Contents

Chapter 1

Introduction

Melanoma is a very aggressive form of skin cancer that develops from the pigment-containing cells known as melanocytes and it's one of the most dangerous type of skin cancer. An early diagnosis is essential to increase survival chances. An abnormal evolution in size of a skin lesion can be a symptom of melanoma. Understanding how skin lesions evolve over time can be useful to set an upper bound to the normal lesion size variations.
The MoleMapper [1] application assists dermatologists in the diagnosis of skin cancers through an automated analysis of the lesion images. We implement its segmentation algorithm to analyze a dataset of dermatoscopic images and extract statistics about the normal variation of lesions area over time. The dataset was obtained from two years of use of the MoleMapper application installed in two tablets used by dermatologists.
First, we recover the data and create Java classes to use the data in the segmentation program.
Then, we develop a set of Java classes that perform the segmentation. We modify the algorithm to work with cropped images and we change the way area is computed to improve its precision.
We analyze some calibration images and then the entire dataset. Calibration images are important to determine the accuracy of our program. Comparing the results of the analysis with the calibration error values we can prove an idea about the average variation rate of lesions area.

Chapter 2

The dataset

The dataset is composed of two folders containing the directory trees of the lesion images and several database files. We have a total of 2781 high-resolution (3264x2448) lesion images and several database files containing information about the patients, the lesions and the visits. The total size of the dataset is 4GB.
For every patient, there are many lesions monitored by the MoleMapper application. For each lesion we have several images taken over a two year period.

2.1 Database reverse engineering

In the dataset folders, there are several SQLite database files with no ER diagram. Each file corresponds to an autosave of the MoleMapper application, and we need to find that with the higher amount of records.

Building the ER diagram Having no ER diagram of the database, the first thing we do is understand the relationships between tables and rebuild the complete diagram. Not all table fields have self-explaining names, therefore we need to understand their meaning by comparing the database with the image dataset.
We use DBVisualizer, a free database management software, that creates the relationships diagram given the database file. Figure 2.1 shows the resultant diagram.
Checking the correctness of the data, we find some anomalies in the relation-

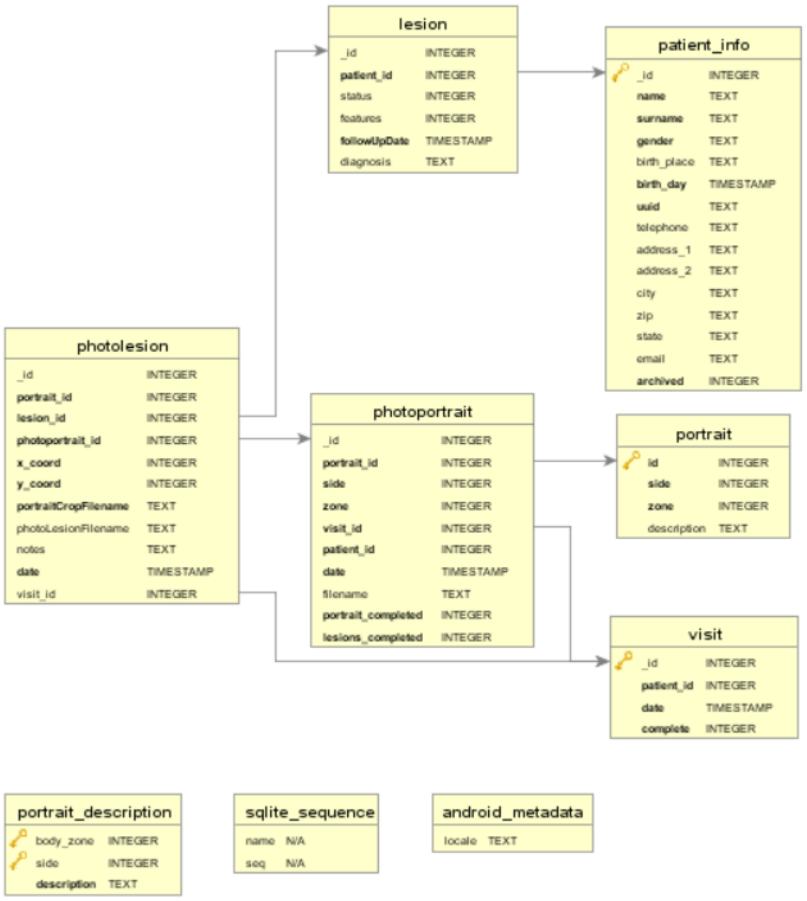

Figure 2.1: Database diagram extracted with DBVisualizer

ships. First, although the database diagram shows a relationship between photolesion_info, lesion, and patient, we observe that all records in the lesion table point to a patient with id = 0, which doesn't exist. After a fast analysis of the model, we find out that we can connect each lesion to the right patient through the visit table.
Each photo of a lesion has a photo portrait. A photo portrait is an image of the lesion body zone. The table photoportrait contains the fields side and zone, integers. We don't have any specifications of the meaning of these fields. Analyzing the portrait images in the dataset we can build a new table, portrait_description, that matches each (side, zone) pair with a description.

Finding the right database We have a few dozens of database files. To be sure that in the next steps we use the one with the higher number of records, we should not just take the most recent one. Instead, we build a short Java program that queries all the database files and returns the one with the highest number of record. We focus on the number of records in the photolesion_info table since we are mainly interested in analyzing the highest number of lesion images. By running our code, we find the two databases (from the two tablet applications) with the highest number of lesion photos. From now, we only use these two database files.

Missing images To check the consistency of the data, we write a Java test. The test compares the photolesion table with the images in the dataset and returns the number of records with no image in the dataset, and the number of records without an image path. Running the program, we find that every non-null path in the photolesion table has its image in the dataset, but there are 107 records with a null image path.

2.2 Mapping the database into Java objects

We map the database entities into Java objects so that we can use them in the next steps code. First, we decide the objects hierarchy that satisfies our research requirements. Then, we write the Java code that maps the required database information into the objects.

Objects hierarchy Our research requires us to analyze, for each patient, the evolution of a lesion over time. We observe that, for this purpose, the visit table is superfluous, since its only useful information is the date. We can then move this date field into the object related to the photolesion table. In the same way, the portrait and photoportrait tables may not be mapped, and the pair (side, zone) can be assigned directly to the related photolesion object. The resultant hierarchy, simple but still effective, is that of Figure 2.2.

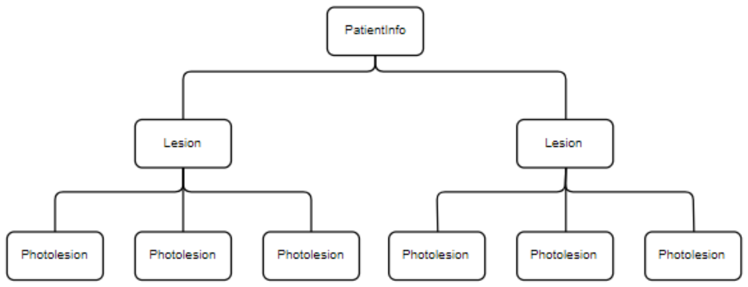

Figure 2.2: Objects hierarchy

Model We write the three classes of the model:

1. **PatientInfo:** contains the fields of the patient_info table and a list of Lesion objects.

2. **Lesion:** contains all the fields of the lesion table and a list of Photolesion objects.

3. **Photolesion:** contains all the fields of the photolesion table, the date and the pair (side, zone). Note that this object contains the path of the image, and doesn't load the Bitmap on instantiation due to the high number of records in the photolesion table.

These three classes will be used in the code of the next steps to access the entities information.

Mapper The Mapper class executes queries on the database, extracts the data and fills the objects. Starting from the records in the patient_info table, it creates an array of PatientInfo objects. Then, for each patient, the Mapper retrieves the lesion records belonging to that patient and fills its Lesion list appropriately. It does the same for the Photolesion list of each Lesion object. The mapper then returns the array of PatientInfo objects. We can now iterate patients, lesion, and images using the hierarchical structure.

Chapter 3

Segmentation

In this chapter, we present the segmentation algorithm and some particular adaptations we made. We then present our Java implementation.

3.1 Algorithm

The segmentation technique we use is the MEDS [1]. We apply a change to how we compute the lesion area once we have segmented its boundaries in order to achieve higher precision. The segmentation is obtained through the following stages:

1. **Principal Component Analysis:** We reduce the dimensionality of the given image to obtain a grayscale image.

2. **Color clustering:** We create the histogram of the image, and we find the threshold value above which a pixel should be considered lesional skin.

3. **Boundary detection:** We use a downsampling technique to find the boundaries of all lesional connected regions of the image.

4. **Area computation:** We compute the area of each connected region and return that of the largest and an array of its boundary points.

Principal Component Analysis PCA is a dimensionality reduction technique that linearly maps the data to a lower dimensional space, maximizing the variance of the mapped data and therefore losing the least information possible. In our case, we perform the PCA on the input RGB image, in which every pixel can be seen as a three-dimensional vector $\langle r, g, b \rangle$. First, we normalize the image by subtracting the R, G and B means to every pixel. Then, let m be the number of pixel in the image, we calculate the covariance matrix $\mathbf{C}$ as $\mathbf{C} = \mathbf{M^T M}$ where the ith row $\mathbf{m_i} = \langle r_i g_i b_i \rangle$ of the m x 3 matrix $\mathbf{M}$ represents the three color components of the ith pixel in the image. We compute the three eigenvectors of the matrix $\mathbf{C}$ and we take the dominant one, that corresponds to the highest eigenvalue of $\mathbf{C}$. The dominant eigenvector is the one over which we have the highest variance of the data. We then substitute each pixel with its projection onto the dominant eigenvector, obtaining a grayscale image.

Color clustering Starting from the grayscale image, we apply a thresholding to obtain a binary image, in which each pixel is either lesional or non-lesional. First, we create the histogram $h(x)$ = number of pixels with value x. We then apply a square root operator and a moving average operator to the histogram. The square root operator enhances the smaller values, whereas the moving average operator smooths the histogram. We have:

$$h'(x) = \sqrt{h(x)} \qquad h''(x) = \frac{1}{11} \sum_{y=x-5}^{x+5} h'(y) \tag{3.1}$$

The resulting histogram shows two peaks, corresponding to the two clusters of pixels: the skin ones, which we assume are higher in number, and the lesional ones. We call M_l the histogram value corresponding to the lesional color and M_s the value of skin color. M_s corresponds to the global maximum of $h''(x)$, whereas M_l is given by

$$M_l = \arg\max_x (h''(x)(h''(M_s) - h''(m_x))) \quad x \neq M_s \tag{3.2}$$

where $h''(m_x)$ is the minimum of h" between x and M_s.
We now determine the threshold T, given by

$$T = \arg\max_x \left((h''(M_s) - h''(x)) \left(\frac{x - M_l}{M_s - M_l} \right)^{\gamma} \right) \tag{3.3}$$

where γ is the single "tuning" parameter. A higher value of γ gives a tighter segmentation. We apply the threshold to the grayscale image to obtain a binary image, in which pixels above and below the threshold are respectively lesional and non-lesional.

Boundary detection We obtain the boundaries of the lesion by a two-step approach. First, we downsample the image to ease the detection of lesional connected component boundaries. Then, we scan the image, obtaining the boundaries and eliminating the false positives, taking only the largest. As defined in [1], a boundary pixel is a lesional pixel whose 4-neighborhood contains exactly 3 lesional pixels. In the downsampling phase, we partition the image into 3x3 boxes. Each pixel in the box takes the value of the central pixel. After downsampling, the eight-neighborhood of a boundary pixel contains exactly 2 boundary pixels. We can scan the image and, starting from each boundary pixel found, "walk" the boundary of each lesional component and obtain a set of disjoint cycle graphs.

Area computation We now calculate the area of each lesional component and select the bigger one. We observe that approach described in [1] can lead to small errors in the computation of the area. This method takes a connected component boundary and for each row selects all the pixels between two pairs of boundary pixels. This approach can count skin pixels as lesional when handling row vertices. Looking at Figure 3.1 we see that by taking pairs of boundary pixels in the same row we select pixels that are outside the connected component. We can eliminate these kinds of errors by using a different algorithm.
In our version, we use the algorithm described in [2] that uses the Surveyor's formula [3]. The algorithm takes a not self-intersecting polygon given by its vertex points in clockwise or counter-clockwise order as input and returns the area. Since our components are surely not self-intersecting (otherwise a boundary pixel would have more than three boundary pixels in its neighborhood) and we detect boundary pixels in clockwise or counter-clockwise order, our input meets the specifications.
The area A of a lesional connected component is then given by:

$$A = \frac{1}{2}\sum_{i=1}^{n} x_i(y_{\text{i+1}} - y_{\text{i-1}}) \tag{3.4}$$

in which when $i = n$ then $y_{\text{i+1}} = y_1$ and when $i = 0$ then $y_{\text{i-1}} = y_n$.

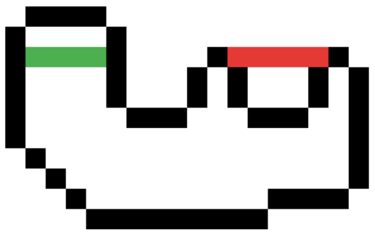

Figure 3.1: In green: area selected correctly. In red: wrongly selected area outside the component

3.2 Implementation

We develop a Java implementation of the algorithm above. The implementation consists of a collection of classes each of which represents a segmentation stage, and some additional classes to break the code into separate tasks. We use the mapped objects described in section 2.2 and the Apache Commons Math 3.6 library for matrix computation.

Image borders removal At the very first run of the segmentation program, we observe that the circular shadow created by the lens that took the photos has a very similar projection onto the dominant component to that of the lesions. Therefore, in the thresholding phase, a large part of the shadows is considered lesional tissue. We, therefore "crop" the images to exclude the circular shadow on the borders and other elements that can disturb our algorithm. The crop is done by a program we developed, which doesn't modify the image, but creates a binary mask for the pixels that have to be computed. We will discuss this program later in section 5.1. The binary mask is then translated into an array of **ImageRow** objects, one for each row, that contain the intervals (described by their endpoints) that have to be computed in that row. This achieves a little speed-up in the process since we don't have to check whether each pixel has to be computed, but we instead compute pixels from the right-bound pixel to the left-bound pixel. Since the crop area is considerably smaller than the size of the image, the total number of pixel operations is considerably smaller. All the following steps will use the array of ImageRow to perform operations on valid pixels only.

Structure The class we use to represent a matrix is **RealMatrix** from the Apache Common Maths library. This is an abstract class that provides methods for element access and matrix or vector operations such as multiplication, dot product, sums, and subtractions. We use **RealVector** to represent vectors, instead of using 1 x n RealMatrix objects. The algorithm implementation consists of these Java objects:

1. **Interval**: represents an interval, described by its endpoints.
2. **ImageRow**: represents a single row of an image, and contains the intervals of pixels that have to be computed in that row.
3. **ImageMatrix** extends RealMatrix: represents the input image as a matrix. Takes the image and the related array of ImageRow objects as input. It performs the PCA on the image and its elements (corresponding to the image pixels) are set with the projection values.
4. **Histogram** extends RealVector: represents the histogram of the image projection, for values from 0 to 255. It performs all the histogram operations and returns the thresholding value.
5. **BinaryMatrix** extends RealMatrix: represents the binary image as a binary matrix in which every pixel is either lesional or non-lesional.
6. **PostProcessing** extends RealMatrix: constructed from the binary matrix, visits the lesional connected components and marks boundary/non-boundary pixels. Returns a **PhotoLesionData** object.
7. **PhotoLesionData**: contains the output information of the analyzed lesion image, such as boundary pixels and area.

PCA First, we perform the PCA on the input image. The constructor of ImageMatrix takes a BufferedImage and an array of ImageRow as input. Let h and w be respectively the height and width of the image, we create the n x 3 RealMatrix **M**, where $n = hw$ the total number of pixels. The three columns of the matrix **M** represents the red, green and blue values of the n pixel rows. Since we consider only the valid pixels, given by the intervals of the ImageRow array, we fill the matrix $\mathbf{M}$ with only those pixels.

We then calculate the covariance matrix $\mathbf{C}$ as $\mathbf{C} = \mathbf{M}^{\mathbf{T}}\mathbf{M}$. We don't use the built-in methods of RealMatrix for this computation since the method multiply() performs the row per column multiplication for the whole resultant 3 x 3 matrix. Knowing that the covariance matrix $\mathbf{C}$ is symmetric, we can instead compute only the 6 products and copy the 3 upper values on the bottom half of the matrix.

We compute the three eigenvectors using the EigenDecomposition library class, and we take the dominant eigenvector V. We then iterate on every valid pixel of the image, computing its projection coefficient onto V and setting the correspondent element in the ImageMatrix with that value. Then, we scale the values in the [0,255] interval of integer numbers.

Color clustering The Histogram class extends RealVector and represents the histogram for values in the [0,255] discrete interval. Its constructor takes an ImageMatrix and the related array of ImageRow as input and sets the ith entry of the Histogram with the number of pixels with value i of the image matrix. It then performs the square and mean operation discussed in section 3.1. The Histogram class has the two methods *centers()* and *lesionThreshold()* that perform the operations described in section 3.1. The *centers()* method returns the two centers (peaks) of the histogram. The method lesionThreshold() returns the value of the threshold below which a pixel should be considered lesional. The γ valued cited in section 3.1 we use is a crucial factor for the precision of our analysis, and will be discussed later in chapter 4 .

Binary image The BinaryMatrix class extends RealMatrix and represents the binary matrix of the image. The constructor takes an ImageMatrix and the related ImageRow array and labels the BinaryMatrix entries as *LESIONAL* or *NONLESIONAL*.

In this phase, we perform an additional clean-up of the border shadows that may persist inside the cropped area. Knowing that the lesion doesn't touch the crop border, we consider as shadow every *LESIONAL* pixel touching the border. We, therefore, visit all the connected component of those pixels and

label it as *NONLESIONAL*.
We perform the visit using a queue. We put every shadow pixel in the queue and we iterate on the queue. At each iteration, we poll a pixel and we put in the queue every *LESIONAL* pixel touching it. The polled pixels are labeled as *NONLESIONAL*. This method could lead to the mislabeling of the lesion as shadow if some pixels of the shadow inside the cropped area touch the lesion, making it connected with the shadow. However, we observe that this happens so rarely in our dataset that we can manually run the segmentation algorithm on those images without this clean-up.

Boundary detection The PostProcessing class extends RealMatrix and performs the boundary detection and area computation phases described in section 3.1. We operate on the PostProcessing matrix instead of the BinaryMatrix because we need to change some entry values but we want to maintain the BinaryMatrix intact. The PostProcessing constructor takes the BinaryMatrix as input and sets its entries with the same values. After downsampling, we scan the image, starting from the left and every time we encounter a lesional pixel we start the boundary visit. The scan is done each dth-row, with d large enough to "jump" small clusters of pixels but tight enough to not miss the lesion.

The boundary visit starts from the discovered lesional pixel, labels it as *VISITED* and continues with the next. This assures that the lesion boundary is visited in clockwise or counter-clockwise order. When the visit doesn't find a non-visited boundary pixel near the last visited pixel, it has completed the boundary and stores the collection of boundary pixel coordinates in an array. At the end of this procedure, we have a set of disjoint boundaries.

Area computation We compute the area of each boundary found and select the largest, which we assume is the lesion. A simple algorithm that realizes the Surveyor's formula for computing the area is the following ([2]):

```
X = array of x coordinates, Y= array of y coordinates
polygonArea(X, Y, numPoints)
  $area \leftarrow 0$
  $j \leftarrow numPoints - 1$
  $i \leftarrow 0$
```

```
for $i < numPoints$ do
$area \leftarrow area + (X[j] + X[i])(Y[j] - Y[i])$
$j \leftarrow i$
end for
return $area/2$
```

Chapter 4

Dataset analysis

We use the classes describen in the previous chapters above to analyze our dataset. The resultant Java program can perform the segmentation on several images sequentially. The program prints the computed area values both in an XML format which can be easily read by many programming languages and a .dat format that can be read by Matlab.

First, we run the program on calibration images, to determine the accuracy of the segmentation and define the tolerance boundaries of the analysis. Then, we run the program on all dataset images and we represent the output in Matlab plots.

4.1 Analyzer program

The analyzer program can perform the analysis on several images sequentially, either for a single value of γ or for a range of values. Its core consists of the **Analyzer** and **MEDSProcessor** classes, whereas the output is managed by the **OutputManager** class.

Structure We present a brief description of the classes we use:

1. **OutputManager**: saves the analysis results in XML and .dat format.
2. **MEDSProcessor**: realizes the segmentation, starting from a Photolesion object and returning the result.

3. **Analyzer**: contains methods to run the analysis for single or multiple images, lesion or patients and sends the results to the OutputManager.

Analysis The Analyzer class can perform the analysis on a single or a range of patients, a single or a range of lesions of a given patient, or a single or a range of images of the same lesion. Every analysis can be done for a single value of γ or for a range of values. We decided to break the analysis of the entire dataset as described above due to the high computational time required for the complete analysis. The complete segmentation of a single image requires up to 8 seconds, plus the output time. We analyze each image for multiple values of γ, and therefore the analysis of the entire dataset requires 2781 x t_T x n_γ seconds, where t_T is the total time for the segmentation of a single image and n_γ is the number of different values of γ used in the segmentation.

Output The first kind of output is the debug images produced by the MEDSProcessor. For every stage of the segmentation, it produces an image of the output. We have then for each lesion image analyzed and for every value of γ used an image of the PCA result, one of the histogram, one of the binary image and one of the boundaries detected. These images are crucial to determine the conditions in which our algorithm fails and understand the causes. An example collection of these images is represented in Figure 4.1.

The second output is produced by the OutputManager and consists of an XML file and a collection of .dat files. The XML file contains all the analyzed data and has the following structure:

```
<ROOT>
  <LESION LESION_ID = [lesion id]>
      <PLOT_ENTRY AREA=[area], DATE_DAY=[day], DATE_MONTH=[month],
         DATE_YEAR=[year], GAMMA=[gamma], PLOT_ID=[photolesion id]>
  </LESION>
  .
  .
  .
</ROOT>
```

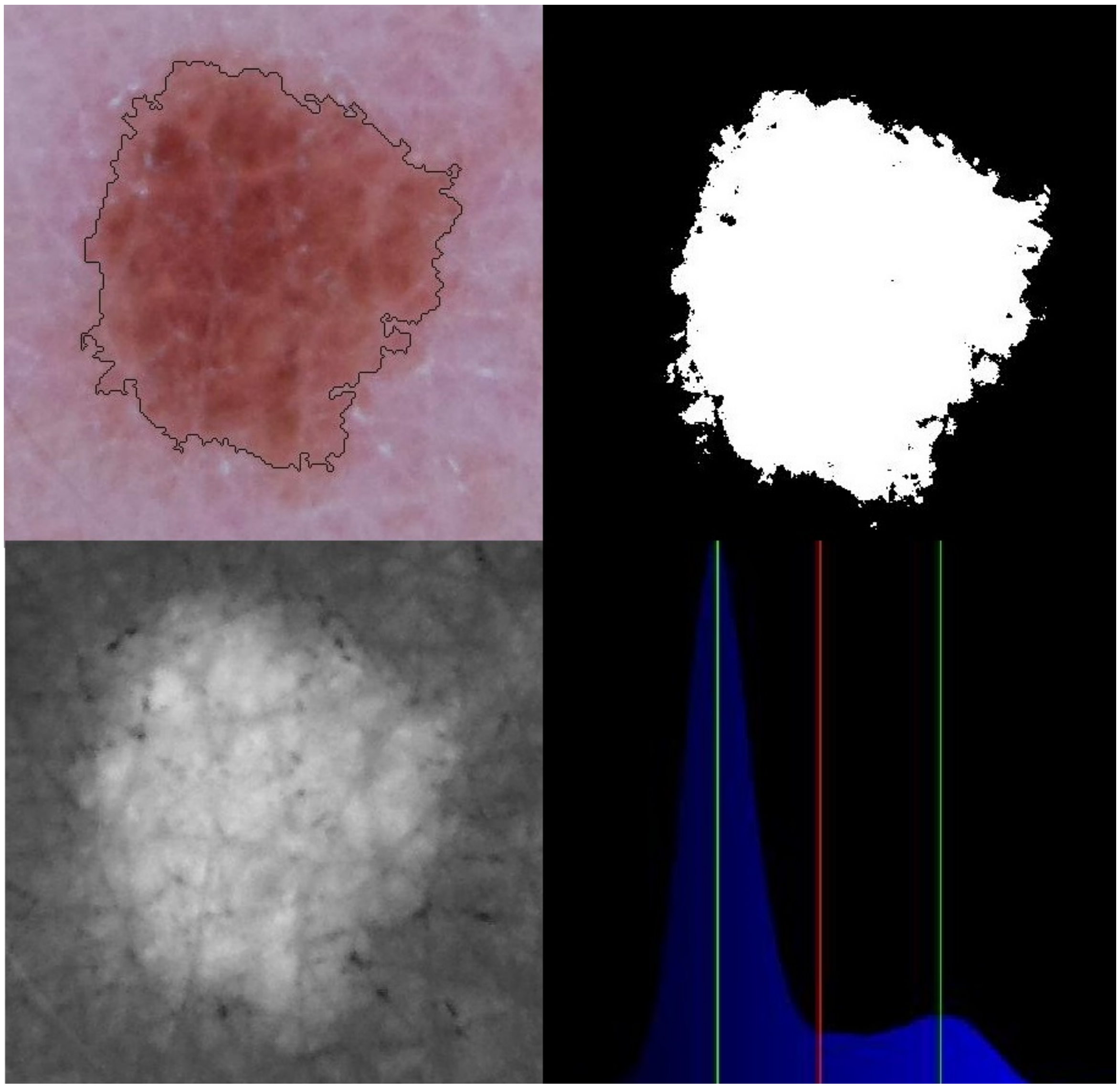

Figure 4.1: Sample of debug images

We don't take care of the patient in this structure since every patient's lesion is uniquely identified by its id. However, we group images of the same lesion under the correspondent lesion since we always consider them together.

A .dat file is produced for each lesion, with name lesion_id.dat. The .dat file is structured as a table with columns:
DATE | GAMMA | GAMMA | ....

1. The first row shows the gamma values from 0 to 2 with steps of 0.1. The first value of the first row is -1, since the first column is the date.
2. The following rows contain the data. The data is expressed as the number of days from the first image. The first data row has DATE = 0. The fields under the gamma values represent the area of the lesion in pixels.

An example of a .dat file is:

```
-1      0.1     0.2     0.3     0.4     0.5     ...     2
0       12311   12401   12678   12898   12901   ...     13999
7       12221   12801   13478   13598   13601   ...     14099
14      12131   12674   12778   12999   13401   ...     14467
```

This example file contains data from three images of the same lesion taken once a week. By reading the values of the same column from top to bottom we can determine the evolution of the lesion over time. Each column uses the γ value stated at its top. Reading values of the same row from left to right we can see how the segmented area changes by increasing the value of γ.

4.2 Analysis of the dataset

We use the analyzer program to extract the area of every lesion in the dataset. We analyze calibration images first, we discuss the results and then we analyze the whole dataset.

Gamma value A crucial point in the dataset analysis is the value of γ used in the thresholding phase. A lower gamma value results in a tighter segmentation whereas a higher value enlarges it, allowing to detect also the

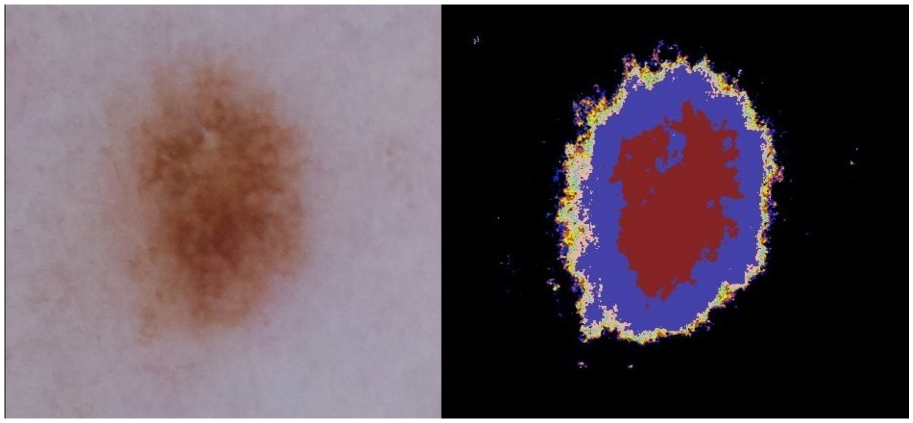

Figure 4.2: Lesion layers detected with increasing values of gamma (one for each color).

fuzzy areas around the lesion core. Figure 4.2 shows how the segmented area increases with the increase of the γ value in the range [0.1, 2] with steps of 0.1.

Calibration images We have two calibration lesions. Each lesion consists of a collection of six images taken on the same day, and we, therefore, expect the lesion area to not change significantly between images of the same lesion. Figure 4.3 shows the two calibration lesions that in this scope we will call lesion 1 and lesion 2. We see that lesion 1 has a darker core and fuzzier boundaries, while lesion 2 is more homogenous and has sharper boundaries. We run the analyzer program on the 12 images. Letting m be the mean area of a lesion and A a given area, we calculate the deviation $d(A) = \frac{A}{m} - 1$ for every image of the lesion. We repeat the process for different values of γ and we plot the results.

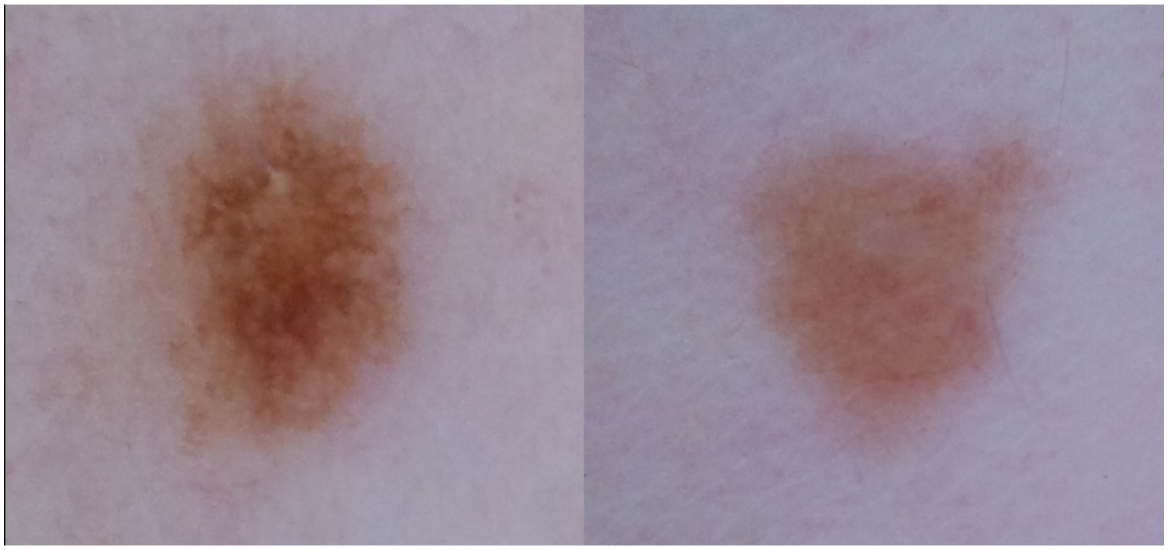

Figure 4.3: Lesion 1 on the left, lesion 2 on the right

In lesion 1 we have a lower accuracy than in lesion 2. This reflects the fact that lesion 2 has a more uniform color and sharper boundaries. We observe that while in lesion 1 the maximum deviation oscillates, in lesion 2 we observe a growing trend for growing values of γ. For what concerns the average deviation, it seems to grow with higher values of γ in lesion 1 while it is the opposite in lesion 2. Our algorithm then results to be more accurate with lesions that have sharper boundaries and more uniform color.

Differences in lesion images A source of inaccuracy of our segmentation doesn't depend on the algorithm. We observed that images have an "offset" distortion that modifies the lesion. This distortion could be due to light variations and to the pressure of the dermatoscope that bends the skin. We show a pair of calibration images taken on the same day, with the same dermatoscope, and the same zoom in Figure .

Results We analyze the entire dataset and we compare the obtained area variations with the calibration error. If the variations are contained in the calibration error bounds, we can at least assume that the normal area variation of the lesions is not higher than the error. On the other hand, if the variations exceed the error bounds we can determine the actual area varia-

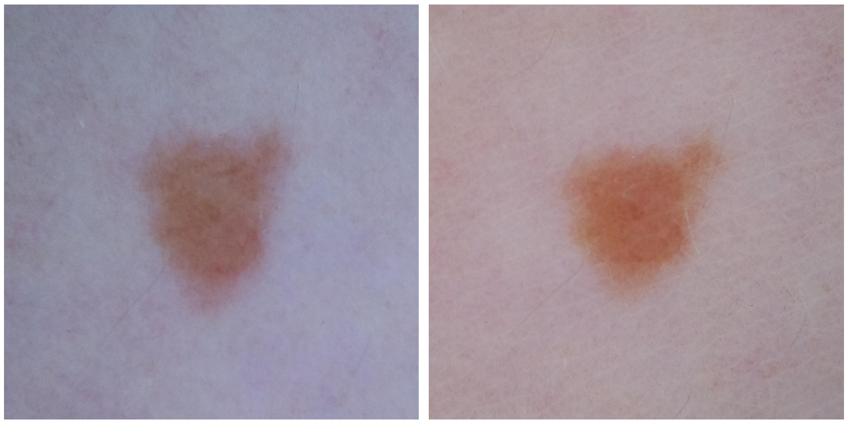
(a) (b)

Figure 4.4: Lesion in (a) is taller and slightly differ from (b) in shape

tion bounds. A complete analysis of the dataset is beyond the scope of this thesis.

Chapter 5

Related work

5.1 Cropping program

The cropping program we developed allows us to virtually crop the image without modifying it. The program provides a graphical interface that allows us to set the elliptical crop and to remove other objects such as plasters, tattoos, or marker signs with a ”free-hand” drawing. Instead of modifying the image, the program creates a binary mask by calculating which pixels are inside the cropped area. The output is given by the binary mask stored as an image and an XML file that contains the cropped intervals for each row.

Graphical interface The interface allows us to navigate the dataset images. We can navigate through patients, lesions, and images with the related drop-down menu. The crop is always displayed and can be moved and resized by dragging it or dragging its four lateral points. The default size of the crop is equal for every image of the same lesion. All pixels outside the crop will not be computed in the segmentation. In fact, we observed that the segmented area slightly changes varying the crop size. Additionally, it is possible to manually remove certain areas by clicking the ”Draw shape” button and drawing the shape on the image panel. Every click draws a vertex, and to close the shape we must click on the initial one. All the area inside the defined shape will not be computed in the segmentation. Figure 5.1 shows the program interface.

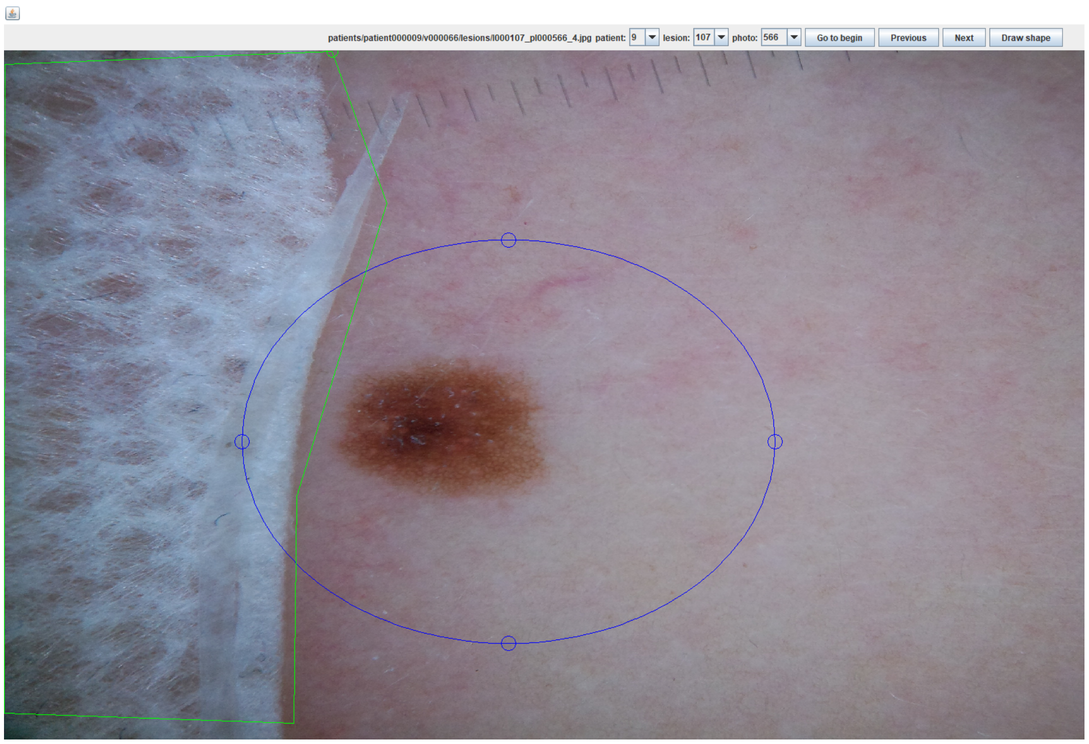

Figure 5.1: Crop program interface. The blue ellipse is the crop, the green lines define the shape.

Program output When we crop an image, an element with the crop information is added to the output XML file. An example of the output file structure is the following:

```
<IMAGE_DATA IMAGE_HEIGHT="2448" IMAGE_WIDTH="3264" IMAGE_ID="567"
XSCALE="1.7" YSCALE="2.5367875647668394">
   <CROP BOTTOM_RIGHT_X="1276" BOTTOM_RIGHT_Y="583"
   TOP_LEFT_X="536"TOP_LEFT_Y="23"/>
   <SHAPE>
      <POINT POINT_X="1182" POINT_Y="70"/>
      <POINT POINT_X="1142" POINT_Y="568"/>
      <POINT POINT_X="1281" POINT_Y="534"/>
      <POINT POINT_X="1366" POINT_Y="232"/>
   </SHAPE>
</IMAGE_DATA>
```

For each image we save its size, its ID, the scale factors and the crop information. A crop is composed by the elliptical crop (that we simply call ”crop”) and the shape crop (that we call ”shape”). The elliptical crop is represented by the two points that define the rect containing it. The shape element contains a list of point elements described by their coordinates. We don’t store in the XML file the valid intervals for each row since they can be obtained by this information and because these are the information that the Java swing library requires to draw the shapes.

Computing valid intervals Our implementation of the algorithm requires that for each row we define the intervals of valid/invalid pixels. Since we store only some information of the elliptical crop and the shape crop, we need to compute every boundary pixel of these crops. Starting from the two points $\boldsymbol{p_1}$ and $\boldsymbol{p_2}$ of the rect surrounding the ellipse, we obtain the center $\boldsymbol{c} = (c_x, c_y)$ as $\boldsymbol{c} = \frac{1}{2}(\boldsymbol{p_1} + \boldsymbol{p_2})$ and width and height as $w = |\boldsymbol{p_2}_{\mathrm{x}} - \boldsymbol{p_1}_{\mathrm{x}}| \quad h = |\boldsymbol{p_2}_{\mathrm{y}} - \boldsymbol{p_1}_{\mathrm{y}}|$.
Knowing that the equation of an ellipse centred in (c_x, c_y) is:

$$\frac{(x - c_x)^2}{a^2} + \frac{(y - c_y)^2}{b^2} = 1 \tag{5.1}$$

where $a = w/2$ and $b = h/2$, we can now obtain every pixel by iterating x from $c_x - a$ to $c_x + a$, obtaining the correspondent y values as

$$y = c_y \pm \sqrt{b^2 - \frac{(x - c_x)^2}{a^2}} \tag{5.2}$$

Then, for every value of y we create the correspondent interval $[x_1, x_2]$. Every pixel of the row belonging to that interval will be computed in the segmentation, whereas every pixel outside of it will be neglected.

To obtain the shape crop boundary pixels between two of its vertexes $\boldsymbol{V_1}$ and $\boldsymbol{V_2}$ we compute the versor $\boldsymbol{u}$ of the line connecting $\boldsymbol{V_1}$ and $\boldsymbol{V_2}$ as

$$\boldsymbol{u} = \frac{\boldsymbol{V_1} - \boldsymbol{V_2}}{||\boldsymbol{V_1} - \boldsymbol{V_2}||} \tag{5.3}$$

We can then obtain every pixel $\boldsymbol{p}$ between $\boldsymbol{V_1}$ and $\boldsymbol{V_2}$ as

$$\boldsymbol{p_i} = \boldsymbol{V_2} + i\boldsymbol{u} \quad for \quad i = 0...||\boldsymbol{V_1} - \boldsymbol{V_2}|| \tag{5.4}$$

We can obtain every boundary pixel of the shape by applying the formula above for each connected pair of vertexes. We obtain the resulting invalid intervals by taking pairs of boundary pixels in each row starting from the left.

Finally, for each row we compute the valid intervals by subtracting the shape crop intervals to those of the elliptical crop. As result an image row can have multiple valid intervals if the shape is contained in the ellipse.

All the values computed in this phase are then scaled to the original size of the image, since the point coordinates are given relatively to the window size. We use the h_{ratio} and w_{ratio} as scale factors, where

$$h_{\text{ratio}} = \frac{h_I}{h_W} \quad w_{\text{ratio}} = \frac{w_I}{w_W}$$

in which (w_I, h_I) is the size of the image and (w_W, h_W) is the size of the program window.

5.2 Unifying the two databases

To facilitate future research on the data, we unify the two databases and their related image sets. We create a single database simplified for the purpose of the research and we unify the image paths.

Structure We have only the following three tables: patient, lesion, and photolesion. We don't consider the visits since we have the date for every photolesion record. The portrait information is moved into the photolesion table.

Unifying records We create a single database file and we copy all the records from the first database with a short Java program. Then, we insert records from the second database by offsetting their IDs of the maximum ID in the first database. Every time we modify an ID we eventually modify the foreign key in the related records. Then, we write the images. We create the directory tree patient[number]/lesion[number]. Then, for each record in the photolesion table we copy the image from the original path to the right lesion folder just created.

Bibliography

[1] Francesco Peruch, *(SEMI)-AUTOMATED ANALYSIS OF MELANOCYTIC LESIONS*, Università degli Studi di Padova, 2015

[2] Darel Rex Finley, *http://alienryderflex.com/polygon_area*, 2006

[3] Bart Braden, *The Surveyor's Area Formula*, The College Mathematics Journal, 1986