UNIVERSITY OF CALIFORNIA
SANTA CRUZ

AUTOMATIC IMAGE ORIENTATION

A thesis submitted in partial satisfaction of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Aleksey Trofimov

December 2010

The Thesis of Aleksey Trofimov
is approved:

Professor Manfred K. Warmuth, Chair

Professor David P. Helmbold

Professor Alex Pang

Tyrus Miller
Vice Provost and Dean of Graduate Studies

Table Of Contents

List of Figures

Abstract

Automatic Image Orientation

by

Aleksey Trofimov

Suppose we have a set of correctly oriented images and a set of similar images that may or may not be correctly oriented. In this thesis we propose a framework for automatically orienting images in the second set. We describe how image orientation can be broken up into two problems of feature extraction and orientation prediction. We extract binary features and labels and then learn linear weights using machine learning algorithms as models for orientation prediction.

Using RGB pixel values we compute binary feature vector $\mathbf{x}_j \in \{-1, +1\}^n$ and label $y_j \in \{-1, +1\}$ for whether the orientation is correct. The classification algorithms learn a linear weight vector $\mathbf{w}$ that models the correct image orientation. The prediction is the orientation that maximizes the dot product of the feature vector $\mathbf{x}_j$ and parameter vector $\mathbf{w}$.

We investigated three different ways to extracting features: color presence, edge direction (binned EDH) and light-dark contrast. The results were labeled and used to compare the prediction accuracy of Perceptron, Boosting variants, Logistic Regression and SVM algorithms. With 628 outdoor photos and four orientations we achieved $\sim$93% prediction accuracy (compared to 25% guessing at random) using the best algorithm and a single set of features. By selecting 900 features from each feature set with AdaBoost and appending them together we increased the prediction accuracy to $\sim$99%.

Acknowledgments

I would like to thank the members of my reading committee for taking their time to evaluate my thesis. I would like to thank my advisor, Manfred Warmuth for teaching and guiding me, pushing me to accomplish as much as I could and always believing in me. I would like to thank David Helmbold for serving on my committee and providing valuable feedback. I would like to thank Alex Pang for taking the time out of his very busy schedule to serve on my reading committee. I would also like to thank Kohei Katano for his feedback about my understanding of boosting and introducing me to SoftRankBoost. I am very grateful to Robert Abbott, Maya Hristakeva, Samuel Johnson, Alex Lyzlov and Samantha Shireman for proofreading numerous versions of this thesis.

1 Introduction

There are settings where digital images are produced and not oriented or labeled with the correct orientation. Low-end digital cameras, scanned photos and cell-phone photos produce images that may not be oriented correctly. It is a tedious task to orient them manually. The goal of this thesis is to present a method for using computer vision and machine learning techniques to orient images automatically. We describe how image orientation can be broken up into two problems of feature extraction and orientation prediction. If we had the solutions to these two problems, we could orient images. We extract features from an image by looking at brightness levels (F_{HAAR}), edge directions (F_{EDGE}) and color presence (F_{COLOR}). We then use classification algorithms such as Perceptron, AdaBoost, AdaBoost*, MadaBoost, SoftRankBoost and Logistic Regression to learn models for predicting image orientation.

This thesis is structured in the following way: In the introduction we give an overview and methodology for our method. In section 2 we cover the feature extraction algorithms in some detail and move to the machine learning algorithms in section 3. Section 4 gives an overview of work related to image orientation and classification. We dedicate a section to implementation notes followed by a look at parameter tuning in section 6. Section 7 gives the results of running algorithms on features described earlier. We conclude the thesis with a look at some partial work and a future work sections.

1.1 Notation and definitions

1.1.1 Common Terms

Dot Product - Sum of products of individual elements of two vectors. It is a measure of how similar two vectors are.

Confidence - Measure of how confident an orientation prediction is.

Rejection - Abstaining from orienting an image based on low confidence.

Accuracy - Orientation prediction accuracy. The fraction of images oriented correctly out of the total number of images that are not rejected.

Rejection Curve - A plot of prediction accuracy vs the percent of images that can be rejected.

Spatial Pyramid - A partitioning of an image into increasingly fine sub-regions (cells).

Grid - A single level of a spatial pyramid.

Feature - A single real-valued piece of information.

Decision stump - A piece of information that only has two values ($+1$ or -1). It is computed by checking whether a feature is larger than some threshold.

Binary Feature Vector - A vector of decision stumps. Just 'feature vector' in this thesis.

Feature Set - Feature vectors computed from a single feature type.

Label - An indicator for whether a feature vector describes a sample with correct orientation (label $= +1$) or not (label $= -1$).

RGB - Red, Green and Blue values representing a color in RGB color space.

Channel - A double array of pixel values for a single color like Red (C_{RED}).

Tuning - Trying a range of parameter values and picking an optimal based on prediction accuracy.

SubTrain Set ($K_{SUBTRAIN}$) - Set of images to train on to tune parameters.

Validation Set ($K_{VALIDATION}$) - Set of images to test on to tune parameters.

Train Set (K_{TRAIN}) - Both SubTrain and Validation Sets ($K_{SUBTRAIN} \cup K_{VALIDATION}$). The set of images to train on to make a model for testing.

Test Set (K_{TEST}) - Set of images to test a model on to report the results.

Experiment - A single execution learning on $K_{SUBTRAIN}$ and recording the prediction accuracy on K_{TEST} .

1.1.2 Overview

We took 628 outdoor photos from a private collection and split them into two sets: 471 image for training and 157 images for testing. We worked with images at a resolution of 360×360 pixels. We rotated each image 4 ways $(0°, 90°, -90°, 180°)$. For each orientation $q \in Q(4)$ of image i, we computed a binary feature vector $\mathbf{x}_i^q \in \{-1, +1\}^d$ and assigned it a label $y_i^q = +1$ if q is the index of the correct orientation and -1 otherwise. Since all the images in the training set are correctly oriented the first orientation is the correct one (i.e. $\tilde{q} = 1$). So for each image i we computed 4 feature vectors $\mathbf{x}_i^1 \ldots \mathbf{x}_i^Q$ with labels $+1, -1, \cdots -1$ respectively. Combining $\mathbf{x}_i^q$ with its label y_i^q we get tuples $(\mathbf{x}_j, y_j)$ as examples for the machine learning algorithms to train on.

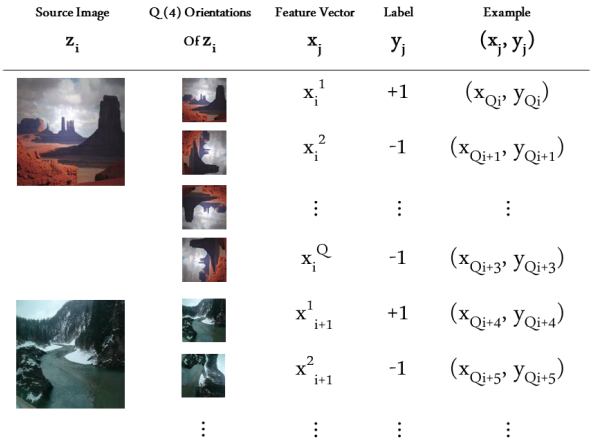

Figure 1: Overview of generating examples from images. Note that a binary feature vector for image i with orientation q is the same as sample $Qi+q-1$ (i.e. $\mathbf{x}_i^q = \mathbf{x}_{Qi+q-1}$).

If we are training on m images with Q orientations, we have a total of $n = mQ$ examples. The tuples $(\mathbf{x}_j, y_j)$ are organized by putting all labels into a vector $\mathbf{y}$ and all the training samples into a matrix X (see figure 2). From them, we can learn a linear weight vector $\mathbf{w} \in R^d$ that we use to predict orientation. Given a new

image z_k, the q possible orientations are ranked via a dot product with $\mathbf{w}$ and the one with the highest score is the classifier's prediction (i.e. $\hat{q}_k = \arg\max_{q \in Q} \mathbf{x}_k^q \cdot \mathbf{w}$). The prediction is correct if $\hat{q} = \tilde{q}$ (i.e. $y_{Qi+\hat{q}-1} = +1$) and wrong otherwise.

1.1.3 Feature Extraction

We are given an image in three two-dimensional arrays (matrices) C_{RED}, C_{GREEN}, C_{BLUE} which contain the channel information for red, green and blue respectively. The value in each channel ranges from 0 to 255 for each pixel. The dimension of the image is $s \times s$ pixels. For the spatial pyramid (described in section 1.1.5) we vary the size of the grid from $B_S \times B_S$ to $B_B \times B_B$ when we compute the features. For each cell (region) we compute a local feature vector f_c such that $f_c(i) \geq 0$. A side of each cell is $\sigma = \frac{s}{B}$ pixels where B is the size of the grid at current level of spatial pyramid. All the f_c appended together make F, a feature vector containing all the features in the spatial pyramid. F has values in $\mathbb{Z}^+$. To convert F to a binary feature vector $\mathbf{x}_j$ we use a threshold of $\theta = \frac{1}{2}\frac{\sum_i F_i}{d}$ (half of the mean value).

1.1.4 Machine Learning

We are given n feature vectors. Each sample $\mathbf{x}_i^q$ is a binary feature vector of length d computed from image i with orientation q. For each sample j we are given a label y_j that is $+1$ if this orientation is upright and -1 otherwise. A classification algorithm is allowed to run up to T iterations [of the outer-most loop] to learn the linear weight vector $\mathbf{w}$ which is used to predict the orientation of an image.

The data is laid out in the following way: $\mathbf{y}$ is a vector of length n where each value is y_j and X is an $n \times d$ matrix where row j is $\mathbf{x}_j$ and column k is h_k. The algorithms used in this thesis return a linear real valued weight vector $\mathbf{w}$ of length d. If $w_k > 0$ it indicates that F_k is correlated with correctly oriented samples.

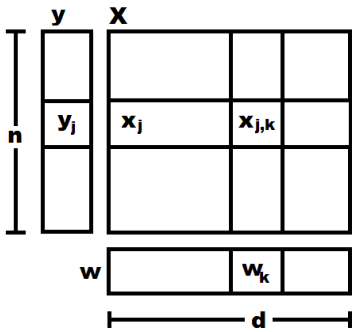

Figure 2: Data Layout with samples in a Matrix **X** and labels in a Vector **y**.

1.1.5 Spatial Pyramid

Computer vision techniques use spatial features [2, 19, 6, 21, 24, 25, 27, 26, 30, 8, 32]. In our experiments we relied on the concept of a spatial pyramid [17] for feature organization. In image orientation the same colors and shapes will be present in all candidates (possible orientations). In order to predict the orientation correctly we must look at both presence of features and their locations. For that we divide each image into a $B \times B$ grid with B^2 cells and then look for features in each cell. What grid size to choose depends on the specific data and computational constraints. It has been shown that good results are achieved if a variety of grid sizes are used together [2, 17, 19]. If you imagine grids of increasing sizes stacked together in order, it will look like a pyramid, hence the name - spatial pyramid.

In this thesis, we computed the features in the context of a spatial pyramid. Instead of using a single grid, we used grids of sizes $B_S \times B_S \ldots B_B \times B_B$.

Figure 3: An example of a spatial pyramid with grid size = 2 . . . B.

1.2 Methodology

Our dataset is separated into two sets. K_{TRAIN} is a set of images to train and tune parameters on. K_{TEST} is a set of images on which to test in order to report results. All parameters that are fixed during testing were set using K_{TRAIN} alone. By dividing K_{TRAIN} into two subsets ($K_{SUBTRAIN}$ and $K_{VALIDATION}$) we were able to tune parameters by training on $K_{SUBTRAIN}$ and testing on $K_{VALIDATION}$.

1.2.1 Feature Extraction Parameter Tuning

Hardware constraints limited the size of the feature vectors we could use. Because of this we had to tune the number of features computed from each region vs the number of the regions.

We choose the optimal parameter values based on *prediction accuracy* = $\frac{\#\text{correctly oriented}}{\#\text{ images in } K_{VALIDATION}}$. Through experimentation we determined good base values for parameters. Then we tuned them one-by-one using the Average Perceptron algorithm (described in section 3.1). We chose it because of its efficiency and lack of parameters to tune. We made the assumption that features that work well for this simple algorithm work well for other machine learning algorithms.

1.2.2 Machine Learning Parameter Tuning

The machine learning algorithms described in this thesis learn a linear weight vector **w** that will classify the training data perfectly. Because of the high dimensionality of the feature vectors the data is linearly separable (can be perfectly separated by linear weights). Learning to separate all of $K_{SUBTRAIN}$ is not optimal because the set includes outliers. Over-fitting occurs when the machine learning algorithms learn how to separate the outliers; they fit noise instead of general patterns. In our experiments we use early stopping as a way to prevent over-fitting. Not letting machine learning algorithms run to convergence prevents **w** from being too custom-tailored to the training data.

In order to figure out a good number of iterations T (iterations of the outermost loops of the algorithm) we train on $K_{SUBTRAIN}$ and test on $K_{VALIDATION}$ every few iterations. We stop when the current iteration number t is twice as large as the optimal so far. To make sure that the value of T is not in large part due to a particular permutation of K_{TRAIN} , it is broken up into $K_{SUBTRAIN}$ and $K_{VALIDATION}$ several times. It is customary to only let each image in K_{TRAIN} be in $K_{VALIDATION}$ once. If this cross-validation is performed 5 times, then the size of $K_{VALIDATION}$ must be $\frac{1}{5}$th of the size of K_{TRAIN} and this will be a 5-fold-cross-validation. The optimal value of T is then the one that yields the highest prediction average accuracy during the cross-validation runs.

Some of the machine learning algorithms used in this paper also have parameters that must be fixed. To fix the parameters we used the images K_{TRAIN} and tuned them with 15-fold-cross-validation with feature-extraction parameters tuned beforehand.

1.2.3 Carrying Out Experiments and Reporting Results

Using the parameter values fixed with images in K_{TRAIN} we compared the performance of different feature sets and machine learning algorithms. For each pair, we tuned the value of T using cross-validation, then, used that value train on the entire K_{TRAIN} . We tested the resulting $\mathbf{w}$ on K_{TEST} and reported the resulting prediction accuracy.

Specifically, using 5-fold-cross-validation single experiment consisted of doing the following:

1. For run $= 1 \ldots 5$
   - Take $\frac{1}{5}$th of images that haven't been chosen before from K_{TRAIN} and put them into $K_{VALIDATION}$.
   - Put the rest into $K_{SUBTRAIN}$.
   - Train the learning algorithms on the $K_{SUBTRAIN}$ set.
   - Test and record on $K_{VALIDATION}$ every few iterations.
2. Determine the optimal value for T using average accuracies from cross-validation.
3. Train on K_{TRAIN} using T iterations
4. Test on the K_{TEST} set to obtain the accuracy for this experiment.

We compare accuracy achieved with different pairs of features and algorithms against each other. We also compare how algorithms described here fare against Support Vector Machines (SVM) (SVM$_{light}$ Implementation [14]).

The prediction accuracy of an experiment was computed by taking the prediction index ($\hat{q}$) for every image in K_{TEST} . The sum of correct predictions ($\hat{q} = \tilde{q}$) divided by the total number of testing images is the prediction accuracy (i.e. accuracy $= \frac{1}{m}\sum_{i=1}^{m} I_i$ where $I_i = 1$ if $\tilde{q} = \arg\max_{q \in Q} \mathbf{w} \cdot \mathbf{x}_i^q$ and 0 otherwise).

Because K_{TEST} contains images that are hard to orient, it is beneficial to abstain from orienting an image if determining its orientation is too difficult. Rejecting

an image for this reason can be done in a variety of ways. The simplest way is to reject an image if the dot product of the weight vector $\mathbf{w}$ and predicted orientation vector $\mathbf{x}_i^{\hat{q}}$ is less than some threshold. If an image is rejected it is not included in the prediction accuracy computation. We used this simple rejection scheme resulting in various proportions of rejected images. Plots of orientation accuracies with different levels of rejections are called rejection plots and they may give further insight into accuracies achieved with different learning algorithms. In addition to numerical summaries we provide rejection plots.

1.3 Dataset

The general case of natural image orientation is a very complex problem and we believe it cannot be done without object recognition. With 4 orientations and without object detection the best reported prediction accuracy for this problem is $\sim 75\%$ [19]. For image orientation to work images in K_{TRAIN} must have something in common with the images in K_{TEST} . For our dataset we chose 628 natural photographs that were all taken outdoors, during daytime with some background visible. For simplicity, we worked with square images at a resolution of 360×360 pixels. We used four orientations $(0°, 90°, 180°, 270°)$.

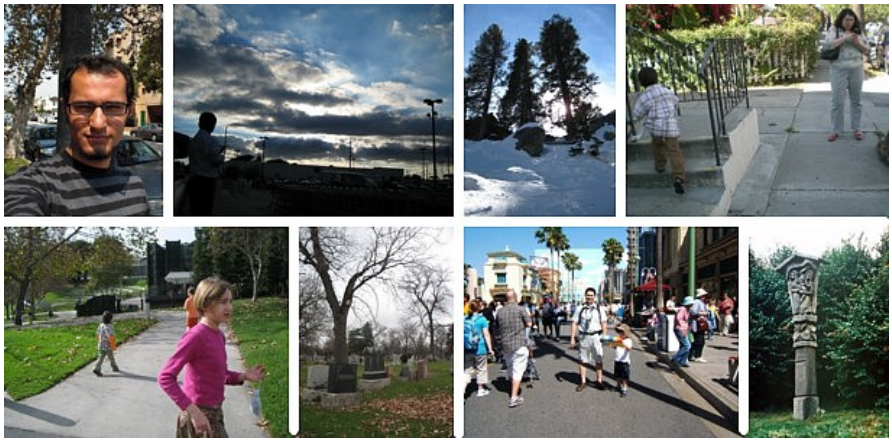

Figure 4: Some examples of the photographs we used for this thesis.

We only used images that had color in them (i.e. no grayscale or sepia). There had to be at least two distinct colors in the image so the colors are not all put into the same color bin. The square images produced for this thesis were made by re-sampling the pixels at 380×380 resolution using nearest-neighbor with 2x anti-aliasing. Blurring an image, computing edge magnitudes and passing the image through other filters produces noise. This is why we down-sized images with an extra 20 pixels of padding that were not used when computing features.

The dataset we used is good, but it is not perfect. Some of the images are very similar. They were taken in similar locations by the same photographer and include the same people over and over. Some of the images are definite outliers (e.g. photo in a pool with yellow wall above). Ideally we would have another data set from a new source to test on for more accurate reporting. Certain images would have to be filtered out, otherwise the expected orientation accuracy might be very low [2]. For this reason we take the best method and check its performance on another dataset in section 8.4.

2 Features

We use spatial features that is presence of features in certain parts of the image. All the features we use were picked for their simplicity and low calculation costs. We looked at Haar-Wavelet-Like (F_{HAAR}) features based on the ones described by Viola and Jones [28]. They are very fast to compute and they have been successfully used for facial detection. We looked at color presence (F_{COLOR}) features because there is a strong correlation between location of colors and orientation [29]. And we looked at orientation gradient (F_{EDGE}) features because they are widely used in computer vision for object detection [2, 6, 19, 25, 8, 32].

We computed the features in the context of the spatial pyramid. For each grid and for each region within a grid, we computed the features, and then appended them together into a feature vector. It is imperative to scale values in the regions by the number of pixels so that regions in higher levels of the spatial pyramid don't have more importance. Afterwards, features were converted into binary features via a threshold.

The feature vectors were calculated in the following way:

1. Pre-process the image
2. **For** $B = B_S \dots B_B$
   - Compute the features using a grid size of B
   - Adjust the features by multiplying by B^2
3. Convert the features vectors into decision stumps via a threshold.

2.1 F_{HAAR} Features

Haar-Wavelet-Like features or F_{HAAR} became accepted through the Viola and Jones' [28] paper where they successfully used them for face detection. F_{HAAR} are

features that resemble Haar-Wavelets. They are a collection of region-pixel-brightness comparisons. The features that we use are slightly different from those described in the original paper. For example in the original paper, a feature of [1 0 1] (a bright region followed by a dark region followed by a bright region) was computed by taking the sum of pixels in the white regions and subtracting double the sum of pixels in the middle region. The resulting feature value was a negation of the response to [0 1 0]. Our computation was different in the following way: first we ensured that for [1 0 1] the middle region was darker than the two others (otherwise the feature value is 0), and when computing the feature value, we subtracted the sum of pixels in the middle region from the darkest neighbor so that the resulting feature value was proportional to the average pixel-brightness difference between a light and the dark regions. In our implementation, the feature value for [1 0 1] is not the negation of [0 1 0], so both had to be included in the feature space.

We chose to try F_{HAAR} because in spatial pyramid regions are already defined and F_{HAAR} features can be extracted in a straight-forward manner by using every cell as a region. So in a level of the spatial pyramid of size 3 by 3, there will be three responses for [0 1 0].

Viola and Jones [28] describe how to highly optimize the computation of F_{HAAR} features by precomputing an integral image (see algorithm1) which can then be used for quick look-ups of the sum of the gray-scaled pixel values for any rectangular part of the image.

Algorithm 1 F_{HAAR} Computation

1. Inputs:

   $s \in \mathbb{Z}^+ =$ size of a side of the image (in pixels)
   $S = \{C_{RED}, C_{GREEN}, C_{BLUE}\}$
   $B_S, B_B \in \mathbb{Z}^+ : B_S < B_B =$ grid size interval

2. Initialize:

   $I = .3C_{RED} + .59C_{GREEN} + .11C_{BLUE}$
   $I = [I - I_{min}]\frac{256}{I_{max} - I_{min}}$
   For $x \in \{1 \dots s-1\}, y \in \{0 \dots s-1\}$
   $I_{x,y} = I_{x,y} + I_{x,y-1}$
   For $y \in \{1 \dots s-1\}, x \in \{0 \dots s-1\}$
   $I_{x,y} = I_{x,y} + I_{x-1,y}$
   $F = [\ ]$

3. Iterate:

   For $B = B_S \dots B_B$
   For $i = 1 \dots B, j = 1 \dots B$
   $\sigma = \frac{s}{B}$
   $Y_{i,j} = [I_{(i+1)\sigma,(j+1)\sigma} + I_{i\sigma,j\sigma} - I_{(i+1)\sigma,j\sigma} - I_{i\sigma,(j+1)\sigma}]B^2$
   Foreach $H \in \{[0\ 1], [0\ 1]^T, \dots\}$
   $F = F \cup f_c : f_c$ is the response to feature H
   E.g. $H = [0\ 1\ 0]$
   $f_c = [\ ]$
   For $i = 1 \dots B, j = 1 \dots B - 2$
   $f_c = f_c \cup \{\max(0, Y_{i+1,j} - \max(Y_{i,j}, Y_{i+2,j}))\}$
   $F = F > \frac{1}{2}F_{mean}? + 1 : -1$

4. Output:

   Vector F

In the beginning of each iteration, I contains the integral image computed with dynamic programming by first accumulating the sum over the rows and then columns. $Y_{i,j}$ is the sum of pixel values in region of row i column j scaled by B^2 so that every region in the spatial pyramid has the same scale and σ is the size of

a region in pixels.

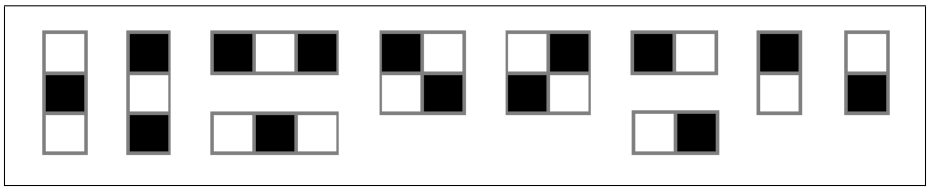

Figure 5: The HAAR features we used

The decision stumps computed using HAAR features can be interpreted in a straightforward way. If a decision stump is +1 for [1 0 1] it means that the bright regions are both brighter than the dark region. It further means that out of all features, the difference in brightness for this feature was larger than others.

2.2 F_{EDGE} Features

There has been reported success using directions and magnitude of Canny Edges in object detection and image orientation and classification [30, 27, 13, 24, 2]. The Canny Edge Detector takes two edge responses from a blurred image: V for vertical edges and H for horizontal edges. Then it computes the total magnitude of the edge by taking Euclidean Distance. There is a large choice of edge operators to compute H and V: Roberts, Prewitt, Sobel, Step and Double Step. We used Double Step as it worked best.

We use a variant of these features denoted as F_{EDGE} . The way we compute them is by first finding an approximation to Canny Edges (see section 5). Then, within the context of the spatial pyramid, for each cell we use directional bins that collect strength of edges in that direction. This can be thought of as first computing an edge histogram (a plot of edge intensity vs. edge direction angle) then dividing edge directions into equal intervals and computing the average value in the histogram for each interval. After adjusting for the size of the cell, appending

these bins together from the spatial pyramid produces a feature vector which can be turned into a binary feature vector through a separation by a threshold.

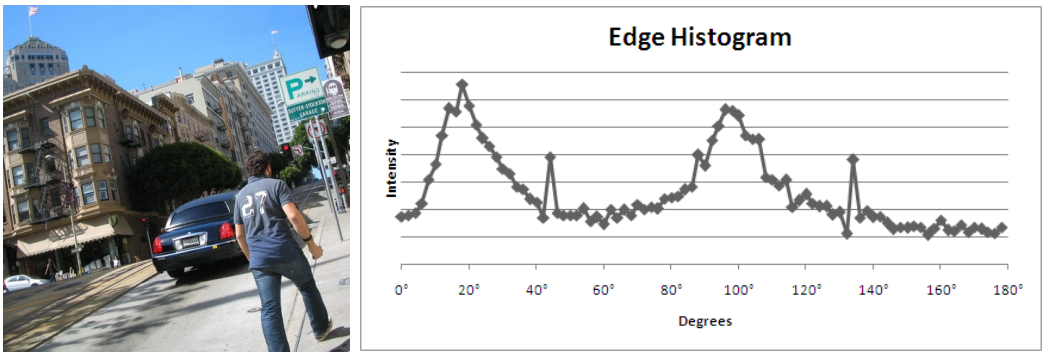

Figure 6: An example of an edge histogram. On the left is the source image and on the right is a plot of edge directions and their combined magnitude.

Matrix I is an $s \times s$ matrix containing the gray-scaled values of the pixels. It is normalized. The convolution of I with a horizontal double step filter produces a response matrix H which contains the strength of a horizontal edge at each pixel. Similarly, V contains the strength of a vertical edge at each pixel. For each angle we are trying to bin them into b orientation bins and matrix D contains an index of the angle.

Algorithm 2 F_{EDGE} Computation

1. Inputs:

   $s \in \mathbb{Z}^+ =$ size of a side of the image (in pixels)
   $S = \{C_{RED}, C_{GREEN}, C_{BLUE}\}$
   $B_S, B_B \in \mathbb{Z}^+ : B_S < B_B =$ grid size interval
   $b = \#$ of orientation bins

2. Initialize:

   $I = .3C_{RED} + .59C_{GREEN} + .11C_{BLUE}$
   $I = [I - I_{min}]\frac{256}{I_{max} - I_{min}}$
   $H = I * \begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ 1 & 0 & -1 \end{bmatrix}$
   $V = I * \begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 0 \\ -1 & 0 & -1 \end{bmatrix}$
   $M = |V| + |H|$
   $D = \lfloor \tan^{-1}\left(\frac{V}{H} + \frac{\pi}{2}\right)\frac{b-1}{\pi} \rfloor$
   $F = [\ ]$

3. Iterate:

   For $B = B_S \ldots B_B$
   For $i = 1 \ldots B, j = 1 \ldots B$
   $f_c = \mathbf{0}$
   $\sigma = \frac{s}{B}$
   For $x = i\sigma \ldots (i+1)\sigma, y = j\sigma \ldots (j+1)\sigma$
   $f_c(D_{x,y}) = f_c(D_{x,y}) + M_{x,y}$
   $F = F \cup (f_c B^2)$
   $F = F > \frac{1}{2}F_{mean}? + 1 : -1$

4. Output:

   Vector F

There is only one parameter that can be tuned: $b =$ number of degree bins $=$ how many discrete values we are taking from the edge histogram. This is inversely proportional to depth of the spatial pyramid.

2.3 F_{COLOR} Features

The last feature set that we used was color presence. Color contains an enormous amount of information in the image and has been successfully used as a feature for image orientation [30, 27, 13, 24, 2]. It makes sense to look for blue sky on top and green grass on the bottom of a correctly oriented image. There are many ways to make a computer look at color: HSV, LUV, CMYK, YCbCr, but we simply indexed the RGB (red, green and blue) values. Digital images naturally come in three channels, so it is efficient to work with RGB values. Related works tend to compute Color Moment (CM) features - mean and variance of color descriptors in some color-space. However because we chose to work with binary features, we have to index colors and it is easiest to work RGB values. This allows us to include features like absence of colors in certain regions of the features, which CM features do not capture.

Using all 24 bits of color information was too much for features, so we reduce the number of color values per channel from 256 to d.

Figure 7: An example of an image after color reduction. On the left is an image with 16 million colors (24 bits of color depth) and on the right it is after color reduction to 216 colors

Algorithm 3 F_{COLOR} Computation

1. Inputs:

   $s \in \mathbb{Z}^+ =$ size of a side of the image (in pixels)
   $S = \{C_{RED}, C_{GREEN}, C_{BLUE}\}$
   $B_S, B_B \in \mathbb{Z}^+ : B_S < B_B =$ grid size interval
   $b = \#$ of colors per channel

2. Initialize:

   $\forall\ C \in \{R, G, B\}\ C_C = C_C \frac{d}{256}$
   $I = C_{BLUE} + C_{GREEN} d + C_{RED} d^2$
   $F = [\ ]$

3. Iterate:

   For $B = B_S \ldots B_B$
   For $i = 1 \ldots B, j = 1 \ldots B$
   $f_c = \mathbf{0}$
   $\sigma = \frac{s}{B}$
   For $x = i\sigma \ldots (i+1)\sigma, y = j\sigma \ldots (j+1)\sigma$
   $f_c(I_{x,y}) = f_c(I_{x,y}) + 1$
   $F = F \cup (f_c B^2)$
   $F = F > \frac{1}{2} F_{mean} ? + 1 : -1$

4. Output:

   Vector F

This results in d^3 possible color indexes per cell. The only parameter to tune is d, how many values are there per channel.

3 Algorithms

The task for the machine learning algorithms described in this thesis is to learn to predict a correct image orientation. It can be accomplished by learning a linear weight vector $\mathbf{w}$, whose dot product with feature vectors of correctly oriented samples ($\mathbf{w} \cdot \mathbf{x}_i^q$) is high. This can be thought of as finding a separating hyperplane

in the feature space that separates $\mathbf{x}_i^1$ from $\mathbf{x}_i^q$ for $q > 1$. For orienting a single image i that means if $\tilde{q} = 1$ (the first sample for every image is upright) we want $\mathbf{w} \cdot \mathbf{x}_i^1 > \max(\mathbf{w} \cdot \mathbf{x}_i^2, \ldots, \mathbf{w} \cdot \mathbf{x}_i^Q)$.

In this thesis we describe 6 algorithms that have been used effectively for binary classification and ranking : Perceptron [9], AdaBoost [10], AdaBoost* [22], MadaBoost [7], SoftRankBoost [12] and Logistic Regression [3]. The linear weight vector $\mathbf{w}$ returned by the algorithms is perpendicular to the separating hyperplane pointing in the direction of positively-labeled samples. For binary classification tasks a threshold (hyperplane's distance from the origin) is necessary for the dot product $\mathbf{x}_j \cdot \mathbf{w}$ to classify into one of two classes. However thresholding is not necessary for image orientation because we choose the correct orientation based on the highest dot product.

3.1 Perceptron

The Perceptron Algorithm [9] works in the following way: it begins with some guess of $\mathbf{w}$ (usually $\mathbf{0}$), and then for every sample it updates $\mathbf{w}$ if it does not predict the sample label correctly. There are many versions of Perceptron. The simplest version, upon update adds the sample multiplied by the label (i.e. $\mathbf{w} = \mathbf{w} + y_j \mathbf{x}_j$). If the data is linearly separable (which is the case with our data), Perceptron will make a finite number of mistakes and will converge to a $\mathbf{w}$ that separates the training samples. There are many modifications to this algorithm that prove beneficial. Using a learning rate α that decreases over time so the update is $\mathbf{w} = \mathbf{w} + \alpha y_j \mathbf{x}_j$ makes Perceptron more stable. Training Perceptron on differences in samples (i.e. $\mathbf{x}_j = \mathbf{x}_l - \mathbf{x}_m$ s.t. $\forall l, m : \; y_l = +1, y_m = -1$) allows for a better use for ranking [4]. Averaging $\mathbf{w}$ makes the Perceptron generalize better. Freund and Schapire [9] described a Voted Perceptron which uses all of the weight vectors

produced after each example is processed and makes a prediction using a weighted combination of the different weight vectors' predictions. They proved that Voted Perceptron achieves a better margin than the original Perceptron algorithm.

We experimented with different versions of the Perceptron Algorithm and settled on the Averaged Perceptron with an update rule modified for our specific task. We did not use the Voted Perceptron because it does not return a single $\mathbf{w}$. In our experiments, Averaged Perceptron was just as good as Voted Perceptron. The Averaged Perceptron we used works in the following way: for each image i if $\mathbf{w}$ doesn't predict the correct orientation ($\hat{q} \neq 1$), it adds $\mathbf{x}_i^1$ and subtracts $\mathbf{x}_i^{\hat{q}}$ from $\mathbf{w}$. Adding $\mathbf{x}_i^1 - \mathbf{x}_i^{\hat{q}}$ to $\mathbf{w}$ changes it in such a way that new dot product with $\mathbf{x}_1^i$ will be higher and the dot product with $\mathbf{x}_{\hat{q}}^i$ will be lower. The algorithm continues to iterate for T iterations of the outermost loop (passes over the data, see algorithm 4) even if it is not making any more mistakes. Every iteration it keeps the average value of the weight vector $\mathbf{w}_{average}$ so far and returns that in the end.

When the Perceptron runs past separation ($t > t_s$ where $\mathbf{w}_{t_s}$ separates the training samples),$\mathbf{w}_{average}$ continues to change and in the limit converges to $\mathbf{w}_{t_s}$. Adjusting T helps generalize the $\mathbf{w}$ for future samples. Our data contains outliers and if T is small $\mathbf{w}_{average}$ will not be influenced by them much. On the other hand if $\mathbf{w}_{t_s}$ separates the data with a large margin and T is high enough, it will overshadow bad values of $\mathbf{w}$ during first iterations. It is beneficial to tune T with cross-validation.

Algorithm 4 Averaged Perceptron

1. Inputs:

   $Q \in \mathbb{Z}^+$: number of possible orientations.
   $m \in \mathbb{Z}^+$: number of images.
   $S = (\mathbf{x}_0^1, \ldots, \mathbf{x}_0^Q, \mathbf{x}_1^1, \ldots, \mathbf{x}_m^Q)$
   $\tilde{q} = 1$: correct orientation on all training samples
   $T \in \mathbb{Z}^+$: number of iterations.

2. Initialize:

   $\mathbf{w}_0 = \mathbf{0}$
   $\mathbf{w}_{average} = \mathbf{0}$

3. Iterate:

   For $t = 1 \ldots T$
   For $i = 0, \ldots, m-1$
   $\hat{q} = \arg\max_{q \in Q} \mathbf{w} \cdot \mathbf{z}_i^q$
   If $\tilde{q} \neq \hat{q}$ **then** $\mathbf{w}_t = \mathbf{w}_{t-1} + (\mathbf{x}_i^{\tilde{q}} - \mathbf{x}_i^{\hat{q}})$
   Else $\mathbf{w}_t = \mathbf{w}_{t-1}$
   $\mathbf{w}_{average} = \frac{t-1}{t}\mathbf{w}_{average} + \frac{1}{t}\mathbf{w}_t$

4. Output:

   Vector $\mathbf{w} = \mathbf{w}_{average}$

In our experiments, Averaged Perceptron found the optimal $\mathbf{w}$ in a very small number of passes over the data $(3 - 11)$ and was much faster than all the other algorithms we tried. For that reason we used Perceptron to tune parameters for feature extraction. This is not ideal, but it worked well in practice.

3.2 AdaBoost

Boosting algorithms combine predictions of weak classifiers into a stronger classifier. The decision stumps that make up the feature vectors in our problem can be thought of as classifiers (weak learners). The linear weights that boosting al-

gorithms can assign to them make up the linear weight vector $\mathbf{w}$. AdaBoost [10] is an adaptive boosting algorithm that chooses weak learners in a greedy fashion and assigns weights to them. AdaBoost finds a hyperplane in the feature space that separates the training samples with a positive margin.

The original AdaBoost [10] works with hypotheses (label predictions of a classifier). With binary features, the hypotheses are the decision stumps (see figure 8) $\mathbf{h}_k$ (where k is the number of the feature). AdaBoost maintains a distribution (weight vector where weights are positive and sum up to 1) $\mathbf{D}$ for the samples. Every iteration $t \in T$ when $\mathbf{h}_{k_t}$ is selected a weight α_t is added to the corresponding $\mathbf{w}_{k_t}$.

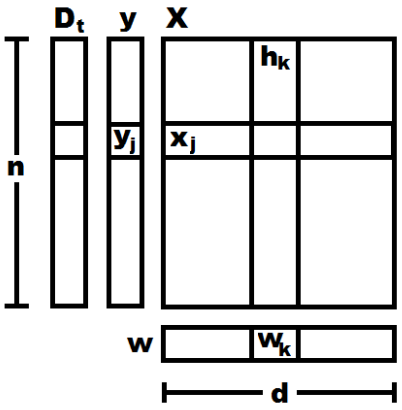

Figure 8: A graphical representation of the data used for AdaBoost, AdaBoost* and MadaBoost

In the original formulation of AdaBoost, the prediction of the final classifier for $\mathbf{x}_j$ is the sign$(\sum_t \alpha_t \mathbf{h}_t(\mathbf{x}_j))$ resulting in a binary classification algorithm. For our problem there are 4 orientations and we pick the output with the highest $\mathbf{w} \cdot \mathbf{x}_j$. The separating hyperplane has a margin for sample j of $y_j \mathbf{w} \cdot \mathbf{x}_j$ (assuming $||\mathbf{w}||_1 = 1$). This measure of the margin is the Manhattan Distance for sample j to the separating hyperplane. The dual problem to maximizing the minimal margin

is minimizing the maximal edge γ. Edge γ_k for hypothesis k is a measure of how much better than random is hypothesis $\mathbf{h}_k$. More formally, $\gamma_k = 1 - 2$ error so if $\gamma_k = 1, h_k$ predicts perfectly and if $\gamma_k = 0, h_k$ is as good as a random guesser). The definition of γ_k with respect to distribution $\mathbf{D}$ is $\gamma_k = \sum_j^n \mathbf{D}(j) y_j h_k(j)$. Every iteration $t \in T$ of AdaBoost, the next hypothesis $\mathbf{h}_t$ is the one that maximizes the edge with respect to the current distribution $\mathbf{D}_t$.

The range for γ_k is [-1, +1]. The set of all hypotheses H is complement closed, so when computing $\mathbf{h}_t$ for every k both $\mathbf{h}_k$ and $-\mathbf{h}_k$ are considered. Because of this γ_t is never negative (for every $\mathbf{h}_k$ for which $\gamma_k < 0$ there's its negation for which $\gamma > 0$). A simple way of making sure the set of hypotheses is closed under negation is to append the negatives of the decision stumps to the feature vectors. Working with decision stumps for AdaBoost has been looked into by Rennie [23]. The details of the way we implemented it are covered in section 5. AdaBoost converges in the limit, however it only achieves half the optimal margin (if the maximal margin is ρ^*, the resulting margin from AdaBoost is $\frac{1}{2}\rho^*$) [22].

Once the $\mathbf{h}_t$ is selected, AdaBoost computes a weight for it (α) and updates $\mathbf{D}$ accordingly. The update is $\mathbf{D}_t(j) = D_{t-1}(j) \exp(-\alpha_t y_t h_t(j))$ which adjusts the weights in an exponential manner. In the limit $\mathbf{D}$ has all the weight put on outliers. In most cases, AdaBoost does not over-fit [10]. In our experiments, over-fitting occurs. Therefore early stopping is a reasonable thing to do and T is a parameter that needs to be tuned.

Because the distribution $\mathbf{D}$ is adjusted in an exponential manner, it is possible that a change from one iteration to the other overshoots the optimal point. To prevent this from happening in an ad-hoc way, we cap α_t so that the update is not as drastic. This makes the algorithm learn slower but does not change the final outcome. However, it allows us to fine-tune T. We use this AdaBoost with capped α.

Algorithm 5 AdaBoost with capped α

1. Inputs:

   $n \in \mathbb{Z}^+ =$ number of samples (mQ).
   $d \in \mathbb{Z}^+ =$ size of the feature vectors $\mathbf{x}_j$.
   $T \in \mathbb{Z}^+ =$ maximum number of iterations.
   $S = \{\mathbf{y}, \mathbf{h}_1, \mathbf{h}_2, \ldots, \mathbf{h}_d\}$
   $\alpha^C \in \mathbb{R}^+ =$ the maximum value for α

2. Initialize:

   $\mathbf{w} = \mathbf{0}$
   $\mathbf{D}_0 = \frac{\mathbf{1}}{\mathbf{n}}$, initialize sample weights to $\frac{1}{n}$

3. Iterate:

   For $t = 1 \ldots T$
   $k_t = \arg\max_k \gamma_k$ where $\gamma_k = \sum_j^n \mathbf{D}_{t-1}(j) y_j h_k(j)$
   $h_t = h_{k_t}$
   $\gamma_t = \gamma_{k_t}$
   $\alpha_t = \frac{1}{2} \log\left(\frac{1+\gamma_t}{1-\gamma_t}\right)$
   $\alpha_t = \min(\alpha_t, \alpha^C)$
   Update the distribution
   $\mathbf{D}_t(j) = D_{t-1}(j) \exp(-\alpha_t y_j h_t(j))$
   $\mathbf{D}_t = \frac{\mathbf{D}_t}{||\mathbf{D}_t||}$
   $w_{k_t} = w_{k_t} + \alpha_t$

4. Output:

   Vector $\mathbf{w}$

As with the Perceptron Algorithm, we can benefit from early stopping via setting a limit on T. This limit can be tuned with cross-validation.

3.3 AdaBoost*

A limitation of AdaBoost is that it is only stable if a separation exists (the optimal margin $\rho^* > 0$). If it does, in the limit AdaBoost achieves half the optimal margin.

AdaBoost* [22] is a variation of AdaBoost that achieves the optimal margin even if the margin is negative (training data is inseparable). AdaBoost* has a guarantee that after $T = \frac{2 \log n}{\nu^2}$ iterations it is guaranteed to be within ν from the optimal margin (i.e. the achieved margin is at least $\rho^* - \nu$). In fact, AdaBoost* explicitly maximizes the minimal margin ($\max_{\mathbf{w}} \min_j y_j(\mathbf{w} \cdot \mathbf{x}_j)$ s.t. $||w||_1 = 1$) to a precision of ν. AdaBoost* approaches the optimal margin by using the best estimate of what it is at every iteration $t \in T$.

The modification to the algorithm is very simple: α_t is updated using the smallest edge (γ_t) seen so far by subtracting $\frac{1}{2} \log \left(\frac{1-\gamma_t^{min}+\nu}{1+\gamma_t^{min}-\nu} \right)$ from it.

Algorithm 6 AdaBoost*

1. Inputs:

   $n \in \mathbb{Z}^+$ = number of samples (mQ).
   $d \in \mathbb{Z}^+$ = size of the feature vectors $\mathbf{x}_j$.
   $T \in \mathbb{Z}^+$ = maximum number of iterations.
   $S = \{\mathbf{y}, \mathbf{h}_1, \mathbf{h}_2, \ldots, \mathbf{h}_d\}$
   $\nu \in \mathbb{R}^+$ = desired accuracy parameter.

2. Initialize:

   $\mathbf{w} = \mathbf{0}$
   $\mathbf{D}_0 = \frac{\mathbf{1}}{\mathbf{n}}$, initialize sample weights to $\frac{1}{n}$

3. Iterate:

   For $t = 1 \ldots T$
   $k_t = \arg\max_k \gamma_k$ where $\gamma_k = \sum_j^n \mathbf{D}_{t-1}(j) y_j h_k(j)$
   $h_t = h_{k_t}$
   $\gamma_t = \gamma_{k_t}$
   $\gamma_t^{min} = \min(\gamma_t, \gamma_{t-1}^{min})$
   $\rho_t = \gamma_t^{min} - \nu$
   $\alpha^t = \frac{1}{2}\log\left(\frac{1+\gamma_t}{1-\gamma_t}\right) - \frac{1}{2}\log\left(\frac{1-\rho_t}{1+\rho_t}\right)$
   Update the distribution
   $\mathbf{D}_t(j) = D_{t-1}(j)\exp(-\alpha_t y_j h_t(j))$
   $\mathbf{D}_t = \frac{\mathbf{D}_t}{||\mathbf{D}_t||}$
   $w_{k_t} = w_{k_t} + \alpha_t$

4. Output:

   Vector $\mathbf{w}$

AdaBoost* has two parameters that can be tuned: ν - the precision parameter to how close to the optimal margin we want to get and T - the number of iterations the algorithm is allowed to take. Even though we have a bound on the number of iterations given ν, our data is not i.i.d. so in practice the algorithm converges much faster. Our samples also contain outliers, so early stopping (T) is a parameter that needs to be tuned.

3.4 MadaBoost

AdaBoost and AdaBoost* compute a hyperplane that maximizes the minimal hard margin. That means that every sample has the same importance. This is a problem when the data is noisy (contains outliers). To address this, there are versions of AdaBoost that achieve a so-called soft-margin that is noise resistant. The simplest of these algorithms is MadaBoost [7]. The way MadaBoost achieves soft margin is by limiting weights in the sample weight distribution.

The modification from AdaBoost is very simple. The data layout is the same as in AdaBoost (see figure 8) except the sample weight distribution used to compute γ_{k_t} is different. When picking a hypothesis $\mathbf{h}_t$ a distribution $\mathbf{D}'_t$ is used. $\mathbf{D}'_t$ is the same as $\mathbf{D}_t$ except the weights are capped to their original values. $\mathbf{D}'_t$ is computed by capping weights at $\frac{1}{n}$ and then dividing by the 1-norm to make it a proper distribution. Everything else is exactly the same.

Because the weights are capped, α is prevented from getting large and as a result MadaBoost is slower than AdaBoost theoretically. However, in our experiments, MadaBoost was as fast as AdaBoost (see section 7).

Algorithm 7 MadaBoost

1. Inputs:

   $n \in \mathbb{Z}^+$ = number of samples (mQ).
   $d \in \mathbb{Z}^+$ = size of the feature vectors $\mathbf{x}_j$.
   $T \in \mathbb{Z}^+$ = maximum number of iterations.
   $S = \{\mathbf{y}, \mathbf{h}_1, \mathbf{h}_2, \ldots, \mathbf{h}_d\}$

2. Initialize:

   $\mathbf{w} = \mathbf{0}$
   $\mathbf{D}_0 = \frac{\mathbf{1}}{\mathbf{n}}$, initialize sample weights to $\frac{1}{n}$

3. Iterate:

   For $t = 1 \ldots T$
   $\mathbf{D}'_t = \min(\frac{1}{n}, \mathbf{D}_{t-1})$
   $\mathbf{D}'_t = \frac{\mathbf{D}'_t}{||\mathbf{D}'_t||}$
   $k_t = \arg\max_k \gamma_k$ where $\gamma_k = \sum_j^n \mathbf{D}'_t(j) y_j h_k(j)$
   $h_t = h_{k_t}$
   $\gamma_t = \gamma_{k_t}$
   If $\gamma_t \leq 0$ **then break**
   $\alpha_t = \frac{1}{2} \log\left(\frac{1+\gamma_t}{1-\gamma_t}\right)$
   Update the distribution
   $\mathbf{D}_t(j) = D_{t-1}(j) \exp(-\alpha_t y_j h_t(j))$
   $\mathbf{D}_t = \frac{\mathbf{D}_t}{||\mathbf{D}_t||}$
   $w_{k_t} = w_{k_t} + \alpha_t$

4. Output:

   Vector $\mathbf{w}$

MadaBoost converged just as fast as AdaBoost or AdaBoost*. We found that despite being a soft-margin classifier, MadaBoost suffered from over-fitting and benefited from early-stopping so T is a parameter that needs to be tuned.

3.5 SoftRankBoost

RankBoost is an algorithm designed to maximize the area under ROC (Receiver Operating Characteristic) Curves (plots of false positive vs true positive rates) [12]. ROC Curves are computed by varying the classification threshold, which is also the way we reject images. It is possible that maximizing area under the ROC Curve for images in $K_{SUBTRAIN}$ leads to a large area under a rejection curve for images in K_{TEST} .

RankBoost is a bipartite ranking algorithm the goal of which is to output a scoring function that will score any positive sample higher than any negative sample. For decision stumps (binary feature vectors) the scoring function is the dot product with a linear weights vector $\mathbf{w}$.

SoftRankBoost is a modification of RankBoost that achieves an approximate optimal soft margin [12]. It is similar to MadaBoost in that sample weights are capped. SoftRankBoost takes in similar parameters to other boosting algorithms described in this thesis with one core difference: the data is split up into two sets containing positive and negative samples (X^+ and X^- respectively).

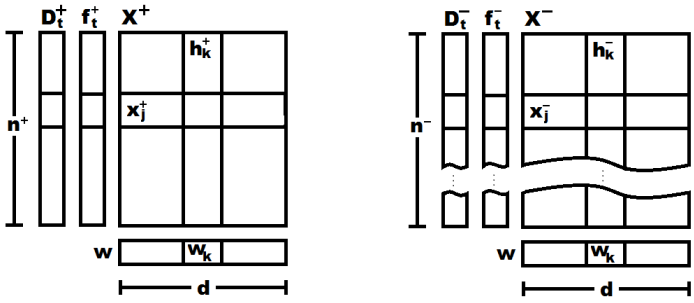

Figure 9: A graphical representation of the data used in SoftRankBoost. It is very similar to the layout in figure 8 except the data here is separated into two matrices, a matrix with positive samples X^+ and a matrix with negative samples X^-.

There is one parameter to the algorithm: $\delta \in (0, 1)$. It determines how close to the optimal soft margin the algorithm will get. SoftRankBoost provably achieves

a margin objective of $(1-\delta)\gamma^*$. Note that this implies that γ_t must be positive and the set of hypotheses must be closed under negations (which it is). The number of iterations in which the algorithm is guaranteed to achieve a certain margin is inversely proportional to δ^2. The original SoftRankBoost paper [12] mentions a stopping criteria based on ν (measure of how unlikely is that the achieved margin is under $(1-\delta)\gamma^*$). We did not use that stopping criteria, because we found that in our experiments SoftRankBoost over-fits and can benefit from stopping earlier.

Algorithm 8 SoftRankBoost

1. Inputs:

   $S^+ = \{\mathbf{h}_1^+, \mathbf{h}_2^+, \ldots, \mathbf{h}_d^+\}$
   $S^- = \{\mathbf{h}_1^-, \mathbf{h}_2^-, \ldots, \mathbf{h}_d^-\}$
   $\delta \in (0, 1)$ = desired precision parameter.

2. Initialize:

   $\mathbf{w} = \mathbf{0}$, $\mathbf{D}_0^+ = \frac{\mathbf{1}}{\mathbf{n}^+}$, $\mathbf{D}_0^- = \frac{\mathbf{1}}{\mathbf{n}^-}$, $\alpha_0^s = 0$, $\mathbf{f}_0^+ = \mathbf{0}$, $\mathbf{f}_0^- = \mathbf{0}$, $\gamma^{min} = +\inf$

3. Iterate:

   For $t = 1 \ldots T$
   $k_t = \arg\max_k \gamma_k$
   where $\gamma_k = \frac{1}{2}(\mathbf{h}_k^+ \cdot \mathbf{D}_{t-1}^+ - \mathbf{h}_k^- \cdot \mathbf{D}_{t-1}^-)$
   $\gamma_t = \gamma_{k_t}$
   $\mathbf{h}_t^+ = \mathbf{h}_{k_t}^+$
   $\mathbf{h}_t^- = \mathbf{h}_{k_t}^-$
   $\gamma_t^{min} = \min(\gamma_t, \gamma_{t-1}^{min})$
   $\hat{\gamma}_t = (1 - \delta)\gamma_t^{min}$
   $\alpha_t = \dfrac{\gamma_t - \hat{\gamma}_t}{2(1 + \hat{\gamma}_t{}^2)}$
   $w_{k_t} = w_{k_t} + \alpha_t$
   $b_t = -\frac{1}{2}(\mathbf{h}_k^+ \cdot \mathbf{D}_{t-1}^+ + \mathbf{h}_k^- \cdot \mathbf{D}_{t-1}^-)$
   $\mathbf{f}_t^+ = \mathbf{f}_{t-1}^+ + .5\alpha_t(\mathbf{h}_t^+ + b_t)$
   $\mathbf{f}_t^- = \mathbf{f}_{t-1}^- + .5\alpha_t(\mathbf{h}_t^- + b_t)$
   $\alpha_t^s = \alpha_{t-1}^s + \alpha_t$
   Update the sample weight distributions
   $\mathbf{D}_t^+(j) = \min(1, \exp(-\mathbf{f}_t^+(j) + \hat{\gamma}_t\alpha_t^s))$
   $\mathbf{D}_t^-(j) = \min(1, \exp(\mathbf{f}_t^-(j) + \hat{\gamma}_t\alpha_t^s))$
   $\mathbf{D}_t^+ = \frac{\mathbf{D}_t^+}{||\mathbf{D}_t^+||}$
   $\mathbf{D}_t^- = \frac{\mathbf{D}_t^-}{||\mathbf{D}_t^-||}$

4. Output:

   Vector $\mathbf{w}$

3.6 Logistic Regression

The logistic growth function is a function that maps real valued input to probabilities (values $\in [0, 1]$). It looks like a stretched-out letter 'S' and is sometimes referred to as the Sigmoid function. This function is defined as $f(z) = \frac{1}{1+e^{-z}}$ and can be used as a model for binary classification. If we have a feature vector $\mathbf{x}_j$ and a weight vector $\mathbf{w}$, then we can predict label as $\hat{y}_j = f(\mathbf{x}_j \cdot \mathbf{w})$. If we need a binary predictor we can separate the predictions into classes by a threshold $\theta \in [0, 1]$ so if $\hat{y}_j > \theta$ then the prediction is 1 and 0 otherwise.

In a binary classification problem what we want is to get perfect predictions. The batch loss for logistic regression is $\sum_j^n \log f(\mathbf{x}_j) - y_j(\mathbf{x}_j \cdot \mathbf{w})$. Finding a $\mathbf{w}$ that minimizes the loss is the goal of logistic regression. The problem is that minimizing this produces a $\mathbf{w}$ that over-fits the training data. A way to regularize it is to ensure that no w_k approaches infinity. This can be done by regularization with $\lambda||\mathbf{w}||_2^2$ where λ is some regularization parameter that needs to be tuned. The way to formulate this problem is $\mathbf{w} = \arg\min_{\mathbf{w}} \sum_j^n \ln(1 + e^{\mathbf{w}\cdot\mathbf{x}_j}) - y_j(\mathbf{x}_j \cdot \mathbf{w}) + \lambda||\mathbf{w}||_2^2$.

A way to approximately solve this equation (find an optimal value) is by Gradient Descent. Gradient Descent is an optimization algorithm that takes steps against the gradient. The gradient for logistic regression with regularization is $\nabla = \sum_j^n (\hat{\mathbf{y}} - \mathbf{y}) \cdot \mathbf{x}_j + \lambda\mathbf{w}$. Subtracting a scaled gradient from $\mathbf{w}$ every iteration $t \in T$ eventually makes $\mathbf{w}$ converge at the optimal point. The update is $\mathbf{w}_t = \mathbf{w}_{t-1} - \eta\nabla$ where η is step size.

The parameter η doesn't affect the outcome, but if it is very small, the algorithm will require a large number of iterations to converge, and if it is too large, then updates for $\mathbf{w}$ might take very large steps and miss the optimal point. For this reason we dynamically change η. If the size of the gradient (g_t) is not decreasing (η is too large), we decrease η. If the size of the gradient is falling very

slowly, we increase η.

To make it accept binary data $\in \{+1, -1\}$ (as opposed to $\{0, 1\}$). We defined $f(z) = \frac{1-\exp(-z)}{1+\exp(-z)}$ so that $f(z)$ returns values $\in (-1, +1)$.

Lastly, early stopping in our experiments proved to be beneficial, so the algorithm is allowed to execute up to T iterations of the outer-most loop (make T steps, see algorithm 9). Optimal value of T can be tuned using cross-validation. Through experimentation (section 7) we found a good value of λ and held it constant.

Algorithm 9 Batch Logistic Regression with Gradient descent

1. Inputs:

   $n \in \mathbb{Z}^+ = \#$ of samples (mQ).
   $d \in \mathbb{Z}^+ =$ size of the feature vectors $\mathbf{x}_j$.
   $S = \{\mathbf{y}, \mathbf{x}_1, \ldots, \mathbf{x}_n\}$
   $\lambda \in \mathbb{R}^+ =$ regularization factor.
   $T \in \mathbb{Z}^+ =$ maximum number of iterations.

2. Initialize:

   $\mathbf{w} = \mathbf{0}$, initialize the feature weights to 0
   $g_0 = +\inf$, initially size of gradient is unknown
   $\eta_0 = \frac{2}{d}$, a good starting value for η

3. Iterate:

   For $t = 1 \ldots T$
   $\forall\ j \in \{0 \ldots n-1\}\ \hat{y}_j = \frac{1-\exp(-\mathbf{x}_j\mathbf{w})}{1+\exp(-\mathbf{x}_j\mathbf{w})}$
   $\forall\ j \in \{0 \ldots n-1\}\ \nabla_j = \mathbf{x}_j \cdot (\hat{\mathbf{y}} - \mathbf{y})$
   $\nabla = \nabla + \lambda\mathbf{w}$
   $g_t = ||\nabla||_1$
   If $g_t - g_{t-1} < 0$ **then** $\eta_t = \frac{1}{2}\eta_{t-1}$
   Else If $\frac{g_{t-1}}{g_t} > \frac{19}{20}$ **then** $\eta_t = \frac{3}{2}\eta_{t-1}$
   $\mathbf{w} = \mathbf{w} - \eta_t\nabla$

4. Output:

   Vector $\mathbf{w}$

Note that we should rank orientations by $\hat{y}_j = f(\mathbf{w} \cdot \mathbf{x}_j)$, but we rank candidates by highest $\mathbf{w} \cdot \mathbf{x}_j$. Since the sigmoid function is monotonously increasing, ranking them by $f(\mathbf{w} \cdot \mathbf{x}_j)$ would produce the same $\hat{q}$.

4 Related Work

In recent years there have been numerous publications about image orientation. There is work that uses low-level cues like colors and edges to determine image orientation and there is work that uses high-level semantic cues like faces, sky etc. This section is structured in the following way: first we define common terms, then discuss previous work and compare with work done in this thesis.

4.1 Common Terms

- Accuracy - Unless stated otherwise this is the portion of the images in the test set that were oriented correctly and were not rejected.
- LUV Color Space - CIE L*U*V* color space that defines every color in terms of human-perceived lightness (L*) and the color coordinate (U* and V*).
- Color Moments (CM) - The first and second moments of a color descriptor, the mean and variance of the color in a cell within a grid. The color is described via three axis in the LUV color space unless stated otherwise.
- Spatial Pyramid - The combination of many grids such as 1×1, 2×2 and so on.
- Support Vector Machines (SVM) - Machine learning algorithm that finds a hyperplane in a multidimensional space that separates data into two classes while maximizing the minimum margin (Euclidean distance to the hyperplane) of all samples.
- Learning Vector Quantization (LVQ) - A type of a neural network classifier with three neural layers: input layer, classification layer and an output layer where a winner-takes-all approach is used.

- Edge Direction Histogram (EDH) - Features that have binned edge strength separated by the direction of the edges. The same thing as F_{EDGE} features used in this thesis.
- Rejection Scheme - Unless stated otherwise, assume this means not orienting an image where a confidence level or posterior probability is below a certain threshold.
- K Nearest Neighbors (KNN) - Machine learning algorithm that classifies testing samples based on the class of the K neighbors closest to it.
- Linear Discriminant Analysis (LDA) - Method of finding linear combinations of features for classification. It is based on the assumption that conditional probability densities are normally distributed. For the papers mentioned here it is only used for feature selection.
- Principal Component Analysis (PCA) - Method of reducing the dimensionality of a feature space by looking into principal components, features that are most descriptive (i.e. have the highest variance).

4.2 Image Orientation

The earliest paper we could find on automatic image orientation was a paper by Thomas Kersten and Silvio Haering from 1997 [15]. They had aerial photographs that were scanned with different scanners at different resolutions. All the images had fiducial marks (recognizable objects) from which the orientation could be determined. Kersten and Haering used the Hough Transform of the image and oriented it. They used a spatial pyramid and looked for these marks within it. They compared them to a correctly-oriented template via Least Squares Matching (LSM) and chose the orientation that matched it the best. With the technology available back then, it took anywhere between 3 and 30 seconds to orient an image automatically.

The first piece of work that tackled the problem of automatically orienting general images was done by Y. Wang and HJ. Zhang in 2001 [30]. They took photographs

from the COREL photo gallery and oriented them using a technique similar to the one described in this thesis. They broke the image into an 8×8 grid and computed CM and EDH features on the peripheral cells. They had 288 features from CM and 925 from EDH. When training only on CM features, they compared LVQ and SVM and got accuracy of 69.2% and 73.7% respectively, concluding that SVMs were better than LVQ for orientation detection. Because the COREL database is so diverse and includes photos that do not have a distinct orientation (like a photo from the top of a bowl on a table) a rejection scheme had to be used. With 50% rejection and both CM and EDH features they were able to achieve 96.5% accuracy using SVMs. They also tried a two-layer SVM framework. They had 8 SVMs learn in parallel and then another SVM as a layer on top that outputted a master prediction. Using both features, two-layer-SVM and a rejection of 50% they were able to achieve 97.5% accuracy.

A year later, Vailaya, Y. Wang and HJ.Zhang published a paper [27] that compared K-NN, SVM, LVQ, Mixture of Gaussians and Hierarchical Discriminating Trees (HDR) as machine learning algorithms for image orientation. They used CM features on a 10×10 grid and a subset of photos from the COREL gallery to as data. After computing large feature vectors and they reduced them with LDA and PCA. They claim that LDA is much better than PCA so they only used LDA when reporting results. In their results, LVQ achieved $\sim$1% better accuracy than SVM and using LDA generally resulted in better performance. They claim that in additional to speeding up learning, using only features selected by LDA results in more general classifiers. The best accuracy of 96.5% was achieved using a Mixture of Gaussians on features selected by LDA.

L. Zhang, MJ. Li and H.J. Zhang then wrote a paper [32] on image orienta-

tion where they compared results from their previous work [30] against using AdaBoost. Their paper claims that SVM perform better than LVQ. They used CM and EDH features calculated from a 5×5 grid with 325 and 475 feature vector sizes respectively. When they tried AdaBoost on the same subset of photos from the COREL gallery they got 76.5% accuracy and 95.4% with 50% rejection. They tried improving the accuracy for AdaBoost by combining features via subtraction (subtracting some features from others to make new features). Then they used a run of AdaBoost to select 1400 features, and then ran AdaBoost again to train. With that they were able to get 96.5% accuracy, which was better but still inferior to the 97.5% accuracy they achieved with two-layer SVM. They also tried improving the results by modifying the rejection scheme to reject more indoor images. On a different dataset, if they first classifying images as being indoor/outdoor and then orient them with their new rejection scheme. Using that rejection scheme increased the accuracy from 72% to 82.1%. They explain the change stating that indoor images are much harder to orient than outdoor.

L. Wang at al. [29] did work where they wanted to see how well they can orient images based on high level semantics. They used a Bayesian Framework to orient 1200 images based on positions of faces, sky, light, texture and symmetry identified by humans. They had compared orienting within 4 orientations and 120 (360 degrees within a 3 degree margin of error). With all high-level cues integrated by Bayesian Integration they were able to achieve 92.6% and 94.1% accuracy with 120 and 4 orientations respectively.

Later J. Luo and M. Boutell did some work on integrating low-level cues and high-level-semantic cues for image orientation [13]. They mention that humans can only achieve 84% accuracy when orienting low-resolution thumbnails and try

to have their system match up. They compute CM features on a 7×7 grid resulting in 294 features and EDH features on a 5×5 grid resulting in 425 features. With a subset of the COREL gallery they achieve 68.8% accuracy with CM features, 54.7% with EDH features and 70.4% with both features combined. When they add semantic cues such as face, sky, ceiling, cloudy sky and grass they get accuracy of 82.7%. With 12% rejection based on confidence levels of their Bayesian Framework, they get 88.3% accuracy. They note that if they also take into account prior orientation probabilities computed from the training set (where 72% of all images are already correctly oriented), they achieve 91.3% accuracy with 4% rejection.

In 2004 Z. Su did work on image orientation where they used Neural Networks (NN) [24]. They used CM features from the HSV space and EDH computed on a 4×4 grid. Then they used PCA to reduce the dimensionality of the features to 228. They tried using all 16 cells in the grid and just the 12 peripheral ones and found that EDH features work best with all 16 and CM work better with just the peripheral ones because the inner-most cells of the grid contain rotation-invariant information that is mostly noise. For the machine learning they use Neural Networks with 16 hidden layers and they compare linear, logistic and softmax activation functions. With the best choice of the activation function, a two-layer Neural Network framework and 50% rejection they achieve 99.4% accuracy on a subset of the COREL gallery.

A little later S. Baluja and H. Rowley did work comparing very low-level cues as features for orientation [2]. They took images from the COREL gallery and from World Wide Web and used SVM to orient them using the following features: R, G, B, Y, I, Q, grayscale, horizontal edges and vertical edges computed via a

convolution with a single-step filter. They also tried normalized versions of image. They computed the features within a spatial pyramid up to level 5 (the largest grid was 6×6). Their feature vector contained mean values and variances for each cell resulting in a total size of $\sim$4000. With 4 orientations they were able to achieve an accuracy of 55.4% and brought it up to 65.2% with 40% rejection. They also looked into what sort of images can get oriented well and which cannot. They found that images taken outside (castles, buildings, valleys etc.) could be oriented with 94% accuracy. Other outdoor images (railways, some animals, African landscape etc.) could only be oriented with 81% accuracy. Images of wild life and people could be only oriented with 59% accuracy. Hard to orient images included doors, caverns, reptiles in grass etc. and could only be oriented with 40% accuracy. Lastly, certain images could not be oriented (had orientation accuracy of 25%, the same random guessing) included color backgrounds and textures top-views on flowers, caves with crystals, aerial photographs etc. Their results seem to imply that normalizing single channels hurts accuracy (removing normalized channel features results in better performance) and variance caries some information (they report a 1% drop in accuracy without variance features).

Around the same time M. Datar and XJ. Qi tried orienting images using Self Organizing Maps (SOM) [6]. They computed CM features on a 4×4 grid and only used peripheral cells. They used both first and second moments in the HSV color space as features resulting in feature vectors of size 72. Then trained a SOM classifier with 16 hidden layers and achieved 75% accuracy on a private dataset.

Later L. Lumini and L. Nani oriented images using a variety of features and a variety of algorithms and then tried a variety of algorithms on top of that to boost the performance [19]. They used their own dataset. They tried the following fea-

tures: CM, EDH, Harris Corner Histogram (HCH) and Phase Symmetric (PHS), and the following algorithms: Linear Discriminant Classifier (LDC), Quadratic Discriminant Classifier (QDC), Parzen Window Classifier (PWC) and Radial and Polynomial SVM (RSVM and PSVM). For CM and all-combined features QDC was the best and for all the other stand-alone features RSVM showed best accuracy. They used all classifiers with all the features, and tried combining the predictions via Min Rule, Max Rule, Vote Rule, Mean Rule, Dynamic Classifier Selection (DCS) and BORDA count. BORDA count yielded the best accuracy of 62%. They modified the algorithm a little and achieved a 73.5% accuracy with it and further improved it to 88% with 50% rejection. For comparison, they tried RSVM on CM+EDH features with the same data and with 50% rejection achieved only 74% accuracy.

Around the same time E. Tolstaya did work on image orientation using AdaBoost for classification [25]. They used a private dataset of outdoor images with 800 samples. They compute CM features and color-angle features in an $N \times N$ grid. They only classify within 3 orientations, but as a sequence of two binary classification problems. First they classify as upright or not, and if not classify which way it should be rotated (90 degrees left or right). With .4% rejection they achieve accuracy of 87% and with 40% rejection they achieve 90%.

In 2009, there was a patent application submitted by D. Wang et al. that described an image orientation system [8]. They used CM features computed on a 7×7 grid and EDH features computed on a 5×5 grid resulting in 294 and 425 features respectively. They used the features to classify photos into one of the 17 classes from birds, plants, animals, flowers, butterflies etc. After classification, they used Approximate Nearest Neighbour (ANN) classifier to orient the images.

They achieved 97% - 100% accuracy on their private dataset that only included images from the 17 classes.

4.3 Discussion

The majority of the work done in the image orientation field has been somewhat similar. Color Moments (CM) and Edge Direction Histograms (EDH) are the two ways to compute features used in almost all attempts to orient images. Most papers use features based on low-level cues. This thesis is no different. Humans use primarily low-level cues to know which way is up [29] so it makes sense to let machines use them to orient images.

SVM seems to be the machine learning algorithm of choice for learning orientations. None of the papers address ease of implementation as a criterion for choosing a machine learning algorithm. Only work by Lumini and Nani [19] compares various machine learning algorithms against each other. One of the main criteria for choosing algorithms tried in this thesis was ease of implementation. A number of algorithms was implemented and compared to the SVM standard (SVM_{light} implementation).

The work for this thesis was done years after previous work in image orientation, and with faster computing power available we were able to consider data with a much larger number of features. Previous work in image orientation used feature vectors of size ~ 1000 with all features combined. We used ~ 8500 decision stumps. However, because our feature vectors are binary, each vector is only $1KB$ of data, the same as ~ 250 real-valued features stored as floats. We are convinced that using a small number of features (or decision stumps) is detrimental to the prediction accuracy based on figures 11, 23 and 25. The number of features computed should be as large as the hardware will allow.

The standard way of capturing color information is computing Color Moments. We noticed that CM features do not take into account whether certain colors are present or absent. For example shades of red are uncommon in upper regions of a correctly oriented outdoor image. If a lower region of an image contained primarily red and green colors (for example a girl in a red sweater lying on grass), the CM features would compute the color as yellow (V' in LUV space or H in HSV space) with large variance whereas two binary indicators for presence of red and green would better describe the scene. On top of that the absence of red is a feature as well, which cannot be captured by CM features. Furthermore to capture the colors well with CM features the grid must align with the place where the color changes. This motivates the use of a spatial pyramid, which the majority of previous attempts at image orientation did not use.

In this thesis we focused on machine learning algorithms that can return a linear weight vector as a classifier. We put importance on the fact that using linear weights is the fastest way to predict orientation and it requires the least space. This allows for the use of linear weights computed with these classifiers as weak learners, or the first level of a final classifier.

In his paper Baluya [2] demonstrated that orientation accuracy depends in large part on the type of images being oriented. The majority of image orientation work has been tested on subsets of the COREL photo gallery which has a large number of photos with an ambiguous correct orientation [24]. That is why we chose to use our own dataset where a human could orient every photograph. We admit that we did not choose a very hard dataset. That allowed us to achieve good accuracy without any rejection (see figure 24).

Some work in image orientation took the approach of computing a large number of features and then selecting a subset of those to use by some criteria. Most common ways of selecting features are PCA and LDA. Vailaya et al. [27] stated

that selecting features (pruning out unnecessary features) might be beneficial because there will be fewer features to capture noise. We tried selecting features using Naive Bayes as opposed to running AdaBoost and found that the latter was far superior (see figure 23).

5 Implementation Notes

One of the goals for this project was to come up with algorithms that could work fast, so everything was implemented from scratch using Java. The images are loaded using standard Java libraries, but everything else is implemented by us. We chose Java because it's portable, cross-platform, wide-spread and faster than R or MatLab. A lot of things that we implemented have implementations in MatLab, but we chose not to use them because of unnecessary memory allocation problems. Even using Java we had to manually call the Garbage Collector to free up memory.

Prior to computing features we scale-down images using a 4-point average per pixel which turns out to be an order of magnitude faster than using any of the Java built-in methods for this. The representation of $C_{RED}, C_{GREEN}, C_{BLUE}$ is a one-dimensional integer array with second, third and fourth bytes (8 consecutive bits) correspond to the eight bits for the red, green and blue channels respectively. We use a one-dimensional array because accessing elements in it requires fewer instructions.

A big optimization step we took was instead of rotating images, and then computing the features from them, we compute the features on only one of the orientations, and then use a rotation function that rotates the feature vector. Rotating

feature vectors (or decision stumps) is linear in the size of the vector. F_{COLOR} features are straight-forward to rotate by re-indexing the spatial pyramid. F_{EDGE} features can also be rotated with ease: for each region, shift (re-index) the orientation bins corresponding to the rotation. It is important that a discrete number of bins correspond precisely to the rotation amount (with 4 orientations, the number of bins must be a multiple of 4).

There is one problem with F_{HAAR} features that has to be pointed out: it is not straightforward to rotate the feature vectors. Because the features are computed using more than one region, it is hard to map features to their rotated counterparts. However it is possible to rotate the spatial pyramid with region pixel-sums precomputed, which is nearly as efficient as rotating the feature vectors.

When computing F_{EDGE} features ideally we would use Gaussian Blur to blur the image, but we just averaged over a 3×3 grid. This did not hurt the quality of the features. For optimization reasons we used the Manhattan Distance instead of Euclidean, and it had no effect on the result as well.

Our dataset is organized in such a way that $\tilde{q} = 1$ (the first orientation is always the correct one). This eliminates the lookup for a label. However it poses a problem: when there's a tie between between two predictions, the one with lowest q will always be picked. For this reason all ties must be resolved at random.

For the Averaged Perceptron algorithm implementation, there is no need to keep track of the average $\mathbf{w}$. It is enough to keep track of the total by adding $\mathbf{w}_t$ times the number of samples it survived to the total. The final $\mathbf{w}$ is then the normalized $\mathbf{w}_{total}$.

For AdaBoost variants, when picking $k_t = \arg\max_k \gamma_k$ both $\mathbf{x}_j$ and $-\mathbf{x}_j$ are considered. This is usually accomplished of learning from $[\mathbf{x}_j; -\mathbf{x}_j]$ (decision stumps and their negatives appended together as one vector). That is a waste of memory. Instead we pick $k_t = \arg\max_k |\gamma_k|$ and use $|\gamma_k|$ instead of γ_k. When we update w_{k_t} we subtract α_t instead of adding it if $\gamma_t < 0$, so the update becomes $w_{k_t} = w_{k_t} + \text{sign}(\gamma_t)\alpha_t$.

For AdaBoost variants, also note that quantities $y_j h_k(j)$ are used a lot and are constant. They can be precomputed and stored into a matrix $\mathbf{U}$ where $u_{j,k} = y_j h_k(j)$.

We ran out experiments on a machine with 64-bit JVM, 8-Thread CPU and 12GB of RAM. We implemented the algorithms ourselves so that they could make full use of the resources. We couldn't implement SVM or find a Java SVM package that would work on 64-bit architecture. The Java Native Interface for SVM_{light} that we used had to be run on a different machine with a 2-Thread CPU and 2GB of RAM.

6 Tuning Parameters

There is a number of constants and algorithm parameters that needed to be defined. We couldn't tune all the parameters at the same time and instead picked some reasonable values, and then repeatedly tuned them one-by-one. F_{COLOR} features and the Perceptron algorithm were used to find the optimal size for an image and the best re-sizing technique. Then using Perceptron Algorithm we tuned the parameters for feature extraction. We limited the size of feature vectors to 3300 for performance considerations. The parameters for machine learning algorithms were tuned with respect to the prediction accuracy achieved on the

$K_{VALIDATION}$ image set. The optimal number of iterations T is computed through cross-validation.

Before any feature extraction or machine learning, certain image acquisition parameters had to be defined, namely what size to down-scale the images to. Because the spatial pyramid sizes we were considering included grids up to 12×12, the size had to be a multiple of 12 and numbers smaller than it. We tried different images sizes and found that the optimal size under 500×500 was 360×360 because 12, 10, 8, 6, 5, 4, 3, 2 and 1 go evenly into it. If we downsized to a resolution of 150×150 which is a common thumbnail size, the prediction accuracy was $\sim 2\%$ lower.

Through experimentation we found parameter values that worked well and then varied the parameters one at a time and fixed them on their optimal values. This process was repeated until the all the parameters settled in a local optimum. The tuning shown in this section assumes all other parameters are set to their optimal values.

6.1 Perceptron Parameter Tuning

The Perceptron algorithm (as described above) returns a different $\mathbf{w}$ depending on how many iterations over the data it is allowed to make. Even if it has already separated the data, with more iterations $\mathbf{w}$ converges to the last value of $\mathbf{w}_t$ and eventually it over-fits the training data. The number of passes over the data T used on K_{TRAIN} is computed through cross-validation.

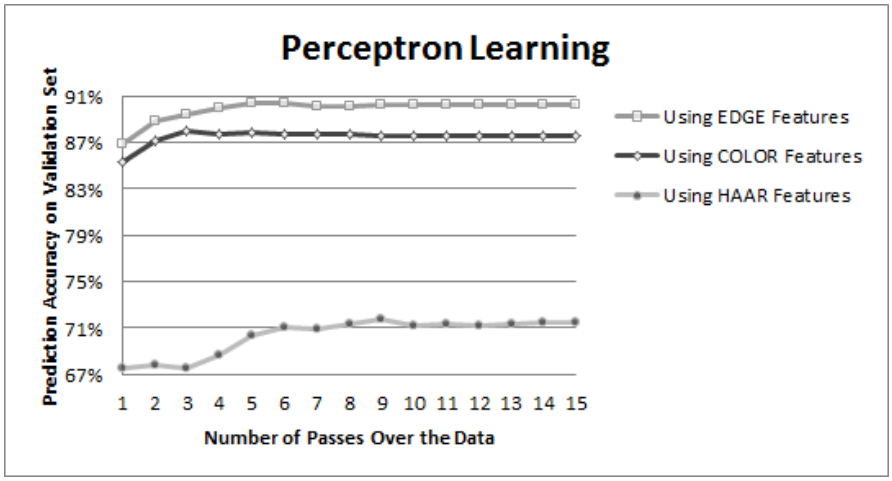

Figure 10: This plot shows the prediction accuracy at every iteration of Perceptron averaged over 25 cross-validation runs.

Even though Perceptron is a simple algorithm, it is capable of separating the $K_{SUBTRAIN}$ set with 100% accuracy. The final value of $\mathbf{w}$ is not the optimal, because it over-fits $K_{SUBTRAIN}$ as is evident by the declining accuracy rates with a larger number of iterations. The optimal value of T, the number of passes over the data throughout our experiments was under 10.

6.2 F_{HAAR} Parameter Tuning

With feature extraction, there's the size of the pyramid that needs to be tuned. There are 10 features per cell that we are looking into, so that means to fit the limit of 3300 we can use a pyramid from 2×2 to 9×9 or a single layer pyramid with grid size of 18×18. The best results were achieved using a spatial pyramid with the maximum grid size of 9×9.

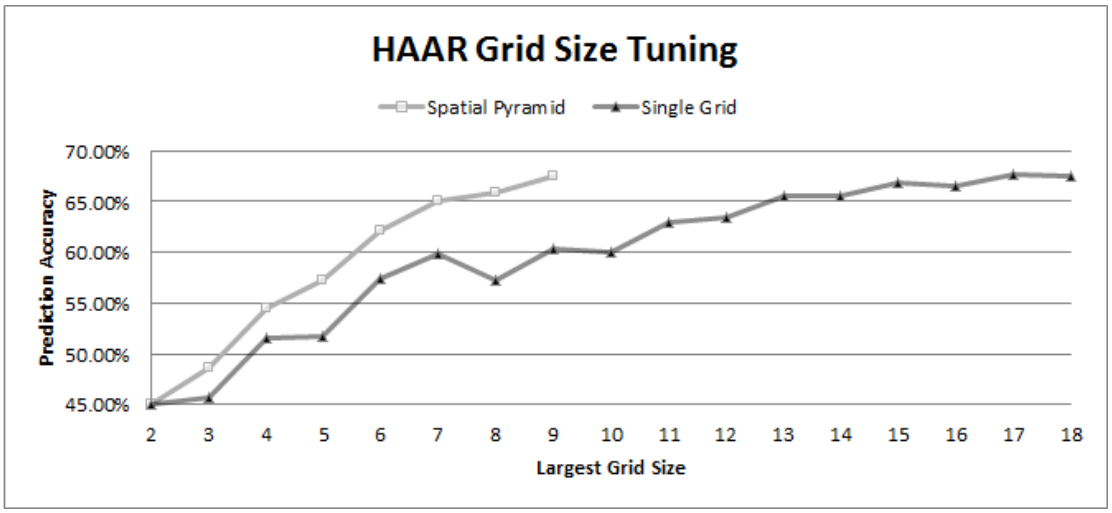

Figure 11: A comparison of using a single grid versus a spatial pyramid when computing F_{HAAR} features. With a limit on the feature vector of 3300, these are the plots.

F_{HAAR} features are very simple, and we expected that a very good feature captured by them would be that outside images are usually brighter on top than on the bottom and Haar-like-wavelets on a 2×2 and 3×3 grids capture it.

6.3 F_{EDGE} Parameter Tuning

Because with F_{EDGE} we have a choice of having more features extracted from a single cell in our experiments using a spatial pyramid was better than a single grid. When tuning the largest grid size for the spatial pyramid, the following relationship exists between the number of degree bins (b) and the grid size (g):

$$b = \left\lfloor \frac{6 \times 3300}{g(g+1)(2g+1)} \right\rfloor$$

There is a trade-off between how many layers of the pyramid and how many degree bins can be used.

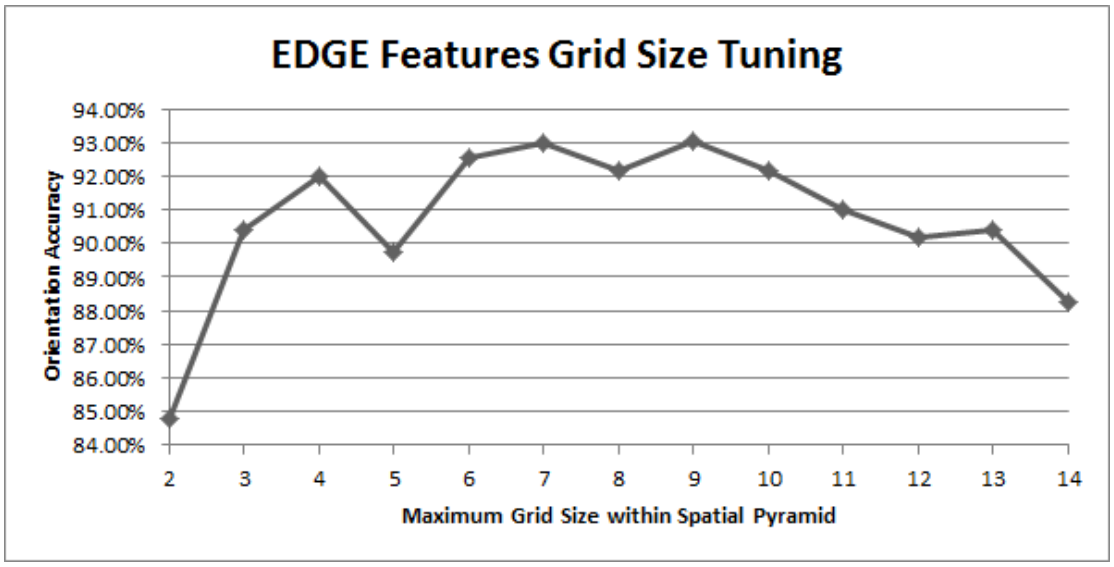

Figure 12: This plot shows the trade-off between using a finer grid size and having more orientation bins per cells.

The optimal combination came out to be $g = 9$ so $B_S = 2$ and $B_B = 9$ and the corresponding number of degree bins $b = 10$.

6.4 F_{COLOR} Parameter Tuning

There is a limit on the size of the feature vectors of 3300, so there's a trade-off between how many levels of the spatial pyramid we can use and how many values per channel we can index. The best results were obtained using $B_B = 6$ (Largest level in the pyramid is a 6×6 and $b = 3$ (three colors per channel and total colors $= 3^3 = 27$).

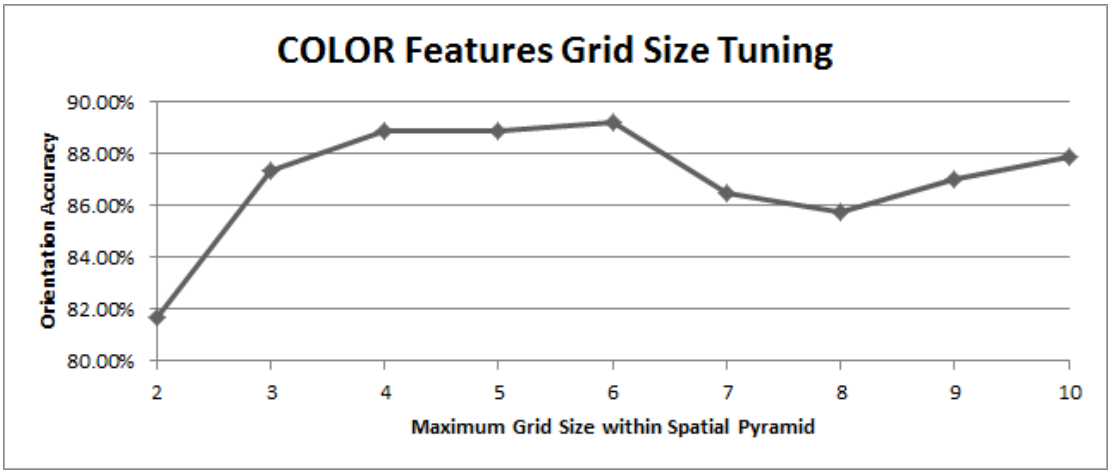

Figure 13: There's a trade off between the number of possible colors and how fine the grid is when computing presence of colors.

After grid size of 10, just having 2 colors per channel (so a total of 8 colors) results in a vector that's over the limit. It is important to note that it's possible to achieve better accuracy with larger feature vector. We can achieve 2% better accuracy with $b = 4$ and 4500 decision stumps.

6.5 AdaBoost with Capped α Parameter Tuning

We limited α to α^C to slow down AdaBoost. For AdaBoost with capped α there is a parameter α^C that needs to be tuned.

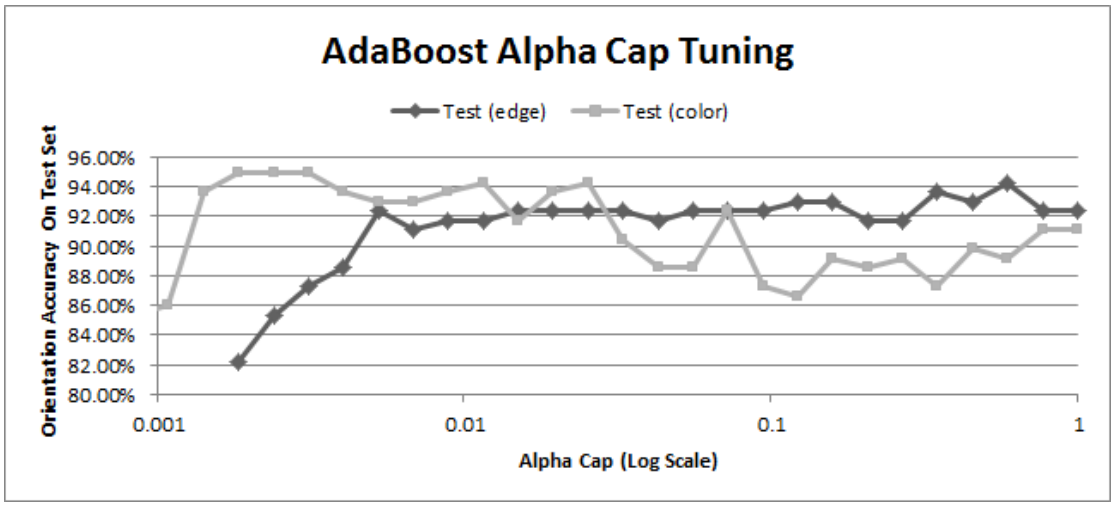

Figure 14: Tuning α^C on F_{COLOR} and F_{EDGE} features. They have features of different strength, so the optimal values differ a little. This particular plot was generated on a shuffle with an easier test-set, so optimal accuracy values are high.

For different features α^C has different optimal values. We hypothesize that F_{COLOR} features have some strong features that do not generalize well and they should not have a large influence and so a small α^C is better. F_{EDGE} features on the other hand do not have that problem so α^C can take on a variety of values and still maintain relatively same accuracy. We fixed it at $\alpha^C = .025$.

6.6 SoftRankBoost Parameter Tuning

SoftRankBoost has one parameter δ that needs to be tuned. This parameter is a measure for how close to the optimal soft margin the algorithm will converge to. We have noticed only a negligible change in accuracy by varying δ. The time for convergence of SoftRankBoost is inversely-proportional to δ^2 so we chose a value on the high end. The range for δ is between 0 and 1. Based on figure 15 we fixed $\delta = 0.9$ for the rest of the experiments.

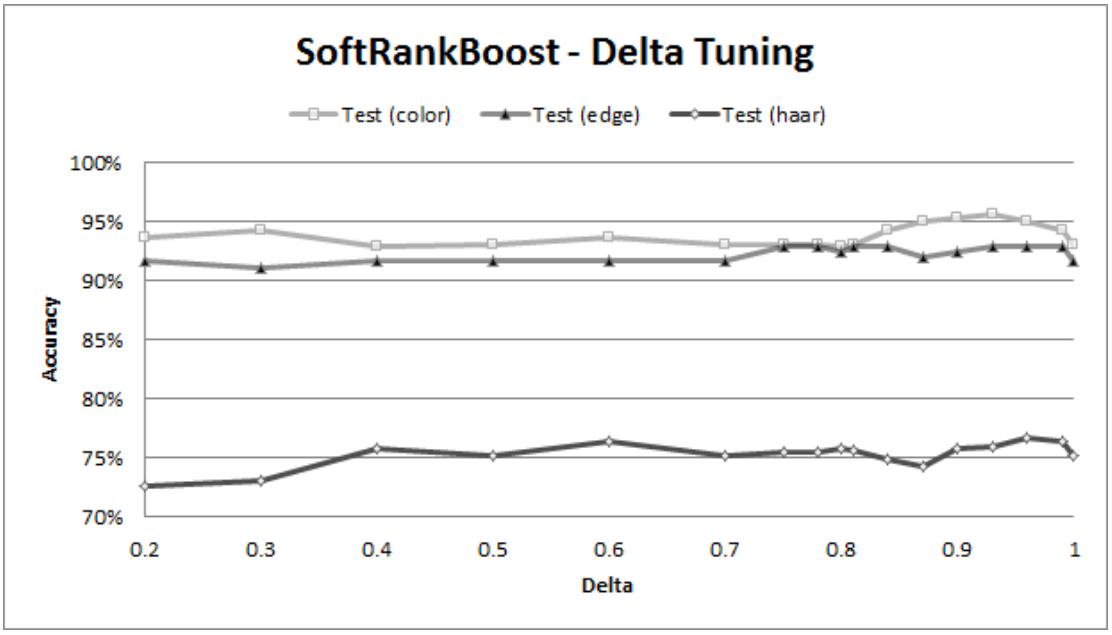

Figure 15: Tuning δ for SoftRankBoost.

6.7 Logistic Regression Parameter Tuning

Logistic Regression implemented with Gradient descent has one parameter that needs to be fixed: λ - the regularization factor. The maximum number of iterations of the outer-most loop T is tuned during cross-validation.

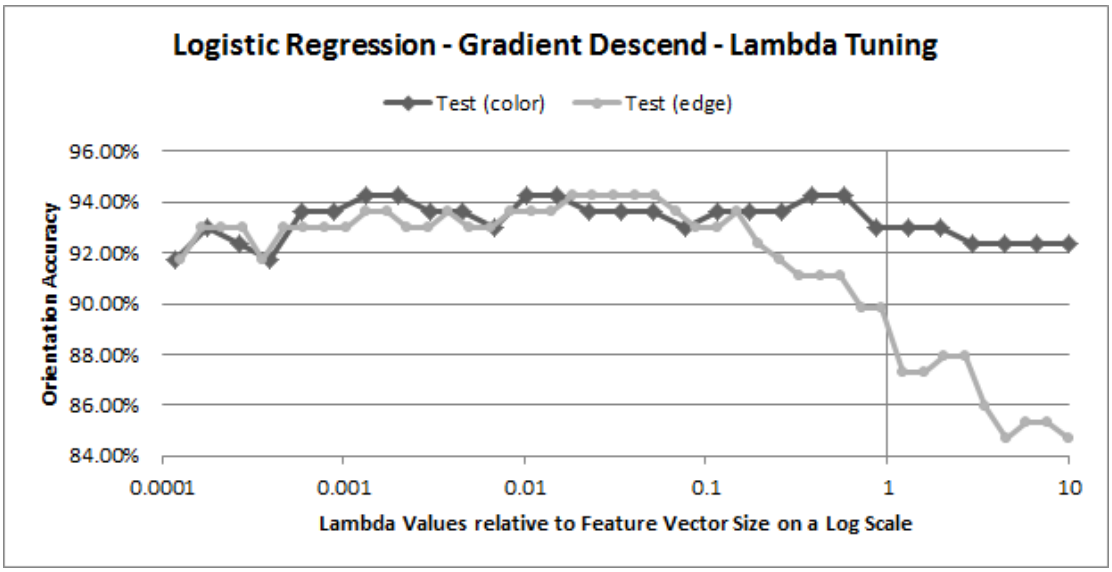

Figure 16: Tuning λ for Gradient descent for Logistic Regression. Actual λ used is this value multiplied by the number of the decision stumps.

The accuracy seems nearly constant for values $\in [.001, .01]$ relative to the size of the decision stumps. Since smaller value λ result in slower convergence, we chose the higher value and set $\lambda = .01\times$ the size of the feature vector.

6.8 SVM$_{light}$ Parameter Tuning

For linear SVMs the only one parameter that can be tuned is c - the trade-off between training error and margin. We varied the value of c and recorded the results.

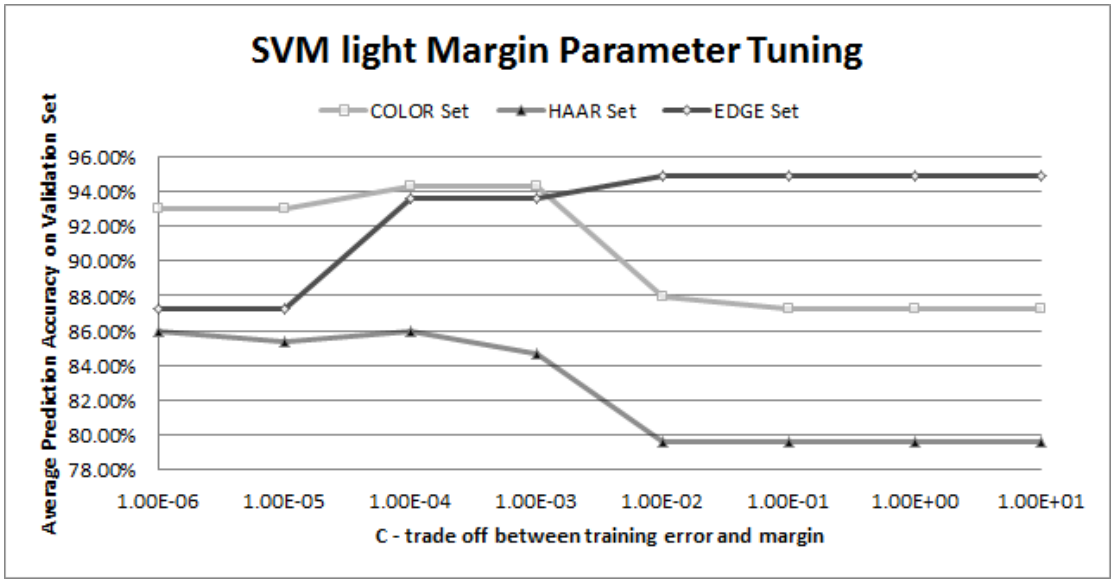

Figure 17: The plot shows average accuracy on the validation set using different values for regularization parameter for SVM$_{light}$.

Based on figure 17 we set $C = 0.0005$.

7 Experiments

The results we report in this thesis are averages computed from a number of experiments. Images were split into 75% and 25% for K_{TRAIN} and K_{TEST} respectively. At least 5-Fold-Cross-Validation was used, so optimal values of T are based on averages of at least 5 training sessions. For performance considerations we limited the size of the binary feature vectors $\mathbf{x}_j$ to 3300. This limit was chosen so that we could append three feature vectors together (3300 $\times$ 3 $\times$ 628 images $\times$ 4 orientations $\times$ 64 bits = 1.5 GB) and it could fit into the 2GB RAM machine we ran experiments on.

The goal of our experiments was to see how well the algorithms classified orientation using feature vectors computed from different feature sets. For each feature set we permuted the images and recorded the accuracy and number of iterations for each of the machine learning algorithms. We repeated the experiment 24 times and reported the averages. Because our predictions take the arg max, we could not produce ROC curves, so we can only provide a numerical summary (see figure 18) along with rejection curves.

Average accuracy (24 runs)				
Feature Set:	F_{EDGE}	F_{COLOR}	F_{HAAR}	Combined
# Features:	2850	2430	2840	8120
Perceptron	$92.25 \pm .56\%$	$89.07 \pm .52\%$	$79.30 \pm .96\%$	$\mathbf{96.97} \pm .40\%$
AdaBoost	$90.87 \pm .55\%$	$90.29 \pm .61\%$	$78.45 \pm .87\%$	$96.60 \pm .35\%$
MadaBoost	$91.77 \pm .51\%$	$90.13 \pm .62\%$	$78.56 \pm .84\%$	$96.66 \pm .31\%$
AdaBoost*	$91.35 \pm .47\%$	$90.39 \pm .58\%$	$76.96 \pm 1.08\%$	$\mathbf{96.97} \pm .29\%$
SoftRankBoost	$90.98 \pm .61\%$	$87.70 \pm .65\%$	$77.87 \pm .81\%$	$96.60 \pm .24\%$
LR-GD	$92.94 \pm .45\%$	$\mathbf{91.14} \pm .50\%$	$\mathbf{80.63} \pm .73\%$	$96.82 \pm .24\%$
SVM_{light}	$\mathbf{93.05} \pm .68\%$	$90.87 \pm .58\%$	$79.25 \pm .81\%$	$96.69 \pm 1.04\%$

Figure 18: This table shows the average accuracies achieved with different algorithms on decision stumps computed with different feature sets. Accuracy achieved with Combined features using the SVM algorithm was computed separately and

With a single set of features the best accuracy was achieved using F_{EDGE} and SVMs. Logistic Regression with Gradient Descent performed as well as SVMs (it is within the margin of error). For F_{COLOR} and F_{HAAR} both SVMs and LR-GD gave optimal results and were within the margin of error of each other. Combining the decision stumps from all the feature sets yielded better results with all the algorithms. With combined features all algorithms gave roughly the same accuracy $(96 - 97\%)$ with the best accuracy achieved by AdaBoost*.

Because all the algorithms tested return a linear weight vector, the prediction time is the same. However, the training times were different. The computation time for a single iteration of Logistic Regression with Gradient Descent (LR-GD) is 70ms, roughly double that of AdaBoost and its variants. Both LR-GD and AdaBoost variants had optimal values of T (number of iterations of the outer loop) in the range of $200 - 1500$. Averaged Perceptron has optimal value of T in

the range of $3 - 11$, however one iteration of the algorithm took roughly as much as time as one iteration of LR-GD. So learning with the Averaged Perceptron Algorithm took about two orders of magnitude less time than LR-GD (3s vs 5min including 5-fold cross-validation).

The rest of this section covers the rejection plots in some detail.

7.1 F_{HAAR} Feature Set

With no rejection, LR-GD and SVMs achieve about the same accuracy. With a small rejection rate there is a difference and LR-GD achieves better prediction accuracy. From the boosting family of algorithms SoftRankBoost performs the best. The superiority of SoftRankBoost is evident in the fact that the accuracy increases with rejection. For AdaBoost, MadaBoost and AdaBoost* having $10 - 15\%$ rate of rejection does not increase accuracy which is an indicator that they do not learn good weights.

Based on the Rejection Plot it seems that using LR-GD is optimal. We got similar results with the SVM package. Averaged Perceptron is recommended for speed of learning and ease of implementation if a drop in accuracy of a few percentage points is acceptable. Using boosting with only the F_{HAAR} feature set is not advised.

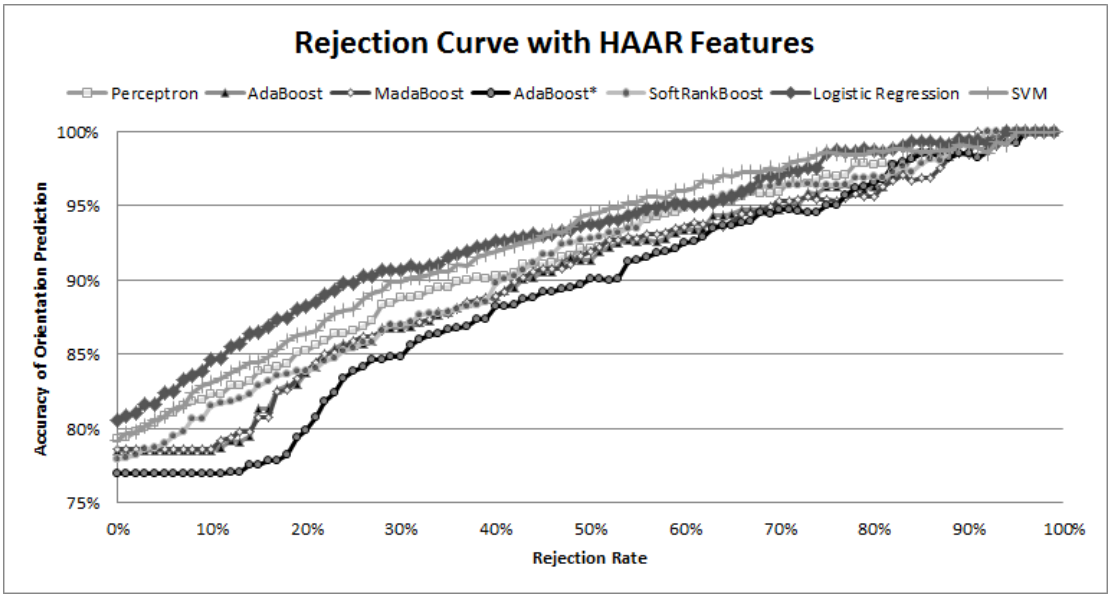

Figure 19: Average accuracies on the F_{HAAR} set using different algorithms with varying levels of rejection.

7.2 F_{COLOR} Feature Set

With no rejection using F_{COLOR} , SVM and LR-GD are the best choice of algorithms. If rejection is expected SoftRankBoost becomes another candidate with accuracy equal to LR-GD and SVM. The worst accuracy is achieved using Averaged Perceptron, which can be explained by a presence of noisy features that Averaged Perceptron cannot filter out.

Based on the Rejection Plot (see figure 20) either one of LR-GD, SVMs or SoftRankBoost can be used interchangeably with F_{COLOR} features. The use of Averaged Perceptron is only advised if a significant loss in accuracy is worth the significant decrease in training time.

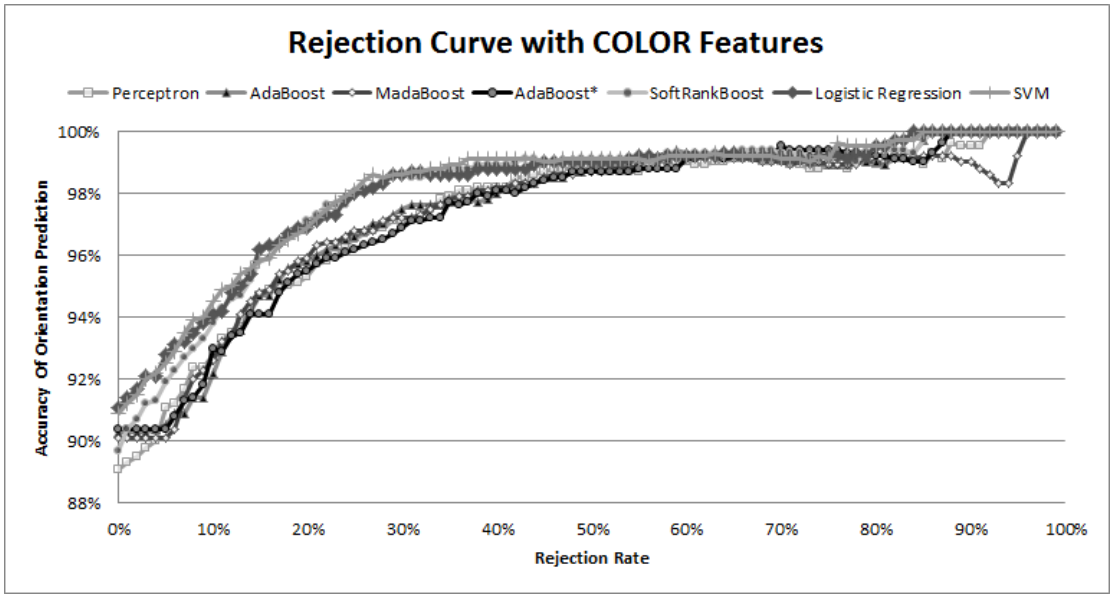

Figure 20: Average accuracies on the F_{COLOR} set using different algorithms with varying levels of rejection.

7.3 F_{EDGE} Feature Set

If only one set of features if usedF_{EDGE} will give the best prediction accuracy. With no rejection 93% accuracy can be achieved using either LR-GD or SVMs. If any rejection is used SVMs is the clear choice. Using Averaged Perceptron is advised if the speed of learning is an issue and a little loss in accuracy is tolerable. LR-GD did not separate the data with a good margin which is evident by a very slow rise in accuracy with rejection compared to the other algorithms.

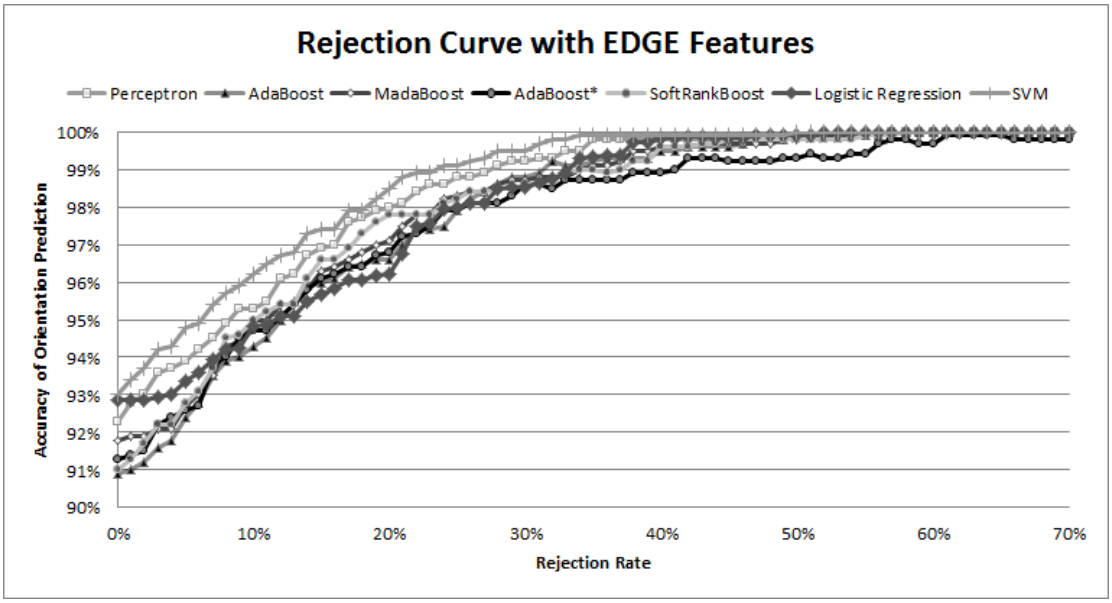

Figure 21: Average accuracies on the F_{EDGE} set using different algorithms with varying levels of rejection.

7.4 Features Appended Together

If the decision stumps from all the features are appended together we get very large vectors to learn from. With no rejection, all the algorithms achieve relatively same accuracy. With some rejection, the accuracy increases dramatically using any of the algorithms and the best accuracy can be achieved with Averaged Perceptron and SoftRankBoost, which is expected since it is designed to maximize the area under such curves.

The algorithm that gave the best accuracy with no rejection and the one to reach near perfect orientation classification with least rejection rate is AdaBoost*, despite the fact that it does not result in good rejection curves when learning single feature sets. The combination of features allows for separation with a larger margin and AdaBoost* uses that information well.

The use of Averaged Perceptron on all decision stumps combined is highly

advised because of the ease of implementation, speed of learning and high accuracy with small rejection rates (under 15%). For best results and statistical guarantees AdaBoost* and SoftRankBoost should be used, however they require parameter tuning. The rejection curves in figure 22 were computed on a 64-bit machine with 12GB of RAM to comfortably have 6 algorithms learn from data with vector sizes of approximately 9000. We were unable to run SVM_{light} on that machine. Instead we ran it on the 32-bit machine with 2GB of RAM, where we were unable to generate a rejection curve. It is probable that SVMs would perform as well as the Perceptron algorithm on figure 22.

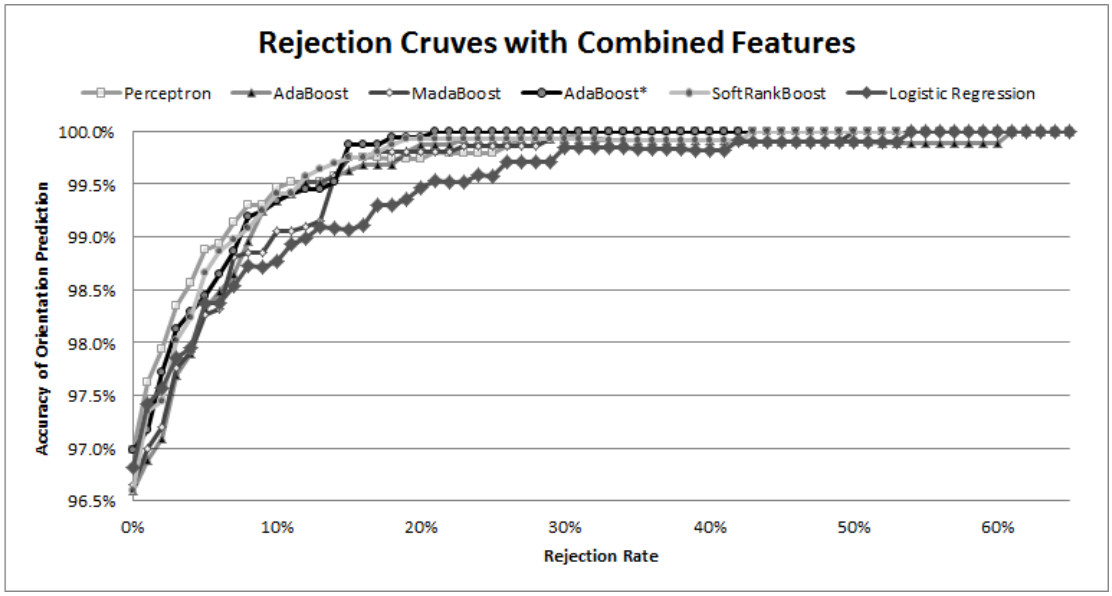

Figure 22: Rejection Curves for Accuracies achieved with all decision stumps combined. Please note that SVM is not included here (SVM_{light} exited stating it was out of memory!).

8 Partially Complete Work

In this thesis we established a base for image orientation. There are many directions in which future work can go. Some of these directions are explored in

this section. We limited the number of decision stumps, combined predictions, predicted 1 out of 12 orientations and used the best method so far on a new dataset.

8.1 Feature Selection

When we combined feature vectors from different features sets we got new feature vectors of size 8120. Not all features were used by the classifiers ($\mathbf{w}_k = 0$). We tried selecting l features from each feature set for a total number of $3l$.

We considered two methods of feature selection: selecting the first l decision stumps chosen by AdaBoost (without α capping) and selecting the best l decision stumps chosen by Naive Bayes (selecting based on $|\ \mathbf{y} \cdot \mathbf{h}_k|$).

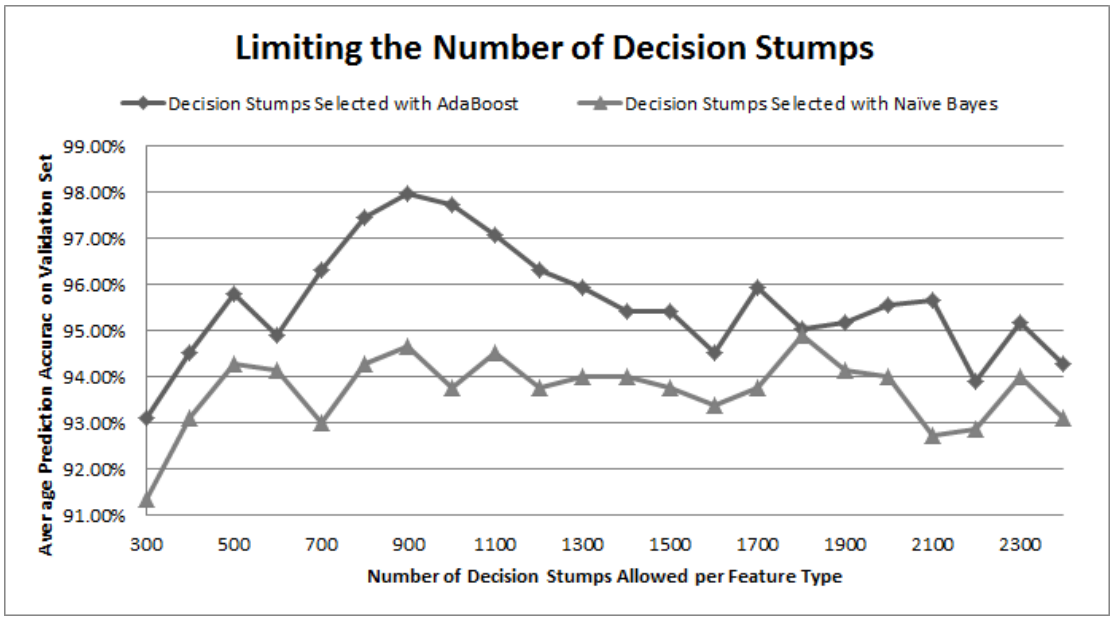

Figure 23: This plot shows accuracy achieved on the validation images using Logistic Regression with features combined from every feature set with at most l decision stumps from each. The two curves are showing selecting decision stumps with AdaBoost and Naive Bayes.

Limiting the number of decision stumps with AdaBoost [32] was shown to be

beneficial and overall better than limiting with Naive Bayes. We believe that choosing only $l = 500$ decision stumps per feature set still results in comparable prediction accuracy on the validation set. When selecting decision stumps with AdaBoost using $l = 900$ was optimal. We then took the 2700 $(900 \cdot 3)$ decision stumps, did 12 experiments and recorded the average. Out of the algorithms that we tried, Logistic Regression performed the best with an accuracy of $\mathbf{98.99 \pm .29}\%$. SVM was a close second with a few percent of rejection. With more than 5% of rejection any algorithm other than AdaBoost with capped αs could be used to achieve near-perfect 99% accuracy.

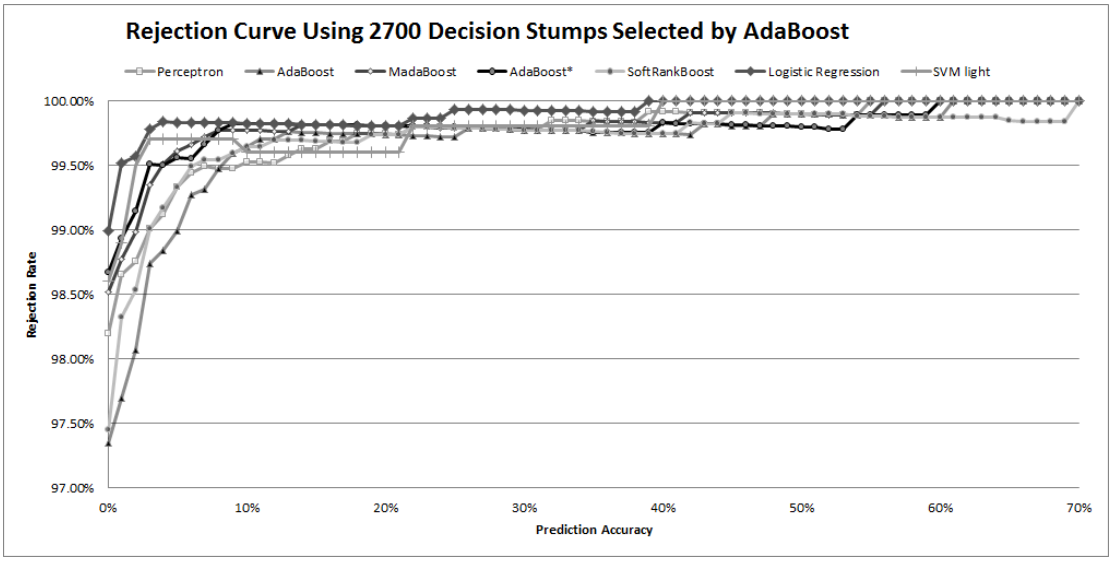

Figure 24: These rejection curves are for **w** learned with 2700 decision stumps (900 from each feature set selected with using original AdaBoost [10] (no α capping)) with average accuracies out of 12 experiments.

It should be noted that for individual feature sets, selecting some sub-space of decision stumps is not beneficial at all. This is evident from figure 25. Compared to other works with similar features [25, 19, 13, 32, 24] we used large vectors (i.e. many decision stumps). A direct relationship between the number of features and accuracy was observed. Therefore the maximum number of features should be

used when using a single feature set.

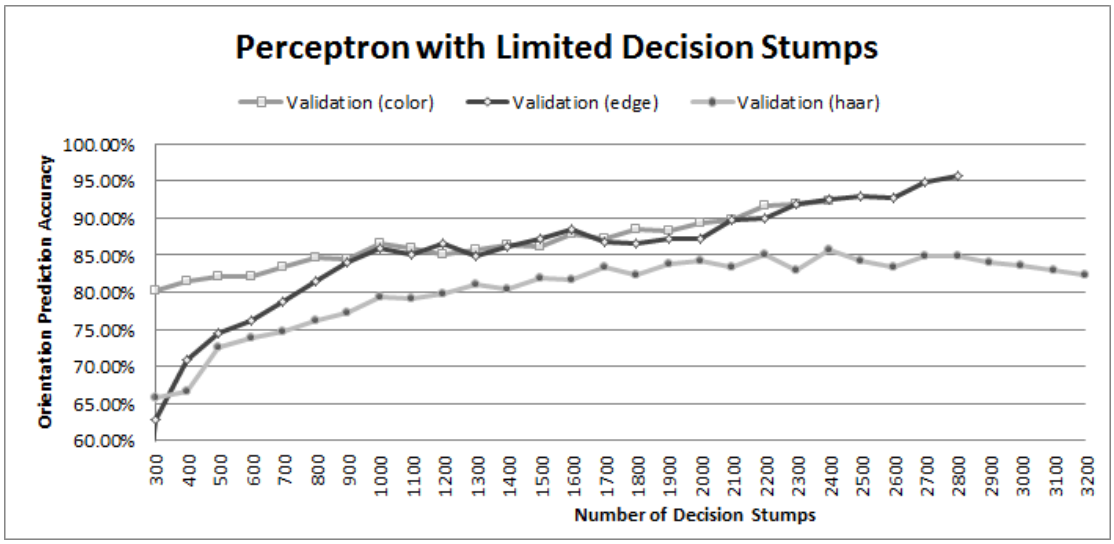

Figure 25: These plots show accuracy on $K_{VALIDATION}$ with different features sets with varying l - maximum number of decision stumps used. The curves are averages from the 5 cross-validation runs using the Perceptron Algorithm.

8.2 Combining Predictions

Instead of appending decision stumps together, it is possible to train classifiers on each feature set separately and then combine predictions. We trained using Logistic Regression and got base accuracies of 93.0%, 94.3% and 80.9% with F_{EDGE} , F_{COLOR} and F_{HAAR} feature sets respectively.

We used the original AdaBoost [10] to boost the resulting **w**s (from Logistic Regression). The accuracy of the final classifier was 91.7%, which is lower than the 94.3% obtained using **w** computed from F_{COLOR} alone. We tried improving boosting accuracy by adding more hypotheses (optimal **w** from each of cross-validation runs), but the final classifier's accuracy remained the same.

Then we combined predictions using a measure of confidence. We defined a con-

fidence of prediction as $c = \frac{s_1 - s_2}{s_1 - s_Q}$ where $s_1 = \max_q \mathbf{w} \cdot \mathbf{x}_i^q, s_Q = \min_q \mathbf{w} \cdot \mathbf{x}_i^q$ and s_2 is the second highest dot product between $\mathbf{w}$ and $\mathbf{x}_i^q$ where $\mathbf{x}_i^q$ is the score of the second candidate. Setting the final prediction for an image i as the prediction with the highest confidence, we got an accuracy of 95.5%. If we look at predictions from F_{COLOR} and F_{EDGE} we get an accuracy of 97.5%. In fact, including predictions from F_{HAAR} always hurts the final classifier accuracy. Instead of taking the prediction with the highest confidence, we considered predicting using the sum of confidences and got the exact same result.

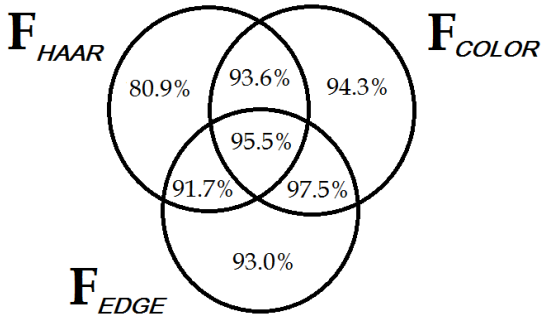

Figure 26: This diagram shows prediction accuracies for classifiers trained on F_{COLOR} , F_{EDGE} and F_{HAAR} decision stumps and their combinations (chosen by max prediction confidence).

8.3 More than 4 orientations

Everything we described in this thesis can be generalized beyond $Q = 4$ orientations. The limitation for four orientations comes from the fact that we are using square grids. A rotated grid must align with other grids. With square grids there are only four orientations where that happens. The solution to having more orientations ($Q > 4$) is to use circular grids (grids in polar coordinate system).

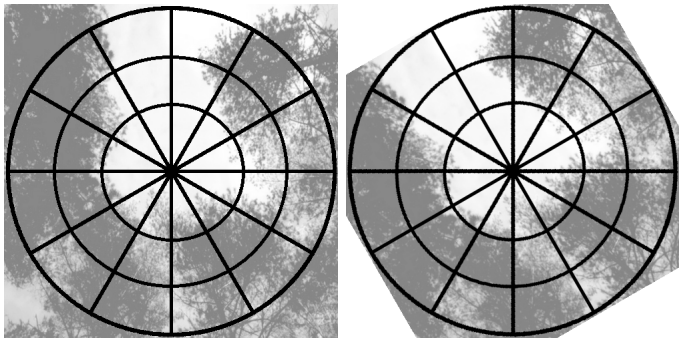

Figure 27: An example of a polar grid. Images shown are of a single polar grid with 12 angular divisions and 3 radial division positioned over an image with 0° and 330° rotation.

The downside to using polar grids is that the corners of the image are not used for feature extraction. On the other hand images can be oriented with higher precision. Polar grids can have small or large cells, so features can be computed within a spatial cone (as opposed to spatial pyramid). Only one feature vector (or one vector of decision stumps) has to be computed. The feature vectors (or decision stumps) for other orientations can be computed by re-arranging values. In that respect, computing features from images with any number of orientations can be done in constant time with respect to number of orientations. Note that computing time the first feature vector overshadows any feature-rotation.

We used F_{COLOR} features on a single polar grid with 12 angular divisions and 3 radial divisions (as in figure 27). We set $Q = 12$ and left everything else the same way. Resulting prediction accuracies were between 50% and 60%.

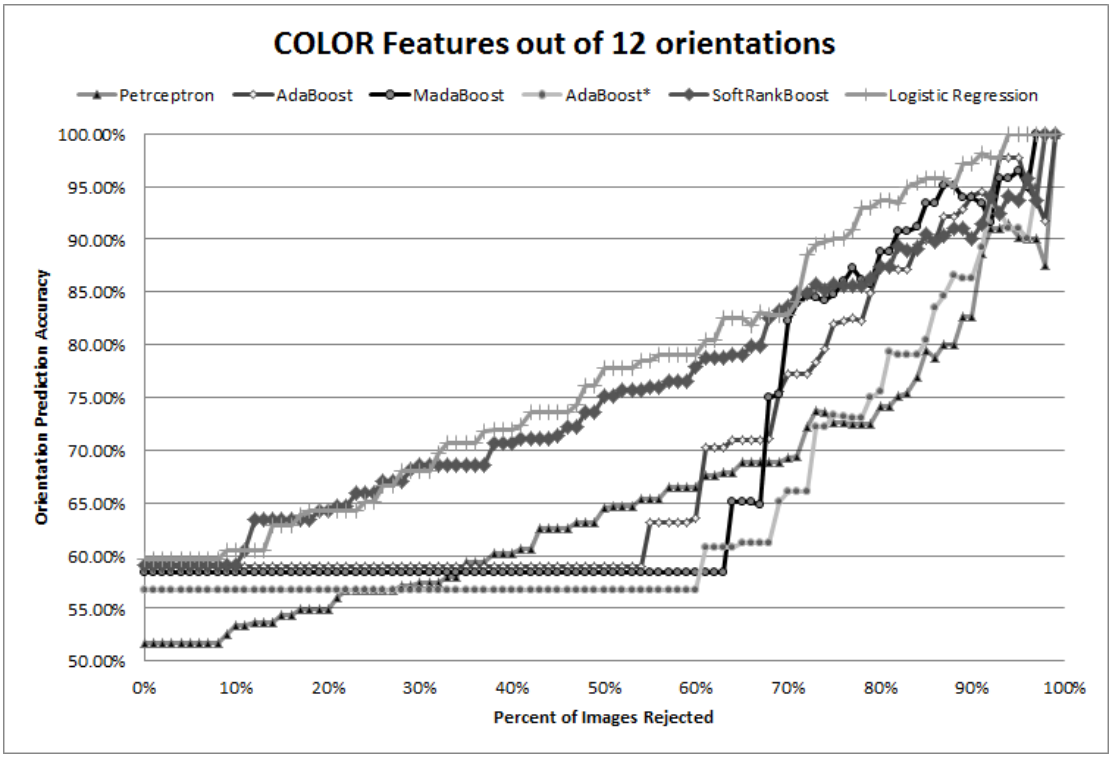

Figure 28: This plot shows an average accuracy from 10 permutations of data using F_{COLOR} features and 12 candidate orientations. Logistic Regression and SoftRankBoost clearly are best for this task. AdaBoost, MadaBoost and AdaBoost* performed poorly compared to the other algorithms because they were not any less confident about wrong predictions (the accuracy does not increase with rejection). The Perceptron algorithm predicted with bad accuracy, because it only learns against one other orientation as opposed to one versus all. An interesting thing to note is that most wrong predictions were within 30° of the correct one.

Randomly guessing out of 12 orientations would yield an accuracy of 8.3%. In comparison, the results we got are much better. There is still a lot of room for improvement.

8.4 A Different Dataset

The dataset we used thus far contained images by the same photographer in the same geographical region that is representative for a problem of photo orientation

for personal use. For other problems (like handling image upload) a different dataset is more representative. We collected 609 images from popular image search engines by using keywords such as: outdoor, valley, outside, nature and national geographic. We collected thumbnails of the images (smallest side had size of 150 pixels). All the images had to meet the same criteria as our original dataset: they must be taken outside during daylight and pieces of background must be visible.

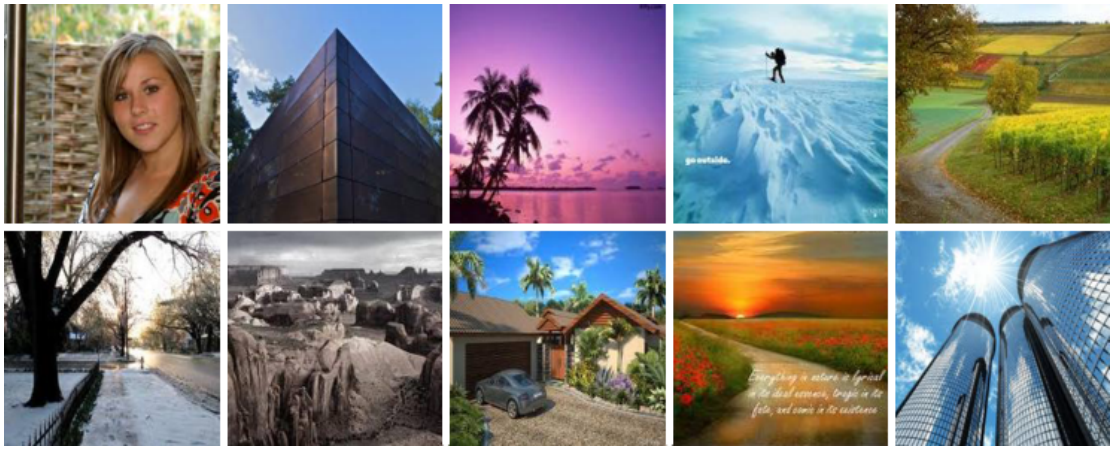

Figure 29: Examples of images in the dataset acquired from the Internet.

The method that worked before - Logistic Regression learning from 2700 decision stumps was tried it on this Internet dataset. We got an accuracy of $94.55 \pm .56\%$ on K_{TEST} . From the Rejection Curves(see figure 30) it is evident that with 30% rejection rate Logistic Regression can achieve 100% accuracy. For fast learning, Averaged Perceptron is a good choice because it can achieve 99% accuracy with 20% rejection.

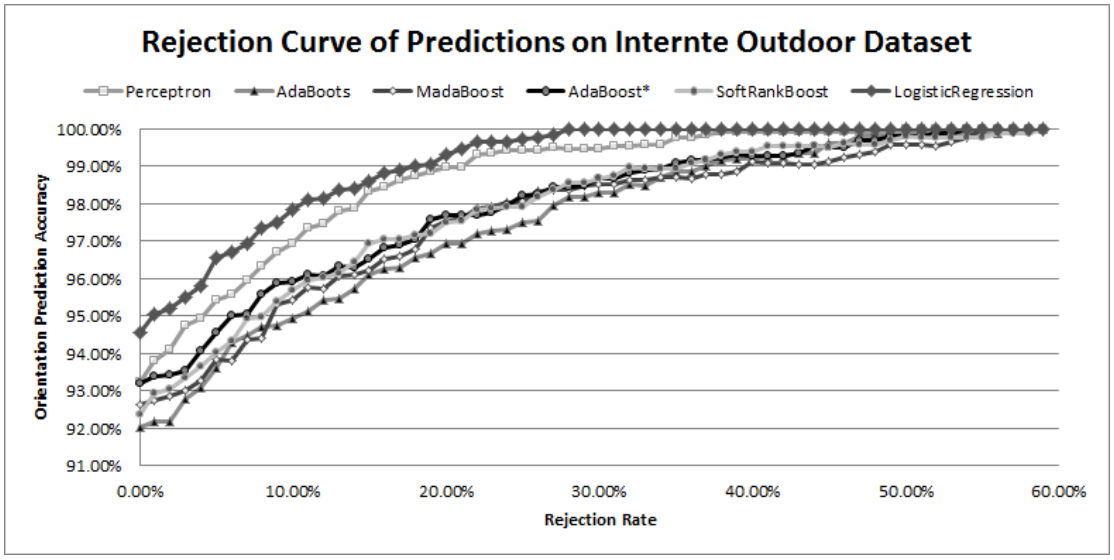

Figure 30: Rejection Curves for Prediction Accuracy with machine learning algorithms training on 2700 decision stumps composed of 900 per feature set chosen by AdaBoost.

9 Future Work

In this thesis we covered basic automatic image orientation on an easy dataset. There are many ways in which this work can be built upon. There are more features and other algorithms that can be tried. Features and algorithms can be combined to make stronger features and algorithms. More special purpose and generalized automatic image orientation techniques can be explored.

A simple way to generate new features would be to combine existing features. Features can be combined in a linear fashion with subtraction/addition and with binary operators. Other ways to get low-level features include: phase symmetry, Hough transforms, colored-edges, texture, color values in different color spaces etc. High level features (semantic cues) can be used for special-purpose image orientation such as: recognizing text, faces or landmarks.

If a lot of features are used, there is a need for good feature selection. There

are many ways of feature selection that can be tested like PCA, LDA or selecting decision stump k if w_k is higher than threshold where $\mathbf{w}$ is computed using one of the algorithms described in this thesis. It may be beneficial to first append all decision stumps together and then select the best ones from them.

There is a lot of potentially rewarding work that can be done with prediction combination. It is possible to combine predictions with SVMs and other AdaBoost variants such as Entropy Regularized LPBoost [11]. There are other ways to calculate prediction confidence other than taking the dot product of $\mathbf{w}$ and $\mathbf{x}_i^q$ and the other methods described here. Lumini and Nanni [19], Luo and Boutell [13] and Zhang et Al [32] covered substantial work combining predictions for image orientation. An area of interest might lie in combining different prediction combination techniques with different rejection techniques.

For general image orientation future work involves trying specialized rejection schemes for certain types of images and a more complicated automatic image orientation framework. Excellent results have been achieved in a limited-class setting [8] by first classifying images into a number of classes and then orienting them within that class. General-purpose image orientation techniques should combine classification, semantic cue recognition and rejection schemes for optimal orientation prediction results.

As a new approach to image orientation detection with more than 4 possible orientations, exploring more about using polar grids is an interesting topic. Future work in this area might include trying different polar grid sizes and a spatial cone grid, trying F_{EDGE} features, trying a polar version of F_{HAAR} features, combining features or having error-tolerance of some number of degrees (for example training using 12 orientations, and predicting out of 4).

10 Conclusion

A method for orienting images was proposed that consisted of first extracting decision stumps and then training using machine learning algorithms that return a linear weight vector. Three types of features were considered: F_{HAAR} (presence of strong light-dark contrast), F_{COLOR} (color presence) and F_{EDGE} (presence of directional edges). The following machine learning algorithms were considered: Averaged Perceptron, AdaBoost (with limited α), MadaBoost, AdaBoost*, SoftRankBoost, Logistic Regression and an implementation of SVMs (SVM_{light}).

A dataset consisted of 628 images. Each image was taken with the following criteria: taken outdoors during daylight, had a visible background and could be oriented by a human. Binary feature vectors were computed from images at the resolution of 360×360. Local features were computed in the context of a spatial pyramid with the largest grid size such that the total number of decision stumps was under 3300 per feature set.

As a single set of features F_{HAAR} features were the worst. We were only able to achieve 80.6% prediction accuracy with Logistic Regression. If we allowed a 50% rejection rate, using SVM we could get 94.4% prediction accuracy. F_{COLOR} features with Logistic Regression had $91.1\%, 96.9\%$ and 99.1% accuracy with $0\%, 20\%$ and 50% rejection rate respectively. F_{EDGE} features with SVMs gave $93.0\%, 98.5\%$ and an 100.0% accuracy with $0\%, 20\%$ and 50% rejection rate respectively. Combining all the decision stumps together (8120) gave accuracy of 97.0% using AdaBoost* and accuracy of 100% with only 21% rejection rate.

The best results (99.0% accuracy with no rejection) were achieved by selecting (with the original AdaBoost) 900 decision stumps from each feature set and then training with Logistic Regression on all 2700 of them. However this approach does not yield 100% without a 40% rejection rate.

Throughout our experiments, using Logistic Regression yielded highest or nearly highest prediction accuracies. If the training time is an issue, then Averaged Perceptron is the definitive choice because it trains in two orders of magnitude less time. If a rejection scheme must be used, we find SoftRankBoost to be the appropriate algorithm because it works well with rejection schemes and mostly gives better accuracy than all other boosting variants with some rejection.

Additional work was done in the following areas: trying more than 4 orientations, building a second-layer classifier on top of the algorithms presented here, using a better feature selection method and trying a different dataset. There is possible future work that can be done in automatic image orientation including trying new features, trying new algorithms, using different rejection schemes and confidence measures.

References

[1] Erin L Allwein, Robert E Schapire, and Yoram Singer. Reducing multiclass to binary: A unifying approach for margin classifiers. *Journal of Machine Learning Research*, pages 113–141, 2000.

[2] Shumeet Baluja and Google Inc. Large scale performance measurement of content-based automated image-orientation detection. In *IEEE International Conference on Image Processing*, pages 514–517, 2005.

[3] Christopher M. Bishop. *Pattern Recognition and Machine Learning*, chapter 4, pages 205–206. Springer, 2006.

[4] Leon Bobrowski. *Linear Ranked Regression - Design Principles*, chapter Advances in Soft Computing, pages 105 – 112. Springer, 205.

[5] Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. In *In CVPR*, pages 886–893, 2005.

[6] Manasi Datar and Xiaojun Qi. Automatic image orientation detection using the supervised self-organizing map, 2005.

[7] Carlos Domingo and Osamu Watanabe. Madaboost: a modified version of adaboost. In *Thirteenth Annual Conference on Computational Learning Theory*, pages 180–189, 2000.

[8] Dong Wang et Al. System and method for automatic digital image orientation detection, 2009.

[9] Yoav Freund and Robert E Schapire. *Large Margin ClassificationUsing the Perceptron Algorithm*, chapter 37, pages 277–296. Kluwer Academic, 1999.

[10] Yoav Freund and Robert E Schapire. A short introduction to boosting. *Journal of Japanese Society for Artificial Intelligence*, 14:771–780, September 1999.

[11] Karen A. Glocer. *Entropy Regularization and Soft Margin Maximization.* PhD thesis, University of California, Santa Cruz, 2009.

[12] Jun ichi Moribe, Kohei Hatano, Eiji Takimoto, and Masayuki Takeda. Smooth boosting for margin-based ranking. In *17th International Conference on Algorithmic Learning Theory*, pages 227–239, 2008.

[13] Matthey Boutell Jiebo Luo. Automatic image orientation detection via confidence-based integration of low-level and semantic cues. In *IEEE Transactions on Image Processing*, volume 27, pages 715–727, 2005.

[14] Thorsten Joachims. *Making large-Scale SVM Learning Practical*, pages 40–56. MIT-Press, 1999.

[15] Thomas Kersten and Silvio Haering. Automation of interior, relative, and absolute orientation. *ISPRS journal of photogrammetry and remote sensing*, 52, 1997.

[16] Minsky M L and Papert S A. Peceptrons. *IEEE ASSP Magazine*, pages 4–22, 1969.

[17] Svetlana Lazebnik and Cordelia Schmid. Beyond bags of features: Spatial pyramid matching for recognizing natural scene categories. In *CVPR*, pages 2169–2178, 2006.

[18] Wei Chao Lin, Michael Oakes, and John Tait. Real adaboost for large vocabulary image classification, 2008.

[19] Alessandra Lumini and Loris Nanni. Detector of image orientation based on borda count. *Pattern Recognition Letters*, pages 180–186, 2006.

[20] Christina Pavlopoulou and Stella X Yu. Indoor-outdoor classification with human accuracies: Image or edge gist? In *CVPR*, 2010.

[21] Andrew Payne and Sameer Signh. Indoor vs. outdoor scene classification in digital photographs. *The Journal Of The Pattern Recognition Society*, pages 1533–1545, 2005.

[22] Gunnar Ratsch and Manfred K Warmuth. Efficient margin maximizing with boosting. *Journal of Machine Learning Research*, pages 2131–2152, 2005.

[23] Jason Rennie. Boosting with decision stumps and binary features, 2003.

[24] Zhou Su. Automatic image orientation detection. Master's thesis, University of Sheffield, 2004.

[25] Ekaterina Tolstaya. Content-based image orientation recognition, 2007.

[26] Aditya Vailaya, Mario A T Fagueiredo, Anil K Jain, and Hong Jiang Zhang. Image classification for content-based indexing. In *IEEE Transactions on Image Processing*, volume 10, January 2001.

[27] Aditya Vailaya, Hongjiang Zhang, Senior Member, Changjiang Yang, Feng-I Liu, and Anil K. Jain. Automatic image orientation detection. In *IEEE Transactions on Image Processing*, pages 600–604, 2002.

[28] Paul Viola and Michael Jones. Rapid object detection using a boosted cascade of simple features. In *2nd Intl. Workshop on Statistical and Computational Theories of Vision*, pages 511–518, 2001.

[29] Lei Wang, Xu Liu, Lirong Xia, Guangyou Xu, and Alfred Bruckstein. Image orientation detection with integrated human perception cues (or which way is up). In *Proceedings of Int. Conf. on Image Processing*, pages 539–542, 2003.

[30] Yongmei Wang and HongJiang Zhang. Content-based image orientation detection with support vector machines, 2001.

[31] Rong Xiao, Lei Zhang, and Hong-Jiang Zhang. Feature selection on combinations for efficient learning from images, 2004.

[32] Lei Zhang, Mingjing Li, and HongJiang Zhang. Boosting image orientation detection with indoor vs. outdoor classification, 2002.