Fachbereich 4: Informatik

Ein Klassifikationsframework in Qt/C++

Studienarbeit
im Studiengang Computervisualistik

vorgelegt von

Miriam Grunwald

Betreuer: Dipl.-Inform. Stephan Wirth, Institut für Computervisualistik, Fachbereich Informatik, Universität Koblenz-Landau
Erstgutachter: Dipl.-Inform. Stephan Wirth, Institut für Computervisualistik, Fachbereich Informatik, Universität Koblenz-Landau
Zweitgutachter: Prof. Dr.-Ing. Dietrich Paulus, Institut für Computervisualistik, Fachbereich Informatik, Universität Koblenz-Landau

Koblenz, im Oktober 2009

Kurzfassung

Der Prozess der Mustererkennung gliedert sich in mehrere Teilschritte, wobei letztlich aus unbekannten Datensätzen Muster erkannt und automatisch in Kategorien eingeordnet werden sollen. Dafür werden häufig Klassifikatoren verwendet, die in einer Lernphase anhand von bekannten Testdaten trainiert werden.

Viele bestehenden Softwarelösungen bieten Hilfsmittel für spezielle Mustererkennungsaufgaben an, aber decken nur selten den gesamten Lernprozess ab.

Im Rahmen dieser Studienarbeit wurde aus diesem Grund ein Framework entwickelt, welches allgemeine Aufgaben eines Klassifikationssystems für Bilddaten als eigenständige Komponenten integriert. Es ist schnittstellenorientiert, leicht erweiterbar und bietet eine graphische Benutzeroberfläche.

Abstract

The process of pattern recognition is divided into several sub-steps where ultimately patterns on unknown data samples are expected to be detected and automatically labeled. Classifiers are most commonly used for this and are being trained on known test data samples.

Many existing software applications provide tools for selected tasks in pattern recognition but not for all steps in the training phase.

Therefore in this work a framework was developed which implements common tasks of image classification systems as individual components. It is plugin aware, easily extensible and provides a graphical user interface.

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Die Vereinbarung der Arbeitsgruppe für Studien- und Abschlussarbeiten habe ich gelesen und anerkannt, insbesondere die Regelung des Nutzungsrechts.

Mit der Einstellung dieser Arbeit in die Bibliothek bin ich einverstanden.	ja ☒	nein ☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	ja ☒	nein ☐

Koblenz, den 29. Oktober 2009

Inhaltsverzeichnis

1 Einleitung **11**
- 1.1 Problemstellung und Motivation 11
- 1.2 Kapitelvorschau 12

2 Mustererkennung **13**
- 2.1 Einzelschritte 14
  - 2.1.1 Datenerfassung 14
  - 2.1.2 Vorverarbeitung 14
  - 2.1.3 Segmentierung 14
  - 2.1.4 Merkmalsextraktion 15
  - 2.1.5 Klassifikation 15
- 2.2 Software für Mustererkennung 16

3 Framework zur Mustererkennung **17**
- 3.1 Anforderungen 17
- 3.2 Technische Grundlagen 18
- 3.3 Benutzeroberfläche und Funktionalität 18
- 3.4 Implementation 20

4 Zusammenfassung und Ausblick **27**

Abbildungsverzeichnis

2.1 Klassifikationspipeline 14

3.1 Benutzeroberfläche der Anwendung 19
3.2 Preprocessing-Widget 21
3.3 UML-Klassendiagramm für das entwickelte Framework 25
3.4 Ausschnitt aus UML-Klassendiagramm: *Feature extraction* 26

Kapitel 1

Einleitung

1.1 Problemstellung und Motivation

Die Mustererkennung beschäftigt sich damit, Ähnlichkeiten und Muster in Datensätzen zu finden, diese als einen Regelsatz oder über mathematische Funktionen zu beschreiben und anschließend auf unbekannte Datensätze anzuwenden.

Dafür gibt es unterschiedliche Verfahren und abhängig vom konkreten Anwendungsgebiet verschiedene Herangehensweisen. Ein universelles Klassifikationswerkzeug, das für jegliche Art von Daten geeignet ist, scheint kaum realisierbar. Doch generell lassen sich allgemeine Klassifikationsaufgaben wie beispielsweise Vorverarbeitung, Segmentierung oder Merkmalsextraktion ausmachen, die grundsätzlich bei den meisten Klassifikationsproblemen zur Lösung erforderlich sind. Jedoch benötigt jeder individuelle Anwendungsfall passende Algorithmen und Hilfsmittel.

Werden Klassifikatoren verwendet, können diese in einer Lernphase trainiert werden. Für den zügigen Ablauf eines Lernvorgangs bietet sich eine Softwarelösung an, die die gängigen Teilschritte in einer Applikation vereint und so ein benutzerfreundliches und schnelles Arbeiten ermöglicht und gleichzeitig individuelle Gestaltungsoptionen besitzt.

Das in dieser Arbeit vorgestellte Framework verfolgt diesen Ansatz und bietet ein pluginfähiges Klassifikationssystem mit einer intuitiven Bedienoberfläche. Es ist ein allgemein gehaltenes Framework, das genügend Schnittstellen bietet, die wesentlichen Klassifikationsaufgaben auf den jeweiligen Einzelfall anzupassen. Daher ist es nicht notwendig, mehrere verschiedene Softwarelösungen für ein Klassifikationsproblem verwenden zu müssen.

Das entwickelte Programm stellt somit die Möglichkeit zur Verfügung, Klassifikatoren zu trainieren und Aufgaben wie Vorverarbeitung, Segmentierung und Merkmalsextraktion mit eigenen Algorithmen auszustatten.

Es verwendet die Qt-Bibliothek[1] und greift auf viele vorteilhafte Techniken und Funktionen, sowie vorgefertigte Programmierbausteine zurück. Dies ist gleichzeitig ein Vorteil für den Nutzer, der auf leichte Art und Weise eigenen Programmcode integrieren und die Applikation nach seinen Anforderungen erweitern kann.

Ein Anstoß zur Verwirklichung des Projekts ist eine in Zusammenarbeit mit der Universität Koblenz-Landau entwickelte Studienarbeit, die Verfahren zur Detektion von Polypen und Tumoren in Koloskopievideos [Ame08] testete. Es stellt ein Beispielszenario dar, für das das entwickelte Framework ein effizientes Werkzeug und eine große Arbeitserleichterung darstellen kann.

1.2 Kapitelvorschau

Im folgenden Kapitel sollen in einer kurzen Einführung die Grundzüge der Mustererkennung wiedergegeben werden, um die grundlegenden Begrifflichkeiten und Arbeitsschritte zu erläutern. Darüberhinaus wird kurz dargelegt, welche Softwarelösungen in diesem Kontext bereits existieren und in wie weit sie die Teilbereiche der Mustererkennung abdecken.

Als nächstes folgt ein ausführliches Kapitel über das in der Studienarbeit entwickelte Klassifikationsframework. Es wird neben den Anforderungen, denen es genügen soll, vor allem auf den graphischen Aufbau und die Implementation eingegangen.

Im Anschluss daran werden in einer Diskussion Vor- und Nachteile sowie Probleme bezüglich der Entwicklung aufgegriffen, bevor die Studienarbeit mit einem Ausblick über mögliche Verbesserungen, Erweiterungen und Anmerkungen abschließt.

[1] http://qt.nokia.com/

Kapitel 2

Mustererkennung

Bei der Mustererkennung werden Strukturen und Zusammenhänge in Datensätzen gesucht - es entspricht also genau dem, was beim Menschen Wahrnehmungsvermögen genannt wird und größtes Vorbild für die Mustererkennung in der Informatik ist. Die für den Menschen essentiell wichtige Fähigkeit, den ständig - besonders optisch - wahrgenommenen Informationsfluss zu erfassen, relevante Daten herauszufiltern, umzuwandeln und letztlich abzuspeichern um ein späteres Abgleichen und somit ein Wiedererkennen zu ermöglichen, gilt es maschinell nachzuahmen [DHS01].

Der Prozess der Mustererkennung geht in mehreren Schritten vonstatten, die sich je nach Anwendungsgebiet etwas unterscheiden können. Im Allgemeinen werden die zu analysierenden Daten erfasst, vorverarbeitet und zur Vereinfachung in kleinere Datenobjekte (Segmente) unterteilt. Anschließend werden relevante Merkmale extrahiert, die Datenmenge reduziert, mit bereits vorhandenen Beschreibungen (Modellen) abgeglichen und einer Klasse zugeordnet (vgl. Abb. 2.1).

Es gibt strukturelle, statistische und syntaktische Ansätze bei der Mustererkennung, auf die hier jedoch nicht im Detail eingegangen werden soll.

Die Mustererkennung stellt ein äußerst hilfreiches Instrument zum Beispiel bei der automatischen Sprach- und Texterkennung, Fingerabdruckidentifikation, DNS-Analyse, in der Robotik, Künstlichen Intelligenz und in vielen Bereichen der Industrie dar.

Eine ausführlichere Beschreibung der Teilschritte bezüglich der Mustererkennung auf Bilddaten folgt im nächsten Abschnitt.

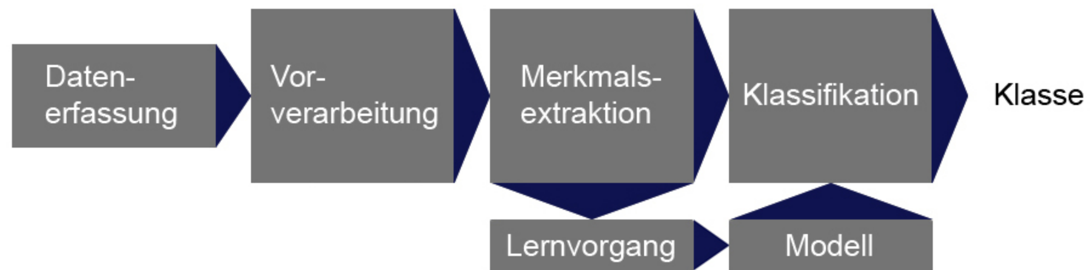

Abbildung 2.1: Klassifikationspipeline

2.1 Einzelschritte

2.1.1 Datenerfassung

Erster Schritt der Mustererkennung ist das Erfassen der Daten des zu klassifizierenden Objekts, bei Bildern beispielsweise mit Hilfe einer Kamera. Bei der Abbildung der eingehenden Signale auf dem Kamerasensor, der anschließenden Digitalisierung und Quantisierung, kann es zu ungewollten Bildstörungen kommen, wie Bildrauschen, Helligkeitsunterschieden, Verzerrungen, Unschärfe, Farbfehlern und ähnlichem [CS05].

2.1.2 Vorverarbeitung

Um oben genannte Störungen zu reduzieren, bietet sich im nächsten Schritt eine Vorverarbeitung an. Das Bild wird mit Vorverarbeitungsmethoden bearbeitet, welche Bildpunktoperationen (z.B. LUT-Operationen, Grauwerttransformationen, Histogrammebnung), lokale Nachbarschaftsoperationen (Filter) oder globale Nachbarschaftsoperationen (geometrische Transformationen) sein können. Gleichzeitig wird versucht, den Aufwand bei der anschließenden Segmentierung zu verringern. Der Vorverarbeitungsschritt ist im Allgemeinen anwendungsunabhängig und behält den vorhandenen Datentyp bei, d.h. es findet keine Umwandlung in ein anderes Datenformat statt [PH03].

2.1.3 Segmentierung

Bei der Segmentierung wird das Bild erstmals inhaltlich analysiert und in verschiedene Bereiche unterteilt, die Homogenitätskriterien, wie zum Beispiel Farbe, Grauwert und/oder Textur genügen. Dies kann u.a. anhand kantenbasierter (z.B. Wasserscheidentransformation), punktorientierter (z.B. Schwellwertverfahren), regionenbasierter (z.B. Region Growing, Split and Merge, CSC) oder modellbasierter

(z.B. Houghtransformation, Template Matching) Verfahren geschehen. Auch dieser Schritt ist nicht von dem jeweiligen Anwendungsgebiet abhängig, vorhandene Informationen über Bildinhalte können jedoch hilfreich bei der Parameterwahl für die Algorithmen sein [PH03].

2.1.4 Merkmalsextraktion

Auf jedes Segment werden anschließend Merkmalsextraktionsmethoden angewandt, d.h. Algorithmen ausgeführt, die aus den Objekten Informationen auslesen, anhand derer dann Aussagen über Inhalt oder Semantik der Bilder gemacht werden können. Merkmale können mit visuellen Eigenschaften wie Form, Farbe, Lage etc. einhergehen oder durch mathematische Berechnungen bestimmt werden.

Ein Merkmal kann redundant sein, falls es sich zu einem anderen Merkmal proportional verhält [DHS01]. Es erfolgt eine Merkmalsreduktion, falls irrelevante Merkmale vorhanden sind. Extrahierte Merkmale der Anzahl m werden in einem m-dimensionalen Vektor, dem so genannten *Merkmalsvektor*, abgespeichert und haben in der Regel eine geringere Dimension als das vorverarbeitete und segmentierte Bildsignal. Wichtig zu beachten ist, dass die Annahme, mehr Merkmale, d.h. eine höhere Dimension, führe zu genaueren Ergebnissen bei der Klassifikation und Erkennung von Bildinhalten, generell nicht bestätigt werden kann [PH03, Bel61].

Ein Merkmalsvektor beschreibt die Position des Objekts im m-dimensionalen Merkmalsraum. Die Merkmalsvektoren, die zu ähnlichen Objekten gehören, bilden so genannte *Cluster* (Punktwolken).

2.1.5 Klassifikation

Ziel der Klassifikation ist es, die Bilddaten geeigneten Klassen zuzuordnen. Die durch die Merkmalsextraktion entstandenen Cluster, die jeweils eine Klasse bilden, gilt es zu erkennen und geeignete Klassengrenzen zwischen ihnen zu ziehen. Es gibt numerische und syntaktische Klassifikationsalgorithmen, die Verwendung finden, je nachdem ob es sich bei den Merkmalen um reelle Zahlen, Vektoren und dergleichen handelt, also als mathematische Funktion beschrieben werden können, oder ob es Symbole und Symbolketten sind, die die Struktur der Daten mit Hilfe formaler Sprachen beschreiben.

Es gibt zwei Strategien, das *überwachte Lernen* und das *unüberwachte Lernen*, die beim Trainieren des Klassifikators angewandt werden können.

Überwachtes Lernen

Falls die Klassen, denen die Daten zugeordnet werden sollen, bekannt sind, kann das überwachte Lernen angewandt werden. Dabei wird ein Teil aus der Datenbasis

als Trainingsdatensatz ausgewählt, bei dem zu jedem Objekt die zugehörige Klasse festgelegt wird. Mit diesem Trainingsdatensatz wird dann ein Modell (z.B. eine Menge von Regeln) erstellt, das zu einem gegebenen Merkmalsvektor die zugehörige Klasse bestimmen kann.

Unüberwachtes Lernen

Unüberwachtes Lernen ist dann sinnvoll, wenn keine Trainingsdaten vorhanden sind und man nicht weiß, wieviele Klassen es gibt. Dann wird versucht, aus der Merkmalsverteilung im m-dimensionalen Raum die Cluster abzuschätzen (Clusteranalyse), d.h. aufgrund von bestimmten Kriterien (z.B. räumliche Nähe) zu einer Klasse zusammenzufassen.

2.2 Software für Mustererkennung

Da es reichliche Anwendungsgebiete für Mustererkennung gibt, gibt es auch zahlreiche Softwareanwendungen dafür, einige wenige seien hier genannt.

Libraries wie zum Beispiel OpenCV[1] oder LIBSVM[2] können in Softwarelösungen integriert werden, unterstützen einige Aufgaben der Mustererkennung und stellen unzählige Algorithmen zur Verfügung.

SVMLight[3] oder Toolboxes für Matlab wie etwa PRTools[4] sind weitere hilfreiche Komponenten, bestimmte Klassifikationsaufgaben zu bewältigen. Allerdings sind Vorverarbeitung, Segmentierung und Merkmalsextraktion oftmals nicht Teil der Softwarelösungen und müssen vorher anderweitig durchgeführt werden.

Eine sehr umfangreiche Software bietet Weka[5]. Sie wird auch mit einer ansprechenden graphischen Benutzeroberfläche angeboten und stellt viele Hilfsmittel für Klassifikation, Regression, Clustering, Assoziationsanalyse und Visualisierung bereit.

---

[1] http://opencv.willowgarage.com/wiki/

[2] http://www.csie.ntu.edu.tw/~cjlin/libsvm/

[3] http://svmlight.joachims.org/

[4] http://www.prtools.org/

[5] http://www.cs.waikato.ac.nz/ml/weka/

Kapitel 3

Framework zur Mustererkennung

Das im Rahmen der Studienarbeit entwickelte Framework zur Mustererkennung wird in diesem Kapitel ausführlich vorgestellt.

Angefangen bei den technischen Daten und einer kurzen Skizzierung der zum Verständnis relevanten, wesentlichen Grundzügen der verwendeten Entwicklungsumgebung, wird der Aufbau des realisierten Frameworks erläutert. Ferner wird anhand einer detaillierteren Beschreibung eines Moduls die Implementation deutlich gemacht.

3.1 Anforderungen

Ziel für die Umsetzung des Frameworks war, das Programm so flexibel und anpassungsfähig wie möglich zu gestalten. Daher ist es schnittstellenorientiert und modulhaft aufgebaut, um so dem Nutzer die Möglichkeit zu geben, das Framework individuell an seine Anforderungen anzupassen.

Im Rahmen der Studienarbeit wurde die Realisierung jedoch auf Bilddaten als zu klassifizierenden Eingangsdatensatz reduziert und die Module dahin gehend angepasst.

Darüber hinaus ist lediglich die Lernphase umgesetzt worden, das heißt, dass bislang kein Modul eingebettet wurde, welches es dem Nutzer ermöglicht, einen Klassifikationsalgorithmus auf unbekannte Datensätze anzuwenden.

Es sind teilweise Beispielimplementationen im Prototypen vorhanden, um die Funktionsweise der Anwendung besser vorführen zu können. Dennoch muss vor der konkreten Inbetriebnahme des Programms für jedes der fünf Module eigener Programmcode generiert und integriert werden, was jedoch je nach Anwendungsfall mit mehr oder weniger großem Aufwand zu bewerkstelligen sein dürfte.

3.2 Technische Grundlagen

Entwickelt wurde in C++ im Rahmen der plattformübergreifenden IDE Qt Creator 1.1.0, die Qt 4.5.1 verwendet[1]. Als Compiler fungierte GNU gcc 4.3.2.

Die Vorteile bei der Verwendung von Qt liegen neben der umfangreichen Qt-Bibliothek, bei der Bereitstellung eines benutzerfreundlichen GUI-Designers und einer ausführlichen Dokumentation.

Darüberhinaus ist in Qt das Signal-Slot-Konzept umgesetzt, welches eine Alternative zu Rückruffunktionen darstellt und bei der Realisierung des Frameworks im Rahmen dieser Studienarbeit von großem Nutzen war. Das Prinzip dabei ist, dass Programmobjekte miteinander kommunizieren können, indem sie Signale senden und über Slots Signale empfangen können. Signale werden also an ein oder mehrere Slots gekoppelt und aufgrund eines eintretenden Ereignisses gesendet, wodurch anschließend die Slotfunktion aufgerufen wird.

Eine weitere Charakteristik von Qt ist das Verwenden von sogenannten *Widgets* für die graphische Benutzeroberfläche. Ein Widget ist ein graphischer Baustein, der ein eigenständiges Fenster oder auch zum Beispiel ein Button und somit Teil eines anderen Widgets sein kann. In letzterem Fall hat das Widget einen *parent*, also ein übergeordnetes Widget, wodurch die Lebensdauer und Sichtbarkeit vom Elternwidget abhängig ist. Bei einem Widget ohne *parent* hingegen handelt es sich um ein *top-level widget* oder auch Fenster und muss somit explizit gelöscht werden, sobald es in der Anwendung nicht mehr benötigt wird[2].

3.3 Benutzeroberfläche und Funktionalität

Für die graphische Umsetzung erschien eine Karteireiterstruktur am geeignetsten, die Abfolge der Teilschritte eines Klassifikationssystems zu repräsentieren. So enthält das Hauptanwendungsfenster für jedes der fünf Module einen Karteireiter, wie in Abb. 3.1 zu sehen ist.

Zwischen den Tabs und somit den implementierten Modulen *Load image*, *Preprocessing*, *Segmentation*, *Feature extraction* und *Save samples* kann mit Hilfe der sich unten rechts befindlichen *Back*- und *Next*-Buttons gewechselt werden.

In jedem Karteireiter befindet sich oben rechts eine Input-/Output-Ampelanzeige. Jedes Modul benötigt einen Input, der be- oder verarbeitet wird und dann an das nächste Modul weitergeleitet wird. Die Ampeln sollen eine optische Hilfestellung sein um festzustellen, ob und wann ein Input im Modul angekommen beziehungsweise Output generiert und weitergeschickt wurde. Sind Eingabedaten beim Modul

[1] `http://qt.nokia.com/`

[2] `http://doc.trolltech.com/4.5/index.html`

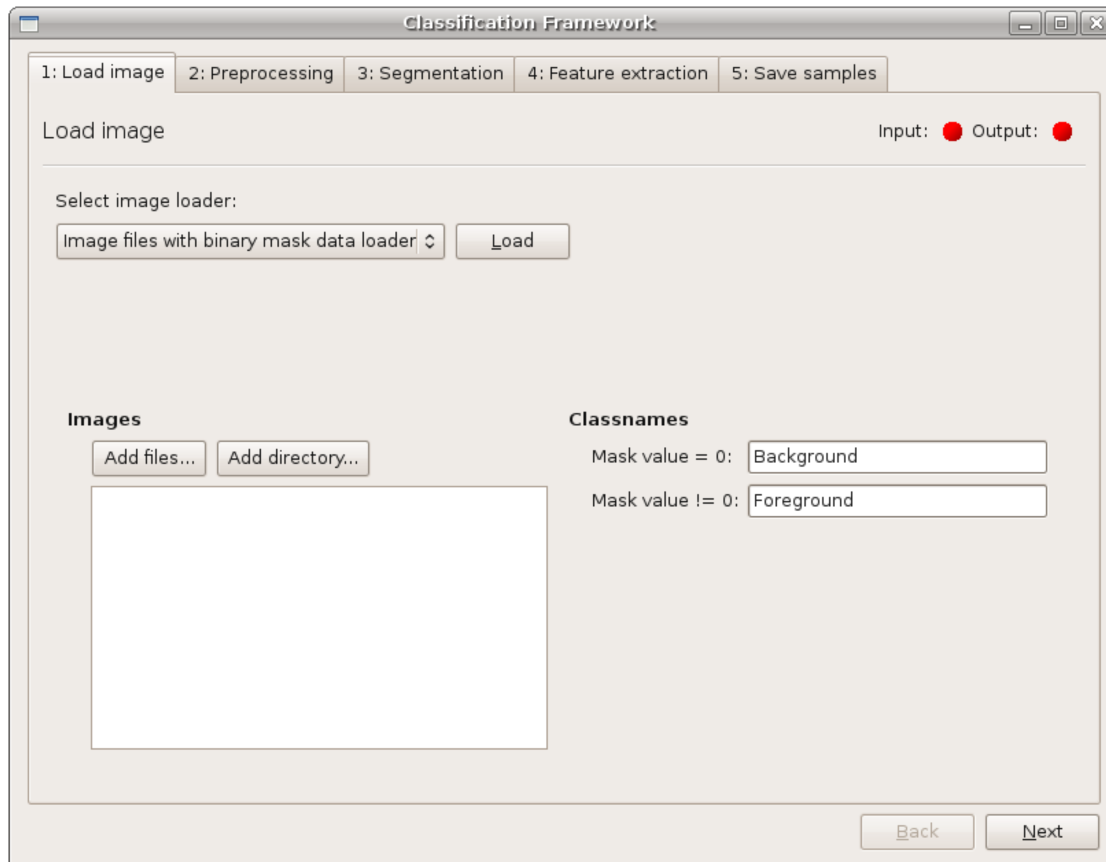

Abbildung 3.1: Benutzeroberfläche der Anwendung

eingetroffen, wechselt die Inputanzeige von rot auf grün, analog verhält es sich bei der Outputanzeige, wenn Ausgangsdaten generiert wurden.

Der Inhalt der jeweiligen Karteireiter ist abhängig von der Implementation des Nutzers. Grundsätzlich wurde für jedes Modul ein Widget erstellt, das im entsprechenden Karteireiter eingefügt wurde. In jedem dieser Widgets wiederum befindet sich mindestens ein Dropdownmenü und ein Button, der entsprechend seiner Funktionalität beschriftet ist, beim *Load image*-Modul ist er beispielsweise mit „Load“ betitelt. Funktion davon ist, dass bei Programmstart das Dateisystem nach Plugins für jedes der fünf Module durchsucht wird, die dann gegebenenfalls in das jeweilige Dropdownmenü geladen werden. Hat der Nutzer passende Plugins programmiert und in den richtigen Systempfad kopiert, werden diese nun im entsprechenden Dropdownmenü mit Namen angezeigt. Da es für die Funktionstüchtigkeit der Anwendung notwendig ist, ein sogenanntes *Settingswidget* zu implementieren, wird dieses unmittelbar nach dem erfolgreichen Laden der Plugins im Modulwidget unterhalb der Dropdownmenübox angezeigt. Die Be- oder Verarbeitung der Daten

findet nun im jeweiligen Settingswidget statt, mit Hilfe der Methoden die vom Benutzer implementiert wurden.

Beim Klick auf den bereitgestellten Button, also „Load" beim *Load image*-Modul, „Preprocess" beim *Preprocessing*-Modul, „Segment" beim *Segmentation*-Modul usw. wird die im Plugin implementierte Funktion aufgerufen und der Output erzeugt, der gleichermaßen den Input für das folgende Modul darstellt. Wird also beispielsweise der „Load"-Button im *Load image*-Modul angeklickt, wird die implementierte *load*-Funktion aufgerufen, die Daten werden geladen - falls es intern zu keinen Fehlern kommt, was jedoch mit einer Fehlernachricht angezeigt würde - und an das *Preprocessing*-Modul übergeben. Dort stehen sie nun der Vorverarbeitung zur Verfügung. Gleichzeitig springt die Outputanzeige des *Load image*-Moduls auf grün, genauso wie die Inputanzeige des *Preprocessing*-Moduls.

Analog verhält es sich bei den restlichen Modulen. Lediglich beim *Preprocessing*- und beim *Feature extraction*-Modul befindet sich unterhalb des Dropdownmenüs und dem „Preprocess"- bzw. „Extract"-Button ein weiteres Feld, in dem die geladenen Vorverarbeitungs-/Merkmalsextrahierungsmethoden aufgelistet werden können. Dies ist sinnvoll, da meist mehrere Methoden nacheinander auf einen Datensatz angewandt werden. Wird eine dieser Methode ausgewählt, sprich angeklickt, erscheint rechts daneben das zugehörige Settingswidget, über das die Methode konfiguriert werden kann. Über Drag-and-Drop kann die Reihenfolge geändert und über den „Remove"-Button die Methode gegebenenfalls wieder aus der Liste entfernt werden. Wurden Einstellungen für alle aufgelisteten Methoden vorgenommen und der entsprechende Button angeklickt, werden diese Methoden von oben nach unten der Reihe nach auf die Eingangsdaten angewendet (vgl. Abb. 3.2).

3.4 Implementation

Generell lassen sich die Quelldateien semantisch in mehrere Bereiche unterteilen. Zum einen gibt es die Klasse `Cf` (für classification framework) mit zugehöriger GUI-Datei, von der aus alle Modulklassen koordiniert und der Datenfluss in der Anwendung gesteuert wird. Sie bildet das Hauptanwendungsfenster und wird von der main-Methode aufgerufen. Darüber hinaus werden hier die Signale für die Ein- und Ausgabe mit den Slots verknüpft und die Modulwidgets in die Karteireiter eingebunden.

Es gibt fünf Modulklassen, `LoadImage`, `Preprocessing`, `Segmentation`, `FeatureExtraction` und `SaveSamples`. Sie haben jeweils eine zugehörige GUI-Datei und eine entsprechende Schnittstellenklasse. Letztere muss für jedes Modul vom Benutzer implementiert werden.

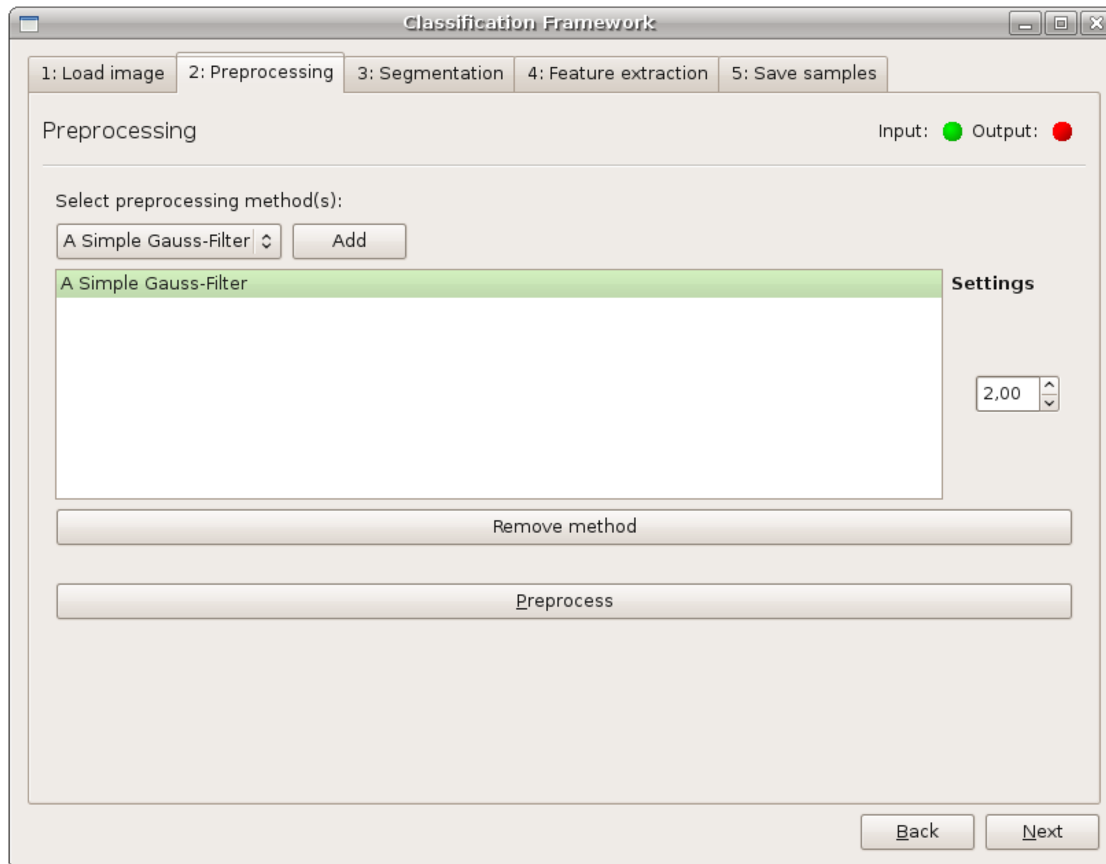

Abbildung 3.2: Preprocessing-Widget

In jedem Modulwidget ist das *Icons*-Widget eingebettet, welches die Eingabe/Ausgabe-Icons anzeigt. Diese werden vom zugeordneten Modul über Signale und Slots angesprochen.

Des weiteren gibt es Coredateien, die für den Datenaustausch zwischen den Modulen passende Datentypenklassen zur Verfügung stellen. Dazu gehören die Klassen `TaggedImage`, `Mask`, `Segment`, `SegmentRegion`, `Feature` und `Sample`.

Von der Klasse `SegmentRegion` muss vor Benutzung entsprechend abgeleitet werden, im Prototypen wird allerdings schon die Beispielklasse `AABBRegion` (für Axis Aligned Bounding Box) zur Verfügung gestellt.

Außerdem müssen zu den Trainingsbilddatensätzen passende Maskenbilder geladen werden, damit die Klassenzugehörigkeit festgelegt ist.

Alle im Framework verwendeten Klassen und ihre Beziehungen untereinander werden in Abb. 3.3 als UML-Klassendiagramm[3] veranschaulicht.

[3]Erstellt mit Visual Paradigm, `http://www.visual-paradigm.com/`

Um den Aufbau und die Realisierung besser nachvollziehen zu können, sollen die Schnittstellen kurz vorgestellt und anschließend ein Modul herausgegriffen und anhand dessen die Umsetzung näher erläutert werden.

Die fünf Schnittstellenklassen `DataLoaderInterface`, `PreprocessingMethodInterface`, `SegmentationMethodInterface`, `FeatureExtractionMethodInterface` und `SampleSaverInterface` sind analog aufgebaut. Sie haben eine `getSettingsWidget()`- und `getName()`-Methode, die das vom Nutzer implementierte Settingswidget bzw. den Namen des Plugins zurückgibt. Darüber hinaus gibt es jeweils eine weitere virtuelle Methode, die je nach Modultyp unterschiedliche Aufgaben erfüllt. So gibt es beim `DataLoaderInterface` die Methode `load(...)`, die die Bilddaten mit zugehörigen Masken in die Anwendung lädt. Beim `PreprocessingMethodInterface`, `SegmentationMethodInterface` und `FeatureExtractionMethodInterface` heißen diese Methoden `preprocess(...)`, `segment(...)` bzw. `extract(...)` und üben eine Vorverarbeitung, Segmentierung bzw. Merkmalsextraktion auf den Daten aus. Für das `SampleSaverInterface` muss die Methode `save(...)` implementiert werden, damit die erstellten Samples abgespeichert werden können.

Im Folgenden die genaue Implementation des Feature-Extraction-Moduls, dessen Aufbau in Abb. 3.4 ersichtlich ist.

Zunächst soll auf die Schnittstellenklasse `FeatureExtractionMethodInterface` eingegangen werden. Die Klasse enthält neben Qt-spezifischen Befehlen lediglich einen leeren Konstruktor, einen virtuellen Destruktor, sowie folgende drei virtuelle Methoden:

```
1 virtual Feature extract(const QString& imageFileName,
2         SegmentRegion* segmentRegion, int* ok) = 0;
3
4 virtual QWidget* getSettingsWidget(QWidget* parentWidget) = 0;
5
6 virtual QString getName() const = 0;
```

Alle drei Methoden müssen vom Nutzer implementiert werden. Die `extract(...)`-Methode bekommt den Dateinamen eines Bildes sowie eine Region vom Typ `SegmentRegion` übergeben und gibt ein `Feature` zurück. Auf den Parameter `ok` wird später genauer eingegangen. Die Methoden `getSettingsWidget(...)` und `getName()` geben das implementierte Settingswidget bzw. den Namen des Plugins zurück.

Wie im Abschnitt „Benutzeroberfläche und Funktionalität“ schon erläutert, werden bei Programmstart zu den Modulen passende Plugins gesucht und gegebenenfalls in die Anwendung geladen. Dies geschieht im Konstruktor der jeweiligen Modulklasse über den Aufruf der Funktion `loadPlugins()`. Diese Funktion durchsucht Unterordner im Installationspfad auf Plugins und gibt die Dateinamen von gefundenen Plugins an die Funktion `populateComboBox()` weiter. Dort werden die Plugins daraufhin untersucht,

ob sie zur Schnittstelle des Moduls passen. Falls dem so ist, wird der Dateiname in das Dropdownmenü eingefügt.

Über den „Add“-Button werden die ausgewählten Methoden in die sich darunter befindliche Liste eingefügt. Intern wird bei jedem Anklicken vom „Add“ ein Eintrag in einer *Map* gemacht, wodurch jede hinzugefügte Methode indiziert wird und somit die Methodenliste kontrolliert werden kann. Beim Klicken des „Remove“-Buttons wird die Methode nicht nur graphisch aus der Liste, sondern auch intern aus der *Map* entfernt.

Die Funktion `showWidgetForItem(QListWidgetItem* item)` ist dafür zuständig, dass das jeweils für die in der Liste markierte Methode zugehörige Settingswidget erscheint. Auch hier wird die Methodenmap benutzt um die selektierte Methode zu bestimmen.

Im Segmentierungsschritt werden die Daten in Segmente unterteilt, d.h. im *Segmentation*-Modul werden die erstellten Segmente jeden Datenobjekts in einem Vektor von Segmentpointern abgelegt und anschließend als Output an das *Feature extraction*-Modul weitergereicht. Die Klasse `FeatureExtraction` besitzt daher den Slot `setInput(Segment[])`, welcher den Output des *Segmentation*-Moduls empfängt.

Sind vom Nutzer alle Einstellungen getätigt worden, werden mit Drücken des „Extract“-Buttons die Methoden auf den Datensatz angewandt. Es wird die Methode `generateOutput()` aufgerufen, wo nun bei erfolgreichem Verlauf ein *SampleSet* erstellt wird:

```
1 //! Generate output and emit signal
2 void FeatureExtraction::generateOutput()
3 {
4     //! Output sample set:
5     QVector<Sample*> sampleSet;
6
7     //! Created sample:
8     Sample* sample;
9
10    Feature feature;
11
12    bool everythingFine = true;
13    for (int i = 0; i < m_input.size()
14                 && everythingFine == true; ++i)
15    {
16        //! Input segment:
17        Segment* inputSegment = m_input.at(i);
18
19        for(int j = 0; j < m_ui->comboBoxPlugins->count()
20                         && everythingFine == true; ++j)
21        {
22            //! Variable to check whether the method 'extract'
23            //! can handle the file type:
24            int ok;
```

```
25
26                 m_ui->comboBoxPlugins->setCurrentIndex(j);
27
28                 feature = m_FeatureExtractionMap.value(m_ui->
29                     comboBoxPlugins->currentIndex())->
30                     extract(inputSegment->getImageFileName(),
31                     inputSegment->getSegmentRegion(), &ok);
32
33                 if(ok < 0)
34                 {
35                     ... //! Return error message box
36                     ... //! and set everythingFine = false;
37                 }
38                 else
39                 {
40                     sample->addFeature(feature);
41                 }
42             }
43             sampleSet.append(sample);
44         }
45         emit outputReady();
46         emit outputGenerated(sampleSet);
47 }
```

Aufgabe dieser Funktion ist es, alle Merkmalsextraktionsmethoden auf jedes der Eingabesegmente anzuwenden und die entstehenden *features* als ein *Sample* zwischenzuspeichern (Zeile 40). Sind alle Methoden durchlaufen wird das so erstellte *Sample* dem *SampleSet* hinzugefügt (Zeile 43) und das gleiche Prozedere für das nächste Inputsegment durchlaufen.

In Zeile 30 wird die vom Benutzer implementierte `extract(...)`-Funktion aufgerufen. Die Variable `ok` (Zeile 24, 31, 33) soll sicherstellen, dass `extract(...)` mit den Eingabesegmenten umgehen kann. Ist dies nicht der Fall wird eine Fehlermeldung ausgegeben und die Schleifendurchläufe durch Setzen der Variablen `everythingFine` auf *false*, abgebrochen (Zeile 36).

Zuletzt werden noch das Signal für die Ampelanzeige (Zeile 45) und für den generierten Output (Zeile 46) an das nächste Modul gesendet.

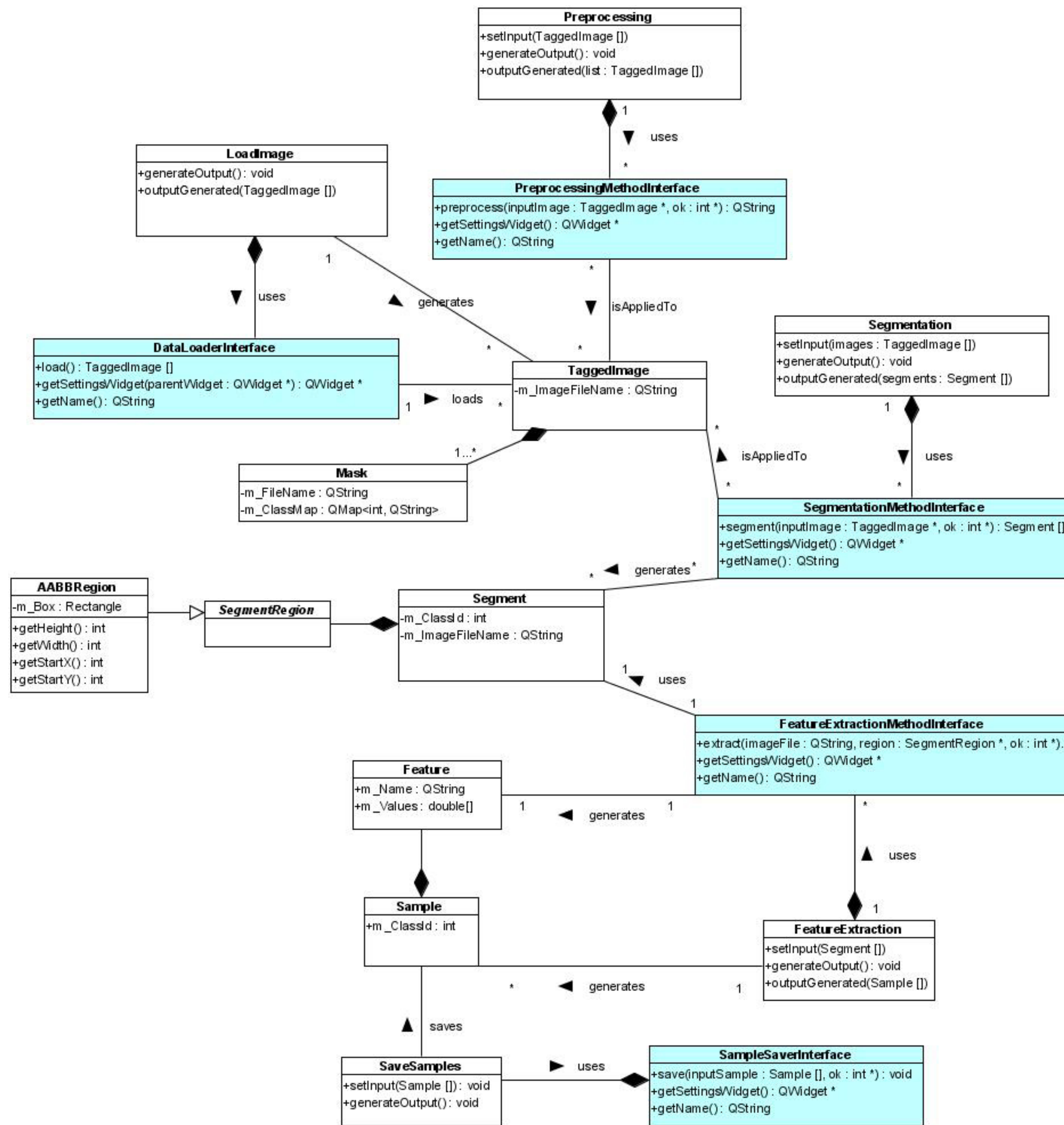

Abbildung 3.3: UML-Klassendiagramm für das entwickelte Framework

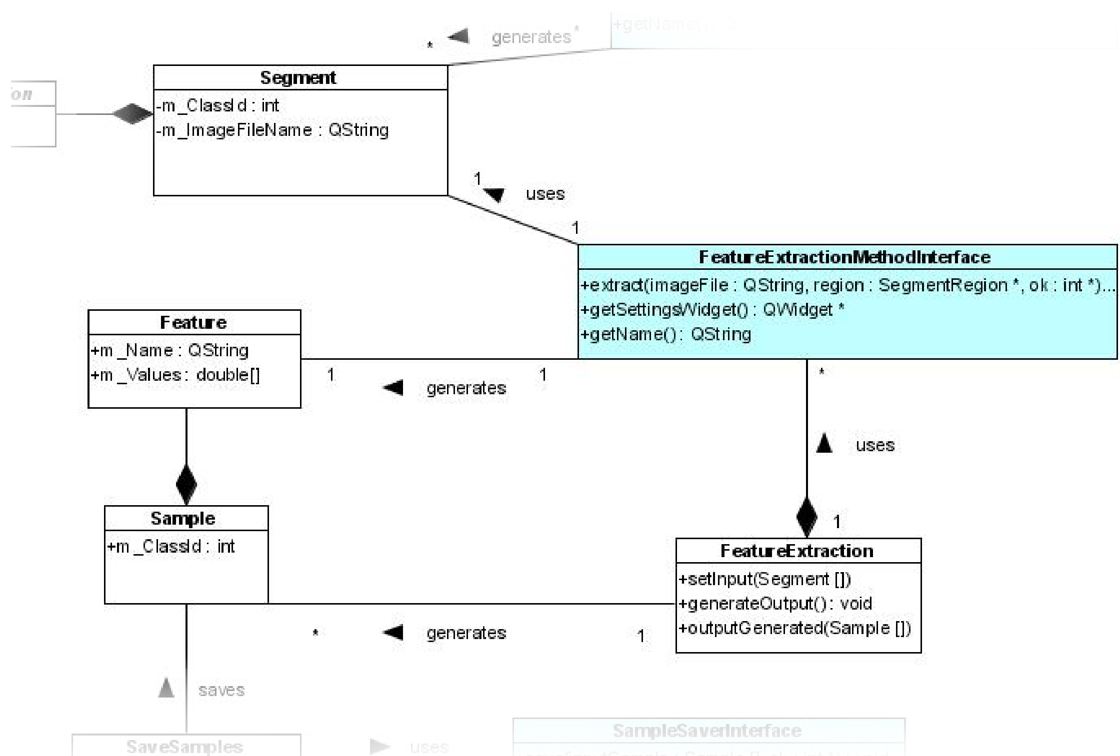

Abbildung 3.4: Ausschnitt aus UML-Klassendiagramm: *Feature extraction*

Kapitel 4

Zusammenfassung und Ausblick

Das vorgestellte Klassifikationsframework bildet ein einfaches, aber nützliches Instrument für die Durchführung von herkömmlichen Mustererkennungsaufgaben. Es ist ein pluginorientiertes, flexibles System, das je nach Anwendungsfall nahezu beliebig angepasst und erweitert werden kann. Somit bietet es Individualisierungsmöglichkeiten, was eine breite Zielgruppe ansprechen dürfte.

Weitere Bibliotheken können auf einfache Weise integriert, sowie zusätzliche Module und Methoden eingebettet werden. Darüber hinaus stellt es eine übersichtliche graphische Benutzeroberfläche bereit, die intuitiv bedienbar und verständlich ist.

Aufgrund des Zeitrahmens, in dem das Framework entwickelt wurde, konnten nur die wichtigsten Ziele und Vorhaben umgesetzt werden. Doch auf Basis des angefertigten Prototyps lassen sich mit Sicherheit recht einfach viele Erweiterungen realisieren.

Um aus der bisherigen Anwendung ein volles Klassifikationsprogramm zu erstellen, das nicht nur die Lernphase, sondern auch die Arbeisphase beinhaltet, müssen noch weitere Module integriert werden. Auch Aufgaben wie eine Merkmalsreduktion oder die Bereitstellung von vorgefertigten Methoden sowie Visualisierungen sind leicht zu verwirklichen.

Ebenso muss das Framework nicht auf Bilddaten zur Klassifikation beschränkt sein, sondern kann durch Modifikationen im Quellcode durchaus zum Beispiel auch für Audiodaten nutzbar gemacht werden.

Literaturverzeichnis

[Ame08] AMELING, Stefan: *Polypen- und Tumordetektion in Koloskopie-Videos.* 2008

[Bel61] BELLMAN, R. E.: *Adaptive Control Process.* Princeton University Press, 1961

[CS05] CHAN, Tony F. ; SHEN, Jianhong: *Image Processing and Analysis.* Siam, 2005. – ISBN 0–89871–589–X

[DHS01] DUDA, Richard O. ; HART, Peter E. ; STORK, David G.: *Pattern Classification.* 2. A Wiley-Interscience Publication, 2001. – ISBN 0–471–05669–3

[PH03] PAULUS, Dietrich W. R. ; HORNEGGER, Joachim: *Applied Pattern Recognition.* 4. Vieweg, 2003. – ISBN 3–528–35558–1