UiO : Department of Informatics
University of Oslo

Computer-Aided Screening of Capsule Endoscopy Videos

Zeno Albisser

Master's Thesis Autumn 2015

Computer-Aided Screening of Capsule Endoscopy Videos

Zeno Albisser

August, 2015

Acknowledgements

I would like to express my gratitude to my supervisors, Pål Halvorsen and Michael Riegler, for the opportunity of working on this research topic, for all the valuable advice I have received and for reviewing my work.

Further, I would like to say thank you to my fiancée Olga, my parents Beatrice & Urs and my sister Brigitte, who always support and encourage me.

Abstract

Colon cancer accounts for almost 10% of all cancer cases worldwide. It is also the fourth most common cause of death from cancer globally. However, many cases of colon cancer could be prevented by early screening and removal of colon polyps - a common precursor of colon cancer. In this respect, capsule endoscopy is a non-invasive screening method with the potential to significantly reduce the cost of screening as well as the discomfort caused for the patient using traditional endoscopy examination. The financial cost of evaluating the recorded video footage, as well as the availability of specialists, currently prevents the deployment of capsule endoscopy for mass screening.

With this work, we research solutions for automating the evaluation of capsule endoscopy video sequences using machine learning, image recognition and extraction of global image features. Rather than focusing on a single approach, we build tools that can be used for conducting further experiments with different methods and algorithms. We present the prototype of an integrated software solution that can be used for collecting videos from hospitals, annotating videos, tracking objects in video sequences, building training and testing datasets, training classifiers and eventually, testing and evaluating the generated classifiers.

We evaluate our software by training classifiers that are based on three different image recognition approaches. We also test the generated classifiers with different datasets and thereby evaluate the different approaches for their feasibility of being used to recognize colon polyps.

Our main conclusion is that state of the art image recognition methods, such as the use of Haar-features or Histogram of oriented Gradients based detectors, are not suitable for detecting lesions in the intestine because of the enormous variety of possible appearances and orientations of such lesions. Global image features such as Joint Composite Descriptor on the other hand, lead to very promising results. Performing leave-one-out-cross-validation with all 20 videos of the ASU-Mayo Clinic polyp database, our system achieves a weighted average precision of 93.9% and a weighted average recall of 98.5%.

Contents

1 Introduction **1**
- 1.1 Background . . . 1
  - 1.1.1 Screening Methods . . . 1
  - 1.1.2 Computer Aided Diagnosis . . . 4
  - 1.1.3 Collecting Data for Training and Testing . . . 4
- 1.2 Problem Definition / Statement . . . 5
- 1.3 Limitations . . . 5
- 1.4 Research Method . . . 5
- 1.5 Main Contributions . . . 6
- 1.6 Outline . . . 7

2 Related Work **9**
- 2.1 Building a Training Dataset for Machine Learning . . . 9
  - 2.1.1 Video Segmentation . . . 9
  - 2.1.2 Video Annotation . . . 11
  - 2.1.3 Object Tracking . . . 13
- 2.2 Frame Reduction . . . 14
- 2.3 Methods for Automatic Detection of Colon Cancer . . . 15
  - 2.3.1 Geometrical Analysis . . . 15
  - 2.3.2 Machine Learning . . . 17
  - 2.3.3 Index of Global Image Features . . . 21
  - 2.3.4 Virtual Colonoscopy . . . 21
- 2.4 Summary . . . 22

3 Object Tagging **25**
- 3.1 Design and Implementation . . . 26
  - 3.1.1 Prototype 1: Object Tagging and Manual Tracking . . . 26
  - 3.1.2 Prototype 2: Web-Based Object Tagging . . . 27
- 3.2 Evaluation and Discussion . . . 30
- 3.3 Summary . . . 30

4 Object Tracking **31**
- 4.1 Design and Implementation . . . 31
  - 4.1.1 Object Tracking in JavaScript . . . 31
  - 4.1.2 Object Tracking in Google Native Client . . . 32
  - 4.1.3 Object Tracking on the Server Side . . . 32
  - 4.1.4 A Native Tool for Object Tracking . . . 33
- 4.2 Evaluation and Discussion . . . 35
- 4.3 Summary . . . 36

5 Preprocessing and Image Filtering **37**
5.1 Design and Implementation 37
5.1.1 Border Detection 37
5.1.2 Edge Detection 39
5.1.3 Shape Detection 40
5.1.4 Sepecular Highlight Filtering 41
5.2 Evaluation and Discussion 43
5.2.1 Border Detection 43
5.2.2 Edge Detection 44
5.2.3 Shape Detection 44
5.2.4 Specular Highlight Filtering 44
5.3 Summary 46

6 Machine Learning **47**
6.1 Design and Implementation 47
6.1.1 Cascade Classifier Training 49
6.1.1.1 Exporting Positive Samples from TagAndTrack 50
6.1.1.2 Exporting Negative Samples 51
6.1.1.3 Exporting Metadata for the Samples 52
6.1.1.4 Using opencv_traincascade to Train a Classifier 53
6.1.1.5 Building an OpenCV based Detector Tool 54
6.1.1.5.1 Use a Separate Thread for Decoding 56
6.1.1.5.2 Introduce Multiple Threads 57
6.1.2 Histogram of Oriented Gradients Detector 60
6.1.2.1 Adding HOG-training to TagAndTrack 60
6.1.2.2 Visualizing the training result for a HOG detector 61
6.1.2.3 Implementing a separate HOG-Trainer 63
6.1.2.4 Exporting data from TagAndTrack to HOGTrainer 65
6.1.3 Index of Global Image Features 66
6.1.3.1 Global Image Feature Indexer 67
6.1.3.2 Global Image Feature Classifier 69
6.2 Evaluation and Discussion 77
6.2.1 Cascade Classifier Training 78
6.2.1.1 Experiment 1 78
6.2.1.2 Experiment 2 79
6.2.1.3 Experiment 3 80
6.2.2 Histogram of Oriented Gradients Detector 80
6.2.3 Index of Global Image Features 82
6.2.3.1 Benchmarking single Image Features 82
6.2.3.2 Finding an optimal Image Feature Subset 82
6.2.3.3 Classifier Performance Evaluation 83
6.2.3.4 Scalability Evaluation 85
6.3 Summary 86

7 Conclusion **87**
7.1 Summary 87
7.2 Main Contributions 88
7.3 Future Work 88

A Source Code **91**

Glossary **93**

Acronyms **101**

List of Figures

1.1 An overview of the terms used to describe the digestive system. 2
1.2 Equipment currently used for screening. 2
1.3 Images of a capsule with a single camera. 3
1.4 Images taken with a camera capsule. 3

3.1 The dataset creation process is a prerequisite for training an algorithm and eventually building a fully automated detection tool. 25
3.2 User Interface of the prototype written in Python using Qt and OpenCV 26
3.3 The user interface of the HTML5 based video tagging tool. 28

4.1 The user interface of the tracking software implemented in C++ using Qt and OpenCV. . 34
4.2 Time comparison of processing 100 video frames, using manual or automatic tracking. . 35

5.1 Original video frame and masked video frame after border detection. 38
5.2 A video frame in the original form, and with two different edge detection thresholds. . . 39
5.3 Shape detection in the learning and detection phase. 41
5.4 Specular Highlight Reduction by Gaussian Filtering. 42
5.5 Specular Highlight Reduction by using Gradients and randomized Kernels. 43
5.6 Shape detection of stars on the american flag. 45
5.7 Haar-based polyp detection on the same frame, with and without Specular Highlight Filtering. 45

6.1 An inconclusive list of irregularities that can be diagnosed using colonoscopy. 48
6.2 Uncorrected rotation of Region of Interest (ROI). 51
6.3 Calculated geometries for roatating and exporting positive samples. 52
6.4 Processing Times for Multi Threaded Cascade Classifier Detection, processing 2,025 frames at a resolution of 768x576. 59
6.5 Visualizations of HOG detectors for faces and for polyps with a window size of 80x80. . 62
6.6 Visualizations of HOG detectors for faces and for polyps with a window size of 30x30. . 62
6.7 The structure and basic elements of a lucene index. 67
6.8 The overall architecture of our global image features based approach, consisting of Indexer and Classifier as separate tools. 67
6.9 Console output of the classifier using the features JCD and Tamura. 69
6.10 HTML output of the classifier using the features JCD and Tamura. 70
6.11 TagAndTrack showing the results from the global image features classifier. 72
6.12 Visualizations of a HOG detector that was trained with rotations of multiple pictures of multiple polyps. 81

List of Tables

2.1 Performance comparison of polyp detection approaches discussed in this chapter. 22

6.1 Processing Times for Multi Threaded Cascade Classifier Detection, processing 2,025 frames at a resolution of 768x576. 59
6.2 Performance of a simple HOG detector with different window sizes. 63
6.3 Training Time for a Haar-based Cascading Classifier 78
6.4 Performance of a classifier using the same dataset for training and detecting. 79
6.5 Performance of a classifier trained with a subset (50 positive frames) of the testing dataset. 79
6.6 Detector performance of HOG-based classifiers trained in TagAndTrack. 81
6.7 Time consumption for training classifiers with HOGTrainer. 82
6.8 Leave-one-out cross-testing combined for all supported image features. 83
6.9 Top combinations of 2 image features for video wp_68, sorted by F-score. 84
6.10 Top combinations of 2 image features for video wp_61, sorted by F-score. 84
6.11 Performance evaluation by leave-one-out cross-validation for all available videos, using JCD and Tamura features. 85

Chapter 1

Introduction

1.1 Background

Colorectal cancer is the third most common type of cancer diagnosed for men and the second most diagnosed type for women according to the International Agency for Research on Cancer [1]. With a mortality rate of 10 for men and 6.9 for women per $100,000$, colorectal cancer is also among the three most lethal types of cancer. Typically, more than 65% of the new cases occur in countries with a high development standard, especially central Europe, despite available health care and high nutrition standard. An increase of colorectal cancer incidence and mortality can be observed in many countries transitioning towards higher levels of human development, according to the previously cited report. The reasons for this are beyond the scope of this thesis, but considering that the numbers provided are age-standardized, potential reasons might be higher intake of fat or red meat, obesity, diabetes and lack of physical exercise.

Colorectal polyps are a common precursor to colon cancer, and if found and removed in early stages, colorectal cancer could often be prevented. According to Stryker et al. [2], if a polyp is not removed, the cumulative risk of cancer being diagnosed at a polyp site is 2.5% at 5 years, 8% at 10 years and 24% at 20 years after the polyp was diagnosed. Just the ability of reliable polyp detection would therefore already be very valuable. Consequently, screening rates for polyps and colorectal cancer using colonoscopy have increased in the recent years [3]. The U.S. Preventive Services Task Force (USPSTF) recommends screening of people between 50 and 75 years every 10 years [4]. The requirements for a screening program in Norway are similar [5]. However, the cost of this screening process is very high. In the US, a single colonoscopy based screening costs about $1100, and with an annual cost of $10 billion, it is the most expensive cancer screening process currently in use [6]. The screening procedure is usually rather uncomfortable for the patient [7,8], and it also requires a lot of valuable time from medical personnel [9].

1.1.1 Screening Methods

The usual method of screening with an endoscope enables an expert to examine both ends of a patient's gastrointestinal tract . This in particular includes the foodpipe, stomach, duodenum, colon and terminal ileum (for details, see Figure 1.1). However, a wired endoscope does have several limitations. It is, for example, not possible to examine the small intestine with this method. It is also a rather expensive screening method, as it requires specialists to prepare and possibly sedate the patient and to conduct the actual examination. Further, the procedure is usually rather uncomfortable for the patient [7]. Figure 1.2(a) shows a typical colonoscope.

Another method, which is less invasive than conventional colonoscopy, is Computed Tomography Colonography. This method uses special x-ray equipment, as shown in Figure 1.2(b), to produce images of the colon. The data that is collected can then either be visualized as two-dimensional images, or as a three-dimensional model that can be traversed virtually. The main disadvantages of this method are: it is hard to identify polyps with a diameter smaller than one centimeter, it is not possible to collect any tissue samples during the procedure and there is only calculated imaging data and no optical one. While the direct discomfort using this method is smaller and, in many cases, it might be a more safe method,

the patient is exposed to a significant amount of radiation during the procedure [10]. The method also requires the availability of computed tomography equipment and specialists who can operate it [11].

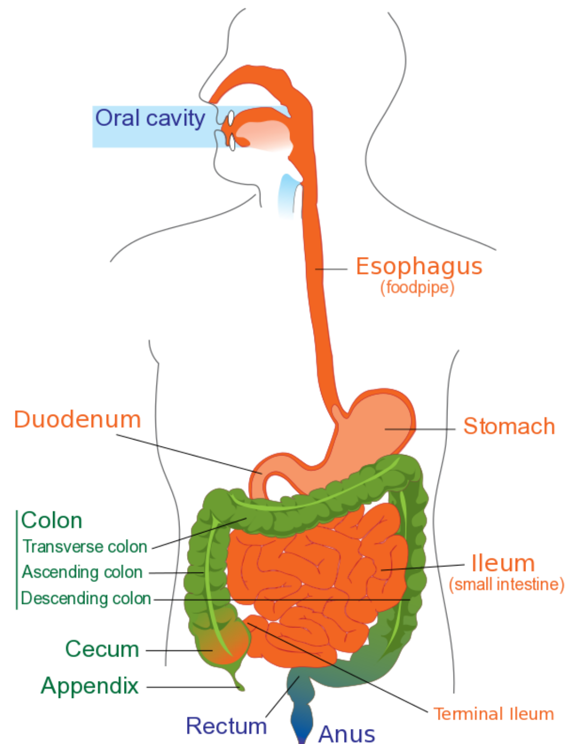


Figure 1.1: An overview of the terms used to describe the digestive system.[1]

(a) A typical video colonoscope. The controls are visible on the top left, and the flexible tube with camera are rolled up.[3]

(b) A computed tomography scanner.[4]

Figure 1.2: Equipment currently used for screening.

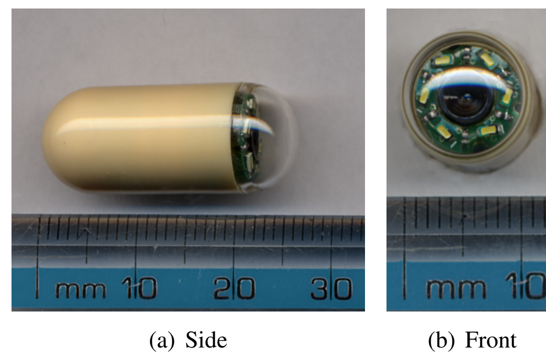
[1]The figure is derived from a diagram created by Mariana Ruiz and Joaquim Alves Gaspar. The original has been released into the public domain by the author. http://en.wikipedia.org/wiki/File:Digestive_system_diagram_edit.svg

[3]The figure is a modified version of an image created by Wikimedia User Gilo1969. CC BY-SA 3.0 / Attribution to Gilo1969 at English Wikipedia; http://commons.wikimedia.org/wiki/File:Colonoscope.jpg

[4]This image has been released into the public domain by the Wikimedia author Nithin Rao. http://commons.wikimedia.org/wiki/File:Ct-scan.jpg

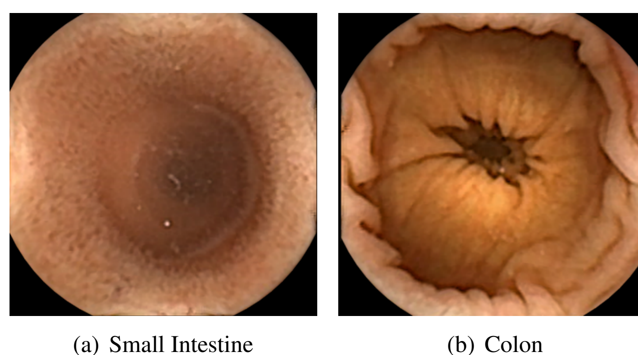
Wireless Capsule Endoscopy (WCE) has the potential to overcome several of the issues listed previously. It is conducted using a small swallowable capsule (Figure 1.3). The size of such a capsule is similar to the size of a large vitamine pill. Also, the capsule is equiped with one or two miniature cameras, a battery and a wireless transmitter. When the capsule is swallowed, it passes through the entire gastrointestinal tract , continuously generating images and transmitting those to a receiving device, which is usually worn by the patient on a belt. Thereby, wireless capsule endoscopy already overcomes two limitations, which apply to the conventional endoscopy. It allows to examine the small intestine [12], and the procedure can in theory be conducted independently by the patient. In future, a patient could then buy a kit containing a capsule and instructions in a pharmaceutical store. They could record the video data on a mobile device and upload it to a screening service.

The quality of camera capsules has steadily improved in the recent years and will continue to do so. The production cost would now allow for mass screening as well. However, the amount of data produced by capsule endoscopy and the processing requirements are already challenging. A capsule completes the journey through the whole digestive system in between 5 and 8 hours. Examining such a video sequence that is several hours long exeeds the time of a manual examination. It is a tiring and expensive piece of work and a questionable use of human resource [13]. To meet the needed efficiency in future health care systems, it is therefore necessary to develop tools to automate as much of the screening as possible.



(a) Side (b) Front

Figure 1.3: Images of a capsule with a single camera.[5]



(a) Small Intestine (b) Colon

Figure 1.4: Images taken with a camera capsule.[6]

A camera capsule usually takes thousands of images during this procedure. Two examples of images acquired with a camera capsule are presented in Figure 1.4. Currently, capsules with a variable frame rate between 2 and 6 frames per second (FPS) and a resolution of up to 512x512 pixels are available. Only 5

[5] Both images have been released by the author into the public domain;
https://commons.wikimedia.org/wiki/File:CapsuleEndoscope.jpg
https://commons.wikimedia.org/wiki/File:CapsuleEndoscopeEnd.jpg

[6] CC BY-SA 3.0 / Attribution to Dr.HH.Krause at English Wikipedia;
https://commons.wikimedia.org/wiki/File:Normales_Colon.PNG
https://commons.wikimedia.org/wiki/File:Dünndarm.PNG

hours of video at this resolution with the minimum frame rate of 2 FPS already equates to about 28 GB of uncompressed video data. According to Deligiannis et al. [14], contemporary video capsules employ conventional coding schemes operating in a low-complexity, intra-frame mode like Motion JPEG, or do not even use compression at all. Given the technological progress, both frame rate and resolution are very likely to increase in the near future. To allow mass screenings, it is therefore necessary to design a system which is reliable as well as scalable.

1.1.2 Computer Aided Diagnosis

Computer Aided Diagnosis (CAD) are computer-based systems in medicine, which help doctors with the interpretation of medical imaging data. Commonly, the term is used in radiology for systems that process data from X-ray or Magnetic Resonance Imaging (MRI). As the use of camera capsules spreads and becomes more and more common, it is also used for systems, which process capsule endoscopy videos and images. For CTC, there is already a range of systems available, developed by various universities and hospitals, which yield results close to a 100% detection rate for polyps greater than 1 cm in diameter [15–18]. Most of these systems are based on the reconstruction of the isotropic volume from the axial images, which were recorded by Computed Tomography (CT), and the detection of geometric features. For the final classification, artificial neural networks or Support Vector Machine (SVM)s are commonly used [19]. Artificial neural networks are statistical learning algorithms, which are inspired by natural neural networks, such as the human brain. They consist of a usually large number of simple processing units, which are simulating neurons, and weighted connections in between those units [20]. SVMs are machine learning methods, searching for a function that defines a hyperplane, separating the different classes of data points with a maximal margin. The concept of SVMs is discussed in more detail in [21]. Similar solutions for processing camera capsule videos are currently being researched [22, 23], but the data collected by capsule endoscopy is significantly different from the one collected by CTC. While CTC provides images of cross-sectional area, containing strong geometrical features and accuracy, capsule endoscopy provides two-dimensional photos, which show a projection of three-dimensional space. In addition, the images are usually subject to perspective distortion and variable focal sharpness due to the wide-angle fixed-focus lens. These conditions make it significantly more difficult to implement a CAD solution for capsule endoscopy videos.

Currently, there are several research prototypes, which show promising results. Most of these systems use feature extraction and SVMs for detecting polyps. In this thesis, we will show that despite promising results, the problem of automatic detection has not yet been solved. Also, several of these proposed solutions have undergone only minimal testing, most likely due to the lack of enough test and training data. Therefore, the first issue we address is the collecting of data.

1.1.3 Collecting Data for Training and Testing

Collecting data for training and testing is a difficult task due to numerous issues, including the sensitive context of patient information and the lack of appropriate tools. A test dataset does not only contain the video sequence created by the camera capsule, it must also contain information on what is visible in each video frame. The video must therefore be precisely annotated frame by frame, and regions must be selected where a lesion is visible. To make sure that the dataset is correct, this annotation needs to be done by a specialist. If done completely manually, this is a lot of tiresome work. Ideally, a specialist would only mark a lesion once and a tracking software would then generate the complete dataset. There are several annotation tools available, for example, Arthemis [24], VATIC [25] or Anvil [26], but they are either not easy to use or not easy to deploy. Therefore, related to the data collection task, we aim for a tool, which is exceptionally easy and time saving to use for the end user, and is easily deployable in a restrictive medical environment.

1.2 Problem Definition / Statement

A video sequence acquired by a camera capsule can last for several hours. A specialist must then evaluate the entire video manually to detect any lesions. As shown before, this procedure does not scale very well, because specialists and their time are a very limited and expensive resource. If we manage to further automate this process with the help of computers and limit the time a specialist has to spend, it might be possible to screen larger groups of the population and diagnose any conditions early on. Assuming, that an algorithm is reliable, it can potentially also reduce human-error imposed by watching and evaluating long video sequences of capsule endoscopies. This could eventually lead to earlier and more effective treatment of conditions and, therefore, decrease the loss of life, due to digestive diseases such as colon cancer.

Some research has been done to find specific methods or algorithms for detecting lesions on endoscopy images. This research is, however, still in its infancy. Also, there is no well adopted standard solution for the machine processing of capsule endoscopy videos. To allow for further experiments with different methods and algorithms, we aim for a prototype of an integrated solution that can be used for the following tasks:

- Collecting and Annotating of Data
  - Collecting video sequences from hospitals (collecting data).
  - Tagging of lesions / irregularities by a specialist (manual classification).
  - Tracking of tagged lesions in the same video sequence (semi-automated dataset creation).
- Machine Learning
  - Applying filters and preprocessing steps to the video sequence (video pre-processing).
  - Machine learning based on the previously created dataset (training a classifier).
  - Automated detection of lesions by previously trained classifier (automated classification).
  - Visualizing and evaluating the detected lesions in the video on screen (data visualization).

For testing our proposed solution, we also experiment with different types of classifiers. Most of the methods that have already been researched by others are either based on extraction of geometric features or on Local Binary Pattern (LBP)s forwarded to a SVM. We test different state-of-the-art image recognition and retrieval methods that are already available and used for other purposes, such as face recognition or Content Based Image Retrieval (BRIR).

1.3 Limitations

With this thesis, we research the process of building an integrated solution for automatic detection of lesions using capsule endoscopy videos. In this early research stage, it is mostly a process of gathering knowledge about the subject and implementing the required building blocks. Designing, prototyping and implementing solutions for collecting the required data is part of the thesis. The goal is to develop a prototype of the entire system to prove the basic concept of using machine learning for the detection of lesions and to allow further experiments with various detection methods. The deliverables do, however, not include a production ready system or classifier.

1.4 Research Method

For this thesis, we follow the *design paradigm* described by the *Association for Computing Machinery (ACM) Task Force on the Core of Computer Science* [27]. We design, implement and evaluate a prototype solution for the automatic processing of capsule endoscopy videos. To achieve the desired outcome, we first study literature that focuses on machine processing of endoscopy video data and on machine learning for object detection. We then prototype tools for acquiring video data and creating a complete dataset

based on the acquired data. These tools are evaluated by presenting and discussing the prototypes with physicians, who are supposed to use the developed software. In the next step, we implement software for training classifiers with the acquired dataset. We experiment with three different methods for object detection and implement the visualization of the resulting detector output. Eventually, we evaluate the three methods using well known machine learning metrics and measuring the time needed for training and running the respective classifiers.

1.5 Main Contributions

In this work, we present methods that address the problem of detecting polyps or other lesions in colonoscopy videos. Since the issue of collecting video sequences and creating complete datasets from these videos, as well as the thorough testing and comparing of different approaches, has not been sufficiently addressed previously, we have put our focus on implementing a solution to facilitate this.

We have therefore developed an HTML5-based web application, which can be used by physicians to upload videos and do an initial tagging of lesions. Further, we have developed a portable application, which can be used to create complete datasets from the initially tagged videos by using automated object-tracking and manual correction. We have built different filtering and detection mechanisms into the same tool. The developed software is a prototype of an integrated machine learning and detection pipeline and can be used for conducting further experiments with different classifiers and detection methods. We have also conducted experiments with different machine learning approaches. Specifically, we have evaluated if a Haar-feature based cascading classifier, a Histogram of Oriented Gradients (HOG)-classifier or a search based global-image-feature-classifier can be used for detecting lesions in colonoscopy videos. An important conclusion of these experiments is that state of the art Haar-based and HOG-based object-recognition approaches are not suitable for detecting lesions in the intestines. This is mostly because of the unknown orientation of the objects to detect, but also because of the huge variety of appearances such lesions can have.

However, our experiments with a search-based approach using global image features are very promising. Doing a leave-one-out cross-validation on the whole dataset of the ASU-Mayo Clinic polyp database[7], we achieve a weighted average precision of 93.9% and a weighted average recall of 98.5%. The measures precision, recall, accuracy and the process for leave-one-out cross-validation are explained in Section 6.2. For this search-based approach, we have implemented separate tools for indexing and classifying, which can also be used beyond the use case of detecting lesions in the intestines. We have implemented all our classifiers with the requirement to minimize the processing-time. All our classifiers achieve a processing-time which would allow for mass-screenings. However, the poor accuracy of our Haar-based and HOG-based methods for detecting lesions clearly disqualifies those methods for our use case. A further problem with particularly the *OpenCV-traincascade* based approach is the time required for training the classifier. Depending on the number of samples, this process can take several weeks and it does not allow for incremental training to refine a classifier over time.

A part of the software that we have developed has already been presented at the ACM Multimedia Systems 2015 conference in Portland [28]. We have received positive feedback, and we have also received requests for providing the code of our implementations under an open source license, to allow others to reuse it for further research.

All the software developed for this thesis is available under the terms of the GNU General Public License version 3. The source code can be obtained from Bitbucket, and all the repositories are listed in Appendix A.

[7] `http://www.polyp2015.com/wp/?page_id=164`

1.6 Outline

This thesis is subsequently structured as follows:

- In Chapter 2, we discuss literature that focuses on topics related to the automated detection of lesions in the colon. This particularly includes image recognition, existing screening methods, machine learning and dataset creation.
- Chapter 3 presents the details of designing, implementing and evaluating the tools, needed for collecting ground truth from physicians and hospitals.
- In Chapter 4, we present the experiments we have conducted and the tools we have implemented for creating complete datasets, by tracking previously tagged regions of interest. We further evaluate our proposed tool, by measuring the time that can potentially be saved using it.
- Chapter 5 presents the design, the implementation and the evaluation of several filtering mechanisms as preprocessing steps for our toolchain.
- Chapter 6 presents our research with regards to three different machine learning approaches for detecting lesions in colonoscopy videos. The chapter contains in-depth information on the design and implementation, the classifier training processes and the evaluation of all three approaches.
- In Chapter 7, we provide a short summary of our work, list our main contributions and give an outlook on future work.

Chapter 2

Related Work

With this thesis, we are researching potential solutions for automating the detection of lesions in capsule endoscopy videos. We are building the tools needed to test and evaluate different algorithms and approaches for such a classifier. This involves a variety of sub-tasks, such as collecting video sequences, preprocessing them, annotating them and training a classifier using machine learning or global features indexing. In this chapter, we present samples of the most interesting and relevant existing work that is related to these tasks. We collect and analyze information that we can use to create an integrated solution for detecting colon cancer or its precursors. The chapter is organized as follows. In the first section, we discuss work related to building a training dataset that can be used for machine learning. In the second section, we analyze an article that proposes an approach for reducing the amount of frames that need to be reviewed. In the subsequent section, we analyze different approaches for the automatic detection of colon cancer and publications related to this topic. Finally, we summarize the results and evaluate them against our own requirements.

2.1 Building a Training Dataset for Machine Learning

Typically, the first problem one has to solve for training a classifier using machine learning is building an appropriate training dataset. This dataset has to fullfill several requirements. It must be big and diverse enough to allow generalization and avoid overfitting, but it must also be specific enough to avoid learning random data due to the correllation between the samples being too small. In machine learning, overfitting is the phenomenon of a classifier encoding random noise, caused by training with too few training samples or too long training cycles. In such cases, it can happen that the learner encodes random features, which are common among the few available training samples, but do not abstract the relation between the samples [29]. This is why it is crucial to collect a big amount of high quality training data. We therefore also look into related work about collecting and segmenting of video sequences, and annotating and tracking of regions of interest.

2.1.1 Video Segmentation

Capsule endoscopy usually produces a video sequence with about 60,000 frames. Due to the time consumption for screening all these frames, it may be necessary to split such a video into smaller segments. The goal of video segmentation is, to divide a video into smaller meaningful sections. These sections can then be processed separately and different processing can be applied depending on the content of the section.

Li et al. [12] research the feasibility of applying motion analysis to video sequences as a preliminary step for automatic video analysis. A study of the two approaches - Adaptive rood pattern search (ARPS) and Bayesian multi-scale differential (BMSD) optical flow - is presented. These approaches are typically used for general motion analysis. To test the feasibility, the article focuses only on motion analysis and ignores other features such, as color and texture. A basic problem of motion analysis with capsule endoscopy videos is the variability of the content captured on frames. This variability is caused by the

chaotic movement of the capsule itself, as well as the peristalis of the intestine. Further, the mucosa is a non-rigid body which also undergoes complicated motion patterns.

ARPS is a simple fast block-matching algorithm: A video frame is partitioned into blocks of a predefined size, and then these blocks are compared to blocks in the subsequent frames, searching a specific range for the purpose of extracting a motion vector for each block. BMSD is a combination of differential constraints in local spatial regions. It is described in more detail by **Simoncelli et al. [30]**. Both algorithms eventually produce a motion vector field per frame and a motion intensity curve per video sequence. A peak value along the motion vector curve respresents a big motion activity. A global threshold can then be used to decide whether a value along the curve is a change of content. Comparing the two algorithms and their respective motion vector fields, it seems at first sight that BMSD would be more suitable for accurate description of motion patterns for capsule endoscopy. In contrast to ARPS, BMSD does not assume that each block in a frame undergoes only translation without scaling and rotating. However, an experiment, conducted in [12], evaluating 10 short video segments with 200 frames each and a resolution of $512x512$ pixels shows that the best average performance for recall using ARPS (71.89%) is higher than for BMSD (55.8%). Thus, while BMSD can describe the motion better, it causes inferior performance for capsule video endoscopy segmentation. The article concludes that motion analysis may not be sufficient to describe the scene change in a capsule endoscopy video and that additional features, such as color and texture, should be considered as well. This might allow increasing both recall and precision.

Primus et al. [31] propose a method for segmentation of endoscopic videos, based on spatial and temporal differences of motion in subsequent video frames. This paper is aming at providing a method for automated segmentation of endoscopy videos to allow for content based analysis and search of big video archives. The proposed segmentation approach is based on temporal and spatial differnces of motion patterns caused by camera movement or the movement of endoscopic instruments. Feature points are defined and traced in subsequent video frames. These results are then spatially grouped and a single motion value per group is calculated. If all the groups provide similar and high values, the movement is caused by the camera. If the values are similar and low, there is no relevant movement. If the values of the groups are significantly different, then this is most likely caused by movement of endoscopic instruments. This approach is specifically researched for, but not necessarily limited to the usage with endoscopy video sequences.

The approach is split into three steps. The first step is motion detection, using matching point features in consecutive frames. These point features must be spread eavenly to detect various kinds of motion. The second step area motion estimation is conducted by dividing the frame into smaller rectangular areas and combining the motion vectors within these areas into a single agregated motion vector. In a third step, all the aggregated motion vectors are compared. If no relevant motion happened, all the agregated vectors are similar to each other and are close to zero. If the camera was moved, all the motion vectors should be similar to each other and should be above a certain threshold. In case endoscopy equipment appears on the frame, the motion vectors of the different groups should be significantly different. The areas where the object appears, should then have a high value for the motion vector and all the others should have a value close to zero. This procedure is repeated with several different settings for window size and threshold in order to find the best fitting border frame.

This approach was evaluated with 20 distinct videos of laparoscopic and endoscopic thyroid surgeries, recorded in HD resolution. The transition from one segment to another cannot always be exactly defined, as the transition might span across 25 frames or more. Therefore, a detected segment transition within 25 frames of the actual transition is considered a true positive. An average presision and recall of 86% was reached with this approach on the testing data.

We consider this article interesting in particular for the idea of detecting motion in spatial sub-regions of different sizes and combining the respective results. However, motion analysis is not a specific focus point for this thesis. This research is related, but mostly as a preprocessing step for the automatic processing and detection of lesions.

Del Fabro and Böszörmenyi [32] provide a chronologically sorted overview of several previously published approaches for video segmentation. All the given approaches are classified into seven different groups based on their usage of low level features: visual-based, audio-based, text-based, audio-visual-based, visual-textual-based, audio-textual-based and hybrid. These class names describe the features which are used for the segmentation. Where visual features are the optical content of the video frames, textual data is meta data coming with the video, audio data is the audio track of the video. Hybrid approaches are essentially combinations of the other previously mentioned approaches. The challenge of video segmentation is described as segmenting a video into several separate semantically meaningful scenes. A scene thereby may consist of multiple video shot, and a shot is described as sequence of frames limited by two shot boundaries or end of the video sequence. A shot boundary is defined as the physical boundary, where camera changes happen. The initial step of most the scene segmentation approaches is to perform shot detection. Detecting scene boundaries based on keyframes of the shots, then allows to reduce the computational complexity for the scene detection. The fundamental task of the scene detection is then to identify semantically coherent shots, which are temporally close. There are three basic methods for scene segmentation:

- *Rule-Based Methods* were proposed which consider for example *film-editing rules* or *film grammar*. These *film grammar* describe rules for the production of movies. For example, that all cameras showing the same scene should be on the same side to preserve the background, or that the direction of motion should be preserved in consecutive shots of the same scene. These rules can be leveraged for the detection of scenes.

- *Graph-Based Methods* transform the scene detection into a graph partitioning problem, where shots are clustered based on similarity and sometimes also temporal closeness.

- *Stochastic-Based Methods* use stochastic models for the boundary detection. This turns boundary detection into an optimization problem for maximizing the probability of correct placement of the scene boundaries.

The paper also provides an overview of how the different approaches were evaluated, and what kind of video data has been used. It further contains a section on different strategies for the segmentation of different types of video genres. The video genres that have been used are Movies, TV series, News, Sport Videos, Documentaries, Home Videos, Cartoons and TV shows. For evaluating the quality of a method, the measures Precision & Recall and F-score are most commonly used. These measures are explained in Section 6.2. The paper concludes that it is not possible to provide an accurate quantitative comparison of the different algorithms presented in the paper, due to non-unified datasets and evaluation methods. It is further mentioned that all the different approaches have their respective strengths and weaknesses. The paper can therefore be used as a guide for finding the right approach to start with for a given problem.

We consider this survey only remotely relevant to our work. Partially this is the case, because we cannot rely on any audio or textual information, which already reduces the applicable approaches significantly. Also, at the current early stage of our research, we expect to receive already short video segments, which will not require further segmentation at this point. While the topic of segmenting videos is clearly related as a preprocessing step of an automatic endoscopy video processing pipeline, it is only of a rather remote interest for our current research.

2.1.2 Video Annotation

To build a ground truth dataset, it is necessary to collect video sequences with frames that show an object to be detected (positive frames), as well as frames where no such object is present (negative frames). It is then necessary to select the region on the positive frames where such an object is visible. Such selection builds a dataset containing both positive and negative samples. To create such a dataset, the appropriate tools ar required.

Liu et al. [24] describe a very advanced annotation tool called *Arthemis*. *Arthemis* is part of an integrated capturing and analysis system for colonoscopy, called *Endoscopic Multimedia Information System (EMIS)*. *EMIS* provides functionality for collecting and archiving endoscopy videos, uploading videos to a storage server, removing redundant video frames, separating or merging video sequences, segmentation based on audio features (speech recognition), post-processing and analysis of colonoscopic procedures. *Arthemis* was designed to facilitate the process of reviewing videos, locating and annotating important content, and exporting annotated content for research, teaching and training purposes. This tool supports annotation according to the Minimal Standard Terminology (MST). MST is an internaltionally accepted terminology standard for digestive endoscopy, proposed by the European Society of Gastrointestinal Endoscopy (ESGE) in collaboration with the American Society for Gastrointestinal Endoscopy (ASGE) and the World Organization of Digestive Endoscopy (OMED).

Arthemis supports annotation by ellipse selection and free-hand-drawing. It is written in Java and C, and uses a third-party MPEG encoding/decoding-library[1] for an extra fast playback mode, as Microsoft's native multimedia toolkit DirectShow was considered not to provide the needed performance. A special feature of *Arthemis* is that it provides the ability to view automatically detected segments of colonoscopy videos. Annotated figures are stored in a proprietary format based on Extensible Markup Language (XML). The software has been designed to be easy to use by physicians, allow fast playback, and be efficient, robust and extensible.

According to the article, the software starts to become a useful tool for endoscopic research and education. It is proposed to be used by Medical students, residents and fellows, for learning to recognize the common endoscopic anomalities and the therapeutic modalities, used by experienced endoscopists. The user-interface of *Arthemis* is considered to be intuitive and easy to use. The component-based design of the software is listed as a strength, since it allows multiple developers to develop new components at the same time, without worrying about losing the control of the code.

We consider this article very significant, as it describes an existing, complete implementation of an integrated solution for collecting, archiving, processing, annotating and visualizing colonoscopy videos. A potential weakness of the implementation is the use of proprietary components. The deployment requires both a server system and installation on the client side, making the entire solution hard to obtain and distribute. Also, the number of supported features seems rather large. Generally, a large number of features can be considered a strength, it usually also makes a tool less intuitive to use.

Riegler et al. [33] present a web based annotation tool, supporting several different kinds of annotations. The tool is not specifically designed for medical usage. The article researches the benefits of different video annotation features in communicating general concepts of a video game based on captured game sessions. Game play recordings were selected as a domain, because it specifically requires the software to cope with fast camera movements. The tool provides simple and easy to use controls for annotating with temporal and spatial information and functionality to enrich the content of the video with added information. Annotations can be done by free hand drawing on top of the video, either during playback or alternatively, when the video is paused. The annotations will be replayed during video playback; if an annotation was added when the video was paused, the playback pauses for replaying the annotation. An additional feature is the possibility to zoom into the video; annotations created in zoomed-in mode will automatically be replayed in zoomed-in mode. For simple temporal annotations the tool further provides *LikeLines* [34] - a bar below the video, displaying a one-dimensional heatmap. The heatmap displays which parts of the video received explicit "likes". The tool is written entirely in HTML5 and JavaScript, using the HTML5 video element for playback and HTML canvas for drawing.

The tool presented in this article is of particular interest for us, because it provides a straight forward approach for collecting video sequences and creating annotations, which we can potentially reuse for collecting colonoscopy videos from hospitals; we can provide similar functionality to receive basic annotations from physicians. As mentioned in the conclusion for this article, users, who tested the software, still considered it too complicated and considered training necessary for creating good annotations. It is

[1] `http://www.mainconcept.com`

also stated that the ability to slow down the playback when creating an annotation is important and that users want text based annotation in addition to the hand drawn one. The zooming function was not considered important, unless the video sequence to be annotated is of high resolution and high level of detail.

The use of an annotation tool for endoscopy videos is further researched by **Lux and Riegler [35]**. This demo paper focuses on common interaction methods for experts to annotate videos by recording speech and drawing onto the video. The paper aims at gathering information about the recorded videos in an easy and simple way, so that the annotation effort is minimally invasive for the daily routine of the experts. A tool for an Android tablet computer is presented, which uses the touch screen, motion sensor and speech recognition for user interaction. This tool is required to be easily integratable into existing business processes in medical information systems. Hence, complicated installation and hardware requirements were not acceptable. A low cost off-the-shelf tablet computer, however, is considered a good choice. The following features are integrated into the software: The video can be manually segmented into non-overlapping pieces, selected segments can be annotated using speech, text annotations can be added by using the integrated google speech-to-text web service, sketch-like drawings can be added on top of the video, and shaking the device is used for annotating important events in the video sequence. All the annotation information is kept and stored separately from the video file. Audio recording is stored in a compressed form, and the drawings are stored as path information. As a technical challenge, the paper mentions the drawings which have to be drawn on top of the video overlay. The replay is not accurate in terms of frames, but is considered good enough in terms of accuracy and excellent in terms of robustness and performance.

We conclude that it is crucial to integrate in a minimally invasive way with the environment of the experts, who we want to collect information for us. It seems very important to provide them with a solution which is very easy to use and, at the same time, very easy to deploy in a restricitve medical environment.

2.1.3 Object Tracking

Screening a complete endoscopy video and marking any lesions using rubber-band-selection on every single frame is a very tedious and time consuming task. Instead of doing this manually, it might be possible to use software for tracking of regions that were previously selected. Ideally, a lesion then only needs to be selected a single time.

Hare et al. [36] present a framework called *Struck*, which implements adaptive visual object tracking. The method that the paper describes in detail is based on structured output predition. It makes use of a kernelized structure output SVM, which is trained and updated during the tracking. It can therefore adapt to the tracked object, changing appearence or perspective over time. *Struck* uses a budgeting mechanism, that prevents from unbounded growth of support vectors. While the Haar-like feature representation has almost become a standard for tracking by detection, the paper presents a novel approach on integrating the adaptive learning directly with the tracking. Using this approach, there is more information available for the learning task than only a binary classification of the previously detected data. This avoids an artificial binarization step, which is common for other state-of-the-art trackers, and the relative relationships between different samples can be taken into account for learning and updating the tracker. The paper also describes the budgeting mechanism for the support vectors. Basically, whenever the budget is consumed, the support vector resulting in the smallest change to the weight vector is removed. According to the authors it is also a strength of the design, that it is easy to incoroprate new image features and kernels into the presented SVM learning framework.

For performance experiments, 6 different types of Haar-like features were used at 2 scales on a $4x4$ grid, resulting in a total of only 192 features. A radius of 30 pixels was used for searching and 60 pixels for updating the clasifier. The results of these experiments show that *Struck* outperforms state-of-the-art trackers in malmost every test sequence.

A very advanced approach for object tracking is presented by **Kalal et al. [37]**. This paper aims at enabling tracking of objects in video streams taken by hand-held cameras, where various objects move in and out of the scene and the camera itself is moving as well. It is also the goal to allow this type of tracking in realtime. For this purpose, the paper describes a tracking framework called *Tracking-Learning-Detection (TLD)*, which decomposes long-term tracking into the three separate components tracking, learning, and detecting. The tracking component estimates the motion of an object in consecutive frames. The detector performs a full scanning of the image and localizes appearances of previously learned objects. The learning component monitors the performance of the tracker and the detector and generates training samples to learn from previous errors. The learning mechanism uses the tracking output to verify the output of the detector and updates the detector accordingly. The learning method is using a pair of separate "experts". The *P-expert* estimates the object location in the current frame using a frame-to-frame trackier. It identifies false negatives and generates positive samples for subsequent updating of the classifier. The *N-expert* analyzes all the responses of the detector, matches those against the tracking data, reinitializes the tracker with the maximally confident patch and generates negative samples for updating the classifier. The *P-expert* thereby increases the the generality and the *N-expert* increases the discriminability of the classifier. This a semi-supervised learning paradigm and it is called *P-N learning*. The learning method is able to deal with arbitrarily complex video streams with frequent tracking failures and it does not degrade the detector in case there is no relevant information contained in subsequent video frames. The paper also provides a performance evaluation, where *TLD* is compared with competing tracking methods by calculating precision, recall and F-score. The evaluation was done with 10 different video sequences and various different objects were tracked. *TLD* scores best in 9 out of these 10 video sequences and therefore clearly shows superiour overal performance.

We consider this paper highly relevant because of the very good tracking performance that is achieved with this approach. The ability to do tracking in real-time and to continuously adjust and improve the detector and thereby reliably track a non-rigid object is very interesting. Another interesting feature of this method is the built-in functionality to detect when an object is no longer present in a new frame.

2.2 Frame Reduction

As mentioned before, one of the main goals for researching automatic detection of lesions in capsule endoscopy videos is to reduce the amount of time a human specialist has to spend evaluating the video.

Tsevas et al. [38] present a different approach to this problem. The paper proposes an unsupervised method to reduce the number of frames that must be evaluated manually. The approach is based on clustering and non-negative matrix factorization (NMF). A video is thereby summarized in a smaller representative subset of video frames. Those video frames then still need to be evaluated by a human. The approach has shown promising results, however, it was tested on a controlled dataset consisting of a single video sequence only. The method consists of three steps. First, dimensionality reduction of the initial dataset is applied using a square non-negative similarity matrix. This matrix is then used as input for the next step, which creates a predefined number of clusters of video frames by applying fuzzy c-means (FCM) clustering. Finally, the two NMF algorithms symmetric NMF and non-negative lagrangian relaxation are applied on the clusters to extract representative frames. The number of clusters must be chosen manually. According to the article, the number of clusters did not drastically affect the summarization result. A larger number of clusters results in more smooth results for the summarization, but also in an increase of computation time needed.

The article concludes that a significant reduction of the total number of frames was achieved, which should lead to a smaller inspection time and, therefore, an increase in the productivity of the experts. The summary that was created from a single video sequence contained representative frames from every frame neighbourhood present in the video with close similarity of the neighbouring frames.

We conclude that this approach might be valid for reducing the number of frames that need to be further inspected by a detection tool or by a specialist. It is a viable option as a preprocessing step.

However, the severity of dropping any significant frames of a video sequence must be considered. Especially, when taking into account that there has only been very little testing of the method and the only results presented originate from a single video sequence of a single patient.

2.3 Methods for Automatic Detection of Colon Cancer

We identified two different approaches for the automatic detection of colon cancer. These approaches are *Geometrical Analysis* and *Machine learning*. They can be used for imaging data that was recorded with a conventional colonoscope or with a camera capsule. It is also possible to use these methods with data from a *Virtual Colonoscopy*. However, such data is significantly different from camera recorded data, as it is acquired using a CT scanner. We will therefore discuss this approach in a separate section.

2.3.1 Geometrical Analysis

Analysis of geometrical features can be used as a method for detecting polyps in capsule endoscopy video frames. The approach is usually based on a set of rules derived from empiric data, such as size, curvature or texture of a polyp.

Mamonov et al. [39] present an efficient algorithm for a binary classifier to detect colon polyps. The method is called *binary classification with pre-selection*, and it aims at reducing the amount of frames that need to be manually inspected. The algorithm operates on separate input frames and labels each frame as either containing a polyp or not. It is based on the assumption that polyps can be generalized as protrusions that are mostly round in shape and was tested on a dataset created from frames of video sequences of five different patients.

The algorithm converts video frames to grayscale and applies pre-processing to reduce *vignetting*[2]; the image is circularly extrapolated to remove any border regions caused by the lense of the camera. Texture information is used as a source of information for mid-pass filtering. Frames with too little texture are discarded because polyps ususally have a highly textured surface. If there is very little texture this means that there either is no polyp, or the whole picture is blury and a polyp could therefore not be detected anyway. Frames with too much texture are discarded as well, because strong texture is an indication of the frame containing either trash or bubbles, and a polyp can therefore not be detected. This straight-forward discarding of pictures is a potential weakness of this approach. However, even at low frame rates it is statistically likely that a polyp is visible in multiple video frames. Once a frame has been pre-processed and if it was not discarded by the texture analysis, mid-pass filtering is applied to isolate protrusions within certain size limits. This filters out folds of the mucosa, as well as very small artifacts. Then, the binary image is segmented and decomposed into separate components that represent features in the frame. The tensor of inertia is calculated for every feature to remove any features that are stretched beyond a certain treshold, because protrusions which have a very stretched oval shape usually belong to a fold of the mucosa. Finally, a best fit ball radius per feature is used as an input parameter to the binary classifier. If the decision parameter is above a certain threshold, then the frame is classified as containing a polyp. In addition to the video frame, the algorithm also accepts certain numerical parameters as input. These numerical parameters have to be chosen in advance. The manuscript specifically discusses the robustnes of the algorithm with resprect to these input parameters.

When tested with video data of five different patients, the algorithm reached a sensitivity of 81.25% per polyp and at a specificity level of 90%. However, the sensitivity of the algorithm with regards to single input frames is significantly lower and only reaches 47%. The length of an input sequence varied between 2 and 32 frames and a total of 16 sequences were tested. The false positive rate on the total of $18,738$ frames not containing a polyp was 9.8%. Assuming that it is usual to have multiple frames available for a single polyp, these numbers seem quite promising. Basically, it means that time a specialist spends on evaluationg video data could be reduced by about 90%.

[2] Reduction of brightness or saturation at the periphery of the image, compared to the center

A similar approach is presented by **Hwang et al. [23]**. This approach also focuses on shape, in particular on ellipses, which is a common shape for a polyp. Using this method, a frame is first segmented into regions by a wathershed-based image segmentation algorithm. This algorithm is based on the observation that polyps are spherical or hemispherical geometric elevations on the surrounding mucosa. Ellipses are then fitted into the segments by constructing a binary edge map for each segmented region and using a least square fitting method. A threshold-function is used for the creation of the edge map. Regions with too little edge information in their respective edge map are discarded. These ellipses are then further evaluated for matching of curve direction, curvature, edge distance and intensity. The direction of the parabola from any part of the ellipse must be matching the direction of the corresponding part of edges for the ellipse to be considered a polyp. This assures that the detected edges build an ellipse-like shape instead of, for example, a parallel one. The curvature of the ellipse is split into six parts. At least two adjacent parts must have a strong edge pattern, otherwise, the ellipse is discarded. Lumen areas are filtered out by applying a threshold on the intensity of the ellipse.

The speciality of the approach presented in this paper is that after the first frame a potential polyp was detected, subsequent frames are also searched for the same characteristics using a mutual-information-based image registration technique. This allows to apply a threshold in number of frames for the detection to reduce the number of false positives.

The paper also provides the results of an experiment, where a video sequence with a frame rate of 15 FPS was processed. Out of 27 available ”polyp shots” 26 were detected correctly with a total of 5 false-positives. Identically to [39], the authors assume that there are multiple frames available of the same polyp and a certain number of false-negatives is accepted in order to balance the number of false negatives. Obviously, the correctness of this assumption depends heavily on the frame rate of the camera that is used for recording the video.

A different method that is not directly connected to the automated detection of colon cancer, but might still be valuable is presented by **Hong et al. [40]**. The article describes a method for fully automated 3D reconstruction of colon segments from individual colonoscopy images. This method does not require a prior CT scan or any other positioning information. The paper mentions several possible applications for the method. Among those applications are post-procedure quality control to determine the percentage of the inspected areas of the colon during withdrawal of the endoscope, real-time quality assistance by creating a 3D map of the uninspected areas, and colonoscopy education. The method takes advantage of the tubular nature of the colon. The reconstruction is conducted in the following steps. First, the colon fold contours are derived. Then, reverse projection is performed, to place fold contours in 3D space and correct distortion caused by the camera lense. In the next step, the distance of the folds from the camera is estimated, based on the intensity of the contours. Finally, the folds are used as a frame for the virtual colon structure, and the surface and wall between neighboring folds are augmented.

The paper also proposes a method for brightness intensity calibration. The calibration is used to compensate for uneven intensity levels caused by characteristics of the camera and its light source. The calibration should be done before pixel intensity is used to estimate distance.

A number of difficulties for creating a 3D reconstruction are mentioned. Only partial edges of colon folds can be visible, because parts of fold surface can be hidden behind another fold. Strong light reflection, blood vessels, stool or other obstacles can appear on images and cause strong edges. The colon shape is not always circular; in certain sections it can be triangular as well. Further, there is no ground truth information available, except for synthetic colon models, because a colon is not a rigid object. Even when comparing CT acquired data with endoscopy images of the same patient's colon, the results would be different due to body movement or variable inflation of the colon.

The method, presented in the article, was tested with images of a synthetic colon model, and the tests have revealed encouraging results with only small average reconstruction errors. The method has also been tested with 38 images from different segments of real colon from 6 colonoscopy procedures, to show that it also works with images of a real colon. As mentioned before, it is, however, not possible to compare the results from the reconstruction of a real colon to any ground truth information, because it is impossible to obtain ground truth information.

While not explicitly mentioned in this paper, a good reconstruction of a colon should also allow the automatic detection of protrusions, as the underlying geometric information is available in the model. However, it is unclear if a 3D reconstruction, as described in this paper, is also possible with imaging data recorded by a camera capsule. When conducting a conventional colonoscopy, the colon is usually inflated with CO_2, but this is not the case for a capsule endoscopy. The images recorded by a camera capsule will therefore not provide a nice tunnel view of the colon and, thereby, also contain less intensity information that can be used to reconstruct the depth.

2.3.2 Machine Learning

Machine learning has many different use cases. A common one is detecting objects on images. Nowadays, there is also research for using machine learning to detect lesions, like tumors, ulcer, polyps or bleeding in capsule endoscopy video sequences. In particular, the use of SVMs is common [11, 13, 41–43].

Li and Meng [43] describe a computerized tumor detection system for capsule endoscopy. The system is based on LIBSVM [44] with radial basis function kernel. LIBSVM is a an open source machine learning library, developed by the National Taiwan University and written in C++. Rather than focusing on the shape, this article focuses on textural patterns for detecting tumors. The article describes two feature selection approaches: sequential forward floating selection (SFFS) and recursive feature elimination (RFE) for refining features and thereby increasing the accuracy of the detection. The proposed system achieves an accuracy of 92.4% in recognizing tumors on test data. The features proposed for detection in the article are robust to illumination change. They are extracted by a combination of LBP and wavelet transform for multiresolution analysis. LBP is a type of feature vector, used for machine learning. It is based on the concept of splitting the image into cells and comparing each pixel within a cell to it's neighbors in a predefined order. If the neighbor pixel's value is bigger than the one of the center pixel, a "0" is written, otherwise a "1", generating an 8-bit binary value for every pixel. The histogram of these generated values over the cell is then used as the feature vector. This is just the most basic concept, and there exist various adaptations of the algorithm.

Wavelet transform is a concept, similar to fourier transform, but instead of decomposing a signal into sine and cosine waves, wavelet transform decomposes a signal into multiple non-overlapping wave-like oscillations, which begin and end at zero, and are therefore limited in time. Applied to image processing, this can be used to filter out noise, perform compression or to create a feature vector for machine learning.

Capsule endoscopy images are usually color images. The feature extraction is therefore applied in the three separate color spaces RGB, HSI and Lab space, and each channel of the respective color space is handled separately. The concatenation of the features from these three color spaces is used as a candidate to represent an image. The extracted features are used as an input for feature selection with a SVM. Comparing the two feature selection approaches SFFS and RFE; both approaches have their respective advantages and problems. Both approaches led to an improvement in detection accuracy. SVM-RFE shows better overall detection performance than SFM-SFFS, potentially due to the built-in regularization mechanism to avoid overfitting, which might be relevant in particular due to the redundancy of data, caused by analysing multiple color spaces of the same image. An advantage of SVM-SFFS is that it uses the weight of support vectors to determine the significance of a candidate feature. It can thereby implicitly compensate for redundancy within different features and choose a nearly optimal set of features. For SVM-RFE, on the other hand, the performance greatly depends on the amount of features, which must be chosen manually. The best detection rate with RFE was achieved by using a total of 90 features. A disadvantage of SFFS is that it is very time consuming.

The proposed features are also compared to several other common feature selection methods, and they do show superior performance. However, common problems also with this approach are false positives or obstruction of view, due to the precense of bubbles, caused by turbid fluids, foods, and faecal materials in the colon.

There are several other sources, which discuss the use of LBP for classification. **Häfner et al. [45]** present a method to describe local texture properties within color images, based on LBP. This article does not focus on detecting lesions, but rather classifying images of previously detected lesions, based on the pattern on the surface. Most LBP based methods either process grayscale images, or process every color channel separately. The method described in this article builds a color vector field from an image. A new operator called Local Color Vector Patterns operator (LCVP) is used to build a 1D-histogram based on the color vector field. This LCVP operator is the main contribution of the article. It was designed to achieve better performance while maintaining good classification results. The 1D-histogram is used for classification using the k-nearest-neighbors algorithm. This is a rather simple algorithm, where k denotes the amount of nearest neighbors to the data point to be classified. The data point is then classified by a majority vote of its neighbors, where each neighbor votes for the class itself belongs to. This algorithm might not be the most powerful classifier, but it is suitable for presenting the discriminative power of the features used.

LCVP compares pixel block intensity averages instead of pixel values as it is done in regular LBP. For the examples provided with the article, a block size of $3x3$ pixels is used. Changing the block size allows to apply LCVP at different scales. Experiments using this method were conducted with a total of 716 color images from 40 different patients. The resolution of an image was $256x256$ pixels. The results are compared to other LBP based methods. While a different operator, described by **Häfner et al [46]**, generally shows better classification results, LCVP outperforms this one and several other LBP based methods, when it comes to computational demand. LCVP is up to 7.5 times faster than the methods it was compared with in the article. The imaging data used for testing the LCVP was recorded using a 150-fold magnifying endoscope. The images therefore contain a very high level of detail. Further advantages of multi-scale LCVP are the small amount of parameters the operator requires and that the operator is generic and may also be applicable to different scenarios than the one presented in the article.

While this operator seems to show very good results, it might not be suitable for the needs of this thesis, as it requires a fairly high amount of detail. The video material that we have available was mostly recorded with regular non-magnifying endoscopy equipment.

Another approach for detecting polyps on capsule endoscopy video frames using a SVM is proposed by **Zhou et al. [13]**. The described method is based on the observation that a polyp usually reflects more light than the regular tissue of the mucosa due to its protruding shape. A video frame is being processed as follows. First, the average value for each RGB channel is calculated. Every channel value for a pixel below a certain threshold is then lifted to the average value for the respective channel. After having removed any dark regions in the frame, the variance of RGB channels is calculated. The variance is then applied to the image as a threshold, and pixels above that threshold are changed to white color. A statistical region merging (SRM) algorithm is applied to merge separate pixels of similar color into connected areas. Based on the assumption that a polyp reflects significantly more light than the remaining tissue, it should then be clearly visible as a white area in the image. An iterative method, radially searching the surroundings of any white pixel, is then used to calculate the radius of the white marked areas. Finally, a SVM with 2 linear classifiers, accepting the variance of the frame and the radius of a region, is used to decide on the region being a polyp or not. The paper further describes a mechanism for calculating the real size of a polyp based on the focal distance and the radius of a polyp.

For polyp detection, the method achieved an overall accuracy of 90.77% on the testing data. The dataset contained a total of 359 video capsule endoscopy images from different video clips, where 294 of the images were used for training and 65 for evaluation. Using this testing data the sensitivity was 75% and the specificity 95.92%. The error ratio of the radius calculation is expected to be about 9.77%, however this cannot be verified as the real sizes for the polyps are unknown.

Alexandre et al. [11] present a method for detecting polyps in endoscopic video using SVMs. The method described in this article is intended to indicate which parts of a video must be evaluated more closely by a physician. Experiments with the described method yielded a result of 93.16 +/- 0.09% of the area under the Receiver Operating Characteristic (ROC) curve.

The ROC curve is a graphical representation that describes the performance of a binary classifier. The curve plots the true positive rate (sensitivity or recall) on the ordinate against the false positive rate (1 - specificity) on the abscissa. The best possible classification would be characterized as a point in the top left corner of the graph with no false negatives and no false positives. Completely random guessing would lead to a point on the diagnoal. Therefore, the bigger the area under the curve, the better is the quality of the classifier.

The method described in this article is based on a rather simple feature extraction method. After preprocessing an image is subdivided into pieces of $40x40$ pixels. The only input data for the SVM is the RGB components and the coordinates for each pixel in the sub-image. This is a total of 5 features per pixel or $8,000$ features per sub-image. Considering the simplicity of the feature extraction, the performance of this approach is certainly remarkable. An interesting part of this article is the preprocessing of the data for the training process. The dataset used for training contains the video frames as well as a binary mask for each frame where polyps are marked black and the rest of the image is plain white. A video frame is first cropped to remove any black borders and get a proper rectangular shape. The image is then subdivided into $40x40$ pixel sub-images, exactly as it is done for the detection, and each of these sub-images is processed separately. The binary mask for the respective video frame is then projected onto the sub-image and the black pixels within the sub-image are counted. Based on a threshold, it is decided if the sub-image shows a polyp or not. The value of this treshold can vary between 1 and 1600. For further experiments, a value of 1300 was selected. The influence of the value is studied and the results are presented in the paper.

The training dataset contained 35 video frames, which were obtained by a video endoscope system. After preprocessing, each frame was subdivided into 132 sub-images, leading to a total of $4,620$ images of $40x40$ pixels. As a classifier, LIBSVM was used with a radial basis function kernel. Two-fold cross-validation on the $4,620$ sub-images was used to evaluate the error.

While this experiment shows good performance results, it is unclear how well the system would work on a completely different dataset. Only a total of 35 different video frames were used, and it is unclear how many different polyps are visible on these frames, or how many different patients were involved. Further, it seems problematic that the 2-fold cross-validation was used on the sub-images instead of the full video frames. It is therefore very likely that neighboring sub-images of the very same frame ended up in the training dataset and in the testing dataset during the same testing round. A sub-image does usually not contain a complete polyp, and the contour information does therefore not play a significant role. The SVM therefore mostly detects the actual color patterns, which are very similar on neighboring sub-images. Considering these factors, we conclude that the resulting value must be interpreted with extreme caution.

Another promising approach for object detection is presented by **Dalal and Triggs [47]**. The specific detector presented in this paper uses a linear SVM with grids of HOG descriptors to detect humans on images. The implementation is benchmarked with a pedestrian database provided by Massachusetts Institute of Technology (MIT), as well as with a more challenging dataset of $1,800$ individual images containing a wide range of poses and backgrounds. Detecting humans is considered especially challenging, because there are numerous of appearances and poses a human can adopt. The presented HOG-based descriptors provide significantly better performance than other existing feature sets, such as wavelets. On the database provided by MIT, the detector yields essentially perfect results. The presented method is based on the evaluation of normalized local histograms of image gradient orientations. These gradients are located in a dense grid, and do partially overlap. For this purpose, the detector window is tiled and the cells, where HOG feature vectors are extracted overlap. The combined vectors are then forwarded to a linear SVM for classification. The detection window is run over the image in all possible positions with various scales, to detect objects of variable size. A strength of HOG representation is the ability to capture edge or gradient structure that is a characteristic feature of shape rather than just color patterns. For color images, a separate gradient is calculated for each color channel. Out of these, the one with the largest norm is selected. A gradient is calculated using a predefined kernel size for every single pixel in a cell. The pixel then provides a weighted "vote" for the orientation. The votes are then accumulated

into orientation bins for the local cell. Local contrast normalization is essential for good performance of the algorithm. Smoothing the image before calculating gradients significantly reduces the quality of detection. This emphasizes that a good amount of the image information is extracted from abrupt edges at fine scale. Blurring the image to reduce sensitivity is therefore considered a mistake.

In this article, HOG is presented as an exceptionally strong solution for object detection. We conclude that it is an approach that is worth trying with colon polyps as well. However, considering that HOG is in particular suitable for detection based on strong edges and less on more subtile color differences, it is questionable if the approach is suitable for our purpose. The main problem that is not adressed in this article is rotation of objects. Humans usually stand on their feet, but polyps can be rotated in any possible direction. Another problem might be the quality of endoscopy images. Not all pictures from endoscopy videos are always totally crisp and in focus, due to the very short focal distance. Also, there is plenty of fluid, which might stick to the lense, obstruct the view and lead to less clear edges.

Giritharan et al. [42] present an interesting method for the detection of bleeding in capsule endoscopy videos. The method uses an ”SVM Ensemble”, an approach that makes a classifying decision based on multiple separate classifiers. In the described experiment, the video is preprocessed by removing the bounding black region, applying an averaging filter to remove random noise and dropping poorly illuminated or over-exposed frames. After that, three features are extracted. The first one is the histogram of hue, saturation and value. The second feature is the dominant color, and the third one is the co-occurrence matrix of the dominant colors with a window of $5x5$ pixels. Each of these features is then handled by a separate SVM, and the final classification outcome is computed from the results of the SVMs using a decision function.

This color-based approach seems to be a very good fit for the detection of blood. However, we expect that the sole relying on colors for classification is also preventing this approach from being applicable in a more generic way. Nevertheless, we consider this approach very interesting, in particular for using multiple separate, rather simple SVM classifiers simultaneously and applying a decision function on their result to do the final classification.

In the subsequent chapters, we will repeatedly come across Haar-like features. Those are digital image features that were first described by **Viola and Jones [48]** as a machine learning approach for visual object detection, capable of processing images extremely rapidly and achieving high detection rates. The name is derived from ”Haar Basis function”, which was proposed by Alfréd Haar. The article descries the implementation of a highly performant face recognition system, including an integral image representation and a learning algorithm, capable of selecting a small number of critical visual features and yielding a very efficient classifier. The integral image representation, can be calculated with a few instructions per pixel. Once it is available, Haar-like features can be computed in constant time for any location or scale. The classifier is built by selecting a small number of important features through a modified version of AdaBoost [49].

Further, the concept of a ”cascade” is introduced as an object specific focus-of-attention mechanism, which allows to quickly determine where in an image an object might occur and reserve more complex processing for these regions only. According to the article, the proposed algorithm is roughly 15 times faster than any previously introduced approach.

We consider this article very interesting because of the performance gain that was achieved by applying variable processing effort to selected subregions through a cascade. The expected efficiency makes the approach in particular interesting for a screening usecase that is meant to scale to a big number of video sequences.

2.3.3 Index of Global Image Features

Most the related work that we have evaluated is focusing on detecting the exact position and shape or color of a lesion. However, global image features are one more possibility to approach the problem in a potentially more efficient way. Global image features can be used as descriptors for images. This allows to compare the descriptors and to apply a distance measures.

Lux and Chatzichristofis [50] present a library called *Lucene Image Retrieval (LIRe)*, which is written in Java and can be used for content based image retrieval. The library contains implementations for the indexing and storing of state of the art global image features. The implementation is based on the text search engine Lucene. The library is intendend for developers and researches, to integrate content based image retrieval into existing applications. *LIRe* can be used to extract image features and store them in an Lucene index for later search and retrieval. This implies finding and retrieving images independent from metadata, only relying on the image features that can be extracted from the image itself. Implementations for extracting multiple image features such as color histograms, edge histogram, tamura texture features and color and edge directivity descriptor are provided with the library.

The paper further describes an additional *LIRe Demo* package, which provides a graphical user interface that can be used with *LIRe*. The implementation of this demo is using multiple threads to index photos from Flickr. Both the library as well as the demo package are licensed under a GNU GPL license. *LIRe* uses linear search for searching the generated indices but takes advantage of the fast disk access layer provided by Lucene and allows indices bigger than the available main memory. *LIRe* also provides an Application Programming Interface (API), which is extensible through interfaces, so further image features can be added by implementing an interface and implementing an algorithm to build a textual representation of a given image. The paper also lists indexing performance measurements for the different image features. According to the provided list, the two most computational expensive image features are *Auto color correlation* and *Tamura*.

We consider this paper very interesting and relevant for our research. Instead of trying to locate a specific item on an image, this approach uses global image features to describe the image in a more generic way. If we could use this approach to build clusters of positive and negative images respectively, this would allow us to circumvent the problem of precisely locating lesions in the image. This might also allow us to reduce the amount of preprocessing steps and therefore build a much more simple and efficient pipeline. Further we expect the indexing of global image features to be significantly less expensive than conducting machine learning for the same purpose.

2.3.4 Virtual Colonoscopy

Virtual Colonoscopy or CTC is not by itself a method for automatic polyp detection. However, CTC is another non-invasive method that can be used for imaging the colon. This method uses helical CT instead of a colonoscope or a camera capsule. For the exampination, the colon is inflated using carbon dioxide or regular air. The time for the entire examination is usually around 10 minutes.

Heiken et al. [10] present a report on the status of Virtual Colonoscopy for colorectal cancer screening. The report describes the process and the methods used for CTC. It compares CTC with conventional colonoscopy and lists advantages and disadvantages. One of the main advantages of CTC is that it allows to eliminate blind spots behind haustral folds or beyond bends of the colon by providing an endoluminal view in forward and backward direction. A common problem with CTC is that retained fluids or incomplete distension of colonic segments can obscure lesions and thereby cause false negatives. Also, false positives can occur, for example, due to retained stool or folds in the colon. Further, CTC does not allow immediate biopsy or removal of polyps that are identified. According to the report, studies performed with CTC have shown sensitivities of 82% to 100% and specificities of 90% to 98% for polyps above 10mm. The data generated by CTC is viewed interactively and it is usually available in two-dimensional or three-dimensional format. A remaining problem with CTC is the variability of diagnostic accuracy among readers. According to the report, this issue could potentially be addressed with CAD.

Perumpillichira et al. [19] provide an overview on research towards CAD systems related to Virtual Colonoscopy. Most the research prototypes that were built employ the steps of extracting the colonic wall, detecting polyp candidates, reducing false-positives and, eventually, displaying of detected polyps. For the extraction of the colonic wall, the contrast of the CT values between wall and air inside the inflated colon lumen is used. The isotropic volume is reconstructed from the axial CT images and thresholding is applied. It is possible to extract approximately 98% of the eintire colon. Extraction of geometric features is then used to detect polyp candidates. Various methods have been developed and shown to be effective for detecting polyps. Among them are sphere fitting, surface normal overlap, volumetric shape index and curvedness, and others. False-positive candidates can be reduced using methods, such as texture analysis and gradient concentration, CT attenuation, random orthogonal shape section, and optical flow. In particular, texture analysis has shown to be effective for filtering out false-positives caused by stool. For the final decision on classifying a candidate as a polyp or not, a statistical classifier can be used. Several different solutions have been researched for this purpose. Among them are quadratic discriminant analysis, neural networks and SVMs. The article quotes several studies and concludes that CAD has a performance comparable to the one of a human detecting polyps. As an important finding of one of the studies, it also mentions that CAD significantly decreased the interobserver variability among three radiologists.

2.4 Summary

In this chapter, we discussed several articles related to our work. We have found some approaches that seem promissing and might be worth considering for implementing a detection pipeline. Further, we also highlighted problems or issues where we could find them. It is worth mentioning, that there is a significant amount of related literature available, suggesting that the problem we are trying to address has already been researched for quite a while. Several algorithms or methods have been proposed and have achieved very promissing results in their respective testing environment. However, we have noticed that in some cases, it is unclear how well the approach would perform under ”real world” conditions. Several articles use a rather small amount of training and testing data. This makes it very difficult to generalize beyond that specific dataset; overfitting could therefore be the reason for the good results.

Table 2.1 presents a summary of the discussed approaches for detecting polyps. The different articles provide different metrics for measuring the performance and use different datasets for training and testing. Therefore, some cells in the table are incomplete and it is difficult to conclude with a direct comparison of the different approaches.

Table 2.1: Performance comparison of polyp detection approaches discussed in this chapter.

Publication	Detection Type	Recall	Precision	Specificity	Accuracy
Mamonov et al. [39]	polyp / shape	47%	N/A	90%	N/A
Hwang et al. [23][3]	polyp / shape	~96%	~83%	N/A	N/A
Li and Meng [43]	tumor / textural pattern	88.6%	N/A	96.2%	92.4%
Zhou et al. [13]	polyp / intensity	75%	N/A	95.92%	90.77%
Alexandre et al. [11]	polyp / color pattern	93.69%	N/A	~76.89%	N/A

For classifying video endoscopy imaging data, we have found that most approaches either already use an SVM in some way, or could be used as a pre-processing step for one. The features, which are fed to the SVM or the binary classifier vary a lot depending on the approach. We have discussed methods that use physical dimensions, grayscale intensitiy values, gradient orientation or RGB color information as input to the classifier. Each of these methods has its respective advantages and disadvantages. In the subsequent chapters, we will describe the design of a solution, which we can use to overcome the dataset

[3] This classifier is classifying complete shots, and not single frames.

problem by collecting more data for testing. The same solution can also be used to implement and test different machine learning approaches and algorithms for image recognition.

Chapter 3

Object Tagging

With this thesis, we research solutions for automating the detection of lesions in colonoscopy videos. Automating this process is a prerequisite for mass-screening for colon cancer and its precursors, using camera capsules. We evaluate different filtering and machine learning approaches for this purpose. For experimenting with machine learning methods, we first need to build a ground truth dataset. Therefore, we have built several tools for creating training and testing datasets from video sequences.

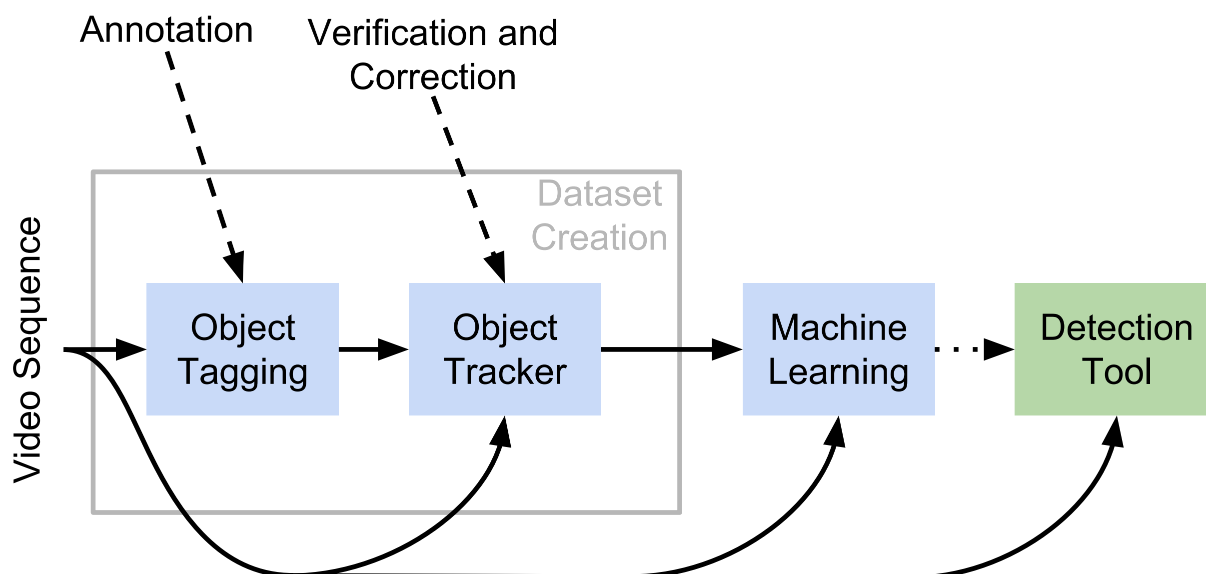


Figure 3.1: The dataset creation process is a prerequisite for training an algorithm and eventually building a fully automated detection tool.

The dataset creation process can generally be split into two separate tasks. The first one is to identify and mark any irregularities in a video sequence; we refer to this step as *Object Tagging*. The second task is to track the movement of previously tagged objects within the same video sequence; we refer to this second step as *Object Tracking*. The process for the dataset creation and the subsequent steps for eventually creating a fully automated detection tool are shown in Figure 3.1.

In the first section of this chapter, we discuss the design and implementation of the tools we have developed for collecting data by *Object Tagging* and annotating videos. *Object Tracking* will be discussed in Chapter 4. In the second section of this chapter, we evaluate the tools we have created by presenting them to potential users, and finally, we conclude this chapter with a short summary of our achievements.

3.1 Design and Implementation

The work of evaluating colonoscopy video sequences and detecting irregularities is a task that has to be done by a specialist. This is to make sure that all relevant regions, and relevant regions only, are marked. Or in other words, as this is our ground truth dataset, we have to make sure to avoid any false positives and false negatives. We have therefore implemented and experimented with several different tools to make it easy for medical specialists to provide us with the needed information. In this section, we describe the different tools and implementations that we came up with.

3.1.1 Prototype 1: Object Tagging and Manual Tracking

The purpose of this very first object-tagging prototype[1] was to find a convenient way, for specialists at Rikshospitalet, to collect ground truth information by tagging lesions in colonoscopy videos. The tool, as seen in Figure 3.2, was written in *Python* using *Qt* and *OpenCV*. We chose *Python* because it is a very convenient and efficient language for doing rapid prototyping, and because there are bindings for many toolkits and third party libraries available. For the same reasons, we chose *Qt* for implementing the user interface. At this point, *OpenCV* was only used to read and decode video files. A further reason to choose this combination of technologies was that all of them are available on all major platforms such as Windows, Linux and Mac OSX.

This first prototype has a very basic User Interface (UI) providing the regular controls of a video player. A specialist would use the application to watch a colonoscopy video, and whenever he finds a lesion, he would use the mouse cursor to mark it as a ROI. A mouse click into the video frame creates a round semi-transparent overlay that will then move with the mouse cursor and must be kept to cover the ROI on subsequent video frames. A second mouse click on the video frame hides the overlay again and stops the recording of positions and dimension of the ROI. The size of the ROI can be adjusted using the mouse wheel.

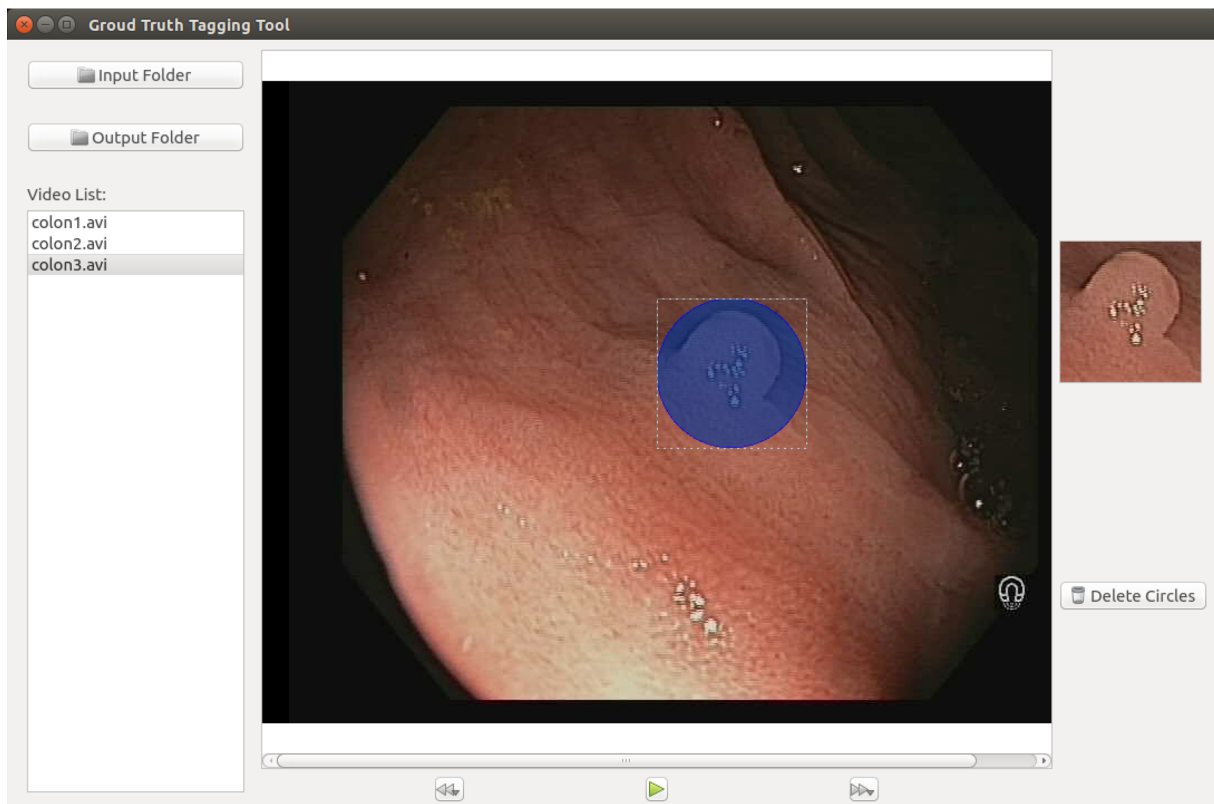


Figure 3.2: User Interface of the prototype written in Python using Qt and OpenCV

[1]This first prototype was jointly developed by Jiang Zhou from Dublin City University.

We have been experimenting with the first prototype ourselves for a while, and we came to the conclusion that this approach has several flaws, in particular with regards to the usability of the tool. Some of the issues we found are listed subsequently:

- Using a semi-transparent selection area attached to the mouse cursor for hovering an object while the mouse button is not pressed, seems to be counterintuitive and confusing for the user. The fact that positions for frames are recorded from a first mouse click to a second one is in particular confusing, as the mouse cursor is also used outside of the video frame to control the playback buttons.

- Depending on whether the mouse wheel or touch pad being used, it is very difficult to control the size of the selectied area. Especially on laptops with a touch pad, this is a problem, as size and position can not be modified at the same time. Yet, to stop the video or change the playback speed, it is necessary to move the cursor outside of the video frame to use the player controls.

- While all the technologies used are available for all major operating systems, the combination of several libraries and their dependencies are hard to manage when supporting multiple target operating systems.

- *Qt* is written in C++ and is also meant to be used as a C++ toolkit. *PyQt* is a third party product that provides *Python* bindings for *Qt* libraries. These bindings must be created matching a specific major, minor and bugfix version of *Qt*. Using *Python* bindings for a toolkit-API, which is meant to be used with C++ often feels very unnatural and leads to a rather poor development experience.

- Packaging of the tool is a difficult task, and there is no standard solution across several operating systems. Packages for deployment must be built using different third party tools for different operating systems. For *Windows*, *py2exe*[2] can be used for deployment, and for *Mac OSX*, *py2app*[3] is a solution. This is mainly a problem because *Python* is an interpreted language, which does not compile to binaries in form of a single file that could then be deployed to a target system.

- *Python* exists in two major versions 2 and 3. While *Python 3* is meant to replace *Python 2* in the long run, both versions are still actively developed, and *Python 2* is still widely in use. *PyQt* is nowadays targeting *Python 3*, but, unfortunately *OpenCV* is still usually targeting *Python 2*. So, we need to create special builds of one of the libraries to be able to use *PyQt* and *OpenCV* together.

A first short presentation to specialists at Rikshospitalet, who are currently the primary user of the software developed for this thesis, led to the conclusion that a more automated approach is expected. In particular, the time a specialist spends on tagging a video should be minimized.

Because of this requirement and due to the technical reasons mentioned before, we decided to research an approach, where we divide the process into two separate parts: *Object Tagging* and *Object Tracking*. While the expertise of a specialist is needed for recognizing and tagging any irregularities, a specialist's knowledge is not necessary to track a previously tagged object in a video sequence. Therefore, the specialist is only needed for the first part of the process, and a lot of his time can be saved by automating the tracking of any initially tagged regions. Further, this architecture leads to simplifications for deployment and allows us to choose technologies that are more suitable for the specific task of tagging or tracking, respectively. *Object Tracking* is discussed in detail in Section 4.

3.1.2 Prototype 2: Web-Based Object Tagging

Based on the experience we gained from our first *Python* based prototype and the finding that the dataset creation process might benefit from being split into two separate steps, we started evaluating web technologies for the object tagging part.

The requirements for a technology that we can use to implement object tagging in videos are rather low; the only real requirements are the ability to playback video and to allow recording selections that

[2] `http://www.py2exe.org/`
[3] `https://pythonhosted.org/py2app/`

reference a specific point in playback time of the video. The HTML5 video element and JavaScript provide both. We considered a web-based solution to be especially suitable for our needs, because it circumvents the need for deploying our software to a target system. Assuming that a recent web browser is installed, there is no need to install any additional software on the computer our system is accessed from. Deployment to a target system in a restrictive medical environment can be rather difficult due to data privacy concerns and the exceptionally sensitive data that is usually processed in such an environment. A solution based on web technologies circumvents this problem in a rather elegant way, as the only requirement to the target system is an HTML5 compliant web browser and an Internet connection.

As described in Chapter 2, there already exists a web-based video annotation tool called Videojot [33]. Despite our use case being slightly different, this tool was a very good starting point. Videojot is using free hand annotations. For our use case, we needed to implement rubber-band-selection. We have also added a slider for controlling the playback speed, allowing to screen irrelevant parts of the video more quickly and assess relevant parts more thorough at a slower playback rate. The UI of our tagging tool is shown in Figure 3.3. It provides the usual start, stop and pause controls of a regular video player. Additionally, we added a seek bar that highlights the playback position and any regions of interest in colors. We also added ”seek-forward” and ”seek-backward” buttons that allow stepping to the next or previous ROI.

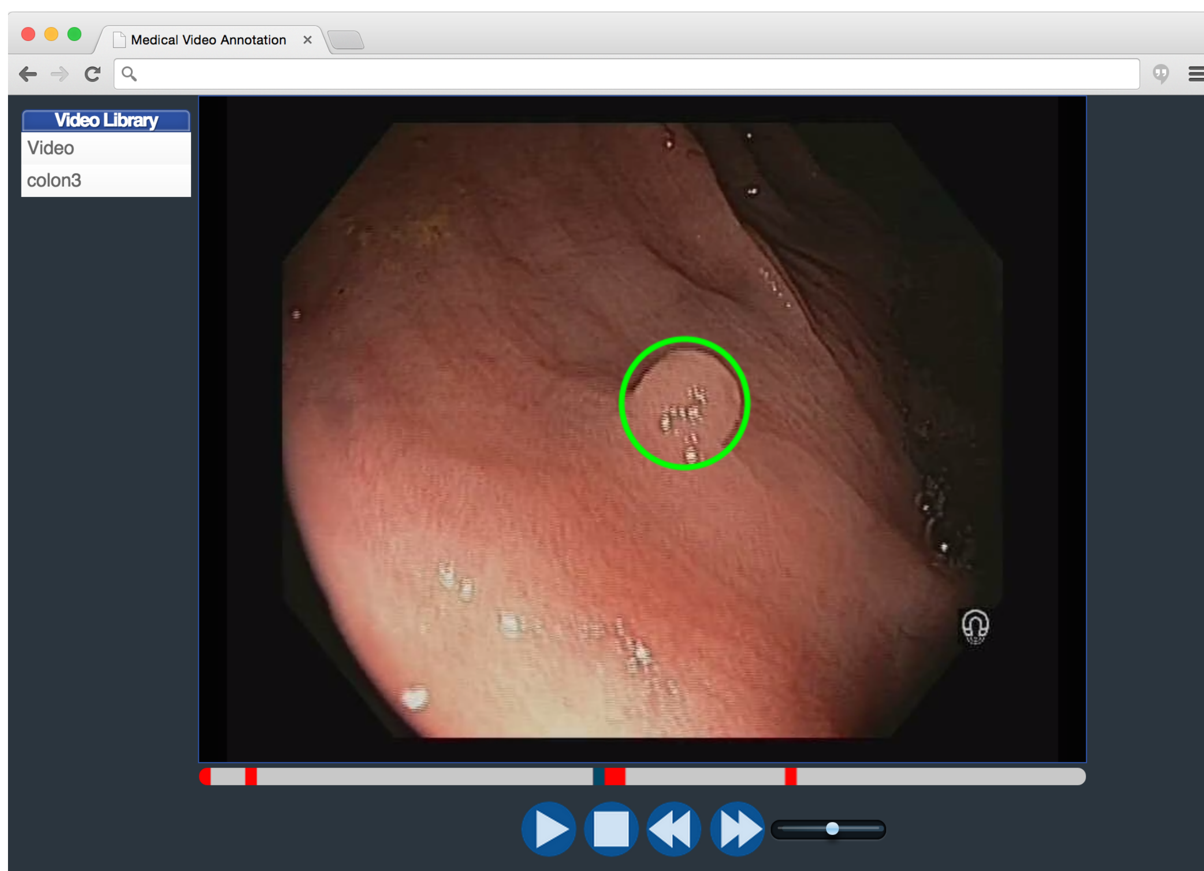


Figure 3.3: The user interface of the HTML5 based video tagging tool.

Our web application is mostly written in HTML5 and JavaScript. Specifically, it makes use of the HTML5 video element. Listing and uploading videos and storing tagging information is implemented in Java and runs in an Apache Tomcat servlet container[4]. All video sequences are uploaded to the server through the web interface. On the server, we use a Java Servlet, which spins off a job to transcode the video to H.264. For transcoding we use *libav* and *avconv*[5]. Transcoding is necessary in case the original video file is not encoded in a codec that is supported by the browser. H.264 seems to be a good choice,

[4]http://tomcat.apache.org
[5]https://libav.org/avconv.html

as all major web browsers currently support it. The transcoding job is running asynchronously, so a connection to the server is not needed to keep the job alive. Once the transcoding process has finished, the video shows up in the list of available video sequences for processing.

As the video playback in HTML5 is running outside of the JavaScript execution thread, we do not have a strict control over the video frames being displayed. The playback position is only provided as a floating-point value property, called *currentTime*, in seconds. The property can be read and it can also be set in order to seek to a specific position. When executing JavaScript code, this property can be read at an arbitrary point in time. Since a single video frame is usually being displayed for about 40ms[6], this means that, when playing a previously tagged video sequence, we will most likely not read the same value from the *currentTime* property again as we were reading when creating the tag. Therefore, visibility of a previously created tag cannot be guaranteed during playback, and we must use the seek buttons to seek to the next or previous ROI.

Whenever a ROI has been selected, an editor shows up and allows the specialist to enter a classification and a comment. This information will be stored together with the tagged rectangle in JavaScript Object Notation (JSON) [51] format on the server.

An example of two selections stored in JSON format is presented in Listing 3.1. All numeric values are stored as floating-point numbers. This is not because of the need for accuracy beyond the size of a single pixel, but a result of the direct exporting of the data from JavaScript after using division operations to transform selection points from viewport coordinates to video coordinates. The *singleShot* key/value pair is needed to strictly mark a selection as a single event, which should not be considered for geometry updates on subsequent frames. The web application for tagging will always set this property to *true*, as geometry updates for subsequent frames will only become relevant in the *object tracking* step. An actual JSON file stored on the server usually contains more redundant information due to the direct export of JavaScript data structures.

Listing 3.1: JSON format used to store tagging information.

```
{
   "markings":[
      {
         "startTime":37.863281,
         "stopTime":0,
         "geometryUpdates":[
            {
               "time":37.863281,
               "x":326.60123365077436,
               "y":302.35123365077436,
               "width":99.29753269845128,
               "height":99.29753269845128
            }
         ],
         "classification":"polyp",
         "description":"This is a typical example of...",
         "singleShot":true
      },
      {
         "startTime":45.5625,
         "stopTime":0,
         "geometryUpdates":[
            {
               "time":45.5625,
               "x":363.3332569774822,
               "y":213.55200697748225,
               "width":100.89598604503551,
               "height":100.89598604503551
            }
         ],
         "classification":"bleeding",
         "singleShot":true
      }
   ]
}
```

[6] assuming a usual frame rate of 25 frames per second

3.2 Evaluation and Discussion

By talking to doctors and researches, we have learned that they usually have more urgent duties than spending a lot of time tagging videos and delivering information to non-experts. Because of this, it is very important to make the task of collecting and tagging video sequences as easy and time efficient as possible. We believe that with our online tool for tagging video sequences, we have developed a prototype for a solution that minimizes the time a specialist spends on collecting and tagging data for our purpose. In particular, the proposed solution includes the following features, which should make the whole process more convenient for a specialist:

- Drag and drop can be used to upload entire video sequences. No manual steps are needed for delivery of the data.
- A wide range of video codecs is supported and will be transcoded automatically on the server side. A user therefore never needs to make sure he uses a specific format or codec.
- Every irregularity only needs to be tagged a single time. The user does not need to track a ROI through multiple video frames. (Requires object tracking as a second step.)
- An irregularity can be tagged in any frame. It is not required that a tag is created in the first video frame an irregularity appears. (Requires object tracking as a second step.)
- No installation on user side is necessary. Since we only use JavaScript and HTML5 technologies, the only requirement is a HTML5 compliant web browser.

We have presented our solution at different stages during development to specialists from Rikshospitalen. We have received the feedback that our prototype based on HTML5 is easy to use and is an acceptable solution for providing us with further video data.

We would like to mention that the availability of training data is crucial for any future research project with a similar approach. It is important to have sufficient data available early on. Multiple instances of our web based tagging solution could be run simultaneously for that purpose, and credentials could be provided to multiple entities. This would make it more likely to collect enough data within reasonable time.

3.3 Summary

In this chapter, we discussed our prototyping efforts for tools to collect ground truth from hospitals and doctors. For this purpose we have first developed a prototype in *Python*, for tagging lesions in colonoscopy videos. After a first review of this prototype with potential users, we have decided to split the process of collecting ground truth into two sub-tasks, *Object Tagging* and *Object Tracking*. This allows to reduce the time a specialist needs to spend for the initial tagging of lesions, as we can do the tracking of the selected lesions by ourselves. Also, we can mostly automate this process, once we have the initial tags. Therefore, we have developed a web-based tool for the *Object Tagging* task. Using a web-based solution for the initial *Object Tagging*, also makes the deployment of the software much more easy, as the only requirement is a recent web browser with HTML5 support. The user interface of the tool is minimal, intuitive and time-saving to use, and is an acceptable solution for the potential users. In order to further save time in the tagging operation where the tagged object needs manual tracking through a number of frames, we next introduce an approach to automatically track an object marked in the video.

Chapter 4

Object Tracking

In this chapter, we describe different approaches for the tracking of objects in videos to create a complete training dataset. The first section of this chapter discusses the design and implementation of approaches we have considered and the solution we eventually settled for. In the second section, we evaluate our solution and measure the time we can potentially save using our solution for object tracking. Finally, we summarize our work related to object tracking in the last section.

4.1 Design and Implementation

The object tracking part is to track the ROI on previous and subsequent frames, based on the previously manually created tags during object tagging. The output of this part is a dataset containing a completely annotated video sequence. All occurences of lesions must be correctly tagged, and no false positives or false negatives must be present, as this data will be used as ground truth for machine learning. However, this part is more about tracking an object and adjusting the size and position of the tracked region than about identifying or recognizing irregularities. A specialist's knowledge is therefore not required.

4.1.1 Object Tracking in JavaScript

We have considered implementing the object tracking part in JavaScript as well. There are already object tracking and face recognition algorithms available for JavaScript. A fairly prominent example is *tracking.js*[1]. A web-based solution would also benefit from the implicit abstraction provided by the web platform and make it possible to easily deploy it to any computer that has a web browser and an Internet connection. A further advantage of doing the object tracking on the client side is that the tracking result is immediately available to the person using the system. Such a solution could then be integrated into a single application with the *Object Tagging* step. Unfortunately, a few experiments and more thorough studying of the HTML5 video element API revealed that this approach is not feasible, due to the limited control over the video element.

We have in particular identified the following limitations:

- The HTML standard specifies the attribute *currentTime*[2] as a readable and a writable attribute. On reading, it is supposed to return the current playback position in seconds as a floating-point value. When writing to the attribute, the media player implementation must seek to the position provided as an argument. This is the most fine graned control available for changing playback position of the video. Seeking to a specific frame is therefore only possible by calculating a playback time based on the expected frame rate. For our purpose, this is not accurate enough, as we have to be sure that any selection made by the user or by the tracking algorithm is recorded with the exact frame it belongs to.

- The video element does not provide a direct way to access the video textures. Instead, the video element must be queried at a regular interval, and the *drawImage* function of a *2d canvas context*

[1] `http://trackingjs.com`
[2] `https://html.spec.whatwg.org/#dom-media-currenttime`

must be used to mirror the image to a hidden *canvas* element. Only then, it is possible to access the pixels directly from the *canvas* object using the *getImageData* function.

- The video element does not provide a way to execute a JavaScript function for every new video frame being displayed. The lack of a notification when the next frame is to be read makes it impossible to trigger the object tracking algorithm exactly once for every frame. It would basically imply guessing when a frame should be read and fed to the tracking algorithm.

4.1.2 Object Tracking in Google Native Client

Google Native Client (NaCl)[3] is a technology that allows executing compiled native code through the so called Pepper API within a Chromium or Chrome web browser. This technology requires writing a plugin that needs to be compiled with a special NaCl or Portable Native Client (PNaCl) toolchain. PNaCl allows compiling code to a bitcode executable, which is then translated to host-specific executable as soon as the plugin is loaded by the web browser. NaCl provides full support for C and C++ and allows making use of multiple CPU cores during execution. It also exposes the capabilities such as Single Instruction Multiple Data (SIMD) vectors. NaCl therefore seems like a very good fit for video processing and object tracking. We have given this approach a try and we successfully built *OpenCV* for PNaCl. This allowed us to run simple examples such as face recognition on images or graphical effects on the video stream from a webcam.

For security reasons, NaCl is running inside a sandbox. Direct access to the hardware or the file system is therefore not possible. Any data needs to be fed in through the Pepper Plugin API (PPAPI) using JavaScript. In case of a video stream from a webcam, this is fairly simple. We can get hold of the video stream using the "Navigator.getUserMedia()" API[4]. Using this API, we can request access to the webcam and provide a callback that will receive a handle to the media stream, in case access is permitted. For our case, this API is not very relevant though, as we do not want to access a stream from a camera but rather process an existing video file. Loading the file directly from disk is not possible because of the sandbox. The more appropriate approach would be to make use of the HTML5 video element, and then capture a stream from there to process it. From an architectural point of view, this would be perfect. We would leverage the capabilities of the full multimedia stack of a web browser and still harness the power of native code through NaCl. This is also how the API is designed in drafts for web standards[5]. We could simply use the "HTMLMediaElement.captureStream()" extension to get hold of the media stream from the HTML5 element. Unfortunately, at the time of writing this document, this mechanism is not implemented in Chromium, and this prevents us from using this approach.

The third option for using NaCl would be to load the video from a URL using URLLoader[6]. But, also this option would lead to several problems. It would mean to implement not only the tracking, but the whole playback of the video as well as the selection of ROIs in NaCl. Also, NaCl is currently limited to a total of 1GB[7] addressable memory. For long video sequences, this seems to be a rather tight restriction. Considering this restriction and that there is no real toolkit available for NaCl except for OpenGL ES 2, the NaCl based approach is out of scope for this thesis.

4.1.3 Object Tracking on the Server Side

We have also been experimenting with a solution that would use HTML5 for the user interface, but delegate the actual object tracking to the server. Such an approach would again have the advantage of a very easy deployment to the user. For a big part, we could also reuse the user interface and the server-side setup from the *object tagging* part. Whenever the user selects an ROI for tracking, we send a tracking request containing the coordinates, dimensions of the ROI and the current video playback time to the

[3] `https://developer.chrome.com/native-client/nacl-and-pnacl`
[4] `http://w3c.github.io/mediacapture-main/getusermedia.html#idl-def-NavigatorUserMedia`
[5] `http://w3c.github.io/mediacapture-main/getusermedia.html#idl-def-NavigatorUserMedia`
[6] `https://developer.chrome.com/native-client/pepper_dev/cpp/classpp_1_1_u_r_l_loader`
[7] `https://groups.google.com/forum/#!topic/native-client-discuss/IFuLcxCWWh8`

web server using the XMLHttpRequest API. The server processes such a request in a Java Servlet. This Java Servlet then spins off a separate process that tracks the selected ROI and prints the tracking results for a predefined amount of frames to the standard output stream. The servlet reads the output of the tracking process and forwards the information to the calling web application. The object tracking itself is implemented in C++, using *OpenCV*[8].

The problem with object tracking on the server is the latency introduced by communicating over the network. With this very naive approach, we measured round trip times of about 45ms without any actual tracking being done on the server side. Spinning off a process from the Java Servlet and initiating the tracking easily increased this to above 100ms. Of course, the overhead could be significantly decreased, for example, by implementing the tracking directly in the Java Servlet using *OpenCV* instead of spinning off a separate process. However, getting the tracking anywhere close to 25 FPS, with tracking on the server, seems challenging.

A further problem with this approach is the communication between client and server during the tracking phase. As the client is waiting for the tracking results, we would want the server to reply as quickly as possible. But, after replying with tracking results for the first frame, the server should keep sending updates to the client for subsequent frames. Due to the nature of HTTP this is slightly problematic and not easy to achieve. Using basic HTTP, there is no direct way for a server to push information to a client. Instead, a client has to request information from the server using a GET/PUSH request. This essentially means that the client needs to poll the server by sending a new request for every subsequent frame, or that we need to process multiple frames before sending a first reply. Both of these approaches imply introducing yet a further delay, either due to synchronous polling or due to the time required for processing multiple frames. A potential solution to this problem is the WebSocket[9] standards. Using WebSockets would allow building a two-way communication channel between client and server.

We believe that building a solution for tracking on the server side is possible. But, an implementation of this is beyond the scope of this thesis, and we therefore do not research this approach any further. Instead, we focus on a more simple approach based on building a native tool for the object tracking part.

4.1.4 A Native Tool for Object Tracking

After several experiments, we decided to implement a separate native tool for the object tracking step in C++ using *Qt*[10] for the user interface, and *OpenCV* for reading and processing the video data. We further integrated *Struck* [36] as well as *TLD* [37] for tracking the tagged regions. Both *Struck* and *TLD* work reasonably well. We observed that *TLD* is providing better results when the tracked object is not only moving, but also rotating and therefore changing its appearance over time. Further, *TLD* has a built-in functionality to report when the tracker has lost the object.

The user interface for our tracking software is similar to the web interface described previously and can be seen in Figure 4.1. It features a video widget, play, seek-forward and backward buttons, as well as a seek bar with identical behavior. A slider to increase or decrease the playback speed and an editor for classification and description are present. Further, a button for playing the video in reverse direction and context menus for modifying regions of interest are provided.

After starting the application, a JSON file that was created using the tagging web application can be opened, and the respective video file must be selected. We use the original video file instead of the H.264 encoded one, because we need to be able to play the video forwards and backwards frame by frame. Recreating frames in reverse direction is very expensive with H.264, since frames can be encoded by referencing previously encoded ones. The original files uploaded to our server are usually simple MJPEG video files and are very well suited for playing in both directions. The annotations that are imported from the JSON file are displayed in shape of red rectangles on top of the respective video frames. These selections can then be chosen for tracking, or can be modified by the users. The users use the seek buttons to seek to the next or previous regions of interest. Then they use the context menu to select one or multiple regions for tracking. Playing the video in either direction will then track the

[8] `http://www.opencv.org`
[9] `http://tools.ietf.org/html/rfc6455`
[10] `http://www.qt.io`

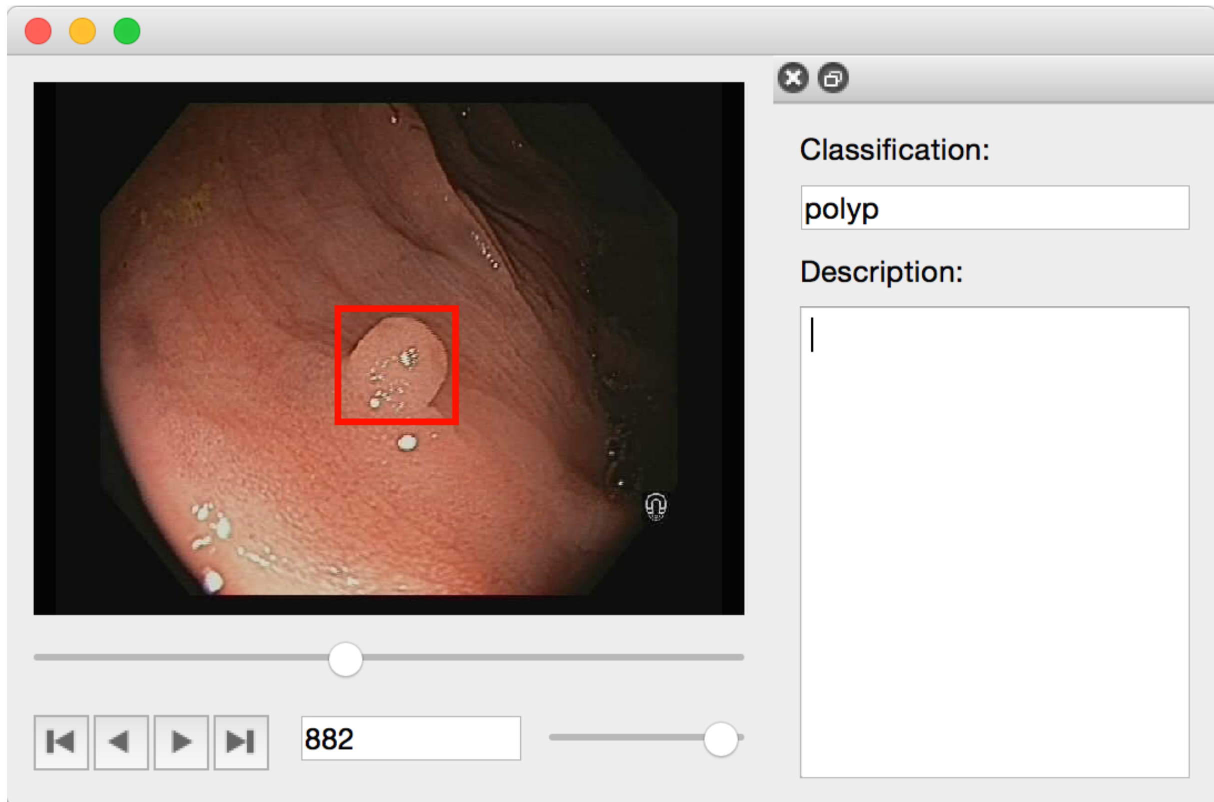


Figure 4.1: The user interface of the tracking software implemented in C++ using Qt and OpenCV.

region in the video frames being displayed. Alternatively, the arrow keys can be used to step forward or backward frame by frame. The playback can be paused at any time to adjust size or position of the tracked region in case the dimensions changed significantly or the tracking result is not satisfying any longer.

Using double buffering allows reading and processing the next frame while the previous frame is still being displayed. The processing (reading of frames and tracking of regions) is therefore running in a separate thread. The communication between the user interface and the worker thread is implemented using *Qt*'s events delivery mechanism[11]. This is an event loop based delivery mechanism, which delivers event-objects to instances of *QObject*-derived classes, calling their respective *event()* function.

Whenever the tracking algorithm fails to track a region, the playback stops automatically. It is then up to the user to decide if the tracked region should be removed or if the tracking should be re-initialized with an updated region. The user can seek forwards and backwards freely to review the tagging and tracking results and adjust and move a region, or restart the tracking of it at any point during the process. Once the tracking of regions is complete, all the information can be saved to a JSON file. This file will also reference the video file that was used. If the JSON file is re-opened, the video file will be found automatically. In case the video file has been moved, the user will be asked to select the matching video file manually.

The format of the stored JSON file is an extended form of the JSON format that is already used by the web-based tagging application. A shortened example of this format is presented in Listing 4.1. In comparison to the JSON format from Listing 3.1, a marking can now contain multiple updates for its geometry. In this case, the *singleShot* property is removed and a valid *stopTime* property must be added to mark the first frame where the marking is not present anymore. Further, the format also contains a list of frames that can be used for negative samples (*negativeFrames*) and the absolute path to the video file.

After processing a video sequence with the tracking software it is also possible, to feed the dataset, consisting of the video sequence and JSON file containing the annotations, back into the tagging software for displaying with a web browser and verification by a specialist.

[11] `http://doc.qt.io/qt-5/eventsandfilters.html`

Listing 4.1: JSON format used to store tracking information.

```
{
    "markings": [
        {
            "classification": "polyp",
            "description": "description for polyp1",
            "geometryUpdates": [
                {
                    "frame": 874,
                    "height": 112,
                    "time": 34.960000000000001,
                    "width": 112,
                    "x": 342,
                    "y": 258
                },
                {
                    "frame": 876,
                    "height": 112,
                    "time": 35.039999999999999,
                    "width": 112,
                    "x": 341,
                    "y": 257
                },
            ],
            "startTime": 34.960000000000001,
            "stopTime": 35.079999999999998
        },
    ],
    "negativeFrames": [ 1129, 1130 ],
    "videoName": "/Users/zeno/Desktop/Medical Videos/colon3.avi"
}
```

4.2 Evaluation and Discussion

The solution we have implemented for object tracking is meant to reduce the amount of work that is needed, to create a dataset by tagging irregularities in a video sequence. For creating a dataset, we want to extract all the regions of interests from all video frames they appear in. We therefore first need to select all the regions on all the video frames, so we know which parts of the video frames should be extracted. Instead of using object tracking, it would also be possible to do this step manually. We would then have to mark any ROI on any video frame manually. For doing this, we could either use the web based tagging tool, or the *TagAndTrack* tool that we have developed. We consider *TagAndTrack* to be more suitable for the task, as it provides more fine-grained control over the video playback. Being familiar with the tool ourselves, we manage to manually process about 100 frames in 6 minutes. Using the built-in object tracking mechanism instead, we manage to process the same 100 frames in about 20 seconds on average. During these 20 seconds, we had to select the region once and adjust the size of the tracked region twice, meaning we have reduced the amount of time spent on this task by about 94%. These numbers are also presented in Figure 4.2. This is of course just a very rough measurement and calculation. The actual speed of completing a dataset very much depends on the need for seeking/stepping forwards and backwards and on the speed the camera is moving at. Nevertheless, the rough numbers provide a feeling for the amount of time that we can save using object tracking.

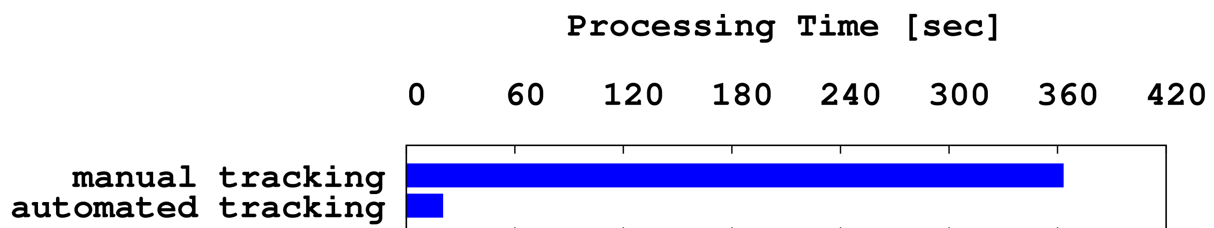


Figure 4.2: Time comparison of processing 100 video frames, using manual or automatic tracking.

The weakness of our implementation is that the tracking rectangle always retains the same size and is not resized when the tracked object changes in size. So, whenever a ROI significantly changes in size, a manual adjustment of the tracking rectangle is necessary, still slowing down the process significantly. So, it might be worth trying to implement an automatic resizing of tracking regions to reduce the time a user spends on creating a dataset even more.

4.3 Summary

In this chapter, we have evaluated several different options for the *Object Tracking* task. The purpose of *Object Tracking* is to track any previously selected regions in the video, and thereby generate a complete dataset that we can use for machine learning. We have found that a regular native application is the most suitable solution for our purpose and have developed a tool that we call *TagAndTrack*. *TagAndTrack* can load the output data from our web-based tagging tool, select new regions, track selections and eventually export the data in formats suitable for machine learning using *opencv_traincascade* or a *dlib* based HOG-Trainer.

Using the automated tracking that we have built into *TagAndTrack*, we can reduce the time needed for creating a fully tagged dataset by 94%. A fully tagged dataset is a video where any lesion that was selected once, is also marked in any previous and subsequent frames it appears. The ability of *TagAndTrack* to play and track selections both forwards and backwards in the video sequence is a further improvement, as this implies that we no longer require a lesion to be marked in the very first frame it appears.

We now have a tool that can efficiently tag and track objects in videos to create datasets for machine learning. The next step is to apply filters to the individual video frames for preprocessing. This is addressed in the next chapter.

Chapter 5

Preprocessing and Image Filtering

While our tools allow rather efficient creation of a dataset for machine learning, the actual video sequences still need to be collected. During the time we did not have enough video sequences available for machine learning, we started experimenting with filtering to enhance the available data. For this thesis, the term filter refers to a function or an algorithm that is used to modify a single image, without applying any kind of machine learning or making use of data extracted from other images. In the first section of this chapter, we present the implementations we have created for several filtering and preprocessing methods. In the second section, we evaluate our implementations and present our findings. Eventually, we provide a summary of our work related to filtering in the final section.

5.1 Design and Implementation

We have looked into four basic filtering mechanisms. The first one is what we call *border detection*, and it is meant to mask any irrelevant regions along the border of the video frames. The second one is called *edge detection*, and it can be used to detect any edges within the video, based on a predefined edge threshold. The third mechanism is called *shape detection*, which is a simple detection mechanism built into *OpenCV* for detecting predefined shapes. The last mechanism we discuss is *specular highlight filtering*. This mechanism is meant to remove or reduce reflections caused by a light source, such as the light built into an endoscope.

5.1.1 Border Detection

Border detection is used to mask any irrelevant regions along the edges of the video frames. There are multiple kinds of regions that we consider irrelevant. Several videos that we have received, do have black stripes at the top and at the bottom, which were most likely added automatically to fit a certain aspect ratio. These stripes are obviously irrelevant to our purpose. The video sequences that we deal with are usually recorded with special endoscopy equipment. Several of the videos contain symbols or metadata along the border (see Figure 6.1(c)). The light source is usually mounted directly into the endoscope and the lighting for the video recording is therefore best at the point the endoscope is directly pointing at and is decreasing radially. Together with the size and shape of the lense of a regular endoscope, the variable intensity of the lighting leads to the viewport, which contains the actually usable part of the video, having a circular shape.

The algorithm for *border detection* is fairly simple and can mostly be implemented using functions provided by *OpenCV*. We first convert a video frame to gray scale and then apply a fixed-level threshold to create a binary black and white image. We define that any value below 20 for a pixel is most likely too dark to be actually useful for our purpose and should therefore be masked.

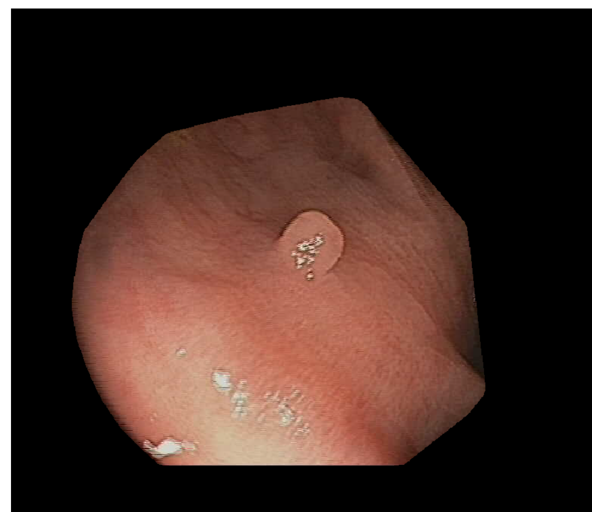
We can now use the *OpenCV* built-in function *findContours* to find any contours in the remaining 8-bit single-channel image. We are only interested in the extreme outer contours. We therefore pass the argument *CV_RETR_EXTERNAL* to *findContours*. This will only return contours of the outer most level, and all the returned contours will be closed.

All the detected contours now need to be processed further. As we are expecting a circular viewport, we want to filter out extremes of the contour detection that could be caused by annotations or symbols along the edges of the video. We therefore ignore any detected contour, covering less than 25% of the video frame. *OpenCV* has a built-in fuction *contourArea*, which takes a contour as an argument and returns the enclosed area in number of pixels. The minimum size of 25% of the video frame has been chosen after short experimentation and could probably be reduced and fine tuned significantly. But, we have not experienced any false negatives with these settings so far. We then wrap a convex hull around any remaining contour and then draw the contour onto a new image buffer initialized with black color and of the same size as the original image filling the enclosed area with white color. With this approach, we have only ever had a single contour remaining, and the result is therefore a black and white image of a convex shape that can be used with a bitwise-and operation to mask the irrelevant borders of the original video frame. The algorithm is described in more detail with comments in Listing 5.1. An example for comparing the original and the resulting video frame is presented in Figure 5.1.

Listing 5.1: Algorithm for border detection

```
static inline void detectBorder(const cv::Mat *video_frame, int borderThreshold, cv::Mat &croppedVideo) {
    // Allocate a buffer that can hold a single contour.
    // cv::drawContours expects a vector of contours, so we need a vector despite the size of 1.
    std::vector<std::vector<cv::Point>> borderContourContainer(1);
    std::vector<cv::Point> &borderlineContour = borderContourContainer[0];
    static cv::Mat gray, threshold_output; // Make intermediate buffers static to avoid re-allocation.

    cvtColor(*video_frame, gray, cv::COLOR_BGR2GRAY); // Create an 8-Bit gray scale copy of the video frame.

    // Apply a threshold on every pixel. Pixels below a value of "borderThreshold" will be painted black.
    cv::threshold(gray, threshold_output, borderThreshold, 255, cv::THRESH_BINARY);

    // Find any extreme outer contours. Store contours in simply approximated form.
    std::vector<std::vector<cv::Point> > contours;
    std::vector<cv::Vec4i> hierarchy;
    cv::findContours(threshold_output, contours, hierarchy, CV_RETR_EXTERNAL, CV_CHAIN_APPROX_SIMPLE);

    static cv::Mat drawing; // Avoid reallocation of the drawing buffer.
    drawing = cv::Mat::zeros(gray.size(), CV_8UC3); // Make sure the buffer is initialized to black.

    // Iterate through all top-level contours. There is usually only 1.
    for (int i = 0; i >= 0; i = hierarchy[i][0]) {
        // A contour has to be at least 1/4 of the image, otherwise we will ignore it.
        double area = cv::contourArea(contours[i]);
        if (area < drawing.size().area() / 4) continue;

        cv::convexHull(contours[i], borderlineContour); // Wrap a convex hull around the contour.

        // Paint the convex hull to the borderContourContainer buffer.
        cv::Scalar color = cv::Scalar(255, 255, 255);
        cv::drawContours(drawing, borderContourContainer, 0, color, -1 /*fill enclosed area*/, 8 /*omit outline*/);
    }

    // Use bitwise_and to mask the video frame with the generated drawing mask.
    cv::bitwise_and(*video_frame, drawing, croppedVideo);
}
```

(a) original



(b) border detected

Figure 5.1: Original video frame and masked video frame after border detection.

5.1.2 Edge Detection

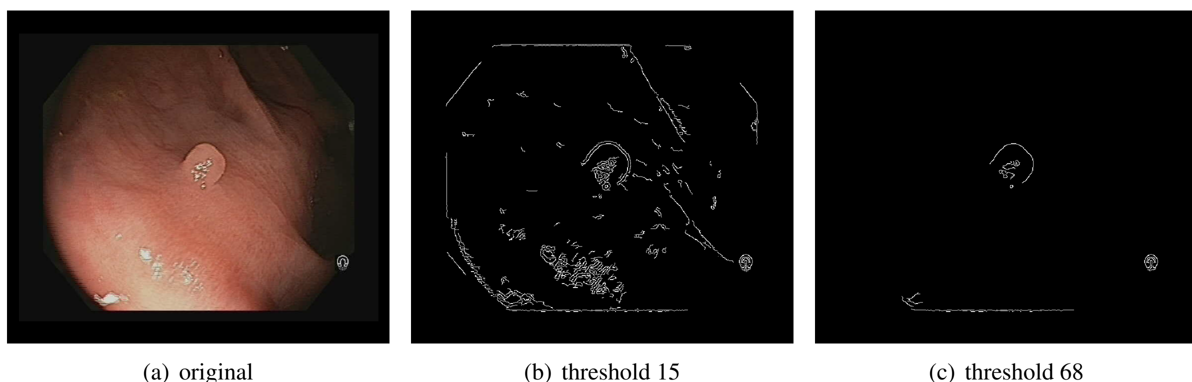
Edge detection is a method to identify points in an image, where the intensity changes rapidly. Such points are usually organized in curves or line segments, and can, for example, be used for feature extraction. To get a better understanding of the potentials for *edge detection*, we have implemented *edge detection* with a variable edge threshold. The process of Canny-based edge detection is starting by smoothing the image using a Gaussian filter. In the second step, the intensity gradient of the image is extracted. The edges within the intensity gradient are thinned by reducing them to the local maxima. Then the low and the high thresholds are used to classify pixels into suppressed pixels, weak edge pixels and strong edge pixels. Eventually, all edges consisting of weak pixels only are surpressed. According to Canny's recommendation [52], the high and the low threshold for the Canny function should be in the range of two or three to one. We therefore define the high threshold to be exactly three times the low threshold. Depending on the selected threshold, we get a different granularity of edges detected. The low threshold can be adjusted using a slider with a range of 0 to 100.

We used *edge detection* to experiment with various settings in an attempt to extract shape features from the image that could then be used further to detect irregularities. The algorithm for this is presented in Listing 5.2. We first convert the video frame to grayscale and apply a blur effect with a 3x3 kernel, to reduce the noise in the image. Eventually, we use the *OpenCV* Canny function to detect edges and generate an output image.

Listing 5.2: Algorithm for edge detection.

```
1 static inline void detectEdges(const cv::Mat *video_frame, int lowThreshold, cv::Mat& edgeDetected)
2 {
3     cv::cvtColor(*video_frame, edgeDetected, cv::COLOR_BGR2GRAY); // Convert the original video frame to grayscale.
4     cv::blur(edgeDetected, edgeDetected, cv::Size(3,3)); // Reduce noise with a kernel 3x3.
5 
6     int kernel_size = 3;
7     int highThreshold = lowThreshold * 3;
8     cv::Canny(edgeDetected, edgeDetected, lowThreshold, highThreshold, kernel_size);
9 }
```

The problem we found with this approach is that, depending on the video sequence and the lighting conditions, we need very different settings for the edge threshold to get a meaningful extraction of shapes. Also, the noise level is very high. In other words, for many video frames, it is possible to find a threshold that leads to the resulting image showing almost only the edges of an irregularity. But, the threshold needed for this is highly specific to the particular video frame, so a different video frame would need a completely different threshold setting. Figure 5.2 shows the same video frame with different *edge detection* settings.



(a) original (b) threshold 15 (c) threshold 68

Figure 5.2: A video frame in the original form, and with two different edge detection thresholds.

Given the possibility of extracting edges of irregularities, we considered that it might be valuable to add *shape detection* on top of *edge detection* with a very low threshold. The idea is to extract shape information using *edge detection* and use the extracted shape information to detect previously learned shapes.

5.1.3 Shape Detection

Based on the previously described *edge detection*, we decided to experiment with a simple approach of manually selecting detected edges for learning, so that we can later detect similar shapes again. We call this method *shape detection*. For this approach, we added a data structure for storing shape data to our program. Also, we added rubber-band-selection and a context menu for selecting and learning shapes.

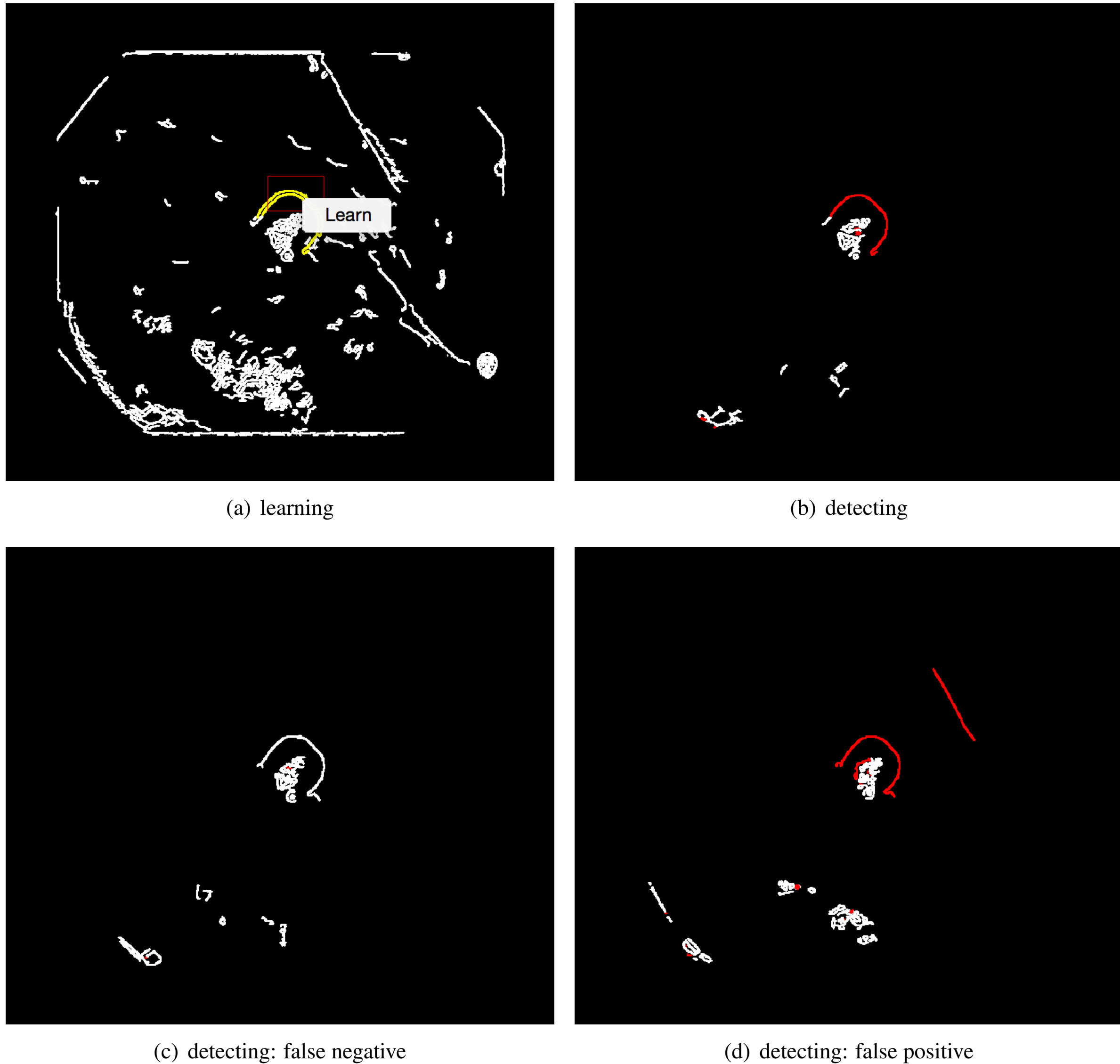
In this method, newly detected shapes are compared to previously learned ones. Shapes are stored as a vector of points and the *OpenCV* function *matchShapes* is used for comparing shapes based on Hu-Moments. In image processing, a moment is a defined weighted value, describing a certain property or geometric interpretation of an image. Hu-Moments are a set of seven moments, which are invariant to rotation, translation and scale and were first presented in [53]. The function *matchShapes* compares two shapes and returns a metric describing the similarity of the two shapes. The smaller the value, the better is the match. The algorithm for *shape detection* is presented in Listing 5.3 with inline comments for explanation.

Listing 5.3: Algorithm for shape detection

```
1  static inline double matchShapes(const std::vector<cv::Point> c, const std::vector<std::vector<cv::Point>> &learnedC)
2  {
3      double match = DBL_MAX;
4      // Iterate through all learned shapes and compare given shape.
5      for (auto it = learnedC.begin(); it != learnedC.end(); ++it) {
6          double d = cv::matchShapes(c, *it, CV_CONTOURS_MATCH_I1, 0.0f);
7          // Only update the value if the match is better (smaller value) than any previous one.
8          match = match < d ? match : d;
9      }
10     return match;
11 }
12
13 static inline void detectShapes(const cv::Mat *img, double detectorSensitivity, const Marking *selection,
14     const std::vector<cv::Point> &borderlineContour, const std::vector<std::vector<cv::Point>> &learnedShapes,
15     std::vector<std::vector<cv::Point>> &selectedShapes, cv::Mat &shapeDetectedVideo)
16 {
17     // Detect shapes in the image and store them in a tree structure, representing the relative positioning.
18     std::vector<std::vector<cv::Point>> shapes;
19     std::vector<cv::Vec4i> hierarchy;
20     cv::findContours(*img, shapes, hierarchy, CV_RETR_TREE, CV_CHAIN_APPROX_NONE);
21
22     shapeDetectedVideo = cv::Mat::zeros(img->size(), CV_8UC3); // Initialize the output image with black color.
23
24     // Clear the data structure holding any selections.
25     selectedShapes.clear();
26
27     // Iterate through all detected shapes and compare them to the learned ones.
28     for (size_t i = 0; i < shapes.size(); ++i) {
29         // Check shapes for being within rubber-band-selection.
30         cv::Scalar color = cv::Scalar(255, 255, 255);
31         if (selection) {
32             std::vector<cv::Point> selectionShape = selection->contour();
33             for (size_t j = 0; j < shape.size(); ++j) {
34                 if (pointPolygonTest(selectionShape, shape[j], false) > 0.0) {
35                     color = cv::Scalar(0, 255, 255);
36                     selectedContours.push_back(shape);
37                     break;
38                 }
39             }
40         }
41
42         // Compare shapes to learned ones and color shapes which are below detectorSensitivity threshold.
43         double match = matchShapes(shapes[i], learnedShapes);
44         if (match < detectorSensitivity) color = cv::Scalar(0, 0, 255);
45         cv::drawContours(shapeDetectedVideo, shapes, i, color, 2, 8);
46     }
47 }
```

In theory, the rotation-, translation- and scale-invariance of Hu-Moments matches our requirement to detect irregularities such as polyps in an arbitrary position and size. But, unfortunately, our experiments quickly showed that basing this matching of shapes, which we call *shape detection*, on top of the previously described *edge detection* does not lead to the expected result. Shapes visible in the image are often detected as multiple individual contours. A contour is just defined by a set of points, and the starting and end points are not strictly defined. Also, for example, the shape of a U-shaped polyp might have small extensions on its sides towards the regular tissue. So, even a very simple shaped polyp is already very hard to detect in that manner. The threshold for detection would therefore have to be very small, but this would then lead to many false negatives. The problem can easily be seen in Figure 5.3. Figure 5.3(a) shows a shape that is learned for later detection of similar shapes. In Figure 5.3(b), that single learned shape is re-detected in a subsequent frame. In Figure 5.3(c), the shape of the polyp that we want to detect has just changed marginally, yet the shape is not detected as a match anymore. In

Figure 5.3(d), on the other hand, the polyp is successfully detected again. But, unfortunately, in this last image, we also have a false positive. For these three examples, only a single shape was learned, and the threshold has been kept the same for all the detections. Yet, we have a correct detection, a very obvious false negative and a very obvious false positive in this sequence, leading us to the conclusion that this very simple approach is not usable for our purpose.



(a) learning (b) detecting

(c) detecting: false negative (d) detecting: false positive

Figure 5.3: Shape detection in the learning and detection phase.

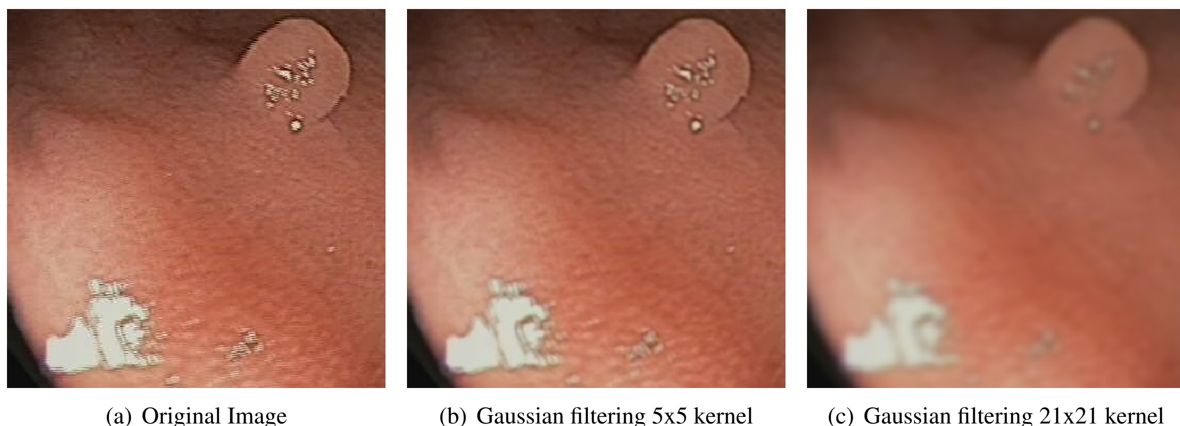
5.1.4 Sepecular Highlight Filtering

Specular highlights are bright spots or regions on a shiny object, caused by a light source. In colonoscopy videos, we can usually find many of these regions, because of the wet and shiny surface of the mucosa. The protruding, often round shape of a polyp, usually leads to reflections right on top of the polyp itself. Without filtering, this would become the most characteristic feature of a polyp; also, this leads to many false positives, because not every reflection actually is a polyp.

Specular highlight filtering is regularly used in photography to remove reflections on skin caused by a flash, for example on portrait pictures. The state of the art method used for this purpose is Gaussian filtering [54]. For photography, this leads to good results, and, as a side effect, the filtering also smoothens out normal irregularities of the skin.

We have tried the same approach for removing specular highlight in colonoscopy videos that we used for the training of our classifiers. However, the results with Gaussian filtering are not sufficient for our

purpose. While Gaussian filtering smoothens the edges of the reflections, the reflections were usually too big to be removed entirely from the image. The reflections we have to deal with are usually significantly bigger than the ones we would find on a portrait picture, mostly due to the wet and shiny surface of the mucosa. On many video frames, we find relatively large areas which do not have any color information left at all. Gaussian filtering would make these areas more fuzzy and smaller, but at the same time, we would also loose a lot of valuable texture information by applying this kind of filtering to the rest of the image. The effect of Gaussian filtering is presented in Figure 5.4. The first Figure 5.5(a) shows the unmodified image containing specular highlights. Figure 5.4(b) shows the same image after applying Gaussian filtering with a kernel of size $5x5$ pixels. In this image, it is visible that the edges of the specular highlight are more smooth and the reflections appear slightly smaller. However, the reflections remain visible, and the whole image appears slightly out of focus. This becomes even more obvious in Figure 5.4(c). It is the same image again, just that for making it more obvious, we have now applied a Gaussian filtering with a kernel of size $21x21$ pixels.

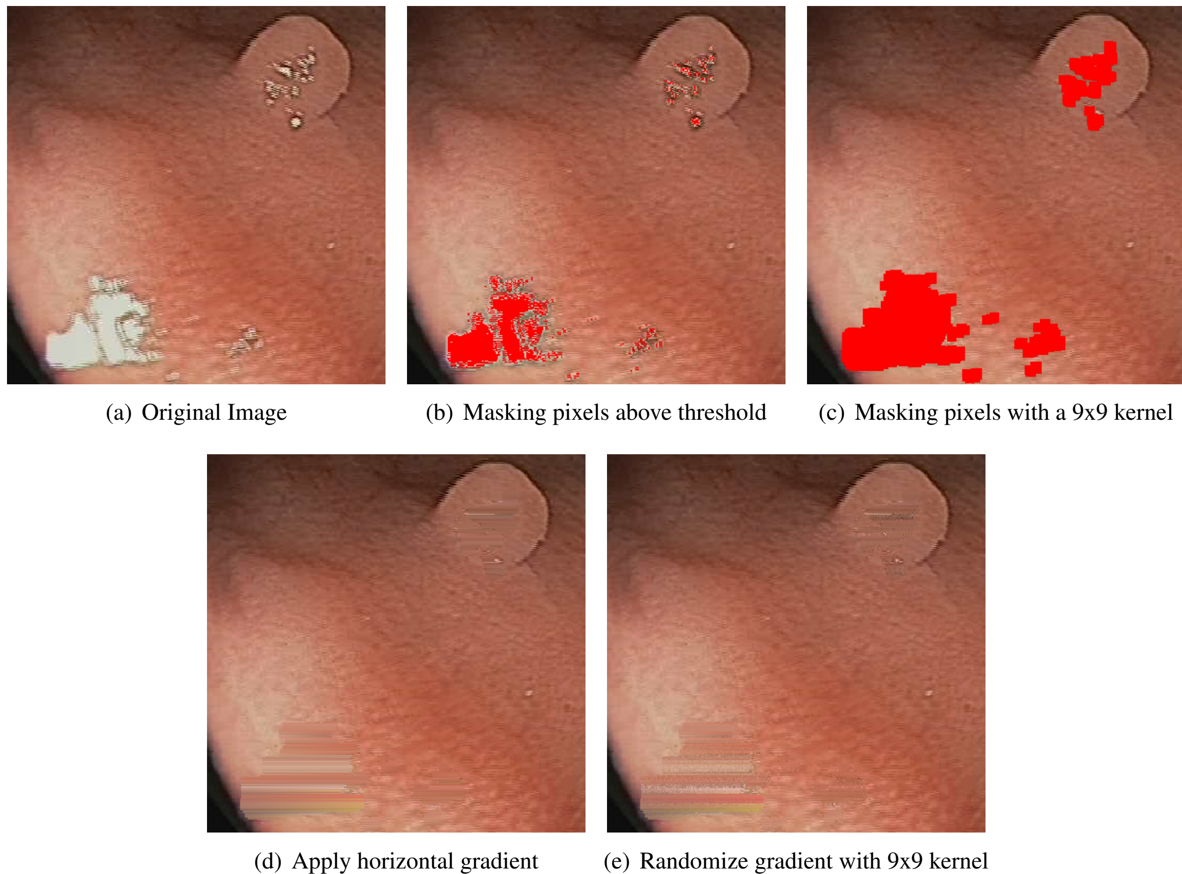


(a) Original Image (b) Gaussian filtering 5x5 kernel (c) Gaussian filtering 21x21 kernel

Figure 5.4: Specular Highlight Reduction by Gaussian Filtering.

As seen in Figure 5.4, Gaussian filtering is not a viable option for our use case. The reflections are too strong to be filtered out, and, at the same time, we loose a lot of detail. In order to minimize the effect of the reflections in our training dataset, we have therefore tried to find a better solution for replacing reflections in the image with estimated or randomized data. The result and the intermediate steps of the algorithm we came up with, are shown in Figure 5.5. The code for this contains a lot of loops and index operations to deal with image borders, pixel offsets and image masks. The code is therefore rather tedious to read and does not explain the concept in the best possible way. We will therefore use images of intermediate steps instead of source code, to describe the concepts.

To detect specular highlight regions, we transform the image into gray scale and use a simple threshold. Every pixel above this threshold is masked. To make this more visible, we have marked any masked pixels in Figure 5.5(b) with red color. As we can see in this figure, there is often a dark contrast line around a specular highlight. This dark ring we do want to remove together with the already masked pixels. For this purpose, we use a kernel of $9x9$ pixels to extend the mask around every previously masked pixel. The result of this is shown in Figure 5.5(c). Once we have determined all the pixels we want to replace, we use a gradient to fill the masked area line by line. Instead of doing this operation line by line, we could do it column by column with essentially the same result. In order to minimize the building of visible lines, we calculate the arithmetic mean of the gradient and the value of the pixel's neighbors in vertical direction for every pixel in Figure 5.5(d). To further reduce the visible lines caused by the modified pixels, we use a $3x3$ kernel for all the previously masked pixels, and randomly distribute the pixels within the kernel. The final result is presented in Figure 5.5(e).



(a) Original Image (b) Masking pixels above threshold (c) Masking pixels with a 9x9 kernel

(d) Apply horizontal gradient (e) Randomize gradient with 9x9 kernel

Figure 5.5: Specular Highlight Reduction by using Gradients and randomized Kernels.

5.2 Evaluation and Discussion

In this section, we will discuss our findings from implementing and experimenting with the filtering mechanisms that we have built into *TagAndTrack*. An algorithm based on geometric analysis, which seems very promising for implementing automated detection, is presented in [39]. We believe, it is possible that an approach based on filtering and machine learning would lead to even better results. However, the focus of this thesis is to use machine learning and building up related software infrastructure for the processing of capsule endoscopy video data. Filtering of the imaging data has therefore only been considered for preprocessing steps. These built-in filtering mechanisms are comparably simple and are meant to be used for experimenting with the technologies available, rather than conducting actual detection tasks.

5.2.1 Border Detection

The mechanism we have implemented for removing border regions is basically masking anything outside of the biggest detected contour in the image, after having applied a threshold on the intensity of the frame. This method has worked very well for all the video sequences that we had available. Using this mechanism, we have successfully removed any border regions which were too dark to contain any really valuable information. For our purpose, we have simply defined this threshold to be a grayscale pixel value of 20. This is an arbitrary number, which we chose after a few experiments and might therefore not be the optimal value. As there is only very little information contained in these dark regions, those usually do not interfere with the actual detection and the significance of *border detection* as a prepocessing step just for this purpose is therefore questionable. However, as a side effect of abstracting the remaining contour to a convex hull, we also remove any symbols or annotations along the borders of the original video frame. With the video sequences that we had available, this side effect was rather important. Most of these videos were not recorded using camera pills, but rather using endoscopy equipment, which serves

more purposes than just taking videos. Several of the videos therefore contained symbols or annotations added by the recording equipment, which interfered with our classifiers and caused false positives.

We therefore conclude that the need for *border detection* as a preprocessing step depends on the video material being used for training or detection. If any symbols or annotation are present, this step is needed. Instead of storing such information directly in the image, we would suggest the video capsule manufacturers to store any annotations in form of separate metadata, similarly to how subtitles for movies or geotagging information for photography are handled. This would most likely make *border detection* as a preprocessing step redundant.

5.2.2 Edge Detection

Edge detection is one of the most simple and basic detection mechanisms. If the threshold is configured manually, it is usually quite easy to find a setting where objects such as polyps stick out visually. Unfortunately, the needed value for this threshold varies a lot, depending on the surface, the light and the quality of the recording. So this method cannot be used in a generic way for detecting polyps or other irregularities.

If we had a dataset only containing video sequences that were created with the same camera equipment or camera capsule, and the dataset would therefore be more uniform, it might be possible to find a threshold value, which can be used for a first screening step. This could be valuable in a setup, where we would use multiple different classifiers to detect different properties and apply a ranking system on the output of the different classifiers.

The edges that can be detected and made visible using this method, usually contain a lot of noise caused by structure of the mucosa and shades or reflections on its surface. We usually also detect a rather clear edge along any protrusion, caused by the different angle of the surface of a protrusion to the light source. The amount of reflected light is bigger on top of the protrusion than around it, leading to a rather clear edge being detected along a protrusion. We therefore wanted to try using *shape detection* to recognize the protrusions, which are typical for polyps, from the previously detected edges.

5.2.3 Shape Detection

Having *edge detection* in place, it was very straight forward to add *shape detection* as well, as *OpenCV* provides built-in functionality for finding matching shapes. We had to add some basic functionality for selecting shapes (based on edges) and remembering those by storing their points in an appropriate datastructure.

The results of *shape detection* based on *cv::matchShapes* are not very promising. Matching shapes is a well defined operation, and a threshold for the similarity of shapes can be defined. But the shapes generated by *edge detection* are not suitable as input for the *shape detection*, because *edge detection* just detects actual edges and not complete objects. While *edge detection* might make an object clearly visible on screen, such an object is usually a composition of multiple separate edges. If we then use our simple shape matching implementation to learn the shapes of a selected object, we will be comparing separate edges of an object, rather than its whole connected shape. This leads to both many false positives and false negatives, even with very simple objects. Figure 5.6 makes this problem very obvious. Trying to detect stars on the American flag, only the exact same star, as has been learned, is detected. But, at the same time multiple edges of the stripes are detected erroneously.

We conclude that basing *shape detection* on the output of *edge detection* is not usable for our purposes. It can be neither used as a detection mechanism, nor for preprocessing.

5.2.4 Specular Highlight Filtering

Specular highlights are a significant problem with our machine learning approach. The amount of light reflected by protrusions - in particular by polyps - is very strong on most of our samples. It does, however, not make a good characteristic feature, because there is also a plenty of reflections on healthy mucosa. Therefore, a classifier trained with unfiltered data mostly detects reflections rather than actual polyps or protrusions.

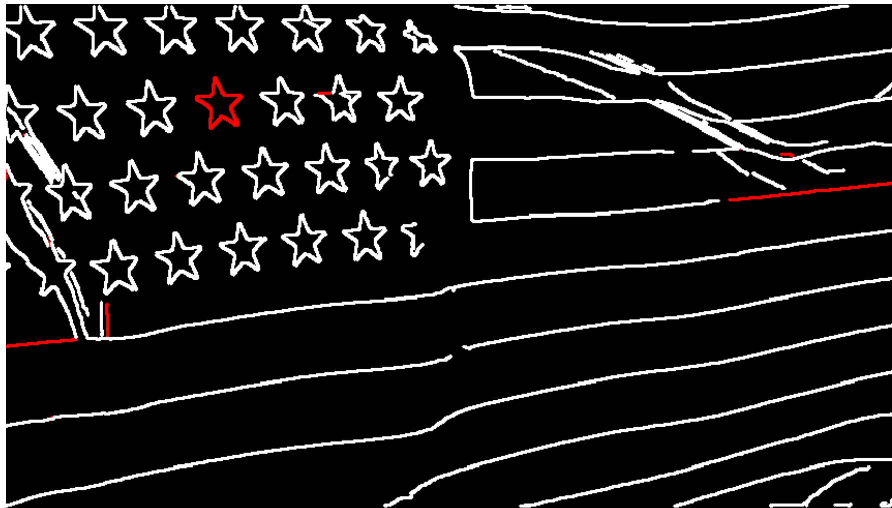
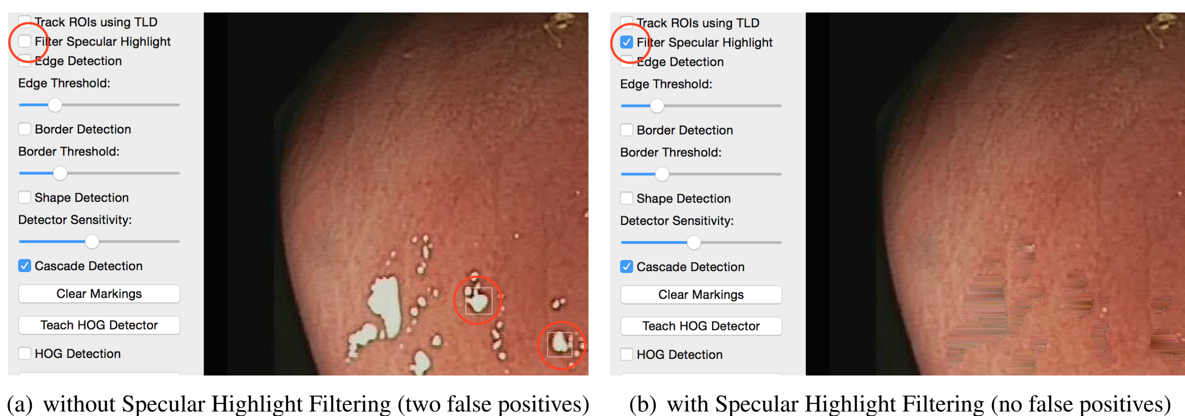


Figure 5.6: Shape detection of stars on the american flag.

We have experimented with Gaussian filtering, the standard solution for specular highlight filtering used in photography. This resulted in a blurry image and thereby loss of valuable texture information. At the same time, the reflections on the mucosa are too strong to be removed or significantly reduced by this kind of filtering.

In many places on the video frames we are using, there is basically no color information left. The reflection causes a completely white spot in the image without any real information. We have therefore developed our own solution for filtering out completely white spots without texture information, removing the surrounding shades and augmenting the original image with randomized gradients. We use these gradients to smoothen the transitions to the surrounding colors of the augmented areas. To make sure that our classifiers do not learn to detect gradients instead of the actual target objects, we randomly distribute the pixels within the gradient lines to reduce any gradient patterns.



(a) without Specular Highlight Filtering (two false positives) (b) with Specular Highlight Filtering (no false positives)

Figure 5.7: Haar-based polyp detection on the same frame, with and without Specular Highlight Filtering.

The noise caused by reflections is significantly reduced in the resulting images, and the classifiers we have trained with this data, detect significantly less false positives caused by reflections (see Figure 5.7). We therefore conclude that this is a valuable preprocessing step for the data used to train a classifier. However, it is likely that specular highlights are less of a problem when data is being recorded with actual camera capsules. All the data we had available was recorded using conventional endoscopes, which have rather strong lights placed next to the camera. Also, during an endoscopy examination, the doctor usually directs the camera and the light source directly at polyps, which leads to even stronger reflections. With camera capsules, the deliberate directing of light and camera would not be the case; also, the light source would most likely be significantly less strong. This is because a camera capsule has a very limited size and can therefore only store a limited amount of energy in its battery. This energy

has to be managed carefully to allow capturing video frames over several hours. The amount of energy available for the light is going to be much lower than with the conventional endoscope.

5.3 Summary

In this chapter, we have discussed the implementation of four different filtering mechanisms for preprocessing and for detecting shapes in video frames. We have also evaluated these filtering mechanisms separately. We have found that *border detection* is useful to filter out artifacts and irrelevant regions along the edges of the video frames. We therefore consider it a valuable preprocessing step for our training and testing datasets.

We have also implemented *edge detection* with a variable edge threshold. We evaluated *edge detection* as a preprocessing step only, as it only affects the visual representation of the image, but cannot be used to detect any lesions or polyps by itself. Video frames processed with this mechanism, can make protrusions such as polyps well recognizable for the human eye. However, the amount and the quality of the edges produced depends heavily on the threshold, which must be adjusted to fit the specific frame or video sequence. Based on the edges detected we have implemented functionality to select, learn and recognize learned edges. We call this mechanism *shape detection*. We considered using *shape detection* to learn and detect the shape of polyps. However, our experiments have shown that this approach is not suitable for our purpose, mostly because the edges produced by *edge detection* for a protrusion are not always equally well connected and, therefore, produce various different shapes.

We have further developed a mechanism for *specular highlight filtering*, which we can use as a preprocessing step for training and testing data in machine learning. This is relevant, because the reflections on the mucosa, caused by the light of an endoscope, are very strong and cause the classifier to learn detecting reflections instead of lesions. This mechanism removes any pixels with an intensity above a certain treshold and replaces those pixels with a randomized gradient to reduce the creation of patterns that could interfere with the classifier training. We consider *specular highlight filtering* a valuable preprocessing step for our further experiments.

After implementing and experimenting with different filtering mechanisms, we next aim for a system that can automatically detect polyps in video sequences. This is investigated by introducing machine learning in the next chapter.

Chapter 6

Machine Learning

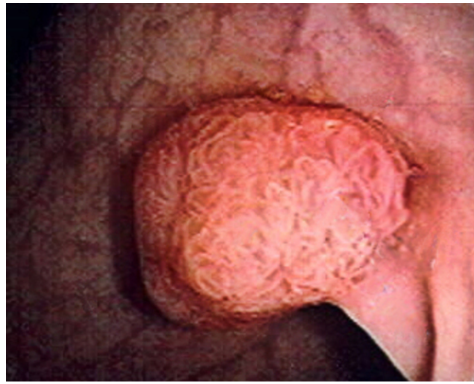
An existing procedure for detecting colon polyps is CTC [55], also called Virtual Colonoscopy [10]. A computer-aided detection approach for CTC imaging data is described in [19]. According to [19], most of the computer-aided detection systems, which have been developed as research prototypes, use algorithmic extraction of the colonic wall to reconstruct a three-dimensional model of the whole colon. The detection thereafter is based on detecting geometric features that characterize polyps. In the case of capsule endoscopy videos, the precise position and direction of the capsule at the moment a picture is taken are not available. So, it is not possible to use the same process of extracting the colonic wall. Instead of recreating a complete three-dimensional model of the colon, we therefore rely on the color and geometry features available on single images. The main idea is to use machine learning to train an algorithm to detect irregularities on an image, in the same way as a human would do manually.

Our goal is to categorize any input images or video frames into two classes: *showing a polyp* and *not showing a polyp*. This is a typical binary classification problem and is therefore suitable for using an SVM. An SVM is a supervised learning model and requires us to collect labeled training data. There is also a concept of unsupervised learning, e.g. clustering, which works on unlabeled data [29]. As we want to base our classifiers on expert knowledge, and considering that the problem is suitable for using a SVM, we have decided to use supervised learning. In [56], the goal of machine learning is described as follows: ”The fundamental goal of machine learning is to generalize beyond the examples in the training set.” To generate this generalization during the training process, a training dataset must contain a sufficient amount of data. The dataset must be diverse and representative enough to avoid overfitting. Overfitting occurs, when the training data being used does not generate an abstract model of the problem we are trying to build a classifier for. In this case, our detector or classifier would not detect a generalization of the problem, but rather encode random noise. An example of how we could cause overfitting is, training an algorithm only based on images of a single polyp. But later, we would like to use the generated classifier for detecting polyps in different videos. In this case, we would most likely end up encoding or learning the exact features of that one single polyp instead of achieving a generalization of all polyps. Such a detector would score very high on the training data, but show very poor performance on any data that has not been part of the training set.

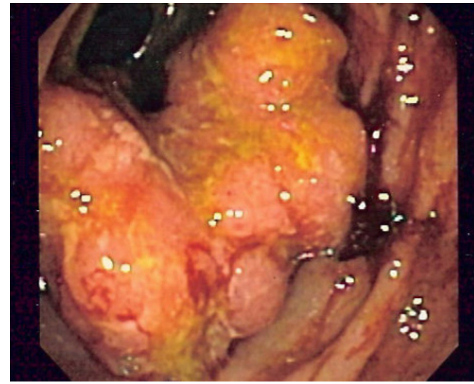
To be able to generalize the features of a polyp or any other lesion, it is therefore crucial to have plenty and diverse training data available. In the case of colon polyps, it is not very easy to collect such data. We neither have the means nor the patients available for this. Neither do we have the expertise to reliably recognize all the disease patterns. So, we have to rely on hospitals to provide us with such data. In the previous chapthers, We have therefore implemented and evaluated several prototypes for collecting such training datasets from doctors or hospitals. In this chapter, we discuss and evaluate our experiments and prototypes for detecting colon polyps using machine learning methods.

6.1 Design and Implementation

As mentioned already before, there are many different kinds of irregularities that can be found and diagnosed using colonoscopy. Examples of such irregularities are presented in Figure 6.1.

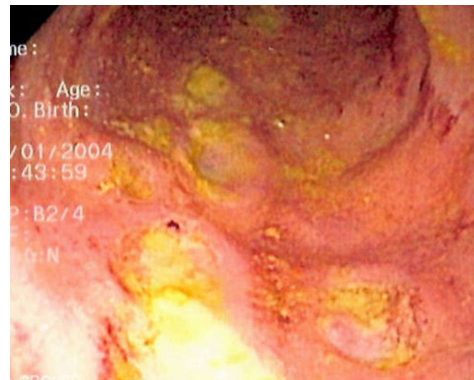


(a) Colon Polyp[1]



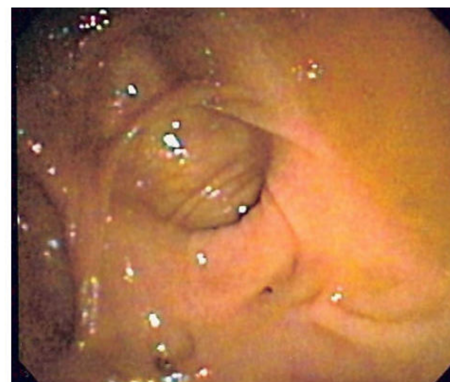
(b) Colorectal Cancer[2]

(c) Ulcerative Colitis[3]

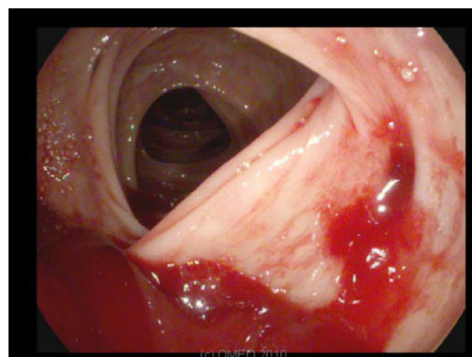


(d) Crohn's Disease[4]

(e) Familial adenomatous polypsis (FAP)[5]



(f) Diverticulosis[6]



(g) Diverticular Bleeding [7]

Figure 6.1: An inconclusive list of irregularities that can be diagnosed using colonoscopy.

As seen in this figure, there are a plenty of different disease patterns, which have very little characteristic features in common, except that they all occur in the colon. While it is not hard for the untrained eye to guess that there is something wrong, it is very hard to actually describe a pattern or even to describe in words what exactly to search for. For simplicity and due to the limited amount of collected data, we will therefore limit the scope for our machine learning process to detecting colon polyps.

Colon polyps are growths of tissue inside the large intestine [55]. Some polyps are mushroom-shaped protrusions on the end of a stalk. Others appear as bumps that lie flat against the intestinal wall. According to another source [57], the word *polyp* derives from the Greek polyposes, "a morbid excrescence", but it now applies to any protrusions from the mucous membrane. To summarize in simple words, we are looking to find any protrusions from the regular wall of the intestine.

A different noninvasive method, which would provide data that is easier to process for this purpose is Virtual Colonoscopy (also known as computed tomography colonography (CTC)). Using Virtual Colonoscopy it is possible to review the recorded images interactively in a two-dimensional or in a three-dimensional format. In the two-dimensional mode, a reviewer screens the dataset in transaxial, coronal and sagittal planes. In three-dimensional mode, the colon is examined from an endoluminal perspective and it can be navigated through the entire length of the colon in both directions, in order to avoid missing polyps, which could be hidden behind a fold of the colon in one or the other direction [10]. Especially the two-dimensional axial images of the Virtual Colonoscopy are very well suited for detecting protrusions. In this view the hollow space of the colon is visible as a black mostly concave area in the image and any convex protrusion into that hollow space is a potential irregularity that needs to be examined more closely.

While this method is noninvasive, well tolerable for the patient and the collected data would even be more suitable for machine processing, it does not solve the problem of scalability. The whole procedure still needs to be conducted by specialists, and expensive equipment is a requirement.

This is why we are instead focusing on imaging data acquired by camera capsules. Of course this leads to a few more challenges. We do not know the exact position of the capsule when a certain picture was taken, neither we know the direction or what it was pointing at. So we are lacking accurate positioning and focusing information. The only information available to us, is the image itself. A common way of recognizing objects on pictures is training a Haar-like feature based cascading classifier. Another method is feature detection using histograms of oriented gradients. Subsequently we will describe the work we have done to train classifiers using these two approaches. This includes descriptions of software we had to write ourselves, as well as configuration and usage of existing tools.

6.1.1 Cascade Classifier Training

Classifier cascading is the concatenation of several classifiers, where the output of each intermediate classifier is used as additional input to the next classifier in the cascade. A face detector implementing this technique has been described in [48] and this is still the basis for Haar-training in *OpenCV*.

OpenCV provides several tools for creating a dataset, training a classifier and even for analyzing the performance of a given classifier. In this section we will describe our efforts for creating a classifier to detect colon polyps and we will also describe the tools we created, to be able to use the previously processed video sequences as input for the training of such a classifier.

[1] CC BY-SA 2.5 / Stephen Holland, M.D., Naperville Gastroenterology, Naperville, IL, USA; http://commons.wikimedia.org/wiki/File:Polyp-2.jpeg

[2] CC BY-SA 3.0 / Jiri Pekhart; http://commons.wikimedia.org/wiki/File:Colorectal_cancer_endo_2.jpg

[3] CC BY-SA 3.0 / Attribution to Samir at English Wikipedia; http://commons.wikimedia.org/wiki/File:Ulcerative_colitis.jpg

[4] CC BY-SA 3.0 / Attribution to Samir at English Wikipedia; http://upload.wikimedia.org/wikipedia/commons/0/0e/CD_colitis.jpg

[5] CC BY-SA 3.0 / Attribution to Samir at English Wikipedia; http://commons.wikimedia.org/wiki/File:FAP.jpg

[6] CC BY-SA 3.0 / Attribution to Samir at English Wikipedia; http://commons.wikimedia.org/wiki/File:Diverticulosis_2.jpg

[7] Copyright by Brugge / endoatlas.org; http://www.endoatlas.org/assets/media/img/xl/weo_colon_diverticulum_active_bleeding_brugge.jpg

For training the classifier, we need a dataset as input to *opencv_traincascade*. Such a dataset consists of a vector file that contains all our positive samples and a selection of images for negative samples. To create a vector file with positive samples, we can use the tool *opencv_createsamples*. This tool is very convenient to use when creating a classifier for a rigid and well-defined object - for example, a company logo. A company logo is a very simple case, as corporate identity usually dictates the logo always to look exactly the same. So, in this case, the classifier only needs to detect exactly one specific object. Of course, this object can be rotated, skewed or have a perspective distortion on any picture we might feed to the classifier. But, it will still remain a single object with precisely defined characteristic features. *opencv_createsamples* makes it very easy to create a vector file containing positive samples for this specific case. All we need is a single image showing the logo. The tool can then create as many samples as we want by applying rotations and distortions. This can be achieved with the following command:

Listing 6.1: Generate samples from a single image.

```
1  opencv_createsamples -img logo.png -num 100 -bg negatives.dat -vec samples.vec -maxxangle 0.5 \
2      -maxyangle 0.4 -maxzangle 0.3 -bgcolor 0 -bgthresh 0 -w 20 -h 20
```

The command presented in Listing 6.1 would generate a vector file *samples.vec* containing *100* samples, generated from a single image *logo.png*, using background images listed in the file *negatives.dat*. The distortions applied to the original image would be limited by a maximum rotation of *0.5 rad* around the x-axis, *0.4 rad* around y-axis and *0.3 rad* around z-axis. The background color of *0* is treated as transparency, which equates to black in grayscale, and the threshold for the background color will be *0*. This threshold means that any completely black pixels in our sample image would simply be ignored in the later training of the classifier. This would work great, for a simple logo on a black background. Unfortunately, our case is slightly more complicated than the simple case with a single logo. In our case, we are not trying to recognize a single well defined logo, but rather a whole class of objects, each of them slightly different from any other object.

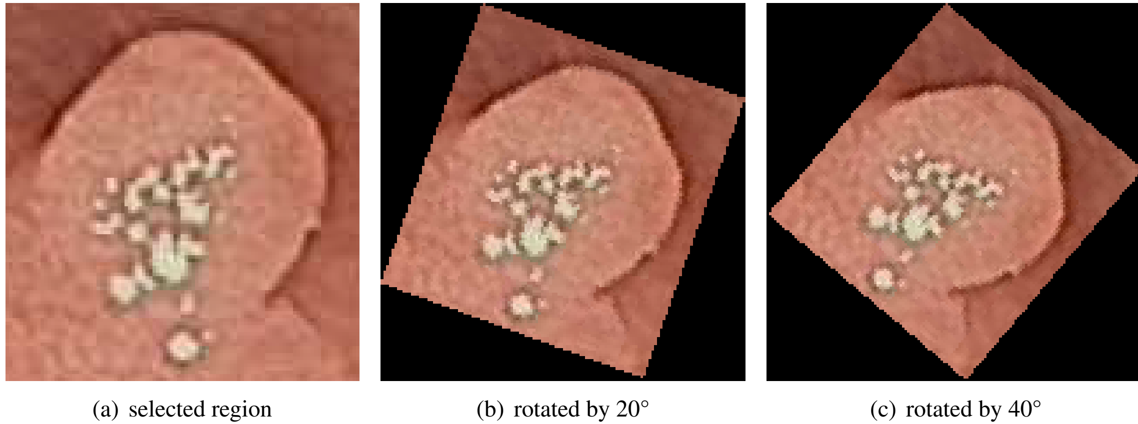
6.1.1.1 Exporting Positive Samples from TagAndTrack

As we have described previously, *opencv_creatsamples* can be used to create many samples from a single image by applying rotations and distortions. It is, however, not capable of doing the same with multiple input images. So we either have to invoke that tool for every single image and merge the resulting vector files, or do the rotations and distortions by ourselves, and just use *opencv_createsamples* to actually bundle a list of images into a vector file. We therefore implemented an export mechanism in our *TagAndTrack* tool. After using *TagAndTrack* to create a complete dataset by object tracking and applying manual corrections, the user can choose the menu option called *Export Samples*. The tool will then ask the user to select a target folder to export all the information to. All regions that have been marked as positive samples will be rotated clockwise in steps of 20 degrees, resulting in a total of 18 samples per region. Each of these samples will then be cropped to the bounding box of the rotated region and will be stored as a separate image. Every image will be named according to the pattern described in Listing 6.2. The pattern encodes the exact time of starting the export task in the image name. This will allow us to easily merge multiple folders containing exported results without renaming any files.

Listing 6.2: Naming pattern for positive samples.

```
1  p_[rotation angle]_sample_[date & time of exporting]_[frame number].png
2
3  An example for a positive sample from frame 1262, rotated by 40 degree exported on the 2nd of December 2014 at 22:32.34:
4  p_40_sample_141202223234_1262.png
```

If we would simply create rotated copies of a ROI, this would look like shown in Figure 6.2. Of course, the most characteristic feature in such a series of samples would then rather be the rotated rectangle, than the actual content of the selected region. To avoid this problem, we therefore have to calculate the correct bounding box of the rotated selection and then crop the sample image accordingly. The dimensions for the region to store in an image after rotating are calculated using the algorithm described in Listing 6.3. We have added inline comments for additional explanations.



(a) selected region (b) rotated by 20° (c) rotated by 40°

Figure 6.2: Uncorrected rotation of ROI.

Listing 6.3: Calculation of region to export after rotating.

```
static void saveSamplesForRect(const QImage &image, const QRectF& rect, const QString sampleName
    , QDir& targetDir, QTextStream &positive)
{
    // Calculate the dimensions needed to accommodate all rotations for the selection.
    int w = rect.width();
    int h = rect.height();
    float r_selection = sqrt(pow(w, 2) + pow(h, 2)) / 2;
    float r_boundingBox = sqrt(pow(2 * r_selection, 2) * 2) / 2;
    int x1 = rect.x() + w / 2 - r_boundingBox;
    int y1 = rect.y() + h / 2 - r_boundingBox;

    // Create a copy of the part of the image needed to accommodate all rotations.
    QImage rotatable = image.copy(x1, y1, 2 * r_boundingBox, 2 * r_boundingBox);
    QRectF selectionRect((rotatable.width() - w) / 2, (rotatable.height() - h) / 2, w, h);

    // Calculate and store all required rotations in incremental steps of 20 degrees.
    for (int i = 0; i < 360; i+=20) {
        QString fileName = sampleName.left(2) + QString::number(i) + "_" + sampleName.mid(2);
        QMatrix trueMatrix = QImage::trueMatrix(QMatrix().rotate(i), rotatable.width(), rotatable.height());
        QImage rotated = rotatable.transformed(trueMatrix);

        QRectF exportRect = trueMatrix.mapRect(selectionRect);
        // Crop the rotated image to the bounding box of the rotated selection and save it with best quality (100).
        QImage cropped = rotated.copy(exportRect.toRect());
        cropped.save(targetDir.absoluteFilePath(fileName), "PNG", 100);
        // Save the metadata to the text stream for positive samples.
        positive << fileName << " " << 1 << " " << 0 << " " << 0 << " " << cropped.width() << " " << cropped.height() << "\n";
    }
}
```

Figure 6.3 visualizes the calculation steps from Listing 6.3. The figure shows an example video frame with a clockwise rotation by 40 degrees for exporting. The green rectangle is the originally selected ROI *rect*. The green dotted circle is defined by *r_selection* and is used as an intermediate step to calculate the radial dimensions of the rotated bounding box. Using the resulting *r_boundingBox* (blue dotted circle) we can then calculate the maximum dimensions we need to accommodate the bounding box for any rotation of the image. For simplicity, we then create a new image *rotatable*, which is a copy of the necessary fraction of the original image (red rectangle). Finally, the blue rectangle *exportRect* is the fraction of the image, which eventually will be stored as a separate sample.

6.1.1.2 Exporting Negative Samples

The *opencv_traincascade* tool requires a set of images for negative / background samples. These images should show regular content that could appear in the background of a picture passed to the detector later in the process. In theory, we could use any image that has not specifically been marked to contain a ROI for this purpose. However, there are several reasons why we would not want to make use of every single non-marked frame in a video sequence. Some parts of a video sequence might, for example, show an irregularity, but be out of focus, which is why we did not mark any regions on these frames. Another reason could be visibility of equipment, used during the colonoscopy to take samples from the mucosa. And, of course, videos might also have parts in the beginning and towards the end, which we are not interested in.

Further, we also had to add functionality for selecting frames to be used as negative samples to our *TagAndTrack* tool. For simplicity, we have decided only to allow selection of whole frames and not

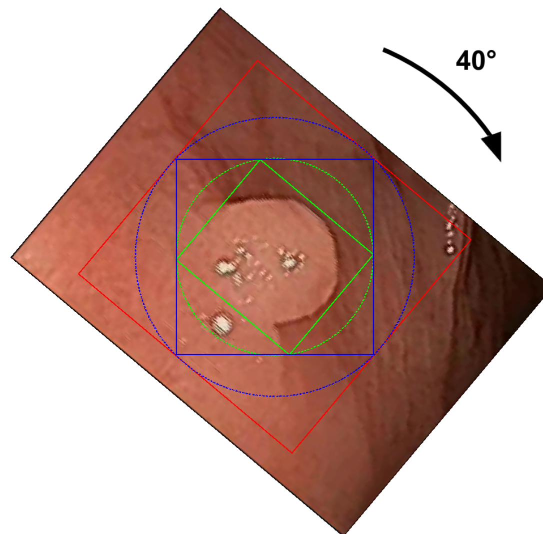


Figure 6.3: Calculated geometries for roatating and exporting positive samples.

sub-areas as negative samples. For this purpose, we added an indicator, which changes color depending on a frame being a positive or a negative sample. By default, we treat any frame like a positive sample, and the indicator is colored red. During playback or while stepping forwards or backwards using the *left-* or *right-key*, a user can press the *Ctrl- / Command-key* to explicitly mark the frame stepping away from as a negative sample. When stepping or seeking back to a previously marked frame, the indicator will then be colored green. To remove the negative marking from a previously marked frame, the same action can be used, but with pressing the *Shift-key* instead of the *Ctrl- / Command-key*. When using the export mechanism, all frames marked as negative samples will be stored as separate images, un-cropped and in full resolution. Every image will be named according to the pattern presented in Listing 6.4.

Listing 6.4: Naming pattern for negative samples.

```
n_sample_[date & time of exporting]_[frame number].png

An example for a negative sample originating from frame 1289, exported on the 2nd of December 2014 at 22:32.34:
n_sample_141202223234_1289.png
```

6.1.1.3 Exporting Metadata for the Samples

In addition to the actual image frames, we also need to export metadata for the pictures to be usable for *opencv_createsamples* and *opencv_traincascade*. When exporting the images, we therefore also create two text files in the same directory. The file containing the meta data for the positive samples we call *positive.dat*, and the file containing the data for negative samples we call *negative.dat*. The *positive.dat* file contains a single line for every positive sample. Every line has the format presented in Listing 6.5.

Listing 6.5: Format of positive.dat.

```
[filename] [number of objects] [[x y width height] [... object 2 ...] ...]
```

As we export a separate image for every single sample, we will only have a single object per line. Since all the samples are also cropped and rotated, the position of the object in the sample always starts at the top left corner, and the width and height of the object are equal to the width and height of the image itself. In our case, the *positive.dat* file will therefore always be similar to Listing 6.6.

Listing 6.6: Example of positive.dat

```
p_0_sample_141202150457_874.png 1 0 0 131 126
p_20_sample_141202150457_874.png 1 0 0 166 163
p_40_sample_141202150457_874.png 1 0 0 181 181
p_60_sample_141202150457_874.png 1 0 0 175 176
p_80_sample_141202150457_874.png 1 0 0 147 151
...
```

The file *negative.dat* contains a line stating the file name of every image to be used as a negative sample. When exporting data for several different video sequences, all of the exported images should have distinct names. If we want to train a classifier with the output for multiple video sequences, we have to copy all the images into a single folder. Further, we have to concatenate all the *negative.dat* files and also concatenate all the *positive.dat* files. As we are potentially dealing with thousands of files, merging directories can be difficult. A simple *cp* in a bash shell might not work because of a too long list of arguments. This can, however, be solved by making use of *xargs*, as shown in Listing 6.7.

Listing 6.7: Using xargs to work around command argument limitation.

```
1 ls video_samples/ | xargs -I{} -n 1 cp video_samples/{} ~/my_dataset/
```

6.1.1.4 Using opencv_traincascade to Train a Classifier

Before we can actually start training a classifier, we still need to create a vector file from the previously exported positive samples. For this purpose, we can use *opencv_createsamples* with the *positive.dat* file as input, as presented in Listing 6.8.

Listing 6.8: Creating a vector file from raw images and positive.dat.

```
1 opencv_createsamples -info positive.dat -vec positive_samples.vec -w 24 -h 24 -num `cat positive.dat | wc -l`
```

This will create a file called *positive_samples.vec*, which contains all the sample images stated in *positive.dat*. All the samples will be scaled to $24x24$ pixels. We also have to pass the number of lines we want to process from the input file. There is no documentation for *OpenCV* that describes what size the positive samples should be scaled to. Some tutorials[8] suggest using sizes around $20x20$ pixels for face recognition. Of course, the scale also affects the level of detail preserved in the samples. The bigger the sample, the more details are available. At the same time, this comes at the cost of a significantly longer time for training the classifier. We have decided to use the default value of $24x24$ pixels to make sure we can detect a reasonable level of detail.

The next step is to train the classifier. For this purpose, *OpenCV* provides a tool called *opencv_traincascade*. We will use the command sequence in Listing 6.9 for the training. The *opencv_traincascade* tool requires us to pass the number of positive samples to be used for the training in each stage with the parameter *-numPos*. We should not just pass the number of available positive samples here, because *opencv_traincascade* will skip samples, which do not have additional training effect for subsequent stages. We would like all our positive samples to be used, but we cannot know the amount of discarded positive images in advance. If we require more images to be used than available after discarding some, the training process will abort. A good rule of thumb is to use 90% of the available samples. Before starting the training process, we therefore shuffle the lines in the *positve.dat* files. As *opencv_traincascade* will process the samples in the same sequence they are mentioned in *positive.dat*, this will make it much less likely that we will miss a complete original sample in the training process. Of course, we will still discard the same amount of samples, but these will most likely be rotations of different samples instead of all the rotations of a single sample.

Listing 6.9: Training the classifier with opencv_traincascade.

```
1  # Randomize the order of the positive samples.
2  gshuf positive.dat > shuffled_positive.dat
3
4  # Determine the amount of positive samples to use per stage.
5  NUMPOS=`cat shuffled_positive.dat | wc -l`
6  NUMPOS=`echo "($NUMPOS * 0.9)/1" | bc`
7
8  # Determine the amount of negative samples to use per stage.
9  NUMNEG=`cat negative.dat | wc -l`
10
11 # Start the training.
12 opencv_traincascade -data output_traincascade -vec positive_samples.vec -bg negative.dat -precalcValBufSize 2048 \
13     -precalcIdxBufSize 2048 -numPos $NUMPOS -numNeg $NUMNEG -w 24 -h 24 -mode ALL
```

[8] http://note.sonots.com/SciSoftware/haartraining.html#Kuranov

The command sequence in Listing 6.9 will start the training process and write any intermediate and final output data into the directory *output_traincascade*. We increase the buffer sizes for precalculated features and indices to 2048 MB each, to make sure the process has enough memory available. Further, we specify to use all available Haar-features for the training. The meaning of all the arguments is described in Listing 6.10.

Listing 6.10: Description of arguments to opencv_traincascade

```
1  -data <output directory>
2  -vec <positive samples vector file>
3  -bg <negative samples dat file>
4  -precalcValBufSize <buffer size for the precalculated features (MB)>
5  -precalcIdxBufSize <buffer size for precalc. feature indices (MB)>
6  -numPos <number of positive samples to be used in each stage>
7  -numNeg <number of negative samples to be used in each stage>
8  -w <width of the positive samples in the vector file>
9  -h <height of the positive samples in the vector file>
10 -mode <set of Haar features to use for training>
```

The training process runs for several days, depending on the required accuracy and the available input data. We were running a training process with a total of $14,472$ positive samples and $10,842$ negative samples. With the settings described previously, this process took about 9 days to finish on a Mac Pro Mid 2010 with dual 2,4 GHz Quad-Core Intel Xeon CPUs, 16 GB of memory and an SSD drive. However, the specifications of the computer are not that significant since for the biggest part of the training process only a single CPU is used. On Linux, it is possible to build *OpenCV* with *Intel Threading Building Blocks (TBB)*[9] support. *TBB* is a C++ template library for task parallelism. It is available on multiple platforms, but *OpenCV* does not make use of it on Mas OS X. Using a Linux machine with a custom built *OpenCV* version to support *TBB*, we observed a significant speed increase. With a computer of similar hardware specification, we could then complete the training in about 4 days.

6.1.1.5 Building an OpenCV based Detector Tool

The output of *opencv_traincascade* is an XML file, which contains the trained classifier. The detector tool processes an input video, classifying every single video frame using the classifier loaded from such an XML file. To be able to concatenate several such tools to a processing pipeline, it should be possible to run the detector in a head-less mode, not requiring a user to operate a user interface. This will also make it very simple to run the detector tool on multiple servers, to process many different video sequences simultaneously. We therefore made this a simple command line tool, without a graphical user interface.

The basic functionality of such a detector is rather straight forward, and is described in Listing 6.11. As in previous listings, the code was simplified and boilerplate code, such as include statements or parsing of command line arguments, has been removed. The boolean *detect* is passed as command line argument *-c* and the cascading classifier will only be used if this option is passed. This allows us to run the tool just for measuring the time needed for reading a video file without any actual processing. We can use this reading-time as a baseline for benchmarking.

We implemented a class *RectMap*, which is essentially a wrapper for a simple *std::vector* holding *cv::Rect* instances. *RectMap* implicitly allocates this vector to be of equal size as the current video's lenght in frames. At a later point in time, this allows us to simply assign *cv::Rect* instances belonging to a specific video frame to a given position in the vector, without having to resize the vector beforehand. We use a *cv::CascadeClassifier* instance to load the classifier data from a file called "cascade.xml" in the same directory. We also use a *cv::VideoCapture* instance, to read the video file frame by frame. Before we start reading any frames, we store a timestamp so we can calculate the processing duration before exiting the program. We then read the video file frame by frame, convert the image to grayscale, equalize the grayscale histogram to improve the contrast of the image and use the *detectMultiScale* function of our classifier, to run the actual detection at multiple different scales. The result of this function call, we insert into our *RectMap*. Once all the frames have been processed, the *RectMap* contains all detection information, and the datastructure could be printed or processed further. For now, we are only interested in the number of regions detected and the duration of processing a video sequence. We therefore acquire a second timestamp, calculate the time difference and output this information on the command line.

[9] `https://www.threadingbuildingblocks.org/`

Listing 6.11: A simple OpenCV based detector.

```
<<< include several headers >>>
using namespace std;

class RectMap {
public:
    RectMap(size_t s) : rectangles(s, std::vector<cv::Rect>()) { }

    void insert(int frame, const std::vector<cv::Rect> &rects) {
        rectangles[frame] = rects;
    }

    int totalNumRects() const {
        int num = 0;
        for (auto it = rectangles.begin(); it != rectangles.end(); ++it) num += it->size();
        return num;
    }

private:
    std::vector<std::vector<cv::Rect>> rectangles;
};

int main(int argc, char* argv[]) {
    <<< check command line arguments >>>
    std::string video_file; // Passed on command line.
    bool detect; // Passed on command line as argument -c.
    cv::CascadeClassifier classifier;
    if (!classifier.load("cascade.xml")) { return -1; }

    cv::VideoCapture capture(video_file);
    double frame_count = capture.get(CV_CAP_PROP_FRAME_COUNT);
    RectMap result(frame_count); // Datastructure for storing ROIs.

    chrono::steady_clock::time_point start = chrono::steady_clock::now();

    cv::Mat image;
    int framesRead = 0;
    while (capture.read(image)) {
      if (!detect) continue;
        static cv::Mat gray;
        cv::cvtColor(image, gray, CV_BGR2GRAY);
        cv::equalizeHist(gray, gray); // Use histogram equalization to improve the contrast.

        std::vector<cv::Rect> detectedRegions;
        classifier.detectMultiScale(gray, detectedRegions);
        result.insert(framesRead++, detectedRegions);
    }

    chrono::steady_clock::time_point end = chrono::steady_clock::now();

    cout << "Regions detected: " << result.totalNumRects() << endl;
    std::chrono::milliseconds ms = chrono::duration_cast<chrono::milliseconds>(end - start);
    std::chrono::seconds sec = chrono::duration_cast<chrono::seconds>(ms);
    ms -= sec;
    cout << "time:" << sec.count() << "." << ms.count() << " sec." << endl;
    return 0;
}
```

As we do not have a separate user interface for the detector, we will need another way for a specialist to look at any detected irregularities, to do a complete diagnose. But, in fact, we already have two tools which can be used exactly for this purpose. If we store the information about detected regions in the same JSON format, as described in Listing 3.1, we can read it with either the web-based tagging tool, or with the *TagAndTrack* software, which we have developed for tracking. For this purpose, we also need to implement a function that can write the *RectMap* data structure to a file in JSON format. The code for this is rather straight forward, and will not be described here in more detail.

For testing the performance of our solutions, we use a video sequence with a duration of 81 seconds and a total of $2,025$ frames. The resolution of the video is $768x576$ pixels. For all the testing, we use the same MacBook Pro Retina 15-inch Late 2013 with a 2.6 GHz Intel Core i7, 16 GB of memory and an NVIDIA GeForce GT 750M graphics card with 2,048 MB of graphics memory. The classifier we use was trained with the same video sequence and it must detect a total of 583 regions in that specific video. The detection in this case contains both false negatives and false positives, but this is irrelevant for the purpose of measuring the time needed for processing the video sequence. To goal is to be able to process a video sequence at its normal playback speed. For this, we need to achieve a frame rate above 25 FPS.

The most simple implementation, as described in Listing 6.11, processes the whole video in 62.403 seconds (32 FPS). If we are not passing the *-c* switch on the command line, the program will only read and decode all the video frames but not run any detection. Without running the detection, the video is processed in 15.278 seconds (132 FPS). The most obvious optimization for this implementation is to separate the reading and decoding of video frames into a separate thread.

6.1.1.5.1 Use a Separate Thread for Decoding

Using a separate thread for reading and decoding of video frames will allow running the detection on a video frame at the same time as the next frame is being read and decoded. Based on the previous results, we would expect to reduce the processing time for the same video sequence by about 15 seconds, which is the amount of time spent on reading and decoding video frames only. The changes we needed to implement this behavior are presented in Listing 6.12. Implementation details which have been described previously, such as *RectMap* or the time measurement, have been removed from the listing intentionally.

Listing 6.12: An OpenCV based detector with a single worker thread.

```
<<< include several headers >>>
using namespace std;

<<< class definition for RectMap >>>

class Frame {
public:
    Frame(int frameNumber, const cv::Mat &img) : id(frameNumber) {
        img.copyTo(image); // Take a copy of the image, as the buffer is being reused.
    }

    std::vector<cv::Rect> process(cv::CascadeClassifier& classifier) {
        cv::Mat gray;
        cv::cvtColor(image, gray, CV_BGR2GRAY);
        cv::equalizeHist(gray, gray);

        std::vector<cv::Rect> detectedRegions;
        classifier.detectMultiScale(gray, detectedRegions);
        return detectedRegions;
    }

    int id;
    cv::Mat image;
};

Frame* framesBuffer[2]; // Buffer for two frame pointers.
std::mutex mtxFrameBuffer; // Mutex for protecting the buffer.
volatile int indexFrameBuffer; // Buffer index for next frame.
volatile bool doneReading = false;

void processFrames(RectMap *r) {
    cv::CascadeClassifier cascade;
    <<< load cascade classifier from xml file >>>

    while (true) {
        Frame *frame = 0;
        mtxFrameBuffer.lock();
        std::swap(frame, framesBuffer[indexFrameBuffer]);
        mtxFrameBuffer.unlock();
        if (frame) {
            std::vector<cv::Rect> rects = frame->process(cascade);
            r->insert(frame->id, rects);
            delete frame;
            frame = 0;
        } else if (doneReading) return;
    }
}

int main(int argc, char* argv[]) {
    <<< check command line arguments >>>

    cv::VideoCapture capture(video_file);
    double frame_count = capture.get(CV_CAP_PROP_FRAME_COUNT);
    RectMap result(frame_count); // Datastructure for storing ROIs.

    indexFrameBuffer = 0;
    framesBuffer[0] = 0;
    framesBuffer[1] = 0;
    int framesRead = 0;
    cv::Mat image;
    std::thread detectorThread(processFrames, &result);

    <<< take start timestamp >>>

    while (capture.read(image)) {
        if (run_cascade) {
            Frame *frame = new Frame(framesRead, image);
            while (true) {
                std::unique_lock<std::mutex> lck(mtxFrameBuffer);
                Frame *buffer = framesBuffer[indexFrameBuffer];
                if (buffer == 0) {
                    framesBuffer[indexFrameBuffer] = frame;
                    indexFrameBuffer ^= 0x1; // Switch to other buffer 0 -> 1 or 1 -> 0.
                    break;
                }
            }
        }
        ++framesRead;
    }
    doneReading = true;
    detectorThread.join();

    <<< calculate time difference >>>
    return 0;
}
```

The most significant change is that we added a class *Frame*, which represents a single video frame and also includes a method *process* that can run the detection on the given frame. Further, we introduced a buffer that can hold two frames, so that we can alternate between these two frames for updating the content and processing it. This essentially is the concept of double buffering. We use an *std::thread* for running the *processFrames* function. This function then checks for a frame being available in the buffer to be processed, calls the *Frame::process* function on such a frame and deletes the frame after processing. We use a minimal critical section in the function *processFrames* to secure the *std::swap* call. This call will swap a frame pointer out of the buffer and replace it with a null pointer. If the swapped out pointer is non-null, we have successfully swapped a frame out of the buffer and can process it outside of the critical section.

With the improvement of reading and decoding the frames on the main thread, but using a separate thread for running the detection, we repeated the previous experiment and we achieved a slightly better result. The total processing time decreases to about 50 seconds (40 FPS). This is an improvement of about 20% and that matches our expectation. However, on our test machine, the CPU usage still only reaches about 40% when running the detector. So, it seems that our CPU is idle about 60% of the time, and this idle time we obviously want to reduce as much as possible. After all, this implies that, taking synchronization overhead into account, we could in theory reduce the processing time again by more than 50%. We therefore implemented a solution that allows us to use multiple threads for running the detection on multiple separate video frames concurrently.

6.1.1.5.2 Introduce Multiple Threads

Considering the changes we made to our detector for running with a single worker thread, we only have to change a few minor parts to allow running an arbitrary number of worker threads. The *Frame* class is already well suited for usage with multiple threads. We need to implement a queue data structure *FrameQueue*, which is protected by a mutex and can therefore be accessed from different threads. This data structure basically replaces the *framesBuffer* from the single worker thread implementation. In the *FrameQueue*, we then store a pointer to each frame that was read and decoded by the main thread. Further, we need to add a *processQueue* function, which we use as an entry point for the spawned threads. In this function, we just spin a loop that queries the queue for new frames. Whenever a new frame is available in the queue, a worker thread will dequeue the frame, call the *Frame::process* function on it, store the detected regions in *RectMap* and eventually delete the frame. Once the queue is empty and the main thread signaled that it finished reading, the main thread will wait for all worker threads to join and display the results to the user. As the class *CascadeClassifier* is not designed to be used by multiple threads, we have to instantiate one classifier per thread. The implementation for this is presented in Listing 6.13. This implementation allows us to run the detector with several different configurations. We can choose to only read and decode, but not run any detection at all, we can choose to run decoding and detection sequentially in a single thread, or we can choose to spawn an arbitrary amount of worker threads for the detection. The processing times achieved with this implementation using different configurations are presented in Figure 6.4 and Table 6.1. We achieved the best result using 4 worker threads. With this configuration, we can process the whole video sequence in about 28.3 seconds (71 FPS).

The CPU usage in Table 6.1 was measured using *Activity Monitor* of *Mac OSX* on the same MacBook Pro we used to conduct all the other measurements. This computer does have four physical cores, which do have support for hyper threading. Therefore, it is theoretically possible to achieve 800% CPU usage. However, as we can see from the table, the fastest processing time is achieved with a total of four worker threads. Adding further threads will show a higher CPU usage, but the processing time will not decrease anymore. This is because all four physical cores are working at almost 100% and using hyper threading to switch in-between threads that are part of the same job does not speed things up any further. The optimal amount of threads should therefore be the amount of physical cores plus a single thread for reading and decoding the video frames.

Listing 6.13: An OpenCV based detector with multiple worker threads.

```
<<< include several headers >>>
<<< implementation of class Frame >>>
<<< implementation of class RectMap >>>

class FrameQueue {
public:
    void enqueue(Frame *frame) {
        q_mutex.lock();
        q.push(frame);
        q_mutex.unlock();
    }

    Frame* dequeue() {
        q_mutex.lock();
        Frame *f = 0;
        if (!q.empty()) {
            f = q.front();
            q.pop();
        }
        q_mutex.unlock();
        return f;
    }

    bool empty() const {
        q_mutex.lock();
        bool e = q.empty();
        q_mutex.unlock();
        return e;
    }

private:
    std::queue<Frame*> q;
    mutable std::mutex q_mutex;
};

bool doneReading = false;
void processQueue(FrameQueue *q, RectMap *r) {
    cv::CascadeClassifier cascade;
    <<< load classifier from xml file >>>

    while (true) {
        Frame *frame = q->dequeue();
        if (frame) {
            std::vector<cv::Rect> rects = frame->process(cascade);
            r->insert(frame->id, rects);
            delete frame;
            frame = 0;
        }
        if (doneReading && q->empty()) return;
    }
}

int main(int argc, char* argv[]) {
    std::string video_file; // Provided on commandline.
    bool run_cascade; // Provided on commandline.
    int num_threads; // Provided on commandline.

    <<< check & parse command line arguments >>>
    <<< open video_file with cv::VideoCapture object >>>

    FrameQueue framesQueue; // Create a queue datastructure for the frames.
    RectMap r(frame_count); // Datastructure for results.

    // Create as many worker threads as requested.
    std::vector<std::thread*> threads;
    for (int i = 0; i < num_threads; ++i) threads.push_back(new std::thread(processQueue, &framesQueue, &r));

    <<< take start timestamp >>>

    cv::CascadeClassifier cascade; // Only used with 0 worker threads.
    <<< if 0 worker threads: load classifier from xml file >>>

    cv::Mat image;
    int framesRead = 0;
    while (capture.read(image)) {
        if (run_cascade) {
            Frame *frame = new Frame(framesRead, image);
            if (num_threads > 0) framesQueue.enqueue(frame);
            else r.insert(frame->id, frame->process(cascade));
        }
        ++framesRead;
    }
    doneReading = true;

    for (auto t = threads.begin(); t != threads.end(); ++t) (*t)->join(); // Join all worker threads.

    <<< calculate time difference >>>
    <<< print results >>>
    return 0;
}
```

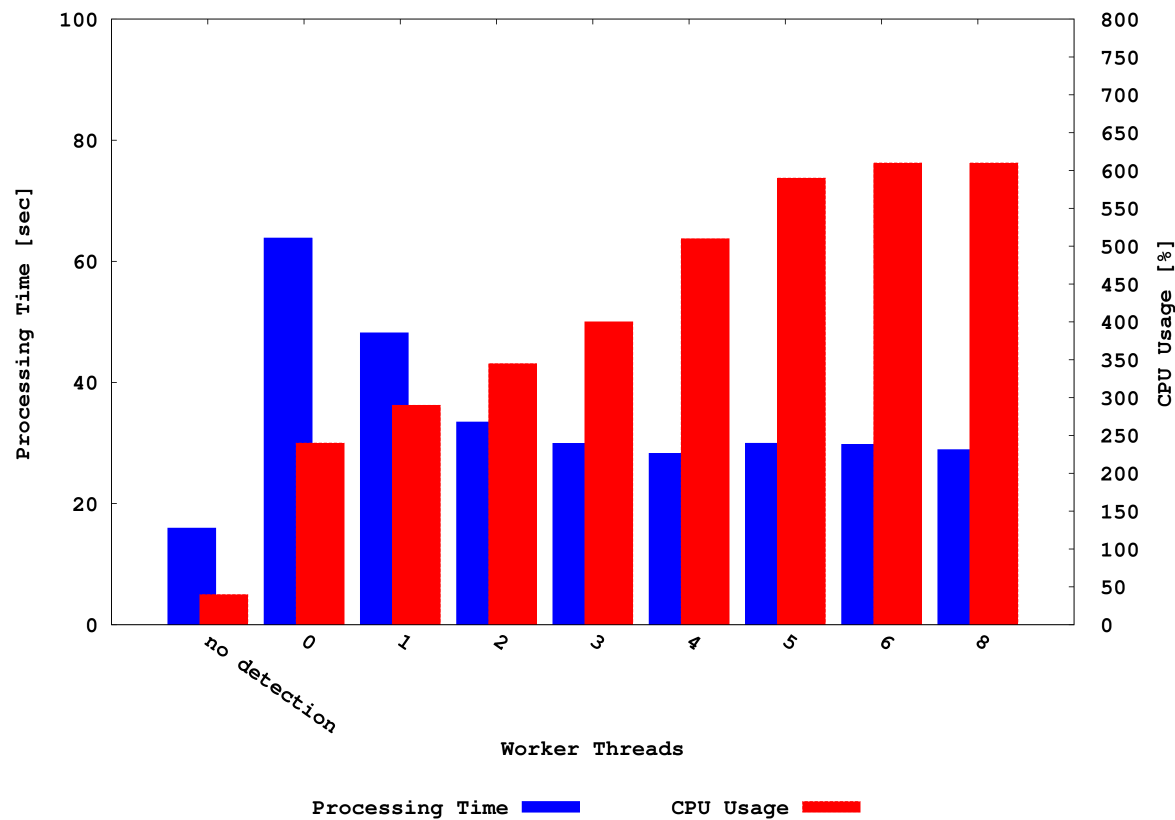


Figure 6.4: Processing Times for Multi Threaded Cascade Classifier Detection, processing 2,025 frames at a resolution of 768x576.

Table 6.1: Processing Times for Multi Threaded Cascade Classifier Detection, processing 2,025 frames at a resolution of 768x576.

Worker Threads	Processing Time [s]	CPU usage [%]
no detection	15.97	40
0	63.87	240
1	48.21	290
2	33.48	345
3	29.96	400
4	28.31	510
5	29.98	590
6	29.78	610
8	28.92	610

Using *Instruments*, the profiling tool provided with *XCode*, it is possible to easily analyze the runtime behavior of the detector. For this purpose we are again running the detector with the optimal amount of four worker threads. The interesting part of the output generated by *instrument* is presented in Listing 6.14.

Listing 6.14: Profiling detector running with 4 threads.

```
1  Running Time        Self    Symbol Name
2  25706.0ms   61.7%    0,0    thread_start
3  25706.0ms   61.7%    0,0      _pthread_start
4  25706.0ms   61.7%    0,0        _pthread_body
5  25649.0ms   61.5%    0,0          void* std::__1::__thread_proxy<std::...
6  25649.0ms   61.5%    0,0            processQueue(FrameQueue*, RectMap*)
7  25646.0ms   61.5%    0,0              Frame::process(cv::CascadeClassi...
8  24942.0ms   59.8%    0,0                cv::...::detectMultiScale(cv::...
9    400.0ms    0.9%    2,0                cv::equalizeHist(cv::_InputArr...
10   304.0ms    0.7%    0,0                cv::cvtColor(cv::_InputArray c...
11     3.0ms    0.0%    0,0              Frame::~Frame()
12    57.0ms    0.1%    0,0          listenOnNotificationPort
13 13913.0ms   33.4%  134,0    start_wqthread
14 13775.0ms   33.0%    1,0      _pthread_wqthread
15 13774.0ms   33.0%    1,0        _dispatch_worker_thread3 <libdispatch...
16     4.0ms    0.0%    0,0      <Unknown Address>
17  2024.0ms    4.8%    0,0    start
18  2024.0ms    4.8%    0,0      main
19  1936.0ms    4.6%    0,0        cv::VideoCapture::read(cv::Mat&)
20    87.0ms    0.2%    0,0        Frame::Frame(int, cv::Mat const&)
21     1.0ms    0.0%    0,0        operator new(unsigned long)
22     1.0ms    0.0%    1,0    _pthread_wqthread
23     1.0ms    0.0%    1,0    objc_opt::objc_stringhash_t::hash(char con...
```

The profiling results show that we are spending about 61.5% of the running time in the *Frame::process* function. Looking at the source code, we see that the function contains just a few lines of code and, most importantly, it calls the *CascadeClassifier::detectMultiScale* function, which is where we expect to spend most of the time. The profiling results verify this by showing that we are indeed spending 59.8% out of the 61.5% of the time within *detectMultiScale*. The remaining running time is consumed for the biggest part by a function called *_dispatch_worker_thread3* of a library called *libdispatch*. This library is an interfacing library to the *Grand Central Dispatch (GCD)* of *Mac OS X*, and is used by *OpenCV* for concurrent code execution. The use of *GCD* with *OpenCV* for *Mac OS X* is implicit and not configurable without modifying the source code. So, without modifying *OpenCV*, this part is out of our control. The last 4.8% of running time is consumed by *VideoCapture::read* for reading video frames from the input file, which is a necessity as well.

All in all, the results seem to be fairly optimal with this configuration. Further optimizations could potentially be done within *OpenCV* code or by changing the configuration of *OpenCV* at compile time. It is also possible that using *Intel Threading Building Blocks* instead of *GCD* on a *Linux* machine would lead to different results.

6.1.2 Histogram of Oriented Gradients Detector

The second machine learning method we look into, is a HOG [47] based object detector. For this purpose we use the *dlib*[10] C++ Library, which provides a HOG trainer and a structural SVM solver. HOG based object detection is known in particular for requiring a low amount of samples to build a detector of reasonably good quality. Further, the time needed to train the detector is expected to be much lower than, for example, the time needed to train a cascading classifier as described earlier. The most common use case for a HOG detector is face detection. An example of this use case is implemented and provided together with sample pictures with *dlib*.

The approach of implementing a detector with *dlib* is slightly different from the approach used with *OpenCV*. While *OpenCV* provides a separate tool for training a classifier, *dlib* only provides an API for implementing such a tool. For simple testing purpose, we therefore decided to add *dlib*-training directly into our existing *TagAndTrack* tool.

6.1.2.1 Adding HOG-training to TagAndTrack

The advantage of adding *dlib*-based HOG-training directly to *TagAndTrack* is that we can reuse a lot of functionality we have implemented previously. We can adopt the whole handling, reading and playback

[10] http://dlib.net/

of video files as well as the selecting of regions of interest. We can also avoid reading/writing any intermediate format such as XML or JSON for the single purpose of training the detector. We therefore added a function *trainHOGDetector* to our class *CVWorker*, which is triggered by pressing a button of the user interface of *TagAndTrack*. We assume that when this function is called, the user has previously loaded a video into the application and has marked some regions of interests that we can then use to train the detector. The code for the function *trainHOGDetector* is described in Listing 6.15. We have added inline comments for additional explanations.

Listing 6.15: Implementation of HOG training in TagAndTrack.

```
// Create an image pyramide data type for downsampling by 3/4.
typedef dlib::scan_fhog_pyramid<dlib::pyramid_down<4> > scanner_t;

void CVWorker::trainHOGDetector()
{
    // Any image in training_images will have its rectangles stored in training_rectangles at the same index.
    dlib::array<dlib::array2d<unsigned char>> training_images;
    std::vector<std::vector<dlib::rectangle>> training_rectangles;

    // Create a map that contains all the markings per video frame.
    std::map<int, std::vector<const Marking*>> &mpf = resolveMarkingsPerFrame(m_cvWidget->markings());

    // Iterate through all the frames in the map. Those are all the frames where regions of interest were selected.
    for (auto it = mpf.begin() ; it != mpf.end(); ++it) {
        cv::Mat* img = seek(it->first); // Seek to the required video frame.

        static cv::Mat gray; // Convert the video frame to grayscale
        cv::cvtColor(*img, gray, cv::COLOR_BGR2GRAY);

        // Copy the raw data of the video frame into a dlib compliant container for video frames.
        dlib::array2d<unsigned char> dlib_img;
        dlib::assign_image(dlib_img, dlib::cv_image<unsigned char>(gray));
        training_images.push_back(dlib_img);

        // Convert the regions of interest into a vector of rectangles, that is usable for dlib.
        auto markings = it->second;
        std::vector<dlib::rectangle> rectangles;
        for (auto it = markings.begin(); it != markings.end(); ++it) {
            const Marking* marking = *it;
            int l = marking->x();
            int t = marking->y();
            int r = l + marking->width();
            int b = t + marking->height();
            rectangles.push_back(dlib::rectangle(l, t, r, b));
        }
        training_rectangles.push_back(rectangles);
    }

    // Add left right flips for symmetrical training.
    dlib::add_image_left_right_flips(training_images, training_rectangles);

    scanner_t scanner; // Setting the size of the sliding window to 80x80 pixels.
    scanner.set_detection_window_size(80, 80);

    dlib::structural_object_detection_trainer<scanner_t> trainer(scanner);
    trainer.set_num_threads(4); // Use 4 threads for training.
    trainer.set_epsilon(0.01); // Stop training when epsilon <= 0.01

    m_hogDetector = trainer.train(training_images, training_rectangles); // Start the actual training process.
}
```

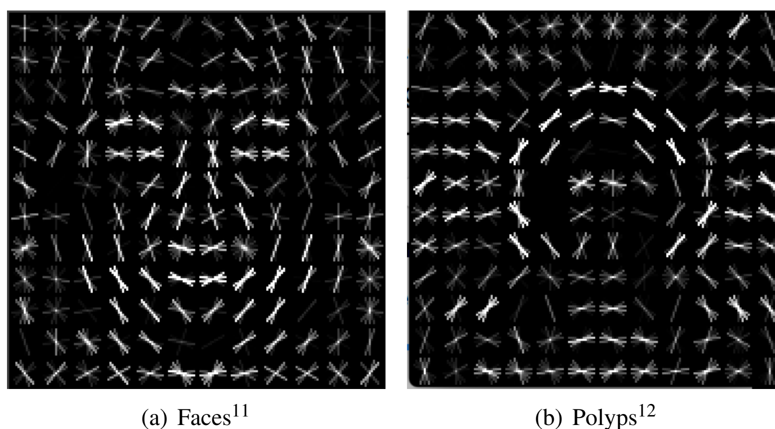
The sliding window size of $80x80$ pixels defines the minimum size of an object to be detected. It also defines the aspect ratio and therefore the shape of the objects we do want to detect. Any selected ROI should have an aspect ratio that is similar to the dimensions defined for the sliding window. If this is not the case, the training will fail and the user will be notified. The sliding window needs a certain size in order to create meaningful results, because multiple pixel values are needed in a cell to create a gradient. A very interesting point is the ability to use multiple threads for the training. This possibility significantly speeds up the training process. As we use a machine with 4 CPU cores, we create 4 threads. The training stops as soon as the epsilon value, in our case 0.01, was reached.

6.1.2.2 Visualizing the training result for a HOG detector

A very interesting feature provided by *dlib* is the ability to visualize an existing HOG detector. The code for this conversion and for displaying the result on screen is shown and explained in Listing 6.16. A HOG detector uses cells; it counts the occurrences of gradient orientation in each of these cells for every sample. We can display these resulting gradients as an image. This is in particularly interesting, because it allows us to analyze the detector not just by looking at the images it succeeds or fails to detect, but also to understand why it might fail to detect some images. We can then draw conclusions on how to adjust the settings for a detector.

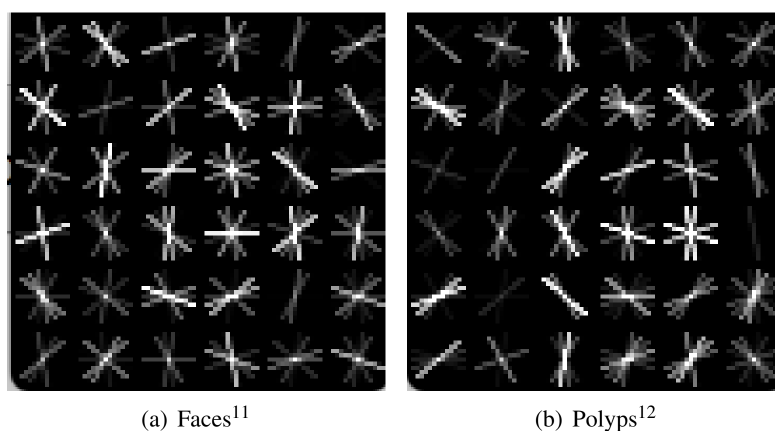
If we take a look at the visualizations in Figure 6.5, we can immediately see, that the image in Figure 6.5(a) shows the visualization of a detector that was trained with faces. The gradients form an oval shape of a face, with the eyes, nose and mouth clearly visible. The training of the detector has basically reduced the training images to the most characteristic features of a face.

The image in Figure 6.5(b) shows the visualization of a detector that was trained with images of a polyp. And, in that case, we can see the typical characteristic features of a polyp. In this case, those are the simple round shape of a protrusion, the connection to the mucosa (towards the bottom of the image in this case), and the specular highlight in the center of the polyp, caused by the light source pointing at it.



(a) Faces[11] (b) Polyps[12]

Figure 6.5: Visualizations of HOG detectors for faces and for polyps with a window size of 80x80.



(a) Faces[11] (b) Polyps[12]

Figure 6.6: Visualizations of HOG detectors for faces and for polyps with a window size of 30x30.

Visualizing HOG detectors also gives us the opportunity to examine a bit more close, how to choose the right size for the sliding window. According to the *dlib* documentation[13], the sliding window defines the minmum size of object that we will be able to detect after the training. So, the obvious choice would therefore be, to make the sliding window very small, to allow detecting small objects as well. However, if we change the size of the sliding window from $80x80$ to $50x50$ or $30x30$ pixels, we will immediately see that the quality of the detection gets significantly worse. In our experiment, the amount of false positives remains stable, but the amount of false negatives increases drastically. For this experiment, we have used a single video sequence. We have marked the same polyp on 10 subsequent video frames of a single video sequence and have then trained the HOG detector with these samples. We have used the

[11]The faces HOG detector was trained with the example pictures provided with dlib.

[12]The polyp HOG detector was trained with 358 pictures from a single video sequence showing the same polyp.

[13]`ftp://ftp.nist.gov/pub/mel/michalos/Software/Optimization/dlib-18.9/docs/dlib/image_processing/scan_fhog_pyramid_abstract.h.html`

trained detector to detect the same polyp on 100 subsequent video frames. On all of these 100 video frames, the polyp is visible exactly once. The numbers for this experiment are listed in Table 6.2.

Table 6.2: Performance of a simple HOG detector with different window sizes.

Window Size (pixels)	Correctly Detected	False Positives	False Negatives
80x80	79	0	21
50x50	29	0	71
30x30	0	0	100

The reason for the detecting performance decreasing with the sliding window size becomes rather obvious when looking at the visualization of the $30x30$ detectors in Figure 6.6. Again, those detectors have been trained with the same data as the ones in Figure 6.5, just this time the sliding window was smaller. As we have described previously and as the name itself is saying, histogram of oriented gradients detection works by counting the occurrences of gradient orientation in sub-portions or cells of an image. Those cells are required to have at least a certain minimal size to allow calculating a gradient and an orientation. If the sliding window is too small, there will only be very few cells and the gradients may not give an accurate enough representation of the object we are trying to detect. Comparing the visualizations of $80x80$ detectors with the ones of the $30x30$ detectors makes this problem very obvious. While the object to detect is clearly recognizable in the $80x80$ visualizations, it is literally impossible to recognize anything in the $30x30$ visualizations. We have therefore decided to use a sliding window size of $80x80$ pixels for our purpose.

Listing 6.16: Visualizing a HOG detector.

```
1  void CVWorker::visualizeHOGDetector()
2  {
3      // Extract the image information from the detector.
4      dlib::matrix<unsigned char> img = draw_fhog(m_hogDetector);
5
6      // Create a grayscale QImage instance matching the size of the visualization.
7      QImage qimg(img.nc(), img.nr(), QImage::Format_Indexed8);
8
9      // Initialize the 8-bit grayscale color table.
10     QVector<QRgb> table( 256 );
11     for( int i = 0; i < 256; ++i )
12         table[i] = qRgb( i, i, i );
13     qimg.setColorTable(table);
14
15     // Copy all the pixel values into the QImage instance.
16     for (int x = 0; x < img.nc(); ++x) {
17         for (int y = 0; y < img.nr(); ++y) {
18             qimg.setPixel(x, y, img(y, x));
19         }
20     }
21
22     // Display the resulting image in a top level window (QLabel).
23     static QLabel *hogDisplay = new QLabel;
24     hogDisplay->setGeometry(100,100, qimg.width(), qimg.height());
25     hogDisplay->setPixmap(QPixmap::fromImage(qimg));
26     hogDisplay->show();
27 }
```

Our experiments with the described solution of building HOG-detection into *TagAndTrack* lead to fairly promising results. The results will be described in more detail in Section 6.2.2. The obvious problem with this approach is that we can only train a detector with the data of a single video. We therefore needed to develop a separate tool that we can use for training HOG-based detectors. The *dlib* library provides convenience functions for serializing and deserializing fully trained detectors. We can make use of these functions to load a detector that was trained by a separate program back into *TagAndTrack* to evaluate the quality of the trained detector.

6.1.2.3 Implementing a separate HOG-Trainer

So far, we have used *TagAndTrack* to train basic HOG-classifiers. However, *TagAndTrack* is designed to process a single video file at a time. We therefore want to develop a separate HOG-Trainer, which can train a classifier using ground truth, exported from *TagAndTrack*. The *dlib* library provides a function

load_image_dataset to load a dataset of images and rectangles for training from an XML input file. For convenience, we want to make use of this function. However, as we are specifically implementing a separate HOG-Trainer for training a detector with input data from multiple video sequences, we have to add a bit more logic to deal with multiple input files. The process of creating the XML files will be described in the subsequent Section 6.1.2.4. In Listing 6.17, we present the basic implementation of the separate HOG-Trainer, we have implemented. We have removed a lot of boilerplate code, such as exception handling and including of headers, to make it more readable and concise. We also removed pieces of code, which have already been discussed in Listing 6.15, such as instantiating the image scanner and training objects.

Listing 6.17: HOGTrainer implementation

```
1   <<< include various header files >>>
2   using namespace std;
3   using namespace dlib;
4   typedef std::vector<std::vector<rectangle>> Rectangles;
5   typedef dlib::array<array2d<rgb_pixel>> Images;
6
7   int main(int argc, char** argv) {
8       <<< Verify the amount of arguments passed. >>>
9       <<< Exception handling is left out intentionally. >>>
10      Images tr_images;
11      Rectangles tr_rects;
12
13      // Import the data from every XML file, and merge it into tr_images and tr_rects.
14      for (int i = 1; i < argc; ++i) {
15          Images images;
16          Rectangles rects;
17          load_image_dataset(images, rects, argv[i]);
18          for (auto it = images.begin(); it != images.end(); ++it)
19              tr_images.push_back(*it);
20          tr_rects.insert(tr_rects.end(), rects.begin(), rects.end());
21      }
22
23      // Add left/right flips of every training image (mirroring).
24      add_image_left_right_flips(tr_images, tr_rects);
25
26      // Add rotations for every training image in steps of 40 degrees.
27      matrix<double,8,1> angles;
28      double step = 40.0 * M_PI / 180.0;
29      for (long r = 0; r < angles.nr(); ++r) angles(r) = r * step;
30      add_image_rotations(angles, tr_images, tr_rects);
31      cout << "training images: " << tr_images.size() << endl;
32
33      <<< Create and initialize image scanner and trainer objects. >>>
34      <<< This has already been described in a previous chapter. >>>
35
36      // Filter out unobtainable rectangles and notify the user.
37      int n_d = 0; // number of discarded rectangles.
38      Rectangles d = remove_unobtainable_rectangles(trainer, tr_images, tr_rects);
39      for (auto it = d.begin(); it != d.end(); ++it) n_d += it->size();
40      cout << "discarding " << n_d << " rectangles." << endl;
41
42      // Start the training process. Make sure to pass the discarded rectangles!
43      object_detector<image_scanner_type> detector = trainer.train(tr_images, tr_rectangles, d);
44
45      cout << "storing detector to: detector.svm" << endl;
46      serialize("detector.svm") << detector;
47      return 0;
48  }
```

We add rotations of every positive training sample to allow detection of an object in any possible orientation. *dlib* provides us with a very convenient function *add_image_rotations* exactly for this purpose. This pre-calculates all the image rotations as well as the rotations for the provided rectangles, and all this data is kept in memory. Even with a fairly big amount of memory (16GB), we were running out of memory very quickly. As *dlib* is designed in a way that requires all training data to be loaded into memory before starting the training, and it does not allow for incremental training either, this just leaves us the option to reduce the number of training samples used. Because of this, we have reduced the number of rotations to 8 images per sample with increments of 40 degrees each.

Our *TagAndTrack* tool does not have any restrictions to the size or the proportions of the selected rectangles. *dlib*, on the other hand, requires us to define a window size for the detector. And, as previously described, we have decided to use a window size of 80x80 pixels. Any selected rectangles, which *dlib* considers not to be close enough to an aspect ratio of 1, can therefore not be detected. If we still pass such an impossible rectangle to the trainer, *dlib* would throw an exception, and the training process would need to be restarted without the offending rectangle. In order to avoid this, we can filter out any impossible rectangles using the *remove_unobtainable_rectangles* function. This function will remove any offending rectangle and return a matching datastructure containing offending rectangles only. We have to make sure that we pass this datastructure as the third argument to the *train* function, to make sure

that the rejected image region is not used as a negative sample during the training process.

The serialized HOG-classifier can simply be deserialized with API provided by *dlib* , as shown in Listing 6.18. Of course, the datatype for *m_hogDetector* must match the datatype that was used when the classifier was exported.

Listing 6.18: Deserialize HOG detector.

```
void CVWorker::openHOGSVMFile(const QString hogSvmFileName) {
    dlib::deserialize(hogSvmFileName.toStdString()) >> m_hogDetector;
}
```

The *HOGTrainer* implementation expects a list of XML files passed as command line arguments. The format of such an XML file is defined by *dlib* and is described in Listing 6.19. Using this format, it is possible to define multiple regions (box tags) within a single image. The contained data is basically the same as we are exporting for the use with *opencv_traincascade*, but the format is slightly different. It would be possible to build a translator to translate between the two formats, but it is more straight forward to add support for exporting data in the required XML format directly into *TagAndTrack*.

Listing 6.19: Format of an XML file used as input to HOGTrainer.

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<?xml-stylesheet type='text/xsl' href='image_metadata_stylesheet.xsl'?>
<dataset>
<name>150125141605</name>
<images>
    <image file='p_image_150125141605_1237.png'>
        <box top='95' left='334' width='108' height='180'/>
        ... more boxes ...
    </image>
    ... more images ...
</images>
</dataset>
```

6.1.2.4 Exporting data from TagAndTrack to HOGTrainer

We want to use *TagAndTrack* to create several ground truth datasets consisting of images and selections that mark polyps in these images. This ground truth we want to export in the predefined XML format, so it can be used as input data for training a classifier with *HOGTrainer*. A usual way for writing generic XML data is implementing an in-memory representation of the XML Document Object Model (DOM), which can be serialized and written to a file in plain text. This is a very generic approach that can deal with literally any kind of data. However, our case does not require a generic implementation, and assuming that the XML format will not become more complex over time, we can save a lot of tedious work for implementing a DOM representation. We know that the header of every file will start with the same two lines, defining the XML-version, the encoding and the XML-stylesheet. This will always be followed by a *dataset* opening-tag, a *name*-tag, and an *images* opening-tag. We can then simply iterate through all the images we have positive regions of interests for. For each image, we append an *image* opening tag, followed by as many *box*-tags needed for that image, and eventually write an *image* closing-tag for it as well. Once we finished iterating through the available images, we just need to append *images* and *dataset* closing-tags. In addition to the XML data, we also have to export the image files containing regions of interest. The algorithm for all this is described in more detail in Listing 6.20. We also added a button to the user interface of *TagAndTrack* to trigger the export action. Clicking this button will then bring up a dialog, where the user can choose a directory to write the exported data to. To make sure the data is kept separate, we only allow for empty folders to be selected. During the export process we pop-up a dialog with a progress bar and a cancel button, in order for the UI to remain responsive and allow the user to abort the export process.

Listing 6.20: Exporting tagging and tracking information for HOGTrainer.

```
void CVWorker::exportHOGXML(const QString &dirName, const QString &uid) {
    if (isPlaying()) stop(); // Stop any running playback.

    // Show a modal progress dialog.
    QProgressDialog progress("Creating Samples...", "Abort", 0, 100, m_cvWidget->window());
    progress.setWindowModality(Qt::WindowModal);

    // Create a map where the key is the frame number and the
    // value is a vector containing all ROIs for the corresponding frame.
    std::map<int, std::vector<const Marking*>> &markingsPerFrame = resolveMarkingsPerFrame(m_cvWidget->markings());

    // Create and open the XML file for writing.
    QFile xmlfile(QDir(dirName).absoluteFilePath(QLatin1String("training_") + uid + QLatin1String(".xml")));
    if (!xmlfile.open(QIODevice::WriteOnly | QIODevice::Text)) {
        std::cerr << "could not open training_"
            << uid.toStdString() << ".dat" << std::endl;
        return;
    }

    // Create a text stream to the XML file, and write the header data.
    QTextStream str(&xmlfile);
    str << "<?xml version='1.0' encoding='ISO-8859-1'?>\n";
    str << "<?xml-stylesheet type='text/xsl' href='image_metadata_stylesheet.xsl'?>\n";
    str << "<dataset>\n<name>" << uid << "</name>\n<images>\n";

    // Iterate through all the available frames for this video.
    int frameCount = m_capture->get(CV_CAP_PROP_FRAME_COUNT);
    for (int frameNumber = 1; frameNumber < frameCount; ++frameNumber) {
        <<< Skip the frame if there are no ROIs. >>>
        <<< Seek to frame and save it in PNG format to the export dir. >>>

        auto m = markingsPerFrame[frameNumber];
        bool imageTagWritten = false;

        // Iterate through all markings for this frame.
        for (auto marking = m.begin(); marking != m.end(); ++marking) {
            QRect rect = (*marking)->rectForFrame(frameNumber).toRect();
            if (!rect.isNull()) {
                if (!imageTagWritten) { // If there is a marking, write an image opening-tag.
                    str << "    <image file='" << sampleName << "'>\n";
                    imageTagWritten = true;
                }
                // Write the rectangle information to the stream.
                str << "        <box top='" << rect.top()
                    << "' left='" << rect.left()
                    << "' width='" << rect.width()
                    << "' height='" << rect.height()
                    << "'/>\n";
            }
        }

        // Update the progress bar and check if the user pressed cancel.
        progress.setValue(100.0 / frameCount * frameNumber);
        if (progress.wasCanceled()) return;

        // If an image opening-tag was written, write a closing-tag.
        if (imageTagWritten) str << "    </image>" << "\n";
    }

    // Write closing-tags for the document.
    str << "</images>\n</dataset>\n";
}
```

6.1.3 Index of Global Image Features

So far, we have looked at machine learning approaches to precisely locate a lesion in a given image. However, our goal is not detecting the precise location, but rather the presence of a lesion. All the images that we process can therefore be separated into two disjoint sets. The first set contains all the images showing a lesion, and the other set contains all the images without any lesion. As we do not necessarily need to detect the precise location of a lesion, we can also consider global image features. *LIRe* [58] is an open source library for content based image retrieval, written in *Java*. It provides a comprehensive set of algorithms to extract different types of global image features. Using *LIRe* therefore allows us to experiment with a whole set of global image features to find out if we can use any of these for classifying or clustering video frames from colonoscopy video sequences. *LIRe* uses *Lucene*[14] indices for storing and searching image feature data. *Lucene* indices are structured in documents, fields and terms. An index contains a sequence of documents, where a document is a sequence of fields, a field is a sequence of terms and a term is a string [59]. A sketch of the structure and the basic elements of a *Lucene* index is presented in Figure 6.7.

Our basic idea is to create an index of as many colonoscopy images as possible. The index should also contain the information whether a certain image does contain a polyp or not. A classifier can then

[14] https://lucene.apache.org/

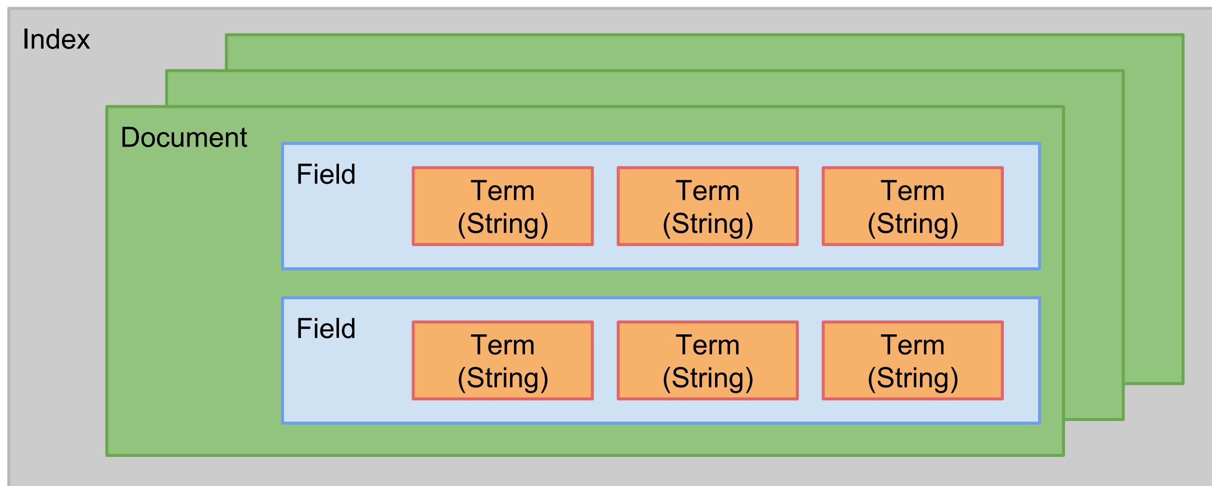


Figure 6.7: The structure and basic elements of a lucene index.

search the index for the images that are most similar to a given input image. Based on the classification of the results, we can then decide which cluster the input image belongs to. For this approach, we implemented two separate tools, an *Indexer* and a *Classifier* (see Figure 6.8). As *LIRe* and *Lucene* are both implemented in *Java*, we have implemented these tools in *Java* as well. The design of these tools is described in the subsequent paragraphs. We have released the *Indexer* and the *Classifier* as a separate project called *OpenSea*[15], under the terms of the GNU General Public License version 3[16].

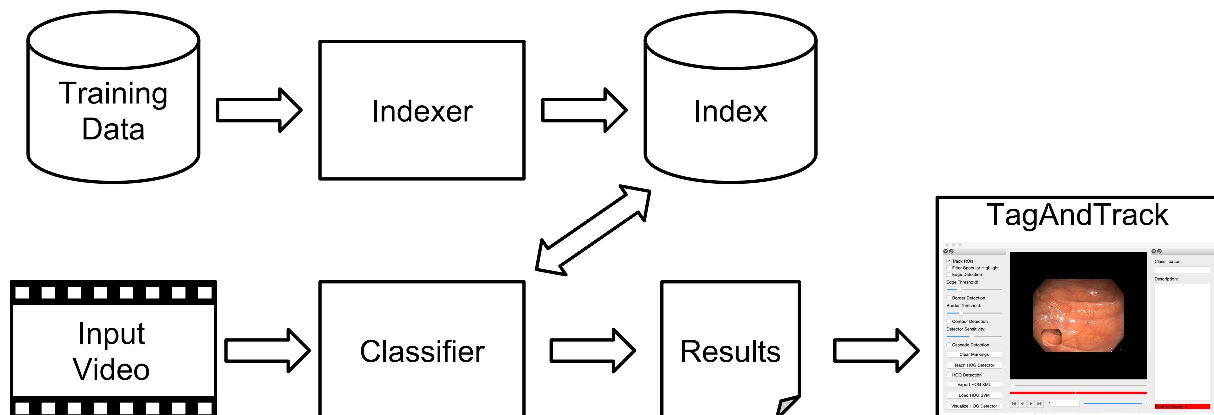


Figure 6.8: The overall architecture of our global image features based approach, consisting of Indexer and Classifier as separate tools.

6.1.3.1 Global Image Feature Indexer

The purpose of the global image feature indexer is to extract image features from input images and store these in a *Lucene* index. Such indices can then be used as input data for the global image features classifier. We have created the indexer as a separate tool, which can be started from the command line tool. It then creates indices for all directories passed on the command line. The image features to calculate and store in the indices can be passed as separate arguments. The indexer indexes all the images in a given directory. It stores the generated index in a subdirectory *index* inside the indexed directory. If multiple directories are passed for indexing, it creates a separate index for each directory. The exact usage of the indexer is presented in Listing 6.22 and the implementation of the *Indexer* is presented in detail in Listing 6.21. We have added inline comments for additional explanations. To make the code

[15] `https://bitbucket.org/mpg_projects/opensea`
[16] `http://www.gnu.org/licenses/gpl-3.0.en.html`

more readable, we have removed a lot of exception handling, reduced the class structure and stripped out a lot of insignificant boilerplate code and error handling. We assume that the machine we use does have hyper threading and we therefore only use as many threads as the cores reported by the Java Virtual Machine (JVM). Unfortunately, there is no guarantee that the JVM reports a correct number of physical cores. In future, it might therefore be worth considering to use a command line option for this purpose as well.

Listing 6.21: Implementation of the Indexer for global image features.

```
1  package no.simula.indexer;
2  <<< Import various Java, Lire and Lucene classes. >>>
3
4  public class Indexer {
5    static ExecutorService pool = null;
6    static IndexWriter indexWriter = null;
7
8    public static void main(String[] args) {
9      <<< Verify and process command line arguments. >>>
10     ArrayList<String> featureNames; // Provided on command line.
11     ArrayList<String> directories; // Provided on command line.
12     if (featureNames.isEmpty()) featureNames.add("JCD"); // Fall back to JCD, if no feature names were passed.
13
14     // Force the class loader to load these classes at startup, to prevent
15     // initialization race when using ImageIO with multiple threads.
16     Class.forName("javax.imageio.ImageIO");
17     Class.forName("java.awt.color.ICC_ColorSpace");
18     Class.forName("sun.java2d.cmm.lcms.LCMS");
19
20     for (String samplePath : directories) {
21       // List all the images in the directory.
22       List<String> images = Collections.synchronizedList(FileUtils.getAllImages(new File(samplePath), true));
23       DocumentBuilder builder = getDocBuilder(featureNames);
24
25       String iPath = samplePath + "/index";
26       <<< Delete any already existing index. >>>
27       // Create a new index writer for the given directory.
28       IndexWriterConfig conf = new IndexWriterConfig(LuceneUtils.LUCENE_VERSION, new WhitespaceAnalyzer(LuceneUtils.
            LUCENE_VERSION));
29       indexWriter = new IndexWriter(FSDirectory.open(new File(iPath)), conf);
30
31       // Create a ThreadPool with (cores / 2) number of threads.
32       int numCores = Runtime.getRuntime().availableProcessors() / 2;
33       if (numCores < 1) numCores = 1;
34       final int numThreads = numCores; // numThreads must be final. We access it from within a lambda.
35       pool = Executors.newFixedThreadPool(numThreads);
36
37       <<< Take start timestamp. >>>
38       final int numImages = images.size();
39       <<< Create a Runnable for each thread. >>>
40       for (int rId = 0; rId < numThreads; ++rId) {
41         final int runnableId = rId; // Accessed inside lambda.
42         Runnable r = () -> {
43           // Each Runnable starts processing the images list at the index position of its own runnableId
44           // and continues processing in offsets of numThreads. This implicitly partitions the work.
45           for (int i = runnableId; i < numImages; i += numThreads) {
46             String imgPath = images.get(i);
47             BufferedImage img = ImageIO.read(new FileInputStream(imgPath));
48             // Create the actual index entry (document).
49             Document document = builder.createDocument(img, imgPath);
50             protectedAddDocument(document); // Add entry to common index.
51             showProgress(i, numImages);
52           }
53         };
54         pool.execute(r); // Start the execution of the threads.
55       }
56       pool.shutdown(); // Wait for all threads to complete.
57       if (!pool.awaitTermination(365, TimeUnit.DAYS));
58       <<< Take end timestamp and print duration. >>>
59       indexWriter.close();
60     }
61   }
62
63   public static DocumentBuilder getDocBuilder(ArrayList<String> features) {
64     // Create a DocumentBuilder which chains multiple different DocumentBuilders, each for a specific image feature.
65     ChainedDocumentBuilder builder = new ChainedDocumentBuilder();
66     for (String fName : features) {
67       // Instantiate document builder by class name.
68       String cName = "net.semanticmetadata.lire.imageanalysis." + fName;
69       Class<? extends LireFeature> c = (Class<? extends LireFeature>) Class.forName(cName);
70       builder.addBuilder(new GenericDocumentBuilder(c, true));
71     }
72     return builder;
73   }
74
75   // This method is synchronized to ensure mutual exclution, when adding a document to the indexWriter.
76   private static synchronized void protectedAddDocument(Document document) { indexWriter.addDocument(document); }
77
78   private static int currentProgress = 0;
79   private synchronized static void showProgress(int index, int numDocs) {
80     // Each thread reports the processed index number. We only allow incrementing the progress.
81     if (index < currentProgress) return;
82     currentProgress = index;
83     <<< Print progress to System.out. >>>
84   }
85 }
```

Listing 6.22: Usage of the Indexer for global image features.

```
java -jar indexer.jar /dir/with/images [/dir/with/more/images ...] [-f ...]
      All the provided paths will be indexed and a separate index will
      be stored in a subdirectory inside each provided directory.

      -f | -feature   A feature to use for classification. JCD is default.
                      Multiple features can be provided. Possible features
                      are for example: JCD, FCTH, EdgeHistogram, CEDD,
                      ColorLayout, LocalBinaryPatternsAndOpponent,
                      FuzzyOpponentHistogram, FuzzyColorHistogram,
                      Gabor, JointHistogram, JpegCoefficientHistogram,
                      OpponentHistogram, PHOG, RankAndOpponent,
                      RotationInvariantLocalBinaryPatterns,
                      ScalableColor, SimpleColorHistogram, Tamura.
```

6.1.3.2 Global Image Feature Classifier

We have implemented a search based classifier for global image features. The classifier can be used to classify video frames from an input video into two groups: positive (containing a polyp) and negative (not containing a polyp). The classifier uses indices generated by the indexer described in the previous section. In contrast to the other classifiers that we have built, this classifier is not trained in a separate learning step. Instead, the classifier searches previously generated indices for similar image features, weights the search results and decides which cluster an input image most likely belongs to. We refer to these previously generated indices, which are searched for similar image features, as *classifier indices* or *indices containing training data*. The classifier expects at least one classifier index and one input source as a command line argument. The input source can either be a video sequence or another previously generated index. If an index is used as an input source for classifying, the classifier will output benchmarking information (Figure 6.9 / bottom) and an HTML page with a visual representation (Figure 6.10) of the results, once the processing is finished. For the classifier to provide correct benchmarking data, the input data indices must contain either negative or positive samples only, or must have the sample type encoded in the file names of the indexed images. If a video sequence is used as an input data source, no benchmarking data can be collected, and no HTML file is produced. Instead, a JSON file is generated, which contains a lists of *positive* and a list of *negative* frames. This JSON file can be opend with *TagAndTrack* for reviewing the results. The exact usage of the classifier is described in Listing 6.23.

```
using 4 threads for classifying.
...
image_hortVD_wp_68_71.jpg -> Tamura:POSITIVE LateFusion:POSITIVE JCD:POSITIVE
image_hortVD_wp_68_8.jpg -> Tamura:POSITIVE LateFusion:POSITIVE JCD:POSITIVE
image_hortVD_wp_68_79.jpg -> Tamura:POSITIVE LateFusion:POSITIVE JCD:POSITIVE
image_hortVD_wp_68_82.jpg -> Tamura:POSITIVE LateFusion:POSITIVE JCD:POSITIVE
image_hortVD_wp_68_78.jpg -> Tamura:POSITIVE LateFusion:POSITIVE JCD:POSITIVE
image_hortVD_wp_68_77.jpg -> Tamura:POSITIVE LateFusion:POSITIVE JCD:POSITIVE
image_hortVD_wp_68_81.jpg -> Tamura:POSITIVE LateFusion:POSITIVE JCD:NEGATIVE
image_hortVD_wp_68_83.jpg -> Tamura:NEGATIVE LateFusion:NEGATIVE JCD:POSITIVE
image_hortVD_wp_68_80.jpg -> Tamura:POSITIVE LateFusion:POSITIVE JCD:POSITIVE
image_hortVD_wp_68_9.jpg -> Tamura:POSITIVE LateFusion:POSITIVE JCD:POSITIVE
image_hortVD_wp_68_102.jpg -> Tamura:POSITIVE LateFusion:NEGATIVE JCD:NEGATIVE
image_hortVD_wp_68_84.jpg -> Tamura:NEGATIVE LateFusion:NEGATIVE JCD:POSITIVE
image_hortVD_wp_68_101.jpg -> Tamura:POSITIVE LateFusion:NEGATIVE JCD:NEGATIVE
image_hortVD_wp_68_103.jpg -> Tamura:POSITIVE LateFusion:NEGATIVE JCD:NEGATIVE
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Feature     TP   TN  FP  FN  Precision Recall    TNRate    FPRate    Accuracy  FMeaseure WFMeaseureMccMeasure
Tamura      165  34  37  21  0.816832  0.887097  0.478873  0.521127  0.774319  0.850515  0.850515  0.399001
LateFusion  172  51  20  14  0.895833  0.924731  0.718310  0.281690  0.867704  0.910053  0.910053  0.661481
JCD         162  45  26  24  0.861702  0.870968  0.633803  0.366197  0.805447  0.866310  0.866310  0.509303

writing html output to: results-1435447137.html
duration: 78.293seconds.
```

Figure 6.9: Console output of the classifier using the features JCD and Tamura.

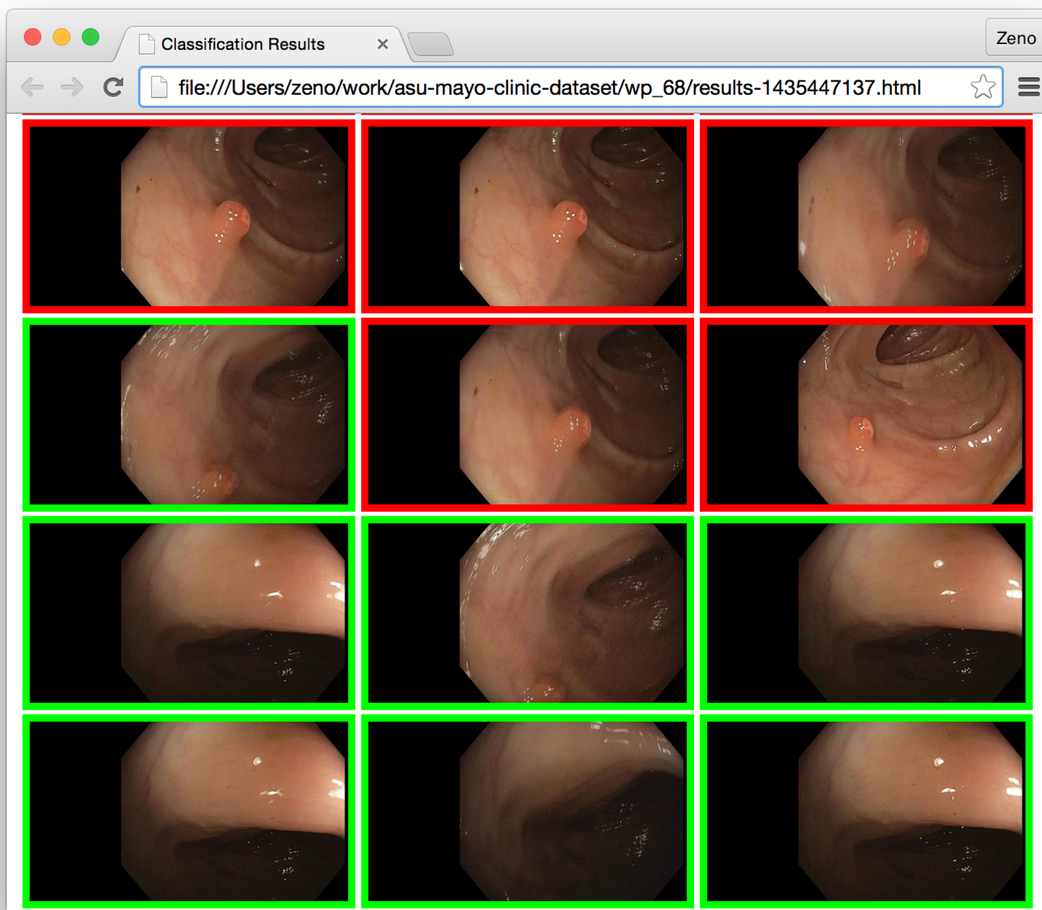


Figure 6.10: HTML output of the classifier using the features JCD and Tamura.

Listing 6.23: Usage of global image feature based Classifier.

```
java -jar classifier.jar -i /to/classify -c /classifier -f feature
    -c | -classifierIndex      Previously indexed training data.
                               It is possible to provide indices of
                               multiple training datasets.
    -p | -posClassifierIndex   Previously indexed training data,
                               containing positives only.
                               It is possible to provide indices of
                               multiple training datasets.
    -n | -negClassifierIndex   Previously indexed training data, containing
                               negatives only. It is possible to provide
                               indices of multiple training datasets.
    -i | -input                Previously indexed data to be classified.
                               Any indexed file starting with 'p' is
                               considered a positive sample.
                               Any indexed file starting with 'n' is
                               considered a negative sample.
    -P | -inputPositive        Previously indexed data to be classified.
                               The generated metrics rely on this index
                               only containing positive samples.
    -N | -inputNegative        Previously indexed data to be classified.
                               The generated metrics rely on this index
                               only containing negative samples.
    -v | -inputVideo           Video file to classify frame by frame.
    -f | -feature              A feature to use for classification.
                               Multiple features can be provided.
                               Possible features are for example:
                               JCD, FCTH, EdgeHistogram, ...
                               The respective feature must be present in
                               any index provided.
    -m | -measure              The measure to use. (any of: classCount,
                               weightedByRank, weightedByDistance,
                               weightedByAverageDistance)

    All command line options must always be used in pairs of option and value.
    At least a single classifier index is required.
```

The implementation of the classifier is described in multiple subsequent listings. In all these listings, a lot of boilerplate code, exception handling and error checking has been removed to improve the readability. The main entry point for the classifier is presented in Listing 6.24. The main purpose of this class is to parse, the command line arguments and initialize any member variables accordingly. The *OpenCV* library must be loaded explicitly to avoid any linking errors at runtime, because *OpenCV* is a native library and not an actual *Java* module. We create an instance of the class *Classifier* and we pass a map that contains all the paths to the *classifier indices* that should be used for classification. Entries in this map contain a path to the index of a training dataset and map every index to a *SampleType*, which can either be *POSITIVE*, *NEGATIVE* or *INVALID*. In case of *INVALID*, the index might contain both *POSITIVE* and *NEGATIVE* samples. In this case, the *SampleType* is deduced from the original file name for each image separately, and all images should therefore be prefixed with either ”p_” for a positive sample or ”n_” for a negative one. The constructor of the *Classifier* class initializes several variables and datastructures and decides on the amount of threads to use, based on the amount of available cores reported by the *JVM*. It also maps all the index files into memory and instantiates *AugmentedIndexReaders* for those files. An *AugmentedIndexReader* is a composition of a *Lucene IndexReader* and a *SampleType* defining the type of samples contained in this index.

The *Classifier* can be used to classify the frames of an input video or an input index. If a video is processed, it is read using *OpenCV* and the output is stored in *JSON* format compatible with our previously developed *TagAndTrack* tool. *TagAndTrack* marks positive frames with a red section in a separate bar below the seek-bar (Figure 6.11). If an input index is processed, the results are stored in HyperText Markup Language (HTML) format. The HTML page that is produced presents every single image of the input index with either a green border for negative samples or a red border for positive ones, as presented in Figure 6.10. The results are also displayed on the console, as presented in Figure 6.9. When processing an index, the *SampleType* of the input is detected the same way as it is done for the classifier indices. This makes it possible to display performance metrics for the classification in addition to the normal classification results. These performance metrics are written to the console.

Listing 6.24: Main entry function for the classifier.

```
package no.simula.classifier;
<<< Import various Java, lire, lucene, opencv classes. >>>

enum MeasureType{COUNT, WEIGHTED_BY_RANK, WEIGHTED_BY_DISTANCE, WEIGHTED_BY_AVG_DISTANCE}
enum SampleType{POSITIVE, NEGATIVE, INVALID}

public class Main {
  // All the variables below are set by command line arguments.
  private static HashMap<String, SampleType> classifierIndices = new HashMap<String, SampleType>();
  private static ArrayList<String> imageFeatures = new ArrayList<String>();
  private static HashMap<String, SampleType> inputDataIndices = new HashMap<String, SampleType>();
  private static MeasureType measureType = MeasureType.COUNT;
  private static String inputVideo = null;

  public static void main(String[] args) throws Exception {
    System.loadLibrary(Core.NATIVE_LIBRARY_NAME); // Load OpenCV library at startup.
    <<< Verify and parse command line arguments. >>>
    <<< Take start timestamp. >>>

    Classifier classifier = new Classifier(classifierIndices);
    ClassificationList classificationList = null;
    if (inputVideo != null) {
      classificationList = classifier.classifyVideo(inputVideo, imageFeatures, measureType);
      classificationList.print();
      classificationList.exportJSON(inputVideo);
    } else {
      classificationList = classifier.classifyDataset(inputDataIndices, imageFeatures, measureType);
      classificationList.print();
      classificationList.createMetrics();
      classificationList.createHTML();
    }
    <<< Calculate and print processing time. >>>
  }
}
```

The *main* function of the classifier creates an instance of the class *Classifier* and, depending on the input data, calls *classifyVideo* or *classifyDataset* on this object. The implementation of those functions is almost identical. The only difference is that in *classifyDataset*, there is no need to spin a thread for reading and decoding the video frames. Instead of using *dequeueVideoFrame* we can call *getNextDocument* to receive the next image to process. Both the functions *dequeueVideoFrame* and *getNextDocument* are synchronized to allow concurrent access by multiple threads; also, both functions return documents from their respective input data source in sequential order.

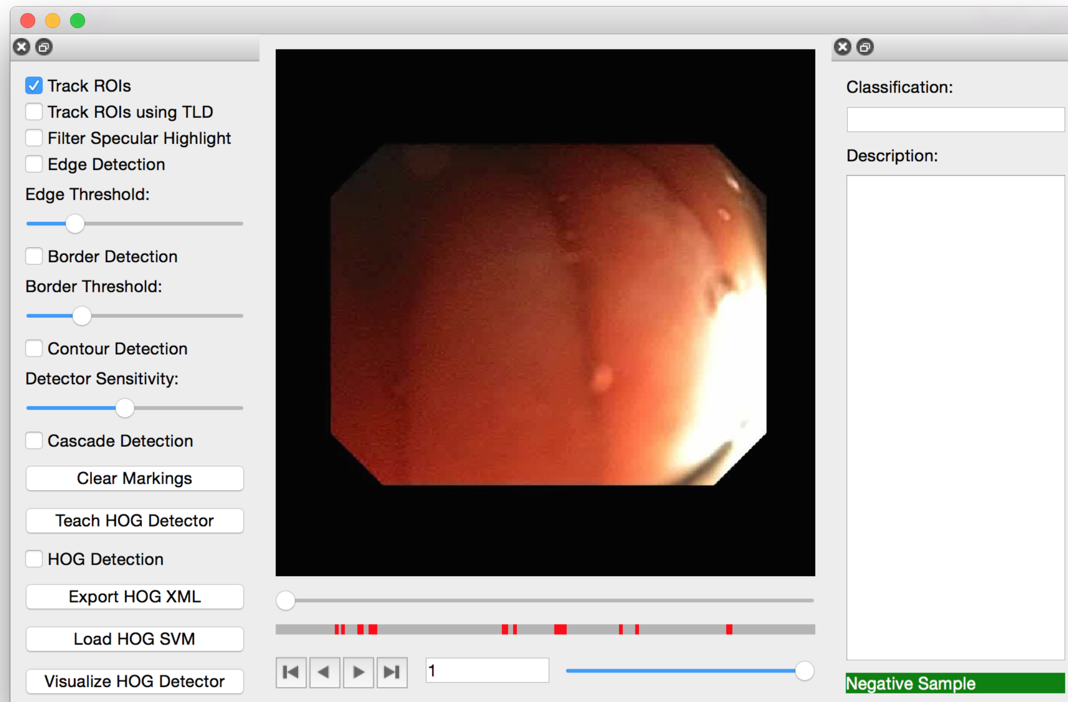


Figure 6.11: TagAndTrack showing the results from the global image features classifier.

We will not discuss the function *classifyDataset* in more detail, because it is basically just a simplified version of *classifyVideo*. The implementation of the function *classifyVideo* is presented in Listing 6.25. We use *lambdas* to create separate *Runnables* for reading the video input file and for processing the buffered images. The *Runnable* that we call *videoRead* uses an *OpenCV VideoCapture* object to read and decode the video frames. The function *mat2img* is used to convert from an *OpenCV Mat* object to a *BufferedImage*. We use a custom document builder to extract all the requested image features from the *BufferedImage* and create a *Document* containing that information. Such *Documents* are then buffered in a queue datastructure called *videoFrames*. We use an *ExecutorService* with a fixed thread pool for processing the *Runnables*. Each processing thread creates a separate list of search providers for all requested image features. It then dequeues video frames as long as there are any frames available and calls *classifyDocument* providing the frame as a *Document*, and the list of *SearchProviders* and a reference to a *ClassificationList* for storing the results as arguments. The *ClassificationList* is the return value of *classifyVideo* and is a single common object shared by all threads. Its insert function is therefore synchronized. If the decoding of the video file is not finished, but no frame is currently available in the buffer, the function *dequeueVideoFrame* will block. Only if no frame is available in the buffer anymore and the decoding of the video file finished, *dequeueVideoFrame* will return *null* and the processing thread will exit. All threads are equally responsible for updating the progress indicator displayed on the console, which therefore must be done in a synchronized block or synchronized function to avoid a corrupted terminal output.

Listing 6.25: Implementation of the function classifyVideo.

```
// Defined by OpenCV, but not exported in Java.
private static final int CV_CAP_PROP_POS_FRAMES = 1;
private static final int CV_CAP_PROP_FRAME_COUNT = 7;

public ClassificationList classifyVideo(String inputVideo
    , ArrayList<String> featureNames, MeasureType measureType) throws IOException {
  ClassificationList classificationList = new ClassificationList();
  VideoCapture capture = new VideoCapture(inputVideo);
  int totalDocuments = (int)capture.get(CV_CAP_PROP_FRAME_COUNT);
  Mat frameMat = new Mat();
  DocumentBuilder builder = getCustomDocumentBuilder(featureNames);
  ExecutorService pool = Executors.newFixedThreadPool(numThreads);

  Runnable videoRead = () -> {
    while (capture.read(frameMat)) {
      BufferedImage frame = mat2img(frameMat);
      double pos = capture.get(CV_CAP_PROP_POS_FRAMES);
      Document doc = builder.createDocument(frame, String.valueOf((int)pos));
      enqueueVideoFrame(doc);
    }
    synchronized(videoFrames) {
      finishedReadingVideo = true;
      videoFrames.notifyAll();
    }
  };
  pool.execute(videoRead);

  for (int runnableId = 0; runnableId < numThreads; ++runnableId) {
    Runnable r = () -> {
      ArrayList<SearchProvider> searchProviders = setupSearchProviders(featureNames);
      while (true) {
        Document imageDocument = dequeueVideoFrame();
        if (imageDocument == null) break;
        <<< Show progress on screen. >>>
        classifyDocument(imageDocument, searchProviders, classificationList);
      }
    };
    pool.execute(r);
  }

  pool.shutdown();
  pool.awaitTermination(365, TimeUnit.DAYS));
  return classificationList;
}

public static BufferedImage mat2img(Mat img) {
  byte[] data = new byte[img.cols() * img.rows() * (int)img.elemSize()];
  img.get(0, 0, data);

  int type = BufferedImage.TYPE_3BYTE_BGR;
  if(img.channels() == 1) type = BufferedImage.TYPE_BYTE_GRAY;

  BufferedImage bufferedImage = new BufferedImage(width, height, type);
  bufferedImage.getRaster().setDataElements(0, 0, width, height, data);
  return bufferedImage;
}
```

The function presented in Listing 6.26 creates a *ChainedDocumentBuilder*, chaining a variable amount of separate document builders. The actual amount and type of document builders to be chained is defined by the user on the command line as a set of feature names. The function derives the class names of *LIRe* features from the feature names and creates matching instances of *GenericDocumentBuilder*.

Listing 6.26: Implementation of function getCustomDocumentBuilder.

```
public static DocumentBuilder getCustomDocumentBuilder(ArrayList<String> featureNames) {
  ChainedDocumentBuilder builder = new ChainedDocumentBuilder();
  for (String featureName : featureNames) {
    String className = "net.semanticmetadata.lire.imageanalysis." + featureName;
    Class<? extends LireFeature> c = (Class<? extends LireFeature>) Class.forName(className);
    builder.addBuilder(new GenericDocumentBuilder(c, true));
  }
  return builder;
}
```

The function *setupSearchProviders* (Listing 6.27) returns a list of *SearchProviders* based on the list of *LIRe* feature names passed as an argument. Each of the processing threads will need its separate *SearchProviders*, hence, this function must be called once per thread. It creates a separate *SearchProvider* instance for every single image feature by passing *numNeighbors*, *featureName*, and *featureDescriptor* as arguments to the *SearchProvider*-constructor. The value *numNeighbors* defines how many nearest neighbors the *SearchProvider* should report. This value should be significantly smaller than the amount of samples of the *SampleType* with the fewest samples in the classifier index. For example, if our *classifier index* contains 30 *POSITIVE* and 100 *NEGATIVE* samples, and we define *numNeighbors* to be 40, any list of nearest neighbors would always contain at least 10 *NEGATIVE* neighbors. Depending on how the results are weighted, this could significantly reduce the detecting performance. For this thesis, we have hard-coded the value *numNeighbors* to 77. This value could most likely be reduced to

make the classifier faster. However, we have not made an attempt to find an optimal value, as this value most likely depends on the specific dataset that is processed.

Listing 6.27: Implementation of the function creating SearchProviders for a list of feature names.

```
private ArrayList<SearchProvider> setupSearchProviders(ArrayList<String> featureNames) {
  ArrayList<SearchProvider> sp = new ArrayList<>(featureNames.size());
  for (String featureName : featureNames) {
    String featureDescriptor = ((String) DocumentBuilder.class.getField("FIELD_NAME_" + featureName.toUpperCase()).get(null));
    SearchProvider searchProvider = new SearchProvider(numNeighbors, featureName, featureDescriptor);
    sp.add(searchProvider);
  }
  return sp;
}
```

The class *SearchProvider* extends the class *BitSamplingImageSearcher* and allows us to store the feature name it is processing in the instance of the class. Storing the feature name in the *SearchProvider* is needed, because we allow processing of multiple image features simultaneously. When collecting the search results, we need to know which image feature the results are for. We decided to add this information directly to the SearchProvider. This is the easiest way to ensure data integrity, when using multiple threads and multiple instances of the class per thread. The implementation of the class is presented in Listing 6.28

Listing 6.28: Implementation of class SearchProvider.

```
public class SearchProvider extends BitSamplingImageSearcher {
  SearchProvider(int numNeighbors, String featureName, String featureDescriptor) {
    super(numNeighbors, featureDescriptor, featureDescriptor + "_hash"
      , (LireFeature) Class.forName("net.semanticmetadata.lire.imageanalysis." + featureName).newInstance(), 1000);
    m_featureName = featureName;
  }
  private String m_featureName;
  public String featureName() { return m_featureName; }
}
```

The functions used for managing the input queue for video frames are presented in Listing 6.29. The function *enqueueVideoFrame* is called from the video decoder thread to add a new video frame to the queue. To avoid any inconsistencies caused by simultaneous access, the enqueue as well as the dequeue operations are synchronized on *videoFrames*. Once a new frame was added, we wake up a single waiting thread. The function *dequeueVideoFrame* is called from any of the worker threads processing the search. If the queue is empty, we yield and wait for a frame to become available, or, alternatively, if the video decoding has already finished, the function returns *null*.

Listing 6.29: Implementation of functions managing the input queue for video frames.

```
private void enqueueVideoFrame(Document doc) {
  synchronized (videoFrames) {
    videoFrames.addFirst(doc);
    videoFrames.notify(); // Notify a single waiting thread.
  }
}

private Document dequeueVideoFrame() throws InterruptedException {
  synchronized(videoFrames) {
    while (videoFrames.isEmpty()) {
      if (finishedReadingVideo) return null;
      videoFrames.wait(); // Wait for a frame to become available.
    }
    return videoFrames.removeLast();
  }
}
```

The function *classifyDocument* (Listing 6.30) is called once for every document. It takes the document to be classified as an argument, as well as a list of *SearchProviders* and a reference to a *ClassificationList*. The document is then processed by each *SearchProvider* separately and all the results are stored in a *Classification* object. The function also adds a late fusion value and eventually adds the *Classification* object to the list of classifications. In machine learning, there is generally two ways of combining feature vectors. The first one is *early fusion* and the second one is *late fusion*. When using *early fusion*, the extracted features are combined before the learning step. When using *late fusion*, the features are processed separately, and only the results are weighted and combined. We have decided to use late fusion, because this fusion scheme tends to provide better performance for video data [60].

Listing 6.30: Implementation of the function for classifying an image document.

```
private void classifyDocument(Document img, ArrayList<SearchProvider> providers, ClassificationList cl) {
  Classification classification = new Classification(img);

  float lateFusionValues[] = new float[SampleType.values().length];
  for (SearchProvider searchProvider : providers) {
    SampleInformation detectedSampleInfo = getMatchingSampleTypeForDocument(img, searchProvider, measureType);
    lateFusionValues[detectedSampleInfo.type.ordinal()] += detectedSampleInfo.confidence;
    classification.insert(searchProvider.featureName(), detectedSampleInfo.type);
  }

  classification.insert("LateFusion", lateFusionValues[SampleType.POSITIVE.ordinal()]
    > lateFusionValues[SampleType.NEGATIVE.ordinal()] ? SampleType.POSITIVE : SampleType.NEGATIVE);
  cl.insert(classification);
}
```

We use the function *getMatchingSampleTypeForDocument* to decide which type cluster a certain image document belongs to. We search all *indexReaders* that contain training data for the most similar documents and combine the results in a max heap. The implementation of the class *Utils.MaxHeap* is described in Listing 6.32. We use the heap structure to automatically sort the documents in a hierarchical order. That way, we always end up with the least similar document (biggest distance) at the top of the heap. This allows us to keep a fixed amount of most similar documents in the data structure. Whenever we want to insert a new document, we only have to peek at the top document, to decide if the new document should be inserted into the heap, or if it should not be inserted, due to being less similar than the least similar document currently contained in the heap. Once we have combined all the results from the different *indexReaders* we can repeatedly call *poll()* to remove the top document from the heap. This allows us to process the combined search results in a prioritized order and apply weighting to the results. We distinguish between different measurement types. *COUNT* is simply counting the amount of occurences of the same type in the result set. *WEIGHTED_BY_RANK* counts the amount of occurences and applies weighting to each occurence, based on the rank of the individual results. *WEIGHTED_BY_DISTANCE* applies weighting on the distance of the occurences, and *WEIGHTED_BY_AVG_DISTANCE* classifies based on the average of the weighted distances. The function returns a *SampleInformation* object which contains a *SampleType* value and a confidence value. For our use case, the *SampleType* value will always be either *POSITIVE* or *NEGATIVE*, because we only do binary classification. However, the implementation is done in a generic fashion, so we could add classification into further subcategories later. The confidence value is the amount of samples that contributed to the result. This value is used later on for combining the results for multiple image features using late fusion. The implementation of the function *getMatchingSampleTypeForDocument* is presented in Listing 6.31.

Listing 6.31: Implementation of the search based classification for a single image feature.

```
public static final String SAMPLE_TYPE_DESCRIPTOR_NAME = "sampleType";

public SampleInformation getMatchingSampleTypeForDocument(Document document
  , AbstractImageSearcher searcher, MeasureType measureType) {
  // Initialize a HashMap to keep the score per type.
  HashMap<SampleType, Utils.MutableFloat> typeScore = new HashMap();
  for (SampleType type : SampleType.values()) typeScore.put(type, new Utils.MutableFloat());

  // Search the indexReaders, and store all the results in a MaxHeap.
  Utils.MaxHeap matches = new Utils.MaxHeap(numNeighbors);
  for (AugmentedIndexReader augmentedReader : indexReaders) {
    ImageSearchHits imageSearchHits = searcher.search(document, augmentedReader.indexReader);
    for (int i = 0; i < imageSearchHits.length() && i < numNeighbors; i++) {
      Document matchingDocument = imageSearchHits.doc(i);
      // Add the sample type to the document.
      matchingDocument.add(new StoredField(Utils.SAMPLE_TYPE_DESCRIPTOR_NAME, augmentedReader.sampleType.name()));
      matches.insert(imageSearchHits.score(i), matchingDocument);
    }
  }

  int numMatches = matches.size();
  // Process all the matches in order of highest priority
  // and calculate the respective (weighted) scores for all possible types.
  while (!matches.isEmpty()) {
    Utils.PrioritizedDocument prioritizedDocument = matches.poll();
    SampleType sampleType = Utils.getSampleTypeFromDocument(prioritizedDocument.document);
    switch (measureType) {
      case COUNT: typeScore.get(sampleType).increment(1); break;
      case WEIGHTED_BY_RANK: {
        float wc = 1.0f / ((float) matches.size() + 1.0f);
        typeScore.get(sampleType).increment(wc); break;
      }
      case WEIGHTED_BY_DISTANCE:
      case WEIGHTED_BY_AVG_DISTANCE: {
        float weight = 1.0f / ((float) matches.size() + 1.0f);
        // Range of score is 0 to MAX_FLOAT, where 0 is a perfect match.
        float was = prioritizedDocument.score * weight;
```

```
38            typeScore.get(sampleType).increment(was); break;
39          }
40        }
41      }
42
43      // Determine the highest and lowest scoring types.
44      SampleType highType = SampleType.INVALID, lowType = SampleType.INVALID;
45      float high = 0, low = Float.MAX_VALUE;
46      Iterator it = typeScore.entrySet().iterator();
47      while (it.hasNext()) {
48        HashMap.Entry pair = (HashMap.Entry)it.next();
49        if (pair.getKey() == SampleType.INVALID) continue;
50        Utils.MutableFloat score = (Utils.MutableFloat)pair.getValue();
51        float f = score.get();
52        if (measureType == MeasureType.WEIGHTED_BY_AVG_DISTANCE) f = f / score.getIncrements();
53        if (f > high) high = f; highType = (SampleType) pair.getKey();
54        if (f < low) low = f; lowType = (SampleType) pair.getKey();
55      }
56
57      // Calculate the confidence in the result.
58      SampleInformation info = new SampleInformation();
59      switch (measureType) {
60        case COUNT:
61        case WEIGHTED_BY_RANK:
62          info.confidence = (float)typeScore.get(highType).getIncrements();
63          info.type = highType; break;
64        case WEIGHTED_BY_DISTANCE:
65        case WEIGHTED_BY_AVG_DISTANCE:
66          info.confidence = (float)typeScore.get(lowType).getIncrements();
67          info.type = lowType; break;
68      }
69      info.confidence = info.confidence / numMatches;
70      return info;
71    }
72
73    private class SampleInformation {
74      SampleType type = SampleType.INVALID;
75      Float confidence = 0f;
76    }
```

We need a *MaxHeap* datastructure to keep a fixed amount of documents in a prioritized order and to efficiently replace entries with more similar search results. We therefore implemented a *MaxHeap* class, which inherits from a regular *PriorityQueue*. Our *MaxHeap* contains objects of type *PrioritizedDocument*, which contain a score value. The score value is the value by which we want to sort the elements in the heap. In *Java*, the *PriorityQueue* is implemented as a regular mininum heap. To reverse the ordering of the items in the heap we therefore also needed to implement a new comparator class *MaxHeapComparator*. We pass the maxiumum size of the heap as an argument to the constructor. The insert method of the *MaxHeap* checks the current size of the heap, and if the maxiumum size is reached, the object with the highest score is dropped. The implementation for the class *MaxHeap*, *PrioritizedDocument* and *MaxHeapComparator* are presented in Listing 6.32.

Listing 6.32: Utils class containing several helper functions and helper classes.

```
1   public static class MaxHeap extends PriorityQueue<PrioritizedDocument> {
2     private int maximumSize;
3     public MaxHeap(int maximumSize) {
4       super(maximumSize, new MaxHeapComparator());
5       this.maximumSize = maximumSize;
6     }
7
8     public void insert(float score, Document document) {
9       PrioritizedDocument top = peek();
10      if (top == null || size() < maximumSize) {
11        add(new PrioritizedDocument(score, document));
12      } else if (top.score > score) {
13        poll();
14        add(new PrioritizedDocument(score, document));
15      }
16    }
17  }
18
19  public static class MaxHeapComparator implements Comparator<PrioritizedDocument> {
20    public int compare(PrioritizedDocument x, PrioritizedDocument y) {
21      float c = y.score - x.score;
22      if (c < 0) return -1;
23      if (c > 0) return 1;
24      return 0;
25    }
26  }
27
28  public static class PrioritizedDocument {
29    Document document;
30    float score;
31    public PrioritizedDocument(float score, Document document) {
32      this.score = score;
33      this.document = document;
34    }
35  }
```

There are a few additional classes and functions which we just describe briefly instead of listing the complete implementation. The class *ClassificatonRate* is used as a storage container for true-/false-positive and true-/false-negative rates for a single image feature. The class *ClassificationMetrics* is a map that stores one such *ClassificationRate* object per image feature. This class further has a *print* function, which calculates several metrics such as precision, recall, accuracy, F1 score, and weighted F1 score. This information is then printed to the console (Figure 6.9 / bottom) and can be used to assess the performance of the classifier.

The class *Classification* is used to store information for a single image. It stores expected sample type, image name or frame number, and classification results for several image features. The class *ClassificationList* implements a linked list of *Classification* objects. It also contains a special comparator to sort the linked list by frame numbers stored in the *Classification* objects. This is for example used for writing a sorted list of negative frames and a sorted list of positive frames to a *JSON* file.

The classes *MutableInt* and *MutableFloat* are simple wrappers for integer and float values with some additional convenience functions. These classes allow us to increment or alter the value of the variable in place, without allocating a new integer or float. This is in particularly useful when an integer or float is stored in a map and needs to be updated. Using a *MutableInt* or a *MutableFloat*, this can be done in a single Java-statement and must not be broken down into read, update and write statements. The function *getSampleTypeFromDocument* returns the sample type stored in a *Lucene* document, if it is available. If the document does not contain a stored sample type, the function will fall back to *getSampleTypeFromName*. The function *getSampleTypeFromName* is used when the sample type is not stored in the document, but encoded in the image name instead.

6.2 Evaluation and Discussion

We have experimented with three different machine learning approaches for detecting colon polyps. The main challenge we were facing was the small amount of data we had available for the machine learning process. Having a small amount of data, which is not diverse enough, makes it more likely that a trained classifier will be overfit. This basically means that the classifier will have poor detecting-performance, because, instead of generalizing from the input samples, it will simply ”memorize” training data.

It is relatively easy to find some colonoscopy pictures on the Internet, but to be able to do machine learning, we need significantly more than just a few pictures per disease pattern. For colon polyps, the ASU-Mayo Clinic polyp database[17] is the currently biggest publicly available dataset. Hence, we have been using this dataset for many of our experiments.

To assess the performance of our classifiers, we use the measures *precision*, *recall* and *F-score*. *Precision* measures the fraction of the detected-positive instances, which are true-positive. The formula for calculating the *precision* is listed subsequently. *TP* is the number of true-positive instances, *FP* is the number of false-positive instances, *TN* is the number of true-negative, *FN* is the number of false-negative and *P* is the number of positive instances.

$$PRECISION = \frac{TP}{(TP + FP)}$$

Recall is the fraction of all true-positive instances, which are also detected positive. The formula for calculating the *recall* is:

$$RECALL = \frac{TP}{P}$$

F-score (also *F-measure* or *F1-score*) is the harmonic mean of *precision* and *recall*. It is thereby a combination of these two measures in a single number.

$$F = 2 * \frac{PRECISION * RECALL}{PRECISION + RECALL}$$

[17] `http://www.polyp2015.com/wp/`

In several sections, the measure *accuracy* is used as well. *Accuracy* is the proportion of correctly classified items out of all the items classified.

$$ACCURACY = \frac{(TP + TN)}{(TP + TN + FP + FN)}$$

Where applicable, we also conduct a **leave-one-out cross-validation**. This is a technique to assess the generalization of a predictive model. In our case, it describes the process where the training and testing datasets are rotated, leaving out a single different non-overlapping item or portion for testing, and using the remaining items for training. This process is repeated until every item or portion has been used for testing exactly once [61].

6.2.1 Cascade Classifier Training

We have used *opencv_traincascade* for training a Haar-based cascading classifier with data exported from our *TagAndTrack* tool. Since we only had a relatively small amount of imaging data available and to allow detecting polyps in various orientations, we added functionality to *TagAndTrack* to export rotations of every selected region. This functionality generates a total of 18 training samples per selected region. Training such a classifier is an extremely lengthy and computationally intensive task. The procedure usually takes several days or even weeks. Unfortunately, this also limited us in trying many different combinations, training settings or datasets. Unless specified otherwise, we have always been using the same machine for the training, a fairly powerful Mac Pro Mid 2010. The computer was barely usable for anything else during training, because we allowed *opencv_traincascade* to use almost all the available memory. We have listed the training times for different datasets in Table 6.3. The exact settings and the computer that was used are specified in detail in Paragraph 6.1.1.4.

Table 6.3: Training Time for a Haar-based Cascading Classifier

Positive Samples	Negative Samples	Training Time
900	1427	3h 30min
4518	1427	13h 12min
10278	24241	13d 2h 23min
42300	24107	72d 1h 11min

We have conducted three separate experiments for measuring the performance of the classifiers.

6.2.1.1 Experiment 1

For the first experiment we have selected a video sequence with a total of $2,025$ video frames, where 251 of these frames show the same polyp from different positions (due to the endoscope being moved). The video sequence also contains a still sequence of 117 frames, showing the same polyp. We have removed these 117 frames from all subsequent calculation, so they will not influence our measurements. We have also removed 481 frames with surgical equipment visible, as surgical equipment might interfere with our detection and should never appear in a capsule endoscopy video. This leaves us with a total of $1,678$ frames. Those are 251 frames where a polyp is visible (positive frames) and $1,427$ frames without any polyp (negative frames). We have then trained a classifier using all these negative samples and the usual amount of 18 rotations for the positive samples. We then used the exact same video frames and measured the performance of the classifier. We were doing this, to show that some of the good results reported in related work are caused by overfitting. The second part of our evaluation uses strictly separated training and testing sets. And in this second case, the results seem considerably less promising, leading to the insight that despite reports of good results, the classification problem is not solved yet.

The results for the first experiment (overfit) are listed in Table 6.4. The number of counted false negatives matches our expectation. The number of false positives seems rather high, considering that

we used the same data for training and testing. Especially, since a single false positive frame might contain multiple false detections. Taking a closer look at the false negatives, we observed, that false positives often occur on video frames that are not in focus. Very often, such out-of-focus frames even contain multiple false positives. Further, we noticed that in this video, there are a few sequences where the camera points towards one side of the colon, making the whole wall of the colon look like a giant protrusion due to the distortion caused by the short focal length of the camera lens. This leads to the conclusion that, despite the minimal amount of training data and the obvious overfit classifer, a small degree of generalization was achieved. However, instead of the desired generalization, the effect in this case is that we have a rather high amount of false positives.

Table 6.4: Performance of a classifier using the same dataset for training and detecting.

True Positives	False Negatives	False Positives	Total Frames
250	1	272	1678

Recall	Precision	F1 Score
0.996	0.4789	0.6468

6.2.1.2 Experiment 2

For our second experiment, we essentially use the same dataset as we used for the first experiment. However, instead of using all the 251 positive samples, we are only using a randomly selected subset of 50 positive samples and their respective rotations. Therefore, we have a total of 900 positive samples and 1,427 negative samples for training. For evaluation, we then used all the 1,678 frames again. A total of 50 different images is for sure a very low amount for training a classifier to detect non-rigid objects. According to the OpenCV documentation[18], thousands of images are needed for training a face detector. With this experiment, we show that using images of the same polyp for training and testing the performance of a classifier defeats the purpose of the measurement, even if the image sets for training and testing are disjoint. This is something we have criticized in some of the related work we have discussed. We therefore expect to achieve a very low amount of false negatives and and a high amount of correctly detected frames again. The results for this experiment are presented in Table 6.5.

Table 6.5: Performance of a classifier trained with a subset (50 positive frames) of the testing dataset.

True Positives	False Negatives	False Positives	Total Frames
236	15	379	1678

Recall	Precision	F1 Score
0.94	0.384	0.545

We interpret the results from Table 6.5 as follows. The amount of false positives is significantly higher, most likely because of a too broad generalization caused by too few positive samples, but this value is only of marginal interest in this experiment. More interesting is that despite reducing the positive samples to only 20% of the available ones, we still correctly detect 94% of the frames showing the polyp. This supports our hypothesis that results for the performance of a classifier are only meaningful, if the training dataset and the testing dataset do not contain any pictures of the same polyp. This is because we only need a very small amount of samples of one specific polyp, to be able to detect the exact same polyp on different images. More specifically, when conducting leave-one-out cross-validation, it is important that we leave out whole video sequences and not just single frames.

[18] http://docs.opencv.org/doc/user_guide/ug_traincascade.html

6.2.1.3 Experiment 3

For the third experiment, we train a classifier with data from a total of 2,350 positive and 24,107 negative video frames of 20 different video sequences, containing frames showing 10 different polyps. This produces a total of 42,300 positive samples. We planned to analyze the performance of the resulting classifier for a video sequence contained in the training sequence, as well as for a video sequence that we put aside for testing only and would not use for training at all. It took several weeks for this training process to complete. However, despite the enormous training effort, the resulting classifier turned out to be not usable at all. On every single video frame, the classifier detects multiple false positives. We usually experience more than 10 false positives per video frame, whereas the exact amount of false positives per frame mostly depends on the resolution of the video sequence. Not a single video frame was free of false positives, no matter if the sequence was used for training or not. The classifier always evaluates to "contains a polyp" and does not provide any actual information. We therefore do not list the results in a separate table.

We have conducted further experiments to find the reason why this classifier is unusable. The most reasonable explaination we have found is that it is caused by the many rotations we create for every original sample and the lack of strong, common features in the available samples. Haar-features used for the detection of faces are presented in [48]. The example for Haar-features that ships with *OpenCV* is also implementing face recognition. Faces do have very strong optical features, and usually the orientation on images only changes minimal, also supporting the hypothesis that our classifiers trained with fewer samples were heavily overfit. These classifiers caused significantly less false positives, because instead of properly generalizing, they simply encoded the training instances. We have also verified this by training several different Haar-based classifiers to detect footballs instead of polyps. Optically, a football clearly separates from the background of a football field. We therefore expected that it should be rather easy to detect footballs. However, there are many different color patterns for the surface of a football, and a football can be rotated any possible direction. The common optical features in all the samples are therefore only the round shape and the light color. With these football training datasets we also experienced the same issue. The more training samples we used, the more false positives were caused by the trained classifier.

We therefore conclude that the unknown orientation of polyps and the lack of strong, common characteristic optical features make Haar-feature-based training an unsuitable machine learning approach for detecting polyps in the intestines.

6.2.2 Histogram of Oriented Gradients Detector

In Section 6.1.2, we have described our first experiments, implementing HOG-based training and detection directly into *TagAndTrack*. This allows us to select regions in a single video sequence for training directly in *TagAndTrack*. The generated classifier can then be used on the same video sequence for testing. As discussed in Section 6.2.1, using frames of the same video sequence for training and for testing is problematic. But as the first experiment for testing the capabilities of HOG-based detectors, this was still a valid approach and it led to some very promissing results, which are presented in Table 6.6. These numbers show that HOG-based detectors are indeed very good at detecting objects based on a very small amount of training data. Also, the number of false positives is incredibly low.

As the results with the *TagAndTrack* integrated implementation were very promissing, we then implemented a separate HOGTrainer. This allows us to train HOG-based classifiers with data from multiple video sequences. Experimenting with that implementation, we quilckly found several limitations. First of all, all training samples for a HOG-based classifier must have a smiliar aspect ratio. We enforce this by allowing the *dlib* implementation to drop any samples with an incompatible aspect ratio. This is a limitation, but not necessarily a problem, as it is not very likely that one would want to detect objects of a very different aspect ratio with the same classifier. We even considered only allowing quadratic selection instead of rectangular one.

Trying to train a classifier with the same number of samples as we did for the Haar-based classifier and using the same number of rotations, we immediately ran out of memory. This is because *dlib* is

Table 6.6: Detector performance of HOG-based classifiers trained in TagAndTrack.

Training Samples	True Positives	False Negatives	False Positives	Total Frames
5	220	31	1	1678

Recall	Precision	F1 Score
0.876	0.995	0.932

Training Samples	True Positives	False Negatives	False Positives	Total Frames
10	241	10	1	1678

Recall	Precision	F1 Score
0.996	0.96	0.978

designed to load all the images into memory before running the training algorithm, instead of loading them on demand. We therefore reduced the number of rotations to 9 per positive frame, and thereby reduced the number of samples for training.

The next problem that we encountered was that any classifier trained with our *HOGTrainer* would never actually trigger; it did not detect the same polyp it was trained with, neither it detected any false positives. This was somewhat a surprise after our first, very promissing experiment. The reason was however easy to find. The only difference in training was that when training with *HOGTrainer*, we also added rotations and mirrored samples to the dataset. Visualizing the resulting detector makes the explanation to this problem obvious. When training a HOG-based classifier with rotations, no matter what dataset is used, the gradients will always end up building a circular pattern, as seen in Figure 6.12.

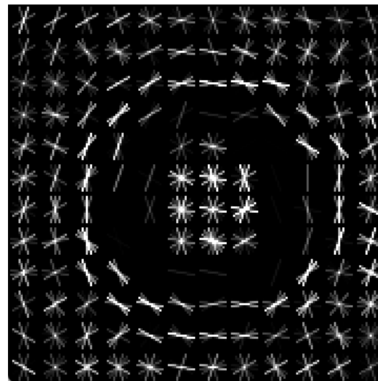


Figure 6.12: Visualizations of a HOG detector that was trained with rotations of multiple pictures of multiple polyps.

We therefore conclude that a HOG-based classifier cannot be trained with rotated images and that this method is only usable if the rough orientation of the object to be detected is known. For detecting humans, this is not a problem, since humans usually have an upright position in an image. But, for the detection of polyps this is a problem, because we would first have to define what the ”upright” position for a polyp is and would then have to transform all the training samples accordingly. Eventually, this would then also require to test rotations of every video frame when processing a video with the detector. This seems not to be a good option either, as it would be very computationally expensive to do this for all videos we eventually feed to the detector. We believe that HOG-based detectors are therefore not usable for the purpose we research.

Nevertheless, we collected a small amount of benchmarking data. Training a classifier with *HOGTrainer* is comparably fast. The time consumption for training classifiers with different datasets is listed in Table 6.7.

Table 6.7: Time consumption for training classifiers with HOGTrainer.

Frames	Samples	Training Time
50	450	3min 4sec
251	2259	17min 25sec
502	4518	44min 26sec
1004	9036	- (killed / out of memory)

6.2.3 Index of Global Image Features

To evaluate the performance of our global image features classifier, we have conducted measurements using the dataset from the ASU-Mayo Clinc polyp database[19]. This dataset consists of 20 separate videos, where 10 of these videos do contain polyps and the other 10 videos do not. The dataset further contains an image-mask for every single video frame, where polyps are marked. We can use this information as our ground truth. As discussed in Chapter 2, the commonly used metrics for measuring the performance of a classifier are *precision*, *recall* and *F-score*. Hence, we use these metrics to assess the performance of our classifier. The *F-score* or *F-score* is the harmonic mean of *precision* and *recall*. A problem of the *F-score* is, however, that a low value does not always indicate a bad-performing classifier. This can also be caused by a non-evenly distributed dataset (e.g., many more positive samples than negative ones) [62]. This problem can be addressed by using a *weighted F-score (WF1)* instead. *WF1* takes negative and positive class results into account and provides a more accurate and robust measure.

6.2.3.1 Benchmarking single Image Features

The tools we have developed allow us to use several different global image features for classification. The more image features we use, the more computationally expensive the classification becomes. Also, not all image features are equally important or provide equally good results for our purpose. As a first step, we therefore need to find out which image features we do want to use for classification.

To find out which image features provide the best results, we generated indices containing all possible image features for all frames of all video sequences from the ASU-Mayo Clinic database. We can use these indices for several different measurements and also for leave-one-out cross-validation. When using our classifier, the built-in metrics functionality can provide us with information on the performance of different image features for benchmarking. The classifier provides us with separate information for every single image feature, as well as the late fusion of all the selected image features.

For our first test, we ran the classifier with all possible image features selected, leaving out one video at the time, repeating the procedure until each video was left out exactly once. This is essentially the procedure for leave-one-out cross-validation. We then combined the reported values for true-positives, true-negatives, false-positives and false-negatives for all the runs, and calculated the metrics for the combined values. The results of this first test are presented in Table 6.8. The single image feature that generally achieves the best score is *Color and Edge Directivity Descriptor (CEDD)*, which is discussed in detail in [63]. Further, also the image features *Joint Descriptor for CEDD and FCTH (JCD)*, *EdgeHistogram*, *Rotation Invariant Local Binary Patterns*, *Tamura* and *Joint Histogram* achieve very good values. The late fusion of all the image features even achieves slightly better numbers. However, it is impractical to do a late fusion of all these image features as the calculation, indexing and searching of all image features is computationally expensive. Therefore, we want to find a small subset of two image features, which provides optimal results despite minimizing the computational effort.

6.2.3.2 Finding an optimal Image Feature Subset

In the previous paragraph, we benchmarked single image features. To create a more robust classification mechanism, we want to find an optimal subset of two image features, which we can combine using

[19] `http://www.polyp2015.com/wp/`

Table 6.8: Leave-one-out cross-testing combined for all supported image features. The best values are marked green.

Feature	**TP**	**TN**	**FP**	**FN**	**Prec.**	**Recall**	**F1**
JointHistogram	3369	13826	1085	511	0.7563	0.8682	0.8084
JpegCoefficientHistogram	3224	13772	1139	656	0.7389	0.8309	0.7822
Tamura	3392	13861	1050	488	0.7636	0.8742	0.8151
FuzzyOpponentHistogram	3341	13552	1359	539	0.7108	0.8610	0.7787
SimpleColorHistogram	2736	13563	1348	1144	0.6699	0.7051	0.6870
JCD	3556	13777	1134	324	0.7582	0.9164	0.8298
FuzzyColorHistogram	2708	13243	1668	1172	0.6188	0.6979	0.6560
RotationInvariantLlBP	3479	13829	1082	401	0.7627	0.8966	0.8243
FCTH	2846	13671	1240	1034	0.6965	0.7335	0.7145
LocalBinaryPatternsAndOpponent	2412	13349	1562	1468	0.6069	0.6216	0.6142
PHOG	2879	13806	1105	1001	0.7226	0.7420	0.7321
RankAndOpponent	2527	13553	1358	1353	0.6504	0.6512	0.6508
ColorLayout	2702	14018	893	1178	0.7515	0.6963	0.7229
CEDD	3705	13796	1115	175	0.7686	0.9548	0.8517
Gabor	1849	10643	4268	2031	0.3022	0.4765	0.3699
OpponentHistogram	2246	14157	754	1634	0.7486	0.5788	0.6529
EdgeHistogram	3548	13737	1174	332	0.7513	0.9144	0.8249
ScalableColor	3231	13684	1227	649	0.7247	0.8327	0.7750
Late Fusion	3710	13894	1017	170	0.7848	0.9561	0.8620

late fusion. Intuitively, we could simply combine the two image features, which score best individually. However, there is no guarantee that another combination would not be more balanced and therefore lead to better results. To test this, we therefore want to try all possible combinations of two image features. We can calculate the amount of possibilities using the factorial formula for binomial coefficients. We have a total of 18 image features available and we want to select two. This gives us a total of 153 possible combinations. Testing all these combinations with the complete database, as we have done for our previous experiment, would probably take many days. Therefore, we have to reduce the amount of samples we use. The ASU-Mayo Clinic database contains videos showing a polyp as well as videos not showing any polyp. For this test, we need video sequences, which contain video frames both with and without a polyp, to make sure our classifier can detect presence as well as absence of a polyp. We randomly selected two video sequences (wp_68 and wp_61) out of the ones that have both positive and negative video frames. We used a simple *Python* script to build all 153 different combinations and run the classifier accordingly. The best results for the first video sequence are presented in Table 6.9 and for the second video, in Table 6.10. Combining the results from these two videos, we found that the late fusion of the image features *JCD* and *Tamura* provides the best performance. *JCD* is in fact by itself already a combination of the image features *CEDD* [63] and *Fuzzy Color and Texture Histogram (FCTH)* [64]. The *LIRe* feature *Tamura* is a combination of three features, which correspond to human visual perception and are described in [65].

6.2.3.3 Classifier Performance Evaluation

Based on the evaluation of different combinations of image features in the previous paragraph, we have decided to use the image features *JCD* and *Tamura* for our further performance measurements. To assess the actual performance of our classifier using these two image features, we conducted a leave-one-out cross-validation with all available video sequences. With these settings, we achieve an average precision of 0.889, an average recall of 0.964 and an average F-score value of 0.916. The problem with this average calculation is that different video sequences contribute values based on different numbers of

Table 6.9: Top combinations of 2 image features for video wp_68, sorted by F-score.

	Feature	**TP**	**TN**	**FP**	**FN**	**Prec.**	**Recall**	**F1**
1	JCD & Tamura	172	51	20	14	0.895833	0.9247	0.9100
2	CEDD & Tamura	170	53	18	16	0.904255	0.9139	0.9090
3	ColorLayout & Tamura	165	59	12	21	0.932203	0.8870	0.9090
4	EdgeHist. & CEDD	175	41	30	11	0.853659	0.9408	0.8951
5	EdgeHist. & Tamura	175	41	30	11	0.853659	0.9408	0.8951
6	JCD & EdgeHist.	175	40	31	11	0.849515	0.9408	0.8928
7	EdgeHist. & PHOG	186	25	46	0	0.801724	1	0.8899
8	CEDD & PHOG	181	30	41	5	0.815315	0.9731	0.8872
9	JCD & PHOG	180	29	42	6	0.810811	0.9677	0.8823
10	PHOG & Rot.Inv.BP	184	23	48	2	0.793103	0.9892	0.8803
11	ColorLayout & PHOG	178	30	41	8	0.812785	0.9569	0.8790
12	FCTH & CEDD	178	30	41	8	0.812785	0.9569	0.8790
13	EdgeHist. & ScalableColor	185	21	50	1	0.787234	0.9946	0.8788
14	JpegCoeff.Hist. & Tamura	170	40	31	16	0.845771	0.9139	0.8785
15	OpponentHist. & Tamura	172	37	34	14	0.834951	0.9247	0.8775

Table 6.10: Top combinations of 2 image features for video wp_61, sorted by F-score.

	Feature	**TP**	**TN**	**FP**	**FN**	**Prec.**	**Recall**	**F1**
1	Rot.Inv.LBP & Tamura	162	22	153	0	0.5142	1	0.6792
2	PHOG & Tamura	161	23	152	1	0.5143	0.9938	0.6778
3	JpegCoeff.Hist. & Tamura	162	21	154	0	0.5126	1	0.6778
4	Gabor & Tamura	162	20	155	0	0.5110	1	0.6764
5	FuzzyColorHist. & Tamura	162	18	157	0	0.5078	1	0.6735
6	FuzzyOpp.Hist. & FuzzyColorHist.	160	17	158	2	0.5031	0.9876	0.6666
7	JCD & Opp.Hist.	135	67	108	27	0.5555	0.8333	0.6666
8	JointHist. & JpegCoeff.Hist.	162	12	163	0	0.4984	1	0.6652
9	ColorLayout & FuzzyColorHist.	162	11	164	0	0.4969	1	0.6639
10	FuzzyColorHist. & JointHist.	162	11	164	0	0.4969	1	0.6639
11	FuzzyOpp.Hist. & JointHist.	162	11	164	0	0.4969	1	0.6639
12	FuzzyOpp.Hist. & SimpleColorHist.	162	11	164	0	0.4969	1	0.6639
13	JointHist. & Rotat.Inv.LBP	162	11	164	0	0.4969	1	0.6639
14	JointHist. & SimpleColorHist.	162	11	164	0	0.4969	1	0.6639
15	FuzzyOpp.Hist. & Gabor	161	13	162	1	0.4984	0.9938	0.6639
16	JCD & JpegCoeff.Hist.	161	13	162	1	0.4984	0.9938	0.6639
17	CEDD & FuzzyColorHist.	159	17	158	3	0.5015	0.9814	0.6638
18	JpegCoeff.Hist. & Rot.Inv.LBP	152	31	144	10	0.5135	0.9382	0.6637
19	JCD & Tamura	162	10	165	0	0.4954	1	0.6625
20	CEDD & Tamura	162	10	165	0	0.4954	1	0.6625

video frames. If we weight the values contributed by every single video sequence with the amount of frames in the sequence, we even achieve an average precision of 0.939, an average recall of 0.985, an average F-score value of 0.957 and an average weighted F-score value of 0.929. The precise numbers are presented in Table 6.11.

Table 6.11: Performance evaluation by leave-one-out cross-validation for all available videos, using JCD and Tamura features.

Feature	TP	TN	FP	FN	Prec.	Recall	Acc.	F1	WF1
np_5	1	680	0	0	1	1	1	1	1
np_6	1	836	0	0	1	1	1	1	1
np_7	1	767	0	0	1	1	1	1	1
np_8	1	710	0	0	1	1	1	1	1
np_9	1	1841	0	0	1	1	1	1	1
np_10	1	1923	0	0	1	1	1	1	1
np_11	1	1548	0	0	1	1	1	1	1
np_12	1	1738	0	0	1	1	1	1	1
np_13	1	1800	0	0	1	1	1	1	1
np_14	1	1637	0	0	1	1	1	1	1
wp_2	140	9	20	70	0.875	0.6666	0.6234	0.7567	0.6851
wp_4	908	1	0	0	1	1	1	1	1
wp_24	310	68	127	12	0.7093	0.9627	0.7311	0.8168	0.6952
wp_49	421	12	62	4	0.8716	0.9905	0.8677	0.9273	0.8293
wp_52	688	101	284	31	0.7078	0.9568	0.7146	0.8137	0.6662
wp_61	162	10	165	0	0.4954	1	0.5103	0.6625	0.3746
wp_66	223	12	165	16	0.5747	0.9330	0.5649	0.7113	0.4584
wp_68	172	51	20	14	0.8958	0.9247	0.8677	0.9100	0.8658
wp_69	265	185	138	26	0.6575	0.9106	0.7328	0.7636	0.7264
wp_70	379	1	0	29	1	0.9289	0.9290	0.9631	0.9609
Average:					0.8894	0.9637	0.8771	0.9163	0.9162
Weighted Average:					0.9388	0.985	0.9370	0.9568	0.9286

6.2.3.4 Scalability Evaluation

One further requirement for our classifier is scalability. The idea is to use such a classifier to do mass screening of people for lesions in the colon. In this case, the video sequences would be recorded with camera capsules, which currently have an average frame rate of 5 FPS and a resolution of either $256x256$ or $512x512$ pixels. As the technology improves, it is likely that these specifications will increase to a regular video playback rate of about 25 FPS and a resolution of $1920x1080$. The videos from the ASU-Mayo Clinic polyp database already have this resolution. It is therefore our goal to process at least 25 FPS on average with this dataset.

There are several conditions that influence the processing speed, such as the performance of the computer, efficient use of the available resources, but also the size and the amount of features contained in the classifier index that is searched for classification. For any of the subsequent time measurements, we used the same computer (Mac Pro Mid 2010), as we have used for all previous experiments.

For our first experiment, we used a classifier index containing all available 18 image features for a total of $12,986$ video frames. But we only used the features *JCD* and *Tamura*, as we did for the previous measurements. We were able to process $1,924$ previously indexed frames in 462.604 seconds; this equates to 4.16 FPS. We have then created a classifier index for the same video frames, but only containing the two needed image features *JCD* and *Tamura*. With this classifier index, processing the same 1924 frames was completed in 105.978 seconds, which equates to 18.15 FPS. With the specified computer and the late fusion of *JCD* and *Tamura*, we are not able to reach the goal of 25 FPS. Further, the video decoding and the on-the-fly calculation of image features for the testing sequence is not even included yet. However, we have already invested a significant effort in parallelizing, profiling and optimizing our tools. To reach a higher frame rate, we can therefore choose less computationally expensive features, reduce the amount of features to a single one, or require a faster computer for

processing.

As a further experiment, we measured the time consumed for processing the same video frames using the same classifier index, but only considering the image feature *JCD*. With this configuration, the processing was completed in 53.851 seconds, which equates to a frame rate of 35.73 FPS. This would be clearly above the required 25 FPS, but would unfortunately not provide an optimal detection performance.

To find out how powerful a machine would have to be to process data in real-time, we have run some more tests on Amazon AWS EC2 instances. On a *c4.8xlarge* instance (Intel Xeon E5-2666 with 36 virtual CPUs), we were able to classify the $1,924$ frames with both features *JCD* and *Tamura* selected, in 29.377 seconds (65.5 FPS). When classifying data from a raw video file and therefore calculating the image features on the fly, the processing time increased to 39.599 seconds (48.6 FPS). When reading the data from a Windows media video (wmv) file, the processing time increased to 40.452 seconds (47.6 FPS). The *c4.8xlarge* instance is the most powerful instance offered by Amazon. We therefore conducted the same tests also on a less powerful *c4.4xlarge* instance (Intel Xeon E5-2666 with 16 virtual CPUs). Using this instance, we were able to process the indexed data in 60.19 seconds (31.97 FPS), the wmv file in 81.17 seconds (23.7 FPS) and the raw video file in 79.718 seconds (24.14 FPS). We therefore conclude that it is possible to process the data in real-time if up-to-date server hardware is available.

6.3 Summary

In this chapter, we have described the functionality we added to *TagAndTrack* to export data for machine learning. We have also described the steps to train *opencv_traincascade*-based classifiers with multiple datasets from *TagAndTrack* and have developed a separate *HOGTrainer* tool to allow the same for HOG-based classifiers. The trained classifiers with either approach can be used with *TagAndTrack* to run the detection. For *opencv_traincascade* based classifiers we have further implemented a separate optimized detector tool, which uses multiple threads to process multiple frames simultaneously. On our test computer, we can process upto 71 FPS at a resolution of $768x576$ pixels using this detector tool. This would therefore allow to run detection in real-time.

However, our experiments show that neither *opencv_traincascade*-based nor HOG-based classifiers are suitable for the purpose of detecting polyps in colonoscopy videos. The unknown orientation of a polyp in the video, the large variety of appearances and thereby the lack of strong common visual features makes it impossible to use these standard types of classifiers.

We have therefore conducted further experiments with a different approach using global image features, based on *LIRe*. We have implemented a solution for a search based classifier using previously created indices of global image features. This solution consists of an indexer and a classifier. The indexer is used to extract global image features from images that were exported together with the respective meta data from *TagAndTrack*; it stores these image features in *Lucene* indices. The classifier can use such indices as training data or as input data for classifying. The classifier implementation also contains benchmarking functionality, for benchmarking the classifier itself. Depending on the input source, the classifier will output an HTML file with a visible representation of the classification results, or a JSON file, which can be opened with *TagAndTrack* to visualize the results.

Our global image feature based indexer and classifier tools allowed us to experiment with several different types of global image features and combine results by late fusion. We have found that a combination of the image features *JCD* and *Tamura* led to the most promissing results. We have assessed the performance of our classifier using these image features by leave-one-out cross-validation for the whole ASU-Mayo Clinic polyp database. Our classifier achieves a weighted average precision of 0.9388 and a weighted average recall of 0.985. The classification process is rather computationally expensive. On our test computer, we managed to process 18.15 FPS at a resolution of $1920x1080$. However, we have shown that with modern server hardware, it is possible to process such videos in real-time.

With both recall and precision above 90% and the ability to process FullHD video in real-time, our classifier based on global image features clearly paves the way for automatic polyp detection in capsule endoscopy videos.

Chapter 7

Conclusion

7.1 Summary

In this thesis, we have investigated different technologies and solutions for automatically detecting lesions in colonoscopy videos. The goal was to automate as much of the screening process for colorectal cancer and its precursors as possible, to pave the way for a much broader screening process using camera capsules.

We have first studied related work that focused on similar problems or has already addressed relevant parts, such as generic image recognition, machine learning or detection of polyps using a different form of input data. We have then built tools to collect ground truth from hospitals or doctors. Such tools must be easy to use, require minimal training and must be easy to deploy and maintain. We have developed two different prototypes, one written in *Python* and *Qt* and the second one based on *HTML5*. The *HTML5*-based approach has shown to be easier to maintain and deploy, especially because no client side installation is necessary and because the processed videos are automatically uploaded to our server.

To make collecting of ground truth for machine learning more convenient and less time consuming, we have researched different approaches for tracking of previously annotated regions of interest in videos. We have implemented these tracking mechanisms in a native tool, written in *C++*, using *Qt* for the user interface and *OpenCV* for reading and processing the video data. This tool creates complete datasets that can be used for machine learning using *opencv_traincascade* or a separate *Histogram of Oriented Gradient* trainer. Further, it provides more fine-graned controls for the video playback and for creating annotations, mechanisms for simple machine learning experiments and several preprocessing steps, such as specular highlight filtering or border detection.

We have built classifiers and detector tools making use of these classifiers, using different technologies for detecting lesions in colonscopy videos. Our first approach was based on an off the shelf image recognition method using a Haar-feature-based classifier cascade, implemented with *OpenCV*. This approach has shown to be not suitable for the purpose of detecting lesions in the colon for several reasons. The training data for detecting colon polyps contains too little common characteristic image features for this approach to work. The unknown orientation of the lesions in the colon is a big problem in particular. Further, the training process is very time consuming and inefficient.

The second approach we researched is making use of a *Histogram of Oriented Gradients*-based classifier. This type of classifier has several advantages over the *OpenCV* traincascade one. It requires only very few samples and very little time to train a detector. However, also with this approach, the enormous variety of different appearances and orientations of lesions is a problem.

Our third and most promising approach is written in *Java* and makes use of *Lucene*, *LIRe* and *OpenCV*. This approach is based on indexing of global image features. We use *LIRe* to extract image features from training data and store the extracted information in *Lucene* indices. Our search based

classifier can extract the same image features for a video sequence to be classified, and classifies each video frame based on searching the most similar images from the provided classifier index, using a distance measure. We have evaluated this classifier using leave-one-out cross-validation with 20 different video sequences from the ASU-Mayo Clinic polyp database and have achieved very good performance. This approach is also easily expandable by adding more features or samples to the index, and the software we wrote can be used for other image classification tasks beyond the detection of colon polyps.

7.2 Main Contributions

In this thesis, we have built an entire set of tools and prototypes for experimenting with different machine learning approaches to detect lesions in colonoscopy videos. We have developed an HTML5-based tool for collecting video data from physicians, and to do an initial tagging of lesions. We have also developed a portable native application, which can be used to track the initially tagged regions automatically, to create a complete datasets for machine learning. We have built several preprocessing, filtering and visualization mechanisms into the same application. These tagging and tracking tools can be used for building training and testing datasets for machine learning, and they build an important part of an integrated machine learning and detection pipeline.

We have evaluated different machine learning approaches, and we have found that neither the standard implementations of *Histogram of Oriented Gradient*-based, nor *opencv_traincascade*-based classifiers are suitable for detecting colon polyps, because of the unknown orientation of a polyp and the large variety of appearances it can have.

Our third approach is a search-based classifier using a combination of the global image features *JCD* and *Tamura*. This classifier is efficient, robust, scalable and also provides very promissing results. We achieve a weighted average recall of 98.5%, a weighted average precision of 93.9%, a weighted average F-score of 0.92 and a weighted average weighted F-score of 0.93 on the full dataset of the ASU-Mayo Clinic polyp database. We have also shown that with recent server hardware, it is possible to process videos at FullHD resolution in real-time. Compared to the other machine learning approaches we have evaluated, this approach also requires significantly less time for training/indexing. We conclude that our global image features based classifier is a very promissing approach for automating the detection of colon polyps in capsule endoscopy videos. With an average recall and precision above 90% and the ability to process FullHD videos at regular playback speed, this paves the way for future mass screening for colon cancer and its precursors using capsule video endoscopy.

Parts of the software that we have developed have already been presented at the *ACM Multimedia System 2015* conference in Portland [28]. All the software developed for this thesis is available under the terms of the GNU General Public License version 3. The source code can be obtained from Bitbucket, and all the repositories are listed in Appendix A.

7.3 Future Work

With this thesis, we have built and tested several different approaches for detecting lesions in colonoscopy videos. The search-based classifier approach using global image features is clearly the most promising of our experiments. Further research is required to make this a production ready solution that incorporates adaptive learning and is easily usable as a service for hospitals.

It is also necessary to further reduce the processing time to allow using much larger datasets with a reasonable performance. The size of the classifier index directly affects the classification performance and the processing time for classification. It is therefore crucial to find the most optimal way to store and search indices efficiently. We have adopted *Lucene* indices, because this is what *LIRe* uses, and we have not yet researched any other solutions. *LIRe* also contains examples for a hashing indexing mechanism, which could potentially reduce the processing time for very big classifier indices.

We have only used video sequences from conventional colonoscopy for this thesis. It is therefore necessary to collect a reasonable amount of capsule endoscopy video sequences, re-evaluate the approach and verify that the same image features are an optimal choice with capsule endoscopy videos. Once an

optimal set of image features has been found, the whole infrastructure provided by *LIRe* and our tools might be too generic. At this point, it is worth considering to develop a new set of tools which is optimized to work only with this specific set of image features. Having defined a specific set of features, which could be used as a well defined feature vector for a SVM, we should re-evaluate if an SVM could handle the task in a more efficient way than a search based classifier.

Our solution currently acts as a binary classifier only. However, most of our software is written in a way that would allow to do classification into more than just two clusters. This could potentially allow a more precise diagnose of disease patterns. Of course, this approach can also be used beyond the scope of detecting lesions in colonoscopy videos. Various use cases, for example, in home- and industry-automation are plausible. The image features to be used most likely heavily depend on the actual use case. Finding the right image features for different use cases is another interesting future research topic.

For the part of our research where we are using global image features, we have neglected any preprocessing steps for the video frames. The preprocessing should be further researched to reduce the amount of redundant or interfering information stored in the classifier index. The information about which image features are relevant could potentially also be used to make camera capsules more efficient and record more relevant information.

Appendix A

Source Code

The source code for all the tools we have developed is licensed under GNU General Public License (GPL) version 3, and is available on Bitbucket.

The repository for the web based tagging tool can be found at:
`https://bitbucket.org/zenoalbisser/medicalannotationtool`

The repository for *TagAndTrack*, the tool for tracking image regions and creating datasets for training classifiers, can be found at:
`https://bitbucket.org/zenoalbisser/tagandtrack`

The repository for *OpenSea*, the global image features-based classifier, can be found at:
`https://bitbucket.org/mpg_projects/opensea`

Glossary

accuracy Accuracy measures the proportion of correct results (true positives and true negatives) out of the total number of items classified *(see also Section 6.2)*. 4, 6, 13, 17, 18, 21, 29, 54, 77, 78

Application Programming Interface An Application Programming Interface (API) is a specified interface that a library or a software program can expose to provide resources or functionality to another piece of software. 21

binary classifier A binary classifier is a classifier that can only make a binary decision. For any input data the classifier will either return true or false. 15, 19, 22

boilerplate In software engineering, boilerplate or boilerplate code is generic code that is often repeated in many places without significant alteration. Typical examples are methods for getting and setting instance variables, initialization code, include or import statements. 54, 64, 68

border detection For this thesis, we refer to border detection, as the mechanism to remove any regions along the edges of the video frame, which are irrelevant. 37, 43, 44, 46, 87

bounding box The bounding box usually refers to the minimum bounding box. This is the rectangle of minimal size that contains all the items in a predefined set of, for example, points or shapes in a two dimensional coordinate system. In computer graphics, bounding boxes are usually axis-aligned. 50, 51

camera capsule A camera capsule is a swallowable capsule that contains a camera, recording pictures while moving through the digestive tract. 3–5, 15, 17, 21, 25, 44, 45, 49, 85, 87, 89

capsule endoscopy Capsule endoscopy is a method for recording images of the digestive tract, using a camera capsule instead of a conventional colonoscope. 3–5, 9, 10, 14, 15, 17, 18, 20, 43, 47, 78, 86, 88

cascading classifier Cascading classifiers or a classifier cascade is the concept of concatenating several classifiers, using the output information from a classifier as additional input information for the next classifier in the cascade. 6, 49, 54, 60, 78

classifier A classifier is a tool or algorithm, that is capable of making classification decisions, based on characteristics of the input object (image). 5, 6, 9, 14, 18–20, 22, 41, 44–47, 49, 50, 53–55, 57, 60, 63, 65–67, 69, 71, 74, 77–83, 85–89

classifier index The classifier index is the index of global image features that the search-based classifier searches for the items most similar to the item currently being classified. 69, 73, 85, 86, 88, 89

colon cancer Colon cancer is also known as colorectal cancer, rectal cancer or bowel cancer. It is the development of an abnormal growth of cells in the colon or rectum. 1, 5, 9, 15, 16, 25, 88

colonoscopy Colonoscopy is the procedure of examining the large intestine (rectum and colon), usually, using a flexible tube endoscope. 1, 6, 12, 16, 17, 21, 25, 26, 30, 41, 47, 51, 66, 77, 86–89

Color and Edge Directivity Descriptor Color and Edge Directivity Descriptor (CEDD) is a global image feature. CEDD incorporates information about the color and the texture of the image in a histogram. 82

computed tomography Computed tomography is a procedure to create detailed pictures of inner body parts. It is using special x-ray equipment to capture the imaging data. 2

Computed Tomography Colonography Computed Tomography Colonography (CTC) is a non-invasive method using special x-ray equipment to produce images of the colon. 1

Computer Aided Diagnosis Computer Aided Diagnosis (CAD) are computer-based systems in medicine, which help doctors with the interpretation of medical imaging data. 4

Content Based Image Retrieval Content Based Image Retrieval (CBIR) is the concept of using computer vision techniques for searching digital image in a large database. The images are searched not by name or metadata, but by image features. 5

convex hull A convex hull in two dimensional space can be described as a set of points in a plane, where all the points are connected by a line without any intersections or any points visited more than once, and the line only ever bending the same direction (clockwise / counter clockwise).. 38, 43

dlib Dlib is a general purpose cross platform software library, written in C++. For this thesis, we use dlib for it's built-in machine learning functionality. 36, 60–65, 80

duodenum The duodenum is the first section of the small intestine. It is directly connected to the stomach and is responsible for most of the chemical digestion. 1

early fusion In machine learning, early fusion is a fusion concept, where extracted features are combined before the learning step. 74

edge detection Edge detection is a method to identify points in an image, where the intensity changes rapidly. Such points are usually organized in curves or line segments, and can for example be used for feature extraction. 37, 39, 40, 44, 46

endoluminal view A view from within. For example, the endoluminal view of the colon, is the view from within the colon. 21, 49

endoscope An endoscope is a device with an attached light that is used to look inside a body cavity or an organ . 1, 16, 18, 19, 37, 46, 78

endoscopy Endoscopy is the procedure of examining an inner body part by using an endoscope. 3, 5, 10–13, 16, 18, 20, 22, 37, 43, 45

False Negatives False Negatives (FN) are the items that a classifier incorrectly detectes as negative items. 77, 78, 82–84

False Positives False Positives (FP) are the items that a classifier incorrectly detectes as positive items. 77, 78, 82–84

feature extraction In machine learning and image processing, feature extraction is the process of extracting data and building informative, derived values (features) from an input image. 4, 17, 19, 39

F-measure . 77, *see* F-score

frame reduction For this thesis, frame reduction is the process of removing the amount of frames required to process further. This may include the removal of irrelevant or blurry video frames. 14

F-score F-score (also F-measure or F1-score) is the harmonic mean of precision and recall. It is thereby a combination of these two measures in a single number *(see also Section 6.2).* 11, 14, 77, 82–84, 88

Fuzzy Color and Texture Histogram Fuzzy Color and Texture Histogram (FCTH) is an image feature combining color and texture information in a single histogram. An FCTH feature descriptor is limited to 72 bytes per image, making the descriptor suitable for large image databases. 83

gastrointestinal tract The gastrointestinal tract is the organ system in the human body that is responsible for consuming and digesting food, absorbing nutrients and expelling waste. It consists of stomach and intestines . 1, 3

Gaussian blur . *see* Gaussian filtering

Gaussian filtering In image processing, Gaussian filtering (also called Gaussian blur or Gaussian smoothing) is the process of convolving each point in the input image with a Gaussian kernel. A Gaussian function thereby defines the weight for each pixel within the kernel. The closer the pixel to the center of the kernel, the higher it's impact. 39, 41, 42, 45

global image feature A global image feature is a feature descriptor that describes the whole image, as opposed to only a sub region of the image. 6, 21, 66, 67, 69, 82, 86–89, *see* Joint Descriptor for CEDD and FCTH, Fuzzy Color and Texture Histogram, Color and Edge Directivity Descriptor & Tamura

ground truth Ground truth data is the data, which we use as the definition of correctness or truth. This data is the foundation of the classifiers we are training, as the classifier will learn to distinguish items based on the features extracted from, and the predefined classification of this data. 7, 30, 31, 63, 65, 82, 87

H.264 H.264 is a video coding format that is, due to its quality and compression, widely used for distribution of video content. 28, 33

haar-feature Haar-features are digital image features that were first described by Viola and Jones [48] as a machine learning approach for visual object detection, capable of processing images extremely rapidly and achieving high detection rates. The name is derived from "Haar Basis function", which was proposed by Alfréd Haar. 6, 54, 80, 87

haustral fold Haustral folds are folds of the mucosa within the colon. 21

Histogram of Oriented Gradients Histogram of Oriented Gradients (HOG) is an image feature descriptor used in computer vision for detecting objects. HOG counts occurrences of gradient orientation in sub-portions of an image. 6

Hu-Moment In image processing, a moment is a defined weighted value, describing a certain property or geometric interpretation of an image. Hu-Moments are a set of seven moments, which are invariant to rotation, translation and scale and were first presented in [53]. 40

hyper threading Hyper-Threading is a proprietary implementation of simultaneous multithreading by Intel. A processor capable of Hyper-Threading consists of two logical processors per core. Each of these logical processors has its own processor architectural state, can be individually interrupted or halted, but shares the same physical core with a second logical core. 57, 68

hyperplane A hyperplane is a subspace of n-1 dimensions, if the surrounding space is n dimensional. With regards to an SVM, the feature vectors build an n dimensional space and the SVM attempts to separate the feature vectors by a hyperplane, which then is n-1 dimensional. 4

input index . 71, *see* testing index

intestine The intestines are the tube running from the stomach to the anus. The intestines are responsible for the absorption of water and nutrients. 6, 10, 49, 80, *see* small intestine & large intestine

isotropic volume A volume of uniform physical properties in all directions. The physical properties are relevant to reconstruct the three dimensional volumes in the context of virtual colonoscopy. 4, 22

Java Servlet A Java Servlet is a Java program that is executed on the server, in response to a client requesting a certain resource or sending a certain request. Commonly this is used to implement applications hosted on a web server. 28, 33

Java Servlet container A Java Servlet container is a web server component that interacts with Java Servlets and manages their lifecycle and access permissions. 28

Joint Descriptor for CEDD and FCTH Joint Descriptor for CEDD and FCTH (JCD) is a global image feature descriptor. It is a combination of the descriptors CEDD and FCTH. 82

kernel A kernel is a square matrix used for image processing such as blurring, sharpening or edge-detection. It is usually used with a sliding window approach, to modify the value of the kernel's center pixel based on the surrounding pixels within the kernel. 13, 17, 19, 39, 42

lambda In software development, a lambda, lambda function is an anonymous function. The definition of such a function is not bound to an identifier. In most programming languages that support this kind of functions, lambdas can be defined inline within other functions and can be easily passed as arguments to other functions. 72

laparoscopic surgery A laparoscopic surgery is a minimally invasive surgery, also called a keyhole surgery, that is performed through a small incision. 10

large intestine The large intestine, also called colon, is the final section of the digestive system in the human body. 49

late fusion In machine learning, late fusion is a fusion concept, where extracted features are processed separately and only the results are weighted and combined. 74, 75, 82, 83, 85, 86

leave-one-out cross-validation Leave-one-out cross-validation is a technique to assess the generalization of a predictive model. In our case, it describes the process where the training and testing datasets are rotated, leaving out a single different non-overlapping item or portion for testing and using the remaining items for training. This process is repeated until every item or portion was used for testing exactly once. 6, 78, 79, 82, 83, 86, 88

lesion A lesion is an irregularity in the tissue, usually caused by trauma or disease. In the context of this thesis, we specifically refer to lesions in the colon, such as polyps. 4, 13, 21, 26, 36, 47, 66

Local Binary Pattern Local Binary Pattern (LBP) is a type of feature vector, used for machine learning. It is based on the concept of splitting the image into cells and comparing each pixel within a cell to it's neighbors in a predefined order. 5

Magnetic Resonance Imaging Magnetic Resonance Imaging (MRI) is a technology that uses a powerful magnetic field and radio frequency pulses to capture pictures of inner body structures. 4

max heap A max heap is a tree based data structure that satifies the heap property. In a max heap, the heap property is satisfied if any parent is always greater than or equal to any of its children. In contrast to a binary tree, an order of the direct children of a node do not need to be maintained. 75

metadata Metadata is data that describes another type of data. In photography, metadata contained in an image can for example include exposure time, aperture or GPS-data. 21, 37, 44, 52

MJPEG Motion JPEG (MJPEG) is a very basic video compression format, where each video frame is compressed separately as a JPEG image. 33

mucosa The mucosa or mucous membrane is the lining in inner body cavities that are exposed to the external environment. For this thesis, we are specifically referring to the mucosa, as the innermost layer of the gastrointestinal tract, surrounding the lumen within the tube. 10, 15, 16, 18, 41, 42, 44–46, 51, 62

native Software is running native, if it is compiled to support a specific platform and operating system and does not use virtualization or emulation. 12, 32, 33, 36, 71, 87, 88

Native Client Google Native Client (NaCl) is a technology for running native code in a web browser. The native code is running in a sandbox for security reasons and communicates with the web browser through a plugin API. NaCl is developed by Google, and is currently available with Chrome and Chromium. 32

neural network Artificial neural networks are statistical learning algorithms, which are inspired by natural neural networks, such as the human brain. They consist of a usually large number of simple processing units, which are simulating neurons, and weighted connections in between those units [20]. 4, 22

neuron A neuron or nerve cell is a cell that can be electrically stimulated and transmits information through electrical and chemical signals. 4

open source Open source software is software that is available under a free license and can be changed, shared and modified by anyone. 6, 17, 66

OpenCV OpenCV (Open Source Computer Vision) is a library for computer vision that was originally developed by Intel and is available under an open-source BSD license. 26, 27, 32, 33, 37–40, 44, 49, 53, 54, 60, 71, 72, 79, 80, 87

OpenGL ES 2 OpenGL for Embedded Systems version 2 is a subset of the OpenGL API. OpenGL is a standardized API for 2D and 3D computer graphics. 32

overfitting Overfitting is the phenomenon of a classifier encoding random noise, caused by training with too few training samples or too long training cycles . 9, 17, 22, 47, 78

Pepper Plugin API Pepper Plugin API, also Pepper API, is a cross-platform API for plugins for web browsers. It is currently an experimental feature of Chromium and Google Chrome. 32

polyp A polyp is an abnormal protrusion from the tissue of the mucosa. It is often considered a precursor of colon cancer. 1, 15, 16, 18, 19, 21, 22, 40, 41, 47, 49, 62, 63, 66, 69, 78, 79, 81, 83, 86, 88

Portable Native Client Portable Native Client (PNaCl) is an addition to NaCl, which allows to compile and run portable code in a web browser. The code is written in C and C++ and is compiled to an LLVM-bytecode intermediate representation, instead of an architecture-specific representation. 32

precision Precision measures the fraction of the detected-positive instances, which are true-positive. *(see also Section 6.2)*. 6, 10, 11, 14, 77, 82–84, 86, 88

priority queue A priority queue is an abstract data type that assures that the element with the highest priority is always served / de-queued first. 76

PyQt PyQt is a product developed by Riverbank Computing Ltd. It provides Python bindings for Qt. 27

Python Python is an interpreted cross-platform programming language. 26, 27, 30, 83, 87

Qt Qt is a cross platform development toolkit, written in C++. It is available under an open-source license. In this thesis, we are using Qt for developing user interfaces and inter-thread communication. 26, 27, 33, 34, 87

recall Recall is the fraction of all true-positive instances, which are also detected positive *(see also Section 6.2)*. 6, 10, 11, 14, 19, 77, 82–84, 86, 88

Receiver Operating Characteristic The Receiver Operating Characteristic (ROC), is an illustration of the performance of a classifier. It plots the true positive rate against the false positive rate. 18

rubber-band-selection Rubber-band-selection is a selection method, where a resizable rectangle is used to select an area on screen. 13, 28, 40

Runnable In Java, Runnable is an interface that can be implemented by a class. Instances of a class implementing this interface can be passed directly to a Java Thread instance for execution within that same thread. 72

sensitivity Sensitivity, also called true positive rate, measures the proportion of correctly detected positive items among all positive items. 15, 18–20

shape detection Shape detection is a method to decide, if a given shape matches any other shape in a previously learned set of shapes. The shapes are collected by edge detection. 37, 39, 40, 44, 46

Single Instruction Multiple Data Single Instruction Multiple Data (SIMD) is a class of computers that allow data parallelization. Modern CPUs usually have support for a SIMD instruction set extension, allowing to process multiple data objects simultaneously with a single instructions. 32

small intestine The small intestine is the part of the intestine in between stomach and large intestine. 1, 3

specificity Specificity, also called true negative rate, measures the proportion of correctly detected negative items among all negative items. 15, 18, 19

specular highlight A reflection; a bright spot or region on a shiny object, caused by a light source. 37, 41, 42, 44–46, 62, 87

supervised learning Supervised learning is a machine learning approach, where each input sample for the training is labeled with the desired response. 47

Support Vector Machine Support vector machines are machine learning methods, searching for a function that defines a hyperplane, separating the different classes of data points with a maximal margin. 4

tagging Tagging is the process of annotating a video, by using rubber-band-selection to select regions of interest and potentially manually describing or classifying them. 6, 26–36, 55, 88

Tamura Tamura is a combination of three image features, which correspond to human visual perception. 21, 82, 83, 85, 86, 88

tensor of inertia A tensor of inertia is a 3x3 matrix of moments of inertia, which defines a spatial movement. 15

terminal ileum The terminal ileum is the end of the small intestine towards the large intestine. 1

testing dataset The testing dataset is the dataset, we are using for testing a classifier. 19, 25, 46, 79, 88

testing index The testing index, or input index, is the index of global image features that the search-based classifier uses as input data. It classifies all the items in this index, by searching the items most similar in the classifier index. This is used for benchmarking of the search-based classifier. 69

thyroid The thyroid or tyroid gland is a gland found in the human neck below the ”Adam’s apple”. 10

tracking With tracking we refer to the process of following a region of interest in previous/subsequent video frames, adjusting the size or the position of the region. 4, 6, 9, 13, 14, 27, 29–36, 50, 55, 87, 88

training dataset The training dataset is the dataset, we are using for training a classifier. 9, 19, 31, 42, 47, 71, 79, 80

transcoding Transcoding is the direct conversion from one encoding to another. For this thesis, this specifically refers to the conversion of a video file from one video coding format to another. 28, 29

True Negatives True Negatives (TN) are the items that a classifier correctly detectes as negative items. 77, 78, 82–84

True Positives True Positives (TP) are the items that a classifier correctly detectes as positive items. 77, 78, 82–84

unsupervised learning Unsupervised learning is a machine learning approach, where the input samples are not labeled with any desired response. It can be described as the attempt to find a hidden structure in the unlabeled data, since there is no reward or error signal given to the learner to evaluate its own results. 14, 47

video segmentation Video segmentation is the process of dividing a video in smaller meaningful segments. 9, 11

video shot a shot is described as sequence of frames limited by two shot boundaries or end of the video sequence. 11

viewport In computer graphics, a viewport is the viewing region. It contains and limits the view on a potentially larger scene. 29, 37, 38

Virtual Colonoscopy . 15, 21, 22, 47, 49, *see* Computed Tomography Colonography

wavelet transform Wavelet transform is a concept, similar to fourier transform, but instead of decomposing a signal into sine and cosine waves, wavelet transform decomposes a signal into multiple non-overlapping wave-like oscillations, which begin and end at zero, and are therefore limited in time. 17

WebSocket WebSocket is a protocol, standardized by the IETF, to allow full-duplex communication channels over a single TCP connection, innitiated via HTTP. 33

XMLHttpRequest XMLHttpRequest is a JavaScript object designed by Microsoft, now being standardized in the W3C. It allows requesting and retrieval of data from a server without doing a full page reload in a web browser. 33

Acronyms

ACM Association for Computing Machinery. 5, 6

API Application Programming Interface. 21, 27, 31–33, 60, 65, *see* Application Programming Interface

CAD Computer Aided Diagnosis. 4, 21, 22, *see* Computer Aided Diagnosis

CBIR Content Based Image Retrieval. 5, *see* Content Based Image Retrieval

CEDD Color and Edge Directivity Descriptor. 82, 83, *see* Color and Edge Directivity Descriptor

CT Computed Tomography. 4, 15, 16, 21

CTC Computed Tomography Colonography. 1, 4, 21, 47, *see* Computed Tomography Colonography

DOM Document Object Model. 65

FCTH Fuzzy Color and Texture Histogram. 83, *see* Fuzzy Color and Texture Histogram

FN False Negatives. 77, 78, 82, *see* False Negatives

FP False Positives. 77, 78, 82, *see* False Positives

FPS frames per second. 3, 4, 16, 33, 55, 57, 85, 86

GCD Grand Central Dispatch. 60

GNU GNU's Not Unix. 6

GPL GNU General Public License. 6, 67, 88

HOG Histogram of Oriented Gradients. 6, 19, 20, 36, 60–65, 80, 81, 86, *see* Histogram of Oriented Gradients

HTML HyperText Markup Language. 6, 71

HTML5 HyperText Markup Language version 5. 6

JCD Joint Descriptor for CEDD and FCTH. 82, 83, 85, 86, 88, *see* Joint Descriptor for CEDD and FCTH

JPEG Joint Photographic Experts Group. 4

JSON JavaScript Object Notation. 29, 33, 34, 55, 61, 69, 71, 77, 86

JVM Java Virtual Machine. 68, 71

LBP Local Binary Pattern. 5, 17, 18, *see* Local Binary Pattern

LIRe Lucene Image Retrieval. 21, 66, 67, 73, 83, 87–89

MIT Massachusetts Institute of Technology. 19

MRI Magnetic Resonance Imaging. 4, *see* Magnetic Resonance Imaging

NaCl Google Native Client. 32, *see* Native Client

Pepper API Pepper API. 32, *see* Pepper Plugin API

PNaCl Portable Native Client. 32, *see* Portable Native Client

PPAPI Pepper Plugin API. 32, *see* Pepper Plugin API

ROC Receiver Operating Characteristic. 18, 19, *see* Receiver Operating Characteristic

ROI Region of Interest. v, 26, 28–30, 35, 50, 51, 61

SIMD Single Instruction Multiple Data. 32, *see* Single Instruction Multiple Data

SVM Support Vector Machine. 4, 5, 13, 17–20, 22, 47, 60, 89, *see* Support Vector Machine

TBB Intel Threading Building Blocks. 54

TLD Tracking-Learning-Detection. 14, 33

TN True Negatives. 77, 78, 82, *see* True Negatives

TP True Positives. 77, 78, 82, *see* True Positives

UI User Interface. 26, 28, 65

WCE Wireless Capsule Endoscopy. 3

XML Extensible Markup Language. 12, 54, 61, 64, 65

Bibliography

[1] International Agency for Research on Cancer. *World Cancer Report 2014 (International Agency for Research on Cancer)*, chapter The global and regional burden of cancer. World Health Organization, 2014.

[2] S. J. Stryker, B. G. Wolff, C. E. Culp, S. D. Libbe, D. M. Ilstrup, and R. L. MacCarty. Natural history of untreated colonic polyps. *Gastroenterology*, 93(5):1009–1013, Nov 1987.

[3] Centers for Disease Control, Prevention (CDC, et al. Vital signs: colorectal cancer screening test use–united states, 2012. *MMWR. Morbidity and mortality weekly report*, 62(44):881, 2013.

[4] Michael PIGNONE and Harold C SOX. Screening for colorectal cancer: Us preventive services task force recommendation statement. *Annals of internal medicine*, 149(9), 2008.

[5] Gondal G., Grotmol T., Hofstad B., Bretthauer M., Eide T. J., and Hoff G. The norwegian colorectal cancer prevention (norccap) screening study. *Scandinavian Journal of Gastroenterology*, 38(6):635–642, 2003.

[6] Elisabeth Rosenthal. The $2.7 trillion medical bill. `http://www.nytimes.com/2013/06/02/health/colonoscopies-explain-why-us-leads-the-world-in-health-expenditures.html`, June 2013. Accessed: 2015-04-15.

[7] PA Cataldo. Colonoscopy without sedation - A viable alternative. *DISEASES OF THE COLON & RECTUM*, 39(3):257–261, MAR 1996.

[8] Espen Thiis-Evensen, Geir S. Hoff, Jostein Sauar, and Morten H. Vatn. Patient tolerance of colonoscopy without sedation during screening examination for colorectal polyps. *Gastrointestinal Endoscopy*, 52(5):606 – 610, 2000.

[9] Linda Villarosa. Done right, colonoscopy takes time, study finds.(health&fitness), 2006.

[10] Jay P Heiken, Christine M Peterson, and Christine O Menias. Virtual colonoscopy for colorectal cancer screening: current status: Wednesday 5 october 2005, 14:00–16:00. *Cancer Imaging*, 5(Spec No A):S133–S139, 2005.

[11] Luís A Alexandre, Joao Casteleiro, and Nuno Nobreinst. Polyp detection in endoscopic video using svms. In *Knowledge Discovery in Databases: PKDD 2007*, pages 358–365. Springer, 2007.

[12] Baopu Li, M.Q. Meng, and Chao Hu. Motion analysis for capsule endoscopy video segmentation. In *Automation and Logistics (ICAL), 2011 IEEE International Conference on*, pages 46–51, Aug 2011.

[13] Mingda Zhou, Guanqun Bao, Yishuang Geng, B. Alkandari, and Xiaoxi Li. Polyp detection and radius measurement in small intestine using video capsule endoscopy. In *Biomedical Engineering and Informatics (BMEI), 2014 7th International Conference on*, pages 237–241, Oct 2014.

[14] Nikos Deligiannis, Frederik Verbist, Athanassios Iossifides, Jürgen Slowack, Rik Van de Walle, Peter Schelkens, and Adrian Munteanu. Wyner-ziv video coding for wireless lightweight multimedia applications. *J Wireless Com Network*, 2012(1):1–20, 2012.

[15] Janne Näppi and Hiroyuki Yoshida. Feature-guided analysis for reduction of false positives in cad of polyps for computed tomographic colonography. *Medical Physics*, 30(7):1592–1601, 2003.

[16] Anna K. Jerebko, James D. Malley, Marek Franaszek, and Ronald M. Summers. Multiple neural network classification scheme for detection of colonic polyps in {CT} colonography data sets. *Academic Radiology*, 10(2):154 – 160, 2003.

[17] Gabriel Kiss, Johan Van Cleynenbreugel, Maarten Thomeer, Paul Suetens, and Guy Marchal. Computer-aided diagnosis in virtual colonography via combination of surface normal and sphere fitting methods. *European Radiology*, 12(1):77–81, 2002.

[18] D.S. Paik, C.F. Beaulieu, G.D. Rubin, B. Acar, Jr. Jeffrey, B., J. Yee, J. Dey, and S. Napel. Surface normal overlap: a computer-aided detection algorithm with application to colonic polyps and lung nodules in helical ct. *Medical Imaging, IEEE Transactions on*, 23(6):661–675, June 2004.

[19] James J. Perumpillichira, Hiroyuki Yoshida, and Dushyant V. Sahani. Computer-aided detection for virtual colonoscopy. *Cancer Imaging*, 5(5):11–16, 4 2005.

[20] David Kriesel. A brief introduction to neural networks. `http://www.dkriesel.com/_media/science/neuronalenetze-en-zeta2-2col-dkrieselcom.pdf`, 2007.

[21] Kristin P. Bennett and Erin J. Bredensteiner. Duality and geometry in svm classifiers. In *Proceedings of the Seventeenth International Conference on Machine Learning*, ICML '00, pages 57–64, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[22] Jorge Bernal, Javier Sánchez, and Fernando Vilarino. Towards automatic polyp detection with a polyp appearance model. *Pattern Recognition*, 45(9):3166–3182, 2012.

[23] Sae Hwang, JungHwan Oh, W. Tavanapong, J. Wong, and P.C. de Groen. Polyp detection in colonoscopy video using elliptical shape feature. In *Image Processing, 2007. ICIP 2007. IEEE International Conference on*, volume 2, pages II – 465–II – 468, Sept 2007.

[24] Danyu Liu, Yu Cao, Kihwan Kim, Sean Stanek, Bancha Doungratanaex-Chai, Kungen Lin, Wallapak Tavanapong, Johnny S. Wong, Jung-Hwan Oh, and Piet C. de Groen. Arthemis: Annotation software in an integrated capturing and analysis system for colonoscopy. *Computer Methods and Programs in Biomedicine*, 88(2):152–163, 2007.

[25] Carl Vondrick, Donald Patterson, and Deva Ramanan. Efficiently scaling up crowdsourced video annotation. *International Journal of Computer Vision*, pages 1–21. 10.1007/s11263-012-0564-1.

[26] Michael Kipp. Anvil - a generic annotation tool for multimodal dialogue. In Paul Dalsgaard, Børge Lindberg, Henrik Benner, and Zheng-Hua Tan, editors, *INTERSPEECH*, pages 1367–1370. ISCA, 2001.

[27] D. E. Comer, David Gries, Michael C. Mulder, Allen Tucker, A. Joe Turner, and Paul R. Young. Computing as a discipline. *Commun. ACM*, 32(1):9–23, January 1989.

[28] Zeno Albisser, Michael Riegler, Pål Halvorsen, Jiang Zhou, Carsten Griwodz, Ilangko Balasingham, and Cathal Gurrin. Expert driven semi-supervised elucidation tool for medical endoscopic videos. In *Proceedings of the 6th ACM Multimedia Systems Conference*, MMSys '15, pages 73–76, New York, NY, USA, 2015. ACM.

[29] Christopher M Bishop. *Information science and statistics*. Springer, New York, 2006.

[30] Eero P. Simoncelli, B. Jahne, H. Haussecker, and P. Geissler. Bayesian Multi-Scale differential optical flow. 1999.

[31] Manfred Jürgen Primus, Klaus Schoeffmann, and Laszlo Böszörmenyi. Segmentation of recorded endoscopic videos by detecting significant motion changes. In Laszlo Czuni, editor, *11th International Workshop on Content-Based Multimedia Indexing*, pages 223–228, Los Alamitos, CA, USA, jun 2013. IEEE Computer Society.

[32] Manfred Del Fabro and Laszlo Böszörmenyi. State-of-the-art and future challenges in video scene detection: a survey. *Multimedia Systems*, 19(5):427–454, 2013.

[33] Michael Riegler, Mathias Lux, Vincent Charvillat, Axel Carlier, Raynor Vliegendhart, and Martha Larson. Videojot: A multifunctional video annotation tool. In *Proceedings of International Conference on Multimedia Retrieval*, ICMR '14, pages 534:534–534:537, New York, NY, USA, 2014. ACM.

[34] Raynor Vliegendhart, Martha Larson, and Alan Hanjalic. LikeLines: collecting timecode-level feedback for web videos through user interactions. In *Proceedings of ACM MM '12*, pages 1271–1272. ACM, 2012.

[35] Mathias Lux and Michael Riegler. Annotation of endoscopic videos on mobile devices: A bottom-up approach. In *Proceedings of the 4th ACM Multimedia Systems Conference*, MMSys '13, pages 141–145, New York, NY, USA, 2013. ACM.

[36] S. Hare, A. Saffari, and P.H.S. Torr. Struck: Structured output tracking with kernels. In *Computer Vision (ICCV), 2011 IEEE International Conference on*, pages 263–270, Nov 2011.

[37] Z Kalal, K Mikolajczyk, and J Matas. Tracking-learning-detection. *Pattern Analysis and Machine Intelligence*, 2012.

[38] S. Tsevas, D.K. Iakovidis, D. Maroulis, and E. Pavlakis. Automatic frame reduction of wireless capsule endoscopy video. In *BioInformatics and BioEngineering, 2008. BIBE 2008. 8th IEEE International Conference on*, pages 1–6, Oct 2008.

[39] A.V. Mamonov, I.N. Figueiredo, P.N. Figueiredo, and Y.-H.R. Tsai. Automated polyp detection in colon capsule endoscopy. *Medical Imaging, IEEE Transactions on*, 33(7):1488–1502, July 2014.

[40] DongHo Hong, Wallapak Tavanapong, Johnny Wong, JungHwan Oh, and Piet C. de Groen. 3d reconstruction of virtual colon structures from colonoscopy images. *Computerized Medical Imaging and Graphics*, 38(1):22–33, 2015/04/19 2013.

[41] Stefan Ameling, Stephan Wirth, Dietrich Paulus, Gerard Lacey, and Fernando Vilarino. Texture-based polyp detection in colonoscopy. pages 346–350, 2009.

[42] B. Giritharan, Xiaohui Yuan, Jianguo Liu, B. Buckles, JungHwan Oh, and Shou Jiang Tang. Bleeding detection from capsule endoscopy videos. In *Proceedings of the EMBS '08*, pages 4780–4783. EMBS, Aug 2008.

[43] Baopu Li and M.Q.-H. Meng. Tumor recognition in wireless capsule endoscopy images using textural features and svm-based feature selection. *Information Technology in Biomedicine, IEEE Transactions on*, 16(3):323–329, May 2012.

[44] Chih-Chung Chang and Chih-Jen Lin. Libsvm: A library for support vector machines. *ACM Trans. Intell. Syst. Technol.*, 2(3):27:1–27:27, May 2011.

[45] Michael Häfner, Michael Liedlgruber, Andreas Uhl, Andreas Vécsei, and Friedrich Wrba. Color treatment in endoscopic image classification using multi-scale local color vector patterns. *Medical Image Analysis*, 16(1):75–86, 2012.

[46] M. Hafner, A. Gangl, M. Liedlgruber, A. Uhl, A. Vecsei, and F. Wrba. Pit pattern classification using extended local binary patterns. In *Information Technology and Applications in Biomedicine, 2009. ITAB 2009. 9th International Conference on*, pages 1–4, Nov 2009.

[47] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 1, pages 886–893 vol. 1, June 2005.

[48] Paul Viola and Michael Jones. Rapid object detection using a boosted cascade of simple features. *2013 IEEE Conference on Computer Vision and Pattern Recognition*, 1:511, 2001.

[49] Yoav Freund and Robert E Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences*, 55(1):119–139, 1997.

[50] Mathias Lux and Savvas A. Chatzichristofis. Lire: Lucene image retrieval: An extensible java cbir library. In *Proceedings of the 16th ACM International Conference on Multimedia*, MM '08, pages 1085–1088, New York, NY, USA, 2008. ACM.

[51] The JSON data interchange format. Technical Report Standard ECMA-404 1st Edition / October 2013, ECMA, October 2013.

[52] John Canny. A computational approach to edge detection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, PAMI-8(6):679–698, Nov 1986.

[53] Ming-Kuei Hu. Visual pattern recognition by moment invariants. *Information Theory, IRE Transactions on*, 8(2):179–187, February 1962.

[54] Vincent Christlein, Christian Riess, Elli Angelopoulou, Georgios Evangelopoulos, and Ioannis Kakadiaris. The impact of specular highlights on 3d-2d face recognition, 2013.

[55] Harvard University. Colon polyps. 01 2011. Copyright - Copyright © 2013 by Harvard University. All rights reserved. HHP/HMS content licensing handled by Belvoir Media Group; Last updated - 2013-06-24.

[56] Pedro Domingos. A few useful things to know about machine learning. *Commun. ACM*, 55(10):78–87, October 2012.

[57] Alvin L. Watne. Colon polyps. *Journal of Surgical Oncology*, 66(3):207–214, 1997.

[58] Mathias Lux. Lire: Open source image retrieval in java. In *Proceedings of the 21st ACM International Conference on Multimedia*, pages 843–846. ACM, 2013.

[59] The Apache Software Foundation. Apache lucene - index file formats. `https://lucene.apache.org/core/3_0_3/fileformats.html#Definitions`, 2013. Accessed: 2015-07-29.

[60] Cees G. M. Snoek, Marcel Worring, and Arnold W. M. Smeulders. Early versus late fusion in semantic video analysis. In *Proceedings of the 13th Annual ACM International Conference on Multimedia*, MULTIMEDIA '05, pages 399–402, New York, NY, USA, 2005. ACM.

[61] Bradley Efron and Robert Tibshirani. Improvements on cross-validation: The .632+ bootstrap method. *Journal of the American Statistical Association*, 92(438):pp. 548–560, 1997.

[62] David Martin Ward Powers. Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation. *International Journal of Machine Learning Technology*, 2(1):37–63, 2011.

[63] Savvas A. Chatzichristofis and Yiannis S. Boutalis. Cedd: Color and edge directivity descriptor: A compact descriptor for image indexing and retrieval. In *Proceedings of the 6th International Conference on Computer Vision Systems*, ICVS'08, pages 312–322, Berlin, Heidelberg, 2008. Springer-Verlag.

[64] S.A. Chatzichristofis and Y.S. Boutalis. Fcth: Fuzzy color and texture histogram - a low level feature for accurate image retrieval. In *Image Analysis for Multimedia Interactive Services, 2008. WIAMIS '08. Ninth International Workshop on*, pages 191–196, May 2008.

[65] Hideyuki Tamura, Shunji Mori, and Takashi Yamawaki. Textural features corresponding to visual perception. *Systems, Man and Cybernetics, IEEE Transactions on*, 8(6):460–473, June 1978.