university of groningen

faculty of science and engineering

Detecting Astronomical Objects with Machine Learning

Master's Thesis Computing Science

January 27, 2021

Student: Michaël P. van de Weerd

Primary supervisor: dr. Michael H. F. Wilkinson

Secondary supervisor: prof. dr. Michael Biehl

Abstract

Over the years, Machine Learning (ML) has solidified its reputation as a quick and easy solution to all problems that are in some way, shape or form related to classification. In a lot of cases, this reputation is justified, as the ratio of effort of implementation to the quality of the results is often very low. As such, finding new areas in which ML might play a role is a worthwhile endeavor. In this master's thesis, an effort is made to apply ML in order to detect astronomical objects. This is done by constructing a max-tree out of astronomical data, computing feature vectors representing the component attributes found in the tree and determining the significance of these components using a Learning Vector Quantization (LVQ) classifier, resulting in a segmentation of the astronomical objects from the background and noise. Using an embedded Python implementation of LVQ, the MTObjects (MTO) segmentation software has been extended in order to produce these results from astronomical data in the optical domain, with their qualities being measured and compared to that of other MTO using a statistical segmentation method. These measurements show that LVQ does improve the *recall* of the segmentations, although at the cost of a significant amount of *precision*. Therefore, it is concluded that LVQ is not a suitable method to classify astronomical objects. Future research is required to further investigate the possibility of utilizing LVQ and ML in general in other ways.

Keywords: computer vision, max-trees, segmentation,
machine learning, learning vector quantization

Contents

1 Introduction **4**
1.1 Segmentation of Astronomical Objects 4
1.2 Reading this Document 5

2 Related Work **7**
2.1 Background 7
2.2 Max-Trees 7
2.3 Component Attributes 8
2.4 Learning Vector Quantization 12

3 Concept **15**
3.1 Component Attributes 15
3.1.1 Perimeter 15
3.1.2 Composite Positional Attributes 16
3.1.3 Intensity Attributes 16
3.1.4 Attributes for Segmentation 19
3.2 Segmentation Method 22

4 Realization **23**
4.1 Computing Component Attributes 23
4.2 Segmentation with LVQ 25
4.2.1 Embedding Python in C 25
4.2.2 Training the LVQ Classifier 27

5 Evaluation **29**
5.1 Hyperparameter Tuning 29

6 Results **31**
6.1 Segmenting Astronomical Data for Evaluation 32
6.2 Quantifying Segmentation Quality 32
6.3 Comparing Time Measurements 33

7 Conclusion **36**
7.1 Segmentation Quality 36
7.2 Performance 36

8 Future Work **37**
8.1 Determining Significance 37
8.2 Parameter Optimization 37
8.3 LVQ Implementation 38
8.4 Alternative Applications 38
8.5 Improving Statistical Segmentation 38

A Implementing Berger's Max-Tree Algorithm **42**

B Implementing Component Attribute Computations **45**

C Breaking the Python/C API **49**
C.1 Debugging the Python Interpreter 49
C.2 Considering Performance 49

Acronyms

ML	Machine Learning
LVQ	Learning Vector Quantization, first proposed by Kohonen [9]
GLVQ	Generalized LVQ, first proposed by Sato and Yamada [16]
SKLEARN	Scikit-Learn, ML framework for the Python programming language by Pedregosa et al. [12]
SKLVQ	LVQ for SKLEARN[1], LVQ extension for SKLEARN by Rick van Veen
MTO	MTObjects, statistical, max-tree-based classifier for the segmentation of astronomical objects by Moschini et al. [11]

[1] https://github.com/rickvanveen/sklvq

Chapter 1

Introduction

In this chapter, the subject of this master's thesis is introduced, leading up to the research questions it aims to answer. Additionally, an outline of the thesis is provided to act as a reading guide.

1.1 Segmentation of Astronomical Objects

The domains of radio astronomy and optical astronomy produce incredibly large amounts of data on a daily basis. For example, the Vera C. Rubin Observatory is estimated to produce images of 3200MP [14]. Information captured in these images is a textbook example of *big data* in more than one way. In the case of radio astronomy, the data that is being collected has a significantly high bit depth and is structured in three-dimensional images of such a high resolution that it is often measured in terms of gigavoxels [11]. It is evident that the extraction of knowledge from this data requires some form of automation in order to ensure feasibility and accuracy. Several methods and tools have been developed to observe a range of aspects and phenomena in the astronomical data. For example, research tools such as the Source Finding Application (SOFIA) and MTObjects (MTO) by Moschini et al. [11] can be used to determine the location of astronomical objects in such data. The result of applying these methods is an image in which each element (pixel of voxel) has been assigned a label, effectively grouping clusters of elements together. An example of such a segmentation using three different methods has been included in fig. 1.1.

This thesis mainly builds upon the works of Moschini et al. [11] and Haigh et al. [7], both of whom focus on the application of MTO using a statistical segmentation method. MTO distinguishes itself due to the fact that it buils a max-tree (MT) in order to represent the input data. The statistical segmentation method filters the nodes in the tree based on the computed ratio of integrated power of the local background (*flux*). The process of constructing a max-tree (MT) is described in more detail in chapter 2. Using a statistical approach requires a well-defined understanding of the objects under observation, which is not always available or hard to validate. An alternative to the statistical method is the use of Machine Learning (ML), which leaves the correlation of attributes of the objects and their significance to an intelligent computational system. This system is able to evaluate the data based on earlier observations of a *ground truth*, which implicates the relation between significance and attributes. This master's thesis explores the applicability of ML to the segmentation problem of detecting astronomical objects. To do so, two research questions are answered. First of all, the input of the ML system must be

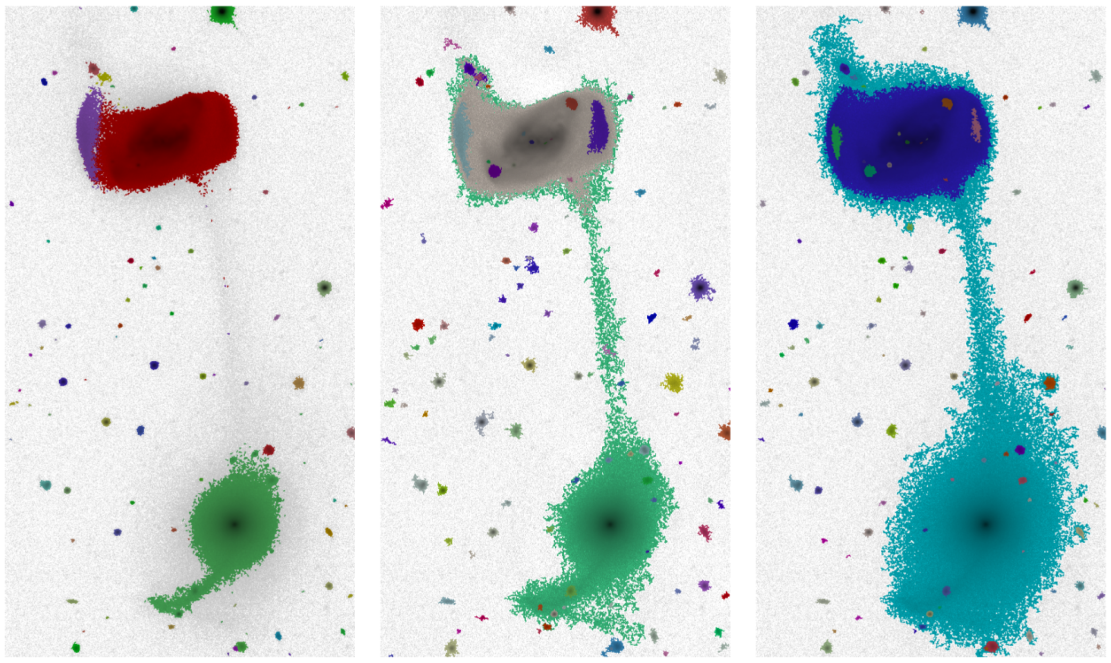


Figure 1.1: Example of a segmentation of a data-set representing two merging galaxies. The segmentations included have been performed by SExtractor (left), MTO with a background estimation by SExtractor (middle) and MTO with a statistical background estimation (right). Images have been taken from [11].

well-defined. This input consists of the attributes of (potential) objects in the input data, leading to

Research question 1. *Which attributes can be considered in order to filter astronomical objects?*

With the attributes, an ML classifier can be *trained* to be used for the segmentation of astronomical objects. Whether the attributes suffice for the classifier to perform this segmentation properly is however impossible to predict. Therefore, an implementation of this approach is realized, allowing for comparisons with other, statistical approaches, such as the ones mentioned above. These experiments pose

Research question 2. *Is Machine Learning a viable approach to the segmentation of astronomical data?*

1.2 Reading this Document

This master's thesis project build upon the work by many other research projects on subjects such as computer vision, ML and attribute computation. A short description of the main related works and the significance of their contents is provided in chapter 2. In chapter 3, the concept of this project is made concrete by defining the attributes to be considered during segmentation, selecting an appropriate ML method and determining the approach to the implementation of a proof of concept (POC). The actual realization of this concept is documented chapter 4, where technical challenges and their solutions are highlighted. Chapter 5 provides an evaluation of the POC in terms of quality and performance, making sure that the functionality that is required is present. Having a working POC, its performance is compared to the alternatives in chapter 6, providing an answer to research question 2. Before reflecting upon opportunities in chapter 8, the success of the master's thesis project is reflected upon in chapter 7.

Chapter 2

Related Work

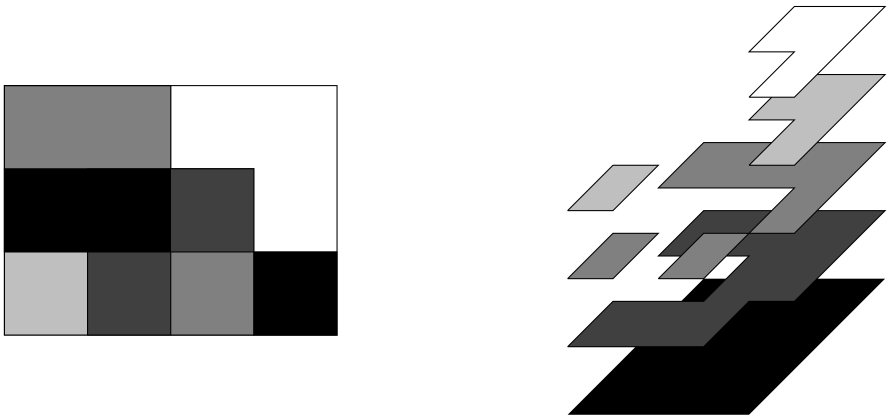
This chapter provides insight into relations between this master's thesis and other scientific work. Furthermore, several concepts are described in more detail, such as MTO and Learning Vector Quantization (LVQ), a promising ML technique.

2.1 Background

As mentioned in chapter 1, the research in this thesis mainly build upon the work by Moschini et al. [11] and Haigh et al. [7]. In the prior, the concept of MTO is demonstrated and in the latter MTO is intensively compared to other well known segmentation methods. This thesis can be considered as an extension of Haigh et al. [7], providing an additional comparison between MTO and another segmentation method. The manner of comparing different results will also be based on the methods used by Haigh et al. [7], in order to make them comparable to the results presented there. The application of ML requires a classifier to be trained on labeled data. To this end, attributes of connected components in a max-tree are to be computed. Several sources to be consulted for these computations are Gonzalez and Woods [6] and Tushabe [19]. The prior provides general definitions of connected component attributes while the latter concerns attributes of components in the context of an actual max-tree, albeit constructed from colored pictures. Still, many of the attributes defined in these works are transferable to components in a max-tree constructed from optical or radio astronomical images.

2.2 Max-Trees

Internally, MTO constructs a max-tree from in images in either two or three dimensions. Salembier, Oliveras, and Garrido [15] proposed the max-tree to be a structured representation of an image in which the maxima within the image are the leaves of the tree (hence its name). The max-tree is closely related to the concept of a component tree, the difference being the fact that the parent nodes of a max-tree do not store the elements of their children as well, avoiding data redundancy [1]. A max-tree can be constructed from a image using an algorithm first proposed by Berger et al. [1], in addition to an algorithm that can be applied to produce a *canonical* max-tree, given a max-tree. Here, the term canonical indicates that at every level in the tree, connected elements are altered to share a single parent within their level, with the parent itself having a parent in the subsequent level [1]. This allows each component to be represented by a single node: the canonical root. A visual demonstration of this concept and of that of a



$$\begin{bmatrix} 3 & \mathbf{3} & 5 & 5 \\ 1 & 1 & 2 & \mathbf{5} \\ \mathbf{4} & \mathbf{2} & \mathbf{3} & \mathbf{1} \end{bmatrix} \qquad \begin{bmatrix} \mathrm{E} & \mathbf{F} & \mathrm{A} & \mathrm{B} \\ \mathrm{J} & \mathrm{K} & \mathrm{H} & \mathbf{C} \\ \mathbf{D} & \mathbf{I} & \mathbf{G} & \mathbf{L} \end{bmatrix}$$

Figure 2.1: Example gray-scale image f (top left) with its matrix representation (bottom left) and the ordering of its elements (bottom right). An "exploded" view of f (top right) gives a more explicit perspective on the order of the elements. The ordering characters indicate the order in which elements are encountered when traversing through the image from highest to lowest value. Level-roots (last encountered elements for their respective value) are marked in a boldface font.

max-tree has been included in fig. 2.2. The algorithm proposed by Berger et al. [1] defined three procedures: one that return the root for a given node, one that computes an ordering and parent matrix for a given image and one that returns a canonical parent matrix for a given image and parent matrix. Note that these parent matrices are simply matrix representations of max-trees. The full definition of the Berger et al. [1] algorithm is included in algorithm 1.

For the implementation of MTO, a modified version of the algorithm in [1] has been used. These modifications allow the algorithm to be executed in a multi-threaded fashion and are based on the work by Moschini, Meijster, and Wilkinson [10]. Effectively, every level in the image is refined in its own thread from a computed pilot tree, greatly improving the performance of the procedure. In fig. 2.3, the difference in performance of MTO is visualized in a comb plot, displaying the time measurements for the distinct stage of segmentation when utilizing 16 threads and a single thread. Here, the speed at which a data set is segmented is improved to less than half the speed of using a single thread for the stage in which the max-tree is refined. The stages mentioned here are discussed in more detail in chapter 3.

2.3 Component Attributes

As mentioned in the previous sections, each level-root in a max-tree represents a connected component within the image. Knowing which elements make up such a component allows for the computation of attributes or descriptors [6] in order to describe its characteristics. An obvious attribute that can easily be computed is the *area*. In Berger et al. [1], this attribute is defined as the number of elements within the component. Tushabe [19] provides the formal

Algorithm 1 Pseudo-code notation of the max-tree algorithm proposed by Berger et al. [1]. Note that $\mathcal{N}(x)$ refers to the set of connected neighbors of elements x in f. A complete GNU Octave implementation of this algorithm has been included in appendix A.

```
procedure FindRoot($x$)
    if zpar($x$) = $x$ then
        return $x$
    else
        zpar($x$) $\leftarrow$ FindRoot(zpar($x$))
        return zpar($x$)
    end if
end procedure

procedure ComputeTree($f$)
    for all $x \in f$ do
        zpar($x$) $\leftarrow$ null
    end for
    $\mathcal{R} \leftarrow$ ReverseSort($f$)
    for all $x \in \mathcal{R}$ do
        parent($x$) $\leftarrow x$
        zpar($x$) $\leftarrow x$
        for all $n \in \mathcal{N}(x)$ : zpar($n$) $\neq$ null do
            $r \leftarrow$ FindRoot($n$)
            if $r \neq x$ then
                parent($r$) $\leftarrow x$
                zpar($r$) $\leftarrow x$
            end if
        end for
    end for
    return ($\mathcal{R}$, parent)
end procedure

procedure CanonizeTree(parent, $f$)
    for all $x \in$ ReverseOrder($\mathcal{R}$) do
        $y \leftarrow$ parent($x$)
        if $f$(parent($y$)) = $f(y)$ then
            parent($x$) $\leftarrow$ parent($y$)
        end if
    end for
    return parent
end procedure
```

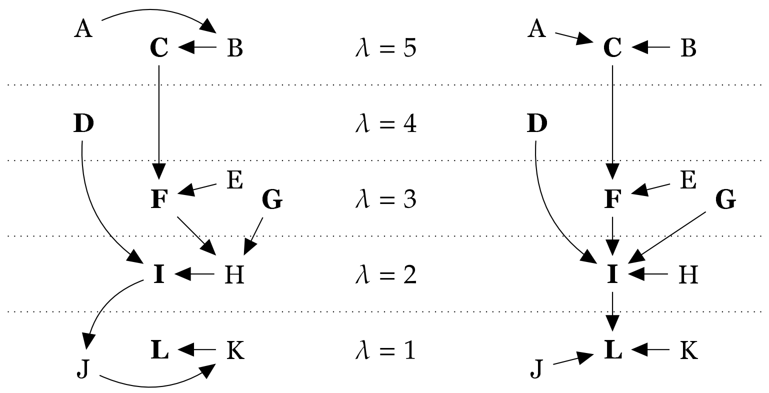


Figure 2.2: Max-tree representations of image f as seen in fig. 2.1, with the canonical variant appearing on the right hand side. Again, level roots have been indicated by a boldface ordering character. Here, λ indicates the level of the respective elements represented by the nodes (cf. the matrix representation of f in fig. 2.1).

mathematical definition of area $A(X)$ of component X as

$$A(X) = \sum_{x \in X} \mathbf{1}_X(x). \tag{2.1}$$

Here, $\mathbf{1}_X(x)$ is a so-called indicator function, resolving to 1 if $x \in X$ or 0 otherwise. Note that the term "area" spawned in the domain of two-dimensional components, but can be applied in higher dimensions as well — e.g. indicating the volume of a cube in three dimensions. Berger et al. [1] provides an algorithm for computing the area of a component in the context of a max-tree, which is included in algorithm 2.

Algorithm 2 Pseudo-code notation for the computation of the area of a component in a max-tree, as defined in eq. (2.1). Taken from [1] with the addition of the $p \neq \text{parent}(p)$ condition. A GNU Octave implementation of this algorithm is included in appendix B.

```
procedure ComputeArea($f, \mathcal{R}$, parent)
    for all $x \in \mathcal{R}$ do
        area($x$) ← 1
    end for
    for all $x \in \mathcal{R} : x \neq \text{parent}(x)$ in direct order do
        area(parent($x$)) ← area(parent($x$)) + area($x$)
    end for
end procedure
```

Another attribute concept closely related — altough not at first glance — is that of the *perimeter*. Gonzalez and Woods [6] describe the perimeter as the length of the boundary, but unfortunately fail to provide a formal definition or algorithm. Interpreting the length of the boundary as the number of elements that are not exclusively connected to elements within the same components, a novel approach to computing the perimiter is presented in chapter 3. Again, it should be noted that the concept of the perimeter is transferable in to dimensions higher than two — e. g. in three dimension it can be interpreted as the surface of a cube.

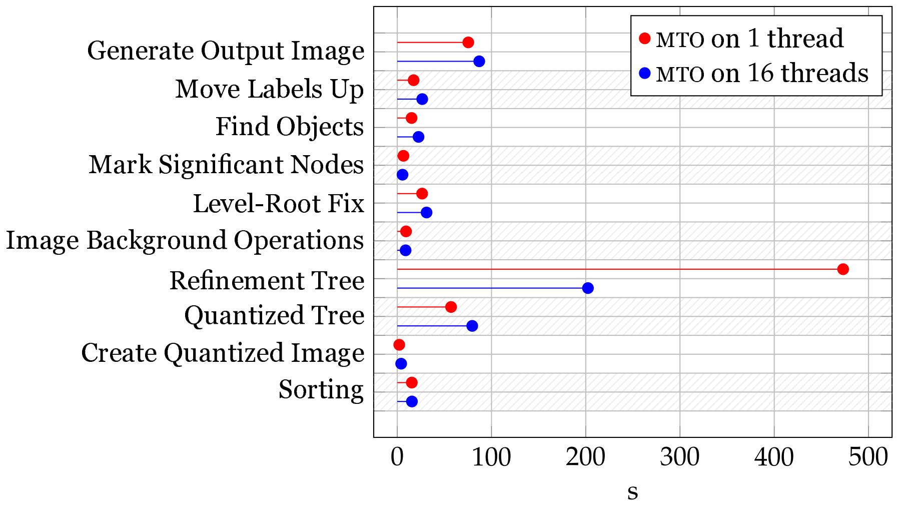


Figure 2.3: Comparison of the time measurements of the segmentation of a data set (cluster 1) with MTO using 16 threads and 1 thread for refinement. Note that the stages in the segmentation process are presented in chronological order on the y-axis, from bottom to top.

Using the attributes area $A(X)$ and perimiter $\mathfrak{P}(X)$, composite attributes can be computed such as *compactness* and *circularity ratio* [6]. Gonzalez and Woods [6] provide the following formal definitions of compactness $C(X)$ and circularity ratio $R_c(X)$ as

$$C(X) = \frac{\mathfrak{P}(X)^2}{A(X)}, \tag{2.2}$$

$$R_c(X) = \frac{4\pi A(X)}{\mathfrak{P}(X)^2}. \tag{2.3}$$

The values of these attributes provide an indication of the shape of the components that they represent. E. g.$R_c(X)$ approaches a value of 1 as the shape of the component approaches the shape of a perfect circle [6].

In addition to the use of (relative) positional information about the elements of a component, the value — also called *intensity* in [6, 19] — they represent can also be indicative of a component attribute. Many examples are presented in literature, most of which are rather straight forward such as the *sum of (squared) intensities* [6] and *grayscale* (i. e. average value within a component) [19]. Tushabe [19] provides the following formal definition of the latter:

$$G(x) = \frac{\sum_{x\in X} f(x)}{A(X)}. \tag{2.4}$$

More sophisticated measurements based on intensity levels are the *power* [19]

$$\mathcal{P}(X, f, \alpha) = \sum_{x\in X}(f(x) - \alpha)^2, \tag{2.5}$$

for an image f and parent intensity level α, and *volume* [19]

$$V(X, f, \alpha) = \sum_{x\in X}(f(x) - \alpha). \tag{2.6}$$

Furthermore, Gonzalez and Woods [6] provide a definition of the *entropy* attribute, citing it to be the average amount of information that each element in a component can convey. Given a discrete set of the distinct grayscale values $\{a_1, a_2, \dots, a_J\}$ that appear in a component of size $M \times N$, the probability of such a grayscale value being encountered in the component is

$$P(a_j) = \frac{a_j}{MN}. \tag{2.7}$$

This function can be used to compute a *histogram* of component X. Using this definition, Gonzalez and Woods [6] provide the formal definition of entropy as

$$H(X) = -\sum_{j=1}^{J} P\left(a_j\right) \log P\left(a_j\right). \tag{2.8}$$

Finally, a component set of attributes of interest is that of the four *invariant moments*: *non-compactness*, *elongation*, *flatness* and *sparseness*. Westenberg, Roerdink, and Wilkinson [21] provide definitions of these, suitable for usage for two- and three-dimensional components (as opposed to the moments presented by Hu [8], which only apply to two-dimensional components). First of all, Westenberg, Roerdink, and Wilkinson [21] define the non-compactness attribute $\mathcal{N}(X)$ as

$$\mathcal{N}(X) = \frac{\mathrm{Tr}\,\mathbf{I}(X)^{\frac{3}{5}}}{A(X)}, \tag{2.9}$$

with the *moment of inertia* tensor $\mathbf{I}(X)$ defined as

$$\mathbf{I}_{ij}(X) = \begin{cases} \sum_X (i - \bar{i})^2 + \frac{A(X)}{12} & \text{if } i = j \\ \sum_X (i - \bar{i})(j - \bar{j}) & \text{otherwise} \end{cases} \tag{2.10}$$

for $i, j \in \{x, y, z\}$. Computing the *eigenvalues* $\lambda_i(X)$ of $\mathbf{I}(X)$ and ordering them such that

$$|\lambda_1(X)| \geq |\lambda_2(X)| \geq |\lambda_3(X)| \tag{2.11}$$

allows the computation of the attributes elongation $\mathcal{E}(X)$, flatness $\mathcal{F}(X)$ and sparseness $\mathcal{S}(X)$ with [21]

$$\mathcal{E}(X) = \left|\frac{\lambda_1(X)}{\lambda_2(X)}\right| \tag{2.12}$$

$$\mathcal{F}(X) = \left|\frac{\lambda_2(X)}{\lambda_3(X)}\right| \tag{2.13}$$

$$\mathcal{S}(X) = \frac{\pi}{6A(X)} \prod_{i=1}^{3} \sqrt{\frac{20\,|\lambda_i(X)|}{A(X)}}. \tag{2.14}$$

2.4 Learning Vector Quantization

A particularly interesting ML technique is that of LVQ, first proposed by Kohonen [9]. LVQ is a framework for prototype- based classifiers and can be considered to be a simplification of a Bayes classifier [2, 3]. Biehl, Hammer, and Villmann [3] explain that the difference between these methods is the fact that LVQ replaces the density estimation with a method where each of the C classes is represented by one or more *prototypes*. This dichotomy of classes (i. e. labels associated with a specific class) and prototypes is formally defined as

$$\left\{\mathbf{w}^j, c^j\right\}_{j=1}^{M} \text{ with } \mathbf{w}^j \in \mathbb{R}^N \text{ and } c^j \in \{1, 2, \ldots, C\}. \tag{2.15}$$

Here, N indicates the number of features of which the data-points consist, and M indicates the number of prototypes to be used for classification. Note that the difintion in [2, 3] requires that $M \geq C$. Having this mapping of prototypes and class labels, an arbitrary feature vector ξ is assigned to the class associated with the nearest prototype $\mathbf{w}^*$ — noted as the class where $c^* = c(\mathbf{w}^*)$ [3]. Formally, this provides the following definition of the closest prototype of ξ:

$$\mathbf{w}^*(\xi) \text{ with } d\Big(\mathbf{w}^*(\xi), \xi\Big) = \min\left[d\left(\mathbf{w}^j, \xi\right)\right]_{j=1}^{M}, \tag{2.16}$$

with distance measure d. This prototype is commonly refered to as the *winner*, or using the shorthand $\mathbf{w}^*$.

Having the means to store an LVQ classifier still requires some meaningfull way determining the values of the prototypes $\mathbf{w}$. Several LVQ classifiers have been defined in literature. In order to introduce the general concept of an LVQ training algorithm, only the LVQ1 training scheme by Kohonen [9] and Generalized LVQ (GLVQ) training scheme by Sato and Yamada [16] will be featured in this section. In [2] the steps of the LVQ1 scheme are summarized as follows:

1. At time step t, select a random labeled feature vector ξ^μ and its label y^μ from data-set $\mathbb{D}$ of size P with a uniform probability $\frac{1}{P}$;
2. Find the winning prototype $\mathbf{w}^*_\mu$ and associated class label c^*_μ;
3. Perform a *winner-takes-all* update, moving the prototype in order to increase its distance to the feature vector when their associated labels do not match, or decreasing the distance otherwise:

$$\mathbf{w}^*_\mu(t+1) = \mathbf{w}^*_\mu(t) + \eta_w \psi\left(c^*_\mu, y^\mu\right)\left(\xi^\mu - \mathbf{w}^*_\mu\right) \text{ with } \psi(c, y) = \begin{cases} +1 & \text{if } c = y \\ -1 & \text{otherwise.} \end{cases} \tag{2.17}$$

Intuitively, this procedure lets data-points either attract or repulse whichever prototype is closest, based on whether or not their labels match. The general idea of this concept is that data-points of the same class can be identified by their feature values and that a prototype can be defined that is closest to all of them. Moving the prototypes in the aformentioned fashion is supposed to find a value that allows this iteratively. The magnitude of these movements can be controlled by the definition of the learning rate η_w. Several — sometimes quite sophisticated — initialization methods are available for the values of the prototypes, such as placing them in the class-conditional mean vectors vectors in the data-set or applying a K-means procedure on each class separately [2, 3, 16].

The popular GLVQ training scheme is very similar to that of LVQ1. Instead of defining a single winner prototype $\mathbf{w}^*$, GLVQ defined a *correct winner* $\mathbf{w}^J$ and an *incorrect winner* $\mathbf{w}^K$, with the prior being the prototype closest to arbitrary data-point ξ of an identical class label y and the latter being the clostest prototype of any other class label [2, 3, 16]. Sato and Yamada [16] provides the following formal definition (note their similarity to eq. (2.16)):

$$\mathbf{w}^J(\xi) \text{ with } d\left(\mathbf{w}^J, \xi\right) = \min\left[d\left(\mathbf{w}^j, \xi\right) : c^j = y\right]_{j=1}^{M}, \tag{2.18}$$

$$\mathbf{w}^K(\xi) \text{ with } d\left(\mathbf{w}^J, \xi\right) = \min\left[d\left(\mathbf{w}^j, \xi\right) : c^j \neq y\right]_{j=1}^{M}. \tag{2.19}$$

Using these definitions, the classification of a data-set of P data-points is evaluated:

$$E^{\text{GLVQ}} = \sum_{\mu=1}^{P} \phi\left(e^{\mu}\right) \text{ with } e^{\mu} = \frac{d\left(\mathbf{w}_{\mu}^{J}, \xi^{\mu}\right) - d\left(\mathbf{w}_{\mu}^{K}, \xi^{\mu}\right)}{d\left(\mathbf{w}_{\mu}^{J}, \xi^{\mu}\right) + d\left(\mathbf{w}_{\mu}^{K}, \xi^{\mu}\right)}. \tag{2.20}$$

Here, $\phi(e)$ is a cost function, the return value of which is in the range $[-1, 1]$ [2, 3, 16]. The updating scheme itself is aimed at minimizing the value of E^{GLVQ}, as a negative value of e indicates a correctly classified data-point. To this end, two prototypes are update at each step. Sato and Yamada [16] provides the following definition of the GLVQ scheme:

1. At timestep t, select a random labeled feature vector ξ^{μ} and its label y^{μ} from data-set $\mathbb{D}$ of size P with a uniform probability $\frac{1}{P}$;
2. Find the respective correct and incorrect winners $\mathbf{w}^{J}$ and $\mathbf{w}^{K}$ with class labels $c_{\mu}^{J} = y^{\mu} \neq c_{\mu}^{K}$;
3. Perform the update, moving the correct and incorrect respectively increasing and reducing the distance to ξ^{μ}:

$$\mathbf{w}_{\mu}^{J}(t+1) = \mathbf{w}_{\mu}^{J}(t) + \eta_{w} \frac{\partial \psi\left(e^{\mu}\right)}{\partial \mathbf{w}_{\mu}^{J}}, \tag{2.21}$$

$$\mathbf{w}_{\mu}^{K}(t+1) = \mathbf{w}_{\mu}^{J}(t) - \eta_{w} \frac{\partial \psi\left(e^{\mu}\right)}{\partial \mathbf{w}_{\mu}^{K}}. \tag{2.22}$$

Chapter 3

Concept

As specified in chapter 2, this master's thesis builds upon the research by Moschini et al. [11]. Here, the segmentation of 6-connected components in a max-tree is achieved by computing their *flux* attribute and using a χ^2 statistical test to determine the significance of the tree's nodes. The goal of this master's thesis is to innovate on this concept by expanding the collection of attributes that will be taken into account during segmentation. These attributes are selected in section 3.1 based on whether they can be computed given the data available during the construction of the max-tree. Furthermore, the nature of the data and the application of this work requires the attributes to be

- rotation invariant,
- translation invariant and
- scale invariant,

which is included in the decission of the final selection. In order to accommodate the segmenatation using the computed attributes, a method needs to be selected that can determine whether two given components belong to the same astronomical object. In section 3.2, a ML technique is chosen to be able to perform this task.

3.1 Component Attributes

In chapter 2, several component attributes have been introduced that can be used to characterize level-roots in a max-tree. In this section, the way in which these measurements can be computed in a practical sense will be layed out, as only one algorithm as been found in the literature (for computing the area attribute). Note that the algorithms presented here store the attribute value in a matrix of the same shape as the input image. The attribute value for a component can be found at the location of its canonical root in the computed attribute matrix.

3.1.1 Perimeter

Computing the perimeter component attribute is a non-trivial task. As defined by Gonzalez and Woods [6], a the perimiter indicates the length of the boundary of a component. In this thesis, this definition is interpreted as the number of elements that are connected to elements that are not part of the component to which they belong themselfs. The complexity of computing the perimiter is mainly due to the fact it has to be taken into account that any element that is part of the boundary of its its own component can also contribute to the boundary of its parent, its parent parent, etc.. In order to solve this problem, a *boundary vector* $\mathscr{B}(X)$ is constructed, which indicates the contribution of a component X to each layer in the tree. Obviously, these contributions only apply to the direct and indirect parents of said component. To construct this vector, we first compute the intermediate contribution set $\Lambda(x)$ for each element x. This set is composed of the layers in which x is part of the perimeter, computed as

$$\Lambda(x) = \left\{ y \in \mathbb{Z} : \arg\min_{n \in \mathcal{N}(x)} \lambda(n) < y \leq \lambda(x) \right\}, \tag{3.1}$$

with $\mathcal{N}(x)$ the neighbors of x and $\lambda(x)$ its level. With this contribution set $\Lambda(x)$ computed for every element x, the contribution of X to the perimeter of layer λ is equal to the count of λ inclusions in the contribution set of elements in X:

$$\mathscr{B}(X)_\lambda = \left|\left\{x \in X : \lambda \in \Lambda(x)\right\}\right|. \tag{3.2}$$

From this, the perimeter $\mathfrak{P}\left(X_\lambda^m\right)$ of component X_λ^m in branch m and layer λ can be determined by summing the contributions of all components to *lambda*:

$$\mathfrak{P}\left(X_\lambda^\mu\right) = \sum_{\nu \in N} \mathscr{B}\left(X_\gamma^\nu\right)_\lambda, \tag{3.3}$$

where, $N = \{\nu \in \mathbb{Z} : \mu \preceq \nu\}$ and $\gamma \geq \lambda$. Here, the set of branches N is constructed out of all branches ν that are either equal to, or divarications of, branch μ, written as $\mu \preceq \nu$. Figure 3.1 is included to illustrate this procedure with and intuitive example, displaying the relation between the components, layers, boundary vector and contribution set, leading up to the final computation of the perimeter attribute. Note that, in order to consider elements on the edges of the image as a whole as part of the perimeter of the root component, non-existing neighbors are considered to be part of the level where $\lambda = -1$. In algorithm 3, an algorithm is included in pseudo-code, demonstrating the procedure that can be applied in order to computer the perimeter of a component in a max-tree.

3.1.2 Composite Positional Attributes

Being able to compute the area and perimeter of a compontent also allows the computation of the compactness and circularity ratio defined by Gonzalez and Woods [6]. No extensive algorithmics are needed for this, as computing these attributes is as simple as applying the equations provided in chapter 2.

3.1.3 Intensity Attributes

Moving on from the attributes related to the position of component elements, the sum of intensities and its squared variant are rather simple to compute. This is done by adding all of the (squared) gray-values of the elements in a component. Two procedures are presented in algorithm 4 that compute the sum of intensities and sum of squared intensities respectively for a given max-tree. The gray-scale attributes indicates the average value of its elements, as defined in the mathematical formulation presented in chapter 2, taken from Tushabe [19]. An algorithm based on that of the area attribute defined by Moschini et al. [11] is included in algorithm 5.

Algorithm 3 Pseudo-code notation for the computation of the perimeter of a component in a max-tree, as defined eq. (3.3). Note the use of the neighbor function $\mathcal{N}$. A GNU Octave implementation of this algorithm is included in appendix B.

```
procedure ComputePerimeter($f, \mathcal{R}$, parent)
    for all $x \in \mathcal{R}$ do
        $\mathcal{B}(x) \leftarrow 0$, perimeter($x$) $\leftarrow 0$
    end for
    for all $\alpha \in \mathbb{R} : \alpha \in f$ do
        for all $x \in \mathcal{R} : f(x) \geq \alpha$ do
            for all $n \in \mathcal{N}(x) : f(x) > f(n)$ do
                $\mathcal{B}(x) \leftarrow \mathcal{B}(x) + 1$
            end for
        end for
    end for
    for all $x \in \mathcal{R}$ do
        $y \leftarrow x$
        while $\mathcal{B}(x) > 0$ do
            perimeter($y$) $\leftarrow$ perimeter($y$) $+ 1$, $\mathcal{B}(x) \leftarrow \mathcal{B}(x) - 1$, $y \leftarrow$ parent($y$)
        end while
    end for
end procedure
```

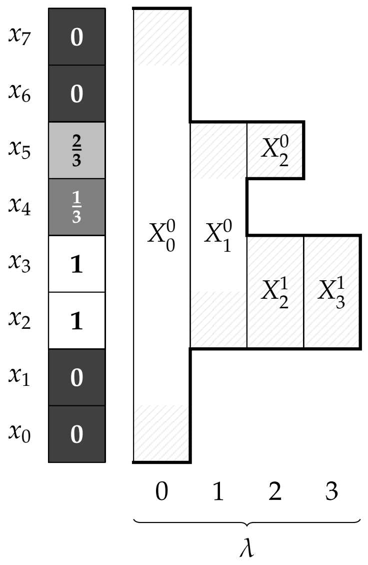
Algorithm 4 Pseudo-code notation for the computation of the sum of (squared) intensities of a component in a max-tree, based on algorithm 2 taken from [1]. A GNU Octave implementation for both procedures is included in appendix B.

```
procedure ComputeSumInt($f, \mathcal{R}$, parent)
    for all $x \in \mathcal{R}$ do
        sum($x$) $\leftarrow f(x)$
    end for
    for all $x \in \mathcal{R} : x \neq$ parent($x$) do
        sum(parent($x$)) $\leftarrow$ sum(parent($x$)) + sum($x$)
    end for
end procedure

procedure ComputeSumIntSquared($f, \mathcal{R}$, parent)
    for all $x \in \mathcal{R}$ do
        sums($x$) $\leftarrow f(x)^2$
    end for
    for all $x \in \mathcal{R} : x \neq$ parent($x$) do
        sums(parent($x$)) $\leftarrow$ sums(parent($x$)) + sums($x$)$^2$
    end for
end procedure
```

$$\begin{aligned}
\mathcal{B}\left(X_0^0\right) &= \{2, & 0, && 0, && 0\} \\
\mathcal{B}\left(X_1^0\right) &= \{0, & 0, && 0, && 0\} \\
\mathcal{B}\left(X_2^0\right) &= \{0, & 1, && 1, && 0\} \\
\mathcal{B}\left(X_2^1\right) &= \{0, & 0, && 0, && 0\} \\
\mathcal{B}\left(X_3^1\right) &= \{0, & 1, && 2, && 2\}
\end{aligned}$$

$$\boxed{\begin{aligned}
\mathfrak{P}\left(X_0^0\right) &= \mathcal{B}\left(X_0^0\right)_0 && = 2 \\
\mathfrak{P}\left(X_1^0\right) &= \mathcal{B}\left(X_2^0\right)_1 + \mathcal{B}\left(X_3^1\right)_1 && = 2 \\
\mathfrak{P}\left(X_2^0\right) &= \mathcal{B}\left(X_2^0\right)_2 && = 1 \\
\mathfrak{P}\left(X_2^1\right) &= \mathcal{B}\left(X_3^1\right)_2 && = 2 \\
\mathfrak{P}\left(X_3^1\right) &= \mathcal{B}\left(X_3^1\right)_3 && = 2
\end{aligned}}$$



$$\begin{aligned}
\Lambda(x_7) &= \{y \in \mathbb{Z} : -1 < y \leq 0\} && = \{0\} \\
\Lambda(x_6) &= \{y \in \mathbb{Z} : 0 < y \leq 0\} && = \emptyset \\
\Lambda(x_5) &= \{y \in \mathbb{Z} : 0 < y \leq 2\} && = \{1, 2\} \\
\Lambda(x_4) &= \{y \in \mathbb{Z} : 1 < y \leq 1\} && = \emptyset \\
\Lambda(x_3) &= \{y \in \mathbb{Z} : 1 < y \leq 3\} && = \{2, 3\} \\
\Lambda(x_2) &= \{y \in \mathbb{Z} : 0 < y \leq 3\} && = \{1, 2, 3\} \\
\Lambda(x_1) &= \{y \in \mathbb{Z} : 0 < y \leq 0\} && = \emptyset \\
\Lambda(x_0) &= \{y \in \mathbb{Z} : -1 < y \leq 0\} && = \{0\}
\end{aligned}$$

Figure 3.1: Illustration of the computation of the contribution sets Λ in one dimension, used to determine the boundary vector $\mathcal{B}$. The contribution of each element to a layer is marked in the component structure.

Based on he intensity values in a component, a histogram can be computed, indicating the number of times each intensity value occurs. In turn, the histogram enables the computation of the power and entropy attributes. As described in more detail in chapter 2, the power attributes indicates the effect of removing a component from its parent [19] and entropy entails the amount of information that each element in a component can convey [6]. In algorithm 6, a pseudo-code notation is included for each of the procedures that can be used to compute the histogram, power and entropy of a given max-tree component.

The invariant moments non-compactness, elongation, flatness and sparseness can be computed using the equations provided in chapter 2. This does however require the sums of the (products of the) elements in every dimension to be known: $\sum x$, $\sum y$, $\sum z$, $\sum x^2$, $\sum y^2$, $\sum z^2$, $\sum xy$, $\sum xz$ and $\sum yz$. Computing these values is trivial, leaving only the application of the aforementioned equations.

Algorithm 5 Pseudo-code notation for the computation of the gray-scale attribute of a component in a max-tree. A GNU Octave implementation is included in appendix B.

```
procedure ComputeGrayScale($f, \mathcal{R}$, parent)
    for all $x \in \mathcal{R}$ do
        gray-scale($x$) $\leftarrow 0$
    end for
    for all $x \in \mathcal{R}$ do
        $y \leftarrow x$
        loop
            gray-scale($y$) $\leftarrow$ gray-scale($y$) $+ \frac{f(x)}{\text{area}(y)}$
            if $y$ = parent($y$) then
                break
            else
                $y \leftarrow$ parent($y$)
            end if
        end loop
    end for
end procedure
```

3.1.4 Attributes for Segmentation

Not all of the aformentioned attributes are suitable for useage in combination with MTO and a ML method. E. g., the computation of a histogram is quite expensive in terms of computational power, as it requires an additional processing step. Therefore, attributes dependent on histogram computation are excluded. An alternative computation method is however present in the initial implementation of MTO for the power attribute, which can therefor be included after all. Ultimately, this leaves the following attributes for the selection of attributes to be computed for segmentation by a ML method:

- area,
- perimeter,
- power,
- compactness,
- circularity ratio,
- gray-scale,
- noncompactness,
- elongation,
- flatness and
- sparseness.

The code base developed by Moschini et al. [11] is extended with these attributes, as documented in chapter 4. Here, a *bottom-up flooding* and *top-down merging* approach is used to construct

Algorithm 6 Pseudo-code notation for the computation of the histogram, power and entropy of a component in a max-tree.

```
procedure ComputeHistogram($f, \mathcal{R}$, parent)
    for all $x \in \mathcal{R}$ do
        for all $\alpha \in \mathbb{R} : \alpha \in f$ do
            histogram$(x, \alpha) \leftarrow 0$
        end for
    end for
    for all $x \in \mathcal{R}$ in reverse order do
        $y \leftarrow x$
        loop
            histogram$(y, f(x)) \leftarrow$ histogram$(y, f(x)) + 1$
            if $y =$ parent$(y)$ then
                break
            end if
            $y \leftarrow$ parent$(y)$
        end loop
    end for
end procedure

procedure ComputePower($f, \mathcal{R}$, parent)
    for all $x \in \mathcal{R}$ do
        for all $\alpha \in \mathbb{R} : \alpha \in f$ do
            power$(x) \leftarrow$ power$(x) +$ histogram$(x, \alpha)(f(x) - \alpha)^2$
        end for
    end for
end procedure

procedure ComputeEntropy($f, \mathcal{R}$, parent)
    for all $x \in \mathcal{R}$ do
        for all $v \in \mathbb{R} : v \in f$ do
            $n \leftarrow n +$ histogram$(x, v)$
        end for
        for all $v \in \mathbb{R} : v \in f$ do
            $t \leftarrow \frac{\text{histogram}(p)}{n}$
            entropy$(x) \leftarrow$ entropy$(x) + t\frac{\log t}{\log 2}$
        end for
    end for
end procedure
```

the hierarchy of the max-tree and compute its attributes all at once. These two stages are referred to as the **compute** and **refine** stages respectively.

In the first stage, a tree is constructed from a *quantized image*. Within such an image, each element (a pixel in two-, or a voxel in three dimensions) is assigned to a *level*, based on their gray value. The flooding algorithms connects the elements as nodes in the max-tree, starting at the bottom level (the *root*) and ending at the *leafs*. This approach allows for parallelization, as shown by Moschini, Meijster, and Wilkinson [10], computing the hierarchy of each level in a separate thread. The output yielded by this procedure is called a *pilot tree* and is passed on to the second stage in which it is refined. Here, the tree structure is not altered, but the computation of its attributes can be completed in order to retrieve the definite max-tree [10]. This allows for attributes that are dependent on other attributes to be computed as well, e.g. the elongation attribute which is dependant on the sum of coordinates in each spatial dimension. In fig. 3.2, the two stages and their output have been visualized. This illustration also indicates the attributes of the pilot tree and the max-tree, and at which point in the process as a whole they become available. Aside from the aforementioned selection of attributes, intermediate attributes required by others are indicated here as well.

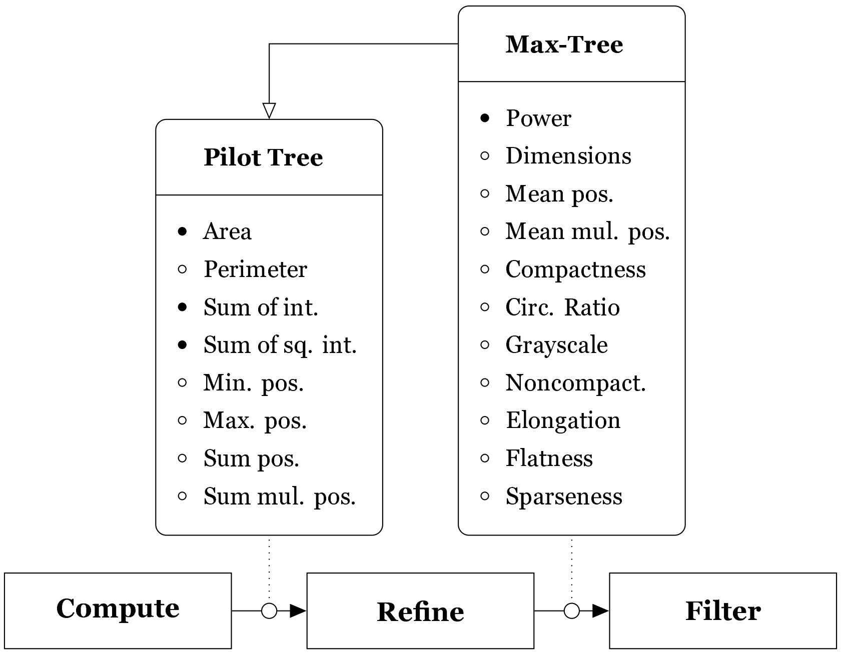


Figure 3.2: Diagram of the three main subprocedures in the software by Moschini et al. [11] that are relevant to the subject of this thesis. Note that the attributes of the pilot tree on the left are also available to the max-tree on the right, as indicated by the inheritance relation. Attributes that where already implemented in the initial version of the code have been indicated by a filled bullet, while attributes implemented as part of this thesis project are indicated by an empty bullet.

3.2 Segmentation Method

After the max-tree is computed and refined, the components it contains are segmented in the third relevant stage: **filter**. The challenge here is to determine whether components on top of other components should be considered to be part of the same object, or to be distinct objects, one in front of the other. In the code by Moschini et al. [11], this is done by flagging nodes in the tree that are “significant”. Here, significance is determined by the flux (also referred to as power) of a component, which has to exceed a certain threshold associated with its area. These thresholds are not computed on-the-fly but hardcoded.

In order to support the extended collection of attributes, the mechanism that determines whether two components are part of the same object or not is replaced with a ML method. For this task, LVQ is chosen, as it is known to be simple, fast, configurable [16, 22] and provides meaningfull insight into the significance of individual attributes and correlation between them [17] as the prototypes can be interpreted directly within the space of the features that are used as input [3]. The latter characteristic will allow for later improvements in performance, by removing insignificant attributes from the procedure. Additionally, this allows future research and discussion on the thresholds found by the LVQ classifier in terms of feature values. Furthermore, Biehl, Hammer, and Villmann [3] state that the performance of LVQ has proven to be competetive for many classification problems (of which segmentation is a variant).

The specifics of LVQ and GLVQ classifiers are discussed in chapter 2. The features to be used as input for the classifier are simply the attribute values for each component in the max-tree, i. e.

$$\xi^X = \left\{A(X), \mathcal{P}(X), C(X), R_c(X), G(X), \mathcal{N}(X), \mathcal{E}(X), \mathcal{F}(X), \mathcal{S}(X)\right\} \tag{3.4}$$

with its label $y^X \in \{0, 1\}$, indicating whether the feature vector should be considered to be an astronomical object or not. As a result, a minimum of two prototypes are to be used in order to express the classifier, given that there are two classes to identify.

Chapter 4

Realization

In this chapter, the concepts introduced in chapter 3 will be implemented in the code produced by Moschini et al. [11]. The code is written in C. Prior to any implementation, some restructuring of the project files has been done to improve maintainability, as well as some minor documentation in the form of comments. The first steps of the realization will focus on the computation of the proposed attributes.

4.1 Computing Component Attributes

As described in chapter 3, the construction of the max-tree is done in two steps in MTO: **compute** and **refine**. In order to augment the prior stage, new steps are added to the attribute functions in the file `src/quanttree.c` that are called to initialize, update and finalize the attributes of a component, based on a pixel (i. e. element) that is determined to be part of it. The attribute `struct` itself is extended to support these new attributes. The new definition of this `struct` is included in listing 1. Attributes such as minimum, maximum, summed, etc., positions in each dimensions are updated each time a new pixel is added to the component. In order to be able to compute the perimiter during the **refine** stage, the boundary of the component is tracked as well in accordance with the definition of the algorithm presented in chapter 3. In order to be able to detect neighboring pixels of a lower level, the level of the component is stored as well. During the computation of the max-tree, MT can decide to merge components. In that case, the attributes of the emergent component must be recomputed from the respective attributes of the components to be merged. In the case of the minimum and maximum positions this is easily solved by taking the lowest minimum and highest maximum and the positional sums are added. In the case of the boundary, a new boundary vector can be constructed by added them element-wise.

During the **refine** stage, attributes are stored in a different `struct`: `Node`. The definition of the `struct` is included in listing 2. Here, computations can be made using the attributes gathered in the previous stage such as the dimensions of the component, mean positions of their elements, the composite attributes, etc.. Furthermore, the moment of intertia tensor is computed in order to be able to compute the four invariant moments. This does require this attributes to be computed before and of the moment can be computed, which in turn requires the summed and mean positions and dimensions of the component to be known. For each attribute, a separate function is defined. At three point during the refinement of the pilot tree, these function can be called in sequence: when the root node of the image is encountered, when

Listing 1 Definition of the `AttributeStruct`, extended with the component attributes to be computed in the **compute** stage of MTO as found in `src/common.h`. Note that this also includes redundant attributes that where already defined in an earlier version of MTO, e.g. `momVec`, `centralMomVec`, `CentralNormMomVec`, etc..

```
typedef struct {
    long level;

    long area;

    long minX;
    long minY;
    long minZ;

    long maxX;
    long maxY;
    long maxZ;

    long sumX;
    long sumY;
    long sumZ;

    long sumXX;
    long sumYY;
    long sumZZ;

    long sumXY;
    long sumXZ;
    long sumYZ;

    double sumIntSquare;
    double sumInt;

    long topleft_x, topleft_y, bottomright_x, bottomright_y;

    double momVec[VECLEN];
    double momVecGs[VECLEN];

    double *centralMomVec;
    double *centralMomVecGs;
    double *centralNormMomVec;
    double *centralNormMomVecGs;

    long *perimeter;
    long *boundary;
} AttributesStruct;
```

a child node is merged with its parent and when two sibling nodes are merged.

4.2 Segmentation with LVQ

As mentioned in chapter 3, the classification method of choice is LVQ. However, implementing a LVQ classifier from scratch in C is not a trivial task. In order to verify that a classifier performs as expected requires a lot of testing and experimenting. Therefore, an existing implementation is embedded into the source-code that is known to work correctly. Unfortunately, there are no proper candidates for this task that have a convenient C application programming interface (API). Therefore, another solution has been found in the form of LVQ for SKLEARN (SKLVQ), a Python library implementing an LVQ and GLVQ. This library is build on top of the Scikit-Learn (SKLEARN) toolset, which provides many additional tools to find the optimal configuration of a classifier [12, 20]. In order to be able to use this library from a C code-base, the Python/C API must be utilized to communicate with the library living in the Python interpreter [4, 13]. This approach does have an impact on the performance of MTO, as the conversion between C objects and objects in the Python interpreter adds a significant amount of time.

4.2.1 Embedding Python in C

The `sklvq` package exposes the `GLVQClassifier` class, which can be provided with a configuration on initialization, specifying parameters such as the distance function, activation function, the number of prototypes per class, etc.. This class is an implementation of SKLEARNS `BaseEstimator`. Having an instance of this class, its method `predict(data)` is available, where `data` is an array-like structure consisting of columns and rows. In the case of the current implementation, rows represent individual components (or rather, the level-root nodes of each component) and columns consist of the component attributes. During the segmentation stage, the attributes of the node in the max-tree are extracted and stored in an intermediate Python `list`. This allows all of the attributes to be sent to the Python interpreter — and in turn to the LVQ classifier — adding to the efficiency of the procedure. Unfortunately, during the implementation of this proces, it has become apparent that the values of the invariant moments are not compatible with the Python interpreter. This might be caused by a fault in the computation of these values or they might simply be to high or require too much precision. In order to be able to continue development, these attributes have ultimately been excluded from usage in the Python interpreter.

For the benefit of the maintainablity of the code base, a separate C file has been created to contain all of the LVQ related code: `lvq.c`, and its associated header file `lvq.h`. These files provide a simple API that “wraps” the API of the `GLVQClassifier` object, following the *façade* design pattern. This pattern allows future researchers and developers to be spared from the troubles of dealing with the complicated setup and teardown procedures required by the Python interpreter [5]. Furthermore, it allows for the nodes to be presented to an LVQ interface as-is, without manualy extracting their attributes. The usage of `lvq.c` requires that the functions `initialize()` and `finalize()` are called before and after performing training or classification. Supporting persistence of the classifier, a file is loaded during initialization and stored during finalization, containing an instance of `GLVQClassifier`. If such a file does not exist, a new classifier is constructed upon initialization. As a result, the training of a classifier and the actual usage can be done in separate runs of the code and different models can be swapped between sessions. An added benifit is that these files can also be loaded in a Python script, which allows the values of the prototypes in the LVQ classifier to be inspected.

Listing 2 Definition of the `Node`, extended with the component attributes computed in the **refine** stage of MTO as found in `src/common.h`.

```
struct Node {
    pixel_t parent;
    greyval_t filter;

    long Area;

    long Width;
    long Height;
    long Depth;

    long double MeanX;
    long double MeanY;
    long double MeanZ;

    long double MeanXX;
    long double MeanYY;
    long double MeanZZ;

    long double MeanXY;
    long double MeanXZ;
    long double MeanYZ;

    long double MomentOfInertia[3][3];

    double Lambda[3];

    double Power;

    double PowerOld;
    double VolumeOld;

    double Compactness;
    double CircularityRatio;
    double Grayscale;

    double NonCompactness;
    double Elongation;
    double Flatness;
    double Sparseness;

    AttributesStruct *attributes;
};
```

4.2.2 Training the LVQ Classifier

The input for the `GLVQClassifier.fit(data, labels)` method consists of the computed attributes of a given node in the max-tree and a flag indicating whether that node is considered to be *significant*. This consideration is based on whether the node has been marked as an astronomical object in the ground-truth. Such a ground-truth is simply the output to be expected for a given input image and can be produced either manualy or using a simulation. To ensure that the max-tree used during the training of the LVQ classifier is identical to the one used during segmentation, the same code is used for both processes. When the refinement of the max-tree is finished, either the training or segmentation procedure is initiated, based on whether a ground-truth has been specified or not. An activity diagram has been included in fig. 4.1 to illustrate the branching in the process described here. After the training has been completed, the classifier is stored and can either be trained more using other ground-truths or be used for segmentation.

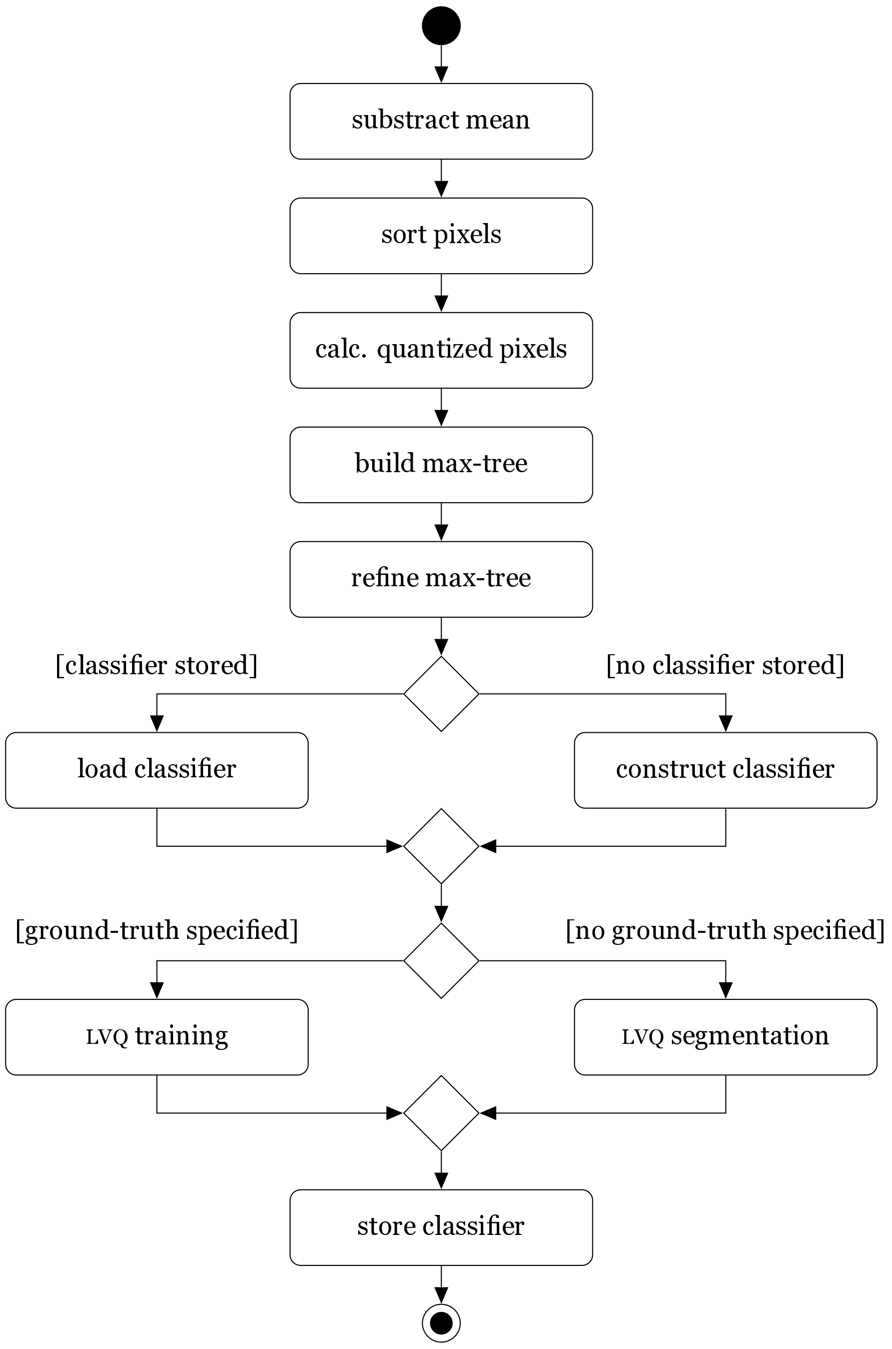


Figure 4.1: Activity diagram of the produced code base, displaying the new LVQ training branch, used to construct or update a classifier.

Chapter 5

Evaluation

Here, the behaviour and configuration of the MTO with LVQ classifier is evaluated in order to ensure optimal performance. This is mainly aimed at finding the right parameters, but does not go into great depths as many different configurations are possible. Furthermore, some optimization of the source-code is evaluated.

5.1 Hyperparameter Tuning

The Python module SKLVQ features support for a *grid search* of the parameter values available for the LVQ classifier. This way, the optimal configuration for a certain scoring parameter such as accuracy, precision and recallibility can be found [12]. This tool is used to find the optimal configuration for the following LVQ classifier parameters:

- Distance Type
- Activation Type
- Solver Type
- Prototypes per Class

In this section, each parameter is evaluated using a grid search. Parameters not being tested are set to their respective default values, which is squared Euclidean for the distance type, identity for the activation type, steepest gradient descent for the solver type and one prototype per class. The grid search is performed using labeled data from the $200 \times 200 \times 4$ data-set, optimizing for the accuracy scoring objective. These comparisons will be based on the quality of the results — the mean fit score — and the time required to achieve this results — the mean fit time. Two distance types are available in SKLVQ: Euclidean and squared Euclidean. Performing the grid search results in an identical mean score for both methods: 0.98. This indicates that either method can be used to the same end for this specific type of data-set. Differences are found in terms of mean fit time however, as the Euclidean function is significantly slower that the squared Euclidean function, with times of 10.38 s and 7.48 s respectively. For the activation type, four different method can be used: identity, sigmoid, soft+, and swish. Again, the grid search results in identical mean score for all methods: 0.98. In terms of mean fit time some slight changes are however noticable. The sigmoid, soft+ and swish functions require a time of 8.56 s, 8.27 s and 8.91 s respectively. However, the identity function comes out as a clear winner with a mean fit time of 7.95 s. The solver types available in SKLVQ are: steepest gradient descent, adaptive gradient descent and adaptive moment estimation. The grid search tool returns an identical score for each of these, namely 0.98. However, the time required to fit the prototypes to the labeled data differs greatly. While adaptive gradient descend and adaptive moment estimation have a mean fit time of 111.93 s and 105.95 s respectively, steepest gradient descent only requires a mere 9.68 s. Increasing the number of prototypes does not improve the mean test score of 0.98. This does however increase the mean fit time, leading to the assumption that the number of prototypes per class should be kept to a minimum of one.

Chapter 6

Results

In this chapter, the results produced by the MTO with LVQ classifier, given the research questions prompted in chapter 1, are evaluated. In order to produce the results presented in this chapter, MTO has been configured according to the optimum found by Haigh et al. [7], i. e. $\lambda = 1, \sigma = 0.00$ and a *move-up factor* of 0. Furthermore, 16 threads are used during the quantization of the max-tree and 32 bit are allocated per pixel.

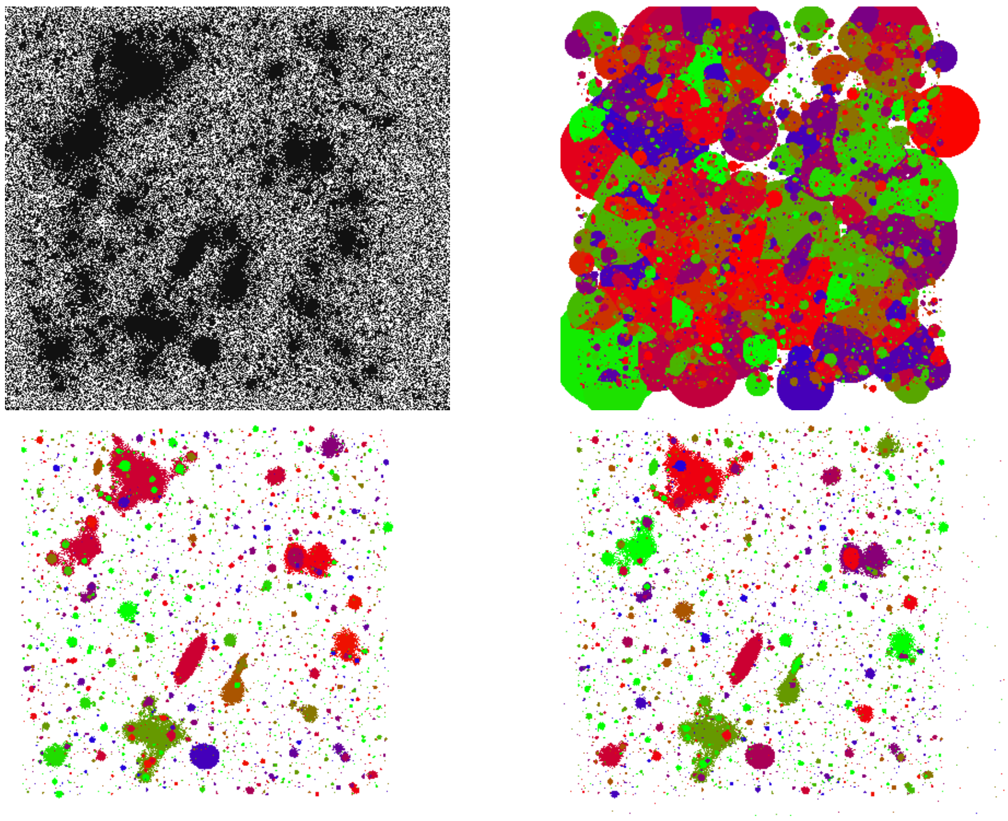


Figure 6.1: Input data set `cluster_10.fits` (top left), ground truth (rop right) and segmentations using the statistical method (bottom left) and the LVQ classifier (bottom right).

6.1 Segmenting Astronomical Data for Evaluation

To evaluate the quality of the results of the LVQ classifier, it is trained on one of the same labeled data sets used by Haigh et al. [7]. A total of 10 data sets are available, named `cluster[n].fits`. As stated, each data set has an associated *ground truth*, named `gt_[n].fits`. With these assets, a classifier can be trained on a first data set and ground truth and be used to classify a second data set, with its ground truth available to evaluate the results. The training scheme has been chosen as follows in order to mimic earlier work by Haigh et al. [7]: the classifier is trained on `cluster1.fits` and labels `gt_1.fits` and used to classify all other data sets. An example of such a classification is included in fig. 6.1. More of such classifications have been performed with the same classifier for the other data sets as well. Note that data set 5 turned out to be corrupted, reducing the total amount of data sets by one. Given the fact that the attributes provided to the LVQ classifier consist of five values, one of which is of the data type **`long`** (32 bit) with the remainder having data type **`double`** (64 bit), an estimation can be made of the total size to be communicated between the C code and the Python interpreter. Assuming approximately nine milion nodes — as is the case with data set cluster 1 — the total amount of data to be processed is *at least* $9 \times 10^6 \times (32\,\text{bit} + 4 \times 64\,\text{bit}) \approx 26 \times 10^9$ bit, about 3.20 GB. It is evident that the time needed for processing such an abbundance of data takes a lot of time, even thought the actual information it represents is already contained in a subset of the total data set. Therefore, the attributes to be provided to the classifier is restricted to those of 10 % of the total number of nodes. In the case of the example provided earlier, this reduces the the data to communicate to at least 324 MB.

6.2 Quantifying Segmentation Quality

The presented segmentation produced by MTO with LVQ shows that the large structures are identified like they are in MTO using the statistical method. However, the LVQ classifier also marked a much higher number of noise structured as objects. As a result, the segmentation is very much capable of recalling the structures in the input data, but is less precise in doing so than its statistical counterpart. In order to quantify this behaviour such that both methods can be compared, the quality of the segmentations is computed using a method presented by Haigh et al. [7]. This method results in (among others) four metrics based on the number of true positives, true negatives and false negatives:

- Detection recall or completeness: the proportion of objects in the ground-truth that have actually been detected;
- Detection precision or purity: the proportion of detections that can be matched to objects in the ground-truth;
- The F-score: the harmonic mean of precision and recall;
- The area score: an overall measure of the quality of the segmentation [7].

To compute these measurements, the position of the peak in each object defined in the ground-truth is determined. Then, the number of detections at these position are counted. This number is then devided by the total number of objects and total number of detections, resulting in the recall and precision measurements respectively. This procedure has been applied to the segmentations of eight data sets, using the MTO and LVQ classifiers. A scatter plot has been included in fig. 6.2, visualizing the precision vs. recall and the F-score vs. area score, as this is also the way in which segmentation are compared in [7]. Classifications have been performed for nine cluster data sets, for each classifier trained on one of the remaining data sets. An obvious difference between the measurements taken for MTO and MTO with LVQ is that the latter has a much lower precision as hypothesized. Again, this is due to the much higher number of noise structured being identified as object by the LVQ classifier. However, this classifier performs significantly better in terms of recall, meaning more of the actual object are identified as such as well. Comparing the F-score of both methods, it is clear that the LVQ classifier again performs at a much lower quality level, although the area scores are nearly identical.

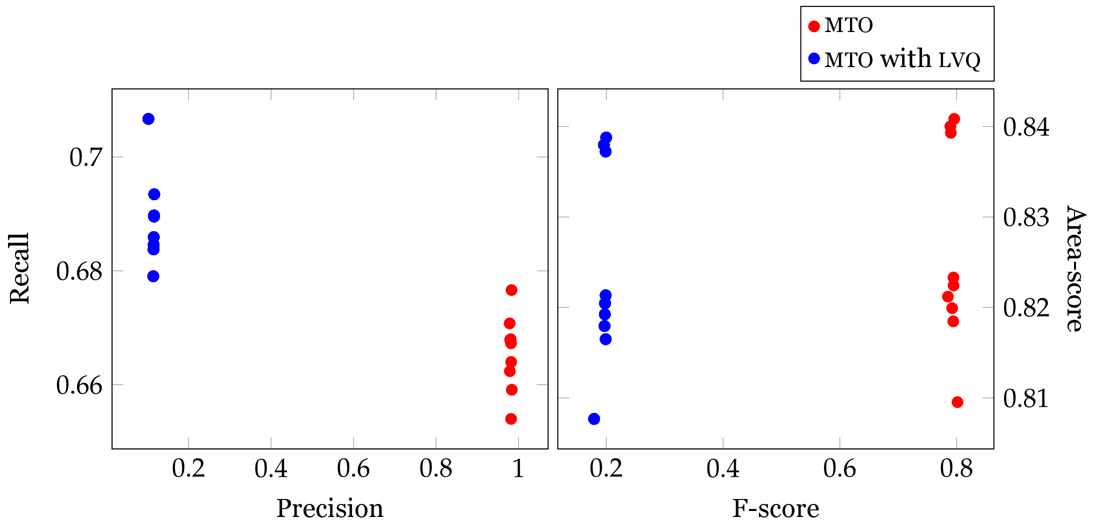


Figure 6.2: Comparison of the measurements computed for segmentations producted by MTO with LVQ and MTO using a statistical segmentation method. Here, LVQ classifiers have been trained for 10% of each cluster data set. For each classifier, segmentations have been produced for the remaining eight cluster data sets, resulting in $9 \times 8 = 72$ segmentations for MTO with LVQ.

A potential explenation for the low precision score of LVQ is the decision to train each classifier on a fraction (10 %) of the data sets. This might not be enough information for the classifier to learn the rule that separates noise from actual astronomical objects. To test the significance of this design choice, a single LVQ classifier has been trained on 100 % of the nodes found in the first cluster data set. Next, segmentations have been produced using this classifier for all of the remaining data sets. The measurements computed from these segmentations have been presented in fig. 6.3, together with the measurements computed from the segmentations produced with the LVQ classifier trained on 10 % of the nodes in cluster 1. Here it becomes apparent that there is little change in results when increasing the percentage of nodes considered during training. Bar two of the measurements, all have remained unchanged, proving that the low precision and F-score cannot be attributed to the number of nodes taken into consideration.

6.3 Comparing Time Measurements

Due to the usage of the Python interpreter to perform the LVQ classification, it is probable that MTO with LVQ requires more time to mark the significant nodes in the max-tree. The much larger number of nodes that are marked as objects will most likely also influence the post-processing of

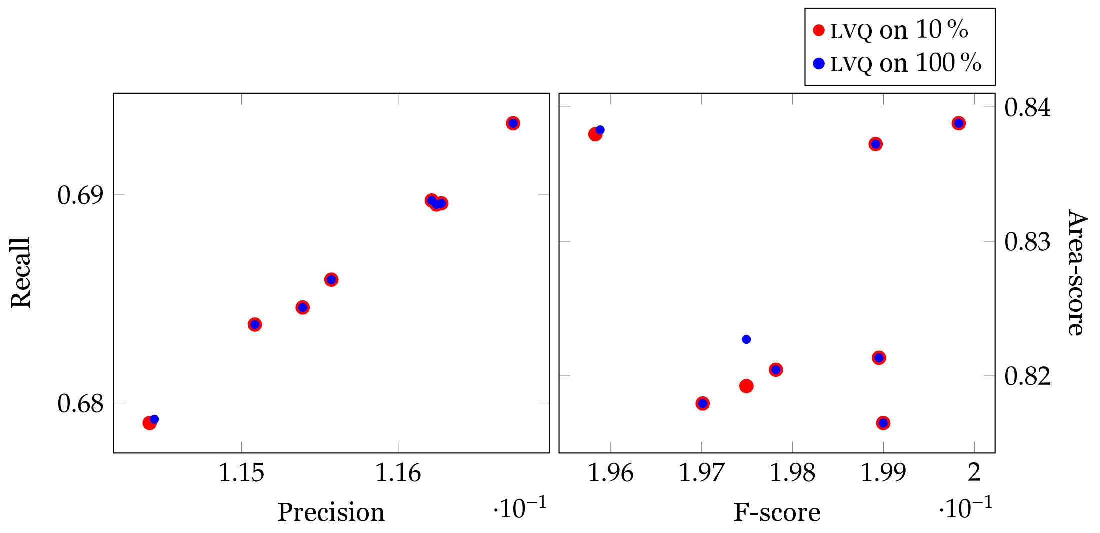


Figure 6.3: Comparison of the measurements computed from the segmentations of all data set bar 1 and 5, using MTO and LVQ trained on both 10 % and 100 % of the nodes in the cluster data set 1. Data points for the latter have been reduced in size to make overlaps with the prior visible.

the max-tree. In order to test this hypothesis, time measurements have been collected per stage of the segmentation process, as described in chapter 3. Data for both segmentation using LVQ and the statistical method has been recored, both of which are presented in a box plot in fig. 6.4. Here it is apparent that MTO with LVQ indeed takes a lot more time to mark the significant nodes. The difference in measurements before this stage are all comparable, but after most stages take longer for MTO with LVQ, even the generation of the output image. This is best explained as a result of the increase in nodes that have been marked as significant.

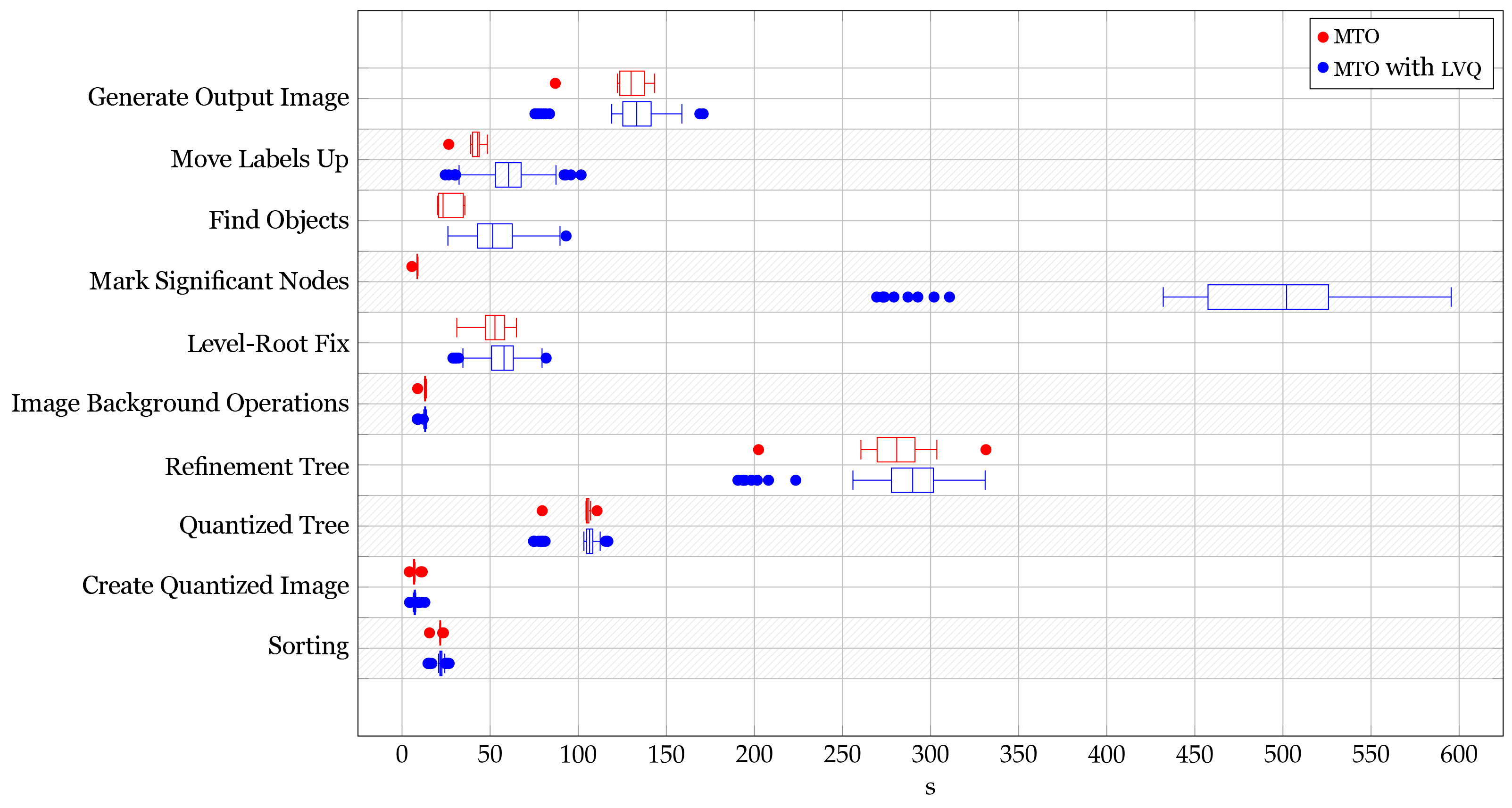


Figure 6.4: Comparison of the time measurements taken for the segmentation of the cluster data sets using MTO with LVQ and MTO with the statistical method. To construct this box plot, 72 and 9 measurements have been used for MTO with LVQ and MTO with the statistical method respectively. Note that the stages in the segmentation process have been presented in chronological order on the y-axis, from bottom to top.

Chapter 7

Conclusion

Given the results gathered in chapter 6, research question 2 as posed in chapter 1 can be answered in this chapter. Recall that research question 1 has been answered in chapter 3 Furthermore, the result that have been produced are discussed in order to evaluate the knowledge that has been gained during the research.

7.1 Segmentation Quality

Although the segmentation produced by MTO with LVQ results in a classification of astronomical object in a visual sense, measurements indicate that the quality of MTO with a statistical method is far superior. This observation can be attributed to the tendency of LVQ to mark small structures in the background noise to be object as well, resulting in a very low precision and F-score. The results are improved in terms of recall and area-score, but as such low rates that the loss in precision and F-score are not justified. Similar behaviour of ML techniques for segmentation problems have been observed in other research, such as the segmentation of tumors in positron emission tomography (PET) scans by Tata [18]. This leads to the conclusion that ML is unsuited for the segmentation of astronomical data, at least on its own. This provides an answer to research question 2.

7.2 Performance

Besides the quality of the segmentation results, MTO with LVQ also increases the time and resources required for segmentation. Measurements show computation time up to ten times as long as MTO using the statistical method. This increase in time needed would be acceptable in case of an improvement in quality of the results. However, improvements in implementation of the LVQ classification might still result in faster segmentation than MTO. This could lead to MTO with LVQ being used as a preliminairy segmentation method, potentially justifying further research.

Chapter 8

Future Work

In this chapter, possible subjects for future research and development is discussed, based on the findings presented in this master's thesis.

8.1 Determining Significance

With the current implementation, all nodes in the max-tree are presented to the LVQ classifier. Not only does this slow down the process of learning and segmentation, it also results in a lot of redundancy in terms of new information presented to the classifier. Developing a more sophisticated procedure in which only relevant nodes make it to the classifier will most likely improve the results. For example, instead of whole branches in the max-tree, only nodes at which new branches diverge could be considered, as this is where objects are more likely to be observed. More approaches can be formulated by examining the behaviour of MTO and MTO with LVQ and determining where it makes mistakes. Using this information, the way in which nodes are flagged to be significant can be improved.

8.2 Parameter Optimization

As discussed in chapter 6, many options are available for the configuration of an LVQ classifier. In this thesis, only the impact on the time measurements has been considered for the selection of these options. Furthermore, only the options that are implemented in SKLVQ at the time of writing have been considered. Future research might focus on further optimization of the parameters in order to improve the actual quality of the results presented by the LVQ classifier. To this end, the measurements considered in section 6.2 can be used to evaluate the impact of different configurations.

In addition to optimizing the parameters of the LVQ classifier, alternative distance functions might also be considered. In this work, the Euclidean distance has been used, but other distance measures might be more sensible, given the nature of the data. Some promising candidates for distance measures are the *Minkowski distance*, the *Mahalanobis Distance* and the *kernalized distance* [3]. Of these, the Mahalanobis distance shows the most potential, as it is noted to perform better with respect to the Euclidean distance when features correspond to different properties in both a quantitative or qualitative sense.

8.3 LVQ Implementation

The time measurements suggest that the use of the Python interpreter is a major impact on the run-time of the MTO with LVQ software. More methods of implementing an LVQ classifier with MTO should be explored in order to find one which reduces the required computation power and time. Possible solutions of interest are implementing both MTO and LVQ in Python, rewriting parts in C for optimization (instead of the other way around as done in this thesis), implementing LVQ in C as a whole or finding a more suitable language than C or Python altogether. Additionally, the implementation object detection algorithm might be improved as well. In the current implementation, every node is used as input for the LVQ classifier. However, it is not unlikely that only nodes appearing at points in the tree where new branches appear are probable to be an object. More sophisticated approaches to selecting nodes on which the classifier is trained can be imagined as well. Limiting the data to use as input for the classifier training procedure to these nodes only might increase precision while reducing the amount of computational power required. More research is required to verify whether such a sophistication of the software will be an improvement in terms of quality.

8.4 Alternative Applications

In this research it has been shown that MTO with LVQ does not perform well when it comes to the segmentation of astronomical data. Whether this behaviour is limited to astronomical data only is unclear, although other research does report on similar observation in medical scans at least [18]. Still, other types of data sets might be suit the LVQ classifier better that the aformentioned types. Therefor, further research is needed to investigate the performance of LVQ for different data-sets. Possibly, increasing the number of nodes to consider during training might result in better results in some cases. Furthermore, in this thesis, only optical data-sets have been used for training and segmentation. Another interesting group of data-sets are those containing radio-astronomical data. Different from optical data, radio data can contain certain patterns in the noise that make it hard to find segments. It is possible that LVQ is capable of classifying objects in these data-sets, extracting valuable information. Therefore, more research into the performance of LVQ in a radio-astronomical context is worth looking into as well. In addition to astronomical data, MTO with LVQ might prevail in other fields of applied computer vision as well. For example, analyzing medical scans might lead to the detection of certain deceases. In order to explore the extent in which this technique can be used in other domains, more research is required.

8.5 Improving Statistical Segmentation

Although MTO with LVQ has shown to perform less well than MTO using a statistical method, it might still be of some use as an additional step in the segmentation process. To put this idea to the test, the a data set can be segmented by MTO, the output of which is used as a ground truth to train an LVQ classifier. Then, this classifier can be used to segment the input image again, hopefully resulting in an improvement with resepect to the result by MTO on its own due to the combination of the high precision provided by MTO and the high recall by MTO with LVQ. In fig. 8.1 a diagram is featured, illustrating this process. Using the comparison methods featured in this thesis, the effect of this approach can be inspected by future researchers.

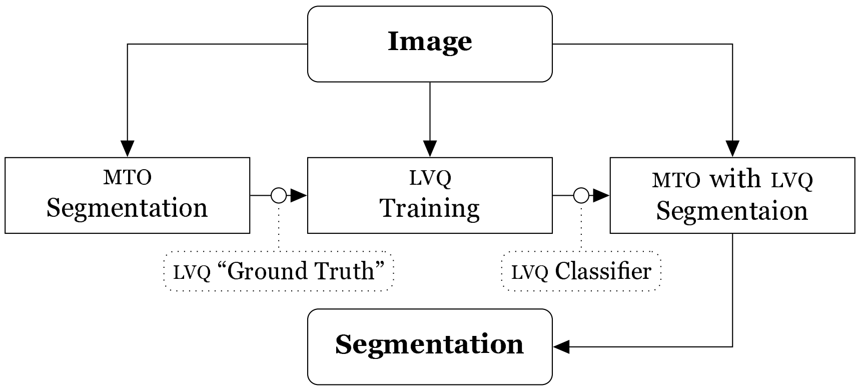


Figure 8.1: Diagram of the proposed process in which MTO with LVQ and MTO using the statistical method can be combined in order to potentially increase the quality of the resulting segmentation.

Bibliography

[1] Christophe Berger et al. “Effective Component Tree Computation with Application to Pattern Recognition in Astronomical Imaging”. In: *Proc. International Conference on Image Processing*. Vol. 4. San Antonio, Texas, US: IEEE, Sept. 2007, pp. 41–44. ISBN: 978-1-4244-1436-9. DOI: `10.1109/ICIP.2007.4379949`.

[2] Michael Biehl. “Learning from Examples. An Introduction to Neural Networks and Computational Intelligence”. Lecture materials for the course Neural Networks and Computational Intelligence. Jan. 2, 2019.

[3] Michael Biehl, Barbara Hammer, and Thomas Villmann. “Prototype-Based Models in Machine Learning”. In: *WIREs Cogn. Sci.* 7 (2016), pp. 92–111. DOI: `10.1002/wcs.1378`.

[4] *Extending and Embedding the Python Interpreter.* Python 3.8.2 Documentation. Python Software Foundation. 2020. URL: `https://docs.python.org/3/exending/` (visited on 04/21/2020).

[5] Eric Freeman et al. *Head First. Design Patterns. A Brain-Friendly Guide*. Ed. by Mike Hendrickson and Mike Loukides. Sebastopol, CA, US: O’ Reilly, 2004. ISBN: 978-0-5960-07126.

[6] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*. Ed. by Michael McDonald. 3rd ed. Upper Saddle River, New Jersey, US: Pearson Education, 2008. ISBN: 978-0-13-505267-9.

[7] Caroline Haigh et al. “Optimizing and Comparing Source Extraction Tools Using Objective Segmentation Quality Criteria”. 2020.

[8] Ming-Kuei Hu. “Visual Pattern Recognition by Moment Invariants”. In: *IRE Trans. Information Technology* 8 (2 Feb. 1962), pp. 179–187. DOI: `10.1109/tit.1962.1057692`.

[9] Teuvo Kohonen. “The Self-Organizing Map”. In: *Neurocomputing* 21 (1–3 Nov. 1998), pp. 1–6.

[10] Ugo Moschini, Arnold Meijster, and Michael H. F Wilkinson. “A Hybrid Shared-Memory Parallel Max-Tree Algorithm for Extreme Dynamic-Range Images”. In: *IEEE Trans. Pattern Analysis and Machine Intelligence* 40.3 (Mar. 2018), pp. 513–526. DOI: `10.1109/TPAMI.2017.2689765`.

[11] Ugo Moschini et al. “Towards Better Segmentation of Large Floating Point 3D Astronomical Data Sets: First Results”. In: *Proc. 2014 Conference on Big Data from Space*. Ed. by P. Soille and P. G. Marchetti. European Comission. Luxembourg, LU: European Union, Nov. 2014, pp. 218–221. ISBN: 978-92-79-43252-1. DOI: `10.2788/1823`.

[12] Fabian Pedregosa et al. “Scikit-Learn. Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[13] *Python/C API Reference Manual*. Python 3.8.2 Documentation. Python Software Foundation. 2020. URL: https://docs.python.org/3/c-api/ (visited on 04/15/2020).

[14] Rubin Observatory. *First National US Observatory to be Named After a Woman!* National Science Foundation. Jan. 6, 2020. URL: https://www.lsst.org/news/vro-press-release (visited on 07/01/2020).

[15] Philippe Salembier, Albert Oliveras, and Luis Garrido. "Antiextensive Connected Operators for Image and Sequence Processing". In: *Trans. on Image Processing* 7.4 (Apr. 1998), pp. 555–570. DOI: 10.1109/83.663500.

[16] Atsushi Sato and Keiji Yamada. "Generalized Learning Vector Quantization". In: *Proc. 8th International Conference on Neural Information Processing Systems*. NIPS'95. Ed. by David S. Touretzky, Michael C. Mozer, and Michael E. Hasselmo. NeurIPS. Cambridge, Massachusetts, US: MIT Press, Nov. 1995, pp. 423–429.

[17] Petra Schneider. "Advanced Methods for Prototype-Based Classification". PhD thesis. Groningen, NL: University of Groningen, 2010. ISBN: 978-90-367-4405-8.

[18] Elsie Tata. "Segmentation of NSCLC Tumors in PET Scans Using Component-Trees and Machine-Learning". Unpublished MSc thesis. Groningen, NL: University of Groningen, 2020.

[19] Florence F. Tushabe. "Extending Attribute Filters to Color Processing and Multi-Media Applications". PhD thesis. Groningen, NL: University of Groningen, Nov. 2010. ISBN: 978-90-367-4630-4.

[20] *User Guide*. Scikit-Learn 0.22.2 Documentation. Scikit-Learn Developers. 2019. URL: https://scikit-learn.org/stable/user_guide.html (visited on 04/15/2020).

[21] Michael A. Westenberg, Jos B. T. M. Roerdink, and Michael H. F. Wilkinson. "Volumetric Attribute Filtering and Interactive Visualization Using the Max-Tree Representation". In: *IEEE Trans. Image Processing* 16.12 (Dec. 2007), pp. 2943–2952. DOI: 10.1109/TIP.2007.909317.

[22] Whenjun Zhang. *Computational Ecology. Artificial Neural Networks and Their Applications*. Toh Tuck, SG: World Scientific Publishing, 2010. ISBN: 978-981-4282-62-8.

Appendix A

Implementing Berger's Max-Tree Algorithm

As part of the exploratory research for this master's thesis, the max-tree construction algorithm proposed by Berger et al. [1] has been implemented for GNU Octave. This implementation is mainly meant to gain understanding of the inner mechanisms of the algorithm and does therefor not make any promises regarding performance or efficiency. However, it might be usefull for the construction and inspection of max-trees for small images. Therefor, the code is made available in the Git repository of this thesis as well as in this appendix. Additionally, the repository provides a few example data-sets as well as a visualization script to display the max-tree of an image, such as the one presented in fig. A.1. The implementations of the procedures listed in chapter 2 are included in listings 3 to 5.

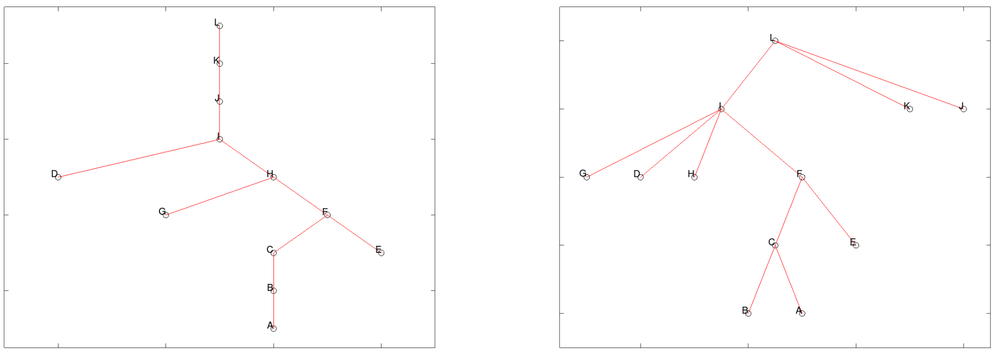


Figure A.1: Example tree plots of example data-set `src/attr/eg.mat` in the "correct" (left) and canonized (right) form. Plotted using the `src/attr/show.m` GNU Octave function. Note that the trees are upside-down and that the layer (λ value) is not reflected in the vertical positions of the nodes of the tree.

Listing 3 Implementation of the FINDROOT procedure, as found in the file `src/attr/root.m`.

```
function [r, zpar] = root(x, zpar)
    if zpar(x) == x
        r = x;
    else
        zpar(x) = root(zpar(x), zpar);
        r = zpar(x);
    end
end
```

Listing 4 Implementation of the COMPUTETREE procedure, as found in the file `src/attr/tree.m`. Note that the main loop iterates over the indices of the reverse sorted elements as opposed to the formal definition of the algorithm by Berger et al. [1], where the elements are iterated directly.

```
function [R, parent] = tree(f)
    parent = zpar = nan(size(f));

    [_, s] = sort(f(:), "descend");
    [_, R] = intersect(s, [1:numel(f)]);

    for p = s'
        parent(p) = p;
        zpar(p) = p;

        for n = N(p, f)
            if isnan(zpar(n))
                continue;
            end

            [r, zpar] = root(n, zpar);

            if r != p
                parent(r) = p;
                zpar(r) = p;
            end
        end
    end

    R = reshape(R, size(f));
end
```

Listing 5 Implementation of the CANONIZETREE procedure, as found in the file `src/attr/canonize.m`. Similar to listing 4, the main loop iteration does not iterate over the elements directly but their indices instead.

```
function parent = canonize(parent, f)
    [_, s] = sort(f(:), "descend");
    [_, R] = intersect(s, [1:numel(f)]);

    for p = fliplr(s')
        q = parent(p);

        if f(parent(q)) == f(q)
            parent(p) = parent(q);
        end
    end

    parent = reshape(parent, size(f));
end
```

Appendix B

Implementing Component Attribute Computations

In order to investigate the way in which the computation of component attributes could be realized, several of them have been implemented in GNU Octave, based on the algorithms provided in chapters 2 and 3. The function provided in this appendix are also available in the Git repository of this master's thesis and can be used to computed their respective attribute for the components in a max-tree constructed using the source-code provided in appendix A. A selection of the implementation have been included in listings 6 to 10.

Listing 6 Implementation of the COMPUTEAREA procedure, as found in the file `src/attr/attr/area.m`.

```
function a = area(f, R, parent)
    a = ones(size(R));

    [_, v] = sort(R(:));

    for x = v'
        if parent(x) == x
            continue
        end

        a(parent(x)) = a(parent(x)) + a(x);
    end
end
```

Listing 7 Implementation of the COMPUTEPERIMETER procedure, as found in the file `src/attr/attr/per.m`.

```
function l = per(f, R, parent)
    b = zeros(size(f));
    l = zeros(size(f));

    for a = unique(f)'
        g = f >= a;
        c = zeros(size(f));

        c = c | g > [zeros(1, size(g, 2)); g(1:end-1, :)]; % n
        c = c | g > [zeros(size(g, 1), 1), g(:, 1:end-1)]; % e
        c = c | g > [g(2:end, :);   zeros(1, size(g, 2))]; % s
        c = c | g > [g(:, 2:end),   zeros(size(f, 1), 1)]; % e

        b = b + c;
    end

    [_, s] = sort(f(:), "descend");

    for x = s'
        y = x;

        while b(x) > 0
            l(y) = l(y) + 1;
            b(x) = b(x) - 1;
            y = parent(y);
        end
    end
end
```

Listing 8 Implementation of the COMPUTECOMPACTNESS procedure, as found in the file `src/attr/attr/comp.m`.

```
function c = comp(f, R, parent)
    a = area(f, R, parent);
    p = per(f, R, parent);
    c = (p .^ 2) ./ a;
end
```

Listing 9 Implementation of the COMPUTECIRCULARITYRATIO procedure, as found in the file `src/attr/attr/ratioc.m`.

```
function r = ratioc(f, R, parent)
    a = area(f, R, parent);
    p = per(f, R, parent);
    r = (4 * pi .* a) ./ (p .^ 2);
end
```

Listing 10 Implementation of the ComputePower procedure, as found in the file `src/attr/attr/pow.m`. Note that this implementation does not follow the computation provided by Tushabe [19], as it follows the procedure as it appeared in MTO developed by Moschini et al. [11].

```
function c = pow(f, R, parent)
    c = zeros(size(f));
    h = zeros(numel(f), length(unique(f)));

    [_, s] = sort(f(:), "descend");

    for x = fliplr(s')
        y = x;

        while true
            h(y, f(x)) = h(y, f(x)) + 1;

            if y == parent(y)
                break;
            end

            y = parent(y);
        end
    end

    for x = s'
        for v = unique(f)
            c(x) = c(x) + h(x, v) * (f(x) - v);
        end
    end
end
```

Listing 11 Implementation of the ComputeSumInt procedure, as found in the file `src/attr/attr/sum.m`.

```
function s = sum (f, R, parent)
    [_, i] = sort(f(:), "descend");

    s = f;

    for x = i'
        if x != parent(x)
            s(parent(x)) = s(parent(x)) + s(x);
        end
    end
end
```

Listing 12 Implementation of the ComputeSumIntSquared procedure, as found in the file `src/attr/attr/sums.m`.

```
function s = sums (f, R, parent)
    [_, i] = sort(f(:), "descend");

    s = zeros(size(f));

    for x = i'
        s(x) = f(x) ^ 2;
    end

    for x = i'
        if x != parent(x)
            s(parent(x)) = s(parent(x)) + s(x);
        end
    end
end
```

Listing 13 Implementation of the ComputeGrayScale procedure, as found in the file `src/attr/attr/grayscale.m`.

```
function g = grayscale(f, R, parent)
    g = zeros(size(f));
    a = area(f, R, parent);

    [_, s] = sort(f(:), "descend");

    for x = s'
        y = x;

        while true
            g(y) = g(y) + f(x) / a(y);

            if y == parent(y)
                break;
            end

            y = parent(y);
        end
    end
end
```

Appendix C

Breaking the Python/C API

Spam, Spam, Spam, Spam, Spam, Spam, baked beans, Spam, Spam, Spam and Spam

Monty Python

While working with the Pytohn/C API, several setbacks have occured due to undocumented, indistinct technical problems. In this appendix, these challenges are described and the solution that has been found is explained (if applicable).

C.1 Debugging the Python Interpreter

During software development, debugging is vital in order to detect and inspect potential issues in the code. During this master's thesis project, the CLion[1] integrated development environment (IDE) has been used, which provides tools to place *breakpoints* within the C code. Unfortunately, debugging the behaviour of the Python Interpreter is less straightforward. Moreover, errors are not even printed to the terminal out of the box. This can lead to a lot of initial frustration when code simply stop execution, only providing an exit code. Attaching a debugger to the Python Interpreter when constructing using the Python/C API has proven to be unfeasible. However, a solution has been found to get a hold of errors that occur within the interpreter. To this end, after *every* execution within the Python Interpreter, the error flag has to be checked and — if it has been set — the error message is printed and flushed[13]. A demonstration of this approach has been included in listing 14.

C.2 Considering Performance

In early implementation of MTO with LVQ, data-points where sent to the Python interpreter one by one. This turned out to be a major delaying step within the process. However, allocating memory to store all data-points and sending all afterwards was not viable, as the amount of space required is unknown. This problem has ultimately been solved by the construction of a Python list, which can be expanded whenever necessary. Although this practice is discouraged

[1]https://www.jetbrains.com/clion/

Listing 14 Demonstration of the way in which errors that occur in the Python Interpreter can be inspected when using the Python/C API.

```
PyObject_CallMethod(pJobLin, "dump", "O, s",
                    pClassifier, classifier);

if (PyErr_occurred() != NULL) {
    PyErr_Print();
    PyErr_Clear();
}
```

in literature such as [13] due to performance issues, the performance lost is still significantly less, with the time required for training the LVQ classifier being reduced to roughly 6%.