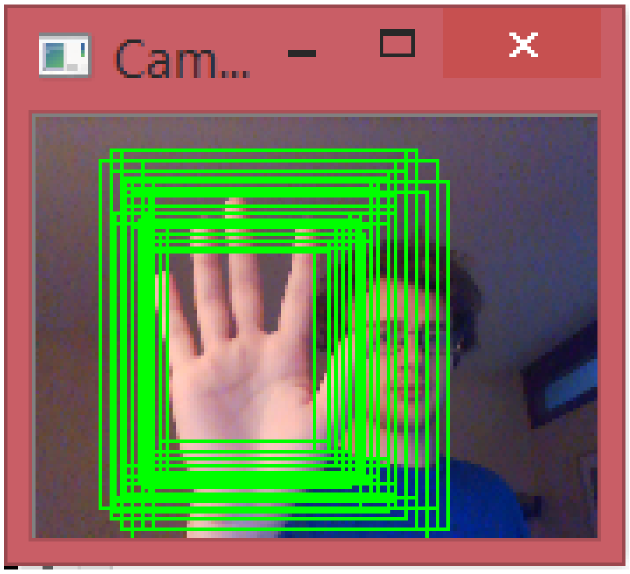

Figure 13: Shows a screen-shot of a typical cascade output from an open-hand detector run on a frame from a laptop camera.

Definition 9. *Two rectangles* $R_i = (P_1, P_2) = (x_i, y_i, x_i + w_i, y_i + h_i)$ *and* R_j *are said to be similar up to* ϵ *if and only if* $\|R_i - R_j\|_\infty \leq \delta_{i,j,\epsilon}$,
where

$$\delta_{i,j,\epsilon} = \frac{\epsilon}{2}\left(\min(w_i, w_j) + \min(h_i, h_j)\right).$$

The idea is that if two rectangles are similar according to def (9), they will also look similar in both shape and position.
For two specific rectangles R_i, R_j, the relation 'similar up to ϵ' can be visualized as all distances, between same type of edges from both rectangles (four distances and four pairs of edges, right pairs, top pairs ...), are smaller than $\delta_{i,j,\epsilon}$. The default in OpenCV is $\epsilon = 0.2$.

Definition 10. *A rectangle* $R' = (P_1', P_2')$ *is said to be inside* $R = (P_1, P_2)$ *up to* ϵ *if* R' *is inside* R *and furthermore:*
The distance from the same type of horizontal edges are less than $dy' = \epsilon h'$ *and for the vertical, less than* $dx' = \epsilon w'$

That is both

$$|P_1 - P_1'| := \begin{pmatrix} |x - x'| \\ |y - y'| \end{pmatrix}, |P_2 - P_2'|, \leq \begin{pmatrix} dx' \\ dy' \end{pmatrix}$$

Being inside up to epsilon, definition 10, is an extension to what normally is meant by being inside. In this case the extension calls for the extra criteria that the small rectangle (besides being inside the big one) also has some margin space to the outer rectangle, where the required margin size is relative to the smaller rectangle's dimension. If R' is a big rectangle, it also needs to be more "inside" R to be "inside up to epsilon" than a small rectangle.

The grouping algorithm works by building an undirected graph $G = \{V, E\}$ where rectangles relatively to the original image define the vertices and an edge $E_{j,i} = E_{i,j} = (V_i, V_j)$ between vertex i and j exists if and

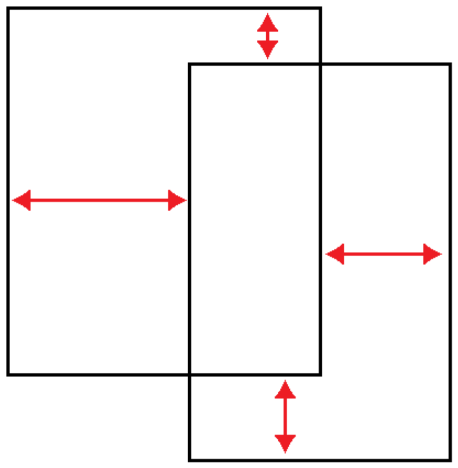

Figure 14: Visualization of the distances from the definition 9

only if the two rectangles are similar up to ϵ (definition 9).

The next step is to find all sub graphs of G by using some standard algorithm (such as Kruskals or Prims algorithm). For each and every subgraph (cluster) C_l, $l \in \{0, 1, ..., k\}$ of rectangles, a representative rectangle is computed by calculating the mid point rectangle $R_l^m = \frac{\sum_{i \in C_l} R_i}{|C_l|}$.

If a cluster has less than a certain specified number ($\theta \in \mathbb{N}$) of member rectangles it is immediately discarded. Otherwise it is considered to qualify for another spatial test, a check to see if its representative rectangle does not contain any small high confidence (large cluster size) rectangle inside. For R_i^m to pass the test no other cluster representative R_j^m from a cluster C_j of size more than $|C_i|$ (or if the cluster to test has less than three members, more than three) shall be inside of R_i^m up to epsilon, see definition 10.

Algorithm 6 Rectangle Grouping in OpenCV 2.410

```
 1: Define the similar parameter $\epsilon > 0$ and the grouping threshold $\theta \in \mathbb{N}$.
 2: Build undirected graph $G = (V, E)$ using rectangles as vertices and the relation between two rectangles,
    from definition 9 with $\epsilon$, as edges.
 3: Calculate all the sub graphs $\{G_1, ..., G_k\}, G_l = \{V, E_l\}$.
 4: Calculate all representative rectangles $R_l = \frac{\sum_{i \in V_l} r_i}{|V_l|}$.
 5: Define the set of objects $O = \{\}$
 6: for $l = 1, ..., k$ do
 7:     if $|V_l| < \theta$ then
 8:         skip current cluster.
 9:     for $m = 1, ..., k$ do
10:         if $l \neq m$ then
11:             if $R_m$ is inside $R_l$ up to $\epsilon$ then
12:                 skip current cluster if $|V_l| < 3$.
13:                 skip current cluster if $|V_m| > \max(3, |V_l|)$.
14:     if current cluster not skipped then
15:         $O \leftarrow \{O, R_l\}$
16: return the set of objects $O$.
```

All representatives that pass the check are then outputted as detected objects.

3.2 Adaption Algorithms

The approach is to unsupervised collect data in test-time about the target environment with the elusive detections and find the features associated with the missed detections. Once features associated with the missed detections are isolated their impact on the classifier can be altered. The adaption occurs as a single event, triggered when enough data about the environment is collected, so it is not online-adaption in the same sense as in the paper by Vidit Jain and Erik Learned-Miller, [36].

As the stage classifier first ranks different regions (rectangles) and then classifies them by thresholding, the idea (inspired by [36]) is to run the original cascade first and then reclassify the regions that are ranked close enough to the stage threshold with the adapted cascade. That is reclassification of the low confident regions at stage level.

In the paper [36] reclassification was formulated with the help of a score updating function, which inspired my own work (see definition 13).

In the paper "Online Domain Adaptation of a Pre-Trained Cascade of Classifiers", of Vidit Jain and Erik Learned-Miller [36], one can read about results from a similar approach with adaption by reclassification of the low confident output on a cascade with Haar features. There it was concluded that the adaption performed on the initial stages had little or no effect. The reason being that the initial stages were not used for "pure" classification but rather for rejecting big amounts of negative samples. As this is also the case of the cascades in my thesis I assume that adaption of the initial stages is pointless.

3.2.1 Overview

The entire adaption procedure involves several different sub-methods which are divided into three sections/parts with respect to when they are executed in relation to each other. These sections can in turn be divided into different steps. All parts and their steps along with the parameters they introduce have been listed below to give the reader a quick overview.

Algorithm 7 Adaption Algorithm

```
1: Input a Viola-Jones based Cascade Classifier with regression trees as weak classifiers.
2: Let $\{X_0, ..., X_n\}$ be a temporal sequence of images from the target environment.
3: Input a sliding window $W$ of order $k$ for tracking false negatives from the sequence. See section 3.2.2.
4: Specify the collection procedure $\mathcal{C}_{collect}$. See first paragraph in section 3.2.2.
5: Specify the leaf update procedure $\mathcal{W}_{update}$. See section 3.2.4.

6: for $i = k, ..., n - k$ do
7:     if $W$ triggers at image $i$ then
8:         for each stage to adapt do
9:             Collect regions and their leaf footprints from $W$ with $\mathcal{C}_{collect}$.

10: Let $\mathbf{w}_s$ denote the leaves for stage $s$ and $N$ the number of images used to collect regions.
11: for each stage $s$ to adapt do
12:     Calculate stage leaf uses $\mathbf{U} = \begin{pmatrix}\mathbf{u} & \mathbf{u_R} & \mathbf{u_{Rc}}\end{pmatrix}$.
13:     Retrieve stage leaf increments $\mathbf{\Delta w}_s \leftarrow \mathcal{W}_{update}(\mathbf{w}_s, \mathbf{U}, N)$.
14: Return all leaf increments $\mathbf{\Delta w}$
```

Part I, Collection

The collection part is the part of the adaption procedure that collects regions associated with the elusive detections. The idea is to let the collection algorithm keep on running until "enough" data of the elusive environment is collected.

1. **Event tracking**
   Event tracking is the initial part of the collection procedure and consists of tracking down an event of a missed detection, see definition 11. Upon a detection of an event a window (i.e. a temporally ordered set) of images W, also referred to as **event-window**, that contains the event of a probably missed detection is returned.

   Parameters introduced by the **event tracking** part are:

   - Type of event.
   - Order of event (or window size W).
   - Object detection parameters.

   However, the only type of events used in this thesis are the *pMissed*, see definition 11.

2. **Collecting regions from events**
   This step consists of selecting images from an **event-window** W followed by collecting data from selected images by running a, for this thesis, especially developed object detection algorithm, similar to the one in section 3.1, called **detectMultiScale_collectAdaptiveData**. This runs an ordinary object detection algorithm with difference being that all candidate regions at specified stages are stored together with their evaluated leaf values.

   Parameters introduced in this step:

   - The frames collected from W (partly determined by the choice of method to define ROI, regions of interest).
   - Number of stages to adapt.

3. **Linking of regions to events**
   Just before the termination of **detectMultiScale_collectAdaptiveData**, all candidate regions are checked to see if they are associated with the event (missed/elusive detection) or not. Regions associated with the event are put into a set called ROI (regions of interest). The rule that decides which regions would be associated (or not) with the elusive detection is the natural parameter introduced by this step.

Part II, Adaption

The goal of the adaption part is to construct a set of new leaves (i.e. leaf values) used by the adapted object detection algorithm. The adaption is performed when "enough" data is collected.

The term "enough" is here left for interpretation, in practice it is actually a parameter.

- The trigger of adaption.

The procedure can be further divided into two consecutive steps, as follows

1. **Processing collected data**
   The collected data consists of regions and their stage-specific score and leaf uses. In this step, statistical properties such as standard deviation and mean value for the collected data are estimated.

2. **Output new leaf values**
   The latter step uses the estimated data from the former to calculate leaf increments $\mathbf{\Delta w}$ for each target stage. At stage level, these leaf increments are used to construct new leaf values $\hat{\mathbf{w}} \leftarrow \mathbf{w} + \mathbf{\Delta w}$.

   Parameters:

   - The choice of leaf-update procedure.

Note that the adaption does not concern any topology of features in the regression tree or any other internal property of features itself, just the leaf values in the bottom of the regression tree.

Part III, Reclassification

As a new set of leaf values (with vector representation $\hat{\mathbf{w}}$) is available, it can be used to carry out the adaption at stage level, by reclassification of low confidence regions.

The adaption object detection algorithm works by first classifying regions with original cascades' leaves $\mathbf{w}$. Whenever a region is classified (using the original leaves $\mathbf{w}$) with a score being within a certain distance (ϵ_r - standard deviations) away from a stage threshold θ_S stage S, the new leaves values $\hat{\mathbf{w}}$ are used to reclassify it. The described reclassification can be formulated using a score update function, see definition 13.

Parameters:

- The relative size of the reclassification area (interval), ϵ_r.

3.2.2 Missed detections

Definition 11. *Event of the probably missed detection*

Let $\mathcal{O}$ be an object detection algorithm and

$$\{X_0, X_1, ..., X_n\}$$

a temporal sequence of frames (images) where $X_i = X(t_i)$ is the frame captured at time t_i.

If assumed that the frames were all captured from a camera in the stated order in time, the event of "probably missed detection of order k" at X_i is the event where $\mathcal{O}$ detects a single object in all images in a window of images W except the middle image X_i.

Here

$$W = \{X_{i-k}, ..., X_i, ..., X_{i+k}\}.$$

The short notation used in this thesis for definition 11 is *pMissed*.

As the event tracking is used for unsupervised training/collection of data from false negatives, optimally the triggering of the tracked event would also be purely correlated with a false negative. In reality such guarantees are in general hard to give.

The idea behind the choice of definition in 11 ("probably missed detection") is to be able to collect enough false negatives while keeping the unwanted triggering low (or non existing). It is obvious that if the detector misses a detection in a series of detections, some "probably missed detection" event of order greater than zero, is triggered.

For Example the trigger of event "probably missed detection" of order 0 is just any negative classified instance. But the same event of order 1 means the object detection algorithm detects exactly one object in both of X_{i-1} and X_{i+1} while not being able to detect anything in X_i, see figure 15. It goes without saying that events of order $k = 0$ guarantee nothing about the cause and are not further discussed.

The converse statement is not necessarily true. There are more things than missed detections that are able to trigger the collection algorithm, see figure 16. The events that could theoretically lead to unwanted tracking (and ultimately the collection of true negatives) are:

False positives:
A single true negative in the middle of a series of false positives.

Obstruction:
The item/object is obstructed.

Low light:
The item is not visible because of low/high light condition.

Fast moving objects:
The object is moving too fast for the camera and is blurred out.

Frame bounds
The object is not fully inside the frame.

e.t.c
Anything else that imposes a missed detection with a true negative in the middle frame.

An assumption used in this thesis is that unwanted triggering could be minimized by applying a sufficiently high frame rate in conjunction with a target detector having a sufficiently low (general) false positive rate.

To motivate the assumption, note that if the frame rate is sufficiently high, obstruction of an object will unlikely affect just one single frame. The same explanation is also valid for the low/high light. If the light varies slowly or is just low, a sufficiently high frame rate is enough to guarantee the effect to be prominent in more than one frame/image.

With definition 11, any event of order k also implies triggering for all events of lower orders, thus lower order tracking also means more collected events/frames.

In summary, the event window from definition 11 (with order $k > 0$) can be used to track missed detection (false negatives), however some unwanted (true negatives) may also trigger the collection. The trade-off between the rates of collected false and true-negatives can be regulated with the window-order. The collection process becomes more conservative with increasing order (window size $|W|$).

For future work, it would be interesting to construct ROC curves to visualize the trade-off between false and true negatives, depending on the window order. This can be done by using events of different orders.

3.2.3 Collection of adaption data

The previous part explains how an event tracking algorithm unsupervised tracks possible false negatives. This part explains procedures to extract data from triggered event windows. The data extracted from a window is the number of collected images and the stage leaf use quantities (see equations (39) -(41)) which heavily depends on the collection method. The leaf uses $\mathbf{u}$ are divided into two categories, one which is assumed associated with the missed detection and the other one which is assumed not.

In this thesis the adaption is performed at stage level on Viola-Jones based cascades classifiers that are utilizing regression trees as weak classifiers. Naturally, data about how the cascade behaves around tracked

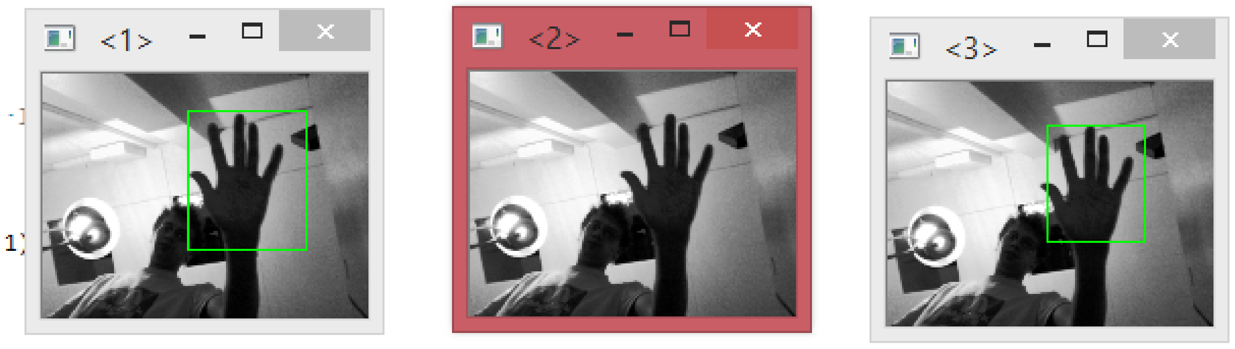

Figure 15: Shows a false negative being tracked. The good case.

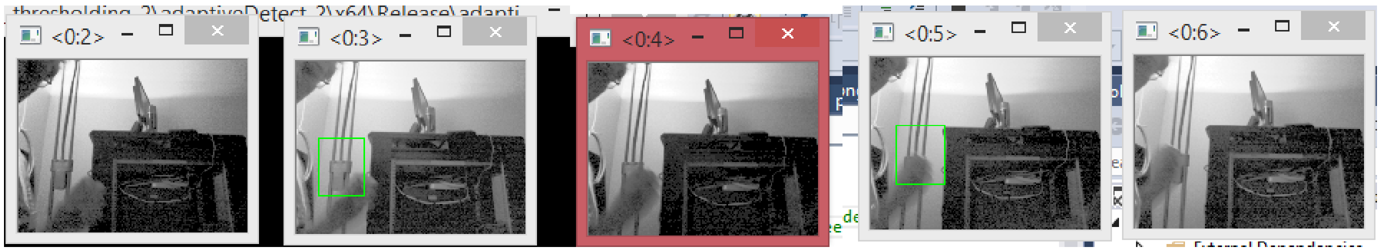

Figure 16: Shows a true negative being tracked. The bad case

events is collected on stage level.

Let $s \in \mathbb{N}$ be a stage in the classifier and $H = H_s$ its binary stage classifier that classifies a data point $\mathbf{x}$ by thresholding its sum of collected stage leaves against a stage threshold $\theta = \theta_s$, see below

$$H(x) = \begin{cases} 1 & \text{if} \quad S_T(\mathbf{x}) > \theta \\ -1 & \text{otherwise,} \end{cases}$$

Here S_T denotes the stage specific ranking classifier with T number of weak trees,

$$S_T = \sum_{t=1}^{T} \alpha_t h_t.$$

Each $h = h_t$ is a regression tree and α_t a real number therefore each term in S_T is also a regression tree. That is, each $\alpha_t h_t$ outputs a real value picked amongst the leaves of the regression tree $\alpha_t h_t$, $\{w_1^t, ..., w_k^t\}$,

$$\alpha_t h_t(\mathbf{x}) \in \left\{w_1^t, ..., w_k^t\right\}, w_i^t \in \mathbb{R}. \tag{37}$$

Here for a stage ordering all leaves $\mathbf{w_s} = (w_1^1, ..., w_k^T)$, a classified region (r) leaf use (and thereby its feature data) have been described by the **leaf footprint function** $\mathbf{L}(r) = \mathbf{L_s}(r)$ of the same dimension. The **leaf footprint function** contains 1 at index i if the corresponding leaf value was picked and 0 otherwise

$$\mathbf{L}(r) := \left(\mathbb{I}\left(\alpha_1 h_1(r) = w_1^1\right), ..., \mathbb{I}\left(\alpha_t h_t(r) = w_1^t\right), ..., \mathbb{I}\left(\alpha_t h_t(r) = w_k^t\right)\right). \tag{38}$$

Here $\mathbb{I}$ denotes the indicator function, see eq. (14).

The collection algorithm collects regions from specific frames/images within every triggered event window.

For an event window of frames, W, let $\Omega_s = \Omega$ be the set of all regions from W that are investigated by the stage classifier, and $C \subset \Omega$ the set of collected (stage) regions from W. Ω is also referred to as the set of all (stage) candidates, and any $r \in \Omega$ a candidate region.

Every collected region is put into exactly one of the two (stage-) sets, ROI (regions of interests) and ROI^c, forming a bipartition of the all collected regions C. The idea is to, with the help of some rule, determine if every individual region is associated with the possible "missed detection" (true negative). If so, the region is stored in ROI.

The total footprint on the stage classifier from each set can then easily be formulated with the help of the **leaf footprint function**,

$$\mathbf{u} := \sum_{r \in C} \mathbf{L}(r) \tag{39}$$

$$\mathbf{u_R} := \sum_{r \in ROI} \mathbf{L}(r) \tag{40}$$

$$\mathbf{u_{Rc}} := \sum_{r \in ROI^c} \mathbf{L}(r). \tag{41}$$

The vectors above contain information on how many times each leaf was used by each class. Each component u_i of $\mathbf{u}$, is the exact number of times the leaf number i was used by collected image patches/candidate regions. Analogously u_{Ri} is exactly the number of times leaf i was collected by patches from the set ROI. Furthermore since ROI and ROI^c is a bipartition, the relation $u_i = u_{Ri} + u_{Rci}$ holds for each index i which implies

$$\mathbf{u} = \mathbf{u_R} + \mathbf{u_{Rc}}. \tag{42}$$

Collect methods

Different definitions of C and ROI yield different collected patches and bi-partitions and are crucial parameters in the adaption algorithm. These choices can be summarized in different collect methods. The ones that were implemented in this thesis are listed below:

Let $W = \{X_{i-k}, ...X_{i+k}\}$ be a triggered event window of images, see definition 11, and r a candidate region ($r \in \Omega$). Note that the collection procedure is also performed at stage levels and therefore Ω refers to all possible candidate regions within a stage and $C \subset \Omega$ refers to the collected regions within a stage.

Collect method 1a: Covering region
The idea is to estimate an area in the middle frame X_i that contains a false negative with the two surrounding detections from X_{i-1} and X_{i+1}. This is done by placing the minimal covering rectangle of the detections inside X_i.

All candidate regions from X_i are collected and any region that is contained within the minimal covering rectangle is put into ROI.
See figure 17.

Collect method 1b: Expanded cover
Same as above but the minimal covering rectangle has an expanded diagonal, thus putting a bigger proportion of regions from C in ROI.

Collect method 2: Is similar
All candidate regions from X_i are collected and any region is put into ROI if it is similar up to ϵ (see definition 9) to at least one of the detections from X_{i-1} and X_{i+1}.

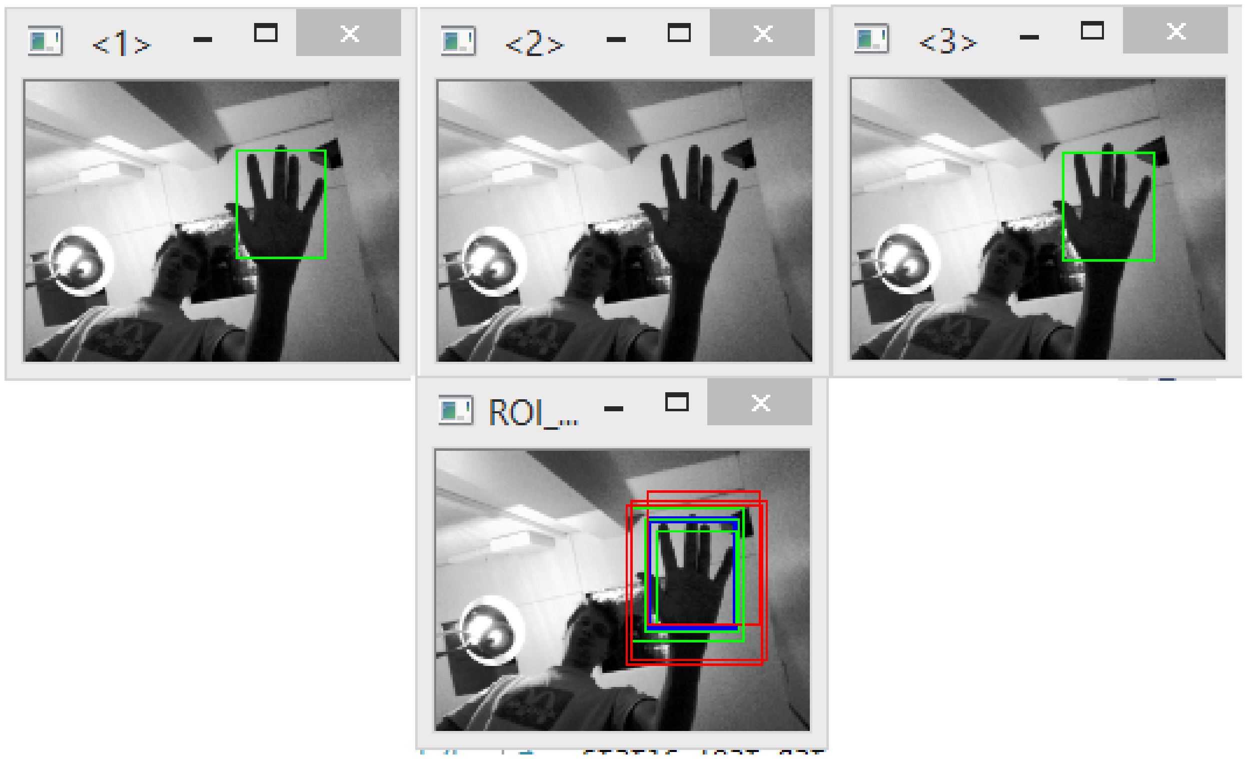

Figure 17: A visualization of collection methods 1 and 2. Here the middle frame lacks detection while the possible candidate regions are depicted in the frame below. The covering rectangle from surrounding detections is shown in blue and all similar candidate regions in green, and the non similar candidate regions in red.

Collect method 3: Cluster members

The idea is to have a method that counters the problem that the middle frame X_i often lacks candidates for later stages. By assuming that X_i has very similar underlying distributions as X_{i-1} and X_{i+1} candidates from these two are collected instead.

Let C be all candidate regions from X_{i-1} or X_{i+1}, then any $r \in C$ is put into ROI if it is also a member of a cluster that generated a detection in $X_{i \pm 1}$.

By counting the number of times individual leaves have been evaluated by candidate regions from ROI and ROI^c, estimation can then be made on how much a leaf is associated with a missed detection. See equations (39) - (41).

An interesting idea is to try all pairs (or $k-$tuples (were $k > 2$ is fixed)) of leaves to see which combinations are most associated with ROI and ROI^c. This is however not further investigated in this thesis.

3.2.4 Leaf update procedures

The purpose of the **leaf update procedure**, $\mathcal{W}_{update}$, used by the adaption algorithm (see algorithm 7) is to produce suggestions on how to alter the impact from features in order to adapt the classifier to the elusive environment without losing too much of its general performance (in other situations).

The cascade classifiers (within the scope of this thesis) utilize regression trees and therefore (combination of) evaluated features correspond to (real valued) leaves of weak classifiers. As already mentioned in the overview section in the beginning of this chapter, modification of the cascade is performed without adding new features, changing any topology of the features in weak classifiers/trees or any other internal property of the LBP feature descriptor. Instead all modifications concern leaf values and are returned from the **leaf update procedure** as leaf value increments. The increments can be represented with a vector of real numbers, $\mathbf{\Delta w} \in \mathbf{R}^n$ and applied by simply adding them to the original leaves $\mathbf{w}$,

$$\hat{\mathbf{w}} = \mathbf{w} + \mathbf{\Delta w}. \tag{43}$$

Here $\mathbf{w}$ denotes the original cascade leaves.

An advantage with this approach is that any adaption result obtained is easy to store, apply and reverse as it just involves a vector addition. For example, by storing earlier obtained (successful) adaption results, one can focus on recognizing the elusive environment and applying the leaf increments (by equation (43)) rather than redo the entire adaption procedure.

In context, the **leaf update procedure**s are invoked when the detector/collection algorithm have been collecting enough missed detection events, see definition 11 and algorithm 7.

The data used by all update procedures are the leaf uses $\mathbf{U} = \begin{pmatrix}\mathbf{u} & \mathbf{u_R} & \mathbf{u_{Rc}}\end{pmatrix}$ (equations 39-41) from all collected regions, the number of frames that have been used to collect the regions N and statistics from the original leaf values.

Standard deviations of the original leaf values are used throughout the update procedures. Experiments (see section 5.1) suggest that for any stage, it may be meaningful to consider different underlying distributions for leaves with different subindices in their weak trees. To counter this aspect, still being able to handle cascades with variable weak tree depths, the leaves were divided into two groups, left and right leaves, depending on their position relative to their parent nodes. Separate estimations of σ for each group of leaf values were considered in the update procedures, σ_{left} and σ_{right}.

Two different approaches have been considered and implemented in this thesis:

- ρ-Method
- Linear Programming-methods

3.2.4.1 The ρ-Method

Definition 12. *For a leaf number i with leaf uses by ROI, u_{Ri} and leaf uses by C, u_{Rci}, the ρ value of leaf i is defined by the quotient $\rho_i = \dfrac{u_{ri}}{u_i}$.*

The idea with the ρ-method is to find, with definition 12 above, decision making rules to single out leaves that are associated with *ROI* or *ROI*^{c} and increase or decrease their values. See algorithm 8.

By applying the ρ-method, the idea is to give the user assurance that only leaves that are sufficiently associated with the wanted or unwanted regions are affected. However, because of the large amount of parameters left for the user to specify the method was never a target for more investigations within this thesis, in order to avoid increasing the complexity of the problem.

3.2.4.2 Linear Programming-methods

The general approach with the LP-methods is to output leaf increments $\mathbf{\Delta w}$ such that if they are to be applied, as in equation (43), they would maximize the increased stage score for candidates in *ROI* while

Algorithm 8 Rho Method

```
1: Let $S$ denote the set of stages to adapt on.
2: for each stage $s \in S$ do
3:     Define lower and upper bounds $0 < \rho_{lw} \leq \rho_{up} < 1$.
4:     Define the rho-adaption weight parameter $\epsilon_{trim} > 0$.
5:     for each leaf $i$ do
6:         $\Delta w_i = \begin{cases} 0 & \text{if} \quad u_i = 0 \\ \epsilon_{trim}\sigma_i & \text{if} \quad \rho_i > \rho_{up} \\ -\epsilon_{trim}\sigma_i & \text{if} \quad \rho_i < \rho_{lw} \\ 0 & \text{otherwise.} \end{cases}$
7: Return $\mathbf{\Delta w}$
```

keeping the general increase in stage score bounded/constrained.

How would the stage classifier be affected if a leaf value w_i were increased with Δw_i?

Let $I_i = \dfrac{u_i}{N}$ be an estimation of the relative leaf use per frame where N is the number of collected frames. By adding the leaf increment Δw_i to the leaf value w_i, the overall outputted score per frame from the stage classifier can be estimated to have increased with $I_i \Delta w_i$. By applying the same estimation for all leaves w_i within a stage, the estimated increased stage score per image can be expressed as a sum (or dot product),

$$\sum_i I_i \Delta w_i = \mathbf{I}^T \mathbf{\Delta w}_s. \tag{44}$$

The estimated increased stage score per image $\mathbf{I}^T \mathbf{\Delta w}_s$ is also referred to as the **impact** from $\mathbf{\Delta w}_s$.

Be analogously defining the quantities,

$$\mathbf{I_R} := \frac{\mathbf{u_R}}{N} \tag{45}$$

$$\mathbf{I_{Rc}} := \frac{\mathbf{u_{Rc}}}{N} \tag{46}$$

Algorithm 9 Optimize Impact Method

```
1: Let $S$ denote the set of stages to adapt on.
2: for each stage $s \in S$ do
3:     Define box parameters $\epsilon_l, \epsilon_u$.
4:     Define the upper bound for increased stage score mass $q > 0$.
5:     Specify the optimization problem $\mathcal{P}$.
6:     Retrieve the leaf increments $\mathbf{\Delta w}_s$ as the solution to $\mathcal{P}$.
7: Return $\mathbf{\Delta w}$.
```

three different linear programming problems over the box space,

$$\text{Box} = \left\{ \mathbf{\Delta w} \in \mathbf{R}^n \middle| \quad \Delta w_i \in [-\epsilon_l \sigma_i, \epsilon_u \sigma_i] \right\}, \tag{47}$$

were considered for the leaf update procedure, see algorithm 9.

LP-Method $\mathcal{P}_1$

$$\begin{aligned} \max_{\mathbf{\Delta w} \in \text{Box}} \quad & \mathbf{I_R}^T \mathbf{\Delta w} \\ \text{subject to} \quad & \mathbf{I}^T \mathbf{\Delta w} = q \end{aligned} \tag{48}$$

LP-Method $\mathcal{P}_2$

$$\begin{aligned} \max_{\mathbf{\Delta w} \in \text{Box}} \quad & \mathbf{I_R}^T \mathbf{\Delta w} \\ \text{subject to} \quad & \mathbf{I}^T \mathbf{\Delta w} \leq q \end{aligned} \tag{49}$$

By setting

$$\mathbf{I}_\gamma = (\mathbf{I_R} - \mathbf{I_{Rc}})^T, \tag{50}$$

the method $\mathcal{P}_3$ is introduced as

LP-Method $\mathcal{P}_3$

$$\begin{aligned} \max_{\mathbf{\Delta w} \in \text{Box}} \quad & \mathbf{I_R}^T \mathbf{\Delta w} \\ \text{subject to} \quad & \begin{cases} 0 \leq \mathbf{I}^T \mathbf{\Delta w} \leq q \\ \mathbf{I}_\gamma{}^T \mathbf{\Delta w} \geq 0 \end{cases} \end{aligned} \tag{51}$$

The differences between the methods above lie within their constraints. Method $\mathcal{P}_1$ has an equality constraint that forces the overall increased score mass to q.

$\mathcal{P}_2$, also referred to as **inequality adaption**, applies an inequality constraint instead, thereby increasing the size of the search space.

Method $\mathcal{P}_3$ uses the extra condition that the biggest proportion of the distributed score mass is placed on the regions of interest. The intention is to avoid adaption of immature stages designed to handle big amount of negative data. The quantity $\mathbf{I}_\gamma{}^T \mathbf{\Delta w}$ is called the edge and the method $\mathcal{P}_3$ is referred to as **edge adaption**.

As the linear programming methods have affine target functions, existence of solutions to them are acknowledge after recognizing that the search spaces are compact (the box space in equation (47) is bounded and closed in $\mathbb{R}^n$).

3.2.5 Reclassification

As the domain for adaption algorithms concerns state of the art classifiers they are assumed to be sufficiently accurate.
In the previous and inspiring work [36] (by Vidit Jain and Erik Learned-Miller) adaption of pre-trained classifiers, that is also assumed to be somewhat good, is finally accomplished by reclassification of low confidence regions. In this thesis a similar approach is deployed, with similar reasoning.

The reason given was that by assuming that the original classifier is good, one can further assume that any data classified with good confidence, i.e. large margin, is very likely to be correctly classified. Therefore adaption is only considered for the low confidence instances (uncertain data).

By 'low confidence' one addresses data that have been classified very close to the threshold. Vidit Jain and Erik Learned-Miller proposed the use of a **score updating function** to represent reclassification. This work inspired mine.

Definition 13.

Let S be the ranking classifier (using regression trees) for a stage with threshold θ and let $\hat{S}$ be a to S identical classifier besides having other (adapted) leaf values $\hat{\mathbf{w}}$.

Then for a data point $\mathbf{x}$, the ***score updating function****, with reclassification margin $\epsilon_r > 0$, is the function*

$$S'(\mathbf{x}) = \begin{cases} \hat{S}(\mathbf{x}) & \textit{if} \quad |S(\mathbf{x}) - \theta| < \epsilon_r \sigma \\ S(\mathbf{x}) & \textit{otherwise}, \end{cases} \tag{52}$$

Here, θ refers to the stage threshold and σ refers to the standard deviation of stage scores.

By deploying the **score updating function** S', the new adapted stage classifier H' can be described as,

$$H'(\mathbf{x}) = \text{sgn}(S'(\mathbf{x}) - \theta). \tag{53}$$

The standard deviation σ is estimated (in this thesis) with collected candidates that were used to train the new leaf increments.

4 Data

4.1 Cascade Classifiers

Four cascade classifiers pre-trained on the pose "open hand" were especially chosen in this thesis. These instances of object detection algorithms, here referred to as A, B, C, D, are based on cascades of boosted classifiers with multiple branch regression trees as weak classifiers. The weak trees use Multi-scale-Block Local Binary Patterns as features, further described in section 2.2.2. These cascades based object detection algorithms in $\{A, B, C, D\}$ are the target for the adaption algorithm.

Other cascade classifiers than the ones in the set above, that are boosted (a procedure in the training process) with the same set of hypotheses were also used in this thesis to collect data about their regression tree leaves. The total number of cascades studied was ten.

4.2 Data to adapt on

Eleven videos were recorded with the native camera from a Nexus 9 device using a fixed frame rate at 30 frames per second and a fixed image resolution of 320×160. The videos were ordered in two sets.

The first set of videos called **Photo_of_hand**, depicts a high resolution, printed photograph taped onto a screen. The background light is varied manually. Both the position of the camera and the direction it faces are fixed during recording of **Photo_of_hand**. The fixed setting of the camera and the photo eliminates some variations, that otherwise would emerge from typical movements of the hand. However, some light is reflected from the photo. It is possible that the reflected light would generate some artifacts. The set contains four sequences with a total of 2507 images.

The second set of videos called **Real_hand** shows a hand, slowly moving over the device in manually varying light. **Real_hand** consists of seven sequences $\{S_1, ..., S_7\}$. In total **Real_hand** contains 5369 images.

Light was varied by tuning (manually) the luminous emittance from a floor lamp in a room with no other light sources.

4.3 Cascade leaves

All leaf values were collected (picked from their regression trees) from cascades in $\{A, B, C, D\}$. Leaves from six other cascades (using the same types of hypotheses) were also collected. The leaf values were stored in files outputted by "adaptiveDetect_2", see section 5.

5 Experiments and Results

In order to carry out all experiments, a framework for my thesis, written in C++, was implemented as an extension to (the open source library) OpenCV (2.410) project "object_detect" called "adaptiveDetect_2". Matlab (student-license) was also used to produce some graphs.

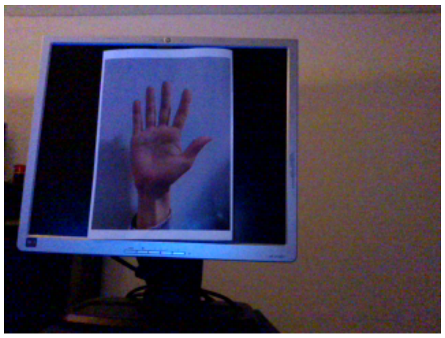

Figure 18: Shows an image sample from the data set ”photo_of_hand” in a low light environment

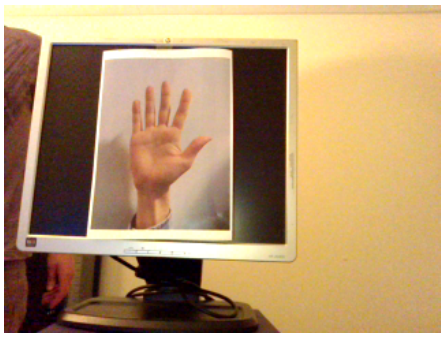

Figure 19: Shows an image sample from the data set ”photo_of_hand” in a light environment

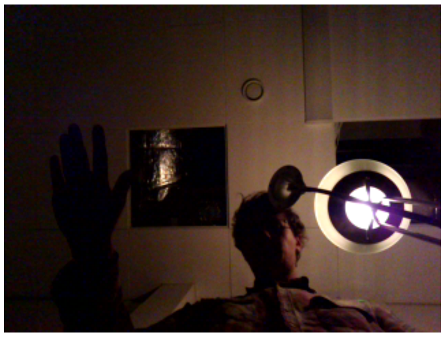

Figure 20: Shows an image sample from the data set ”real_hand” in a low light environment

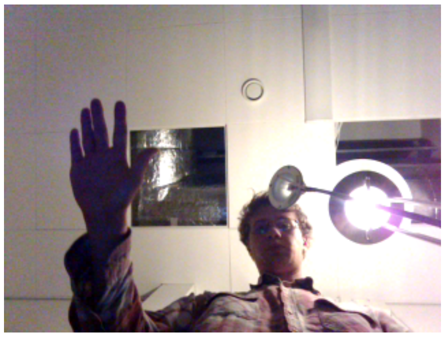

Figure 21: Shows an image sample from the data set ”real_hand” in a light environment

5.1 Examination of cascade leaves

Leaf values from different cascades were examined in order to compare them. The idea was to investigate if it is meaningful to use different estimates for standard deviations depending on stage and position in the weak trees. The position within the weak-tree is given by the sub-index in (37). Standard deviations of leaf values are used by the adaption algorithm, see the linear programming methods in section 3.2.4.

In this section, a cascade from an object detection algorithm (such as D) for convenience is denoted by the name of the corresponding algorithm.

Leaves distribution of cascade D

In order to access the overall structure of the leaf data from D, histograms of all leaf values from D and leaf values from all different stages of D were examined. Figure 22 contains the histogram of all leaf values. Figure 23 exhibits the histogram of all leaf values from a late stage. Figure 24 shows the histogram of all leaf values from an early stage. The reason not all histograms examined are included in the report as figures is that they all had very similar shapes.

These results give the impression that for all stages and the cascade in general the leaf values are gathered in three ordered groups, "left", "right" and "middle", possibly with different underlying distributions.

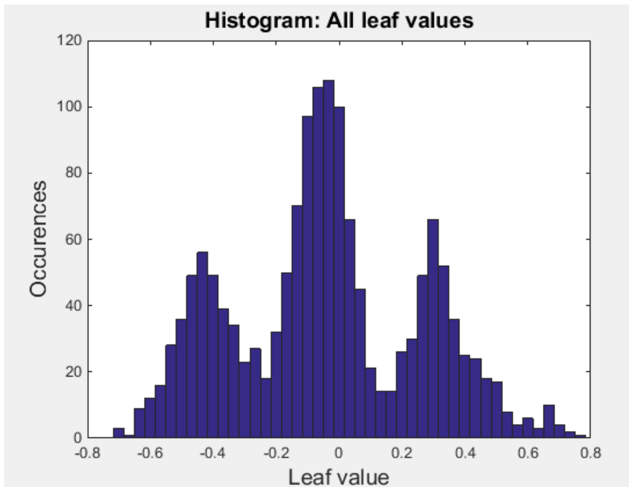

Figure 22: A histogram of all leaf values from cascade D.

In order to get a better understanding of the distribution of different types of leaves, histograms of leaf values from leaves that are hanging from the "right side" of the parent node (feature) and histograms of their left siblings were compared.

Histograms with leaves from the two different sides were examined once for every stage of D and once more for the joint set of all the leaves from D, see figure 25.

Figure 25 clearly shows two well defined peaks for each of the "leaf-side types", note the small peak to the right in the rightmost histogram. When histograms from figure 25 are compared with figure 22 it is obvious that the middle peak, seen in the latter, results from the overlap of the two histograms. The same pattern also appeared for every stage in D, except the very first, which actually doesn't have enough leaves to visually give any structure.

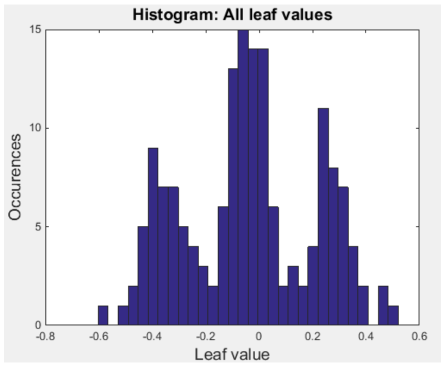

Figure 23: A histogram of all leaf values from a late stage in cascade D.

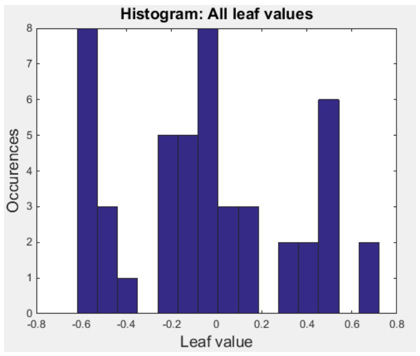

Figure 24: A histogram of all leaf values from an early stage in cascade D.

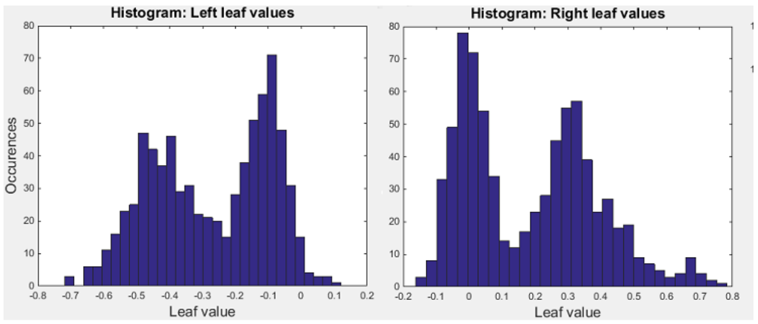

Figure 25: All leaf values from D arranged in two histograms after "side type".

Leaves from other cascades

Other cascades with the same tree depth were also examined in the same manner as done with D, which in all cases led to similar results.

All ten cascades had their mean values and standard deviations calculated for each stage. All stages had their leaves once again divided into different sets depending upon the position of the leaf relative to its parent. For each such class of leaves, the standard deviation and mean value were calculated.

5.2 Discussion

Results obtained by analyzing the stage leaves from cascade classifiers A, B, C, D, suggest that it is meaningful to divide the leaves by stage and position in its weak tree in order to estimate the standard deviation. Within a stage, the position of a leaf w_i^t is given by its sub-index i. See equation (37).

The stage dependency becomes evident when comparing histograms of all leaves from a cascade with the histograms for a specific stage, as the histograms of all cascade leaves sometimes contain more modes (peaks) than leaf-classes. Compare the histograms of figure 25 with figure 23.

The leaf class dependency within each stage is indicated by comparing histograms of stage leaves to histograms showing all classes of stage leaves, where the modes match the number of leaf classes. This pattern was observed in all (non initial) stages within all cascades. See figure 23 and 24.

However, since all cascades investigated were trained with similar types of learning procedures this becomes a weak point in the analysis.
In order to avoid over-complication of the model, the different estimates were only deployed by the leaf update procedures depending on "side type" of a leaf's parent feature (if i is odd or even). Another advantage with the side type approach is that it gives more data for each estimate. This becomes more important for the earlier stages, as they have a smaller number of weak classifiers.

To at least challenge this simpler model, all **ten** cascades had their mean value and standard deviation estimates compared for each stage and class. Two conclusions could be drawn. For every cascade and stage:

- Every right class had a positive mean value estimate $\hat{\mu} > 0$.
- Every left class had a negative mean value estimate $\hat{\mu} < 0$.

For further investigation it would be interesting to examine leaf values from cascades with different types of learning procedures (boosting algorithms) and perform some statistical hypothesis testing.

5.3 Object detection parameters

Since the parameter spaces for the object detection and adaption algorithms are quite large, see section 3.2, some of the parameters were chosen in a perfunctory manner (some choices were simply qualified guesses) and kept constant throughout the testing.

The parameters that are, more or less, chosen perfunctorily in the testings are listed below:

Preprocessing:
The preprocessing followed the same procedure and order throughout the thesis work.

Detect Multi Scale:
In the object detection algorithms multiscale search (see algorithm 5), the scale factor is set to f.

Clustering Algorithm:
While clustering candidate regions (output rectangles, see algorithm 6), the grouping threshold and the similar parameter were set to $\theta = 3$ and $\epsilon = 0.2$. These are default choices by OpenCV.

Collect method:
The collection method used to collect data from the elusive environment were chosen as the **Collect method 3** (Cluster members), see section 3.2.3.

Leaf update procedures:
The leaf update procedures to be tested are the linear programming methods in section 3.2.4.

5.4 Event counting with different cascades

Events in Real_hand
The data-set "real_hand" consists of a total of seven sequences $\{S_1, ..., S_7\}$.

These were examined with two windows W_1, W_2 of images with sizes $|W_1| = 3, |W_2| = 5$ and the four cascade based object detection algorithms described in section 4.1.

	Sequences							Tot
	S_1	S_2	S_3	S_4	S_5	S_6	S_7	
$\|S_i\|$	796	979	189	591	984	851	979	5369
A	18	6	3	14	11	33	14	99
B	11	5	4	14	16	22	16	88
C	22	7	6	16	26	22	1	100
D	17	9	6	17	15	15	0	79

Table 2: Number of events of order 1 recorded in the data set **Real_hand**

	Sequences							Tot
	S_1	S_2	S_3	S_4	S_5	S_6	S_7	
$\|S_i\|$	796	979	189	591	984	851	979	5369
A	7	2	2	9	4	11	4	39
B	4	2	2	6	6	7	7	34
C	12	4	2	9	6	9	1	43
D	6	0	2	6	1	6	0	21

Table 3: Number of events of order 2 recorded in the data set **Real_hand**

The examination of the "real_hand" data set consisted of counting the number of occurrences of the event *pMissed* of order 1 and 2 with sliding windows (of size 3 and 5), (see definition 11), with each type of classifier. The resulting number of events found with each classifier is presented in table 2 and table 3.

During the event collection, the only collected **true negative** was collected, with a window of size 3, by cascade B from **Real_Hand**. All other triggered events were due to **true negatives**. See figure 16.

Events in Photo_of_hand

Event occurrences in **Photo_of_hand** were counted by the same procedure used for **Real_hand** and consisted of a total of four sequences, $\{S_1, ..., S_4\}$.

5.5 Tuning the adaption algorithm on D

In order to exploit the data as much as possible, cross validation is used on the combined data set consisting of both **Photo_of_hand** and **Real_hand**.

Cross-validation (or rotation estimation) here refers to the process of collecting adaptive data on every sequence except some selected test sequence (or sequences) where the adapted cascade is bench-marked instead. In this thesis cross-validations were performed such that the all sequences except one were used to adapt the classifier. In the end, every sequence was used as a test-sequence exactly once.

The adaption is performed by reclassification of low confidence candidates, by using the **score updating function** (53), with a set of adapted leaves. A very naive variant of this approach would be to reclassify all

	Sequences				Tot
	S_1	S_2	S_3	S_4	
$\|S_i\|$	513	962	390	642	2507
A	3	8	1	3	15
B	3	2	1	0	6
C	1	22	0	5	28
D	1	14	3	10	28

Table 4: Number of events of order 1 recorded in the data set **Photo_of_hand**

	Sequences				Tot
	S_1	S_2	S_3	S_4	
$\lvert S_i \rvert$	513	962	390	642	2507
A	0	6	1	3	10
B	2	2	1	0	5
C	0	7	0	2	9
D	0	6	3	7	16

Table 5: Number of events of order 2 recorded in the data set **Photo_of_hand**

	Cascades			
	D	C	B	A
Single True Positives:	3721	4138	5317	5308
False positives:	0	3	4	0
Double detections:	2	4	43	16
Feature evaluations (10^8):	3.950	3.866	8.081	10.396

Table 6: Performance of the original cascades on the joint data set

low confidence regions as positives by using

$$S_{\text{naive}}(\mathbf{x}) = \begin{cases} \text{positive} & \text{if} \quad |S(\mathbf{x}) - \theta| < \epsilon_r \sigma \\ S(\mathbf{x}) & \textit{otherwise}, \end{cases} \tag{54}$$

instead of (53).

In order to motivate the use of the adapted leaves, retrieved by the methods within this thesis, instead of the naive approach (opening holes in the cascades) the results must at least be different. This can be seen as applying Occam's razor to the adaption problem.

The main idea behind these tests (section 5.5) is to measure the **positive-rate**, see section 2.4.2, and tune the adaption method such that its behaviour is differentiated from the **naive approach**.

5.5.1 Adaption Results 1

The first results, obtained by experimenting with different upper bounds $q > 0$, number of stages to adapt and different box constraints for the leaf update procedure (see algorithm 9), are shown in tables 7, 8 and 9.

The other parameters were chosen as:

- LP-method: $\mathcal{P}_1$.
- Reclassification area: $\epsilon_r = 0.4$.

5.5.2 Adaption Results 2

Results from experimenting with different sizes of the reclassification area are presented in tables 10 and 11. These were performed with some conjectured parameter choices obtained from 5.5.1 which are listed below for further experimenting:

- Box constraints: $(-\epsilon_l, \epsilon_u) = (-0.1, 0.1)$.

Type	q	Detections (%)	f_evals (%)
Adapted	0	-2.39	$+0.35$
Adapted	2	$+7.50$	$+0.56$
Adapted	4	$+8.14$	$+0.60$
Adapted	8	$+8.20$	$+0.61$
Adapted	> 16	$+8.20$	$+0.62$
Naive	-	$+8.20$	$+0.18$

Table 7: Adaption of the three last stages in D with varying $q > 0$. The parameters were: $\epsilon_r = 0.4$, $(\epsilon_u, \epsilon_l) = (1, 0.1)$ and LP-method $\mathcal{P}_1$

	Detections (%)		f_evals (%)	
Stages	Adapted	Naive	Adapted	Naive
3	$+7.50$	$+8.20$	$+0.62$	$+0.18$
6	$+13.84$	$+16.47$	$+1.72$	$+1.01$
9	$+16.85$	$+22.01$	$+2.98$	$+2.51$

Table 8: Adaption of D with varying number of stages. The parameters were: $q = 2$, $\epsilon_r = 0.4$, $(\epsilon_u, \epsilon_l) = (1, 0.1)$ and LP-method $\mathcal{P}_1$

Box		Performance	
$-\epsilon_l$	ϵ_u	Detections (%)	f_evals (%)
-0.1	0.1	$+8.12$	$+0.60$
-0.1	0.5	$+7.85$	$+0.57$
-0.1	1	$+7.50$	$+0.56$
-1	1	$+4.14$	$+0.44$
-1	10	-3.01	$+0.33$
-10	10	-8.65	$+0.18$
Naive		$+8.20$	$+0.18$

Table 9: Adaption of the three last stages in D with varying box constraints. The parameters were: $q = 2$, $\epsilon_r = 0.4$ and LP-method $\mathcal{P}_1$

	Detections (%)		f_evals (%)	
ϵ_r	Adapted	Naive	Adapted	Naive
0.4	+8.12	+8.20	+0.60	+0.18
0.8	+13.30	+13.65	+1.14	+0.28
1.6	+16.42	+17.66	+1.93	+0.35
3.2	+16.66	+18.36	+3.43	+0.36

Table 10: Adaption of the three last stages in D with varying reclassification margins (i.e. $\epsilon_r \sigma$). The parameters were: $q = 2$, $\epsilon_u = \epsilon_l = 0.1$ and LP-method $\mathcal{P}_1$

	Detections (%)		f_evals (%)	
ϵ_r	Adapted	Naive	Adapted	Naive
0.2	+10.86	+10.99	+2.08	+1.28
0.4	+21.10	+22.01	+4.08	+2.51
0.6	+28.57	+30.88	+5.97	+3.68

Table 11: Adaption of the nine last stages in D with varying reclassification margins (i.e. $\epsilon_r \sigma$). The parameters were: $q = 2$, $\epsilon_u = \epsilon_l = 0.1$ and LP-method $\mathcal{P}_1$

- Impact constrain: $q = 2$.

5.5.3 Adaption Results 3

Results shown in table 12 and 13 are obtained by repeating the experiments presented in table 10 and 11, with the difference being the constrain in the optimization algorithm used by the leaf update procedure, i.e. the **inequality adaption** method $\mathcal{P}_2$ was used instead of $\mathcal{P}_1$.

5.5.4 Adaption Results 4

A method ($\mathcal{P}_3$) that uses an extra constrain,

$$\mathbf{I}_\gamma{}^T \mathbf{\Delta w} \geq 0,$$

in the optimization algorithm was introduced and tested for different reclassification areas and number of stages. The results are presented in tables 14, 15 and 16.

5.5.5 Adaption Results 5

In table 17 results from comparing $\mathcal{P}_2$ and $\mathcal{P}_3$ are presented.

5.6 General results and discussion

Results from experiments 1 to 5, only concern positive rates and computational aspects. To actually benchmark the adaption, the results need to be overviewed in terms of true positives and false positives.

	Detections (%)		f_evals (%)	
ϵ_r	Adapted	Naive	Adapted	Naive
0.4	+7.71	+8.20	+0.58	+0.18
0.8	+7.71	+13.65	+1.01	+0.28
1.6	+7.71	+17.66	+1.90	+0.35

Table 12: Adaption of the three last stages in D with inequality constraints ($\mathcal{P}_2$). Other parameters were: $q = 2$ and $\epsilon_u = \epsilon_l = 0.1$

	Detections (%)		f_evals (%)	
ϵ_r	Adapted	Naive	Adapted	Naive
0.2	+10.86	+10.99	+2.08	+1.28
0.4	+19.16	+22.01	+3.71	+2.51
0.6	+19.16	+30.88	+4.71	+3.68

Table 13: Adaption of the nine last stages in D with inequality constraints ($\mathcal{P}_2$). Other parameters were: $q = 2$, $\epsilon_u = \epsilon_l = 0.1$

	Detections (%)		f_evals (%)	
ϵ_r	Adapted	Naive	Adapted	Naive
0.2	+4.49	+4.49	+0.30	+0.10
0.4	+7.71	+8.20	+0.58	+0.18
0.8	+7.71	+13.65	+1.01	+0.28

Table 14: **Edge adaption** ($\mathcal{P}_3$) of the three last stages in D with varying reclassification margins (i.e. $\epsilon_r\sigma$). Other parameters were: $q = 2$ and $\epsilon_u = \epsilon_l = 0.1$

	Detections (%)		f_evals (%)	
ϵ_r	Adapted	Naive	Adapted	Naive
0.2	+8.38	+10.99	+1.36	+1.28
0.3	+11.58	+16.72	+1.95	+1.90
0.4	+13.81	+22.01	+2.46	+2.51
0.6	+13.81	+30.88	+3.33	+3.68

Table 15: **Edge adaption** ($\mathcal{P}_3$) of the nine last stages in D with varying reclassification margins (i.e. $\epsilon_r\sigma$). Other parameters were: $q = 2$, $\epsilon_u = \epsilon_l = 0.1$

	Detections (%)		f_evals (%)	
Stages	Adapted	Naive	Adapted	Naive
3	+7.71	+8.20	+0.58	+0.18
6	+13.33	+16.47	+1.71	+1.01
9	+13.81	+22.01	+2.46	+2.51

Table 16: **Edge adaption** ($\mathcal{P}_3$) of D with varying number of stages. Other parameters were: $q = 2$ and $\epsilon_r = 0.4$, $(\epsilon_u, \epsilon_l) = (0.1, 0.1)$

	Detections (%)			f_evals (%)		
Stages	Edge	Ineq	Naive	Edge	Ineq	Naive
3	+7.71	+7.71	+8.20	+0.58	+0.58	+0.18
6	+13.33	+14.97	+16.47	+1.71	+1.90	+1.01
9	+13.81	+19.16	+22.01	+2.46	+3.71	+2.51
12	+13.63	+20.91	+25.67	+3.39	+5.92	+5.02

Table 17: **Edge** and **Inequality** - adaption of D with varying number of stages. Other parameters were: $q = 2$ and $\epsilon_r = 0.4$ and $(\epsilon_u, \epsilon_l) = (0.1, 0.1)$

Cascade Configuration	Detection events			
	TP single	TP double	False Positives	f_evals (%)
$D_{Original}$	3721	2	0	+0.00
$D_{Adapted}$	4218(+13.36%)	2	2	+1.71
D_{Naive}	4332(+16.42%)	3	2	+1.01

Table 18: Detailed results from adaption on 6 stages of D. Stable: $q = 2$ and $\epsilon_r = 0.4$, $(\epsilon_u, \epsilon_l) = (0.1, 0.1)$, edge and inequality constraint

Cascade Configuration	Detection events			
	TP single	TP double	False Positives	f_evals (%)
$C_{Original}$	4138	4	3	+0.00
$C_{Adapted}$	4616(+11.55%)	9	7	+1.65
C_{Naive}	4796(+15.82%)	8	13	+1.19

Table 19: Detailed results from adaption on 6 stages of D. Stable: $q = 2$ and $\epsilon_r = 0.4$, $(\epsilon_u, \epsilon_l) = (0.1, 0.1)$, edge and inequality constraint

The configurations used to adapt and test the cascades were chosen as method $\mathcal{P}_3$ along with parameters:

- Impact constraint: $q = 2$.
- Box constraints: $(-\epsilon_l, \epsilon_u) = (-0.1, 0.1)$.
- Reclassification area: $\epsilon_r = 0.4$.

The reason being that the method $\mathcal{P}_3$ appears to converge faster, with respect to both the reclassification area and the number of stages to adapt for, than the other linear programming methods.

These configurations are then bluntly applied to cascades C, B and A. Tables 20, 21, 19 and 18 contain the more detailed results.

5.6.1 Analysis of the adaption results

The adaption procedures generated additional true positives on the joint data-set **Real_Hand** and **Photo_of_Hand**. However, some extra false positives emerged for the **Real_Hand**. Unlike **Real_Hand**, the classifier's performance on **Photo_of_Hand** is very stable as the results from the original cascades, the adapted ones and the naive approaches did not differ with any significance.

When the adaption algorithm was tested against the naive approach (to simply "let through" all candidates that were up for reclassification) the latter generated more false positives. Because the naive approach and the adapted cascades do not have the same detection rate it is hard to draw any conclusion whether one is beneficial to the other, i.e. if it is assumed that no method has an edge over the other, it would still be anticipated that the naive approach would yield more false positives, since the detection rates for "naive" are also higher than for the adapted cascade.

Cascade Configuration	Detection events			
	TP single	TP double	False Positives	f_evals (%)
$B_{Original}$	5317	43	4	+0.00
$B_{Adapted}$	5536(+4.1%)	46	22	+2.25
B_{Naive}	5560(+4.6%)	48	29	+1.22

Table 20: Detailed results from adaption on 6 stages of B. Stable: $q = 2$ and $\epsilon_r = 0.4$, $(\epsilon_u, \epsilon_l) = (0.1, 0.1)$, edge and inequality constraint

Cascade	Detection events			
Configuration	TP single	TP double	False Positives	f_evals (%)
$A_{Original}$	5308	16	0	+0.00
$A_{Adapted}$	5582(+5.16%)	20	3	+3.28
A_{Naive}	5767(+8.65%)	30	20	+2.94

Table 21: Detailed results from adaption on 6 stages of A. Stable: $q = 2$ and $\epsilon_r = 0.4$, $(\epsilon_u, \epsilon_l) = (0.1, 0.1)$, edge and inequality constraint

Both methods show that it is feasible in terms of feature evaluations to use the approaches in real time.

The hypothesis that any environment beneficial adaption is going to raise the training error, and maybe the general error as well, is not tested or rejected in this thesis work.

Future work

The use of *ROC*-curves would provide the user with a good overview of the general performance. For example, with a *ROC* curve it is easy to compare the classifiers' performances at the same detection rate.

Furthermore, in order to benchmark the adaption algorithm's ability to adapt a cascade classifier to an environment, an environment biased data set needs to be applied. Preferably the biased data set would be equipped with more false positives than **Real_Hand**. At this point it is unclear to me if there exists any standard database for the "open hand" poses in low light, but I guess not.

In order to get an overview of the adapted classifier's performance with the methods used in this thesis, one can first tune in the naive classifier to the same detection rate as the adapted one and then compare them to each other. This would yield comparable results.

5.6.2 Analysis of leaf update procedures

The different LP-update procedures, even though they tend to look somewhat similar, do behave differently. The first observation is that LP-Method $\mathcal{P}_1$, with an equality constraint for the target frame score, has slower convergence of the detection rate with respect to the size of the reclassification area ϵ_r than $\mathcal{P}_2$ and behaved more as the **naive approach**, described by (54), than the other LP-update procedures.
Tables 10 and 11 show the slowly converging positive rate from $\mathcal{P}_1$.

Tables 12 and 13 reveal convergence of the positive rate as early as $\epsilon_r = 0.4$ with the same target impact $q = 2$. By exchanging the equality constrain with an inequality constrain at $\epsilon_r = 0.4$, the positive rate also slightly dropped.

To find possible explanations for the different rates of convergence, the value of the expression $\mathbf{I}^T\mathbf{\Delta w}$ for the optimization solutions for the method used in table 12 was investigated.
The first observation suggested that the different convergence rates were not caused by solutions to $\mathcal{P}_2$ having inactive constrains. Table 12 demonstrates that the solutions to $\mathcal{P}_2$ obtained from Matlab's LP solver (linprog) yielded active constrains. This implies that the solutions to both $\mathcal{P}_1$ and $\mathcal{P}_2$ had active constrains with different convergence rates.

Another observation was that the $\mathbf{I_R}^T\mathbf{\Delta w}$ stood for the larger part of contribution to the gained frame score mass, $\mathbf{I}^T\mathbf{\Delta w}$.

The same did not apply for earlier (and simpler) stages investigated from experiments presented in table 12. The constrains were still active in the more shallow stages, however the contribution from the non interesting

Stage	Method-$\mathcal{P}_2$			Method-$\mathcal{P}_3$			
	$\mathbf{I_R}^T\mathbf{\Delta w}$	$\mathbf{I}^T\mathbf{\Delta w}$	$\mathbf{I}_\gamma{}^T\mathbf{\Delta w}$	$\mathbf{I_R}^T\mathbf{\Delta w}$	$\mathbf{I}^T\mathbf{\Delta w}$	$\mathbf{I}_\gamma{}^T\mathbf{\Delta w}$	$\lvert\lvert\mathbf{\Delta w}_1 - \mathbf{\Delta w}_2\rvert\rvert$
n	1.84	2	1.68	1.84	2	1.68	0
$n-1$	1.65	2	1.3	1.65	2	1.3	0
$n-2$	1.48	2	0.97	1.48	2	0.97	0
$n-3$	1.37	2	0.74	1.37	2	0.74	0
$n-4$	1.26	2	0.5	1.26	2	0.5	0
$n-5$	1.15	2	0.3	1.15	2	0.3	0
$n-6$	0.99	2	-0.02	0.97	1.93	0	$\neq 0$
$n-7$	0.94	2	-0.12	0.89	1.78	0	$\neq 0$
$n-8$	0.92	2	-0.15	0.84	1.69	0	$\neq 0$

Table 22: The LP quantities from adaption behind results in table 15 and 13

regions $\mathbf{I_{Rc}}^T\mathbf{\Delta w}$ became more prominent.

A possible explanation lies within the nature of the problem, to isolate leaves associated with missed detection, and the collection procedure. As the earlier stages are designed to reject large amounts of negative samples, most leaves handle large amounts of negative data and therefore they are probably not well suited for leaf adaption.

The second reason is that the collection procedure used throughout all experiments was the "Collect method 3: Cluster members" (see section 3.2.3) which has the property that the number of members of *ROI* is independent of the stage number. This leaves the members in ROI^c as the only stage variable. As the number of ROI^c members in C is monotonically increasing with descending stage index (the first stage has index 0), *ROI* candidates become more sparse in the earlier stages. As each candidate region evaluates all weak trees in a reached stage, thus always picking the same number of leaves, it becomes harder and harder to find leaves that are more associated with *ROI* than the complement.

To capture this fact, LP-method $\mathcal{P}_3$ was developed. The idea is to try to force the adaption to be weighted in favour of *ROI* and if not, avoid to adapt. The goal was reached as the method produced the same leaf values as long as the extra constrain $\mathbf{I}_\gamma{}^T\mathbf{\Delta w} \geq 0$ was inactive. When $\mathbf{I}_\gamma{}^T\mathbf{\Delta w} = 0$ the overall impact of the adaption $\mathbf{I}^T\mathbf{\Delta w}$ was compromised in order to ensure that *ROI* is contributing to at least half of the impact. See table 22.

$\mathcal{P}_3$ shows convergence for the positive-rate with respect to both the classification area and the number of stages to adapt for, see tables 14 and 15.

Future work

As the leaf values produced by methods $\mathcal{P}_2$ and $\mathcal{P}_3$ (**inequality** and **edge-adaption**) both seemed to imply fast convergence of the positive-rates with respect to the reclassification area (ϵ_r) it would be safe to discard the reclassification approach for them (see definition 13) and instead just apply the new leaf increments. This would save some computational power.

For future work, it would be interesting to develop more exotic leaf update procedures and test them, together with the ones presented here, for more configurations and different collect-methods.

References

[1] URL: `https://en.wikipedia.org/wiki/Photoreceptor_cell`.

[2] URL: https://en.wikipedia.org/wiki/RGB_color_space.

[3] URL: https://en.wikipedia.org/wiki/Histogram_equalization.

[4] URL: http://www.math.uci.edu/icamp/courses/math77c/demos/hist_eq.pdf.

[5] URL: http://docs.opencv.org/2.4/doc/tutorials/imgproc/histograms/histogram_equalization/histogram_equalization.html.

[6] URL: https://en.wikipedia.org/wiki/Summed_area_table.

[7] URL: https://en.wikipedia.org/wiki/Feature_detection_(computer_vision).

[8] URL: https://en.wikipedia.org/wiki/Histogram_of_oriented_gradients.

[9] URL: https://en.wikipedia.org/wiki/Haar-like_features.

[10] URL: https://en.wikipedia.org/wiki/Local_binary_patterns#cite_note-4.

[11] URL: http://fileadmin.cs.lth.se/graphics/theses/projects/facerecognition/1_all_haar_wavelets.png.

[12] URL: https://en.wikipedia.org/wiki/Statistical_classification.

[13] URL: http://blog.echen.me/2011/04/27/choosing-a-machine-learning-classifier/.

[14] URL: https://en.wikipedia.org/wiki/Naive_Bayes_classifier.

[15] URL: http://en.wikipedia.org/wiki/Nearest_neighbour_classifiers.

[16] URL: https://en.wikipedia.org/wiki/Decision_tree_learning.

[17] URL: https://en.wikipedia.org/wiki/Support_vector_machine.

[18] URL: http://documents.software.dell.com/Statistics/Textbook/Naive-Bayes-Classifier.

[19] URL: http://www.cs.ucr.edu/~eamonn/CE/Bayesian%20Classification%20withInsect_examples.pdf.

[20] URL: www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm.

[21] URL: http://www.saedsayad.com/decision_tree_overfitting.htm.

[22] URL: http://docs.opencv.org/2.4/doc/tutorials/ml/non_linear_svms/non_linear_svms.html.

[23] URL: https://en.wikipedia.org/wiki/Robert_Schapire.

[24] URL: https://en.wikipedia.org/wiki/AdaBoost.

[25] URL: http://cseweb.ucsd.edu/classes/sp12/cse151-a/lecture12-final.pdf.

[26] URL: http://math.mit.edu/~rothvoss/18.304.3PM/Presentations/1-Eric-Boosting304FinalRpdf.pdf.

[27] URL: http://docs.opencv.org/2.4/doc/user_guide/ug_traincascade.html#viola2001.

[28] 2014. URL: https://www.youtube.com/watch?v=UHBmv7qCey4.

[29] Léon Bottou and Chih-jen Lin. *Support Vector Machine Solvers.* 2006.

[30] Léon Bottou and Chih-jen Lin. *Support Vector Machine Solvers, p 3–4.* 2006.

[31] Jo Chang-yeon. *Face Detection using LBP features.* 2008.

[32] Tom Fawcett. "ROC Graphs: Notes and Practical Considerations for Researchers". In: (2004).

[33] Yoav Freund and Robert E. Schapire. *A Short Introduction to Boosting.* 1999.

[34] Mike Gashler, Christophe Giraud-carrier, and Tony Martinez. *Decision Tree Ensemble: Small Heterogeneous Is Better Than Large Homogeneous.*

[35] Abdenour Hadid and Senior Member. "Face description with local binary patterns: Application to face recognition". In: *IEEE Trans. Pattern Analysis and Machine Intelligence* (2006).

[36] Vidit Jain, Yahoo Labs Bangalore, and Erik Learned-miller. *Online Domain Adaptation of a Pre-Trained Cascade of Classifiers.*

[37] Michael Kearns. *Thoughts on Hypothesis Boosting*. 1988. URL: `http://www.cis.upenn.edu/~mkearns/papers/boostnote.pdf`.

[38] Rainer Lienhart, Er Kuranov, and Vadim Pisarevsky. "Empirical Analysis of Detection Cascades of Boosted Classifiers for Rapid Object Detection". In: *In DAGM 25th Pattern Recognition Symposium*. 2003, pp. 297–304.

[39] Philip M. Long and Rocco A. Servedio. *Random Classification Noise Defeats All Convex Potential Boosters*.

[40] Llew Mason et al. *Boosting Algorithms as Gradient Descent*. 2000.

[41] Brendan Mccane et al. *Optimizing Cascade Classifiers*.

[42] Ron Meir and Gunnar Rätsch. "An introduction to boosting and leveraging". In: *Advanced Lectures on Machine Learning, LNCS*. Springer, 2003, pp. 119–184.

[43] Danny Pascale. "A review of RGB color spaces". In: (). URL: `http://www.babelcolor.com/download/A%20review%20of%20RGB%20color%20spaces.pdf`.

[44] Dr Noureddin Sadawi. URL: `https://www.youtube.com/watch?v=XcwH9JGfZOU`.

[45] Robert E. Schapire. *A Brief Introduction to Boosting*. 1999.

[46] Robert E. Schapire. *Explaining AdaBoost*.

[47] Robert E. Schapire. "The strength of weak learnability". In: *Machine Learning*. 1990.

[48] V. Vapnik. *Principles of Risk Minimization for Learning Theory*. URL: `http://papers.nips.cc/paper/506-principles-of-risk-minimization-for-learning-theory.pdf`.

[49] Paul Viola and Michael Jones. "Robust real-time face detection". In: *International Journal of Computer Vision* 57 (2004), pp. 137–154.

[50] Prof. Alan Yuille. URL: `http://www.stat.ucla.edu/~yuille/courses/Stat161-261-Spring14/LectureNotes7.pdf`.