University of Liège
Faculty of Engineering

Master Thesis

A workflow for large-scale computer-aided cytology and its applications.

Author : Romain Mormont

Supervisor : Dr. Raphaël Marée

Academic : Prof. Pierre Geurts

Master thesis submitted for the degree of
MSc in Computer Science and Engineering

Academic year 2015-2016

Contents

Summary

Acknowledgement

Chapter 1

Introduction

Chapter 2

Object detection in large images

2.1 General problem

2.1.1 Formulation

Generic formulation of the object detection problem

2.1.2 Implementation issues

What issues an implementor could face when trying to implement object detection in large images

2.1.3 Related works

What solutions are usually presented in the litterature to solve those problems (shallow overview as this is a wide topic)

2.2 Cytology

According to the Collins dictionary, cytology is "*the study of plant and animal cells, including their structure, function and formation*" [Dic16].

2.2.1 Thyroid cytology and nodule malignancy

Nodules are growths that can develop in the thyroid. Usually, they are benign but in some cases they can be a sign of a cancer (?? give prevalence, probabilities ??). Therefore, patients presenting those nodules are subjected to a range of medical examinations. One of the most important step in those examination is the fine needle aspiration biopsy (FNAB) [BLF10]. It consists in taking a sample of tissues directly inside the nodule mass. Those takings are then stained and examined under a microscope by a physician. Especially, the nodule malignity is confirmed by the presence of some specific features such as intra-nuclear inclusions or proliferative architectural patterns. Example of stained takings are shown in Figures 2.1 and 2.2. In the former are shown cells with inclusion which are recognizable because of the

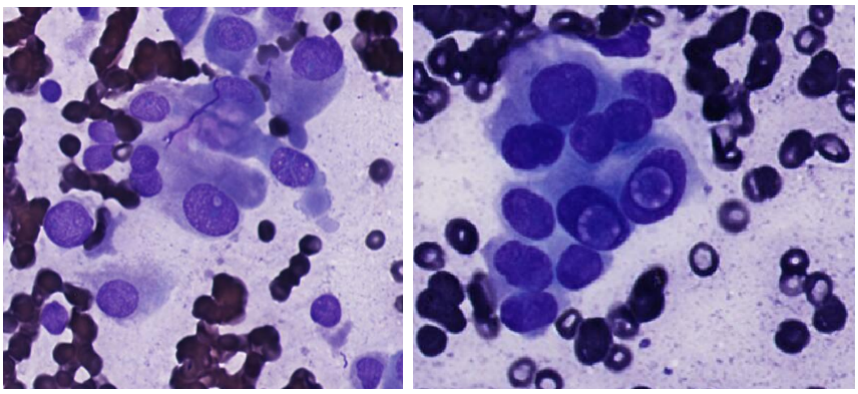


Figure 2.1: Cells with inclusion

typical brighter circular area inside the cell. In the latter are shown architectural patterns. Especially, proliferative patterns are shown in Figure 2.2(a) while non-proliferative ones are shown in 2.2(b).

2.2.1.1 Cytomine

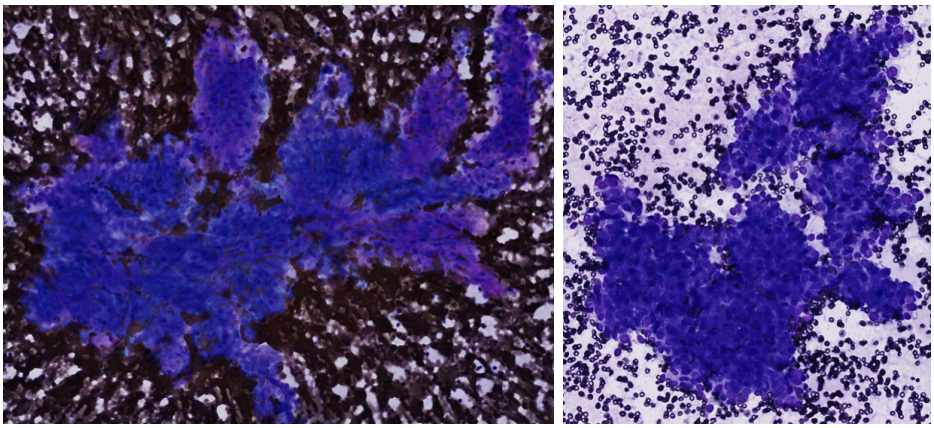
Cytomine [Mar+16] is a web-based environment enabling collaborative multi-gigapixel image analysis. Users can navigate through those images, annotate them and associate domain specific labels to the generated annotations. The platform also integrates machine learning-based image recognition algorithms that can automatically produce annotations. A reviewing system enables experts to proofread those annotations.

2.2.1.2 Dataset

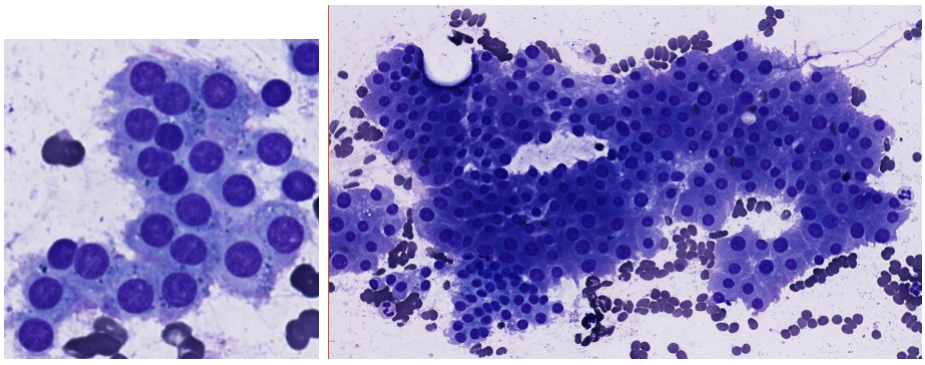
A project dedicated to nodule malignancy detection was created on the Cytomine platform. It contains 61 annotated images with sizes ranging from 4 gigapixels to 18 gigapixels. Those images contain a total of 5784 labelled annotations performed by experts from the ULB (?? ref ??). Those labels (or terms) link the annotation to cytological objects related to the nodule malignancy problem. The terms made available on Cytomine are organized in an ontology which is divided into three main subcategories:

- **Architectural patterns** : includes proliferative and non-proliferative patterns but also an intermediate class for patterns which present minor signs of proliferation.
- **Nuclear features** : includes cells with inclusion, normal cells and some additional cell-related terms
- **Others** : includes artefacts, background but also poly nuclear cells, red blood cells,...

The complete ontology can be found in Appendix B. Among those available terms, the ones that matter the most in the context of nodule malignancy detection

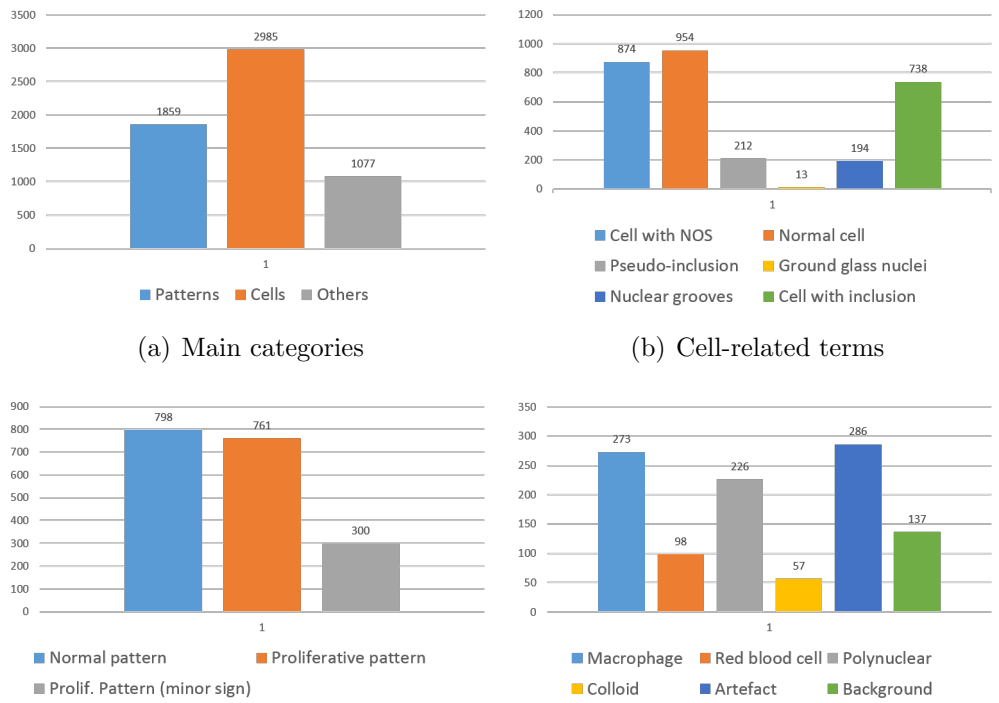


(a) Proliferative



(b) Non-proliferative

Figure 2.2: Stained thyroid takings - architectural patterns



(a) Main categories

(b) Cell-related terms

(c) Pattern-related terms

(d) Other terms

Figure 2.3: Terms of the thyroid ontology - annotation counts histograms

are the cells with inclusion and the proliferative architectural patterns (major or with minor sign). Those terms represent the positive classes of the problem. The distributions of terms are given in Figure 2.3. Those histograms highlight that a significant number of annotations have been made with the terms of interest. Moreover, the number of annotations is relatively balanced for those terms. While they are balanced taken alone, a problem arises when the terms are grouped. For instance, in the binary problem of determining whether a cell contains an inclusion or not, the positive class might be the term *Cell with inclusion* while the negative class might contain the all the other terms from the cell subcategory (and possibly terms from the others subcategory). While being sound, this distribution of terms in both classes results in an imbalance which can be problematic when applying machine learning classification algorithms. The classes distribution for this example problem are given in Figure 2.4.

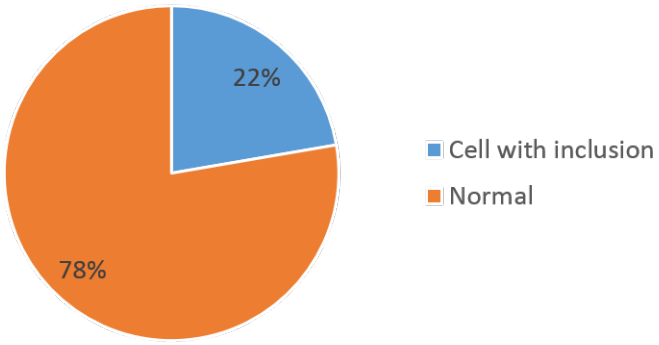


Figure 2.4: Cell inclusion detection problem. Terms in positive class: cell with inclusion. Terms in negative class: normal cells, pseudo inclusion, ground glass nuclei, nuclear grooves, red blood cells and poly-nuclear.

Chapter 3

A generic workflow : Segment Locate Dispatch Classify

In this chapter, a generic workflow for solving problems of objects detection and classification in images is presented. This workflow was first imagined by ?? JM Begon in 2015 ?? as a generalization of the work on thyroid nodule malignancy detection made by Antoine Deblire in [Deb13]. In the context of his master thesis, he had implemented a processing workflow for detecting cells with inclusion and proliferative architectural patterns (see ?? (thyroid)) in digitized thyroid punctions slides. The cells and architectural patterns were detected by segmenting the images and then classified using machine learning. As explained in the Section ?? (thyroid), some patterns could themselves contain cells with inclusion. Therefore, the author implemented a second processing workflow to detect those cells which also relied on a segmentation algorithm to isolate cells in patterns and then used machine learning to assess their malignity. From those workflows, a common pattern emerged: performing detection using a segmentation algorithm and then classifying the detected objects using machine learning.

In 2015, ?? JM Begon ?? developed a first version of a generic workflow based on this pattern and gave it the name *Segment-Locate-Dispatch-Classify* (SLDC). Unfortunately, this implementation suffered from some drawbacks which made it hard to reuse in other contexts. The workflow was therefore re-worked in the context of this master thesis.

In Section 3.1, the workflow is introduced and formalized. Especially, the various steps are detailed and then combined into an algorithm which is gradually improved to reach an acceptable level of genericity. In Section 3.2, the actual implementation of the workflow, so-called *framework*, made in the context of the master thesis is presented (?? lien GitHub ??). First are explained the reasons why the previous implementation was replaced by a new one. The new framework is then presented starting with its requirements as well as a justification for the choice of Python as the implementation language. The software architecture is then broken down and the purpose of each package and important class are explained. Section 3.2.5 presents the developments made for testing the various components of the framework. Finally, the last section illustrates the usage of the framework for solving a toy problem.

3.1 Principle

3.1.1 Algorithm

The workflow is a meta-algorithm[1] that detects and classifies objects contained in images. Particularly, given as input a two-dimensional[2] image $\mathcal{I}$ from the set of all possible images I, it is expected to output the information about the objects of interest contained in this image. Those information include the shape of the object, its location as well as a classification label. Formally, the workflow can be seen as an operator $\mathcal{W}$:

Definition 1. *Let $\mathcal{W}$ be an operator such that*

$$\mathcal{W}(\cdot) : I \to R^N \mid \mathcal{I} \mapsto \{(o_1, C_1), ..., (o_N, C_N)\}, N \in \mathbb{N}_0 \tag{3.1}$$

where N is the number of objects of interest in $\mathcal{I}$ and (o_i, C_i) is a result tuple belonging to the set R of all possible results tuples. The first element of this tuple, o_i, is a representation of the information (shape and location) about the i^{th} object of interest found in $\mathcal{I}$ and the second, C_i, its classification label.

It is worth noting that genericity is of the essence. That is, the meta-algorithm should be able to solve the widest possible range of object detection and classification problems. Moreover, as explained in Section 3, it should produce those outputs using image segmentation and machine learning. As far as the segmentation is concerned, genericity is usually hard to obtain because of the high variability of images across different problems. In order to ensure that the workflow remains generic enough, a particular segmentation procedure is not imposed to the implementer who is expected to provide one that suits the problem. The same goes for the classification models used for predicting the labels of the objects.

In the subsequent sections, some additional operators are defined and used to build the $\mathcal{W}$ operator. First, a basic version of the algorithm is presented and then refined in order to achieve an acceptable level of genericity.

3.1.2 Additional operators

Segmentation is the first operation applied to the image. This step of the algorithm is where the detection is actually carried out:

Definition 2. *Let $\mathcal{S}$ be the **segment** operator:*

$$\mathcal{S}(\cdot) : I \to B \mid \mathcal{I} \mapsto \mathcal{B} \tag{3.2}$$

It is applied to an image $\mathcal{I} \in I$ and produces a binary mask $\mathcal{B} \in B$ with B being the set of all possible binary masks. The pixel b_{ij} of $\mathcal{B}$ is 1 if the pixel p_{ij} of $\mathcal{I}$ is located in an object of interest, otherwise it is 0.

[1] In this context, a meta-algorithm is an algorithm that coordinates the execution of other algorithms.

[2] A third dimension can be dedicated to the images channel (i.e. 3 channels for RGB images, 4 channels for RGBA images).

While the segmented image theoretically contains the necessary information about the detected objects (i.e. shape and position in the image), the format of this information is inconvenient to query mostly because it is embedded into the binary mask and a single object cannot be trivially extracted. An intermediate step that would convert this information into a more convenient format is therefore needed. This format should encode both the shape of the object and its position in the image. It appears that polygons match this specification.

Definition 3. *Let $\mathcal{L}$ be the* ***location*** *operator. It is applied to a binary mask and produces a set of polygons encoding the shapes and positions of every object in the image. Formally:*

$$\mathcal{L}(\cdot) : B \to H^N \mid \mathcal{B} \mapsto \{P_1, ..., P_N\}, N \in \mathbb{N}_0 \tag{3.3}$$

where $\mathcal{B}$ is a binary mask as defined in Definition 2, N is the number of objects of interest in $\mathcal{B}$ and P_i is the polygon representing the geometrical contour of the i^{th} object in $\mathcal{B}$. This polygon belongs to the set H of all possible polygons.

The final step of the workflow is the object classification and is performed by a classifier which is passed a representation of the object (e.g. image, geometrical information,...) and produces a classification label. In this theory, there is no restriction about the nature or representation of the objects processed by the classifiers.

Definition 4. *Let $\mathcal{T}$ be the* ***classifier*** *operator. It is applied to an object of interest and produces a classification label. Formally:*

$$\mathcal{T}(\cdot) : O \to L \mid o \mapsto C \tag{3.4}$$

where O is the set of all possible objects ($o \in O$) and L, the set of all possible classification labels ($C \in L$).

Definition 5. *Let $\mathcal{T}^*$ be an extension of $\mathcal{T}$ which is given a set of objects and produces labels for all of them. Formally:*

$$\mathcal{T}^*(\cdot) : O^N \to L^N \mid \{o_1, ..., o_N\} \mapsto \{\mathcal{T}(o_1), ...\mathcal{T}(o_N)\}, N \in \mathbb{N}_0 \tag{3.5}$$

3.1.3 Single segmentation, single classifier

The most simple construction of $\mathcal{W}$ would be the composition of the operators defined in Section 3.1.2. Particularly, the compositions $\mathcal{S} \circ \mathcal{L}$ and $\mathcal{S} \circ \mathcal{L} \circ \mathcal{T}^*$ would respectively produce the polygons representing the objects and their labels. This construction is summarized in Algorithm 1:

Algorithm 1. *Construction of $\mathcal{W}$ using one segmentation and one classifier:*

1. *Return* $\langle (\mathcal{S} \circ \mathcal{L})(\mathcal{I}), (\mathcal{S} \circ \mathcal{L} \circ \mathcal{T}^*)(\mathcal{I}) \rangle$

As explained in Section 3.1.1, the definition of $\mathcal{S}$ and $\mathcal{T}^*$ would be left at the implementer's hands. As far as the $\mathcal{L}$ operator is concerned, it could be imposed by the workflow without loss of genericity provided that the binary mask format

is defined. Such a construction of $\mathcal{W}$ could already solve any object detection and classification problem on image in which the labels can be predicted by a single classifier. However, in some cases, one classifier is not enough. This happen, for instance, when the image contains objects of very different nature and using several classifiers would yield better results than using a single one. An extension is therefore needed.

3.1.4 Single segmentation, several classifiers

In this attempt to construct a generic $\mathcal{W}$ operator, the image is assumed to contain M distinct types of objects and the workflow uses M classifiers (the i^{th} classifier being noted $\mathcal{T}_i$ with $i \in \{1, ..., M\}$) to classify those objects. As an object should only be processed by one classifier, the workflow has to be added a new step which consists in dispatching each polygon to its most appropriate classifier.

Definition 6. *Let $\mathcal{D}$ be the dispatch operator. It is applied to a polygon and produces an integer which identifies the most appropriate classifier for processing this polygon:*

$$\mathcal{D}(\cdot) : H \to \{1, ..., M\} \tag{3.6}$$

This step being problem dependent, it is the responsibility of the implementer to define the rules used for dispatching the polygons. However, the format of these rules can be defined.

Definition 7. *Let $\mathcal{P}$ be a set of M predicates $p_1, ..., p_M$ which associate truth values to polygons:*

$$p_i(\cdot) : H \to \{true, false\} \mid P \mapsto t, i \in \{1, ..., M\} \tag{3.7}$$

where p_i is the predicate associated with the i^{th} classifier. The polygon P is dispatched to a classifier $\mathcal{T}_i$ if p_i associates true to this polygon. To avoid dispatching an object to several classifiers, the predicates should verify the following property:

$$p_i = true \Leftrightarrow p_j = false, \forall j \neq i \tag{3.8}$$

Given this format, the $\mathcal{D}$ operator can be trivially constructed as it returns i if p_i is *true*. The algorithm resulting from this construction of $\mathcal{W}$ starts the same way as in Section 3.1.3: the image is applied the segment and locate operators. Then, the resulting polygons are dispatched and classified to produce the labels. The resulting algorithm is summarized in Algorithm 2. Figure 3.1 illustrates Algorithm 2 with a workflow that has two classifiers. The first is designed to classify small objects while the second classifies bigger ones.

Even though the workflow can now handle several types of objects, there are still some particular problems that cannot be solved with Algorithm 2. In particular, this algorithm works perfectly as long as objects are not included in one another. In this case, the workflow will consider their intersection as a single object and therefore won't be able distinguish them.

Before extending the algorithm for handling this case, it is worth noting that Algorithm 2 is completely compatible with Algorithm 1. Indeed, if there is only one classifier (i.e. $M = 1$) and the predicate p_1 always returns *true*, then both algorithms are exactly the same.

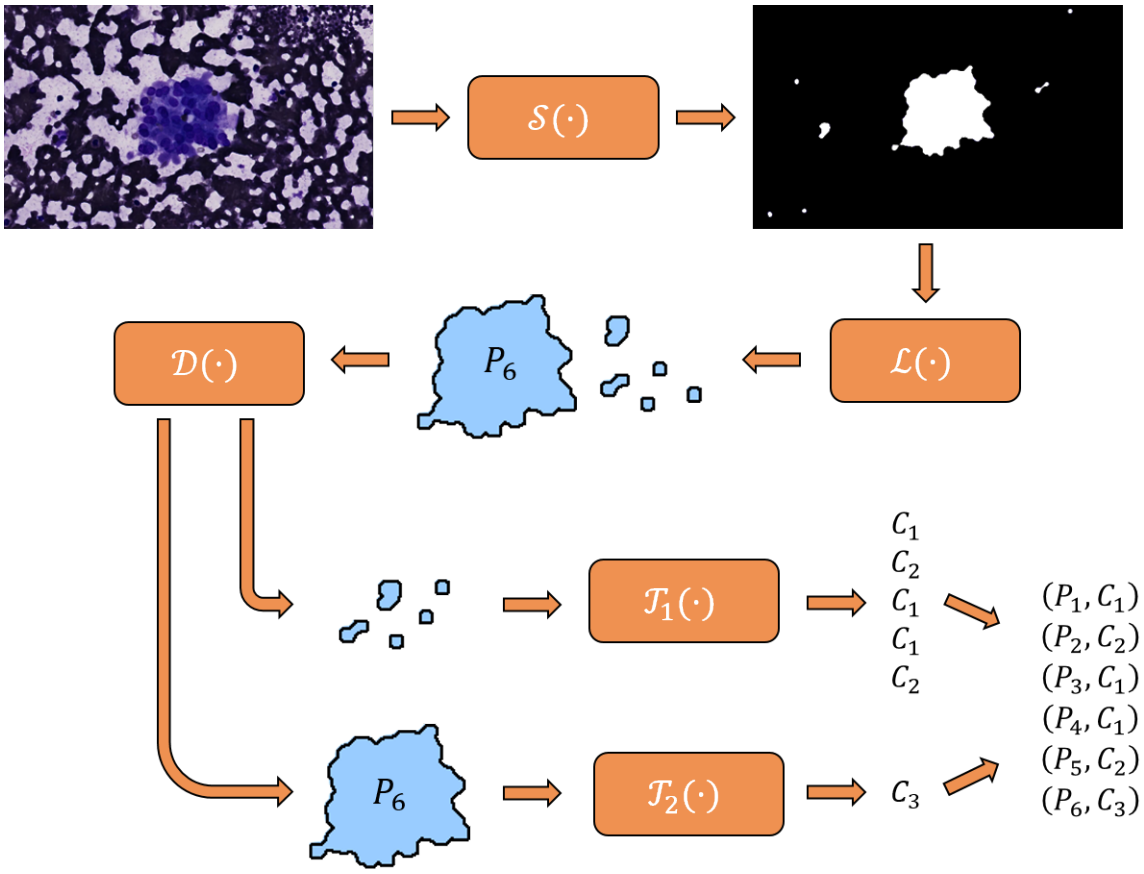


Figure 3.1: Illustration of Algorithm 2.

Algorithm 2. *Construction of the $\mathcal{W}$ operator with a single segmentation and several classifiers.*

1. *Apply the $\mathcal{S} \circ \mathcal{L}$ composition to the input image $\mathcal{I}$ to extract the objects of interest as the set of polygons $S_p \leftarrow \{P_1, ..., P_N\}$*
2. *Initialize the labels set $L \leftarrow \emptyset$*
3. *For each polygon $P \in S_p$:*
   (a) *Compute the classification label $C \leftarrow \mathcal{T}_{\mathcal{D}(P)}(P)$*
   (b) *Place the label in the labels set $L \leftarrow L \cup \{C\}$*
4. *Build and return objects and labels set $S_p \times L$.*

3.1.5 Chaining workflows

To handle the case when some objects are included in others, a solution consists in executing several instances of Algorithm 2 one after another.

Definition 8. *Let $\mathcal{W}_1, ..., \mathcal{W}_K$ be a set of K instances of Algorithm 2. Each algorithm $\mathcal{W}_i$ has its own segmentation procedure $\mathcal{S}_i$ and proper sets of dispatching predicates $\mathcal{P}_i$ and classifiers $S_{\mathcal{T},i}$.*

While $\mathcal{W}_1$ would be applied to the full image $\mathcal{I}$ to extract all the objects of interest, $\mathcal{W}_2, ..., \mathcal{W}_K$ would only be passed image windows containing the previously detected objects. Given those windows, they would have to detect the objects of interest included in the objects found by $\mathcal{W}_1$.

Definition 9. *Let $\mathcal{I}_P \in I$ be an image window extracted from image $\mathcal{I}$ and containing the object represented by polygon P. The window is the minimum bounding box containing this polygon.*

A further refinement would be to provide a way for the implementer to filter the polygons of which the windows are passed to a given workflow instance. Indeed, a given instance $\mathcal{W}_i$ might be designed to process only a certain category of objects and therefore should not be passed windows of objects that doesn't fall in this category.

Definition 10. *Let $\mathcal{F}$ be the **filter** operator. It is given a set of polygons S_P and returns a subset S'_P of polygons:*

$$\mathcal{F}(\cdot) : H^N \rightarrow H^M, N, M \in \mathbb{N}, M \geq N \tag{3.9}$$

Each instance of the workflow $\mathcal{W}_i$ except $\mathcal{W}_1$ is therefore associated a filter operator $\mathcal{F}_i$. The resulting algorithm is given in Algorithm 3 and has now reached an acceptable level of genericity. The algorithm is illustrated in Figure 3.2.

Algorithm 3. *Construction of the $\mathcal{W}$ operator with K instances of Algorithm 2:*

1. *Execute the first workflow and save the results in the result set R: $R \leftarrow \mathcal{W}_1(\mathcal{I})$*

2. *Create the polygons set and initializes it with the polygons found from the execution of* $\mathcal{W}_1$*:* $S_P \leftarrow \{P_{1,1}, ..., P_{1,N}\}$

3. *For each* $i \in \{2, ..., K\}$*:*

   (a) *Extract polygons to be processed by* $\mathcal{W}_i$*:* $S'_P \leftarrow \mathcal{F}_i(S_P)$

   (b) *For each polygon* $P \in S'_P$*:*

      i. *Execute workflow* $\mathcal{W}_i$ *on the image window and saves the results:* $R \leftarrow R \cup \mathcal{W}_i(\mathcal{I}_P)$

      ii. *Add the extracted polygons to the polygons set:* $S_P \leftarrow S_P \cup \{P_{i,1}, ..., P_{i,M_i}\}$

4. *Return the results set* R

3.2 Implementation

This section aims at presenting the implementation of the workflow formalized in Section 3.1. In Section 3.2.1, the reasons why the previous implementation was replaced by a new one are presented. Then, the requirements, design choices and architecture of the new framework[3] are given in Sections 3.2.2, 3.2.3 and 3.2.4. Finally, how to apply the framework is illustrated with a toy example in Section 3.2.6. The framework discussed in this section is available on GitHub ?? at this url ??.

3.2.1 Initial implementation

As explained in this chapter's introduction, a first version of the workflow was implemented in 2015. However, in the context of this thesis, the decision was made to re-implement it for various reasons.

A major issue was the presence of a software component called a *datastore* which had to be defined by the implementer for each distinct application of the workflow. In addition to be a dependency of almost every other class of the framework, it actually forced the implementer to define workflow execution and chaining logic himself although this logic is obviously not problem dependent and could be encapsulated. The major consequence of this design was an increased workload for the implementer to apply the framework to a custom object detection and classification problem. Moreover, the datastore being tightly coupled with other classes, it made writing automated tests quite difficult. Reproducing bugs was even harder because the replication implied to restore the datastore state which was not trivial.

Another issue with the previous implementation was its too high level of genericity. Most of the components of the framework were defined as abstract classes and interfaces to be derived or implemented by the implementer. This made the framework hardly understandable and difficult to apply as he had to define more than just problem dependent components. In some cases, some implementations were

[3]In this section, the term *workflow* will refer to the algorithm while *framework* will refer to the implementation.

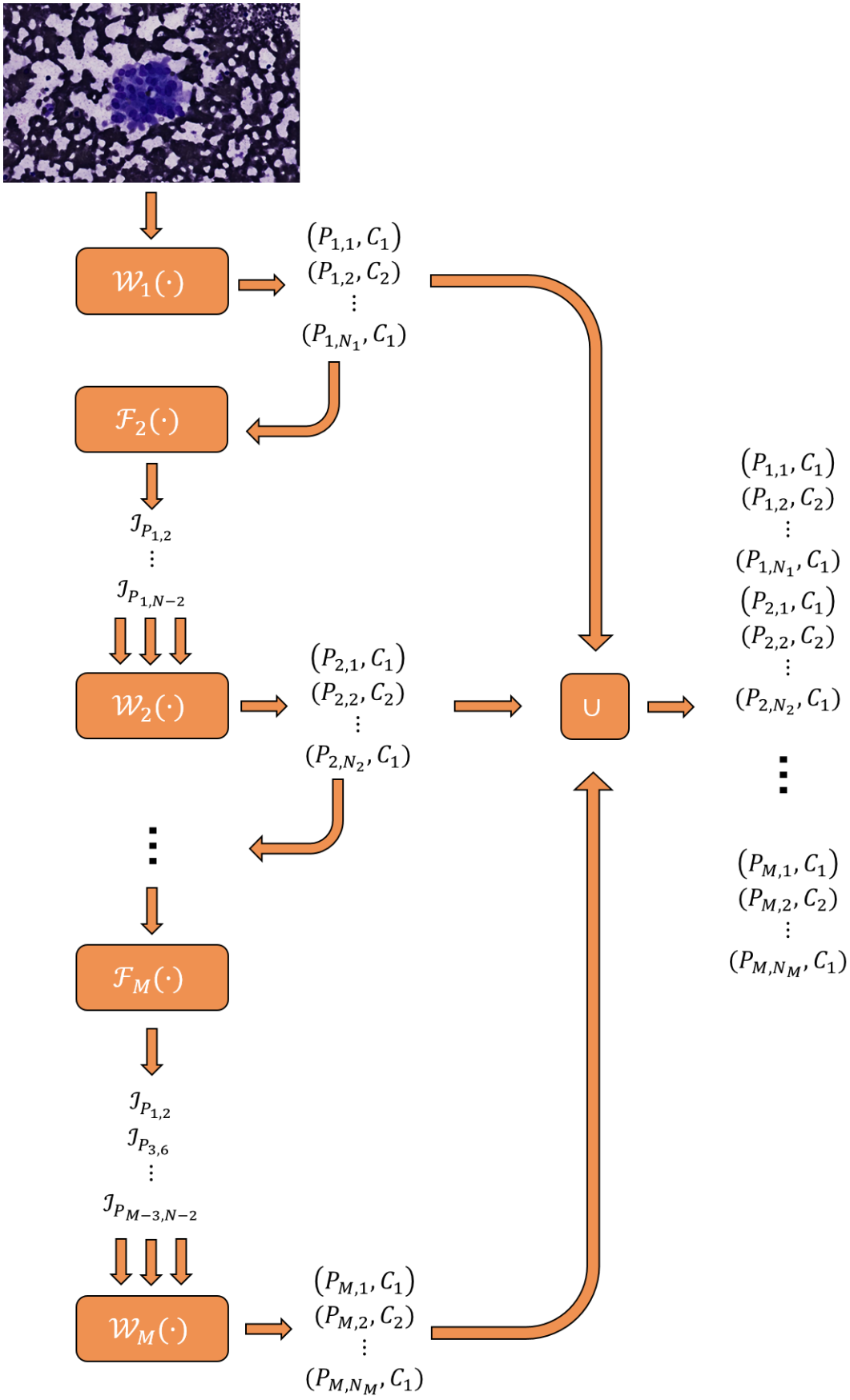


Figure 3.2: Illustration of Algorithm 3

provided but they only increased the complexity of the framework. Indeed, it was not clear whether those classes could be used directly or whether the implementer should provide his own classes.

Another final critical point was the lack of robustness. Especially, when applied to the thyroid case where images were fetched using HTTP requests, any network error would exit the program, leading to the loss of all collected data.

All in all, it was decided to re-implement the framework to get rid of the flawed parts of the design while keeping the good parts. The philosophy behind the new framework is illustrated through a set of requirements in Section 3.2.2.

3.2.2 Requirements

The main requirements for the framework are listed hereafter.

Genericity As for the algorithm, the framework should be able to solve the widest possible range of object detection and classification problems in any context. This property has more implication in the case of the framework design than for the algorithm design, especially when it comes to fixing the representation of the various involved data types (i.e. image, polygon,...).

Efficiency While the framework has no control over the efficiency of the algorithms defined by the implementer (i.e. segmentation or classification procedures), the coordination of those algorithms should not induce a significant overhead in the overall execution.

Large images While large images handling was irrelevant at the algorithm design stage, it becomes critical at this point. To remain generic, the framework should not make any assumption about the size of the images to be processed. Especially, a whole image should not be assumed to fit into memory.

Robustness The framework should be robust to errors. That is, a single error should not interrupt the whole execution. For instance, if the framework executes a set of independent computations and one of them fails, it should only be stopped if this failure is unrecoverable and affects all the other computations. Otherwise, the failure should be reported and those others computations should execute until completion.

Transparency The framework should provide a built-in way to communicate its progress, the duration of each step as well as the errors it encounters with the user. The level of verbosity of this communication tool should be adjustable. Moreover, all the relevant information generated by the framework should be made available to the implementer in a structured and convenient way.

Parallelism Whenever possible the framework should take advantage of parallelism to reduce its execution time but the implementer should be given a way to switch to sequential execution. Moreover, the implementer should be able to adjust the level of parallelism (i.e. the number of available processors).

Ease of use The work of the implementer should be kept as minimal as possible. He should only have to define the logic of the workflow components that are problem dependent : image format, segmentation, dispatching rules, classifiers,...

3.2.3 Language

The first choice occurring in the development of an existing algorithm is obviously the language in which it will be implemented. As far as the workflow is concerned, the chosen language was Python. Indeed, this language provides a simple, accessible and complete environment for solving the kind of problems addressed by the framework and would therefore contribute to the overall ease of use the framework.

First of all, the language has many features which allows developers to quickly come up with solutions to problems. Especially, it is strongly and dynamically typed, multi-paradigm (imperative, functional, object oriented,...), interactive (it can be used in an interactive console), interpreted and garbage-collected. It also supports usual data structures such as lists, arrays, dictionaries and sets natively and provides operations for manipulating them in a concise way.

In addition to its built-in features, Python has become a great language for scientific computing as it has been augmented with excellent open source libraries over the years. First, the SciPy ecosystem which includes the SciPy [Oli07] and NumPy [VCV11] libraries. The first is a collection of numerical algorithms and domain-specific toolboxes (signal processing, optimization, statistics,...). The second is a fundamental package for numerical computations which provides an efficient representation of multi-dimensional arrays and operations on them. Built on top of the SciPy ecosystem comes Scikit-Learn [Ped+11], a library that provides simple, efficient and reusable tools for data mining and machine learning. Image processing is not outdone with a Python binding for the huge OpenCV library [Bra00]. Two alternatives are scikit-image [Wal+14] which is built on top of the SciPy ecosystem or the Pillow library [Cla16]. All of them provide a collection of well-known image processing algorithms. Another useful library is Shapely [Gil13] which provides a representation for geometrical objects (e.g. polygons) and operations on them.

Python was also chosen because the workflow was implemented to be integrated with Cytomine (see Section ??). Particularly, the final goal was the detection and classification of objects in images stored on Cytomine servers. As those images and their metadata are exposed through an API interfaced by a Python client, it was essential that the workflow could use this client to communicate with the back-end. As the Cytomine client was implemented in the version 2.7.11 of Python, this version was also used for developing the framework.

3.2.4 Software architecture

The framework was organized as a Python library of which the root package was called `sldc`.

3.2.4.1 Image representation

The image representation design is a critical point of the framework architecture. Indeed, on the one hand, it should be abstract enough so that implementers can apply the workflow on images in any format. On the other hand, it should provide access to a concrete representation available to the framework because some steps need to access this representation to extract some information. For instance, location is one such step as it processes a binary mask to extract polygons.

The representation should also provide a way of extracting sub-windows from an image. The need for this feature is twofold. First, it is needed by the workflow (see Definition 9). Then, it could be used to address the large images handling requirement and to overcome the fact that a whole image is not assumed to fit into memory. The idea is to split the image into smaller chunks called tiles which could be loaded into memory and processed one after another. Especially, the tiles would be applied the first part of the workflow, that is segmentation and location. As the polygons of each tile are extracted independently, it might occur that a single object of interest which spreads over several tiles ends up being splitted into several polygons. To make sure there is a one to one relationship between a polygon and an object of interest, an additional step must be added to the workflow before the dispatching and would consist in merging the polygons representing a same object. This step is detailed in Section 3.2.4.4.

The abstract image representation and related classes were implemented into the `sldc.image` package presented in the UML diagram shown in Figure 3.3.

The `Image` class is the abstract image representation mentioned above. It provides three abstract methods for checking image dimensions (width, height and number of channels) and a fourth one, `np_image`, which should implement the conversion between the implementer's custom image format and the concrete format mentioned above. NumPy multi-dimensional arrays were chosen to be this concrete representation. In addition to the inherent advantage of using the NumPy library, this choice was also motivated by the fact that those arrays are compatible with the various image processing libraries presented in Section 3.2.3.

An image window is materialized by the `ImageWindow` class of which the design is based on the decorator pattern. It stores information about the position and size of the window as well as a reference to the parent image. Especially, location and size are respectively represented by coordinates of the first top left pixel included in the window (coordinates are referenced to the top left corner of the parent image) and by the window width and height. As an image window instance provides a level of indirection on top of another image, some methods are provided to fetch this base image as well as the absolute offset[4].

[4]The absolute offset is the offset of the window referenced to the base image's top left pixel. It is different from the image window offset if its parent image is also an image window.

A tile is also represented by a class named `Tile` which extends `ImageWindow` and augment it with an integer identifier field. As tiles can potentially be derived, a `TileBuilder` interface was developed. As suggested by the name, a class implementing this interface is responsible for building specific tile objects. This structure is actually an application of the factory method pattern which has the advantage of allowing the framework to build specific tiles objects defined by the implementer while remaining unaware of the construction logic of those objects. The implementer that would derive the `Tile` class to implement a custom loading procedure in `np_image` is advised in the documentation to raise an `TileExtractionError` exception if the loading fails. This allows the rest of the framework to handle loading failure and therefore increase its robustness.

Finally, to make it easier to iterate over the tiles of an image two classes were developed : `TileTopology` and `TileTopologyIterator`. The first is responsible for dividing an image into a set of overlapping tiles. The overlap allows the merging procedure to be simpler as polygons corresponding to a same object will have a geometrical intersection.

The tile topology is fully defined with three parameters: the tile maximum width, w_m, and height, h_m, and the number of pixels that overlap, o_p. The tile topology object also associates unique increasing identifiers to the tiles. An example topology with its resulting tiles and identifiers is shown in Figure 3.4. As soon as the `TileTopology` object is built, it can be queried using those identifiers for building tile objects or for fetching topology information such as one tile's neighbours identifiers. While this organization goes off from the object oriented philosophy a bit, it allows all operations provided by the tile topology object to be $\mathcal{O}(1)$ (see Appendix A). It goes without saying that the overlap parameter should be set carefully because it induces some additional computations. Indeed, some parts of the image will be segmented several times as they are present on more than one tile.

The second class, `TileTopologyIterator`, is an application of the iterator design pattern as its name suggests. It can be created either from a tile topology or directly from a subclass of `Image`. It allows to iterate over the tiles defined by a tile topology. The implementation of this iterator is straightforward. It simply iterates over the tile identifiers and pass them to the corresponding tile topology to build the tiles.

3.2.4.2 Segmentation

As explained in Section 3.1.3, the segmentation is not fixed by the framework and the implementer is expected to provide its own implementation. To represent this constraint in the framework, a `Segmenter` interface was defined in package `sldc.segmenter`. It provides a single method, `segment`, which receives a NumPy representation of the image and is expected to return another NumPy array storing the binary mask marking the objects of interest contained in this image. The binary mask however doesn't conform strictly to Definition 2 as pixels belonging to an object of interest are marked with the integer value 255 (which corresponds to white in the grayscale color space) instead of 1. The `Segmenter` interface is Shown in Figure 3.5.

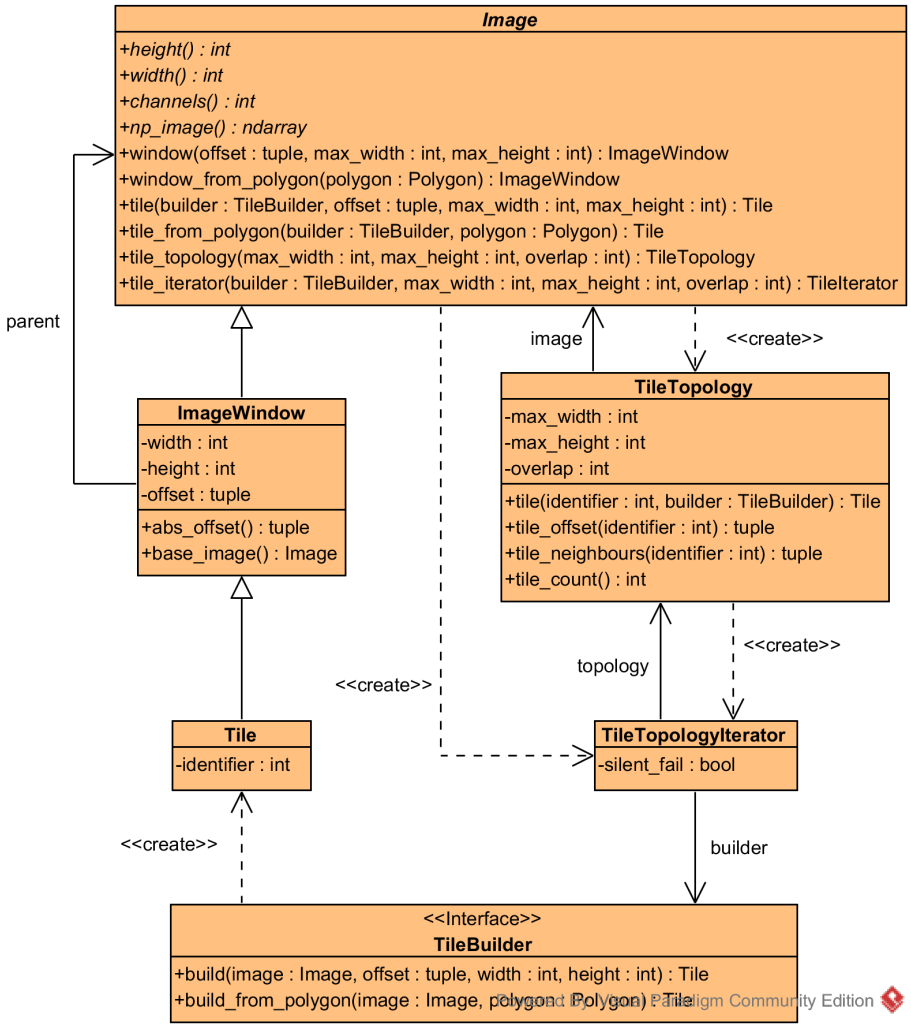


Figure 3.3: Image representation classes - package `sldc.image`

3.2.4.3 Location

As presented in Definition 3, the location procedure extracts polygons representing the geometrical contours of the objects of interest from a binary mask. The implementation of this operation was done in the single method, `locate`, of a class called `Locator` (in package `sldc.locator`). This method takes as parameter the binary mask represented by a NumPy array and returns the expected set of polygons as Shapely `Polygon` objects.

As stated in Section 3.1.3, this operation can be fixed by the framework without loss of genericity. This is made possible by the choice of representation for the method's inputs and outputs. As far as the implementation is concerned, it was largely inspired from another implementation taken from the Cytomine codebase. It uses the `findContours` procedure of the OpenCV library to extract the geometrical information of the objects as a list of coordinates. The implementation provided with the framework has two small additions compared to the Cytomine one. The first is the conversion of those coordinates into `Polygon` objects and the second is an optional translation that can be applied to those polygons. This second modification

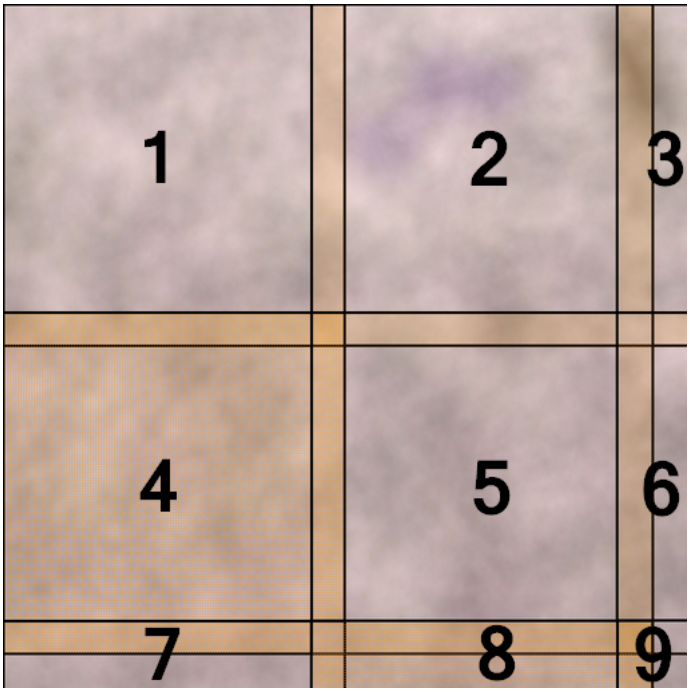


Figure 3.4: A tile topology applied on a 512×512 image (parameters: $w_m = 256$, $h_m = 256$ and $o_p = 25$). The numbers are the tile identifiers.

is needed because of the image division in tiles. Indeed, by default, the location algorithm constructs polygons referenced to the top-left pixel of the binary mask passed to `locate`. Yet, the polygons are expected to be referenced to the full image top-left pixel. An additional parameter was therefore added to the `locate` method prototype allowing the caller to specify a translation offset to apply to the found polygons. The `Locator` class is Shown in Figure 3.5.

3.2.4.4 Merging

The need for a merging phase is a consequence of the image division in tiles and its goal is to merge distinct polygons that actually represent a same object of interest. The main idea behind the algorithm was imagined by ?? JM Begon ??. It consists in building a graph where each node corresponds to a polygon. The algorithm will then add edges between polygons which correspond to a same object. Two polygons represent a same object if the distance between them (i.e. minimum distance between one point of each polygon) is less then a certain tolerance threshold. Generating the final polygons is as simple as finding all the connected components of this graph and computing the intersection of all the polygons in those components.

While working in some cases, the implementation made by Jean-Michel Begon could be improved. First, the interface of the class was inconvenient to use. Indeed, the tiles and their polygons had to be provided in a fixed order (i.e. increasing order of identifiers). And if they weren't, the merging would fail. Moreover, it had issues with some border cases. For instance, with small images containing few tiles. For those reasons, the algorithm was kept but was completely reimplemented to take advantage of the `TileTopology` object (which didn't exist in the previous

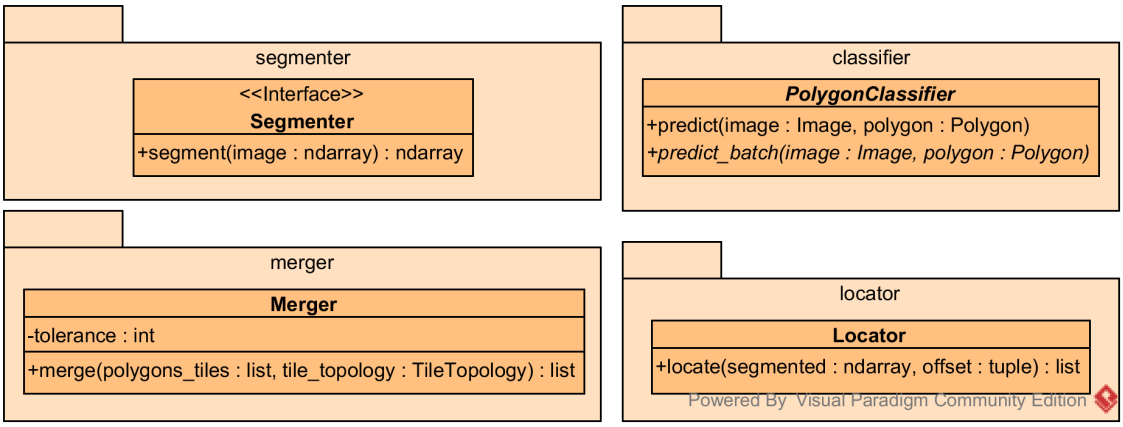


Figure 3.5: Packages `sldc.segmenter`, `sldc.locator`, `sldc.merger` and `sldc.classifier`.

implementation of the workflow).

The classes related to merging were defined in the package `sldc.merger`. The main logic of the algorithm was implemented in a class called `Merger`. Applying a merge is as simple as passing a tile topology as well a the tiles and associated polygons to the `merge` method which return the list of merged polygons. The `Merger` class is Shown in Figure 3.5.

3.2.4.5 Dispatching and classification

As defined in Section 3.1.4, the dispatching of polygons to classifiers is performed using predicates. Those predicates are materialized by the abstract class `DispatchingRule` in package `sldc.dispatcher`. The implementer can extend to define its custom dispatching logic. Especially, this is done by implementing the method `evaluate_batch` which is passed both a list of polygons to dispatch as well as the image from which they were extracted. Passing both the polygons and the image allows the implementer to define a dispatching logic based on either the polygons geometrical properties, or the polygons crops, or both.

The same philosophy was followed for classification. The implementer has to extend the abstract class `PolygonClassifier` from package `sldc.classifier` (see Figure 3.5). For the same reason as for the `evaluate_batch` method, the `predict_batch` methods takes as parameters a set of polygons and the image they were extracted from. Although only a label is produced by the classifier operator in Definition 4, an additional element is returned by the `predict_batch` method: the class probability (i.e. the probability that the predicted label is indeed the label of the object). Indeed, this information can sometimes be extracted using some classifiers (e.g. tree based methods). However, it can happen that the underlying classifier is not able to generate those probabilities. In this case, the implementer is advised to return a probability 1 for each polygon. The class `PolygonClassifier` is shown in Figure 3.5.

While the dispatching and classification logic are problem dependent, the coordination of those steps is obviously not and is implemented in class `DispatcherClassifier`

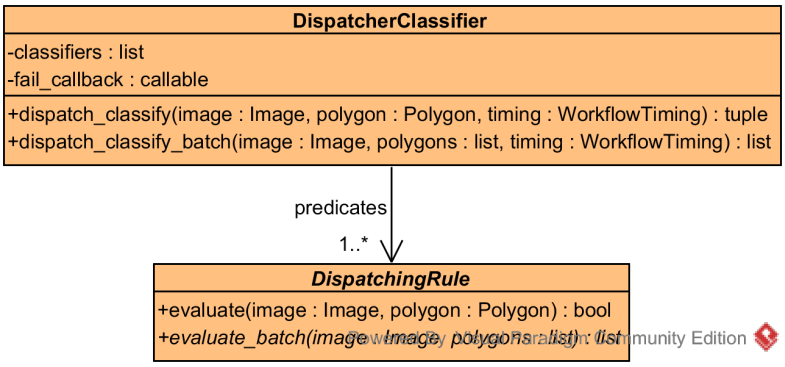


Figure 3.6: Package `sldc.dispatcher`

(see Figure 3.6). This object must be initialized with a set of classifiers and dispatching rules. Some polygons can then be dispatched and classified by passing them to the methods `dispatch_classify` or `dispatch_classify_batch`. Especially, the first will execute the operation on a single polygon while the second allows to process a set of polygons.

To answer the transparency requirement, it is essential that all the relevant information generated by these methods can be accessed by the implementer after the execution. Those information obviously include the classification label and its associated probability but not only. Indeed, another relevant information generated is the identity of the dispatching rule which matched a polygon. In practice, this information can be used by the implementer to distinguish a same classification label returned by different classifiers. The `dispatch_classify` method therefore returns a tuple containing the label, the probability and the identifier of the dispatching rule that matched the polygon (this identifier being the index of the rule in the list passed at construction). The `dispatch_classify_batch` method returns three lists containing the same information for all the passed polygons.

In the workflow, it is assumed that one and only one dispatching predicate can be true at once. In practice, the framework should handle sets of predicates which don't verify this property. Especially, it should handle a first case when more than one rule match a polygon and a second case when no rule matches a polygon.

The first case is handled by ordering the rules and to only consider the first rule that matched. Especially, the ordering is defined by the order of the rules in the list provided by the implementer at construction.

The second case is handled with a fail callback. This function is passed the polygon that didn't match any rule and return a value or object that will be used as classification label. A default callback which always returns `None`[5] is used if the implementer doesn't provide one. Moreover, the dispatch index associated to those unmatched polygons is -1.

[5] `None` is the `null` equivalent of Python.

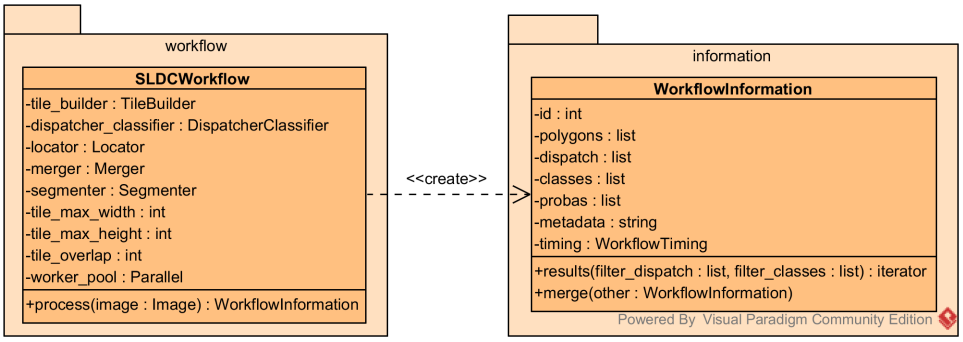


Figure 3.7: Package `sldc.workflow` and class `WorkflowInformation`

3.2.4.6 Workflow

The package `sldc.workflow` contains the actual implementation of Algorithm 2, in `SLDCWorkflow` class. Instantiating this class requires three mandatory parameters: a `Segmenter` which implements the tile segmentation logic, a `DispatcherClassifier` which was initialized with custom dispatching rules and polygon classifiers and a `TileBuilder` for building the tiles of the tile topology. The workflow can then be launched on an `Image` object using the method `execute`. This method returns all the information about the objects of interest found in the image. Those information include the polygons encoding the object's shapes and locations, their predicted classes, the associated probabilities and the dispatching indexes (see Section 3.2.4.5). In order to provide a convenient access to those information, they were encapsulated into an object called `WorkflowInformation`. Especially, this class provides a way to iterate over the results, the method `results`. The UML diagram containing both the `SLDCWorkflow` and `WorkflowInformation` classes is given in 3.7.

3.2.4.7 Workflow chain

To this point, the presented classes provide a way for an implementer to apply Algorithm 2. The package `sldc.chaining` allows to go one step further as it contains the necessary components for applying Algorithm 3. Especially, the class `WorkflowChain` coordinates the execution of several workflows one after another on one or more images and also handles the post processing of the generated data. Those operations are handled by different components defined hereafter. The UML diagram containing the classes of this package is shown on Figure 3.8.

The images to be processed by the workflow must be generated by an implementation of the interface `ImageProvider`. The implementer must define the image generation in the abstract method `get_images`.

The post processing of the generated data must be defined by the implementer as `PostProcessor` object. Especially, he has to implement the method `post_process` which is passed a collection of workflow information objects as well as the image from which they were generated.

As far as the workflow objects to be executed are concerned, they must be encapsulated into subclasses of `WorkflowExecutor`. This component has three main responsibilities.

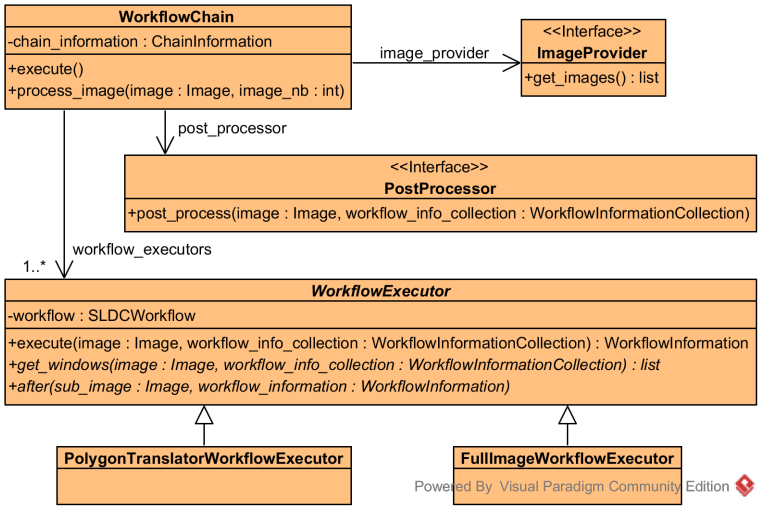


Figure 3.8: Package `sldc.chaining`

The first is to generate the image windows that will actually be processed by the underlying `SLDCWorkflow` object. Particularly, those windows must be generated based on the polygons generated from the previous steps of the chain. This generation must be implemented in the `get_windows` method. This method is also the placeholder for the filtering specified in Definition 10. As far as the first workflow of the chain is concerned, its `get_windows` method should have a slightly different behavior. Indeed, in this case, the full image is yet to be processed and should be returned. In the optic to reduce the work of the implementer, a abstract subclass named `FullImageWorkflowExecutor` was defined to implement this behavior. Its `get_windows` method simply returns the image it is passed.

The second responsibility is to launch the `execute` method of the `SLDCWorkflow` object on the images generated by the executor in `get_windows` and to collect the generated workflow information objects. This is done in the `execute` method.

The last responsibility is the post-processing of the results generated by one workflow execution. This logic must be implemented in the method `after` which is passed the image window that was processed as well as the workflow information object returned by the `execute` method of `SLDCWorkflow`. An example usage of this method is the translation of polygons generated by a `SLDCWorkflow` on an image window. Indeed, in this case the polygons returned by the workflow object are reference to the window top left pixel while they should be referenced to the full image top left pixel. For the same reason as the `FullImageWorkflowExecutor`, a subclass of `WorkflowExecutor` was created. Its `after` method implements the translation logic.

As soon as the `ImageProvider`, `PostProcessor` and `WorkflowExecutor` objects are constructed, they should be passed to the `WorkflowChain` constructor. The chain can simply be started by calling the `execute` method.

3.2.4.8 Logger and workflow timing

To fulfill the transparency requirements, it is essential that the person who executes a workflow chain is able to monitor the progress. He should also have some insights about how the workflow performs on a given problem. For instance, how many tiles must be processed, how many polygons were found, how many polygons were dispatched,... The user should also be informed about the execution times of the various phases. In order to perform those operations two other packages were added.

Logging The first one is `sldc.logger` which provides a flexible, powerful and thread-safe logging system. Especially, it allows to log messages selecting a level verbosity among *silent*, *debug*, *info*, *warning* and *error*. The output can be controlled using a minimum level of verbosity. All messages sent below this level won't be outputted. The implementer can also choose where the messages will be printed (in a file, on the standard output,...).

The logging package is articulated around the abstract class `Logger` which holds the minimum level of verbosity and provides methods to log messages in all the defined levels of verbosity. It also implements the message formatting. Especially, the messages sent by the implementer are augmented with a prefix containing the thread id, the current date and time as well as the level of verbosity at which it was sent.

What this class doesn't define is where the formatted messages will be printed. This is the responsibility of the subclasses. Three of them are provided in the package: `StandardOutputLogger`, `FileLogger` and `SilentLogger`. The first one prints the messages into the program's standard output, the second prints them into a file while the last ignores all messages. If the implementer is not satisfied with one of those implementation, he can define himself a subclass that handle messages in a custom way.

The final component of the package is an abstract class called `Loggable`. Its first goal is self documentation for the classes which extend it. Indeed, those are expected to support logging. The second goal is to provide a logger attribute for the classes which extend it. This way, they don't have to define their own.

The UML diagram of the logging package is shown in Figure 3.9.

Timing The second package, `sldc.timing`, contains the `WorkflowTiming` class which allows to record execution times of the various phases of the workflow but also to report them. The time computation is provided through some `start` and `end` methods for each phase. For instance, for recording segmentation time, the methods `start_segmentation` and `end_segmentation` are provided. The phases that can be recorded are the following: image loading, segmentation, location, merging, dispatching and classification. A last phase is actually a combination of the loading, segmentation and location phases and is called *lsl*. An additional method is needed for this combination because it can be parallelized (see Section 3.2.4.10). Recorded execution times can be extracted with a handful of methods such as `total` which computes the total recorded time for all phases, or `report` which is passed a `Logger` object and prints some statistics about the execution times. The UML diagram of

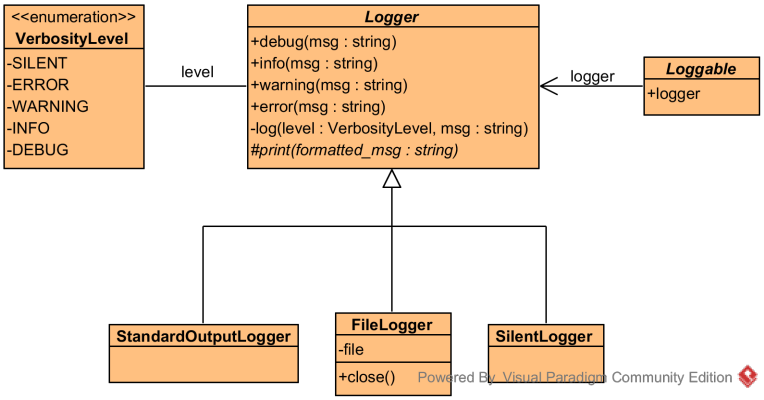


Figure 3.9: Package `sldc.logging`.

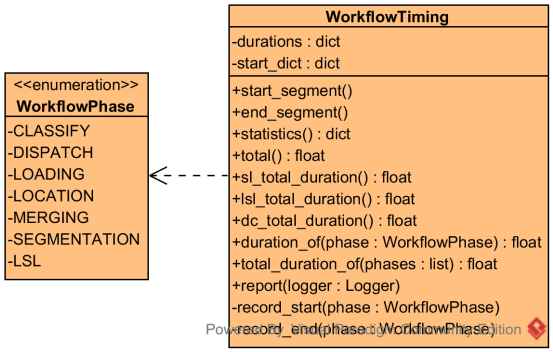


Figure 3.10: Package `sldc.timing`.

this package is shown in Figure 3.10.

3.2.4.9 Builders

The package `sldc.builder` contains two classes `WorkflowBuidlder` and `WorkflowChainBuilder` for making easier the construction of `SLDCWorkflow` and `WorkflowChain` objects respectively. Those classes provide some methods for setting the construction parameters and a `get` method for actually constructing the expected object based on the provided parameters. For instance, the `WorkflowBuilder` provides a method `set_segmenter`. The UML diagram of this package is shown in Figure 3.11.

3.2.4.10 Parallelization

As stated in the requirements, the framework should allow the user to take advantage of parallelism to reduce overall execution time. First, it is important to understand few things about parallelism in Python. The language natively provides packages for parallelizing code: `multiprocessing` and `threading`. A library called `joblib` was built on top of those packages and provides a high level interface for writing parallelized loops in a very concise way. As far as threading is concerned, some implementation of the interpreter (i.e. CPython) prevents the threads to execute concurrently because of the Global Interpreter Lock (GIL). This lock is acquired by the thread which executes and prevents the other threads to access the interpreter

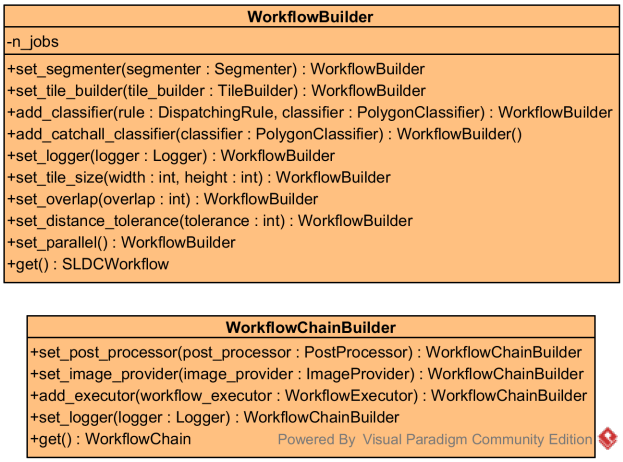


Figure 3.11: Package `sldc.builder`.

for executing their own code. Therefore, it is advised in the language documentation to use multiprocessing to ensure that code is effectively executed in parallel whatever the interpreter implementation. Working over several processes however has the drawback of requiring inter-process communication. Particularly, the processes must be passed the data to treat and return the generated results to the main process. This is handled by `joblib` using serialization. Especially, the elements to be processed in parallel are queued. When a process becomes available, an element is dequeued, serialized and transferred to this process. When it terminates its execution, the results are themselves serialized and returned to the main process. This organization has the drawback of triggering as many serialization and deserialization as there are objects to process. Yet, such operations induce a non-negligible overhead that can be overcome by passing batches of elements instead of single elements to the processes. Another important point is the fact that `multiprocessing` doesn't support nested parallel loops. This constraint imposes therefore at most one level of parallelism. That is, a code executing in a spawned process cannot itself spawn other processes. All these language and library specific constraints were taken into account when including the parallelism to the framework.

How the parallelism will be implemented is now known, the question yet to be answered is where it will be applied. Several steps of the algorithm can be retained as candidates because of their highly parallelizable nature. In this case, highly parallelizable means that the parallelization can be done without any synchronization mechanisms except the ones provided by `joblib`. Typically, this is the case for operations which imply several independent computations. At the workflow level, the candidates steps are tiles segmentation and location, polygons dispatching and polygons classification. At the chaining level, the processing of the images generated by the `ImageProvider` is another candidate.

Whereas applying parallelism at the chaining level is very easy, the idea was abandoned for the following reasons: it would prevent the parallelization at the workflow level (because of the nested parallel loops issue) and it can be done manually by the implementer. Indeed, he just has to launch its program one time on each image to obtain the same result.

At the workflow level, both dispatching and classification parallelization were dropped because it would require more work and the need was less obvious than for segmentation and location. Indeed, it was assumed that the segmentation procedure provided by the implementer was likely to be computationally expensive. So the advantage resulting from this parallelization might be greater than for the two other operations.

The parallelization itself is handled in the `SLDCWorkflow` class. In order to reuse the same pool of processes for every call of the `execute` method, this pool must be passed to the constructor of the workflow object as a `joblib.Parallel` object. In order to provide more feedback about progress to the user in the sequential case, two implementations of segmentation and location were made. Therefore, the workflow switches to one or another implementation according to the number of jobs specified by the user. The parallel implementation first splits the tiles in batches and then submits them to the various processes using the pool. After that, it aggregates the returned data. Especially, in addition to the found polygons, each process returns a `WorkflowTiming` object containing the loading, segmentation and location times it recorded when processing its assigned tiles. Those objects are merged with the `WorkflowTiming` to be returned by the `execute` method.

To avoid any concurrency problems, all classes of the framework were developed to be thread safe. This was done by making those classes immutable whenever possible or by avoiding to use shared resource. The only classes which couldn't verify this rule are `WorkflowTiming`, `StandardOutputLogger` and `FileLogger`. The timing objects can indeed be updated with new time recordings after their creation. The loggers by printing log messages access shared resources (e.g. standard output, files,...). To prevent any problem, the `Logger` was added a lock object to synchronize the call to the `_print` abstract method which actually implements the submission of the message to the resource.

3.2.5 Testing

In order to ensure that the various components of the framework are working as expected in some predefined conditions and to prevent those components to be broken by further refactoring, some tests were written using the `unittest` package of Python. Those tests can also be found on GitHub in the folder `tests`.

The tests were focused on components containing actual logic, that is, the classes `Locator`, `DispatcherClassifier`, `Merger` and `TileTopology`. The workflow construction and execution was also tested on two use cases. The first is presented in Section 3.2.6 and the second consists in finding a big white circle in a image with black background. Finally, there are fifteen tests and they yield a code coverage of 72 %.

3.2.6 Toy example

Now that the implementation was presented and detailed, this section aims at highlighting how easy it is to apply the framework to solve a problem. The problem in

question is very simple and consists in finding grey and white squares and circles within a greyscale image with a black background (see in Figure 3.12). In addition to locate the shapes, the algorithm should return the information about whether a shape is a circle or a square and also return a label indicating its color (grey or white).

To apply the workflow philosophy, the implementer should first encapsulate its image custom format in a class extending `Image`. For this example, a simple NumPy array can be used to represent the image. The definition of the image class is given in Listing 3.1. The next component to be defined is the segmentation algorithm that will actually detects the objects. In this case, this algorithm can be implemented using a simple thresholding (every pixel of which the value is greater than 0 belongs to an object). This logic should be defined in a class implementing the `Segmenter` interface. The definition of this class is shown in Listing 3.4. Thanks to the usage of NumPy arrays, the implementation of the segmentation is really concise.

The next step is the definition of the dispatching rules that will redirect the objects to an appropriate classifier. Especially, the idea is to take advantage of dispatching for detecting whether a shape is a circle or a square. In this case, two rules are needed: one that evaluates to true the circle polygons and another one which evaluates to true square polygons. One way to distinguish circles and squares is using the circularity shape factor. It is a real value between 0 and 1 which measures how close the shape of an object is to that of a circle. Especially, perfect circles have a circularity 1 and straight lines have a circularity 0. In this case, because the shape is discretized in the image, the algorithm will never produce perfect circles so detecting circular shapes must be done by thresholding the circularity. Particularly, polygons having a circularity greater than 0.85 can be considered circles while the others can be considered squares. The implementation of the dispatching rules are given in Listing 3.2. Thanks to the Python list comprehension syntax, the definition of the rules is again really concise.

Now that the segmentation and dispatching rules are defined, the last missing element is the classifier. In this case, it should produce the last desired information which is the color of the shapes. A simple idea is to use the polygon to retrieve the central pixel of the shape. Then, the greyscale value of this pixel can be checked to identify whether the color of the shape is white or grey. The implementation of the classifier is given in Listing 3.3. In the context of this example, the image to be processed might not be large. However, the classifier is implemented so that the full image is never loaded into memory. Indeed, before extracting the pixels, the window boxing the polygon is extracted from the image and only the NumPy representation of this window is loaded into memory (see `image.window_from_polygon()` and `window.np_image` method calls).

Listing 3.1: Toy example - Encapsulating custom image format

```
class NumpyImage(Image):
    """An image represented as a NumPy ndarray"""
    def __init__(self, np_image):
        self._np_image = np_image

    @property
    def np_image(self):
```

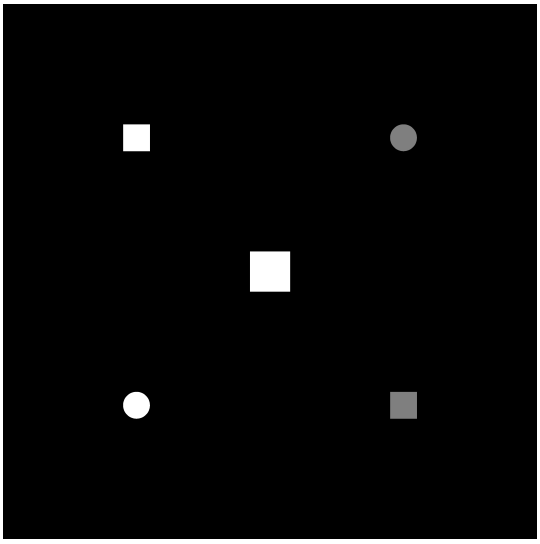


Figure 3.12: Example image to be processed for the toy example

```
        return self._np_image

    @property
    def channels(self):
        shape = self._np_image.shape
        return shape[2] if len(shape) == 3 else 1

    @property
    def width(self):
        return self._np_image.shape[1]

    @property
    def height(self):
        return self._np_image.shape[0]
```

Listing 3.2: Toy example - Dispatching rules

```
class CircleRule(DispatchingRule):
    """Dispatching rule which matches circles"""
    def evaluate_batch(self, image, polygons):
        return [circularity(polygon) > 0.85 for polygon in polygons]

class SquareRule(DispatchingRule):
    """Dispatching rule which matches squares"""
    def evaluate_batch(self, image, polygons):
        return [circularity(polygon) <= 0.85 for polygon in polygons]
```

Listing 3.3: Toy example - Classifier

```
class ColorClassifier(PolygonClassifier):
    """Classifier that predicts the color of a shape"""
    GREY = 0
    WHITE = 1
    def predict_batch(self, image, polygons):
        classes = []
        for polygon in polygons:
```

```
            # Fetch center pixel
            window = image.window_from_polygon(polygon)
            sub_image = window.np_image
            c_x = int(polygon.centroid.x) - window.offset_x
            c_y = int(polygon.centroid.y) - window.offset_y
            pxl = sub_image[c_y][c_x]
            # Generate the label based on the pixel color
            if pxl == 255:
                classes.append(self.WHITE)
            elif 0 < pxl < 255:
                classes.append(self.GREY)
            else:
                classes.append(None)
        return classes, [1.0] * len(polygons)
```

Listing 3.4: Toy example - Segmentation implementation

```
class CustomSegementer(Segmenter):
    """Every non black pixel are in an object of interests"""
    def segment(self, image):
        return (image > 0).astype("uint8")
```

Listing 3.5: Toy example - Applying the framework

```
# Build the workflow
builder = WorkflowBuilder(n_jobs=1)
builder.set_segmenter(CustomSegmenter())
builder.add_classifier(CircleRule(), ColorClassifier())
builder.add_classifier(SquareRule(), ColorClassifier())
workflow = builder.get()

# Execute
results = workflow.process(NumpyImage(image))
```

3.3 Improvements and future work

This section presents some possible improvements that could be performed to increase the ease of use and the performances of the framework.

3.3.1 Memory management

Under the hood, Python wraps everything into objects [Sei13]. This includes primitive types such as integers. Moreover, as memory allocation is an expensive operation, those objects are pooled by the Python runtime. Especially, it stores list of free objects which can be re-used when the user implicitly request them. The problem is that the free objects in those lists are never released and therefore, from the operating system point of view, the memory needed by the program only increases. It might worth checking the impact of this memory management policy over the memory consumption of the framework. Especially, it if happens to be a problem, the framework should be analyzed to check which parts yield a high memory consumption and those parts should be re-written.

3.3.2 Location algorithm

The location algorithm sometimes fails at generating polygons for all the detected objects in the segmentation mask. This happens when some objects' masks are ill-formed yielding invalid polygon containing self-intersections. A self-intersection is a point where two edges of a polygon intersect or are colinear. A first procedure for cleaning the segmentation mask has been implemented but it seems not be sufficient to prevent invalid polygons to be generated. An improvement would therefore consists in understanding which kind of patterns yield self-intersections and to implement a procedure for cleaning those patterns.

3.3.3 Parallelization

Parallelization was successfully applied for tiles segmentation and location but other phases of the workflow might also benefit from it. For instance, when the dispatching and classification procedures cannot be parallelized by the implementer, those steps are executed sequentially no matter what. This can present a major issue especially when the number of objects found in the image is high. A possible improvement would consist in splitting the detected objects in batches and to execute dispatching and classification for each batch on different processes.

Another critical point is the workflow executor. Indeed, when the implementer implements a processing chain, the first executor typically processes the full image and the parallelization can happen at the workflow level. According to the number of objects detected by this first step, the subsequent executors might generate a lot of image windows to be processed by other workflows. Currently, all those windows must be processed sequentially which can potentially yield huge execution times. An improvement would then consist in parallelizing those windows processing. This would have to be implemented carefully as this parallelization shouldn't clash with the one implemented at the workflow level (see Section 3.2.4.10 for the nested parallel loops issue).

Finally, it would be interesting to optimize the parallelization of the tiles processing. Currently, batches of tile objects are passed to the processes which requires a potentially heavy serialization. Indeed, tiles being defined by the implementer, he could stores heavy objects in the class attributes. A possible improvement would therefore consist in passing batches of tiles identifiers instead of the tile themselves and to use the tile topology to re-build the tiles on each process.

Chapter 4

SLDC at work : the thyroid case

In this chapter, the *SLDC* framework is applied to the problem presented in Chapter 2 : the nodule malignancy assessment. This problem is effectively an instance of the object detection and classification problem. Indeed, the goal is to diagnose malignancy by the presence or absence of cells or groups of cells having particular characteristics in digitized microscope slides. This problem is a good use case for the *SLDC* framework: the images are large (i.e. typically 15 giga-pixels), two distinct categories of objects must be found (namely cells and groups of cells) and some of these objects can be included into others which can be handled using dispatching and chaining.

An introduction to the thyroid problem as well as the underlying implementation challenges are presented in Section 4.1. Then, the workflow developed in [Deb13] is briefly presented and its performances are assessed in Section 4.2. Especially, some flawed steps are highlighted and some improvements are proposed. Then, the implementation of the improved workfow is detailed in Section 4.3. Finally, the performances of this implementation are analyzed in Section 4.4.

4.1 Problem and underlying challenges

The problem consists in finding cells with inclusion and proliferative architectural patterns in large digitized microscope slides. To perform this detection, a dataset containing approximately 6000 annotations was created by experts on the Cytomine platform (see Section **??**). The major challenges involved with this problem are detailed hereafter.

Image quality While the image resolution is more than acceptable, the images themselves are by nature not very well suited for object extraction. Indeed, the objects of interests are surrounded with a lot of other undesirable objects. Moreover, due to the imprecise nature of the staining performed before digitization, some staining variations can appear across slides or within a slide.

Image size As explained in Section 2.2.1.2, image sizes range from 4 giga-pixels to 18 giga-pixels. Therefore, the various processing steps should be as efficient as possible to avoid huge execution times. Also, accessing the images must be done

through HTTP requests. Therefore, a particular attention should be paid to the number of requests to be executed for fetching the image. Especially, some caching policy might be needed to reduce those requests execution time overhead.

Class imbalance The dataset of annotations is relatively balanced if all terms are considered separately. However, grouping terms for expressing the detection as a binary (or ternary) problem results in class imbalance, especially for the cells with inclusion versus normal cells problem.

Human annotations The human annotations are imperfect as experts usually annotate objects roughly (i.e. an annotation can be larger than the actual object). Moreover, some annotation drawing tools provided on the Cytomine platform generate particular shapes such as circles or rectangles. Assuming that an algorithm will annotate the cells more precisely, the resulting differences in terms of geometry and information content of the crops might affect the performances of any classifier fitted on those experts' annotations.

4.2 First workflow

The workflow developed by Antoine Deblire in [Deb13] is summarized in Figure 4.1. The idea behind this workflow is fairly simple. First, a segmentation is applied to the whole slide to extract standalone cells and architectural patterns (step 4.3). The detected objects are then differentiated using their area and circularity (step 4.4) and dispatched to a classifier (steps 4.6). Especially, architectural patterns are classified as proliferative or non-proliferative by a first classifier and the cells are classified as inclusion or normal by a second one. Then, architectural patterns are segmented using a second segmentation algorithm (step 4.5) to extract the cells they contain and thos cells are also passed to the cell classifier. The next sections aim at explaining more thoroughly those steps.

4.2.1 Segmentation procedures

Segmentation is carried out in two steps. The first segmentation procedure was designed for processing the whole slide and relies on a process called color deconvolution [RJ01]. This process consists in retrieving the stains concentration of the objects contained in the image based on the RGB values of the pixels (see Section **??** for the staining process applied to the slides before digitization). Especially, given that the original slide was prepared with S stains, the color deconvolution process generates a set of S images where the pixel p^s_{ij} of an image is the concentration level of the stain s in the pixel p^o_{ij} of the original image. This process is particularly useful in this case because cells and patterns have a high concentration of a given stain. The first segmentation procedure starts by generating the concentration image for the first stain. As pixels with a high concentration are supposed to be in an object of interest, a first binary mask is created by thresholding the concentration image. Three morphological operations are then successively applied to the segmentation mask:

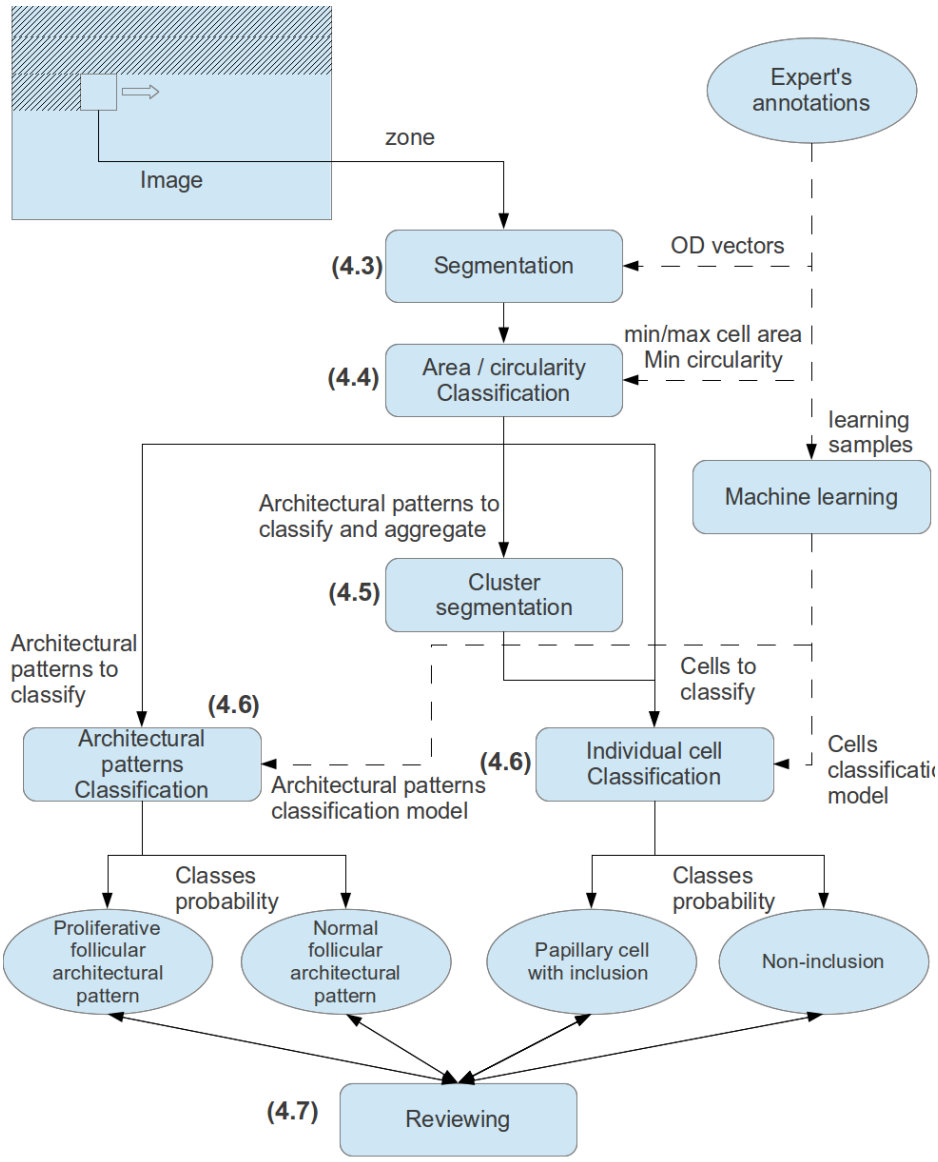


Figure 4.1: Antoine Deblire's workflow (source: [Deb13])

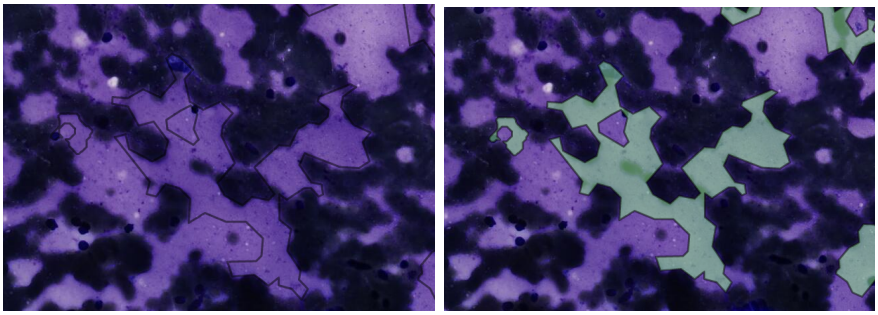


Figure 4.2: Background detected by the segmentation as an object. The background was segmented because of a higher stain concentration which is visible on the left image. The segmented area is delimited by the annotation on the right image.

- Morphological closing to eliminate small holes in the detected objects.
- Morphological opening to eliminate small objects supposed to be irrelevant due to their size.
- Morphological closing to unify close neighbouring objects supposed to be part of a pattern.

Some example segmentations are provided in Figure 4.2 and 4.3. It seems that the procedure is able to detect most of the objects of interest although it sometimes fails at covering those objects' whole area. For instance, in the fourth example image, there is a hole in the mask inside the pattern and this hole covers some cells that should be included in the mask. On the fifth example, one can see three standalone objects above the central pattern. Those objects' masks are smaller than their corresponding cells. Also, on some images, regions containing a high stain concentration were marked as object by the segmentation. This can be seen on the first example image in Figure 4.2 where a portion of the slide background was detected as pattern.

The second segmentation procedure is applied to the detected patterns and was designed to isolate individual cells inside those patterns. The implementation is slightly more complicated than the first (see [Deb13] for the full procedure). Similarly, it starts with a color deconvolution to highlight the cells. However, the stain concentration image is not transformed into a binary mask using a fixed threshold but using Otsu's method [Ots75]. Using the `findContour` procedure of the OpenCV library as well as morphological operations, independent cells are located and cleaned one after another. Finally, a watershed algorithm is applied to separate cells that overlap. Some example segmentations are provided in Figure 4.4. The segmentation seems to work relatively well on "clean" patterns: that is, where cells do not overlap much and are clearly distinguishable from the pattern background (see the first two examples in Figure 4.4). On "dirty" patterns however, the segmentation performs poorly as it either returns large patches which do not correspond to cells or fails to separate overlapping cells (see the two last examples in Figure 4.4). In both cases, one can notice that the detected cells are slightly under-segmented (i.e. the segmentation mask is smaller than the actual object).

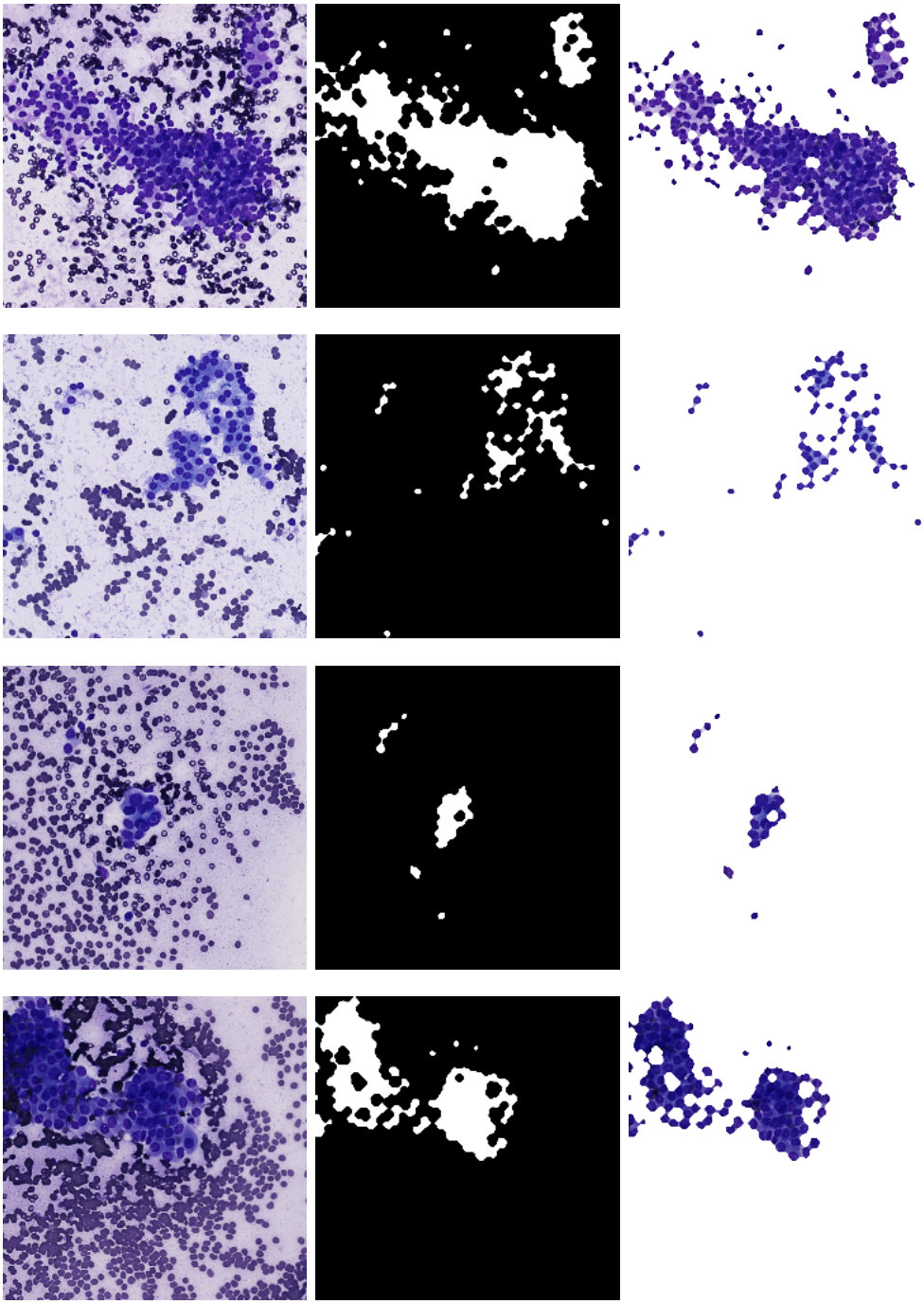


Figure 4.3: Examples of slide segmentation. For each example, three images are given: the original image to the left, the segmentation mask in the center and to the right, the original image in which the pixels that do not belong to the segmentation mask were replaced by white pixels.

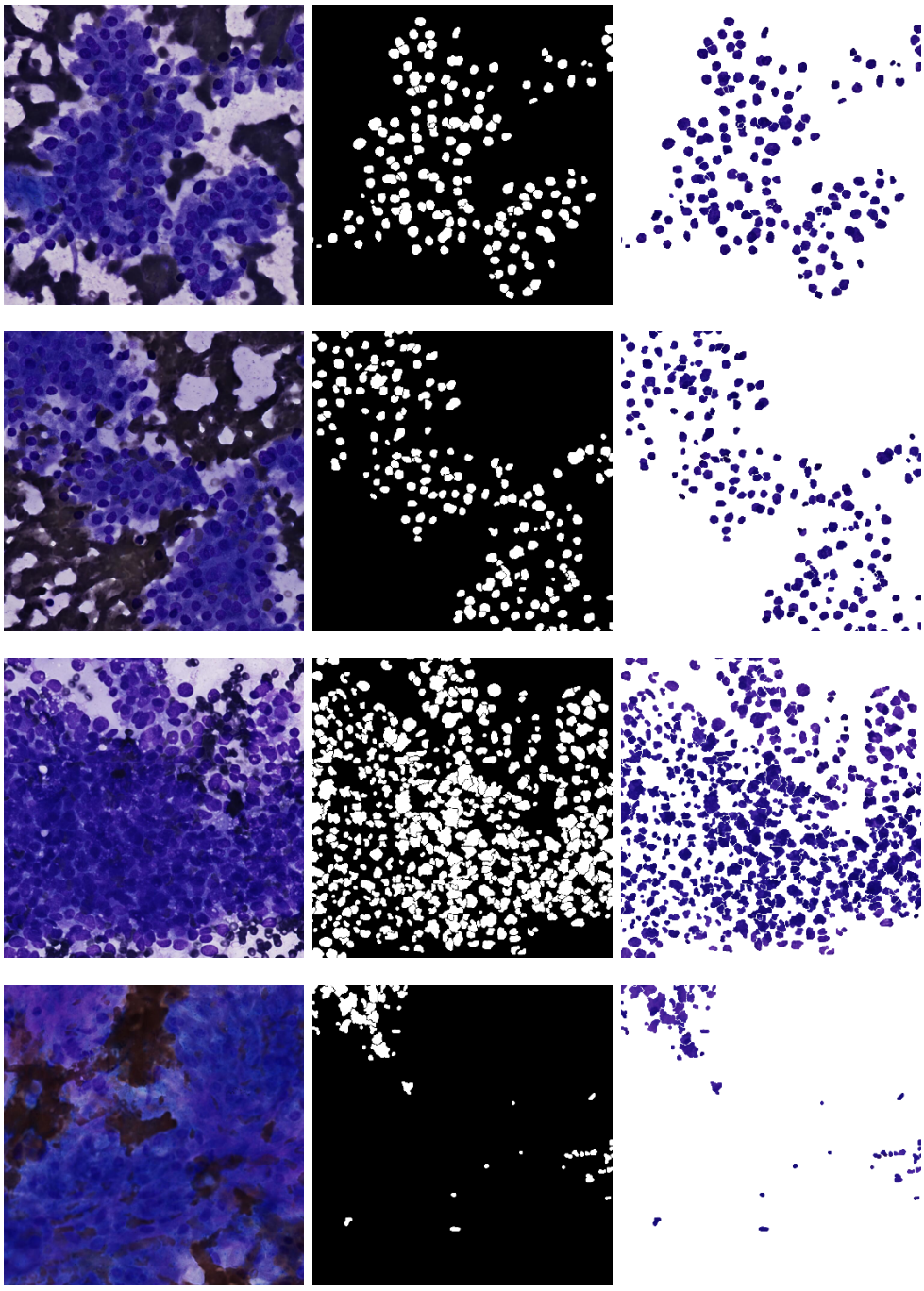


Figure 4.4: Examples of aggregate segmentation. See Figure 4.3 for explanations.

A_{min}	31 μm^2
A_{max}	102 μm^2
C_{min}	0.7
N_{min}	4

Table 4.1: Dispatching parameters presented in [Deb13]

While the presented segmentation procedures exhibit some flaws, they were considered acceptable to test the *SLDC* framework.

4.2.2 Dispatching procedure

4.2.2.1 Slide processing dispatching

The step (4.4) consists in dispatching detected objects into four categories: artefacts, cells, clusters and patterns. The categories *artefact* and *cluster* respectively correspond to irrelevant objects and to groups of cells that contains too few of them to be patterns. Even if the author distinguishes patterns and clusters at the dispatching step, objects of both categories are treated equally in the subsequent steps of the algorithm. That is, they are first evaluated by the pattern classifier (for assessing whether they are proliferative or not) and they are re-segmented. The dispatching is based on four parameters, the cell minimum and maximum areas (respectively, A_{min} and A_{max}), the cell minimum circularity (C_{min}) and the minimum number of cells per pattern (N_{min}). The values of those parameters as defined by Antoine Deblire are given in Table 4.1. The dispatching rules can be summarized as follows:

- **Artifact**: all objects having an area less than A_{min} or an area less than A_{max} and a circularity less than C_{min}
- **Cell**: all objects having an area A such that $A_{min} < A < A_{max}$ and a circularity greater than C_{min}
- **Clusters**: all objects having an area A greater than A_{max} such that the object can contain at most N_{min} cells:

$$A_{max} < A < N_{min} \times A_{max}$$

- **Patterns**: all objects which do not match one of the rules above are considered as patterns

In order to find values for those parameters, the author extracted the crops of the cells annotated by the experts, applied to them one of the segmentation procedures presented in Section 4.2.1, and finally computed the area and circularity of the resulting shapes. While the author does not precise in the thesis which segmentation procedure was applied, it is probably the second (i.e. pattern segmentation). Indeed, as the first was designed to segment both patterns and cells, it would fail at isolating an annotated cell located inside a pattern. While the idea behind this procedure is sound, the geometrical features will probably not be as accurate as expected because of segmentation imperfections. Typical cases when the segmentation fails

(a) Under-segmentation (b) Detection of several objects (c) Detection of another cell included in the mask

Figure 4.5: Cases when the segmentation fails at providing accurate results for area and circularity computations. For each example, images to the left and to the right are respectively the original image with a mask representing the annotation shape and the image resulting from the application of the segmentation mask.

at providing accurate results are, for instance, cell under-segmentation (see Figure 4.5(a)) and detection of several objects instead of one. The second case can happen either by splitting a cell in several sub-objects (see Figure 4.5(b)) or because the expert's annotation covers other cells (see Figure 4.5(c)).

However, one can assume that the author overcame those issues. Indeed, as stated in the thesis, he generated the circularity and area histograms and then used them to evaluate the thresholds presented in Table 4.1. However, the author made a mistake in this process as he didn't check whether those thresholds were valid regarding patterns areas and circularities. Especially, are those dispatching rules likely to dispatch cells as patterns or patterns as cells? In order to evaluate this question, the following methodology was applied: crops of annotated patterns and cells were extracted from Cytomine. The former were segmented using the first segmentation and the latter with the second one. For each annotation, the convex hull of the union between the segmented objects was taken as a temporary mask. This operation allows to handle the multi-objects detection problem and also, in some cases, to mitigate the impact of under-segmentation on the resulting area. Finally, in order to make sure that the temporary mask didn't cover areas outside of the expert's annotation, the final mask was generated by taking the intersection between the annotation mask and the temporary mask. This process is illustrated in Figure 4.6.

The histograms given in Figures 4.7 and 4.8 respectively show the area and circularity distributions of the segmented experts' annotations. First, it appears that, whatever the metric, there is a substantial overlapping between the cells' and patterns' distributions. This has a major consequence for the dispatching procedure presented above. Indeed, as it relies on simple thresholdings, it is ineffective at separating the objects. Another observation is that the parameters given in Table 4.1 are not relevant as most of the cells would be dispatched as patterns with such values. This observation is confirmed with the scatter plot shown in Figure 4.9. In this plot, only few cells are effectively dispatched as such (see the blue box in the top left corner of the plot), the others being dispatched as patterns. Given those observations, it is clear that the dispatching procedure must be re-worked.

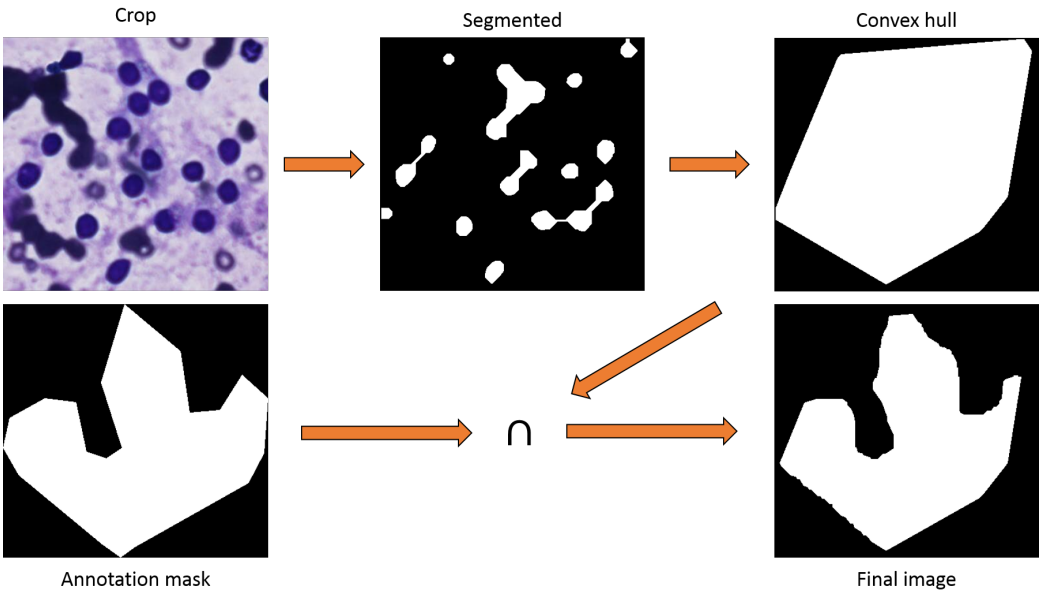


Figure 4.6: Cleaning process for area and circularity assessment.

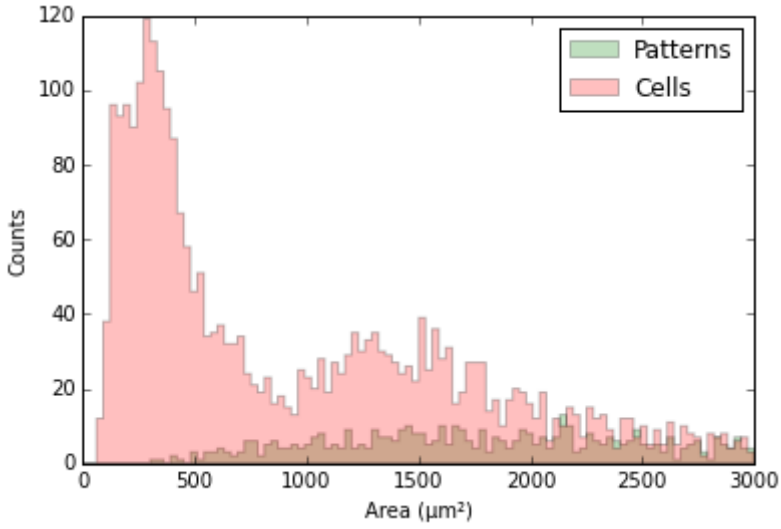


Figure 4.7: Area distributions of the segmented experts' annotations.

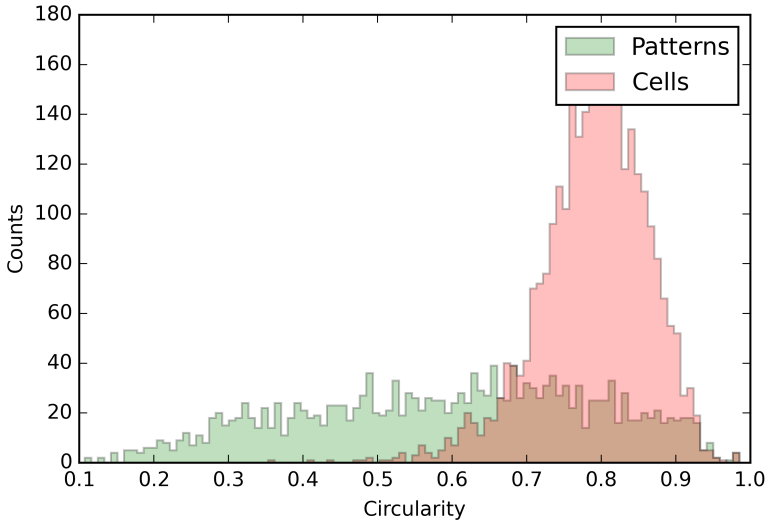


Figure 4.8: Circularity distribution of the segmented experts' annotations.

4.2.2.2 Improvement

As relying solely on geometrical properties is not a viable solution, an alternative consists in using the objects' crop image. Especially, the objects' crop would be classified into one of the dispatching categories (i.e. cell, pattern or other) using the random subwindows image classification algorithm [MGW16] (this algorithm is detailed in Appendix C). A drawback of this solution is that the dimensions of the objects are completely ignored. Given that some patterns might have a similar appearance than cells (color and shape), this might lead to misclassification.

The avoid the dispatching problem induced by simple thresholdings, one could include the geometrical information of the polygons into the learning and prediction processes. Especially, using the ET-FL variant of the random subwindows algorithm, the area and circularity would be appended to the feature vector generated from the extra-trees classifier. This augmented feature vector would then be passed to the SVM classifier for prediction. While intuitively, this solution seems appealing, it might need to be refined a little more. Indeed, the number of features generated from the extra-trees classifier is relatively large (e.g. for the models presented in Section 4.4.1, this number can reach 30000) and the geometrical features might therefore be overlooked by the SVM classifier. To overcome this problem, a kernel that would increase the contribution of the geometrical features could be used. Whereas this solution might yield better results than the first, it requires the experts' annotations of the learning set to be cleaned to avoid the problem mentioned in Section 4.1. Also, it would require a non-negligible modification of the random subwindows algorithm. As the goal is mostly to apply the framework and considering the amount of work implied by this solution, it was considered out of the scope for this thesis and geometrical features were not included among the inputs of the dispatching classifier.

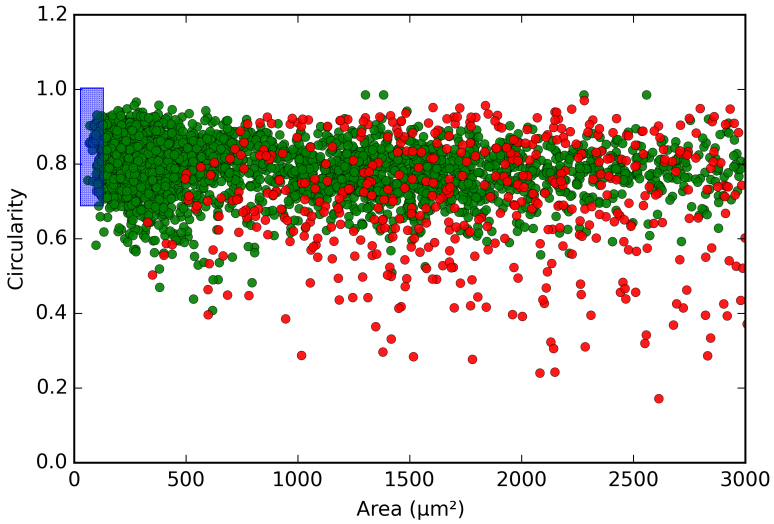


Figure 4.9: Scatter plot, circularity versus area. Green and red dots correspond respectively to cells and patterns. The blue box is the cell dispatching zone.

4.2.2.3 Pattern processing dispatching

The dispatching strategy designed in [Deb13] for the pattern processing phase is simpler. As the objects to be processed by this step are assumed to be cells, the dispatcher filters the detected objects using their geometrical properties. All objects having an area either less than A_{min} or greater than A_{max} or a circularity less than C_{min} are not dispatched the to cell classifier. As explained in previous sections, those parameters are not reliable for dispatching and a new strategy must therefore be defined. It should take into account the flaws of the second segmentation procedure. Especially, it should eliminate the large patches segmented on "*dirty*" patterns. Typically, those are non-spherical (see Figure 4.4) and this property can be used to filter them. Moreover, it might be interesting to filter small artefacts and this can be done by filtering objects having a small area. The resulting dispatching rule is the following: every object of which the area is less than A' or of which the circularity is less than C' are removed. Especially, A' was set to $15\mu m^2$ and C' to 0.6. It can be seen in Figures 4.7 and 4.8 that those thresholds allow to capture almost all cells.

4.2.3 Classification

As soon as objects are dispatched, they have to be classified. In [Deb13], the author uses two classification models: one for cells and another one for patterns. For patterns, a ternary classifier is used and predicts the following classes: *proliferative pattern*, *non-proliferative pattern* and *other*. The author states that the third class is needed because with a binary classifier, some objects were classified as patterns while they were not patterns. Hopefully, with the new dispatching procedure, those objects will be eliminated before reaching the classifier. It was therefore decided to use a binary classifier for performing this classification.

As far as the cell classifier is concerned, it predicts two classes: *cells with in-*

clusion and *non-inclusion*. In addition to the term *cell with inclusion*, the author includes the *pseudo inclusion* in the positive class in order to avoid missing some real inclusions that might look like a pseudo one.

The final classifiers used for this implementation are detailed in Section 4.4.1.

4.3 Implementation

Following the *SLDC* framework philosophy, the problem dependent components have to be defined: the image representation, the segmentation procedures, the dispatching rules and the classifiers. Whenever possible, the components were developed to be reusable for other applications within Cytomine. Those generic components are coloured in blue in UML diagrams while the problem dependent components are coloured in green.

4.3.1 Image representation

The first component to be defined is the actual representation of the images to be processed, that is, the digitized microscope slides stored on the Cytomine platform. This representation is implemented in the `CytomineSlide` class which stores an `ImageInstance`[1] object containing all the information about the slide including its width, height and identifier. To prevent anyone from loading the full image into memory, the implementation of the `np_image` raises a `NotImplementedError` exception.

In general, when the full image can be loaded into memory the default `Tile` class can be used. In this case, as this operation is impossible, a class `CytomineTile` was created to handle the tile image loading. Especially, the call to `np_image` triggers an HTTP request to the Cytomine server to fetch the corresponding image window. If the request fails (e.g. HTTP error) or returns an invalid result (e.g. returned image has an invalid size), a `TileExtractionError` is raised as advised in the documentation. The class `CytomineTileBuilder` was created to build `CytomineTile` objects.

In order to reduce the overall execution time of the workflow, it is essential not to execute two times a HTTP request for loading the same image window. To avoid this, the class `TileCache` was developed. It implements a simple caching policy using the local file system: when an image window is needed, the `TileCache` object first checks whether this image was already downloaded and stored on the disk. If that is the case, the request execution is bypassed and the image is loaded from the file. Otherwise, the image is fetched by calling the `np_image` method of the underlying tile and stored on the disk before being handed back to the caller. The class also provides methods for adding an alpha mask to the returned image.

Some additional classes were developed to handle the addition of an alpha mask to an image window (class `CytomineMaskedWindow`) and to a tile (classes `CytomineMaskedTile` and `CytomineMaskedTileBuilder`). This feature is needed

[1] The `ImageInstance` class is defined in the Cytomine Python client

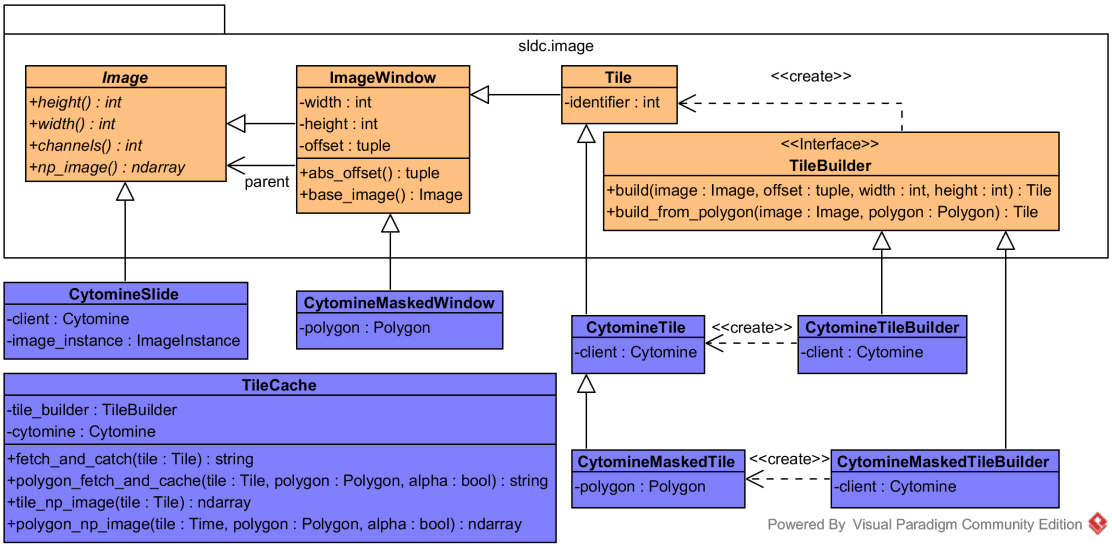


Figure 4.10: UML diagram - Cytomine image representation

for the pattern segmentation procedure which assumes that an alpha mask indicating the position of the pattern is passed with the numpy array. To avoid storing an image representing the alpha mask into memory, the mask is represented by a polygon.

The UML diagram of the package containing the image-related classes is shown in Figure 4.10.

4.3.2 Classifier

In the context of the the thyroid problem, all classification tasks are performed using the random subwindows algorithm [MGW16]. Especially, a Python implementation called Pyxit taken from Cytomine [Mar+16] was used. The central class of this implementation is the `PyxitClassifier` class which provides a scikit-learn like interface to the algorithm (i.e. the methods `fit`, `transform`, `predict`,...). In order to use this class within the framework, a class `PyxitClassifierAdapter` was developed. The `predict_batch` method is implemented as follows.

First, the crops of the polygons passed to the method are fetched and stored on the disk using a `TileCache` (thanks to the cache, the HTTP request is only executed the first time the crops are requested). Because there can be a lot of polygons, the fetching of the crops is parallelized and the number of available processes can be specified at the construction of the `PyxitClassifierAdapter` object. In order to reduce the serialization overhead, each available process is passed a set of polygons (see Section 3.2.4.10). If some crops cannot be fetched for whatever reason, the corresponding polygons are associated a class `None` and a probability 0. Moreover the user is notified with the logger about the crops that couldn't be fetched.

As Pyxit works with images stored on the disk, a list containing the filepath of the images to classify is generated and passed to the `_predict` method. This method implements the generation of the classification labels and probabilities. If a SVM classifier was provided at construction of the `PyxitClassifierAdapter` object, then

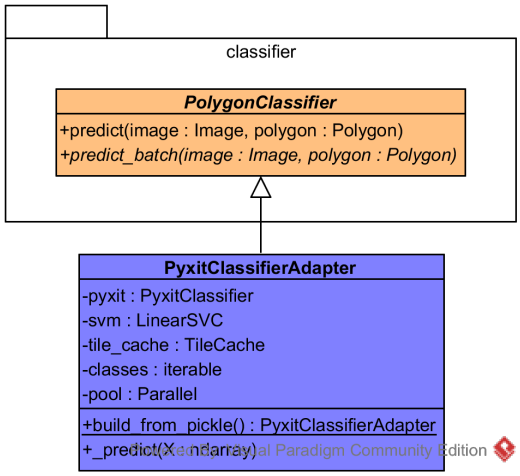


Figure 4.11: UML diagram - Classifier

the ET-FL variant of the random subwindows algorithm is used. That is, the Pyxit classifier is used to generate the features that are passed to the SVM classifier for predicting the labels. Otherwise, the variant that uses the extremely randomized trees as direct classifier is used. Finally, the `predict_batch` method aggregates the results returned by the `_predict` method with the labels and probabilities generated for the polygons of which the crops couldn't be fetched and return those to the caller.

This class also features a static method for constructing a `PyxitClassifierAdapter` object from a serialized Pyxit model. Especially, the method deserializes the Pyxit classifier as well as the SVM classifier if one is provided and passes them to the adapter's constructor. This method also sets the number of process to use for fetching the crops. If the number of available process is less than five, then all of them are used to fetch the crops. Otherwise, five processes are used at most in order to avoid overloading the Cytomine server.

The UML diagram of the `PyxitClassifierAdapter` class is shown in Figure 4.11.

4.3.3 Dispatching rules

As explained in Section 4.2.2, the chosen dispatching method relies on a classifier which predicts the dispatching index. Especially, the classifier was built to predict the label 0 for *cell*, 1 for *pattern* and 2 for *other*.

To take advantage of the features provided by the class `PyxitClassifierAdapter` (i.e. caching, parallel fetching,...), it was reused and encapsulated in two classes extending `DispatchingRule`. The implementation of the rules' `evaluate_batch` method is therefore straightforward. It first calls the `predict_batch` method of the classifier adapter object and then generates a list of boolean values according to the returned classification labels. For the first rule, `CellRule`, `True` is associated to polygons of which the returned label is 0. For the second, `AggregateRule`, `True` is associated with polygons of which the predicted label is 1. After a first run of the algorithm, it appeared that despite the dispatch classifier, a lot small artefacts were dispatched to the cells and patterns classifiers. Therefore, each rule was added a

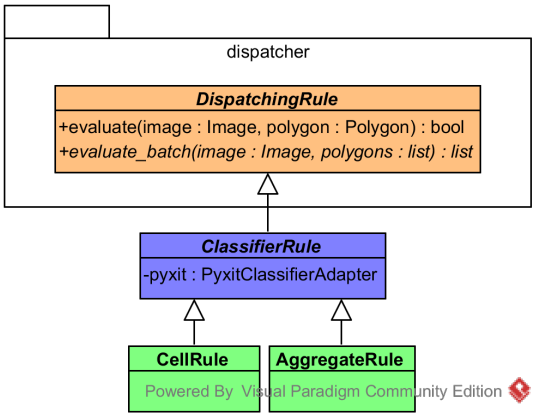


Figure 4.12: UML diagram - Thyroid workflow dispatching rules

filtering procedure which excludes all objects of which the area is less than a given value.

Unfortunately, the fact that the dispatching is implemented with two rules implies that the polygons corresponding to patterns and other objects are classified twice. Indeed, because of the dispatching structure imposed by the framework, the polygons that are not matched by the first rule are evaluated by the second. This could be avoided by refactoring the framework or by implementing another dispatching strategy.

The UML diagram containing the dispatching rule classes is shown in Figure 4.12.

4.3.4 Segmentation

The segmentation procedures were implemented in two classes, `SlideSegmenter` and `AggregateSegmenter`. Both implementation were taken from Antoine Deblire's source code. Whereas the slide segmentation could be used almost directly without modification, the recovered aggregate segmentation procedure did not work. It was therefore re-implemented following the explanations provided in the master thesis as well as the few comments present in the source code. After few tests, it appeared that both segmentation procedures were rather slow because of the color deconvolution. Especially, to execute the color deconvolution on a 4 mega-pixels image yielded more than 2 seconds execution time. A first optimization pass was done over the function in order to reduce its execution time by a factor two. The UML diagram containing the segmenter classes is shown in Figure 4.13.

4.3.5 Chaining

In order to implement the re-segmentation, the chaining package must be used. First, an image provider must be defined to generate the `CytomineSlide` objects to be processed. This logic is implemented in the `SlideProvider` class. Then, the selection of the objects to be processed by the second workflow must be defined as a `WorkflowExecutor`. Especially, the class `AggregateWorkflowExecutor` was imple-

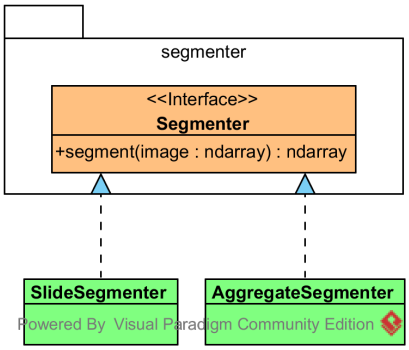


Figure 4.13: UML diagram - Segmenter classes

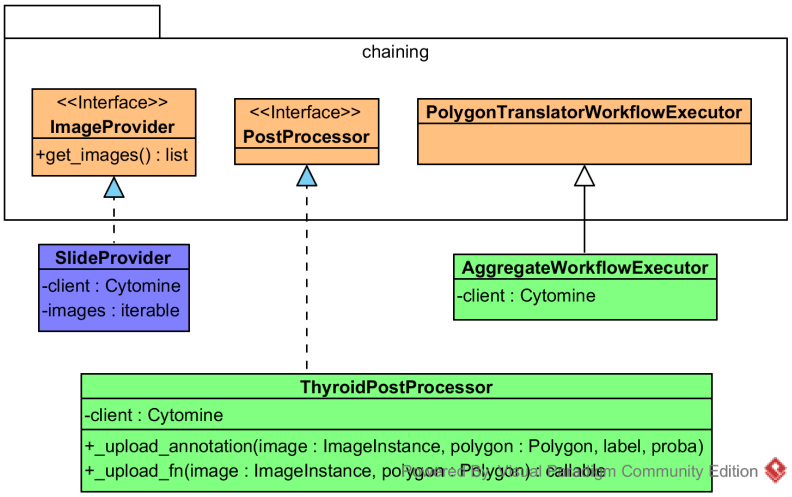


Figure 4.14: UML diagram - Chaining classes

mented to fulfill this role. The class defines the method `get_windows` which implements the generation of `CytomineMaskedWindow` objects to be processed by the second workflow. Moreover, it extends the class `PolygonTranslatorWorkflowExecutor` because the polygons generated by this phase needs to be translated back into the full image reference system. The final component to be defined is the post processor which is passed all the detected objects and their classes. In the context of the thyroid problem, the post processor should upload the generated polygons and classes as annotations on the Cytomine platform. This logic is implemented in the `post_process` method of the `ThyroidPostProcessor` class. Unfortunately, the Cytomine API does not provide any request for uploading annotations by batches and each annotation has to be added with two HTTP requests: one for uploading the geometry and another for uploading the predicted class and associated probability. To avoid waiting for each request to terminate before sending another one, the process was parallelized. The UML diagram containing the chaining classes is shown in Figure 4.14.

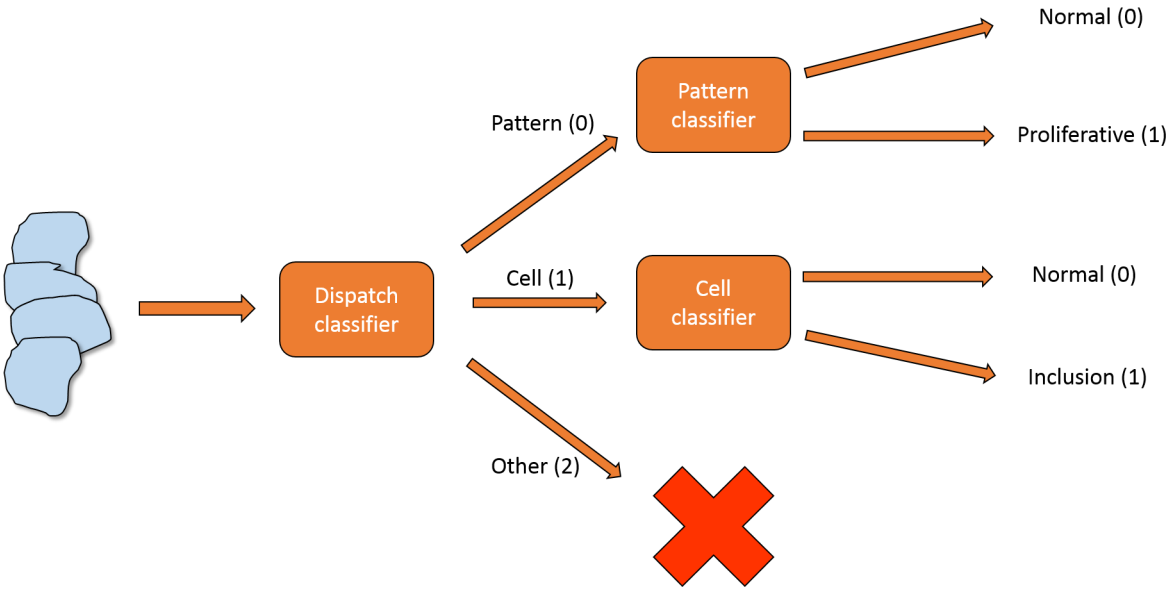


Figure 4.15: Classifiers' roles summary

4.4 Performance analysis

4.4.1 Classification models

As explained in Section 4.2, three classifiers are used by the workflow. The first detects whether an object is a cell, a pattern or another type of object. The second classifies patterns as proliferative or non-proliferative and the last detects whether cells contain an inclusion or not. The roles of these classifiers are illustrated in Figure 4.15. Information about them including the terms associated with the classes, the performances of the models,... are given in Sections 4.4.1.3, 4.4.1.4, 4.4.1.5 and 4.4.1.6. Before exploring the classifiers, the methodology followed for assessing the models is presented in Sections 4.4.1.1 and 4.4.1.2.

4.4.1.1 Test set and metrics

In order to assess a supervised learning model, a classical approach is to split the objects of the input set into a *learning set* on which the model is learned and *test set* on which the error of the model is evaluated. The error extracted using this procedure is called the *generalization error* and is a measure of how accurate the predictions of the model will be on unseen data. Typically, when the number of available objects in the input set is high, a valid splitting strategy consists in leaving approximately 70 % of the objects in the learning set. The motivation of this proportion is twofold. On the one hand, the learning process have enough data to build relevant models. On the other hand, the test set contains enough samples to make the assessment statistically relevant. However, the proportion is not the only factor that needs to be taken into account. Especially, when the input data is gathered from several sources, the split should be done to avoid overfitting the idiosyncracies of those sources. This can be done by placing the data generated from

some sources in the test set and the others into the learning set. Finally, for the assessment to be relevant, both the test and the learning sets should contain instances of all classes. This can be achieved by keeping the target variable's distributions in the learning and test sets close to that of the input set.

The first task performed when it came to build the assessment procedure was therefore to split the input data, namely the experts' annotations, into a test set and a learning set. This was done following the guidelines presented above. Especially, the test set is composed of images selected so that the proportion of annotations it contains is approximately 30 % and the distribution of the various terms of the ontology is close to their overall distribution in input set. Obviously, satisfying all those constraints at once is impossible given the discrete nature of the problem:

- the distributions can only be affected by moving an image into the test set or out of it
- each image typically contains between 3 and 10 different terms in different quantity
- some terms are contained in very few images. It is particularly true for the *Other* subcategory. For instance, the term "*Macrophage*" is contained in three images only

The final split was performed manually. Indeed, due the terms distribution across the images, a random generation was likely to yield a test set in which some terms were missing. The construction process was rather simple: images were taken out from the learning set one after another. When moving an image induced a major imbalance, it was put back into the learning set and another image was taken out instead until an acceptable distribution was reached. The final terms distribution in the test set and learning set is shown in Table 4.2. It can be seen in this table that some terms are slightly imbalanced (e.g. normal cells or artifacts). This is due to the fact that modifying the split to balance those would have broken the balance of more important terms such as cells with inclusion or proliferative patterns.

Now that the test set is built, the metrics that will be used for assessing the models must be defined. In classification, the most common is the *accuracy* which is the proportion of correctly classified objects. In the particular case of binary classification (the target is either positive or negative), two common metrics are:

- **recall**: the number of true positive over the number of positive. Intuitively, it is the ability of the classifier to find the positive objects.
- **precision**: the number of true positive over the number of predicted positive. Intuitively, it is the ability of the classifier not to classify positive objects as negative.

All the previous metrics can be computed based the confusion matrix, a square matrix of order N where N is the number of classification labels. Its element m_{ij} contains the number of objects that are actually associated the i^{th} label but were predicted the j^{th} one by a model. In order to have several indicators of the models performances, all the metrics mentioned above were used.

		Class count and proportions						LS/TS prop.	
		Input set		LS		TS		LS	TS
Terms		Count	%	Count	%	Count	%	%	%
Cell	NOS	874	14.76	567	13.85	307	16.81	64.87	35.13
	Normal	954	16.11	548	13.38	406	22.23	57.44	42.56
	Pseudo-inclusion	212	3.58	160	3.91	52	2.85	75.47	24.53
	Ground glass	13	0.22	8	0.20	5	0.27	61.54	38.46
	Grooves	194	3.28	144	3.52	50	2.74	74.23	25.77
	Inclusion	738	12.46	522	12.75	216	11.83	70.73	29.27
Pattern	Normal	798	13.48	584	14.26	214	11.72	73.18	26.82
	Prolif.	761	12.85	540	13.19	221	12.10	70.96	29.04
	Prolif. (minor)	300	5.07	225	5.49	75	4.11	75.00	25.00
Other	Macrophage	273	4.61	155	3.79	118	6.46	56.78	43.22
	Red blood	98	1.66	24	0.59	74	4.05	24.49	75.51
	Polynuclear	226	3.82	177	4.32	49	2.68	78.32	21.68
	Colloid	57	0.96	37	0.90	20	1.10	64.91	35.09
	Artefact	286	4.83	281	6.86	5	0.27	98.25	1.75
	Background	137	2.31	123	3.00	14	0.77	89.78	10.22
	Total	**5921**	100	**4095**	100	**1826**	100	**70.49**	**29.51**
Images		61		43		18			

Table 4.2: Terms distribution in the learning set and test set.

4.4.1.2 Cross validation and model selection

The random subwindows algorithm has several parameters that can be tuned to improve the model performances. It includes the parameters of the underlying classifier such as the minimum number of objects required to split a node or the maximum number of features to evaluate when looking for the best split in the decision tree algorithm. The algorithm has also proper parameters such as the minimum and maximum sizes of the windows to extract or the colorspace into which the windows must be converted before being passed to the underlying classifier. The complete list of parameters is given in Table 4.3.

In order to maximize the performances of the produced models, a tuning procedure was implemented to extract the best combination of parameters. The procedure creates a set of models using all the possible combinations of parameters values provided by the user. Each model is then assessed using cross-validation and the best parameters are returned. Similarly to model assessment, the fact that annotations come from several images should be taken into account to avoid overfitting. Especially, the performance of a model were assessed using a cross-validation strategy called *leave one image out*. Given a learning set containing N images, each image is taken out from the learning set in turn and the model is learned on the $N-1$ remaining images. The performance score of the model is then computed on the taken out image. This process generates N scores which are averaged and the resulting score is associated to the model. This process is illustrated in 4.16.

Finally, the full assessment procedure consists in splitting the dataset into a learning set and a test set. Then, the best parameters are determined by cross-validation on the learning set, a model is learned on the whole learning set with

Name	Classifier	Description
`pyxit_min_size`	Pyxit	Minimum size proportion of the windows to extract (relative to the full image size).
`pyxit_max_size`	Pyxit	Maximum size proportion of the windows to extract (relative to the full image size).
`colorspace`	Pyxit	Colorspace into which the windows must be converted before being passed to the underlying classifier. Available colorspaces are HSV and normalized RGB
`min_sample_split`	Extra-trees	Minimum number of objects required to split a node
`max_features`	Extra-trees	Maximum number of features to evaluate when looking for the best split in the decision tree algorithm
`C`	SVM	SVM penalty parameter. Only available when the ET-FL variant of the random subwindows algorithm is used.

Table 4.3: Random subwindows algorithm parameters to tune

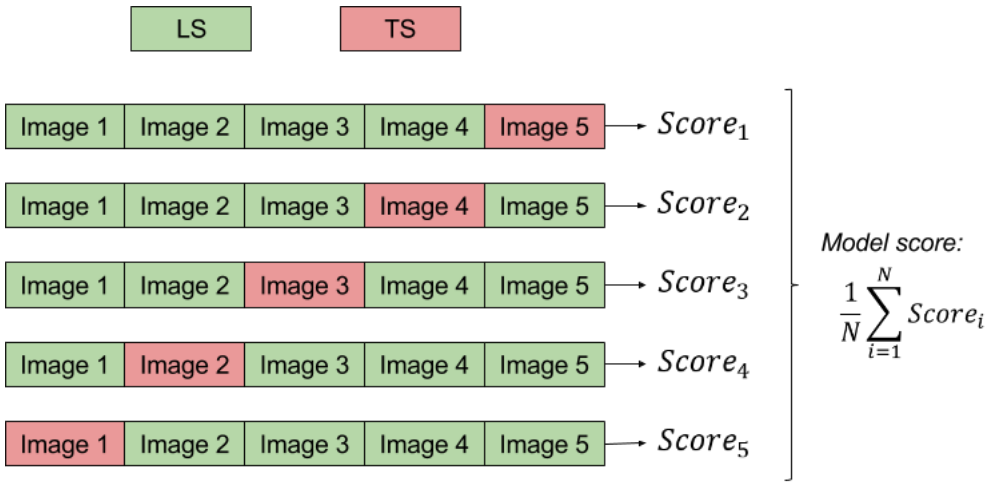


Figure 4.16: Leave one image out cross-validation strategy (number of images $N = 5$)

the best parameters and this model is assessed on the test set using the metrics presented in 4.4.1.1.

4.4.1.3 General comments about classifiers

For each required classifier, both variants of the random subwindows algorithm were evaluated: the first variant where the extra-trees are used as direct classifier (*ET-DIC*), and the second where they are used as features learner to be fed to a SVM classifier (*ET-FL*). Moreover, both variants were evaluated on a first learning set containing only experts' annotations and on a second which was augmented with some reviewed annotations (see Section 2.2.1.1 for reviewing). Especially, the latter annotations were generated by an execution the workflow on an image from the learning set and were reviewed on Cytomine in an attempt to improve the classifiers performances.

As far as the parameters are concerned, a first cross-validation procedure was applied to narrow the search space. Especially, this was applied for finding the ranges in which the window minimum and maximum sizes should be taken. As explained in [MGW16], small subwindows yield better results on images containing highly repeatable patterns (e.g. architectural patterns) because they allow to capture fine details. In contrast, larger windows works best on images in which the shape is prevailing (e.g. cells). This intuition was confirmed by a first cross-validation procedure as the best models for classifying cells were built using large windows while small windows yielded better results for classifying patterns.

For the ET-FL variant, the parameter `min_sample_split` was not tuned. Indeed, as suggested in [MGW16], it could be set to $\frac{W}{1000}$ where W is the number of subwindows in the learning set. Particularly, this value prevents the extra-trees to learn features that are too specific and also reduces the execution time compared to that of a model for which the parameter was set to 1. Indeed, the former model contains less leaf nodes which reduces the size of the features vector passed to the SVM classifier. As far as the ET-DIC variant is concerned, the same parameters was tuned with the following set of values: $\left\{1, \frac{W}{1000}, \frac{W}{100}, \frac{W}{50}, \frac{W}{20}\right\}$.

The parameter `max_features` was tuned whatever the variant and the same four values were always provided: $\left\{1, \sqrt{M}, \frac{M}{2}, M\right\}$ where M is the number of features passed to the underlying extra-trees classifier. As the dimensions of the resized windows are 16×16, the number of features M equals 768 in this case. Those four values were chosen to span over the range $[1, 768]$ and $\sqrt{M}$ is the default value suggested in [GEW06].

4.4.1.4 Pattern classifier

The pattern classifier is a binary model which predicts whether a pattern is proliferative or not. The correspondence between the output classes and the terms of the ontology is rather straightforward and is the following:

- Output **positive** (class 1):
  - *proliferative architectural pattern*

(a) Experts' annotations

	Prolif.		Normal		Total	
LS	765	(56.71%)	584	(43.29%)	1349	(72.57%)
TS	296	(58.04%)	214	(41.96%)	510	(27.43%)
Total	1061	(57.07%)	798	(42.93%)		

(b) Reviewed

	Prolif.		Normal		Total	
LS	908	(60.09%)	603	(39.91%)	1511	(74.76%)
TS	296	(58.04%)	214	(41.96%)	510	(25.24%)
Total	1061	(57.07%)	798	(42.93%)		

Table 4.4: Pattern classifier. Dataset size.

		ET-DIC	ET-DIC (r)	ET-FL	ET-FL (r)
LS	accuracy	0.8285	0.8224	0.8849	0.886
TS	accuracy	0.8664	0.8468	0.8664	0.8625
	recall	0.9662	0.9628	0.9628	0.9493
	precision	0.8314	0.8097	0.83333	0.8363

Table 4.5: Pattern classifier. Best model's performance.

  - *proliferative architectural pattern (minor sign)*

- Output **negative** (class 0):
  - *normal follicular architectural pattern*

The only terms used are the ones from the subcategory *Pattern* as all the objects to be classified by this model are expected to be patterns. Indeed, other objects have normally been filtered by the dispatch classifier. Luckily, this distribution of terms yields some rather balanced learning set and test sets (see Table 4.4). As patterns are usually large objects, only small window sizes were evaluated during cross validation. All the evaluated parameters are given in Appendix D while the one which yielded the better models are given in Table D.6. It seems that the HSV colorspace is particularly well suited to the pattern classification problem.

The best models' performances are given in Table 4.5 and D.7. All models perform relatively well. Especially, almost all models exhibit an impressive recall of 96 % except for the ET-FL variant with reviewed annotations of which the recall drops to 94 %.

As the performances are comparable, the model selected to be used for executing the workflow was the ET-DIC variant without reviewed annotations. Especially, it was preferred because it executes faster than the ET-FL variant (see in Section 4.4.2.1) and also because the model can generate class probabilities.

4.4.1.5 Cell classifier

The cell classifier is a binary model which predicts whether a cell contains an inclusion or not. As there are more terms related to cells, the correspondence between

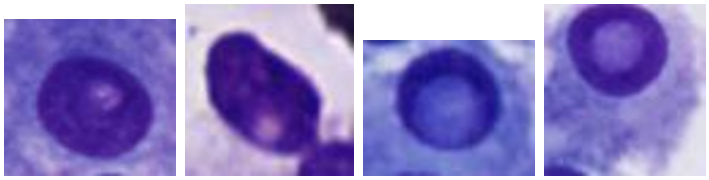
the terms of the ontology and the classification labels is slightly more complex than for the pattern classifier:

- Output *positive* (class 1):
  - *cell with inclusion*
- Output *negative* (class 0):
  - *cell with NOS*
  - *pseudo-inclusion*
  - *ground glass nuclei*
  - *nuclear grooves*
  - *normal cell*
  - *red blood cell*
  - *polynuclear*

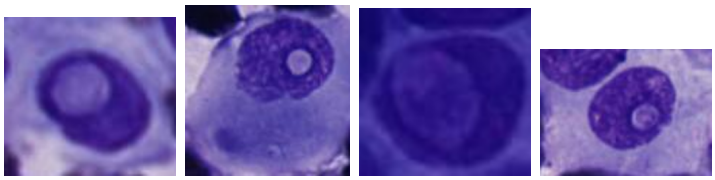
While for the pattern classifer, only objects from the *Pattern* subcategory were used, the cell classifier is trained with two terms from the *Other* subcategory. This was done because those two terms (i.e. red blood cell and polynuclear) corresponds to objects that can be mistaken with actual cells. Especially, if those objects are dispatched to the cell classifier, it has to be able to associate to them the negative class. Choosing this mapping has nevertheless the drawback of emphasizing the class imbalance. Table 4.6 shows that only 25 % of the objects input set are cells with inclusion. For this classifier, the windows sizes were chosen relatively big (see Table D.3) as the cells are small objects with few details and no repeatable patterns.

The performance expectations for this classifier are the following: the consequences of the presence of cells with inclusion is such that those should not be missed. Especially, recall should be as high as possible. Moreover, as the cells found by the workflow would in practice be reviewed by physicians (before making a diagnosis), the number of false positive should also kept as small as possible. This would be expressed by a high precision. The best models' performances are given in Table 4.7 and D.8. Those results show that the models do not meet the expectations. Indeed, whereas they exhibit a relatively good precision, the recall is far from being acceptable which indicates that the models fail at detecting cells with inclusion. The performances are particularly bad and worse than random guessing when the model is built using the ET-DIC variant as only 13 % of the cells with inclusion are correctly labelled as such. The results are better with the ET-FL variant as the recall rises to 50 % on the dataset containing the reviewed annotations. While the increase is spectacular, the model still performs very poorly as random guessing would yield approximately the same recall.

Those poor performances might be explained by two elements. The first is the similarity between the cells containing an inclusion and some cells of the negative class. For instance, the cells with pseudo inclusion are very similar to cells with inclusion as shown in Figure 4.17. The second is the class imbalance in the dataset. It appears that adding the reviewed annotations in the learning has improved the recall



(a) Pseudo-inclusion



(b) Inclusion

Figure 4.17: Similarity between pseudo-inclusion and cells with inclusion

(a) Experts' annotations

	Inclusion		Normal		Total	
LS	567	(26.37%)	1583	(73.63%)	2150	(64.97%)
TS	307	(26.49%)	852	(73.51%)	1159	(35.03%)
Total	874	(26.41%)	2435	(73.59%)		

(b) Reviewed

	Inclusion		Normal		Total	
LS	571	(18.26%)	2556	(81.74%)	3127	(72.96%)
TS	307	(26.49%)	852	(73.51%)	1159	(27.04%)
Total	878	(20.49%)	3408	(79.51%)		

Table 4.6: Cell classifier. Dataset size.

with the ET-FL variant. Therefore, increasing the amount of data by performing more reviewing might be a solution to improve the classifier.

Because it exhibits the best recall score, the model ET-FL with reviewed annotations should be used within the workflow.

4.4.1.6 Dispatching classifier

The dispatching classifier is a ternary model which predicts whether an object is a cell, a pattern or another type of object. The correspondence between the classes and the terms of the ontology is the following:

		ET-DIC	ET-DIC (r)	ET-FL	ET-FL (r)
LS	accuracy	0.7645	0.7613	0.7813	0.7873
TS	accuracy	0.8333	0.8351	0.8696	0.8523
	recall	0.1302	0.1349	0.4512	0.4930
	precision	0.8235	0.8529	0.7462	0.6310

Table 4.7: Cell classifier. Best model's performance.

- Output *pattern* (class 0):
  - *proliferative architectural pattern*
  - *proliferative architectural pattern (minor sign)*
  - *normal follicular architectural pattern*
- Output *cell* (class 1):
  - *cell with NOS*
  - *pseudo-inclusion*
  - *ground glass nuclei*
  - *nuclear grooves*
  - *normal cell*
  - *red blood cell*
  - *cell with inclusion*
- Output *other* (class 2):
  - *background*
  - *artefact*
  - *macrophage*
  - *polynuclear*
  - *colloid*

Mostly, the correspondence is expected. The only peculiarity is the association of the term *Red blood cell* with the second output. Whereas this term is in the *Other* subcategory, red blood cells look quite much like cells with inclusion. So it seemed a good idea to bias the model so that it redirects those to the cell classifier in order not to miss cells with inclusion.

As shown in Table 4.8, the terms distribution induces an imbalance. The third class is particularly under-represented. However, the imbalance is not as critical as for the cell classifier. As far as the windows dimensions are concerned, the first cross-validation revealed that using both small and large ones at the same time yielded better results (see Table D.2 for the dimensions).

The best models' performances are given in Table 4.9 and D.9. A first observation is that all models seem to be relatively good at dispatching cells and patterns. However, they fail at dispatching other objects. This is particularly true for the ET-DIC variant without reviewed annotations which classifies all the other objects as cells or patterns. The addition of reviewed annotations brings a slight improvement as 4 % of the other objects are classified as such.

The ET-FL variant brings a non-negligible improvement as for the classification of other objects as 46 % of them are correctly classified. This model also classifies the patterns more precisely as only 4 % are misclassified against 11 % with the ET-DIC variant. Those improvements come at the the cost of a slight degradation of cell classification as 5 % of them are classified as other objects. The addition of

(a) Experts' annotations

	Pattern		Cell		Other		Total	
LS	1349	(32.94%)	1973	(48.18%)	773	(18.88%)	4095	(69.16%)
TS	510	(27.93%)	1110	(60.79%)	206	(11.28%)	2435	(30.84%)
Total	1859	(31.4%)	3083	(52.07%)	979	(0.1653%)		

(b) Reviewed

	Pattern		Cell		Other		Total	
LS	1511	(25.94%)	2950	(50.64%)	1364	(23.42%)	4095	(76.13%)
TS	510	(27.93%)	1110	(60.79%)	206	(11.28%)	2435	(23.87%)
Total	2021	(26.41%)	4060	(53.06%)	1570	(20.52%)		

Table 4.8: Dispatch classifier. Dataset size.

	ET-DIC	ET-DIC (r)	ET-FL	ET-FL (r)
LS	0.8352	0.7428	0.8991	
TS	0.8498	0.8504	0.8910	

Table 4.9: Dispatch classifier. Best model's performance (the metrics is *accuracy*).

the reviewed annotations (which contain a majority of other objects, see Table 4.8) increases the proportion of correctly classified other objects which reaches almost 60 %. Again, this comes at the cost increasing the misclassification rate for the other classes which reaches approximately 8 % for both of them.

As it exhibits the best accuracy, the ET-FL variant without reviewed annotations should be used within the workflow to dispatch cells.

4.4.2 Execution times

In order to assess the time performances of the workflow, several tests were performed. The tables containing the execution times can be found in Appendix E. Each run[2] was assigned a number which can be found in the tables. Three series of tests were performed:

1. The first consisted in launching the first part of the workflow (first segmentation) several times over the same image by varying the tile dimensions and the number of available processes. Those tests provide a first illustration of the performances of the workflow as well as information about the efficiency of the parallalelization. The results are presented in Section 4.4.2.1.

2. The second consisted in launching the workflow on images from the test set. This allows to assess how the workflow performs on typical images containing possibly a lot of objects. The results are presented in Section 4.4.2.2.

3. The last test was performed in order to assess the pattern segmentation. The results are presented in Section 4.4.2.3.

[2]A "run" is an execution of the workflow.

4.4.2.1 Number of jobs and tile dimensions

The resulting execution times computed for this test series are given in Table E.1. Each run was associated a number which is given in the table. More details about the rows of Table E.1 can be found in Appendix E. In order to interpret correctly the execution times, it is important to kown that all runs did not have to download the tiles and crops from the server as they benefited from the cached files saved by the previous executions:

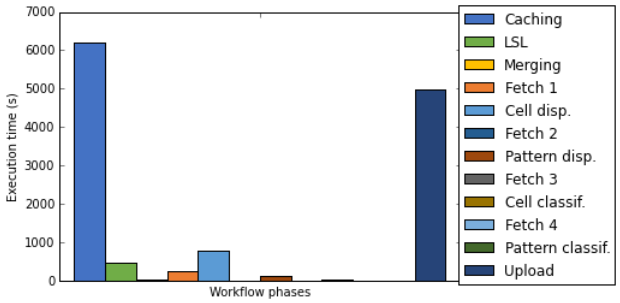
- **Run 1**: it was launched first and therefore had to download all tiles of dimensions 512×512 as well as the crops of the detected polygons.
- **Run 4**: it was launched after run 1 and therefore benefited from the cached crops files. Still, it had to download all tiles of dimensions 1024×1024.

In Figure 4.18 are shown execution times of the various phases of the workflow on runs 1 and 2. For each run, two charts are given: all the phases are displayed on the first and *Caching* and *Upload* have been removed from the second. Those figures provide a good illustration of the workflow performances. First, it can be seen that the time required for fetching and caching the tiles dominates all the others except *Upload* (see Figure 4.18(a)). For run 1, more than one hour and a half was needed to download 29900 tiles while other steps combined (except *Upload*) lasted approximately 27 minutes. For run 4, approximately one hour was needed to fetch 7353 tiles while all other steps combined lasted the same time as for run 1 (see Table E.1).

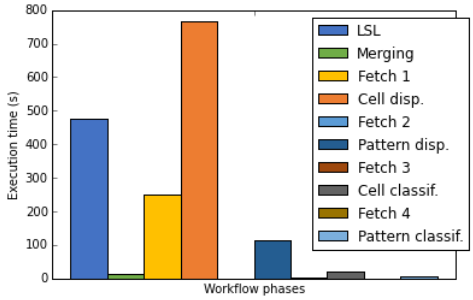
As far as the *Upload* phase is concerned, its execution time for runs 1 and 4 is more than ten times greater than for the other runs. Those abnormal execution times can be explained by the fact that the Cytomine server was probably busy at the time those runs were executed (indeed run 1 and 4 were executed one after another) and the server was therefore slower at responding to incoming requests. As for the other runs, the execution time is quite stable and is approximately 6 minutes (for uploading 6600 annotations). This can be seen in Figure 4.18(c) where the execution time of *Upload* is comparable to that of the other steps.

The effects of the implemented crop caching policy can be seen on both Figures 4.18(d) and 4.18(b). In the first figure, one can see that the crops were downloaded during *Fetch 1* and that the subsequent fetching steps were almost instantaneous. The fact that run 2 benefited from the caching manifests as the instantaneous execution of the step *Fetch 1* in the second figure.

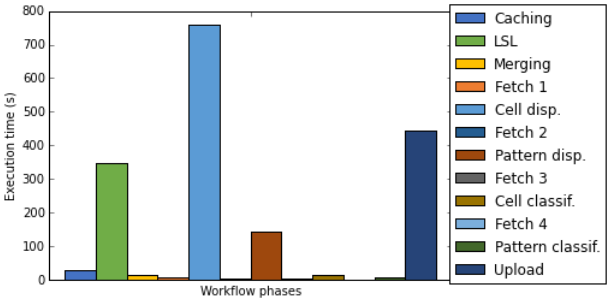
With regard to the other phases, it can be seen in Figures 4.18(b), 4.18(c) and 4.18(b) that step *Cell disp.* dominates the overall execution time of the workflow. This can be explained by the usage of the ET-FL variant of the random subwindows algorithm. Indeed, this variant relies on a SVM classifier which does not support parallelization. Therefore, whatever the number of available processes, the dispatching is mostly executed on a single process. This observation also holds for the *Pattern disp.* phase. Nevertheless, this phase is always shorter as most of the objects are typically dispatched to the cell classifier (for image 728725, 70 % of the detected objects are dispatched as cells and only 20 % as patterns. See Table in E.1).



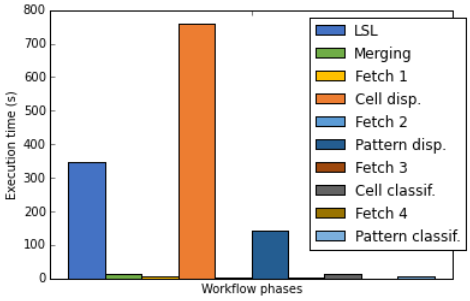
(a) Run 1. Fetch and cache the tiles before executing the workflow.



(b) Run 1. Without *Upload* and *Caching*.



(c) Run 2. Benefits from caching of run 1.



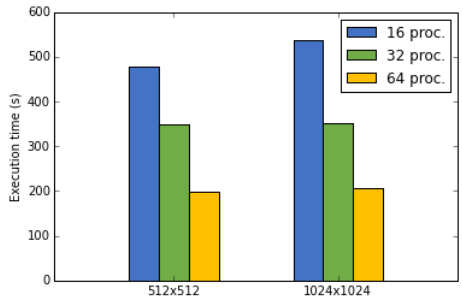
(d) Run 2. Without network phases.

Figure 4.18: Execution times of the workflow phases for run 1 and 2 (executed with tile dimensions of $512x512$ and respectively 16 and 32 processes).

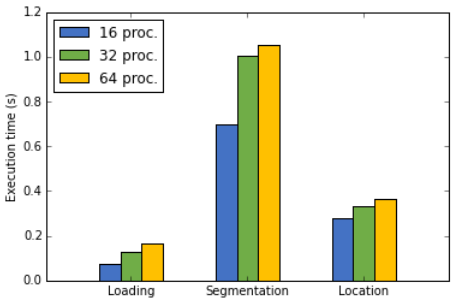
The second longest phase after *Cell disp.* is the tile processing, *LSL*. It lasts approximately 26 % of the overall execution time for runs 1 and 2. As explained in Section 3.2.4.10, this phase is parallelized by the *SLDC* framework. The gain provided by this parallelization is shown in Figure 4.19(a) where the overall execution time of *LSL* is compared for all the runs presented in Table E.1. Especially, one can see that increasing the number of process to 32 (starting from 16) makes *LSL* 1.4 faster with tiles of dimensions 512×512 and 1.5 times faster with tiles of dimensions 1024×1024. Moreover, quadrupling the number of process increases the speed by a factor 2.4 and 2.6 for tiles of dimensions 512×512 and 1024×1024 respectively. This increase is non-negligible and proves that parallelizing this step was worth it. However, as explained in Section 3.3.3, the parallelization can still be improved.

In Figure 4.19(b) is presented the execution times of the sub-phases of *LSL* averaged over all the processed tiles. The chart provides a surprising result. Indeed, it seems that the execution times of the sub-phases increase with the number of available processes. This phenomenon might be understandable for *Loading* which manipulates files. Indeed, if several processes try to open different files at the same time, the operating system might struggle to answer all those file opening requests at once. However, this cannot explain why *Segment* and *Location*, which are purely CPU-bound, experience the same phenomenon.

The impact of the tile dimensions on the execution times of the sub-phases of *LSL* is shown in Figure 4.20. The expectation is that all sub-phases are four times faster on the tile of dimensions 512×512 as those are four times smaller.



(a) Evolution of the execution times of the *LSL* phase when varying the number of available processors



(b) Evolution of the execution times per tile for each step of the *LSL* phase when varying the number of available processors (tile dimensions 1024×1024)

Figure 4.19: Parallelization of the Load Segment Locate (*LSL*) phase.

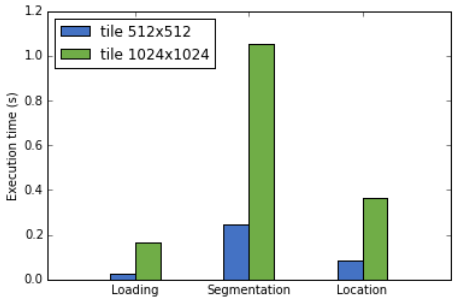


Figure 4.20: Evolution of the execution times per tile for each step of the *LSL* phase when the tile dimensions are changed (64 processes).

The results are slightly above the expectations for *Segment* and *Location* which are respectively 4.3 and 4.2 faster on the smaller tiles. As for *Loading*, it is surprisingly 6.4 times faster. However, no conclusion can be taken from this result as the speed up factor for *Loading* seems unstable. Indeed, it is 3.7 and 2.5 for 16 and 32 processes respectively.

4.4.2.2 First segmentation on the test set

4.4.2.3 Second segmentation on the test set

Chapter 5

Conclusion

Appendix A

Tile topology

As presented in Section 3.2.4.1, the tile topology objects associate unique increasing identifiers to tiles. Using this representation allows to reach a $\mathcal{O}(1)$ time complexity for all the methods of the class `TileTopology`. Indeed, the results produced by those methods can be computed using simple formulas. In the following formulas, i refers to a tile identifier:

- The number t_{row} of tiles on a row is given by:

$$t_{row} = \begin{cases} \left\lceil \dfrac{w - o_p}{w_m - o_p} \right\rceil & \text{, if } w > w_m \\ 1 & \text{, otherwise} \end{cases} \tag{A.1}$$

- The number t_{col} of tiles on a column is given by Equation A.1 applied to the image height h and maximum tile height h_m instead of w and w_m.
- The total number t of tiles in the tile topology is simply $t_{row} \times t_{col}$.
- The neighbour tiles identifiers can be obtained by performing subtractions and additions. For instance, for a tile which is not on the edge of the image, the identifiers of its left, top, right and bottom neighbours are respectively $i - 1$, $i - t_{row}$, $i + 1$, $i + t_{row}$.
- The tile offset $(t_{\text{off},x}, t_{\text{off},y})$ can be retrieved as follows:

$$t_{\text{off},x} = (t_{row} - o_p) \times [(i - 1) \mod t_{row}] \tag{A.2}$$

$$t_{\text{off},y} = (t_{col} - o_p) \times \left\lfloor \frac{i - 1}{t_{row}} \right\rfloor \tag{A.3}$$

Appendix B

Ontology

The ontology associated with the Thyroid project on Cytomine is the following:

1. Architectural patterns:
    - Normal follicular architectural pattern
    - Proliferative follicular architectural pattern
    - Proliferative follicular architectural pattern (minor sign)
2. Nuclear features:
    - Papillary cell NOS
    - Normal follicular cells
    - Normal follicular cell with pseudo-inclusion (artefact)
    - Papillary cell with ground glass nuclei
    - Papillary cell with nuclear grooves
    - Papillary cell with inclusion
3. Others:
    - Macrophages
    - Red blood cells
    - PN (polynuclear)
    - Colloid
    - Artefacts
    - Background

Appendix C

Random subwindows

Random subwindows [MGW16] is an image classification algorithm. The first step of the algorithm consists in transforming the N input images. This is done by extracting a set of N_w random subwindows from each image. A random subwindow is a square patch of random size extracted at a random position in an image. The extracted windows are then resized to a fixed size patch (w, h). Those transformation operations generates a dataset containing $N \times N_w$ objects and $w \times h$ attributes.

The second step consists in passing this dataset to a classifier which will actually predict the image's classification label from those subwindows. In [MGW16], two classification methods are proposed.

The first uses extremely randomized trees [GEW06] as direct classifier: that is, each window is predicted a label and the full image label is determined by a majority vote over the predicted classes of this image's windows.

The second variant uses extremely randomized trees as feature learner rather than a direct classifier and relies on a SVM classifier to produce the prediction. In this variant each image is represented as a vector of which the dimensionality equals the number of terminal nodes in the ensemble of randomized trees and where the i^{th} feature is the number of windows that reached the i^{th} leaf node of the forest divided by the total number of windows. This vector is then passed to the SVM classifier to predict the image classification label.

Appendix D

Cross validation

This chapter presents the parameters that were tuned by the cross-validation procedure presented in Section 4.4.1.2. Each classifier was built using the two variants of the random subwindows algorithm , ET-FL and ET-DIC, and for each variant, the model was learned on two different learning sets: one with the reviewed annotations and one without. The complete lists of parameters are given in Figure D.1 for the dispatching classifier, in Figure D.3 for the cell classifier and in Figure D.5 for the pattern classifier.

Parameters	ET-DIC	ET-DIC (r)	ET-FL	ET-FL (r)
`window size`	(0.3, 0.8), (0.3, 1.0), (0.5, 0.8), (0.5, 1.0)			
`colorspace`	HSV, normalized RGB			
`min_sample_split`	$\{1, 91, 906, 1812, 4530\}$	$\{1, 291, 2913 5825, 14563\}$	91	291
`max_features`	$\{1, 28, 384, 768\}$			
`C`	/	/	$\{0.1, 1\}$	

Table D.1: Dispatch classifier. Tuned parameters.

Parameters	ET-DIC	ET-DIC (r)	ET-FL	ET-FL (r)
pyxit_min_size	0.5	0.5	0.3	0.3
pyxit_max_size	0.8	0.8	0.8	0.8
colorspace	TRGB	TRGB	HSV	HSV
min_sample_split	1	1	91	232
max_features	1	1	1	1
C	/	/	1.0	1.0

Table D.2: Dispatch classifier. Best model's parameters.

Parameters	ET-DIC	ET-DIC (r)	ET-FL	ET-FL (r)
window size	(0.6, 0.7), (0.6, 0.8)			
colorspace	HSV, normalized RGB			
min_sample_split	$\{1, 108, 1075, 2150, 5375\}$	$\{1, 156, 1564 3127, 7818\}$	108	156
max_features	$\{1, 28, 384, 768\}$			
C	/	/	$\{0.1, 1\}$	

Table D.3: Cell classifier. Tuned parameters.

Parameters	ET-DIC	ET-DIC (r)	ET-FL	ET-FL (r)
pyxit_min_size	0.6	0.7	0.6	0.6
pyxit_max_size	0.8	0.8	0.8	0.8
colorspace	HSV	HSV	HSV	TRGB
min_sample_split	1	1	108	156
max_features	1	1	1	1
C	/	/	0.1	1.0

Table D.4: Cell classifier. Best model's parameters.

Parameters	ET-DIC	ET-DIC (r)	ET-FL	ET-FL (r)
window size	(0.2, 0.4), (0.2, 0.3)			
colorspace	HSV, normalized RGB			
min_sample_split	$\{1, 67, 675, 1349, 3373\}$	$\{1, 76, 756 1511, 3778\}$	91	291
max_features	$\{1, 28, 384, 768\}$			
C	/	/	$\{0.1, 1\}$	

Table D.5: Pattern classifier. Tuned parameters.

Parameters	ET-DIC	ET-DIC (r)	ET-FL	ET-FL (r)
pyxit_min_size	0.3	0.2	0.2	0.2
pyxit_max_size	0.4	0.4	0.4	0.4
colorspace	HSV	HSV	HSV	HSV
min_sample_split	1	1	67	76
max_features	1	1	1	1
C	/	/	0.1	1.0

Table D.6: Pattern classifier. Best model's parameters.

(a) ET-DIC

	Normal		Prolif.	
Normal	155	(30.45%)	58	(11.39%)
Prolif.	10	(1.96%)	286	(56.19%)

(b) ET-DIC (reviewed)

	Normal		Prolif.	
Normal	146	(28.68%)	67	(13.16%)
Prolif.	11	(2.16%)	287	(55.99%)

(c) ET-FL

	Normal		Prolif.	
Normal	156	(30.65%)	57	(11.20%)
Prolif.	11	(2.16%)	285	(55.99%)

(d) ET-FL (reviewed)

	Normal		Prolif.	
Normal	158	(31.04%)	55	(10.81%)
Prolif.	15	(2.95%)	281	(55.21%)

Table D.7: Pattern classifier. Confusion matrices.

(a) ET-DIC

	Normal		Prolif.	
Normal	937	(80.92%)	6	(0.52%)
Prolif.	187	(16.15%)	28	(2.42%)

(b) ET-DIC (reviewed)

	Normal		Prolif.	
Normal	938	(81.00%)	5	(0.43%)
Prolif.	186	(16.06%)	29	(2.50%)

(c) ET-FL

	Normal		Prolif.	
Normal	910	(78.58%)	33	(2.85%)
Prolif.	118	(10.19%)	97	(8.38%)

(d) ET-FL (reviewed)

	Normal		Prolif.	
Normal	881	(76.08%)	62	(5.35%)
Prolif.	109	(9.41%)	106	(9.15%)

Table D.8: Cell classifier. Confusion matrices.

(a) ET-DIC

	Pattern		Cell		Other	
Pattern	453	(24.82%)	57	(3.12%)	0	(0.0%)
Cell	11	(0.60%)	1098	(60.16%)	0	(0.0%)
Other	34	(1.86%)	172	(9.42%)	0	(0.0%)

(b) ET-DIC (reviewed)

	Pattern		Cell		Other	
Pattern	460	(25.21%)	41	(2.25%)	5	(0.49%)
Cell	22	(1.21%)	1084	(59.40%)	3	(0.16%)
Other	36	(1.97%)	162	(8.88%)	8	(0.44%)

(c) ET-FL

	Pattern		Cell		Other	
Pattern	492	(26.96%)	12	(0.66%)	6	(0.33%)
Cell	12	(0.66%)	1045	(57.26%)	52	(2.85%)
Other	30	(1.64%)	87	(4.77%)	89	(4.88%)

(d) ET-FL (reviewed)

	Pattern		Cell		Other	
Pattern	468	(25.64%)	16	(0.88%)	26	(1.42%)
Cell	29	(1.59%)	1018	(55.78%)	62	(3.40%)
Other	23	(1.26%)	63	(3.45%)	120	(6.58%)

Table D.9: Dispatch classifier. Confusion matrices.

Appendix E

Execution times

In Tables E.1, ..., are given detailed execution times for executions of the workflow on several images. All execution times are given in seconds. More details about the fields of those tables are given hereafter:

1. **Run information**: global information about the run
    - *Run number*: a number associated with the execution in order to ease referencing those runs in the thesis
    - *Image width and height*: width and height (in pixels) of the image processed by the run
    - *Tile width and height*: width and height (in pixels) of the tiles used by the tile topology to break down the images in smaller chunks
    - *Tiles*: number of tiles containing in the topology
    - *Jobs*: number of processed assigned to the execution
    - *RAM*: maximum amount of memory available for the run to execute
2. **Polygons**: information about the polygons found by the run
    - *Found*: number of polygons found across all tiles
    - *Merged*: number of polygons resulting from the merging phase
    - *Cell*: number of polygons dispatched to the cell classifier
    - *Pattern*: number of polygons dispatched to the pattern classifier
    - *Dispatched*: total number of dispatched polygons
3. **L-S-L**: execution times of the **L**oad-**S**egment-**L**ocate phase. This phase is parallelized.
    - *Loading*: total amount of time for loading tiles into memory (on separate processes)
    - *Segment*: total amount of time for segmenting the tiles (on separate processes)
    - *Location*: total amount of time for locating polygons in segmented tiles (on separate processes)

- *Overall*: actual amount of time for processing all the tiles (wall-clock time)

4. **Dispatching**: execution times of the dispatching phase

   - *Cell model*: variant of the random subwindows algorithm used for dispatching polygons
   - *Fetch 1*: times needed for fetching the crops of the polygons to dispatch from the Cytomine server
   - *Cells*: amount of time needed for finding whether the polygons should be dispatched to the cell classifier or not
   - *Fetch 2*: time needed for fetching the crops of the polygons to dispatch to the pattern classifier. Normally, it should always be small as all the crops have already been downloaded and cached by the*Fetch 1* step
   - *Patterns*: amount of time for finding whether the polygons should be dispatched to the pattern classifier or not
   - *Overall*: total amount of time for dispatching the polygons

5. **Classification**: execution times of the classification phase

   - *Fetch 3*: times needed for fetching the crops of the polygons to be processed by the cell classifier. As for *Fetch 2*, those times should be low.
   - *Cells*: amount of time for classifying the cells
   - *Fetch 4*: times needed for fetching the crops of the polygons to be processed by the cell classifier. As for *Fetch 2*, those times should be low.
   - *Patterns*: amount of time for classifying patterns
   - *Overall*: total amount of time for classifying the dispatched polygons

6. **Net.**: for *Network*, time spent for sending network requests and waiting for responses

   - *Caching*: amount of time needed for fetching and caching the tiles of the topology
   - *Upload*: amount of time needed for uploading the dispatched polygons to the Cytomine server

7. **Total**:

   - *Not net. 1*: total execution time of the run from which was deduced the *Caching* and *Upload* execution times.
   - *Not net. 2*: total execution time of the run from which was deduced the *Caching*, *Upload*, *Fetch 1*, *Fetch 2*, *Fetch 3* and *Fetch 4* execution times
   - *Overall*: total execution time of the run. Might not equal the sum of the various steps executions times. Indeed, some operations performed between those steps are not included in the corresponding execution times

Run information	Run nb.	1	2	3	4	5	6
	Image	728725	728725	728725	728725	728725	728725
	Width	131072	131072	131072	131072	131072	131072
	Height	57856	57856	57856	57856	57856	57856
	Tile width	512	512	512	1024	1024	1024
	Tile height	512	512	512	1024	1024	1024
	Tiles	29900	29900	29900	7353	7353	7353
	Jobs	16	32	64	16	32	64
	RAM (Go)	100	100	100	100	100	100
Polygons	Found	10009	10009	10009	8418	8418	8418
	Merged	7294	7294	7294	7195	7195	7195
	Cell	5172	5169	5169	5141	5118	5128
	Pattern	1581	1567	1572	1528	1540	1554
	Dispatched	6753	6736	6741	6669	6658	6682
LSL	Loading	593.145	1544.394	773.857	537.775	951.596	1210.165
	Segment	4801.758	6736.784	7321.667	5148.477	7376.851	7757.463
	Location	2083.925	2492.705	2472.753	2033.105	2427.312	2690.361
	Overall	**476.952**	**348.405**	**199.245**	**536.385**	**351.635**	**206.755**
	Merging	14.324	14.451	18.086	40.640	40.605	40.774
Dispatching	Model	ET-FL	ET-FL	ET-FL	ET-FL	ET-FL	ET-FL
	Fetch 1	251.098	5.148	1.678	108.064	5.766	1.485
	Cells	765.858	758.031	739.323	762.242	758.993	741.434
	Fetch 2	0.750	1.079	0.959	1.077	0.860	0.830
	Patterns	112.861	140.930	135.782	142.667	141.502	137.354
	Overall	**1130.706**	**905.329**	**877.897**	**1014.201**	**907.267**	**881.252**
Classification	Model	ET-DIC	ET-DIC	ET-DIC	ET-DIC	ET-DIC	ET-DIC
	Fetch 3	1.372	1.602	1.664	1.200	1.431	1.167
	Cells	19.248	14.213	12.614	20.849	14.141	12.165
	Model	ET-DIC	ET-DIC	ET-DIC	ET-DIC	ET-DIC	ET-DIC
	Fetch 4	0.667	0.729	0.851	0.582	0.781	0.623
	Patterns	7.527	5.639	4.978	7.465	5.212	5.136
	Overall	**28.865**	**22.240**	**20.162**	**30.149**	**21.622**	**19.149**
Net.	Caching	6193.270	27.953	4.178	3858.679	10.786	4.255
	Upload	4993.832	444.564	444.000	4039.734	471.113	435.173
Total	Not net. 1	1652.464	1291.571	1116.922	1621.971	1321.591	1148.385
	Not net. 2	6646.296	1736.134	1560.921	5661.706	1792.703	1583.558
	Overall	**12839.566**	**1764.088**	**1565.100**	**9520.385**	**1803.489**	**1587.814**

Table E.1: Effects of varying the tile sizes and the available number processes on the execution times. Test image (728725) has dimensions 57856×131072.

	Run nb.	1	2	3	4	5	6
Run information	Image	716528	728725	728725	728725	8120444	8120444
	Width	163840	131072	131072	131072	172032	172032
	Height	95744	57856	57856	57856	104704	104704
	Tile width	512	1024	1024	512	1024	1024
	Tile height	512	1024	1024	512	1024	1024
	Tiles	61750	7353	7353	29900	17510	17510
	Jobs	16	16	16	16	100	100
	RAM (Go)	32	32	32	100	1000	1000
Polygons	Found	49442	8385	8385	9998	201974	202853
	Merged	42573	7164	7164	7330	183782	183992
	Cell	81	149	2468	2755	/	109510
	Pattern	2384	833	1077	1393	/	28960
	Dispatched	2465	982	3545	4148	174389	138470
LSL	Loading	129001.7494	53514.2437	53710.7671	20688.7785	247867.2302	247462.9641
	Segment	9485.1414	4743.8858	4719.1849	6713.7414	10068.0100	9896.3807
	Location	4209.0709	1971.8888	1961.1859	2738.4343	4458.5427	4471.3817
	Overall	**9158.3703**	**3829.1936**	**3835.3665**	**1930.2026**	**2686.0573**	2724.0464
	Merging	91.0140	57.6108	57.7504	36.0167	2052.1225	1972.3464
Dispatching	Model	ET-FL (r.)	ET-FL	ET-FL	ET-FL	ET-FL	ET-DIC
	Fetch 1	313.2474	146.9693	102.1281	234.7505	19962.1771	8697.8838
	Cells	288.3571	92.5044	493.3059	607.5096	56615.5098	381.4897
	Fetch 2	1.2727	0.4469	0.6815	1.0183	124.6983	153.0141
	Patterns	279.2642	73.3186	156.8081	153.6380	348.1110	155.1389
	Overall	**883.0553**	**313.4013**	**753.0886**	**997.1051**	**81731.1864**	**9391.5655**
Classification	Fetch 3	0.5211	0.2369	0.6788	1.0994	124.6983	20.2389
	Cells	1.0911	1.1579	10.7008	12.6491	277.9153	235.6081
	Fetch 4	0.9426	0.4519	0.4557	1.0570	348.1110	145.5594
	Patterns	13.2939	6.3793	7.2864	10.1172	37.3042	50.3971
	Overall	**15.8702**	**8.2355**	**19.1502**	**24.9634**	**811.8941**	**452.8960**
	Upload	5661.45	290.23	2431.7649	345.5561	28151.8883	8588.6074

Table E.2: Cytomine server

List of Tables

List of Figures

Bibliography

[BLF10] Steven R Bomeli, Shane O LeBeau, and Robert L Ferris. "Evaluation of a thyroid nodule". In: *Otolaryngologic clinics of North America* 43.2 (2010), pp. 229–238.

[Bra00] G. Bradski. "The OpenCV Library". In: *Dr. Dobb's Journal of Software Tools* (2000).

[Cla16] Alex Clark. *Pillow (version 3.2.x)*. 2016. URL: `https://github.com/python-pillow/Pillow/tree/3.2.x`.

[Deb13] Antoine Deblire. "Segmentation et classification automatiques de cytoponctions de la thyroïde." fr. MA thesis. Université de Liège, Liège, Belgique, 2013, p. 74.

[Dic16] Collins English Dictionnary. *Definition of 'cytology'*. 2016. URL: `http://www.collinsdictionary.com/dictionary/english/cytology` (visited on 05/21/2016).

[GEW06] Pierre Geurts, Damien Ernst, and Louis Wehenkel. "Extremely randomized trees". In: *Machine Learning* 63.1 (2006), pp. 3–42. ISSN: 1573-0565. DOI: `10.1007/s10994-006-6226-1`. URL: `http://dx.doi.org/10.1007/s10994-006-6226-1`.

[Gil13] Sean Gillies. *Shapely User Manual (version 1.2 and 1.3)*. Dec. 2013. URL: `https://pypi.python.org/pypi/Shapely`.

[Mar+16] Raphaël Marée et al. "Collaborative analysis of multi-gigapixel imaging data using Cytomine". In: *Bioinformatics* (2016), btw013.

[MGW16] Raphaël Marée, Pierre Geurts, and Louis Wehenkel. "Towards generic image classification using tree-based learning: An extensive empirical study". In: *Pattern Recognition Letters* 74 (2016), pp. 17–23. ISSN: 0167-8655. DOI: `http://dx.doi.org/10.1016/j.patrec.2016.01.006`. URL: `http://www.sciencedirect.com/science/article/pii/S0167865516000179`.

[Oli07] Travis E Oliphant. "Python for scientific computing". In: *CiSE* 9.3 (2007), pp. 10–20.

[Ots75] Nobuyuki Otsu. "A threshold selection method from gray-level histograms". In: *Automatica* 11.285-296 (1975), pp. 23–27.

[Ped+11] Fabian Pedregosa et al. "Scikit-learn: Machine learning in Python". In: *The Journal of Machine Learning Research* 12 (2011), pp. 2825–2830. URL: `http://scikit-learn.org`.

[RJ01] Arnout C Ruifrok and Dennis A Johnston. "Quantification of histochemical staining by color deconvolution." In: *Analytical and quantitative cytology and histology/the International Academy of Cytology [and] American Society of Cytology* 23.4 (2001), pp. 291–299.

[Sei13] Chase Seibert. *Diagnosing Memory 'Leaks' in Python*. 2013. URL: http://chase-seibert.github.io/blog/2013/08/03/diagnosing-memory-leaks-python.html (visited on 05/29/2016).

[VCV11] S. Van der Walt, S. C. Colbert, and G. Varoquaux. "The NumPy array: a structure for efficient numerical computation". In: *CiSE* 13.2 (2011), pp. 22–30.

[Wal+14] Stéfan van der Walt et al. "scikit-image: image processing in Python". In: *PeerJ* 2 (June 2014), e453. ISSN: 2167-8359. DOI: 10.7717/peerj.453. URL: http://dx.doi.org/10.7717/peerj.453.