2013 - Domaine Outils pour la surveillance environnementale **- Action 9** FlowCam PhytoImage

Guide méthodologique. Version actualisée de ZooPhytoImage avec refonte de l'interface graphique

Action 9. FlowCam ZooPhytoImage. Livrable n°1

Rapport final

Philippe GROSJEAN (Université de Mons, Belgique)

Février 2014

:

AUTEURS

Philippe GROSJEAN, Professeur (Univ. Mons, Belgique) - philippe.grosjean@umons.ac.be

CORRESPONDANTS

Onema : Marie Claude XIMENES (Onema), marie-claude.ximenes@onema.fr

Ifremer : Catherine BELIN (Ifremer), catherine.belin@ifremer.fr

AUTRES CONTRIBUTEURS

Kevin DENIS, doctorant (Univ. Mons, Belgique) - kevin.denis@umons.ac.be

Nour ALI, doctorante (Univ. Mons, Belgique & ULCO Wimereux) - nour.ali@umons.ac.be

Guillaume WACQUET (Ifremer) – guillaume.wacquet@ifremer.fr

Droits d'usage : libre accès
Niveau géographique : national
Couverture géographique : nationale
Niveau de lecture : experts

RESUME

Zoo/PhytoImage 4 est un logiciel « open source » basé sur R et ImageJ. Il sert au traitement d'images numériques de plancton obtenues à partir de différents appareils tels le FlowCAM, un scanner à plat, des micro- ou macrophotographies, etc.

Le principe général consiste à détourer individuellement les particules planctoniques visibles sur les images, et ensuite à en extraire des mesures (appelées « attributs ») telles la taille, la forme, la répartition de la transparence, les textures, etc. Ces attributs sont ensuite utilisés par un outil de classification pour définir automatiquement le groupe taxonomique auquel appartient chaque particule.

L'outil de classification est obtenu par apprentissage d'un algorithme de type « machine learning » qui établit un lien entre les attributs des particules et les groupes taxonomiques sur base d'un ensemble de particules d'identité connue (le set d'apprentissage qui est élaboré manuellement par l'opérateur sur base de quelques centaines ou milliers de particules d'exemple). Avant son utilisation, les performances de l'outil de classification sont vérifiées sur un ensemble indépendant d'autres particules connues (le set de test), ou par une méthode non biaisée d'estimation de l'erreur appelée « validation croisée ». Les résultats sont consignés dans un tableau de contingence à double entrée qui compare les identifications automatiques avec les identifications manuelles : la matrice de confusion. Zoo/PhytoImage 4 propose plusieurs outils diagnostics numériques et graphique pour déterminer si l'outil de classification offre des performances suffisantes, dont des visualisations nouvelles, comme le graphique « F-score » par groupe. Si l'outil de classification offre des performances trop faibles, Zoo/Phytoimage permet de tester divers remaniement des groupes taxonomiques pour améliorer le résultat.

L'identification automatisée de particules planctoniques est un exercice difficile, et actuellement, le taux d'erreur reste trop grand pour un usage opérationnel tel quel (autour de 20-30 % d'erreur globale, mais les erreurs par groupe varient énormément et peuvent monter jusqu'à 90-100 % pour les groupes difficiles ou rares). Ainsi, une validation manuelle par un opérateur entrainé est nécessaire (reclassement sur base du visuel des vignettes). Ce travail est long et laborieux. Il limite grandement l'intérêt de l'automatisation. Un nouvelle technique de détection des particules suspectes, combinée à une modélisation et ensuite, une correction statistique de l'erreur, est implémentée dans Zoo/Phytoimage 4. Elle permet de réduire la fraction de vignettes à valider manuellement (par exemple, 20 à 40 % seulement), tout en conservant une erreur de l'ordre de 10 % maximum pour tous les groupes. Une interface graphique utilisateur améliorée est ajoutée à la version 4 pour permettre une validation partielle manuelle des vignettes avec un confort d'utilisation maximum.

En plus de ce traitement, Zoo/PhytoImage 4 gère aussi les métadonnées, le stockage optimisé en espace disque et en vitesse de récupération de l'information. Enfin, ce logiciel calcule et extrait les variables dérivées d'intérêt comme les abondances, les biomasses, les biovolumes et/ou les spectres de tailles totaux et par groupe taxonomique. Il travaille en lot et est capable de traiter toute une série d'échantillons en une seule opération.

Le livrable est le manuel utilisateur de la version 4 de Zoo/PhytoImage (en anglais) qui détaille tous ces outils du point de vue de l'utilisateur final. Il montre comment ces outils sont mis en pratique pour l'analyse de données, y compris sur base de petits exemples et des notes sont incluses aux endroits où des pièges dans l'utilisation du logiciel ont été identifiés. Les binaires et codes sources correspondant à cette version 4 de Zoo/PhytoImage sont mis à la disposition de l'IFREMER également.

Le système couplé FlowCAM / ZooPhytoImage devient un outil véritablement opérationnel en 2014. Totalement adapté aux observations du phytoplancton réalisées dans le cadre du réseau d'observation REPHY, il permettra de mieux répondre aux sollicitations présentes et

futures concernant l'évaluation de la qualité des eaux littorales et marines dans le cadre des exigences européennes, telles que la DCE et la DCSMM. Un des bénéfices immédiats sera par exemple pour l'acquisition des données nécessaires au calcul de l'indice abondance composant l'indicateur phytoplancton pour la DCE en Manche- Atlantique, indice qui est basé sur la proportion de taxons du micro-phytoplancton présents en quantité importante dans un échantillon.

MOTS CLES (THEMATIQUE ET GEOGRAPHIQUE)

Océanographie biologique, Plancton, Surveillance côtière, Analyse automatisée, Analyse d'image, Classification supervisée

TITLE

Methodological manual. Updated version of Zoo/PhytoImage with a refactored graphical user interface

ABSTRACT

Zoo/PhytoImage 4 is an « open source » software based on R and ImageJ. It processes numerical images of plankton particles digitized using a FlowCAM, a flat-bed scanner, micro- or macrophotos, etc.

The general concept consists in the individual outlining of particles on the pictures, followed by their measurements (so-called « attributes ») such the size, the shape, transparency, textures, etc. These attributes are then used by a classification tool to automatically predict the taxonomic group the particles belong to.

The classifier is obtained after a learning stage using a machine learning algorithm and a training set of pre-identified particles. The algorithm learns to recognize the taxonomic group from the set of attributes measured on the picture. The training set is manually build from a few hundreds to a few thousands of representative items by an operator that visually identifies the candidate particles on the images. Before use, the performances of a classifier are checked on another, independent, set of known particles called the test set, or by using an unbiased technique such cross-validation. Results are collected in a two-way contingency table that compares automatic and manual classification of the particles of the test set. This is the confusion matrix. Zoo/PhytoImage 4 proposes a series of numerical and graphical diagnostic tools to assess the performances of the classifiers, including new visualizations like the « F-score » by group plot. If the performances of the classifier are too low, Zoo/PhytoImage allows to test how the reworking of the groups can possibly increase these.

The automatic classification of plankton is a difficult exercise. Currently, the residual error rate still remains too high for a practical use of the fully automatic approach (typically around 20-30 % of global error, but error by group vary much and can reach 90-100 % for rare or difficult groups). Hence, a manual (visual) validation of the classification by a trained operator is required to ultimately lower the error, global and by group. This work is long and fastidious. It greatly limits the interest of the automation provided by this approach. A new technique to detect suspect particles (those with a greater probability to be wrongly classified), combined with a modeling, and then, a statistical correction of the residual error, is implemented in Zoo/PhytoImage 4. The fraction of vignettes that need to be manually validated is lower (for instance, between 20 and 40% only), while keeping residual error per group around 10 % or less. A reworked graphical user interface is added to this version 4 to facilitate partial validation of the vignettes with the higher efficiency and ease of use.

In addition to this process, Zoo/PhytoImage 4 also manages metadata. It optimizes storage on the disk and the speed to gather any information back. Finally, this software computes and extracts derived variables of interest, like abundances, biomasses, biovolumes or size-spectra per total or per taxonomic group. It operates in batch and can deal with a whole series of samples in one pass.

The deliverable is the user manual of Zoo/PhytoImage 4 that details all these features from the user point of view, showing how to apply these techniques for plankton analysis, including with little examples and advices where there are commonly identified pitfalls. Corresponding binaries and source code of Zoo/PhytoImage 4 has been delivered to IFREMER too.

The FlowCAM / ZooPhytoImage is becoming an operational tool in 2014. Completely adapted to the phytoplankton observations performed in the context of the French monitoring network

REPHY, it will allow answering more accurately to the questions of WFD and MSFD concerning the evaluation of marine water quality. For instance, the first benefit will be for the acquisition of data necessary to the calculation of abundance index, part of the phytoplankton index for WFD in Channel and Atlantic water bodies : as a matter of fact, this index is based on the proportion of micro-phytoplankton taxa which are very abundant in a water sample.

KEY WORDS (THEMATIC AND GEOGRAPHICAL AREA)

Biological oceanography, Plankton, Costal survey, Automated analysis, Image analysis, Machine learning

SYNTHESE POUR L'ACTION OPERAT IONNELLE

Le logiciel Zoo/PhytoImage est spécialisé dans le traitement d'image numériques de plancton. Il est « open source » et existe depuis 2004. Actuellement, ses droits sont partagés entre l'UMONS, l'IFREMER et la politique scientifique belge (BelSpo) qui ont tous les trois contribué financièrement à son développement durant ces huit dernières années. La version initiale publique 1 a permis de mettre en place les concepts du traitement d'images numériques de plancton. Il s'agit, en effet, du premier logiciel public dédié spécifiquement à la classification supervisée du plancton sur base d'images numériques (les logiciels ZooProcess associé au Zooscan et Plankton Identifier ont suivi un an plus tard et le premier en particulier s'est très fortement inspiré des fondements établis par Zoo/PhytoImage, alors que Visual Spreadsheet associé au FlowCAM ne permet pas la classification supervisée (en tout cas jusqu'à présent).

La version 2 non publique a incorporé progressivement des ajoutes suite à son développement et son utilisation à l'UMONS et à l'IFREMER. Au début du présent projet, le constat était le suivant : (1) les différents patchs ajoutés à la version 2 nécessitent une meilleure intégration dans (2) un logiciel refondu complètement pour, notamment, s'exécuter sur plusieurs plateformes telles que Linux/Unix et Mac OS X en plus de Windows. Et enfin (3) l'interface graphique utilisateur nécessitait d'être revue pour assurer plus de confort, d'efficacité et de facilité dans le tri manuel des vignettes lors de l'opération indispensable de validation post-traitement automatique. En outre, autant pour le point précédent que pour la manipulation et le stockage de grandes quantités de données qui seront nécessairement générées par une utilisation en routine du logiciel, il a fallu (4) revoir complètement le format de stockage des données pour le rendre le plus performant possible (récupération instantanée de n'importe quelle information : abondance ou biomasse par groupe taxonomique, ou encore, des données ou l'image associée à une particule planctonique en particulier).

Le présent rapport détaille les fonctionnalités de ce logiciel, telles que disponibles dans sa version publique 3, amendée par de nouvelles additions (validation des suspects, correction d'erreur, interface utilisateur pour la validation manuelle retravaillée, …) non publiques dans sa version 4. La version 3 est une refonte complète du code en interne pour rendre le logiciel compatible avec Linux/Unix et Mac OS X (compatible uniquement avec Windows) et pour préparer les fonctions qui seront nécessaires à l'avenir, comme un stockage optimisé pour la récupération des images (« vignettes ») des particules planctoniques ou encore, pour préparer une transition future vers le calcul massivement parallèle. La version 4 rajoute des éléments développés spécifiques en vue d'un déploiement opérationnel du logiciel dans le cadre du suivi de séries planctoniques, particulièrement dans le cadre du REPHY à l'IFREMER.

Logiquement, la future version 5 sera également publique et proposera alors un outil complètement abouti pour une utilisation opérationnelle de l'analyse d'image et de la classification supervisée appliquée au monitoring de plancton. La version actuelle 4 peut donc être considérée comme un produit abouti pour son utilisation opérationnelle **au niveau de son design**, mais entrant en phase de « beta-test » (c'est-à-dire, débogage poussé) de manière concomitante à sa pré-incorporation progressive dans le cadre des activités REPHY/IFREMER. Cela sera réalisé en 2014.

Zoo/PhytoImage version 4 compte près de 10.000 lignes de code en langage R, plusieurs milliers en Java et JavaScript, et est complètement documenté au niveau des fonctions. La version 4 a ajouté près de 3.000 lignes de code, et le reste a subi un « refactoring » complet dans la version 3. Il incorpore des outils, maintenant classiques, d'analyse des particules détourées sur les images numériques issues d'un FlowCAM, d'un scanner à plat, de macro- ou microphotographies, et de classification supervisée de ces particules à l'aide

d'algorithmes tels que « random forest », les machines à vecteurs supports, les algorithmes bayésiens naïfs, la discrimination linéaire, etc.

Zoo/PhytoImage version 4 possède également des **outils méthodologiques nouveaux** particulièrement critiques pour un déploiement opérationnel de la classification supervisée d'image numériques de plancton. Il s'agit de plusieurs outils diagnostics graphiques nouveaux pour l'analyse des performances de l'outil de classification, bâtis autour de la matrice de confusion et de la validation croisée (par exemple, le graphe F-score). Il s'agit aussi et surtout, d'un nouveau procédé de validation manuelle des vignettes après classification automatisée qui permet d'optimiser le temps de traitement d'un échantillon en arrivant à un niveau marginal d'erreur pour tous les groupes en ne validant qu'une partie des vignettes (20 à 40% maximum). Cette technique nouvelle fait l'objet d'un manuscrit en préparation. Elle est basée sur un outil de reconnaissance des particules probablement mal classées (appelées « suspectes ») qui utilise des variables calculant la probabilité pour une particule d'être erronée (confusion entre deux classes, particules dans des groupes rares, information biologique quant à la probabilité d'occurrence d'un groupe taxonomique dans l'échantillon traité, etc.) Ces variables sont combinées aux attributs mesurés sur les vignettes de chaque particule pour identifier les items suspects. Ensuite, la validation manuelle se concentre essentiellement sur ces suspects. A l'usage, il apparaît que l'erreur y est effectivement concentrée, et que la méthode est efficace pour corriger l'erreur plus rapidement et en visionnant moins de vignettes. Enfin, l'erreur observée est modélisée au sein de la matrice de confusion et ce dernier modèle est utilisé pour apporter une dernière correction statistique au niveau des dénombrements par groupes.

Cette technique est itérative. L'opérateur valide 1/20ème de l'échantillon à chaque étape. La détection des suspects et la correction d'erreur sont recalculées à chaque étape. L'opérateur visualise sur un graphique la part d'erreur corrigée à l'issue de chaque étape et peut ainsi décider au cas par cas, quand il est judicieux d'arrêter la validation manuelle. Ainsi, un échantillon contenant plus de particules problématiques pourra faire l'objet d'une validation plus poussée. Cette approche itérative de la validation partielle est couplée, dans Zoo/PhytoImage version 4 à une nouvelle interface graphique pour le tri des particules sur base d'une vue d'ensemble de tous les groupes à partir de laquelle toute particule (mal classée) peut être déplacée à la souris d'un groupe à l'autre.

Conclusion

Le logiciel Zoo/PhytoImage dans sa version 4, et couplé au FlowCAM, devient un outil véritablement opérationnel en 2014. Totalement adapté aux observations du phytoplancton réalisées dans le cadre du réseau d'observation REPHY, il permettra de mieux répondre aux sollicitations présentes et futures concernant l'évaluation de la qualité des eaux littorales et marines dans le cadre des exigences européennes, telles que la DCE et la DCSMM. Un des bénéfices immédiats sera par exemple pour l'acquisition des données nécessaires au calcul de l'indice abondance composant l'indicateur phytoplancton en Manche- Atlantique, indice qui est basé sur la proportion de taxons du micro-phytoplancton présents en quantité importante dans un échantillon.

En effet, un gain important en temps est attendu avec cet outil en comparaison des observations classiques au microscope, ce qui permettra potentiellement d'augmenter le nombre d'échantillons, et donc d'avoir une meilleure représentativité des données acquises sur un site. Le gain en efficacité et en pertinence est également crucial, grâce à la sauvegarde des observations permettant de revenir sur celles-ci a posteriori Il permettra en outre une plus grande homogénéité entre les données acquises sur différents sites, en diminuant l'effet observateur.

Enfin, de nouveaux paramètres pourront être calculés afin d'aller au-delà de l'information

couramment accessible via le proxy de biomasse phytoplanctonique qu'est la concentration en chlorophylle. Il s'agira, à partir des nombreuses mesures fournies par le logiciel, d'estimer systématiquement, par exemple, le biovolume des cellules du phytoplancton permettant alors d'estimer des biomasses en terme de contenu en Carbone et donc d'obtenir une information précise quant à la quantité de phytoplancton disponible dans l'écosystème et contribuant à la structure du réseau trophique.

Le présent livrable est composé de trois rapports :

Zoo/PhytoImage Version 4. Computer-Assisted Plankton Images Analysis. User Manual. Ph. Grosjean Ph. & K. Denis. February 2014

Correction statistique de l'erreur dans le cadre de la classification automatique du plancton. Directeur : Ph. Grosjean. Projet réalisé par Flavien Dereume-Hancart, 2013.

Amélioration de la validation partielle des suspects et de la correction statistique de l'erreur dans Zoo/PhytoImage pour les échantillons REPHY. K. Denis & Ph. Grosjean, 2013.

Ce livrable est complémentaire des trois autres livrables fournis pour cette action :

Livrable 2 Protocole sur les outils de reconnaissance optimisés Manche Atlantique

Livrable 3 Rapport sur la finalisation d'un outil pré-opérationnel FlowCAM / PhytoImage

Livrable 4 Rapport sur l'utilisation conjointe de FlowCAM / PhytoImage et de la cytométrie en flux. Premiers résultats et perspectives.

ZOO/PHYTOIMAGE VERSION 4

Computer-Assisted Plankton Images Analysis

USER MANUAL

The ZooImage development team

February 2014

Ph. Grosjean & K. Denis: Numerical Ecology of Aquatic Systems, UMONS, Belgium
X. Irigoien, G. Boyra & I. Arregi: AZTI Tecnalia, Spain
A. Lopez-Urrutia: Centro Oceanográfico de Gijón, IEO, Spain
M. Sieracki & B. Tupper (FlowCAM plugin)

1. Introduction

Zooplankton or phytoplankton samples analysis is traditionally associated with long and boring sessions spent counting fixed plankton items under the binocular with formaldehyde vapors floating around. Although this picture of a planktonologist will probably remain for a while, there seems to be another way to gather data about zooplankton: computer-assisted analysis of plankton digital images. A whole suite of hardware to take pictures of our animals, both in situ and/or from fixed samples, is now available: Flowcam, laser OPC, VPR, Zooscan,... (more to come with Holocam, Sipper, Zoovis, HAB Buoy, ...), not forgetting the use of a digital camera on top of a binocular or with a macro lens. But digital images of zooplankton are barely usable as such: they must be analyzed in a way that biologically and ecologically meaningful features are extracted from the pixels. A software doing such an analysis is thus indispensable.

Zoo/PhytoImage aims to provide a powerful and feature-rich software solution to use zooplankton or phytoplankton pictures of various origins and turn them into a table of usable measurements (i.e., abundances, total and partial size spectra, total and partial biomasses, ...). Zoo/PhytoImage is not tight with any of the previously cited devices, and it is not going to be a commercial product. It is distributed for free (GPL license, distributed through its web site, http://www.sciviews.org/zooimage) and it is open, meaning it provides a general framework to import images, analyze them, and export results from and to a large number of systems. So, everybody can use Zoo/PhytoImage... but better yet, every developer can also contribute to it! The Open Source approach of wiring many willing developers around the world in a common project has already shown its efficiency: Linux, Apache, but also R or ImageJ in the field of statistics and image analysis, respectively, are good examples of it. Zoo/PhytoImage is based on ImageJ and R, and it runs on Linux... but it can also be run on Windows, Mac OS, or various Unixes[1]. Zoo/PhytoImage's best qualifying is "reusability". It is born by reusing various features of great existing software like ImageJ, or R, and it provides itself reusable components, for the benefit of both users and developers.

Zoo/PhytoImage can be used on images acquired in different situations: *in situ* (like VPR or HAB Buoy) or in the lab (fixed samples scanned with the Zooscan, for instance). The general framework of Zoo/PhytoImage is designed in a way that the software is capable of dealing effectively with images of various origins and characteristics. Consequently, it is not a streamlined and rigid system. It is rather made of a collection of different and customizable applications collected together in a single system. This user's manual will guide you in your first use of Zoo/PhytoImage.

> *This manual describes current version of ZooImage (4.0-0), which is* ***not*** *a public version ! It is geared towards early adoption among our partners : UMONS, IFREMER, BelSpo, ULCO and LISIC. The functions presented here will eventually land in the next public version 5. However, 4/5 of the code is commonwith version 3, whichi is public and downloadable from CRAN (http://cran.r-project.org).*

1 The current version is developed mainly on Mac OS X, but is also tested on Windows and Linux Ubuntu.

2. Changes from version 1 and 2

Zoo/PhytoImage version 1.2 was the latest public version distributed on http://www.sciviews.org/zooimage until now. Version 2 of the software was not public and contained several developments made for us (UMONS university) and our main partners : IFREMER in France and Belspo (Belgian Science Policy) in Belgium.

Version 3 of ZooImage collects most of these developments into a relifted system, and it is distributed on CRAN (http://cran.r-project.org). Finally, recent additions made in version 4 do complete the set of features. Main changes are :

- Updated code for running on latest R version 3,
- Complete internal refactoring to make it compatible with Linux and Mac OS X, in addition to Windows. Version 3 also supports Windows Vista, 7 and 8, in addition to Windows XP.
- A new storage format, called ZIDB, that is much faster to retrieve vignettes.
- Routines to build, sort and use test sets, similarly to training sets.
- Functions to display vignettes directly insode R graphs (using R scripts).
- Improved handling of confusion matrices, with the possibility to change prior probabilities of the classes and inspect how this changes the shape of the confusion matrix.
- A battery of summary statistics for the confusion matrix (recall, precision, F-score, specificity, ...)
- New and improved graphs for the confusion matrix, including F-score plots and dendrograms depicting hierarchical classification of the classes according to their confusion.

2.1. New data storage format

Among all those change, the most important one for end users is probably the new storage format, named ZIDB for Zoo/PhytoImage DataBase.

Data storage format is a key aspect for data analysis software. In statistics, there is a consensus towards a 'case-by-variable' format that is suitable for most (but simplest?) datasets. It presents the data in a two-dimensional table with variables in columns and cases (or individuals) in rows. Additional names for columns and/or for rows are allowed. Such data can be stored in plain text, being ASCII, UTF-8, ... encoded, using a predefined field separator and one row per line. The most commonly used format is CSV for « comma-separated values ». It uses either the comma (,

English version) or a semi-colon (; French version). Another frequent variant is the TSV format, which uses tabulations as field separators.

CSV or TSV are readable by all software, making them the most universal storage format for case-by-variable data. Excel (or other spreadsheet) formats can be used as well, but they are a little bit less widely recognized.

CSV or TSV are not the most efficient formats when it comes to memory usage or speed. Since numbers as stored as character strings, they consume much more memory than their binary counterpart. It is also impossible to retrieve some data in the middle of a table without reading all previous data since the offset in memory where those data are stored in not predictable.

Another shortcoming of the CSV or TSV format is the impossibility to associate metadata in addition the main two-dimensional table. Yet, this format remains one of the best to store small to mid-sized raw datasets and make sure they will be most readable in the future.

> *In Zoo/PhytoImage, we use a variant of the TSV format where the two-dimensional table of features measured on each particle is prepended by a section defining associated metadata in a key=value pairs set. It is the _dat1.zim file.*
>
> *The same data is also duplicated in own binary R format (.RData) which is much faster to load than the original TSV file.*

For the images, there is a large number of formats availables. The most widespread used ones are TIFF, JPEG, GIF and PNG. TIFF is the most versatile one, but the number of subformats that exist makes it difficult to read on some software for the most exotic configurations. It is the preferred format for RAW plankton images to be processed by Zoo/PhytoImage.

JPEG is a lossy-compression format that is restricted to RGB 24-bit images only. It is the most efficient (lowest size of the file) for images that should only be viewed. However, the compression algorithm introduces artifacts in the picture that cannot be reliable analyzed when compression factor is too large. This format is reserved for vignettes (small images containing only one particle) when they are only used for visual classification of the particles (no further image analysis on them).

GIF and PNG are image format that use lossless-compression algorithms. PNG was proposed as an alternative to the older GIF format because of patent and licence problems : the non-free licence of the GIF format was a problem in the past, but now, the patent has expired and this format can be freely used. However, PNG being defined later, it offers more flexibility, like for instance, the possibility to define an alpha channel defining the transparency of the pixels in addition to their color. GIF can only tag one color as being fully transparent, and the other ones are fully opaque. In Zoo/PhytoImage, the PNG format can be used in addition to JPEG for vignettes if a lossless compression is required and the image has a chance to be further analyzed at a later stage after the vignette is created. This alternate format for vignettes was introduced in version 3 of Zoo/PhytoImage.

In Zoo/PhytoImage, data for one sample contain three components :

1. A case-by-variable table containing fetures extracted by image analysis on each blob (particles or individual items in the images). This table is stored in TSV format and in binary R own format .Rdata containing a data frame for quick loading in R,

2. A list of metadata information about the sample contained in a plain test file with an 'ini file' organization with one 'key=value' per line. The same information is also stored as attributes of the R's data frame.

3. A series of 'vignettes' that are cropped subsections of the initial images containing the picture of a single particle, and enhanced for visual identification. These vignettes were stored as JPEG images in version 1 and 2, but PNG format is now also accepted from version 3.

Since there can be easily thousands of particles, and thus vignettes, in one single sample, it is not convenient to keep all these items in separate files on disk. In version 1 and 2, Zoo/PhytoImage did compressed (zipped) these component in a single archive file with the ZID extension (for Zoo/PhytoImage Data). This approach is simple and ensures readability of the data since the unzip program required to extract the components is widely available. However, unzipping the archive to access the vignettes is a

slow operation. This format prevents, thus a fluid and fast display of the vignettes for best user interaction and experience.

Starting from version 3, Zoo/PhytoImage now uses a custom binary format called ZIDB (for Zoo/PhytoImage DataBase). This format is indeed a hash table followed by binary versions of the different components. Fast C functions are used to access the different components for very fast retrieval of any vignettes, the features or the metadata. This format is a little bit less portable, but is easily accessible from R, and R itself is now widely available. In term of disk storage, the new ZIDB format is marginally (usually, around 5%) less compressed. So, you need an little extra storage space too.

Of course, a series of function have also be added to import data from the old ZID format, and to convert back and forth between the two formats.

3. INSTALLATION

3.1. Hardware requirement

Image analysis and automatic classification of images are computer-intensive processes, and you will likely analyze lots of objects (typically, hundreds of thousands, or millions of them). Thus, you need a recent and powerful computer to run Zoo/PhytoImage decently. Consider especially :

- A fast and recent multicore, and multithreaded processor.
- **4Gb of RAM memory** or more. Depending on the size of the images you want to analyze, you may need even more. Very large images issued from a flatbed scanner require at least 1Gb of RAM. Zooscan images may require even more! Nowadays, it is very easy to use 16Gb, or 32Gb of RAM on 64bit systems, so, consider this option seriously.
- After the processor speed and the RAM, the next most important part of your computer to work with images is **the graphic card and the screen**. Chose a rapid, optimized graphic card capable of displaying 1280×1024, or 1600×1200 pixels or more with 24/32bit color depth (millions of colors), associated with a high quality screen of no less than 19'. Dual-screen configuration can help too, since it gives more space for displaying side-by-side images and plots.
- Although Zoo/PhytoImage optimizes disk space by compressing all files, dealing with lots of high-resolution pictures is consumming a lot of space on disk. You will need a **fast hard disk of at least 2-4Tb of capacity**. One small SSD disk greatly improves the speed of the analysis when used to store the few samples currently manipulated.
- Finally, a good **backup system** is also required, unless you use a RAID system.

3.2. Download of the software

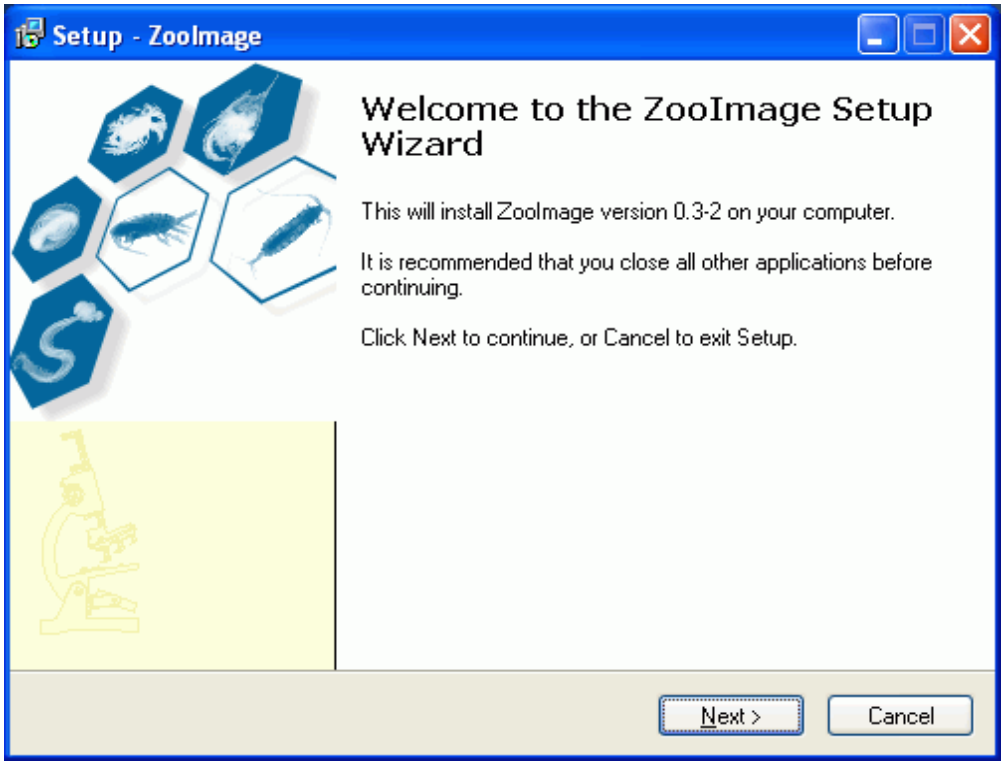
The software is available for download on the ZooImage website (http://www.sciviews.org/zooimage/). It can also be installed from within R, through CRAN : run `install.packages("zooimage")` from the R console.

Linux or Mac OS X users will have no problems installing R, and then Zoo/PhytoImage that way. The following section details installation from the Windows installer, as it exists since early versions (screenshots not updated).

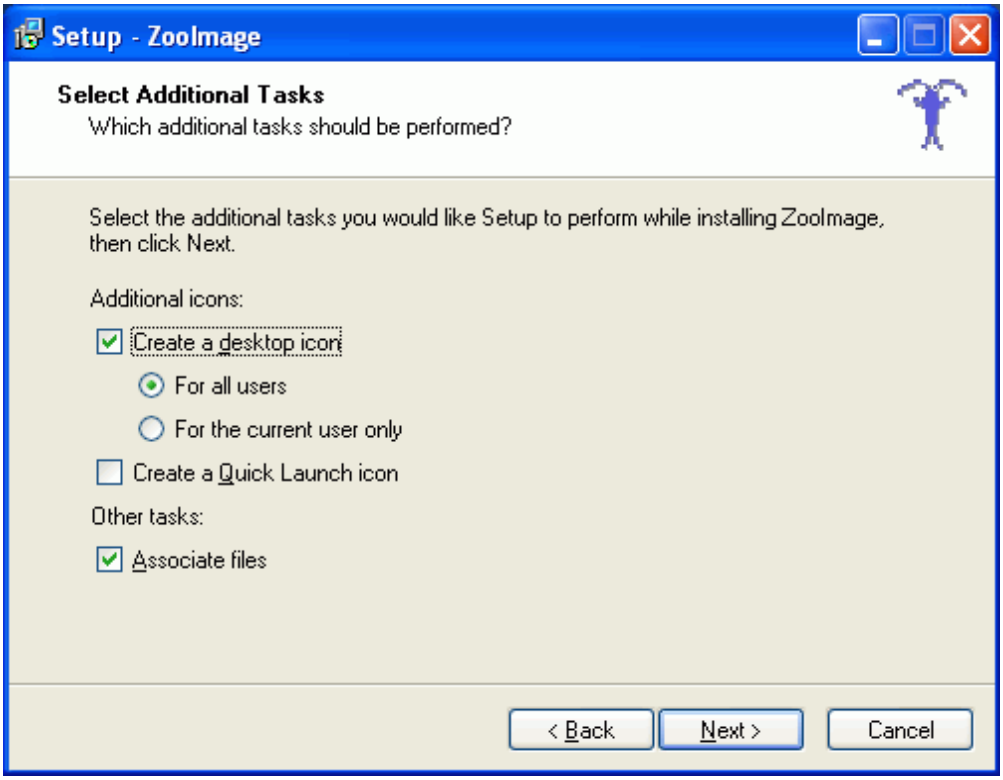
3.3. Installation of Zoo/PhytoImage under Windows

Zoo/PhytoImage will uses about 400Mo of space on your hard disk, when installed. You just have to execute the "ZooImage_[x.y-z]Setup.exe"

file that you downloaded and to follow the installer's instructions step-by-step. Default values for the options should be fine, if you don't understand them.



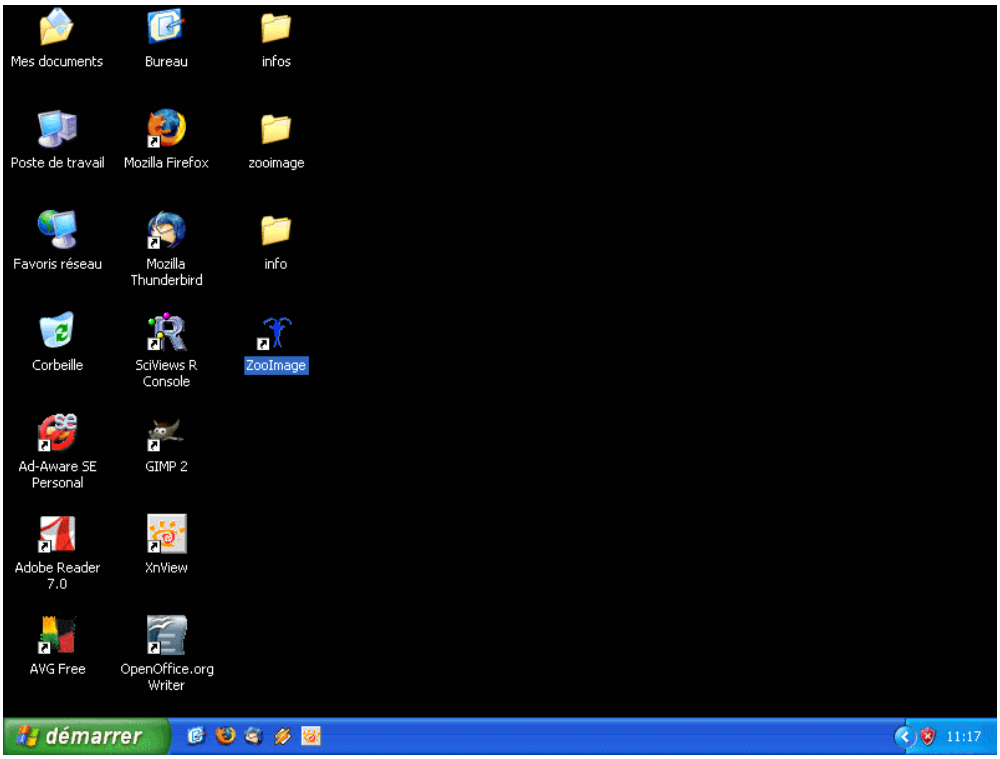
The first screen of the ZooImage installation assistant.



Another screen of the Zoo/PhytoImage assistant. You can create desktop and quick launch icon (in the quick launch bar).

*It is very important to **associate files** with Zoo/PhytoImage: those files have special extensions and it will not be possible to open them by a double-click in the Windows explorer if you don't select this option. So,* leave this option checked unless you have good reason to change it!

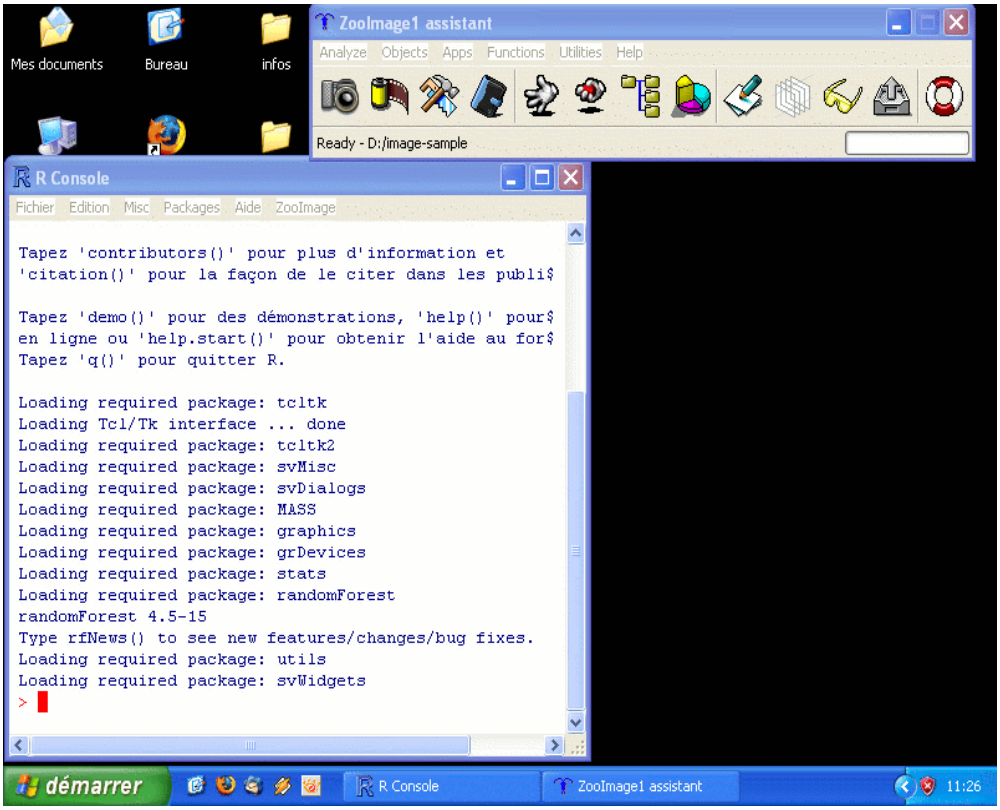
At the end of the installation, you should have a `ZooImage` entry in the start menu, and possibly a `ZooImage` icon on your desktop (if you left that option checked).



An example desktop with the `ZooImage` icon (a little blue copepod) currently selected.

4. First use of Zoo/PhytoImage

This quick tutorial will show you how to analyze the "Spain_Bioman" example images installed with the software. When you double-click on the `ZooImage` icon on the desktop, or select the `ZooImage (R)` entry in the start menu, two windows appear on screen: the **R console** and the **ZooImage assistant**.



The two first windows appearing when you start Zoo/PhytoImage. At bottom-left, the R console (you can interact with R there) and at top-right the ZooImage assistant window.

The **R console** allows you to control R[2] directly through command lines. You should not worry about this window, unless you are familiar with the R language. However, it logs important results and messages from your actions in Zoo/PhytoImage. So, you are better not to minimize it.

The **ZooImage assistant** window is a toolbox with a menu on top and a status bar on bottom. It will guide you during the whole process. Basically, you just have to click on the buttons from left to right to run the various steps of your analysis.

A Zoo/PhytoImage analysis is subdivided in three parts, as it the toolbox. For each part, you have four buttons:

2 R is the statistical software & environment on which ZooImage is based.

The three parts of the ZooImage process, materialized by three times four buttons. The last button shows the ZooImage user's manual.

- The first part deals with image importation and process.

  1.  **Acquire images**. Start an external acquisition software (Vuescan, or any other program).

  2.  **Import existing images**. Possibly convert the format of the images and/or rename them. If images are already in correct format, this function just make sure they have suitable *metadata* associated.

  3.  **Process images**. Basically, ImageJ is started. You are supposed to used one of the ZooImage-specific plugins in ImageJ to process your pictures.

  4.  **Make .zid files**. 'Zid' files stands for 'ZooImage Data' files. They contain all you need for the rest of the treatment, i.e., images of each individual[3], their measurements and the metadata. Yet, they store this information in a compressed way.[4]

- The second part help you to make an automatic classifier optimized for your zooplankton series.

  1.  **Make a training set**. This function prepares a directory with a hierarchy of subdirectories representing your manual classification (you can freely modify this structure at will) and extract vignettes from the samples you want to use for making your manual training set. You then have to manually classify them on screen by moving them to their respective directories with the mouse.

  2.  **Read training set**. Once you manually sorted the vignettes, this function collect this information into ZooImage. Statistics about you calssification (number of vignettes in each group) is the displayed.

  3.  **Make classifier**. Use a manual training to train an automatic classifier. You have the choice of various algorithms. You got some statistics at the end of the process to evaluate performances of your classifier (cross-validation).

3 These particular images are called 'vignettes' in ZooImage terminology.

4 If you started with uncompressed high-resolution 16bit grayscale pictures in TIFF format, you usually end up with .zid files that weight about 100 times less than the original pictures.

4.  **Analyze classifier**. Further analyses of your classifier's performances. Currently, only the confusion matrix showing differences between manual and automatic classification[5], is calculated. Other diagnostic tools will be added in future versions.

- The third part uses this classifier and the measurements done on all objects identified in your pictures (first part) to calculate automatically abundances, biomasses and size spectra in all your samples. You can then visualize results, or export them.

  1.  **Edit samples description**. Series of samples are identified by a list written in a specific Zoo/PhytoImage format. This list contains also further metadata about the series, and you have the opportunity to append various other measurements to the samples data (temperature, salinity, fluorescence, etc.).

  2.  **Process samples**. This is the workhorse function that process each sample of a given series one after the other, (1) identifying all individuals using your automatic classifier, (2) computing abundances per taxa, (3) calculating size classes in total and in each taxa for size spectra representations and studies, and (4) computing biomasses in total and per taxa, using a table of conversion from ECD[6] to cabon content, dry weight, etc. Data are converted per m^3, if suitable 'dilution' information is available in the metadata.

  3.  **View results**. Graphically present results. You can draw composite graphs (up to 12 different graphs on the same page), either time series of abundances or biomasses changes[7], or size spectra of given samples.

  4.  **Export results**. Results are written on the hard disk in ASCII format. This format is readable by any other software (Excel, Matlab, etc.).

> *Although you can export your results to analyze them in a different software, you don't* ***have*** *to do so. Zoo/PhytoImage operates in a R session, and the thousands of R functions are available for producing even the most sophisticated statistical analyses and graphs without leaving Zoo/PhytoImage/R.*

-  **Manual**. Display the PDF version of the user's manual.

5 The confusion matrix is shown both in tabular and in graphical presentations.

6 ECD = Equivalent Circular Diameter.

7 Spatial representations are not handled yet in this version, but they are planned in future versions.

5. ACQUIRE DIGITAL IMAGES OF ZOOPLANKTON OR PHYTOPLANKTON

Zoo/PhytoImage is **not** a digitizing software. It is only designed to analyse existing digital images. However, for convenience, it binds to your favorite external acquisition software (it should be hardware-specific). As an example, if you use a digital camera with a dedicated capture software[8], you can specify that software in Zoo/PhytoImage and start it from the ZooImage assistant in one click.

Zoo/PhytoImage can be used with **Vuescan**, an excellent and very capable software to acquire pictures from more than 400 commercial flatbed scanners and from more than 100 different RAW formats of digital cameras. Here we explain how to use Vuescan with a flatbed scanner to get digital zooplankton images... **but it should be clear that it is just an example: you are free to use any hardware/software combination you like to acquire your images!**

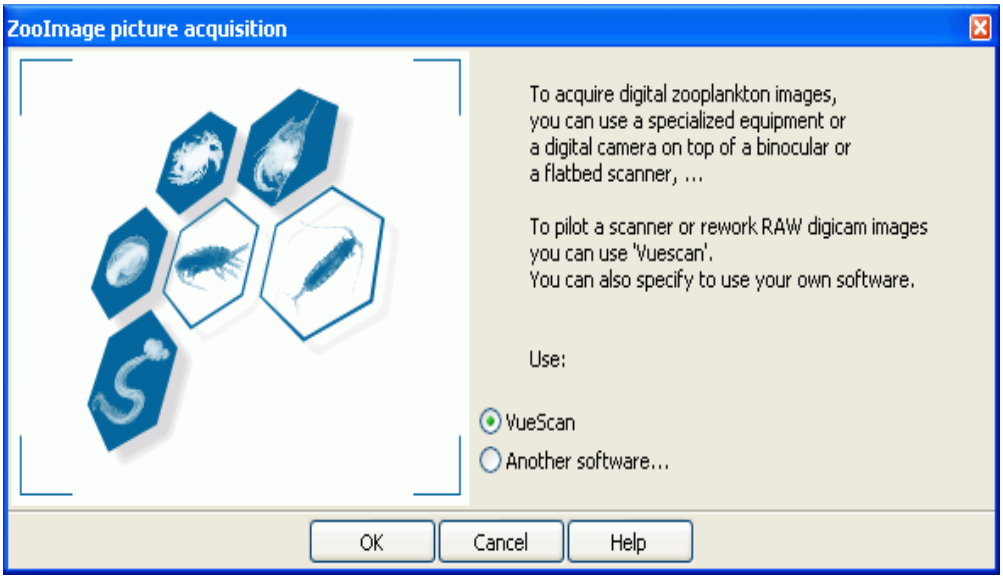
Vuescan is not a free software. It is a shareware distributed in two versions: personal and professional. You need the professional version. Its license is about $89, and you have to register your license with the author of Vuescan (see instruction in the Vuescan online help). We got the right to redistribute the trial version with ZooImage, but you have to unleach full features by entring your license code before you can use it in production.

5.1. The 'acquire images' tool

In this manual, we use examples images installed with the software. So, you do not need to acquire your own image to practice with Zoo/PhytoImage. As an example, we show you how you can get your own images using Vuescan.

To start your image acquisition software from the ZooImage assistant window, use the menu entry `Analyze → Acquire images...`, the shortcut `Ctrl+A`, or click on the first button in the toolbar.

8 For instance, Canon or Nikon digital reflex camera are bundled with specific capture software that you can use to save directly your picture on your hard disk.



You have a dialog box that let you choose the program to start (either Vuescan, or another one). Select Vuescan and click `OK`. Vuescan is opened. Once the software is registered, you can switch in `Advanced` mode by clicking on the corresponding button at the bottom (if Vuescan is started in `Guide me` mode). You have to parameterize Vuescan for your acquisition device (digital camera or flatbed scanner) and the type of images you want. Vueimage allows you to record both uncompressed TIFF files with 16bit gray levels and JPEG 24bit color files. These two types of files correspond respectively to the `Gray16bits 2400dpi` and `Color24bits 600dpi` plugins in ImageJ (see hereunder).

Vuescan offers a wide range of options for digitizing your pictures. A couple of options are very sensitive in the context of your image analysis. Additional documents are in preparation to list best Vuescan options for several digitizing devices (Zooscan, ...). You are also welcome to contribute your own recipe.

6. IMPORT IMAGES

Once your images are stored on your hard disk, you must prepare them for use in Zoo/PhytoImage. Use the menu entry `Analyze → Import images...`, the shortcut `Ctrl+I`, or click on the second button in the toolbar.

Zoo/Phytomage image importation is indeed performing several tasks to make sure your pictures are in correct formats and all required metadata are associated. In the current version, the function just checks the presence of metadata files, but more exhaustive control and processes are planned in future versions. **It means you have to do the rest of housekeeping manually!** Here is what you should do:

- Make sure that all the images you want to process are in one directory on your hard disk. Do not mix pictures you want to process with other ones on the same directory. **Keep them separate**. For instance, have one `d:\ImageProcess` directory where you store your fresh images and place them in one `d:\ImageDone` directory as soon as they are processed.[9]

Since Zoo/PhytoImage always starts from the current active directory when you have to browse for files and subdirectories, it saves time to switch it to the one where you store your raw images. The active directory is displayed in the status bar of the ZooImage assistant window. To change it, use the Utilities → Change active dir... menu entry.

- Make sure your images are in a correct format: uncompressed TIFF with 16bit gray scale (preferably with a resolution of 2400dpi) for the *Gray16bits 2400dpi* plugin and 24bit color JPEG (preferably with a resolution of 600dpi and with the lowest compression level) for the *Color24bits 600dpi* plugin. Other file formats will be accepted in the future. Use general graphic utilities like Imagemagick or XnView to convert image that are not in one of these formats.

- Make sure you respect the naming convention imposed by Zoo/PhytoImage, which is:

  `SCS.YYYY-MM-DD.SS+PP.EXT`

  With this convention, the images are easily identifiable in a long series, both by the software and by the human. In particular, sorting files alphabetically results in a chronologial sorting of the images, according to sampling dates.

  1. `SCS` is the identifying code of the “Series - Cruise - Station”. Use three to four letters to identify the point within all you series/cruises/stations data.

9 Once the images are completelly processed, you just need the resuting .zidb files somewhere on your hard disk. So, you can delete original pictures after making a backup on DVDs or external hard disks and save a lot of disk space!

2. `YYYY-MM-DD` is the date of sampling in `year-month-day` format. If for some reasons the day or the month is unknown, use `00`.
3. `SS` is a code to uniquely identify each sample (useful when several samples are taken the same date at the same station).
4. `PP` is the image identifier. Zoo/PhytoImage manages different images per sample, and even, images of different fractions at different dilutions of the same sample[10]. Zoo/PhytoImage will carry all required calculations, including collecting together results from the six images in a single .zid file, calculating abundances and biomasses per m^3, taking into account the two fractions at different dilutions, etc.
5. `EXT` is the file extension according to the file format. It must be `tif` (lowercase) for TIFF images and `jpg` (lowercase) for JPEG pictures.

> *You do no have to conform to the Zoo/PhytoImage naming convention of the images. However, the minimum is to use NAME+PP.EXT with whatever string you want that uniquely identify one sample, being at least A if you have only one image per sample, and EXT as above. Thus, as a minimum, TIFF images should end with +A.tif and JPEG images with +A.jpg.*

That say, we will now practice on the example pictures.

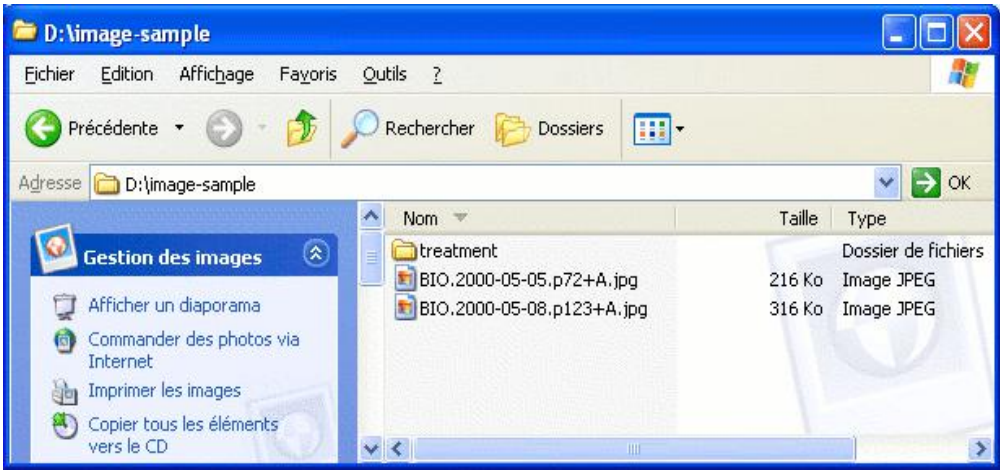
1. Prepare an empty directory on your hard disk (let's say, `D:\image-sample`, but you can freely choose another partition or directory name).

2. Swith the active directory there, using the `Utilities → Change active dir...` menu and select that directory.

3. Copy the two example pictures `BIO.2000-05-05.p72+A.jpg` & `BIO.2000-05-08.p123+A.jpg` that are located in the `\examples\_raw` subdirectories of your Zoo/PhytoImage installation directory (by default, it is `C:\Program Files\ZooImage` on English versions of Windows) in that directory. **Do not copy corresponding .zim files.**

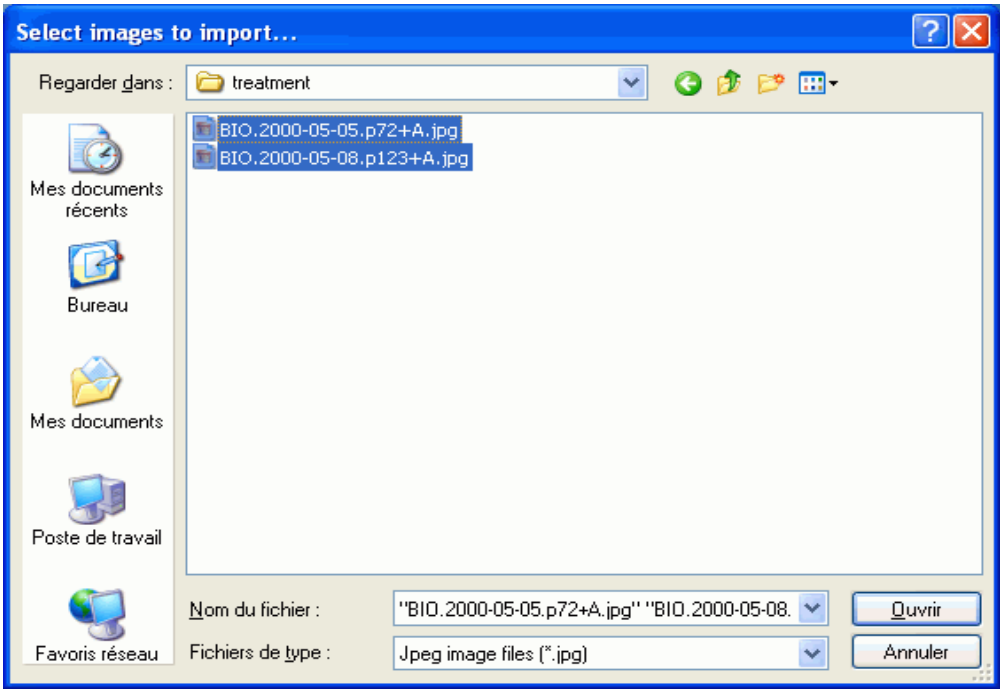
4. You should have something like this (without the treatment subdirectory):

10 For instance, you filter you sample on a 1000µm sieve and apply different dilutions for the 'large' fraction and the 'small' one. Just decide to call your large fraction 'A' and your small fraction 'B'. Now, if you make three pictures for each fraction, PP will be A1, A2, A3, B1, B2, B3, respectively for the six pictures related to the sample.

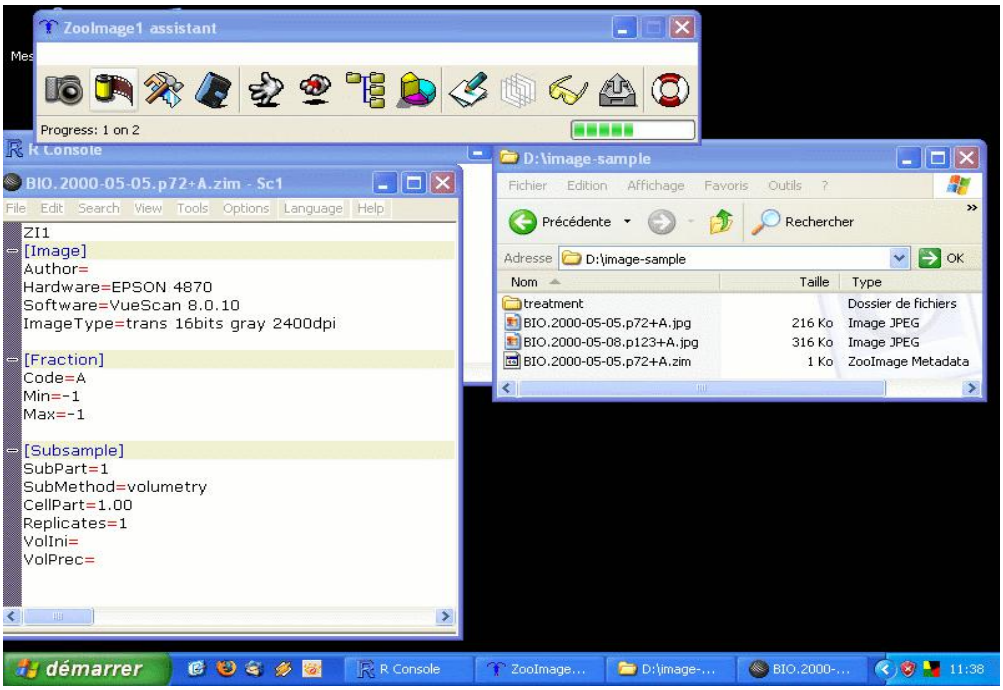


Now, click on the second button on the toolbar, the one with the following icon:

Zoo/PhytoImage asks you for the images that should be imported. Select both images.

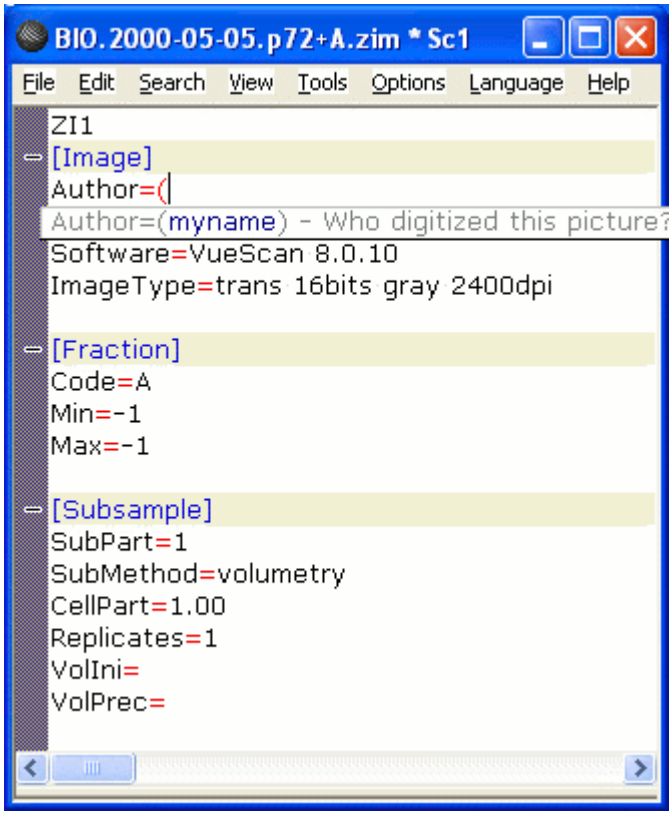


It is then supposed to check that image formats and names are correct, and possibly propose to change or convert them, but that feature is not implemented yet. It then checks if metadata files (files with .zim extensions) are associated. Since you did not copy these files with your images, they are not found and Zoo/PhytoImage creates them. It also displays their content in the built-in metadata editor (Sc1), each file in turn.

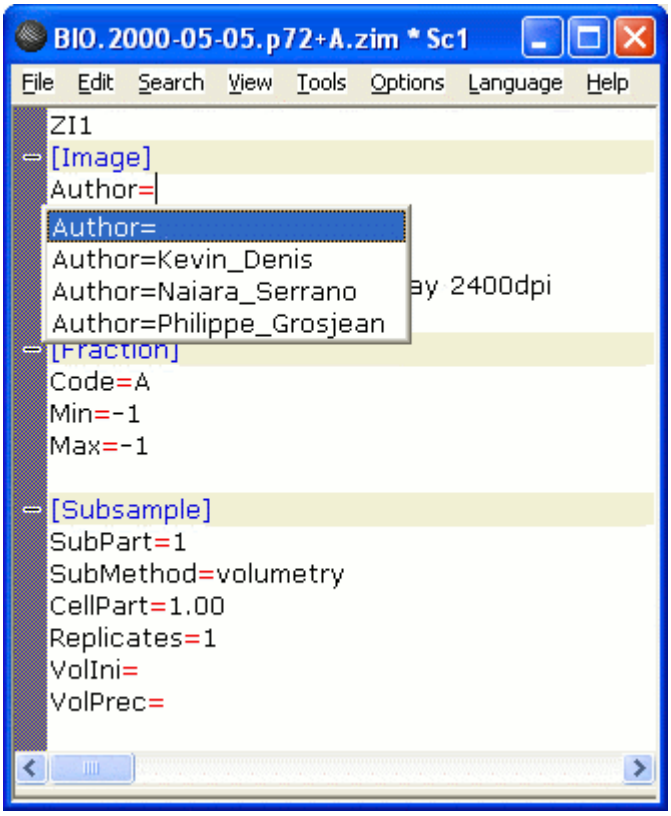


You are supposed to fill these data correctly. Here is how you can use the metadata editor:

- It is a plain text editor. Type your text as usual.
- You don't have to save your changes. When you close the window, changes are automatically saved and Zoo/PhytoImage switches to the next file.
- If you want help about a given entry, type an opening parenthesis just after the equal sign. You got a tip with information about that entry.



- You can also have a list of proposition for that entry. Place the caret just after the equal sign and hit `Ctrl+I`. A list displays default entries. **This way of entering metadata should be preferred, because it avoids typing errors!**



- If needed, you can enter additional metadata. Just use the `key=value` syntax. If you want to create another topic, enter it in a separate line in square brackets like `[topic]`.

Zoo/PhytoImage does not create separate .zim files fo **each** picture. It only create separate .zim files for each fraction of each sample. So, if you have a lot of pictures related to the same sample and fraction (this is likely to be the case if you work with FlowCAM or VPR images), you just have to fill one .zim file for all of them!

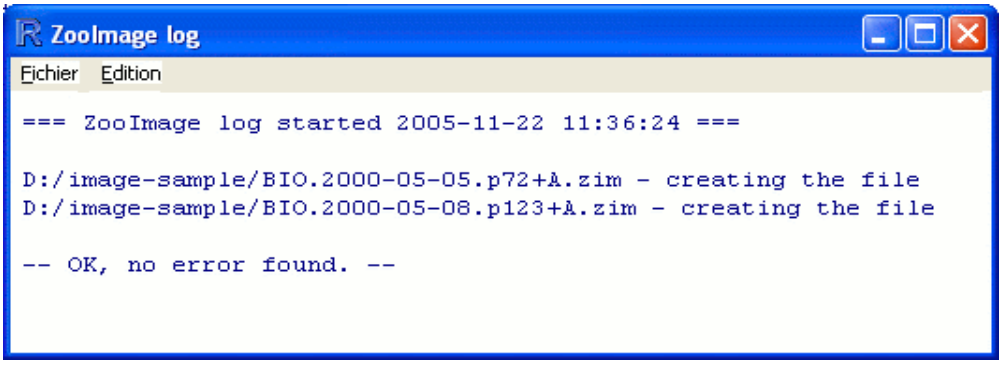
> *You can customize both the default entries in the metadata and the list of proposed values are customizable. Just edit those files: \bin\MetaEditor\templates\default.zim and ...\zim.api from the base Zoo/PhytoImage directory. Note that you cannot use spaces in the list of suggestions in the zim.api file. Use the underscore instead. ZooImage will convert it in a space in due time. So,* `Author=Alfred_Hitchcock` *should be entered in the list of possible completions, instead of* `Author=Alfred Hitchcock`.

Meaning of the metadata entries

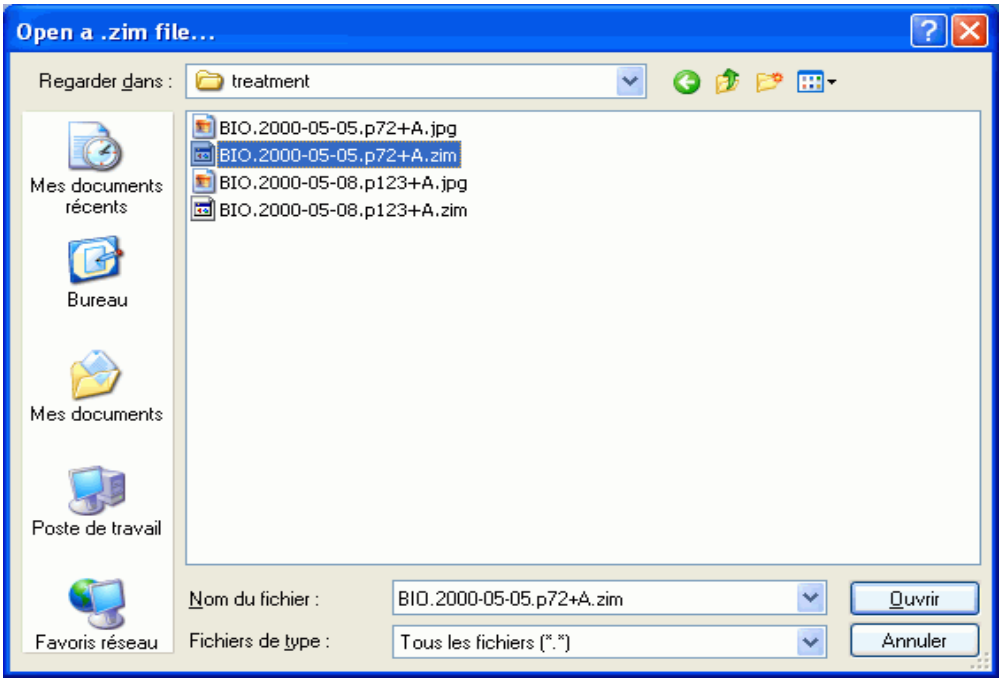
Entry	Topic	Explanation
ZI1	-	This is not an entry. It just tells it is a ZooImage1 file.
Author	Image	Who digitized the picture?
Hardware	Image	Device used to digitize the picture.
Software	Image	Acquisition software and version.
ImageType	Image	Type of image. For instance `trans 16bits gray 2400dpi` means image acquired in transparency of 16bit gray scales and a resolution of 2400dpi.
Code	Fraction	The same fraction identifier as in the file name `A`, `B`, etc.
Min	Fraction	Minimum mesh size used to retrieve this fraction in µm. Use `-1` if none.
Max	Fraction	Maximum mesh size used to retrieve this fraction in µm. Use `-1` for none.
SubPart	Subsample	Part of the sample that was digitized. If the picture contains only 10% of the organisms in your sample, `SubPart=0.1`, for instance.
SubMethod	Subsample	Method used to get the part (`volumetry`, `Motoda`, `Falsom`, etc.)
CellPart	Subsample	Part of the cell containing the plankton that was actually digitized .

Replicates	Subsample	If you did replicated images with the same protocol for that fraction, how many replicates do you have? Note: ZooImage with average results among raplicates instead of summing them.
VolIni	Subsample	The volume of seawater that was sampled in m^3. This is required to calculate abundances and biomasses per m^3.
VolPrec	Subsample	The precision on the sampled volume estimate in m^3. This will be used for error evaluation (not implemented yet).

At the end of the 'import' process, you should get a report in a `ZooImage log` window that pops up. It should look like this:



Take care that you should have the `- OK, no error found. -` message at the end of the log. For only two pictures, this log is not very useful, but imagine the advantage of logging individual error if you import thousands of pictures and when all the checkings (file names, formats, etc.) will be activated! Now, you `D:\image-sample` directory should look like this:



7. Process images

To process your images, use the menu entry `Analyze → Process images...`, the shortcut `Ctrl+J`, or click on the third button in the toolbar.

Zoo/PhytoImage will now switch to ImageJ, a free image processing software. Before doing so, a dialog box proposes to close Zoo/PhytoImage. Whether you can leave Zoo/PhytoImage open at the same time as ImageJ or not depends on the amount of RAM memory required by the image process, compared to the one you got on your computer. The small example pictures we are dealing with do not require much RAM. So, if you have something like 512Mb on your machine, you should be safe to keep both Zoo/PhytoImage and ImageJ opened simultaneously. If you analyze very large pictures, you should close Zoo/PhytoImage and all other running programs **before** starting your image processing in ImageJ. As an example, 16bit gray pictures of 60 million pixels (for instance, 10000×6000 pixels) require 900Mb of RAM allocated to ImageJ[11]. You need at least 1Gb of actual RAM in your computer for dealing with such images.

> *The maximum amount of RAM you can allocate to ImageJ is system dependent. On 32bit system, do not try to allocate more than 1.6Gb to ImageJ[12], or the program will crash! Of course, you need at least 2Gb of actual RAM in your machine to use that maximum value. Although we did not tested the* `Gray16bits 2400dpi` *plugin with images larger than 10000×6000 pixels, the maximum allocatable RAM value should work with images of about 100 million pixels. Thus, currently the largest 16bit gray images you can deal with in ImageJ is something like 10000×10000 pixels[13]. At 2400dpi, it is a little bit less than 10x10cm of cell size. If you have larger cell area, just take several separate pictures and both ImageJ and Zoo/PhytoImage will take them into account (you just loose measurement on objects that are cut at the edges of the composite images). On 64bit systems, you don't have these limitations and should be able to analyze much larger pictures.*

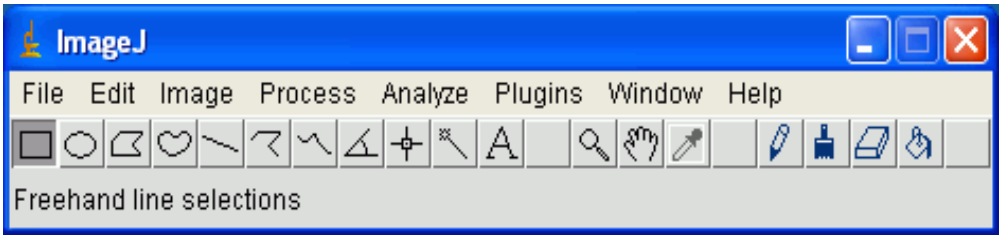
Start now ImageJ by click on the third button

Click `OK` in the dialog box and the ZooImage assistant window is minimized and replaced by the equivalent ImageJ main window as:

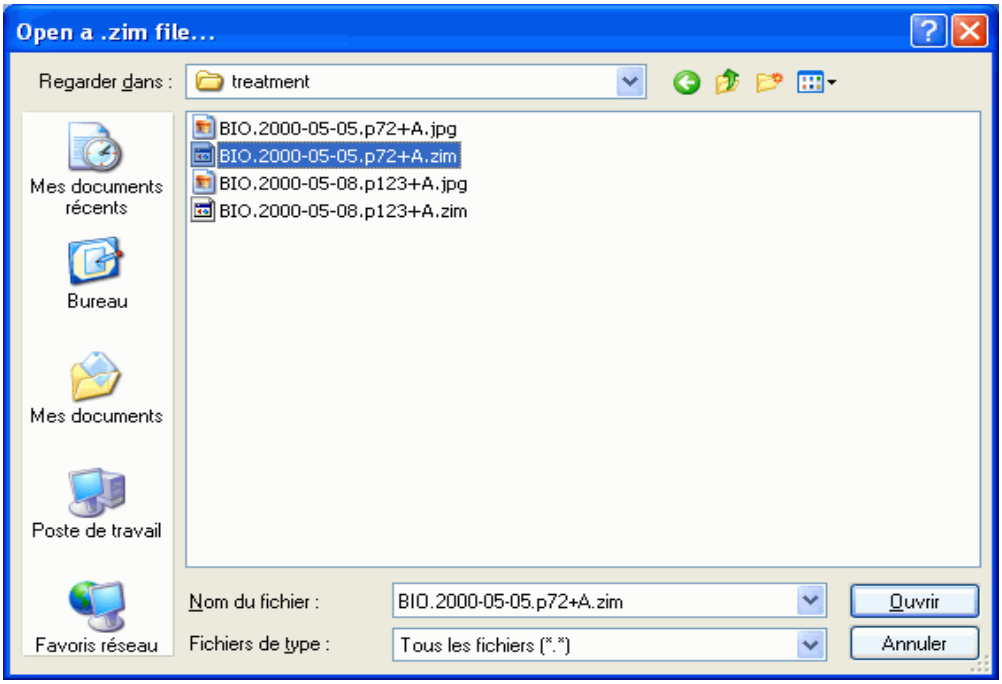
11 The current configuration of ImageJ installed with Zoo/PhytoImage is to allocate a maximum of 900Mb to the program.

12 You can change this value in ImageJ with the menu entry `Edit → Options... → Memory`. You have to restart ImageJ for the changes to take effect.

13 With a different treatment, one could process larger images, but silhouette detection would be less accurate and there will be no background elimination.



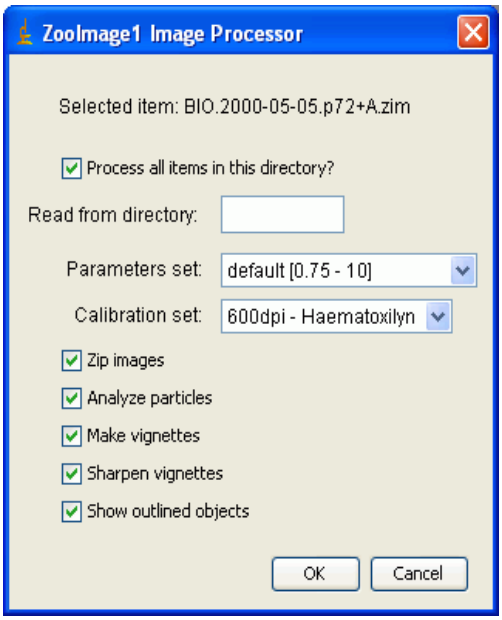
Zoo/PhytoImage plugins are collected together in the menu `Plugins → ZooImage`. For our images, we have to select the `Color24bits 600dpi` plugin. The plugin first asks you to select a .zim file. **Do not select on image file here**.



The reasons why you have to select the .zim file instead of the corresponding image are:

- We are sure you have metadata associated with the image(s),
- As explained here above, you could have several images for the same sample/fraction. The plugin will process **all** images associated with the selected .zim file, not only one. In the example, we have only one image for each .zim file, but that feature is designed with FlowCAM or VPR images in mind.

  You then have a dialog box with parameterization of your process:



- The name of the selected .zim file is displayed.

- You can **process all items in this directory** (all images that have associated .zim files), or only that one [*keep this checked now*].

- You can optionally **read images from a different directory**. This function is useful if you saved your large images on DVDs or external disks. You just have to copy the small associated .zim files in your process directory and you point to the directory that contains the images on your DVD [*leave this blank now*].

- The **parameters set** drop-down list allows you to select alternate configurations. Currently, alternate configurations are hard coded in the plugin, but users will be able to edit them freely in future versions. Parameters set defines minimum and maximum particle size to consider, which measurement is done, which threshold is used for separating particles from background, etc. [*leave the default value now*].

- The **calibration set** drop down list is similar to parameters set, but define calibration data, i.e., pixel size and calibration curves for grayscales and/or color channels, possibly depending on the lighting, staining of the sample, etc. [*leave the default value now*].

- **Zip images** rewrites the pictures in a zip-compressed TIFF format. This is not useful for JPEG images because they are already compressed. [*So, uncheck this option now*].

- **Analyze particles** do the measurements on the particles after processing the images [*leave this option checked now*].

- **Make vignettes** estract small images for each identified object, called 'vignettes' in Zoo/PhytoImage's terminology [*leave this option checked now*].

- **Sharpen vignettes** optionally applies a "sharpen" filter on the pictures in the vignettes. This often enhances the quality of the vignettes, but is not necessary for some kinds of pictures [*leave this option checked now*].

- **Show outlined objects** displays a composite image with the detected object outlines superposed to the grayscale image. This is a very useful diagnostic to determine if segmentation and detection of the objects was correct [*So, leave this option checked now*].

> *The **show outlined objects** option works only for the last picture processed. so, either uncheck **process all items in this directory**, or be prepared to wait for the last picture to get this diagnostic image! You should zoom in the image (`Image → Zoom → 100%` entry menu) and pan it by selecting the hand button and dragging the image content in the window to best see the result.*

When you start the process by clicking `OK` on the dialog box, ImageJ do the following work:

- It opens a `Log` window and reports its activity in it.

- It opens each image in turn, process it, and possibly measure particles and extract vignettes. You can follow the process on the screen. Note that a scale bar is added in the top-right corner of each vignette for convenience.

- It possibly displays the outlined objects of last picture if it was requested. also, the last table of measurements is left open for inspection.

> *If the process failed somewhere look if your images are of the right type, if they are not too big for the RAM memory allocated and if the correct plugin, parameters set and calibration set where selected. Look at the log file and the images produced in the `_work` directory to help you track the problem.*

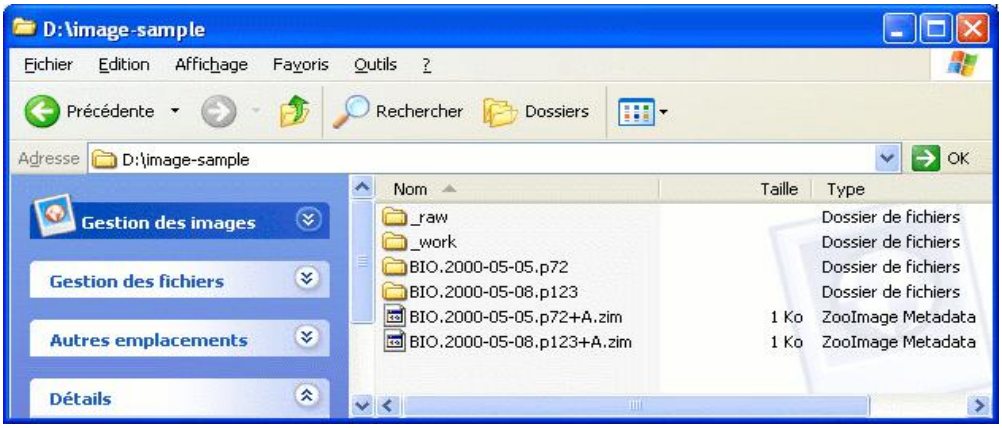
Always check the log file, seeking for errors, and take the habit to inspect outlines objects and table of measurements, at least, for the last image in your series. The plugins created several subdirectories in your process directory:

1. A **_raw** subdirectory contains raw images that were successfully processed.

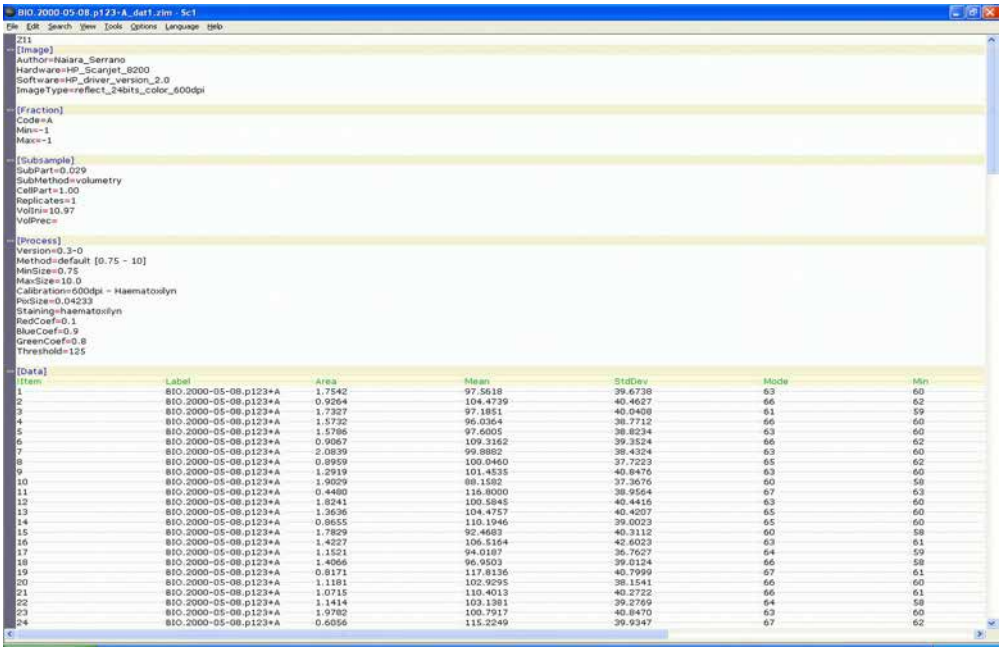
2. A **_work** subdirectory contains temporary intermediary images left there for further inspection and diagnostic. Once you are satisfied with the treatment, you can delete the whole `_work` subdirectory to save

space on your hard disk.

3. One separate subdirectory for each sample, bearing the sample name (everything before the + sign in the images/.zim file names. This subdirectory contains all the vignettes for the sample (possibly combining various images and/or fractions) and `_dat1.zim` file(s) with metadata plus measurements for each image.



Here is how a `_dat1.zim` file looks like. Notice that you have two new sections appended at the end of your metadata: `[Process]` that gives information on the processing parameters used and `[Data]` with a table of measurements don on each particle.



Once you have done with your image processing, you can close ImageJ and return to Zoo/PhytoImage (either restore the ZooImage assistant window, or restart the program, depending if you minimized or close it when you started ImageJ).

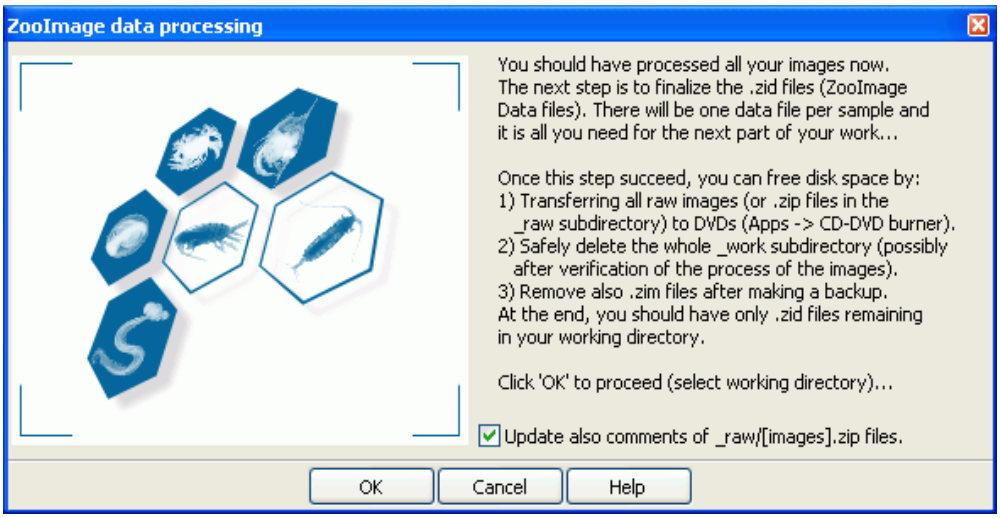
8. CREATE .ZID FILES

To finalize your images import/process, you must now build .zid files. In the ZooImage assistant, use the menu entry `Analyze → Make .zid files...`, the shortcut `Ctrl+Z`, or click on the fourth button in the toolbar.

The first part of your analysis (import and process of your images) is almost done. You have now to create the **.zidb files**. These are special *ZooImage DataBase* files that contain all you need for the rest of the analysis, but saves as much disk space as possible[14]. Those .zidb files represent a convenient solution to keep all required data of even long series (thousands of samples) on a standard hard disk of 100-300Gb. In such a case, high-resolution raw images consume litterally **terabytes** of space and cannot be all kept on the hard disk at the same time! Just process your series bit by bit, and backup raw images from time to time to solve the problem.

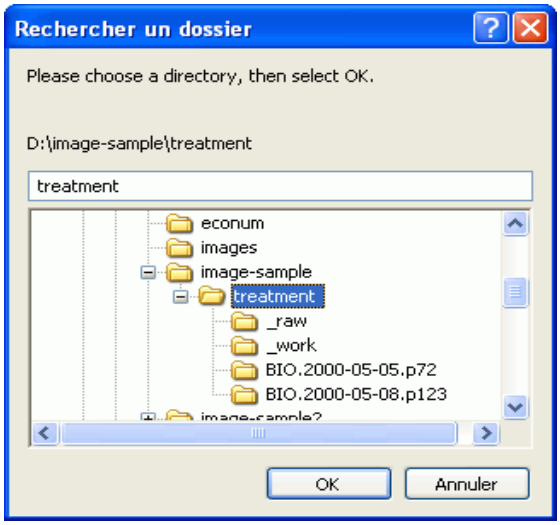
Now, click on the fourth button in the ZooImage assistant:

This shows the following dialog box:

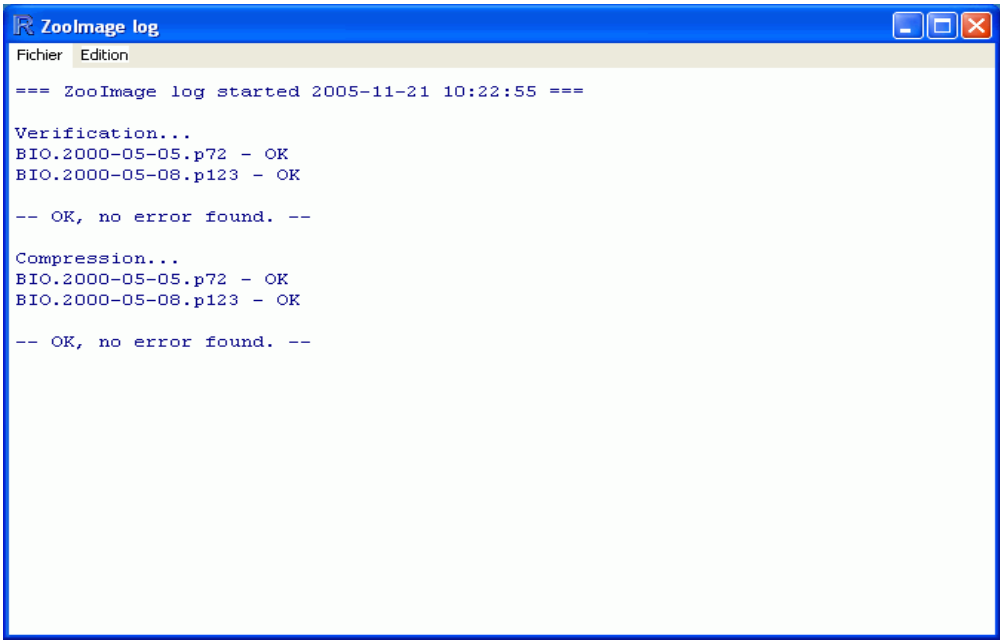


Instructions should be clear. By clicking `OK`, you compute .zidb files for your processed samples. The option **update also comments of _raw/[images].zip files** add .zim data as comments to zipped image files (if you selected that option in the process). [*Since we did not zipped images, we should uncheck that option now and click* `OK`]. You are prompted for a directory where treated data are located; give you working directory (`D:/image-sample/treatment`).

14 You reach easily a compression factor close to 100 or more, starting with uncompressed 16bit TIFF images: 6 times 120Mb of raw images, that is, 720Mb compresses to 4-10Mb in the corresponding .zidb files!



Zoo/PhytoImage computes .zidb files and issues a report at the end of the process. For convenience, it first quicky checks if all files are corrects. Stay in front of the computer during checking. Once it succeed, you can take a coffee break during the process that can be long if you processed a lot of samples. **Make sure there is no error reported once the compression is done.**



Cleaning the hard disk at the end of the process

Once all your .zidb files are created, it is time to save space on your hard disk. You should do the following from time to time:

1. Delete the `_work` subdirectory, once you are confident with the image processing of all your samples.

2. Back up your original images (in the `_raw` subdirectory) + the corresponding .zim files on DVDs, external hard disk, tapes, etc. **Always back up your raw image files: you would**

perhaps have to redo your analysis with a better algorithm in the future… and .zidb files do not contain required data for reprocessing the images! Once it is done, delete the `_raw` subdirectory and all remaining .zim files in the treatment directory to free disk space.

3. Check this: in your processing directory, you should only have .zidb files remaining (one per sample, no mather how many pictures you had for each sample) and no additional subdirectories or files (except, perhaps, .zis files and manual training sets if you already build them, see later in the manual).

9. MANUALLY CLASSIFYING VIGNETTES

In order to train the computer to (semi)-automatically recognize zooplankton taxa on the basis of images measurements done in Zoo/PhytoImage, you have to make a manual training set. In Zoo/PhytoImage, you can have a relatively complex organization of the different groups (taxa, ecological groups, or any other grouping of the plankton that suits your needs) in a **hierarchical tree**. Hence, you have relationship between the groups (for instance, *Sapphirina intestinata* and *Sapphirina ovatolanceolata* are collected together in the *Sapphirina sp* group. *Copilia sp* and *Sapphirina sp* form your *Sapphirinidae* group. *Sapphirinidae* together with *Oncaeidae* and *Corycaeidae* (which contain also corresponding subgroups) are collected together in the *Poecilostomatoida*, etc. Up to the top group called *Copepoda*.

You can also decide to make other groupings, like ecological groups, or even mix the styles. You are here 100% free of the groups you create, but there are a couple of constraints: (1) make logical hierarchy of your groups and subgroups; (2) keep in mind the parameters (abundances, biomasses and partial size spectra) that you want to calculate on these groups; (3) make only groups where you can actually classify vignettes with a reasonable accuracy solely on the visual inspection of these vignettes; (4) it is useless to make groups for very rare items –you need at least ten to fifteen example vignettes in each group in your training set, 30 to 50 is even better–; (5) ultimately, the most pertinent grouping is the one that the computer can actually discriminate with a reasonable accuracy!

You have to classify all kinds of items. Even those you are not interested in (may be, bubbles, marine snow, phytoplankton if you are only interested by zooplankton, etc.). Indeed, you have to recognize those items to elimine them from the countings… and you need a group in the training set for that!

You don't need to classify **all** vignettes. When you have about 50 items in a group and you think it is well representative of the overall variability in shapes of that group, you don't need to add more vignettes. Also, fuzzy objects, unrecognizable ones, multiple or part (except for VPR images), rare taxa, etc. do not need to be classified. Abberrant individuals which are not likely to occur often in your samples should be eliminated too. You have a special top group named '_' in the hierarchy for all these items. **All vignettes in the '_' top group or any of its subgroups will not be considered in the training set.**

For biomasses calculations, it could be useful to further split groups depending on the orientation of the animals: conversions formulas could be different for 'lateral' or 'dorso-ventral' views of the same animals. Make subgroups for them, if you want to take advantage of these different conversion formulas. Ex: *Oithona sp lateral* versus *Oithona sp dorsal*.

Make sure you use **unique names** for **all levels** of all groups. Do not use a classification like *Nauplius* subgroup in *Copepoda* and *Nauplius* subgroup in *Malacostraca*. Indeed, the program will manipulate groups

independently for some treatments and how to differente *Nauplius* from *Nauplius* then, when you don't use the grouping hierarchy? Correct presentation should be: *Copepoda nauplius* in *Copepoda* versus *Malacostraca nauplius* in *Malacostraca*.

Zoo/PhytoImage does not check uniqueness of group names for the moment : you have to care about this by yourself!

9.1. Preparing a manual training set from .zidb files

To install files and directories required for making a manual training set, use the menu entry `Analyze → Make training set...`, the shortcut `Ctrl+M`, or click on the fifth button in the toolbar.

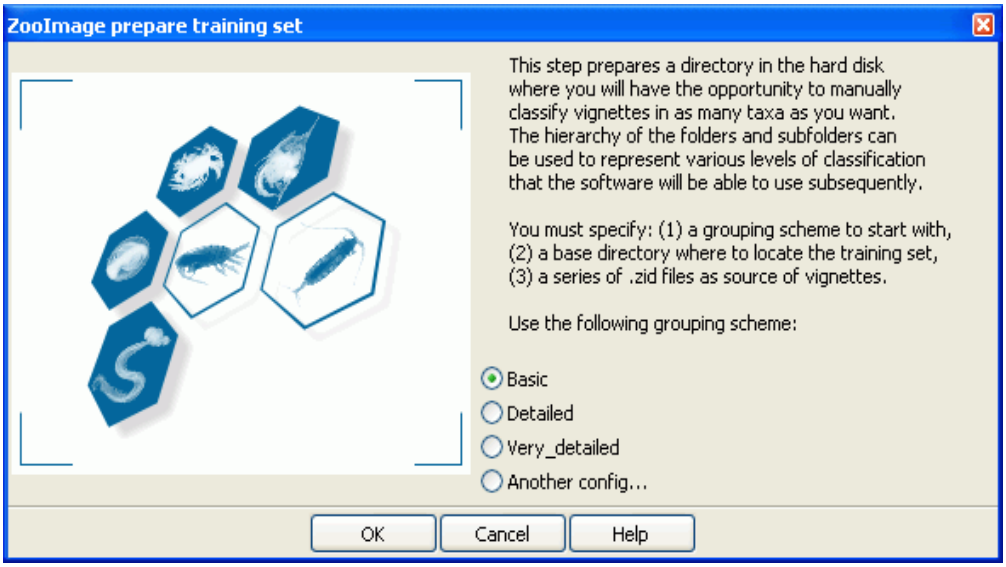
You must first decide which samples you will use in the training set. Select a couple of samples (i.e., a couple of .zidb files) that are representative of the whole variability in your series. Choose samples that span on the whole time scale (possibly several years) and the whole considered geographic area. Choose also samples collected at different seasons, if this applies. Depending on the number of groups you want to make you will need a couple of hundred vignettes to a couple a thousands of them (maximum 10 to 20.000 items for very detailed training sets). Knowing the average number of vignettes you have in a sample, you can determine how many samples you need (usually a couple a tens).

If you want to make your training set rapidly, starting with a long historical series already available in your laboratory, it could be interesting to first choose the representative samples that will be used in the training set and digitize them in priority. That way, you do not have to wait that all the samples in the series are digitized and processed to make your training set! Also, if different people are digitizing the sample (technicians) and making the training set (specialized taxonomists and biostatisticians), you could have work done in parallel once the few samples required for the training set are digitized.

To experiment with our example images, **create first an empty directory dedicated to this training set**. You can create it anywhere on your hard disk, but if you create a subdirectory in your process directory (`D:/image-sample`), make sure you **prepend its name with an underscore** (like `_train`, for instance). That way, ZooImage will ignore it in further processing of your images. Of course, do not use `_raw` or `_work` for the name of this subdirectory, since these names are reserved for the image processing treatment (see importing images). *[Create now an empty `_train` subdirectory in you processing dir].*

Now, click on the fifth button on the ZooImage assistant toolbar:

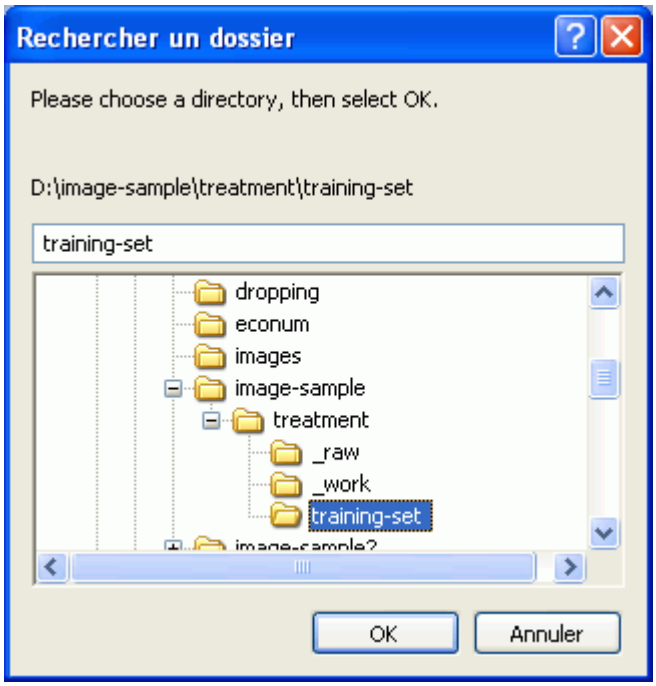
A dialog box with instructions appears on screen.



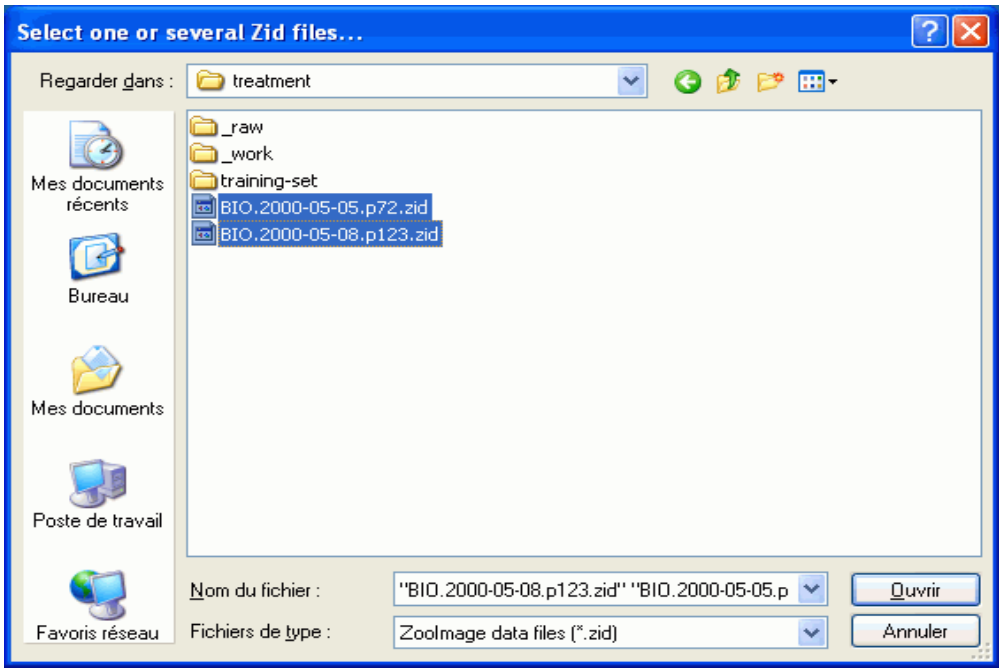
You have to select a config file. That file will create the initial hierarchy of groups as a series of subdirectories in your training set folder. You can choose "Basic", "Detailed" and "Very detailed", or select a different config file with a .zic extension. *[Choose now the "Basic" configuration and click* `OK`*].*

> *Initial groups config files are customizable, and you can save other ones everywhere on your hard disk. Just respect their (simple) syntax and save them with a .zic extension.* `Basic.zic`, `Detailed.zic` *and* `Very_Detailed.zic` *files are located in the subdirectory* `\bin\R\R-2.2.0\library\zooimage\etc` *of the ZooImage root dir (usually* `C:_Program files\ZooImage`*).*

You now have to select the base **empty** directory where you want to install files and folders for your new manual training set:

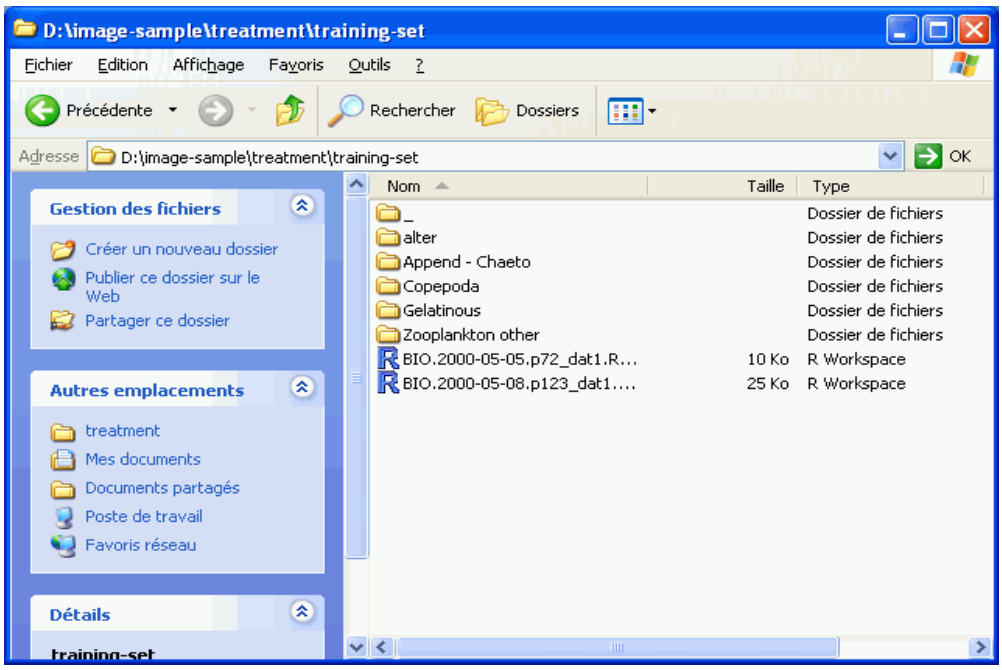


Select your `D:\image-sample\treatment\_train` directory. finally, the program asks you to select the .zidb or .zid files corresponding to the samples you want to use to build your manual training set (they must be all located in the same directory). *[Select now our two example samples* `BIO.2000-05-05.p72.zid` *and* `BIO.2000-05-08.p123.zid`*)].*

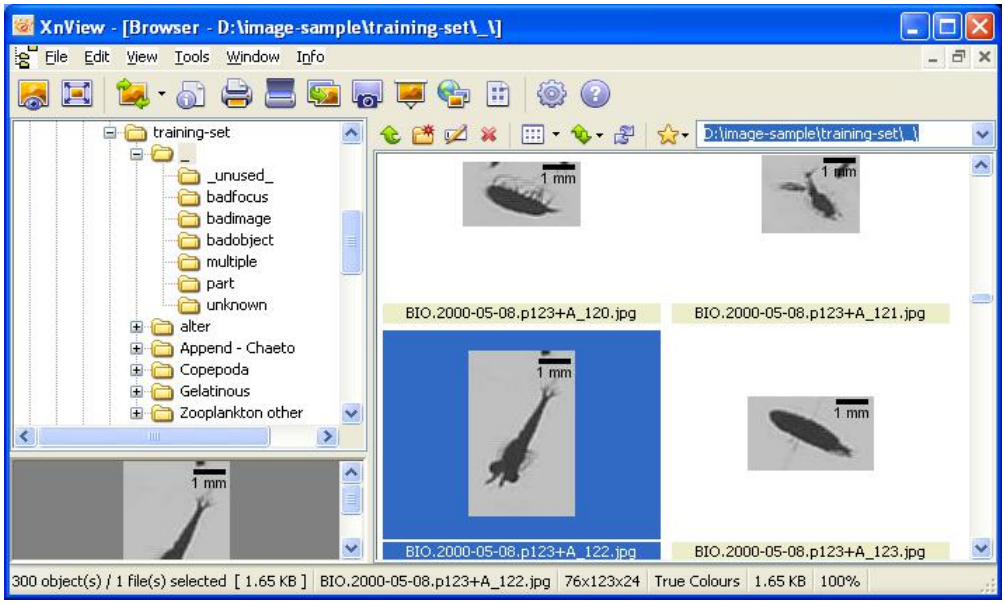


Zoo/PhytoImage creates required folders, extract data about these samples (`_dat1.Rdata`) files in the training set's root directory, and places all corresponding vignettes in the _ subdirectory. A log file indicates if there were errors creating these files and folders. At the end of the process,

Zoo/PhytoImage starts **XnView** in the _ subdirectory. If you inspect the files on your computer, you should see something like this:



Now, switch back to XnView.



XnView is a free software for non commercial use[15]. It is both an image viewer/manager and an image converter. Here, we only use its ability to work with thumbnails of images in directories and manage them. We don't use all its features!

Depending how you organize XnView windows, the browser has a tree of directories, a thumbnail of images and a preview panel for the currently selected picture. You can change XnView configuration in `Tools → Options...`. If the directories tree is not visible, select `View → Folder`

15 If you are in a private company, you will have to get a license for XnView before you can use it!

`Tree`. If you do not have a thumbnail view in the XnView browser main window (you can have an icon list, or tabular view of the files as well), select `View → View As → Thumbnails`. **Both the folder three and the main window in thumbnail mode are required for the rest of the work**.

Now, begin to classify the vignettes manually by moving them in the corresponding directory in the tree by drag&drop with the mouse. It is easier to move vignettes first in top directories (all copepods in *Copepoda*, all appendicularians and chaetognathes in *Append - Chaeto*, etc.). Then, you open the *Copepoda* subdirectory and classify vignettes from there to deeper levels (*Gymnoplea* or *Podoplea*), etc. Of course, this work should be done by, or with help of trained taxonomists.

> *It makes sense to ask different taxonomists to classify the **same** vignettes independently, so that you can check unmatching results and build a consensus that is supposed to bear less errors than a single manual training set. **We may add tools for analyzing and building consensus training sets in the future in ZooImage, but it is not the case yet in the current version.***

You are not restricted to the groups and subgroups already made. You can freely modify the structure of the tree; change directories, add or delete other ones. In the tree panel of XnView browser, you right-click in a directory and select `New Folder`, `Delete` or `Rename` entries to rework the tree. Make sure all people that build the training set (or similar training sets) have the same perception of each group. Define clearly which kind of object should go in which group, print these directives and keep them on your desk for reference when you classify your vignettes.

Also, if you plan to build a consensus training set, collecting together independently trained data, or if you want to build similar training sets for different series, you must work in two stages:

- First define the **structure of the tree** with all concerned people and define clearly which vignette should go in each group. At the end of the process, it should be useful to have a definition file (with a .zic extension) off this reworked tree. Distribute this .zic file to all collaborators and ask them to make their training sets with the same tree **without modifications**.

- Second, build your manual training set with the tree and groups you just defined.

When you classify your vignettes, you should try as much as possible to classify them down to the most detailed subgroups. If there are many vignettes you cannot classify deeper than a certain level, although your tree has more detailed groups, it means that you were too ambitious in the level of details you want to reach in the tree. Rework your tree and eliminate problematic subgroups where you cannot classify those vignettes.

A final pass is required before you can use your training set: you must rework or eliminate rare subgroups were you have too few items in them (let's say, less that 8-10 vignettes). Two alternatives:

1. Merge them with other subgroups, making less detailed groups, but with enough vignettes.

2. Decide not to include these rare groups in the training set. Keep them, but move the directories to the _ top folder (remember that this _ top folder contains all subgroups and vignettes that will be ignored in the classification).

Never forget that including rare groups in your training set will only have the consequence to reduce the total identification accuracy and the accuracy of other, major, groups –due to missclassification of other items in these rare groups–. The only (exceptional) situation where you would like to keep a rare group is when you are specifically interested by tracking target rare organisms in your whole set of images.

When you rework your groups, make sure you do not have also **too many vignettes** in the most abundant ones. It is useless to have hundreds or thousands of items in one group. If it is the case, randomly eliminate vignettes (you can create the same group under the _ top folder and move the vignettes there, so that you keep them correctly classified but do not take them into account in the learning stage). Consider that if you have more than 50 vignettes in a group, you can begin to eliminate randomnly items down to 50 images per group.

Making a manual training set is a difficult and time-consumming task !

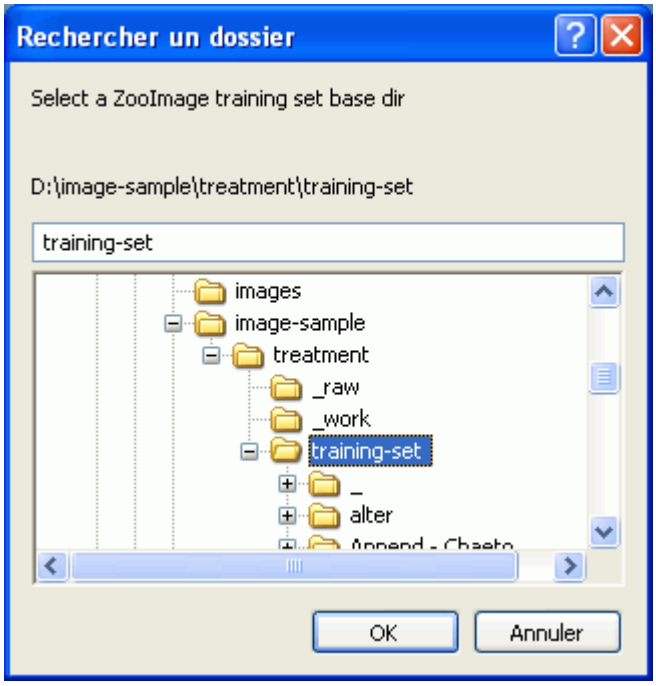
You have an example training set installed with Zoo/PhytoImage. You can inspect it in XnView, or even read it in Zoo/PhytoImage, if you like. This example training set is located in the `\examples\_train` subdirectory of your Zoo/PhytoImage folder (`C:\Program Files\ZooImage` by default on Windows). This training set was build using 29 samples... thus more than the two available in you `\examples` subdirectory. Look at it to have an idea on how you should balance items in the different groups.

9.2. Reading a manual training from disk

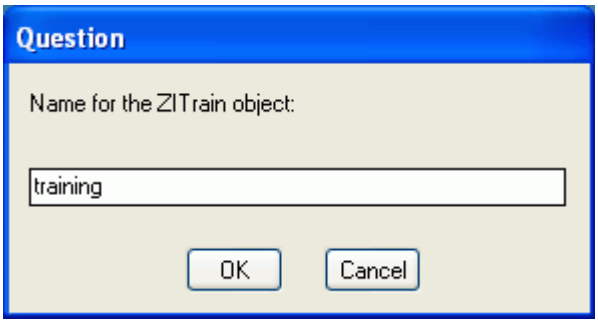
To read a training set from directories where vignettes were manually classified, use the menu entry `Analyze → Read training set...`, the shortcut `Ctrl+T`, or click on the sixth button in the toolbar.

Once you are satisfied with your manual training set (or after reworking it, guided by the inspection of the confusion matrix, see hereunder), you have to read it in Zoo/PhytoImage. Click on the sixth button on the ZooImage assistant toolbar:

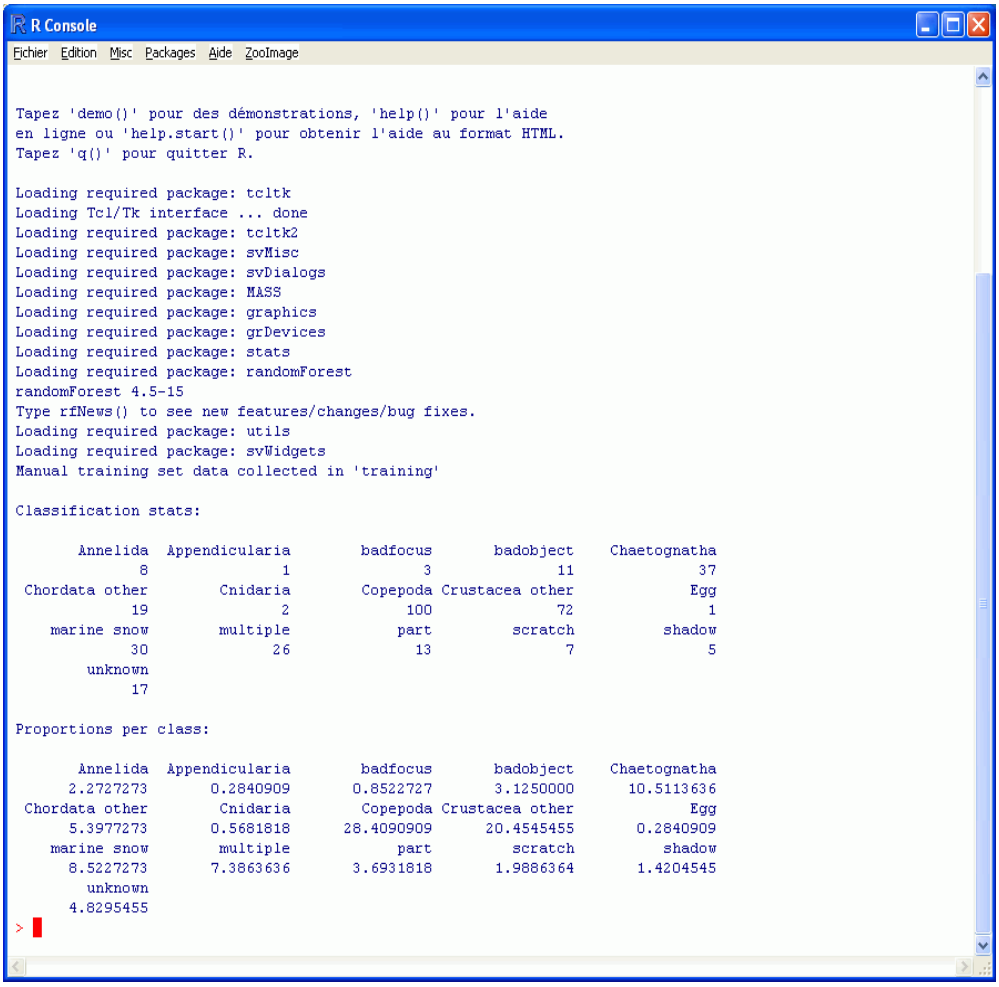
The program asks you for the top folder where your manual training set is located. *[Select now your directory, that is* `D:\image-sample\treatment\_train`*].*



You are then prompted for a name to give to the 'ZITrain' object that will be created:



[Call your object simply 'training' and click `OK`*].* Zoo/PhytoImage processes the tree (it takes a while for large training sets) and then displays basic statistics about your training set, that is, the number of vignettes in each group in the R Console window:



If you see that you have too much or too few items in some groups (like here, only one *Appendicularia* and a hundred *Copepoda*), go back to XnView and rework them before rereading your training set. Note that you have too few samples available in the examples for filling each group with enough items. For the rest of the demonstration, you can read the example training set installed with Zoo/PhytoImage as well.

10. Making and analyzing an automatic classifier

In Zoo/PhytoImage, classifier algorithms used range in a category called “machine learning”. Basically, you ‘feed’ the algorithm with example identifications together with measurements done on the same objects, and the algorithm learns how to recognize the groups according to the measurements. It is a very simple scheme, but it has proven efficient in many situations.

Many algorithms exist, and many are implemented in R over which Zoo/PhytoImage is running. The Zoo/PhytoImage dialog box gives access only to a couple of them. Moreover, in order to simplify the process, only default values are given for parameters. The solution you will obtain is, thus, often suboptimal.

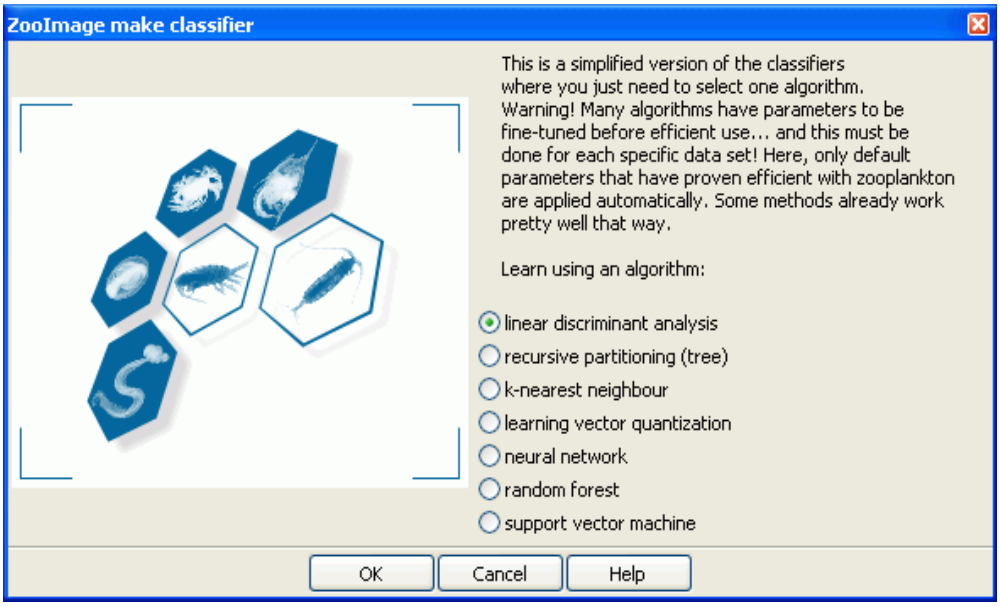
> *Many “machine learning” algorithms should be put in the “do not try this at home!” category. It means that you need a trained biostatistician to get the best from them and to analyze results to make sure they produce* ***consistent****,* ***reliable*** *and* ***accurate*** *identification of your plankton items. Everything was voluntary simplified in the Zoo/PhytoImage dialog box, just to give a flavor of these algorithm to everybody, and to allow a round-trip process of your data in an easy way.* ***Don’t be fooled by the apparent simplicity of the process using Zoo/PhytoImage dialog boxes!*** *For serious analyses, consider to fine-tune your classifier with a biostatistician that will use all the functions provided by R (he will program code in R’s native language, instead of just clicking with the mouse on a few options in the dialog box). There is no warranty on the results, and we would not endorse responsability of the consequences for false results published after using “uncertified” ‘toy’ classifiers!*

10.1. Training a classifier

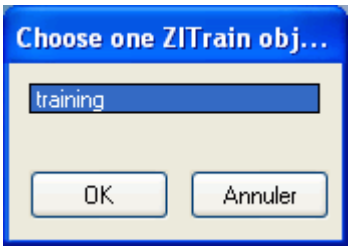
To train an automatic classifier with you manual training set, use the menu entry `Analyze → Make classifier...`, the shortcut `Ctrl+C`, or click on the seventh button in the toolbar.

Having a ‘ZITrain’ object in memory, you can now create a ‘ZIClass’ object, that is, an automatic classifier that learns how to recognize your zooplankton based on the examples you give in your manual training set. Click on the seventh button on the ZooImage assistant toolbar:

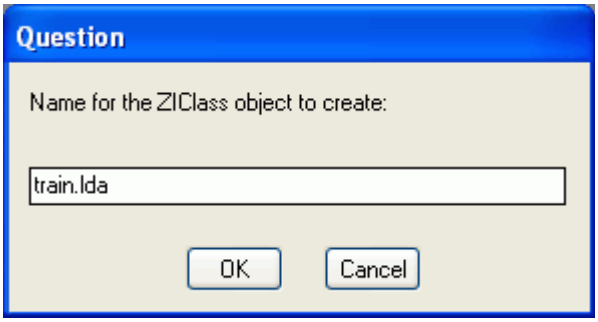
The next dialog box appears. It displays a warning message about the simplified learning phase and proposes a variety of “machine learning” algorithms to use.



Choose the one you want to use. *[Now we will use the simplest algorithm: linear discriminant analysis. select it and click* `OK`*]*. The program asks then which 'ZITrain' object he should use. You have probably only one training set in memory: the `training` object you just created.



[Select it and click `OK`*]*. The program then asks for a name for the 'ZIClass' object that is about to be created.



[Enter `train.lda` *and click* `OK`*]*. The algorithm learns how to recognized your zooplankton, based on your manual training set. When it is done, its performances are assessed using a method called "10-fold cross-validation". Then, a summary of the results (total accuracy and error by group) is reported to the R Console.

If you want, you can now test and compare other algorithms with the same training set. Also, if you notice that one or several groups have

consistently high errors, it means they are not well separated. Could you consider reworking them in the context of your analysis? Look also at the confusion matrix (hereunder) for further diagnostic tools.

10.2. Analyzing classifier performances

Further diagnostic tools are provided to study the performances of your classifier, use the menu entry `Analyze → Analyze classifier...`, the shortcut `Ctrl+N`, or click on the eigth button in the toolbar.

Having a 'ZIClass' object in memory, you should calculate a **10-fold cross-validated confusion matrix** between your manual and the automatic classification. The confusion matrix is a square matrix that compares all groups af the manual classification with all groups of the automatic classification. The number of items in each cell corresponds to the counting of objects. The diagonal (from top-left to bottom-right) corresponds to cells where both identifications are the same. This is thus the counting of **correctly** predicted items. All cells outside of the diagonal depict disagreement in both classifications. They are usually attributed to errors done by the automatic classifier, starting form the hypothesis that there is **no error** in the manual training set.

To calculate and display the confusion matrix for your classifier, click on the eigth button on the ZooImage assistant toolbar:

... and select your 'ZIClass' object in the dialog box *[You probably have only one, so, select it and click `OK`]*.

According to those analyses, you could decide to rework the groups that are difficult to separate in your manual training set, to reread it and train a new classifier with these optimized groups.

> *Other diagnostic tools are also accessible from the same dialog box in version >= 4. Experiment by yourself with it and discover the different diagnostic plots available here...*

11. MANIPULATING ZOO/PHYTOIMAGE OBJECTS

You don't have, of course, to read manual training sets and train classifiers again and again each time you launch Zoo/PhytoImage. You can **save** and **restore** existing objects. The `Objects` menu provides functions to do so:

- `Objects → Load` reloads one or several objects form a .RData file. The .RData file is a binary format that is used by R to save its variables. You can save several objects in the same file, and thus, you reload them all at once in this case. The .RData files can be exchanged between computers, even on different platforms (for instance, .RData files generated on Windows are totally compatible with those made on Linux/Unix or MacOS X).

- `Objects → Save` gives you the opportunity to select one or more 'ZIxxx' objects (Zoo/PhytoImage specific objects) present in memory, and to save them in a file.

- `Objects → List` prints the list of all Zoo/PhytoImage objects currently in memory.

- `Objects → Remove` permanently deletes one or several objects from memory. Consider using this command to free memory if you created a lot of objects that you don't need any more.

> *The .RData files are very convenient to exchange training sets and toroughly-tested classifiers with your colleagues. Everything is included in the .RData files to reuse those manual training sets and/or these classifiers on a different computer.*

R has a mechanism to save and restore automatically all objects in memory when you quit the program and restart it from the same active directory. When you quit R (`File → Exit` on the R Console, or click the close button of the R Console), you have a question: "Save workspace image?" that appears. If you click `No`, R exists without saving anything. If you click `Yes`, it saves the data in the file .RData in the current active directory (the one reported in the status bar of the ZooImage assitant window). It also saves the history of commands in a .Rhistory file in the same directory. The next time you start R, you can restore this .RData file if you like. **It is far better to use the `Objects` menu and selectively save/restore given objects than to systematically rely on this mechanism!** This way, you can also choose a meaningful name and directory where you store your data! So, if you save your objects using the `Objects` menu of Zoo/PhytoImage, you can systematically answer `No` to "Save workspace image?" when you quit R/ZooImage.

12. CALCULATING, VISUALIZING AND EXPORTING SERIES

This section supposes that you have already made .zidb files from your raw images (part I) and that you have a valid 'ZIClass' object in memory (part II) either that you just created, or that your reloaded from a .RData file.

Up to now, all treatments were made at the sample level. You never had more than one sample loaded in memory. A sequence of samples (or images) was always treated one-by-one by Zoo/PhytoImage, possibly reporting long processes in a log file, so that you can leave the software unattended doing the calculation and come back later to see the results (it seems that the koffee room will be more crowded that usual. **This is a feature!** Zoo/PhytoImage is not designed as a toy program that would be just able to calculate a couple of demo examples, but that will crash with an "out-of-memory" message with any serious dataset!

When we speak about serious datasets in the field of zooplankton image analysis, it really means:

- **Terabytes of raw images** to process[16]. Since you can backup your raw images and ZooImage cares about **storing highly compressed data in .zidb files**, you can really process very large series containing thousands, or even tens of thousands of samples with a simple PC. You can store, indeed, all these tens of thousands .zid files in a single hard disk of 200-300Gb[17].

- **Almost unlimited number of images per sample**, and also possibly, **complex samples processes** with replicates and with various separate fractions (different dilutions, or even, different processes for each fraction[18]). Zoo/PhytoImage will perform all the calculations: averaging replicates, adding data from the fractions after applying corrections for different dilutions, and rescaling results to express them per square meter of seawater automatically.

- **Almost unlimited number of objects in each samples** (the current limit is probably around a few hundreds of thousands items per sample, that is, the size of a matrix R can store in memory at once with a 2-4Gb RAM computer). This is not really a limitation because a few thousands to a few tens of thousands of objects are enough to evaluate the composition of a single sample, even for relatively rare taxa (with 10.000 objects measured in a sample, even rare taxa representing 1% of the sample composition will be represented by about 100 individuals).

Of course, processing time is in proportion with the size of the series, but Zoo/PhytoImage proposes various mechanisms to recover after a fealure

16 The only limitation is currently the maximum allocatable memory of 1.6Gb in ImageJ under 16bit systems that limits the size of **one** image to 100 millions of pixels. But 64bit systems, currently available today, overcome that limitation. Otherwise, Zoo/PhytoImage allows an almost unlimited number of images per sample.

17 A typical .zid file with 2000-3000 objects weights only about 5Mb.

18 For instance, using a Zooscan for the large fractions and a FlowCAM for the smaller ones.

to process a sample, and the error is reported in the log file. So, it is possible to spot the error and to reprocess only the guilty sample(s) later on[19].

So, OK, it seems relatively easy to accumulated huge amount of data using Zoo/PhytoImage. But then, how do we digest this huge quantity of information? The third part of the analysis deals with the calculation of biologically meaningful statistics that summarize each sample: abundances, biomasses and size spectra (total or per taxa). Hence, from the measurement of a couple of thousands objects in your images, you summarize the information into a few tens of numbers for each sample. All these numbers are then collected in a single table, with one line per sample. These tables are stored in 'ZIRes' objects (Zoo/PhytoImage Results). They are most suitable for the space-time analysis at the series level, which can be done in R/ZooImage directly, or you can export the tables to analyze them in another software like Matlab, for instance.

12.1. Creating and documenting a series

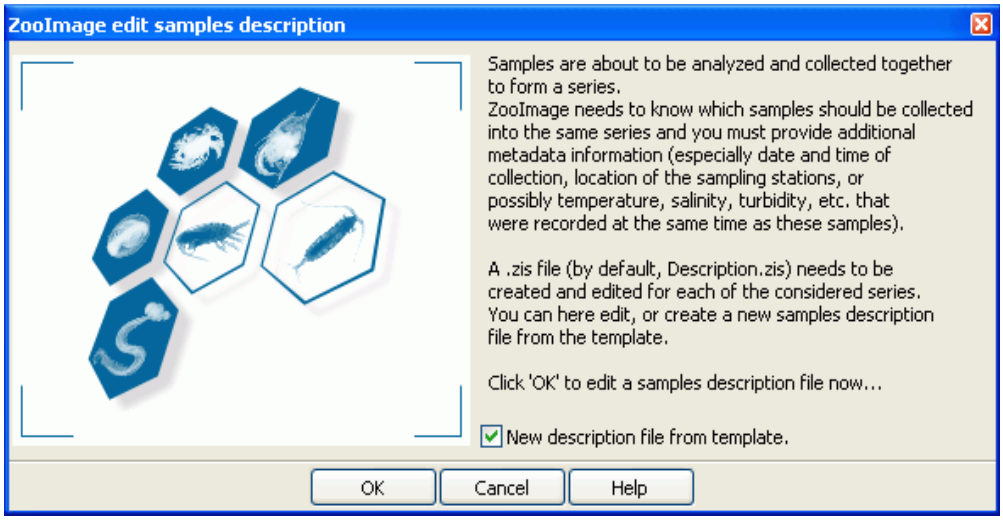
A series is a collection of samples plus a few additional metadata. To edit a series description file (.zis file), use the menu entry `Analyze → Edit samples description...`, the shortcut `Ctrl+D`, or click on the nineth button in the toolbar.

Until now, your .zid files had independent lives, totally ignoring each other. It is now time to tell to Zoo/PhytoImage which .zid files you want to collect together in a space-time series. This is done by editing a samples description file with a .zis extension. You can create as many .zis files as you like, making thus different series (for instance, a variation in time at a single station for one series; a spatial coverage of the area at a given time for another series, etc.).

[As an illustration of this principle, you will create now a mini-series, collecting together the two example samples we are analyzing]. Click on the nineth button on the ZooImage assistant toolbar:

The following dialog box appears with an explanatory message and a single option:

19 Note that Zoo/PhytoImage does not have yet a mechanism to incrementally add data to a 'ZIRes' object, but that mechanism is planned for future versions.



You can either create a new description file from the template (check the option), or edit an existing one (uncheck it). *[Create a new file, and thus, leave the option checked and click `OK` now].* After telling where you want to store the description file, the MetaEditor opens a template. You have to fill it in order to tell to Zoo/PhytoImage which samples are included in the series.

> *The .zidb files corresponding to all samples included in the series are supposed to be in the same directory as the .zis files themself.*

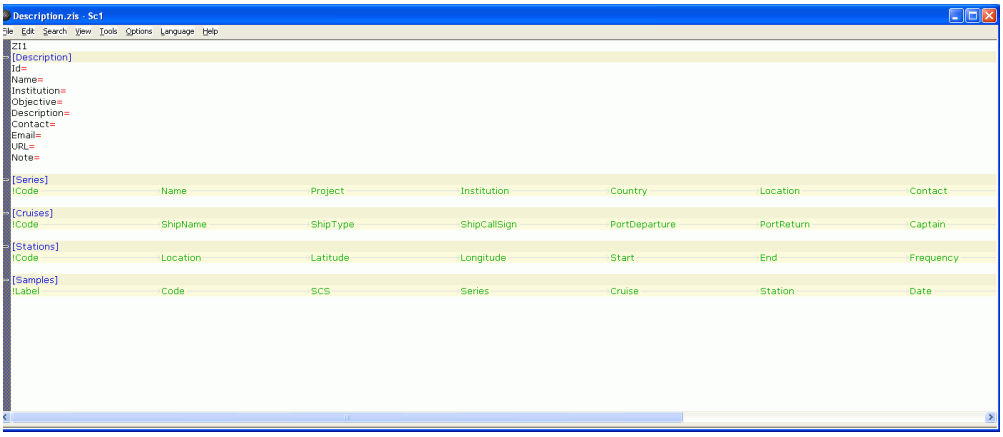
A complete description of data and metadata in .zis files is found in the annexes. You do not have to fill all field. Also, you can add additional keys, if you want. Major fields that you **have to fill correctly** are:

Key	Section	Comment
Id	Description	The short identifiant of the series.
Name	Description	A longer name for this series.
Description	Description	A short description of the series[20].
Contact	Description	The name of a responsible person of this series.
Email	Description	The email address of the contact.
Label	Samples	The complete label of the sample, as in the file names.
Code	Samples	A code for this sample.
Date	Samples	The data of sampling (in `yyyy-mm-dd` format).
Latitude	Samples	The latitude of sampling (in `+/-x.xx`).
Longitude	Samples	The longitude of sampling (in `+/-x.xx`).

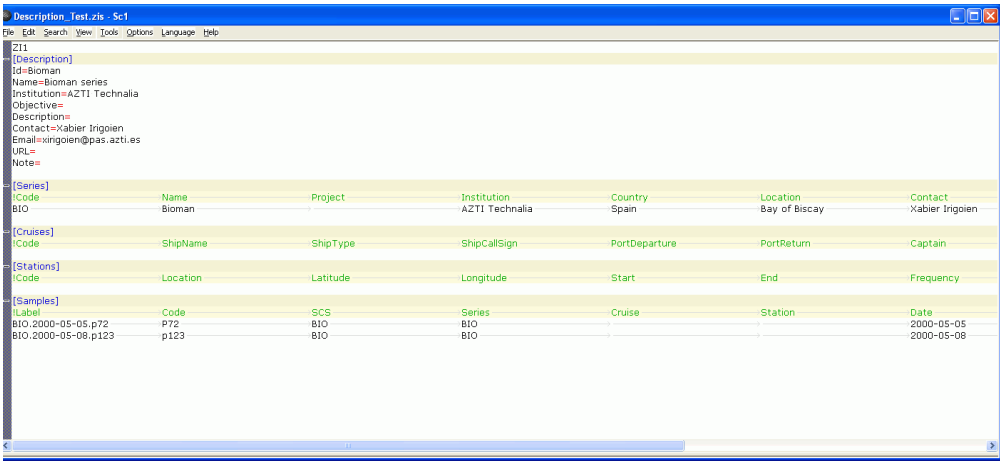
20 Don't overlook these metadata : they will allow you to calculate abundances and biomasses per water volume in the field, to locate your samples in space or time for further analysis, etc.

Considering the large amount of fields in this file, it would be convenient to reimplement it in a database. Any volunteer to reprogram this part of the software in an Open Source database like MySQL out there?

The MetaEditor displays the `Description.zis` template.



You have to fill it to obtain something like this:



You can just close the window, and your changes are saved automatically.

12.2. Calculating samples

To process all samples in one series, use the menu entry `Analyze → Process samples...`, the shortcut `Ctrl+S`, or click on the tenth button in the toolbar.

To process all samples in a given series, click on the tenth button on the ZooImage assistant toolbar:

... and select the corresponding .zis file *[Select your `Description.zis` file]*. The program then asks to select a classifier. *[Select your `train.lada` object]*. You have also to specify the limits for the different size classes to consider for the size spectra. The default value creates a regular sequence from 0.25mm to 2mm with a class width of 0.1mm (`seq(0.25, 2, by = 0.1)`). If you clear this entry, the program understands that you do not want to calculate size spectra for these samples. *[Keep default values and click `OK` now]*.
The last question is a name for the `ZIRes` object to create. *[Give `results` and click `OK` now]*.

We still have to implement the table of parameters for the biomass conversion in the program!

Zoo/PhytoImage calculate each sample in turn and generate a log file. Once the process is done, you should get a log file indocating that there is no error.
Your `ZIRes` object is now created (if no error occur; look at the log). If there are arrors, the most probable cause is a problem in the `Description.zis` file, or corresponding .zid files that are not located in the same directory as the .zis file. Make the corrections and start the analysis again.

12.3. Visualizing results

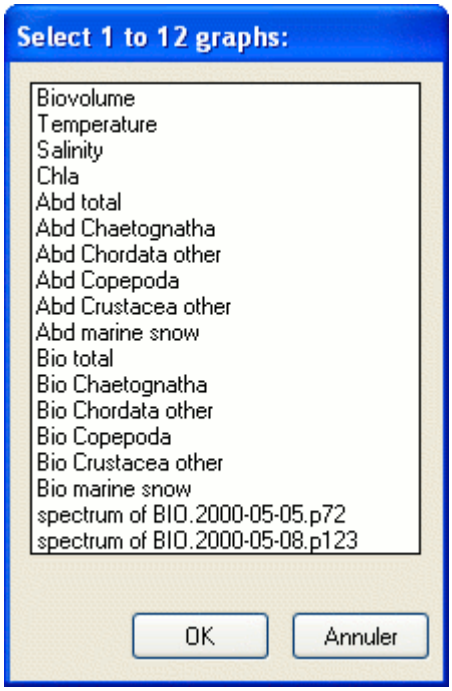
To visualize you series, use the menu entry `Analyze → View results...`, the shortcut `Ctrl+V`, or click on the eleventh button in the toolbar.

Having now calculated a `ZIRes` object that contains abundances, biomasses and size spectra, one can visualize graphs (or composite graphs with up to 12 graphs on the same page) of that series.

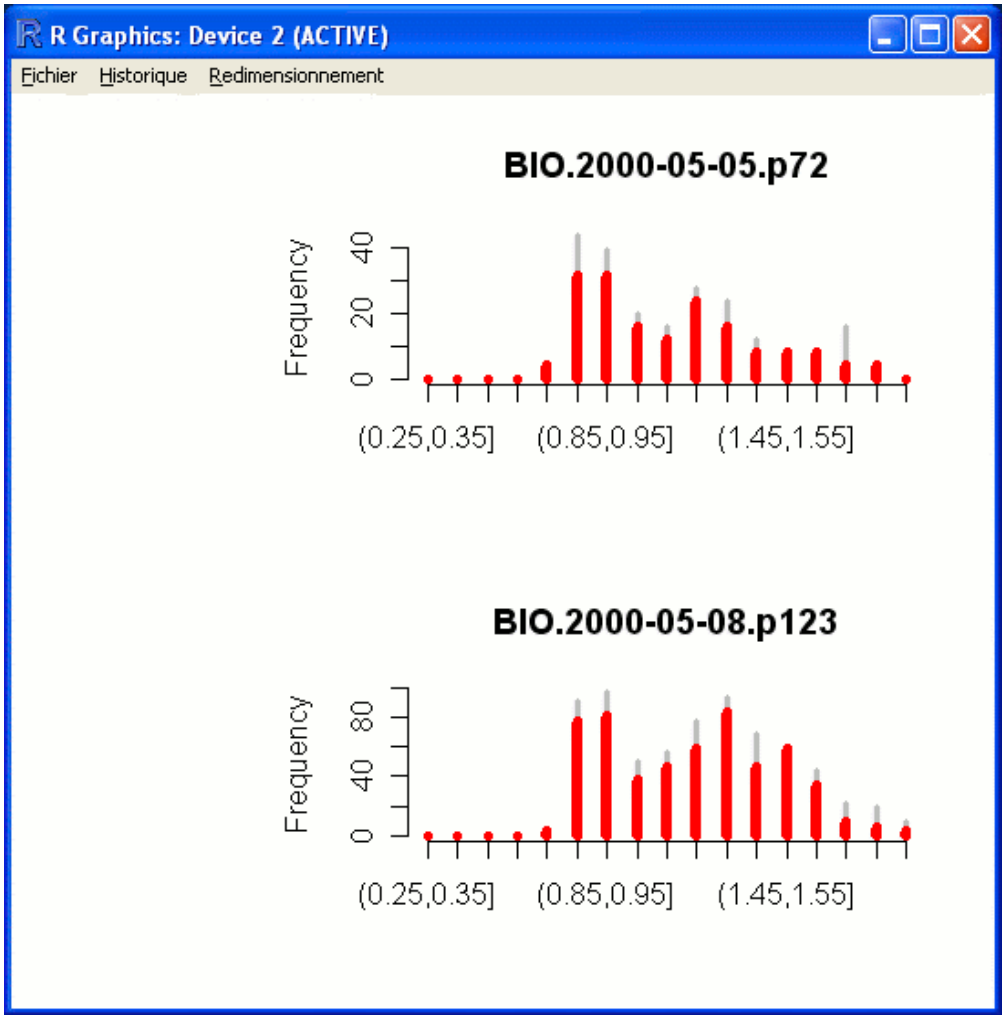
Currently, the program proposes only a limited number of graphs and you cannot customize colors, titles, etc.). These graphs are sufficient for a rapid inspection of time series, but spatial components are not handled yet. Graphs in R are very flexible, and you can visualize your data in many other ways...

To make graphs of your results, click on the eleventh button on the ZooImage assistant toolbar:

... and select the 'ZIRes' object you just created (`results`). You have then a list of possible graphs:



As the title of the list says, you can select between 1 and 12 graphs to draw. If you select the two spectra at the bottom of the list, the program asks also if you want to plot the spectra of a given taxa (in red, superimposed on top of the total size spectra). *[Select Copepoda in the list and click* `OK`*]*. You should obtain a composite graph similar to this one:



You should experiment with the different possible options here.

> *You can copy these graphs in Word. Just use* `File → Copy to clipboard → As Metafile` *in the graph window menu (or use* `Copy as metafile` *in the context menu after right-clicking on the window). Then, paste this graph in Word. If required, you can resize the graph window first, to adjust the size of the graph relative to the size of the text. If you have lots of graphs on the same page, you are better to maximize the graph window first.*

You can open several graph windows simultaneously, for comparison. In the `Utilities` menu of the ZooImage1 assistant, you have three entries in the `R Graphs` submenu: `New`, `Activate next` and `Close all`. They are self-explicit. The `Utilities → R Graphs → Activate next` switches the "active" flag to the next graph window. Indeed, there is only one active graph window at a time. It is the window that will receive the next graph(s). Its name ends with `(ACTIVE)`. The name of all other graph windows, if any, end with `(inactive)`. To send the next graph in a different window as the active one, use the `Activate next` menu entry until the target window becomes active.

12.4. Analyzing results in R

All Zoo/PhytoImage objects inherit from data frames, which are the basic case-by-variable type in R. Consequently, all the analysis and graphing functions of R can also be used without change on Zoo/PhytoImage objects. Look at the abundant litterature and the more than 5000 additional R packages available on CRAN (http://cran.r-project.org) to perform your analyses. Look, in particular at the task views about **environmetrics**, **graphics**, **machine learning**, **spatial**, **spatio-temporal** and **time series** for further tools that can be useful to analyse you plankton samples or series.

12.5. Exporting results

To write the result tables as ASCII files, use the menu entry `Analyze → Export results...`, the shortcut `Ctrl+E`, or click on the twelve button in the toolbar.

If, despite all the potentials of R to analyze your series right in the current environment, you want to export data, you can do it easily. Click on the forelast button on the ZooImage assistant toolbar:

Select your `ZIRes` object in the dialog box and indicate a directory (preferrably empty) where to place the tables. Zoo/PhytoImage exports one table for abundances and biomasses, and then it exports a separate table with size spectra for each sample. These are tabulation-delimited ASCII files. They should be easy to read from any other software (Microsoft Excel, Matlab, Python with Numpy/Scipy/Pandas, Julia, ...).

12.6. Further work with training/test sets

Version 3 of Zoo/PhytoImage introduces additional tools that add more flexibility in building training sets, visualizing how vignettes are automatically classified, and managing test sets.

These tools are accessible through the **Analyze** menu :

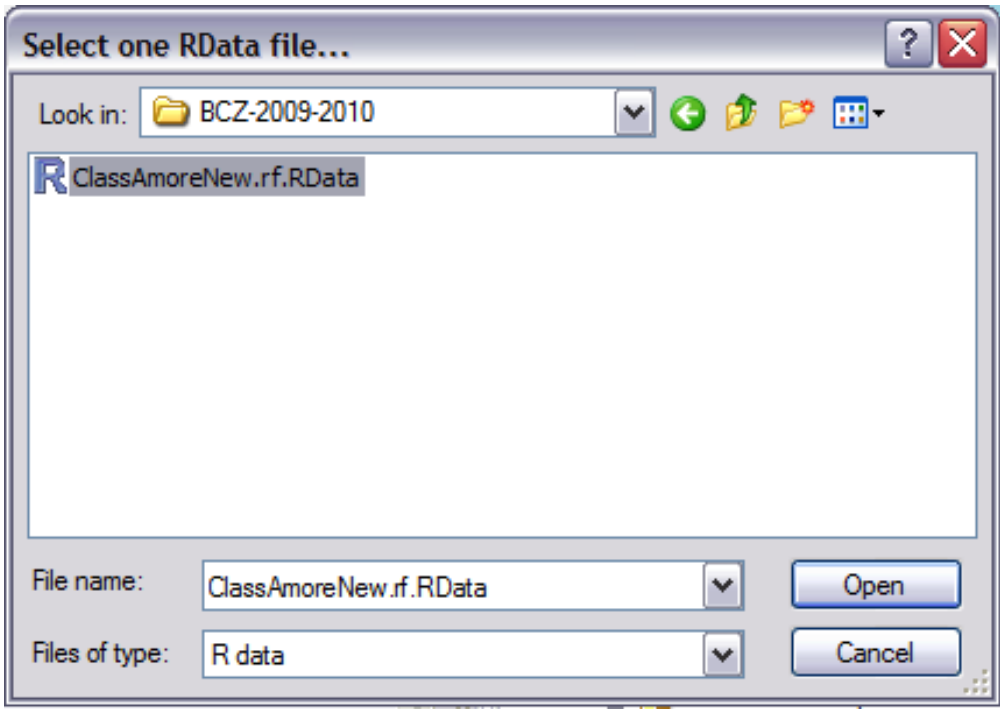
- **Add vignettes to training set** allows to complete existing tra ining sets by adding more vignettes to them without breaking the training set structure,
- **Automatic classification of vignettes** allows to select one sample and to represent the same folder hierachy as the one used in the original training set, with their vignettes pre-sorted according to the automatic prediction done by the chosen classifier. This serves as two purposes : (1) to visually check the quality of the classifier through the vignettes identifications, and (2) to allow for further manual correct (validation) of that classification. In this case, you can read the test set back as you do with a training set and you obtain a fully validated classification of your sample.

- **Validate classification** is a new tool that combines advanced statistical tools and a new user interface to easy (partial)-validation of classification. The tools detects so-called **suspect** items and present them first step-by-step so that optimisation procedure is more efficient. Typically, validation of only one third of all vignettes yields same level of error correction than a 90-95 % random validation procedure ! It is also combined with tools to m*odel* the error specifically for that sample, and to perform statistical correction according to that model. The combination of suspect detection and error correction provides even faster improvement of the validation : by manually validating 15-20 % only of the vignettes, one gets abundance by groups calculations with typically **less than 10 % of error for all groups**.

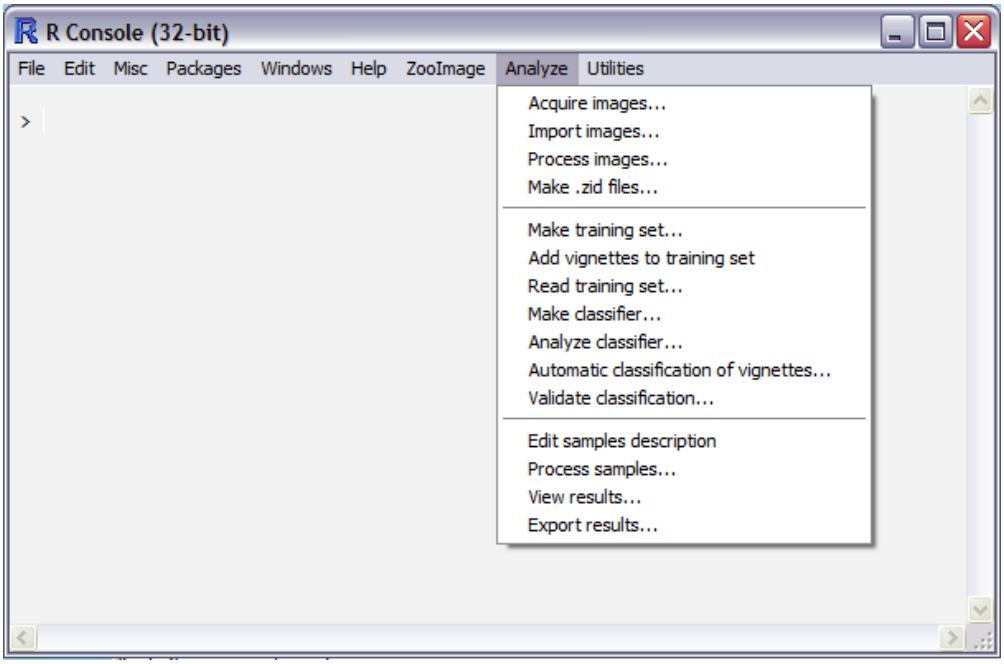
12.7. Smart validation of classification

Here is how to use the **validate classification** tool.

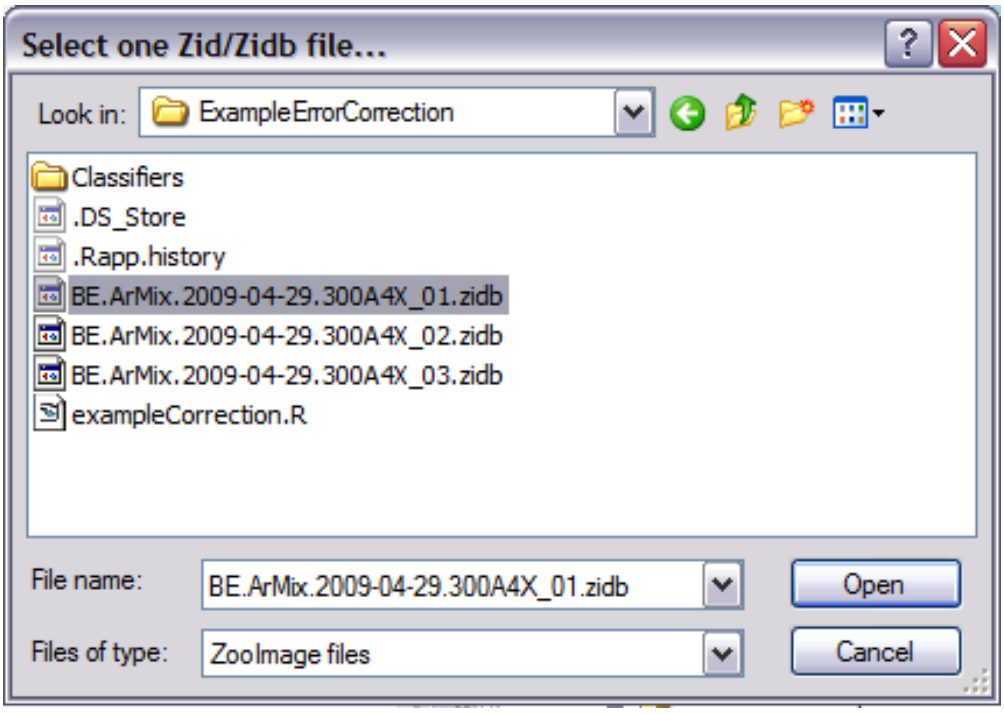
First, make sure you have created or loaded a suitable classifier (ZIClass object). Typically, you save your classifiers on disk in .Rdata files. So, to retrieve one, go to the menu **ZooImage → Load objects**, navigate to the folder where you store your classifier(s) and select the one you need :



Now that your classifier object is in memory, select **Validate classification** in the **Analyze** menu :

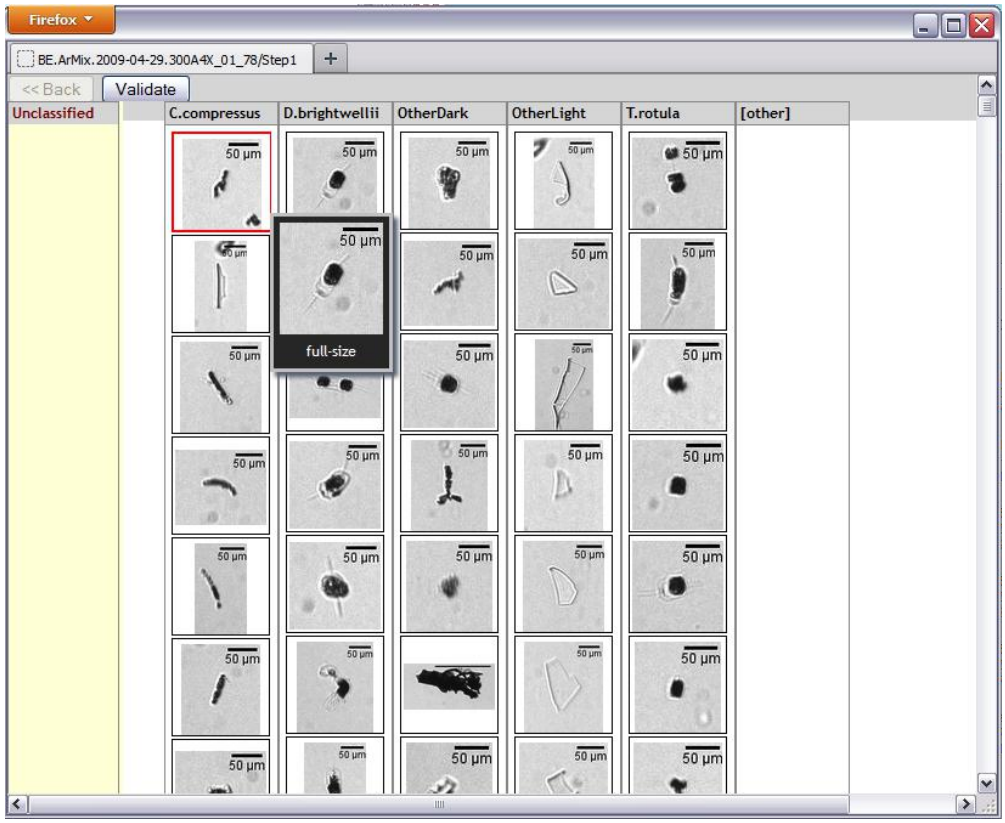


Select the ZID or ZIDB file of the sample you want to validate :



If there is no classifier found in memory, an explicit message invites you to create or load one first. Otherwise, Zoo/PhytoImage asks you now which one of all classifiers found in the current R session you want to use :

Once it is done, Zoo/PhytoImage creates a web page that presents you a first set of (by default) 1/20th of the vignettes in the sample :



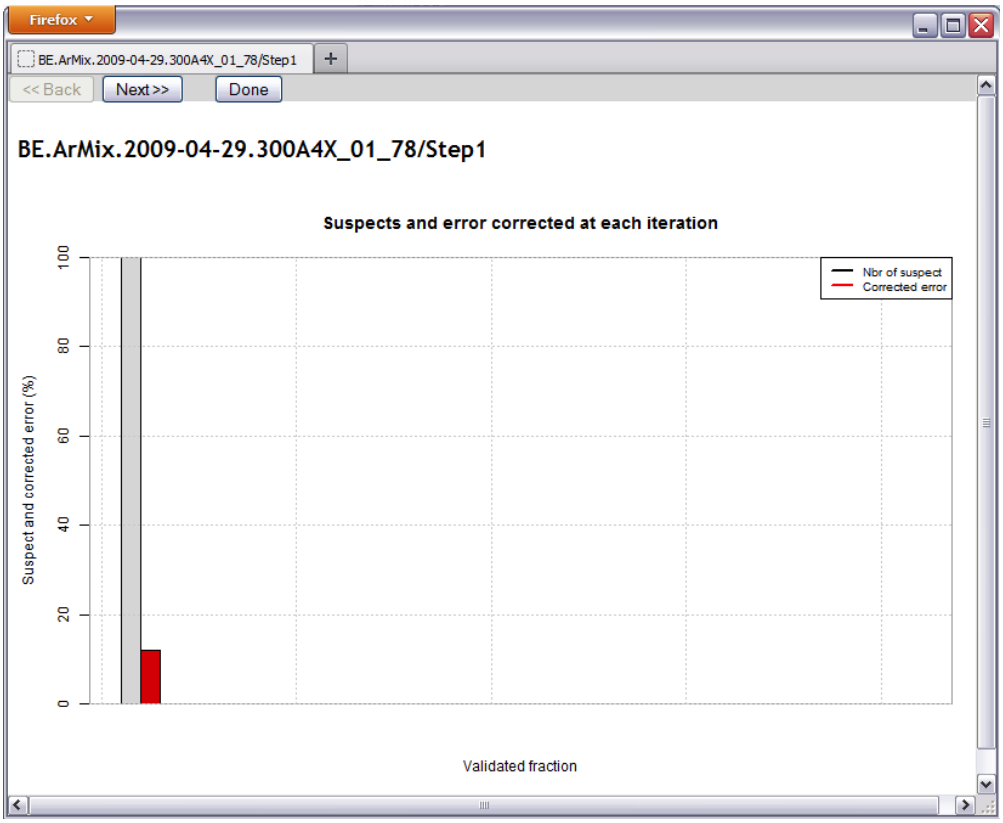
This page presents a first series of particles, randomly selected in the sample, as they are sorted automatically by the chosen classifier. Each class is represented by one column in the page (e.g., **C.compressus**, **D.brightwellii**, etc. in the example). All vignettes classified in one group are presented in the correponding column.

Moving the cursor on top of one vignette automatically triggers a floating window that dispalys the corresponding particle in full-size view for inspection.

All the vignettes can be freely drag and dropped everywhere. Thus, you can rearrange the vignettes in order to performs required corrections. For very long grids with tens or even hundreds of columns, you can use a special yellow area on the left named '**Unclassified**' to temporary store items that you want to relocate in a distant position in the grid. However, you cannot leave items in that special area when you validate your work.

For all particles that you cannot recognize, or that do not belong to the pre-specified classes, you have a special class **[other]** at the extreme right of the grid.

Once you have done with the validation of these vignettes, click on the **Validate** button. A report of the validation process done during that first step is displayed :



It present a barplot with gray bars representing the proportion of suspect items in the fraction just validated. During the first step, no model is calculated yet... so, all items are considered as suspect. A red bar at its right indicates the fraction of items that were incorrectly classified and that you just corrected. In the present case, it amounts at around 15 %. *This is a very good indication of the overall error in that classification, since this first sample is purely randomly selected !* Thus, you know that you have a totla of about 15 % error and that you already corrected 1/20th of that error.

If you continue to validate random subsamples, you still have to look at the remaining 19/20th of the sample. If you decide to accept a remaining error of less than 5 % of the total, you will still need to validate 2/3, that is roughly 12/20th of the whole sample. But wait... **doing so do not guarantee that you have less than 5 % error in *all* groups.** Typically, you will leave far more error in the rarest groups. Thus, you are better to *validate everything*, or...

... The smart validator provides a much more efficient way of validating your sample with this goal in mind of less than 5 % error in *all* groups. To reach this goal, a statistical model and a bayesian probability is calculated for each particle telling if it has a chance to be suspect (understand, probably wrongly classified) or not.

The model also considers several additional aspects :

- The probability returned by the classifier for the second class predicted for the particle is compared with the probability for the first, selected class. The idea is that, if the difference between those two probabilities is small, one should consider the particle is close to the border between the two classes and should be checked,

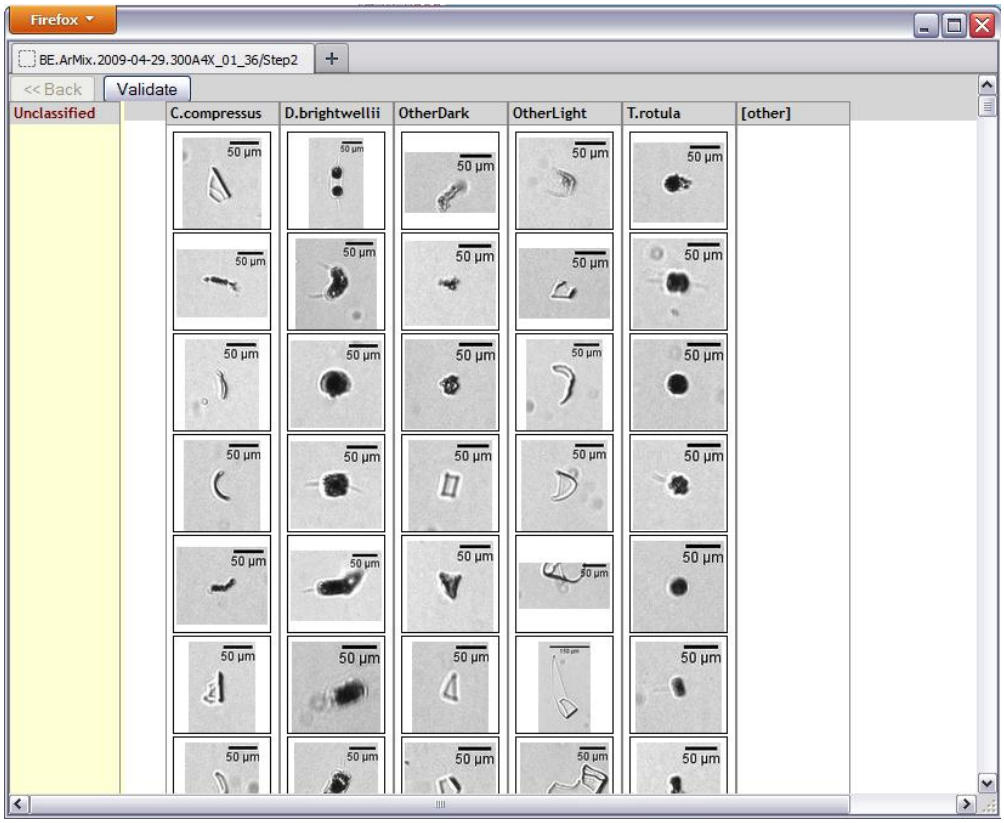
- The number of particles classified in the same class for the whole sample. If there are few of them, it is a rare group. It implies two consequences : (1) the probability of false positive increases, and (2) the class has more probabilities to contains no particles for that sample (because that taxonomic group is absent there, at that time). So, the probability to be suspect increases with the scarcity of particles classified in the same class,

- The information from the confusion matrix is used to determine which classes tend to be less good discriminated. Again, that information increases the probability of the corresponding particles to be suspect,

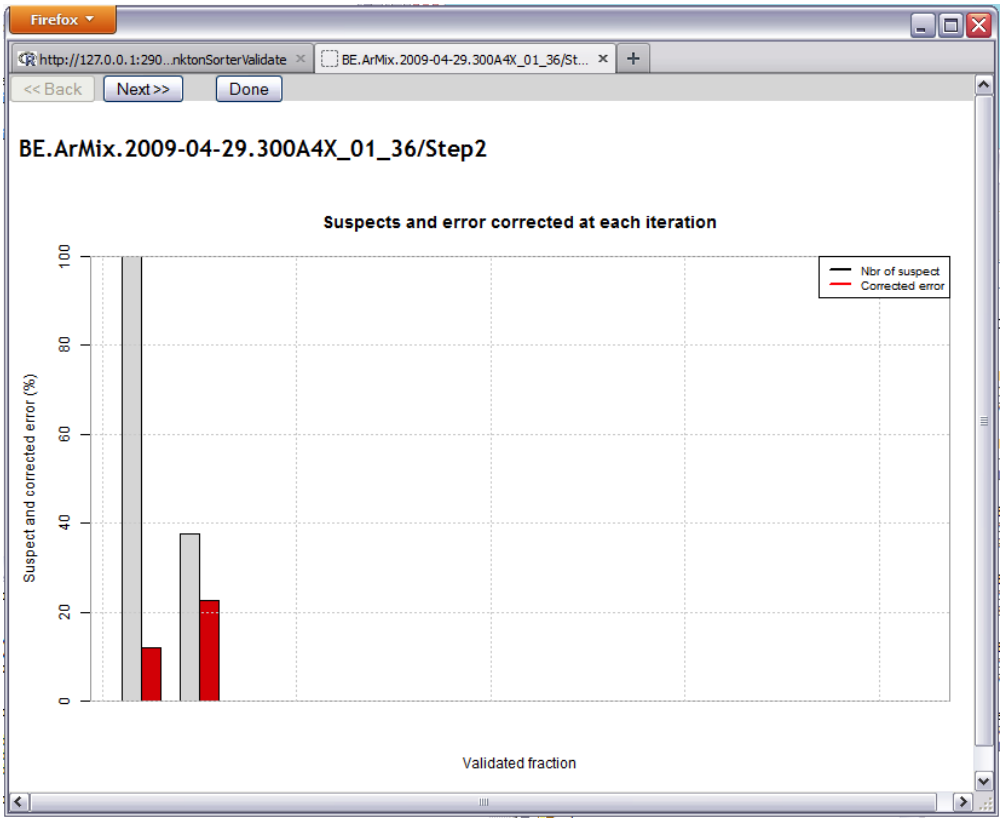
- Possibly, 'biological information' can be supplied too (not from the menu/dialog box, but by calling **correctError()** directly in the R console, see its help page at **?correctError**). That biological information should indicate if a given class has chances or not to be found in that sample. Say you know from the geographic location, from the time of the year, from the water temperature, or simply from a quick inspection of the sample under the microscope that class A is very unlikely to be present, and class B is certainly there. Just indicate a low value (say 0.01) to class A and a high value (say 0.99) to class B. Note that the numbers you provide are not necessarily restricted between 0 and 1, but the concept is easier to consider if you look at these weight like pseudo-probabilities of occurrence of the class in your sample.

Zoo/PhytoImage use the first set of particles as a training set to detect suspect items, using all features peasured on these particle, plus the additional variables described here above. Several algorithms can be use, but random forest is used by default.

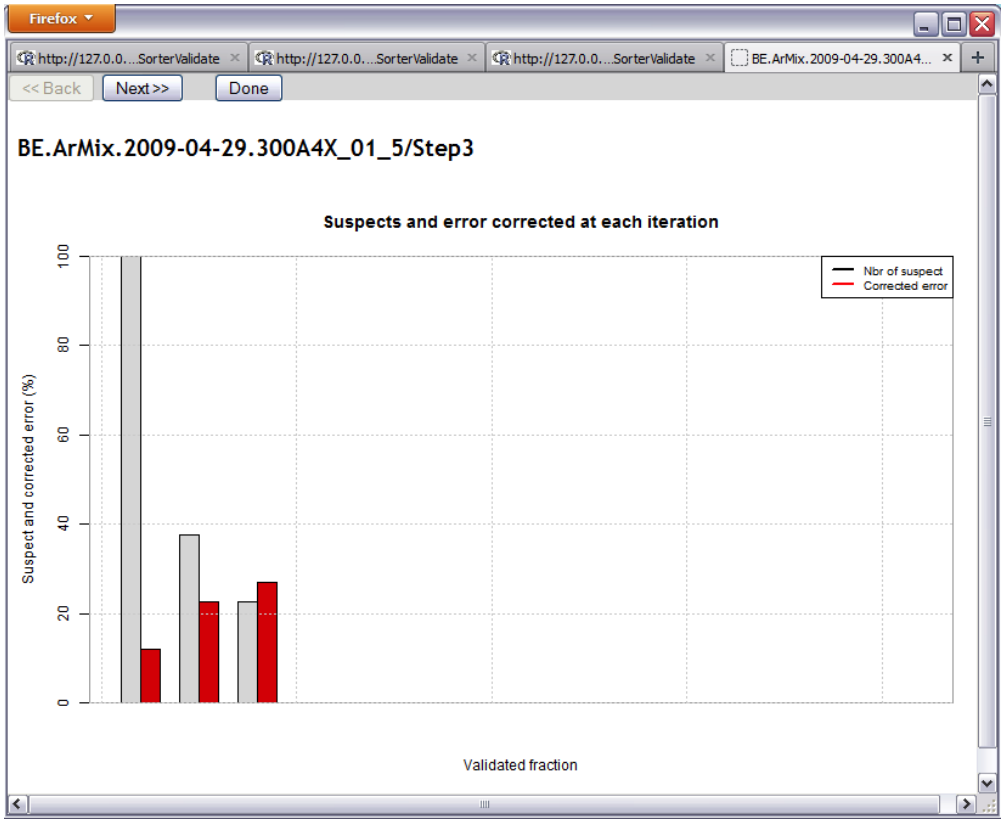
So, when you click **Next**, Zoo/PhytoImage presents you another subset of the particles in the sample. But this time, the subset is not randomly chosen, but rather mainly selected in the suspect items. As a consequence, the proportion of error happens to be higher. Thus your validation work is more efficient because you start to focus on the really problematic particles now !



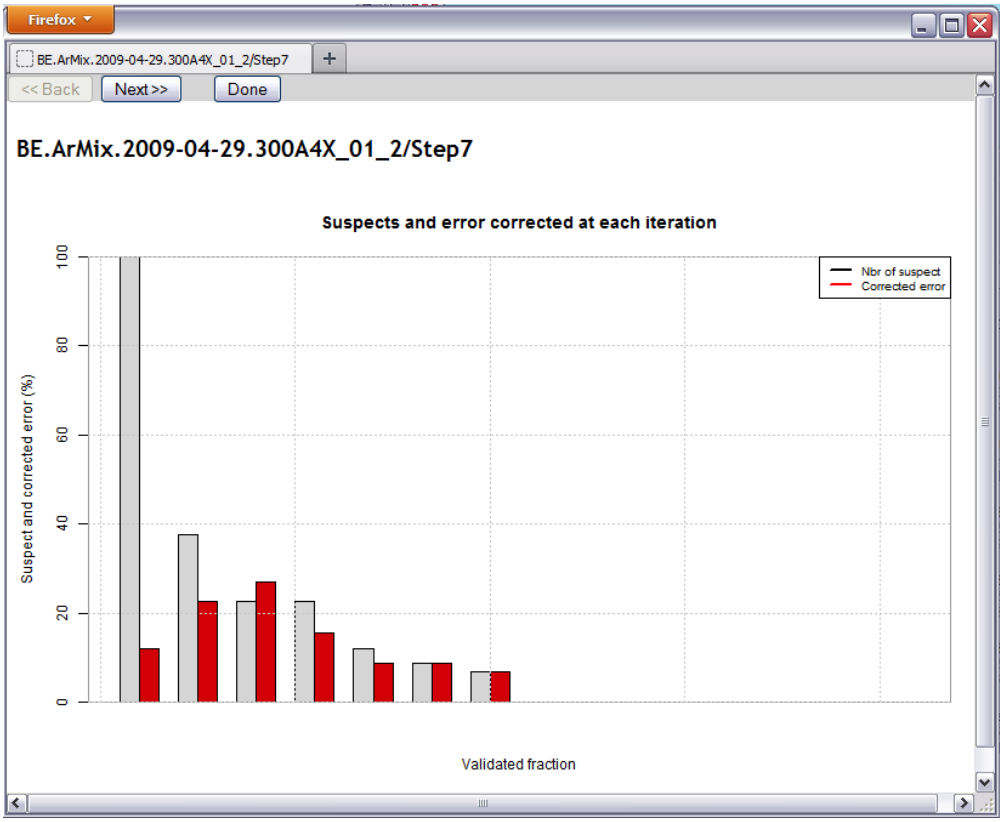
It is usually quite clear that this second set presents much more errors that the previous one… and you will also notice that, indeed, you got also much more « problematic » particles (hard to recognize particles, cropped items, blobs with strange forms, etc.). Do not hesitate to use the **[other]** group to collect what you cannot place elsewhere (but be consistent on what you do here). Click **Validate** when you have done with this second step.



In the report, the barplot has now a second series of gray/red bars. As you can see here, the identification of suspect items is mildly efficient (recall the training set contains very few particles... 1/20th of the whole sample). Yet, you almost doubled the fraction of erroneous particles at that step. Run it a third time :



On this sample, the algorithm predicts a relatively low amount of suspect items (on other samples, with a higher proportion of initial error, this fraction can easily reach 80 to 90 %). Nevertheless, the fraction of erroneous particles has increased a little bit more. You are now concentrating the error more efficiently. Continue with a few sets :



Here, after step 7, you notice two important things. First, the detection of suspects now closely matches actual error. Detection improves with the fraction of sample already validated that can be used for training the detection algorithm. Second, residual error drop to less than 10 %.

From this moment on, you know that you have manually validated all erroneous particles down to about 5 %. But, since the model is also used to calculate a *correction factor* for the remaining items, the calculation of abundances per classes will become quite good. Also remember that particles from rare groups were preferrably selected in the few first sets. This ensures you a good prediction for those rare groups, otherwise often problematic.

So, with this in mind, you can reasonably consider that the validation could end now and that you can trust the correction introduced by this partial validation, further helped with the statistical correction by the suspect detection model.

Click the **Done** button. Look now at the R Console. You got the corrected abundance of particles in the different classes, at it stand after the last step. Moreover, the results are saved in the `<sample>_valid` object. You can further explore it, and of course, you can use it as definitive classification of this sample.

13. Use of Zoo/PhytoImage at the R command line

A complete and detailed description of the use of zooimage functions inside the R Console is described in Chapter 12 of the following book :

Yanchang Zhao and Yonghua Cen (Eds.). Data Mining Applications with R. ISBN 978-0124115118, December 2013. Academic Press, Elsevier.

We encourage the interested readers to download the accompanying files from http://www.sciviews.org/zooimage/Data mining with R/ . There is a fully commented R script and an example dataset that browses the features available at the command line.

Here is an outline of most important tools, in additions to what you can already do using the graphical user interface and to menu in Zoo/PhytoImage 4 :

- Vignettes are accessible directement within R and can be included anywhere in R plots, or displayed as a gallery. The code to do so looks like this :

```
## Lazy loading data from one ZIDB file in R

db1 <- zidbLink(path_to_zidb)

## Contains data in *_dat1 and vignettes in *_nn

items1 <- ls(db1)

vigs1 <- items1[-grep("_dat1", items1)]

## Display a 5*5 thumbnail of the first 25 vignettes (Fig. 13.3)

zidbPlotNew("The 25 first vignettes in MTPS.2004-10-20.H1")

for (i in 1:25) zidbDrawVignette(db1[[vigs1[i]]], item = i, nx = 5, ny = 5)
```

- The summary method of a ZIClass object (a classifier) displays a lot of summary statistics, like recall, precision, specificity, F-score, balanced accuracy, etc. These statistics are calculated group-by-group. See the help page of the ZIClass object (?ZIClass).

- The ZIClass object has a confusion method that creates a confusion matrix with four specific plots : image, barplot, stars and dendrogram. The barplot is a new view of F-score called, « F-score » by group plot. See ?confusion and the example in the R script. The star plot can also be used to compare two classifiers applied to the same test set.

- There are also complements about the way Zoo/PhytoImage calculates abundances and biomasses/biovolumes. You can calculate these quantities at different detail levels and indicate which groups are out of interest (e.g., marine snow and zooplankton if your study focuses on phytoplankton).

- The confusion object can be adjusted for various prior probabilities (abundances per groups) using the prior() function. This allows you to visualize the impact of different sample composition in the false positive and false negative rates per groups.

- Do not forget also all the R tools available to manipulate machine learning objects. See the machine learning task views at http://cran.r-project.org/web/views/MachineLearning.html.

Finally, chapter 12 in the Data mining applications with R book presents a collection of bibliographical references (64), most of them pointing on publications whose analyses were done using Zoo/PhytoImage. This is also an excellent source of inspiration showing in practice how Zoo/PhytoImage can be used.

14. ANNEXES

14.1. Data and metadata in .zis files

Here is the explanation of the **data** and **metadata** in this `description.zis` file:

Key	Section	Comment
ZI1	-	This is not a key, but just an identifiant telling it is a ZooImage1 file.
Id	Description	The short identifiant of the series.
Name	Description	A longer name for this series.
Institution	Description	The institution that owns the series, i.e., where original biological material is stored, if any.
Objective	Description	The goal(s) of this study.
Description	Description	A short description of the series[21].
Contact	Description	The name of a responsible person of this series.
Email	Description	The email address of the contact.
URL	Description	An optional URL pointing to a Web page that further describes the series, if any.
Note	Description	A short general comment about this series.
Code	Series	The code of a sub-series.
Name	Series	The name of a sub-series.
Project	Series	The project in which this sub-series is included.
Institution	Series	The owner of the sub-series, as above for the series.
Country	Series	Country(ies) concerned by this sub-series.
Location	Series	Place(s) concerned by this sub-series.
Contact	Series	As above for the sub-series.
Email	Series	Idem.
URL	Series	Idem.
Note	Series	Idem.

21 Fill these metadata : many of these are used by Zoo/PhytoImage for its calculations !

Code	Cruises	A code for a cruise.
ShipName	Cruises	The name of the ship.
ShipType	Cruises	The type of the ship.
ShipCallSign	Cruises	Immatriculation of the ship.
PortDeparture	Cruises	Self-explicit…
PortReturn	Cruises	Idem.
Captain	Cruises	Name of the captain.
Coordinator	Cruises	Name(s) of the scientific coordinator(s) on board.
Investigators	Cruises	Name(s) of additional scientific staff on board.
Start	Cruises	Date of departure in `yyyy-mm-dd`.
End	Cruises	Date of arrival at the final destination in `yyyy-mm-dd`.
SouthmostLat	Cruises	Southmost latitude reached in `+/-x.xx` (degree.decimal).
WestmostLong	Cruises	Westmost longitude reached in `+/-x.xx`.
NothmostLat	Cruises	Northmost latitude reached in `+/-x.xx`.
EastmostLong	Cruises	Eastmost longitude reached in `+/-x.xx`.
Project	Cruises	The project to which this cruise belongs.
URL	Cruises	An optional URL pointing to a web page that further describes this cruise.
Note	Cruises	A short comment about this cruise.
Code	Stations	A code for this station.
Location	Stations	The name of location of this station.
Latitude	Stations	The latitude of the station (in `+/-x.xx`).
Longitude	Stations	The longitude of the station (in `+/-x.xx`).
Start	Stations	The date at which sampling started at the station (in `yyy-mm-dd`).
End	Stations	The date at which sampling was stopped (if any, in `yyyy-mm-dd`).
Frequency	Stations	The frequency of sampling (in no of samples per day).
Depth	Stations	The maximum depth at the station location (in m).

Description	Stations	A short description for this station.
Note	Stations	A short note concerning this station.
Label	Samples	The complete label of the sample, as in the file names.
Code	Samples	A code for this sample.
SCS	Samples	The `SCS` for that sample.
Series	Samples	The series code to which that sample belongs.
Cruise	Samples	The cruise code corresponding to the sample (if any).
Station	Samples	The station code.
Date	Samples	The data of sampling (in `yyyy-mm-dd` format).
Time	Samples	The time of sampling (in `hh::mm::ss`).
TimeZone	Samples	The time zone (lag from GMT in `+/-x` hours).
Latitude	Samples	The latitude of sampling (in `+/-x.xx`).
Longitude	Samples	The longitude of sampling (in `+/-x.xx`).
CoordsPrec	Samples	Precision of lat./long. (radius in m).
Operator	Samples	Who collected this sample?
GearType	Samples	The type of gear used to collect the sample.
OpeningArea	Samples	The opening area (if collected with a net, in m^3).
MeshSize	Samples	For a net only, size of the mesh (in µm).
DepthMin	Samples	Minimum depth of sampling (in m).
DepthMax	Samples	Maximum depth of sampling (in m).
SampVol	Samples	Volume of seawater sampled (in m^3).
SampVolPrec	Samples	Precision of sampled volume (in m^3).
TowType	Samples	Type of tow (vertical, horizontal, oblique, etc.).
Speed	Samples	Speed during tow (in m/s).
Weather	Samples	Weather conditions during sampling.
Preservative	Samples	Preservative used (for instance, buffered formaldehyde 4%).
Staining	Samples	Staining used (if any).

Biovolume	Samples	Rough estimation of the biovolume after sedimentation (in mm^3).
Temperature	Samples	Temperature of the water at sampling (in degree Celcius).
Salinity	Samples	Salinity of sampled water (in per thousands).
Chla	Samples	Chlorophyll alpha in the sampled water.
Note	Samples	A short note about this sample.
…	Samples	You can add any additional measurement done on the sample here…

UMONS
Faculté des Sciences
Département d'Informatique

Correction statistique de l'erreur dans le cadre de la classification automatique du plancton

Directeur : Mr Philippe Grosjean

Projet réalisé par
Flavien Dereume-Hancart

Rapporteurs : Mr Tom Mens
Mr Mathieu Goeminne

Année académique 2012-2013

Table des matières

Chapitre 1| Introduction

1.1 Contexte

Le plancton est l'ensemble des organismes vivants en suspension dans les eaux douces ou marines. Il peut être de type animal (zooplancton) ou de type végétal (phytoplancton). Le plancton est à la base de la chaîne trophique de beaucoup d'écosystèmes marins. Tant le zooplancton que le phytoplancton sont de bons indicateurs de changements globaux. En effet, ils sont capables de répondre à des modifications du milieu très rapidement par des variations d'abondance et/ou de composition spécifique et de biomasse. Pour cette raison, ces communautés sont abondamment étudiées et fournissent des informations précieuses sur le fonctionnement des océans [1].

Figure 1.1 – Un zooplancton de type copepod

1.2 Problématique

L'analyse complète manuelle d'un échantillon de plancton est une tâche laborieuse nécessitant parfois plusieurs jours de travail et ne permettant pas à elle seule de mettre en évidence les distributions spatiales et temporelles du plancton. Pour cela, depuis les années 1980, l'analyse d'images couplée à la classification supervisée est envisagée afin d'automatiser et d'accélérer le traitement des échantillons. Dernièrement, le laboratoire d'Écologie Numérique des Milieux Aquatiques de l'UMONS a développé, en collaboration avec la Politique scientifique fédérale et l'IFREMER, un logiciel appelé Zoo/PhytoImage [2] destiné à l'élaboration de séries spatio-temporelles du plancton en automatisant les processus de traitement des échantillons. Il s'agit d'un logiciel open source permettant d'analyser diverses sortes d'images numériques du plancton et de mesurer, dénombrer et classer les différents organismes planctoniques présents dans ces images. Un des inconvénients de cette méthode automatique est qu'elle introduit un certain pourcentage d'erreurs dans la classification.

1.3 Historique

La présente section retrace l'évolution de l'étude du plancton et de l'apport de l'informatique dans ce domaine. Elle se base entièrement sur l'article [3].

1.3.1 L'étude du plancton et ses défis

Les premières collectes massives de planctons ont débuté en 1987 dans le but de pouvoir répondre à trois questions :

1. quels types de plancton trouve-t-on dans les océans ?
2. en quelle quantité ?
3. comment évoluent-ils en fonction du temps ?

À cette époque, le plancton est collecté à l'aide de filets et l'énumération des échantillons récoltés est effectuée par des experts capables de distinguer des caractéristiques morphologiques subtiles. L'analyse au microscope des échantillons collectés demande de sous-échantillonner, de compter et de trier un grand nombre de particules dans divers groupes taxonomiques. Cette activité prend un temps considérable et entraine de longs délais d'attente entre la collecte des données et leur interprétation.

1.3.2 Le coup de pouce de l'imagerie numérique

Compte tenu de la lenteur de la collecte, des systèmes d'imagerie numérique et de numérisation furent développés éliminant complètement l'utilisation des filets. Le Video Plankton Recorder [4], aussi appelé VPR, fut le précurseur d'une série de dispositifs d'imagerie *in situ*. Outre ceux-ci, la numérisation des échantillons prélevés par les pompes et les filets fut de plus en plus utilisée pour traiter le plancton. Il existe plusieurs instruments capables d'effectuer cette tâche comme ZOOSCAN [5]. Les bénéfices potentiels de l'imagerie *in situ* et des techniques de numérisation ont mené les chercheurs à développer des moyens efficaces pour extraire des informations utiles du vaste nombre d'images produites à partir des échantillons. La première tentative pour apprivoiser cette quantité d'images consistait à identifier les images contenant certains objets. Certains équipements comme le VPR possèdent des processeurs dédiés qui scannent chaque image à la recherche d'objets pouvant être identifiés. Ces régions d'intérêts sont ensuite isolées en utilisant des routines de binarisation et de segmentation, puis coupées et sauvegardées. Cela permet à l'utilisateur de ne pas avoir à inspecter des milliers d'images. Cette grande quantité d'images doit par la suite être classée. Pour les mêmes raisons que celles citées plus haut, il est primordial d'avoir recours à une procédure automatique pour effectuer ce travail.

1.3.3 L'apport de la classification automatique

Les avancées en termes de « machine vision », « pattern recognition » et « datamining » ont amenés plusieurs chercheurs à développer des logiciels capables de classer automatiquement ces images de plancton. Le gain de temps produit peut alors être utilisé pour obtenir des connaissances utiles sur ces individus et ainsi augmenter la compréhension de leur écosystème. Cependant, la classification doit faire face à une série de défis qui peuvent varier selon différents facteurs (illumination, orientation, nature de l'étude, etc). Malgré la difficulté du problème, plusieurs groupes de travail à travers le monde ont fait de gros progrès dans la construction de classificateurs. Le résultat d'une discussion lors d'un workshop en 2005 [6] suggère que l'utilisation de la texture, de la forme et d'autres caractéristiques permettent de classifier correctement 70 à 80 % des planctons en 10 à 20 groupes taxonomiques, en utilisant des machines à vecteurs de support [7, 8, 9, 10], des arbres de décision [8, 9, 10, 11] et d'autres algorithmes d'apprentissage supervisé [8, 9, 10]. Étant donné que les équipements d'imagerie ont peu d'utilité sans des solutions de traitement et de classification d'images, plusieurs solutions personnalisées ont été développées. Parmi celles-ci on retrouve :

* Visual Plankton [4] qui est utilisé avec le VPR.
* PICES [12] qui est utilisé avec SIPPER.
* ZooProcess en conjonction avec Plantkon identifier pour ZOOSCAN.

1.3.4 Les qualités requises d'un logiciel de traitement et de classification

Pour que chaque logiciel de traitement et de classification soient utiles à la communauté des chercheurs, ceux-ci doivent être capables d'effectuer une série de tâches :

* importer des images (importation).
* séparer les objets cibles du fond (segmentation).
* analyser ces objets pour potentiellement extraire des informations utiles permettant de distinguer un organisme d'un autre (sélection et extraction de caractéristiques).
* fournir un moyen visuel pour trier les images afin de produire un set d'entrainement et un set de tests pour le classificateur (entrainement).
* mesurer les erreurs de classification statistiquement à l'aide d'une matrice de confusion.

Les logiciels cités précédemment sont capables d'effectuer ces différentes tâches. Le plus complet d'entre-eux est Zoo/PhytoImage car il est capable de travailler avec une grande variété d'images et il a d'ailleurs été modifié afin de pouvoir gérer les images produites par le système de numérisation FlowCam [13]. De plus, il est personnalisable et extensible grâce à un mécanisme de plugins.

1.4 Objectifs du travail

L'analyse manuelle des échantillons de plancton est une tâche longue et fastidieuse. Des méthodes automatiques ont donc été conçues afin de palier à ce problème. Cependant, les résultats fournis par ces méthodes ne sont pas fiables à 100% car elles commettent un certain pourcentage d'erreurs qui peut venir fausser considérablement ces résultats. Le service d'Écologie Numérique des Milieux Aquatiques de l'UMONS a alors développé une procédure effectuant un compromis entre les méthodes manuelles et automatiques, permettant d'estimer l'abondance de différents groupes de plancton au sein d'un échantillon. L'idée est de n'analyser qu'une faible portion de l'échantillon en essayant de concentrer les éventuelles erreurs au sein de celui-ci. Cette erreur est ensuite corrigée statistiquement sur base de cet échantillon validé. Cette procédure a été implémentée en *R*, un langage de programmation multi-paradigme utilisé pour le traitement des données et l'analyse statistique. Un chapitre est dédié à ce langage (voir annexe A). Mon travail consiste à optimiser divers aspects de ce programme :

* améliorer la structure du code ;
* diminuer la consommation mémoire ;
* réduire d'au moins un facteur 10 le temps d'exécution ;
* assurer son bon fonctionnement ;
* intégrer de nouvelles fonctionnalités ;
* valider les résultats.

Chapitre 2| État de l'art

Ce chapitre se compose de 3 parties. Dans la première partie, nous aborderons le concept de classification automatique. À cet effet, nous étudierons plusieurs sortes d'algorithmes de classification : la régression logistique [7, 8, 9, 10], les classificateurs bayesiens [8, 9, 10], les k plus proches voisins [8, 9, 10], les arbres de décision [9, 8, 10, 11], les forêts d'arbres décisionnels [8, 9, 14, 15], les machines à vecteurs de support [7, 8, 9, 10] et les réseaux de neurones [7, 8, 9, 10]. Ces algorithmes possèdent plusieurs implémentations dans *R*. Ensuite, dans la seconde partie, nous verrons une méthode de correction statistique des erreurs de classification. Finalement, dans la troisième partie, nous examinerons plusieurs exemples d'implémentations réelles de ces techniques conçues dans le cadre de la classification automatique du plancton.

2.1 Classification automatique

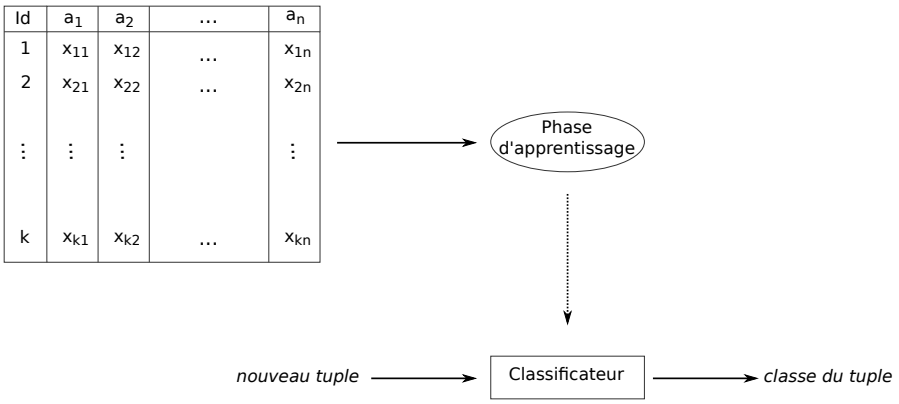


Figure 2.1 – *Sur base d'un ensemble d'entrainement, l'algorithme de classification créée un classificateur capable de prédire la classe pour de nouveaux tuples*

La classification automatique est une tâche consistant à attribuer une catégorie (ou classe) à un objet en se basant sur la valeur de ces attributs. Ce type d'activité trouve des applications dans beaucoup de domaines. Par exemple, les filtres anti-spam analysent les en-têtes et le contenu des mails afin d'identifier les courriers non sollicités [16]. Certains smartphones équipés d'un appareil photo sont capables de déterminer la position des visages en analysant en temps réel les pixels de l'image [17]. Les banques en font également usage afin que les transactions susceptibles d'être frauduleuses leur soient signalées. Dans le cadre de ce projet, les caractéristiques d'organismes planctoniques seront analysées dans le but de leur attribuer un groupe taxonomique et ainsi pouvoir estimer leur abondance.

La classification est un processus qui se déroule en 2 étapes. La première étape est la **phase d'apprentissage** durant laquelle l'algorithme de classification construit un classificateur (ou modèle) décrivant les données présentes dans l'**ensemble d'apprentissage**. Les données contenues dans cet ensemble sont appelées des **tuples**. Un tuple x est un vecteur à n dimensions,

$x = (x_1, x_2, \ldots, x_n)$, contenant les mesures effectuées pour n attributs $a_1, a_2, \ldots, a_n$. Chacun d'eux possède également un attribut de classe, y, désignant la catégorie à laquelle il appartient. Cette étape, aussi connue sous le nom d'**apprentissage supervisé**, peut être perçue comme l'apprentissage d'une fonction, $y = f(x)$, capable d'attribuer à chaque tuple x une classe y. À la seconde étape, le classificateur est utilisé afin de prédire les classes pour de nouvelles données. La FIGURE 2.1 illustre ce procédé graphiquement.

La précision d'une classification peut facilement être estimée. Pour cela, nous utilisons un **ensemble de tests** composé de tuples et de leurs classes associées. Ces tuples doivent être indépendants de ceux utilisés par l'ensemble d'apprentissage car il en résulterait une estimation trop optimiste du fait du surajustement du modèle à ces données. La classe associée à chaque tuple de test est alors comparée à la classe prédite par le classificateur. Le pourcentage de tuples ayant été correctement classés par le classificateur correspond à sa précision. Lorsque sa qualité est jugée suffisante, le classificateur peut être utilisé pour classer des futurs tuples pour qui la classe n'est pas connue. Examinons sans plus tarder comment différents algorithmes de classification implémentent ces deux étapes.

2.1.1 Prédiction numérique : la régression linéaire

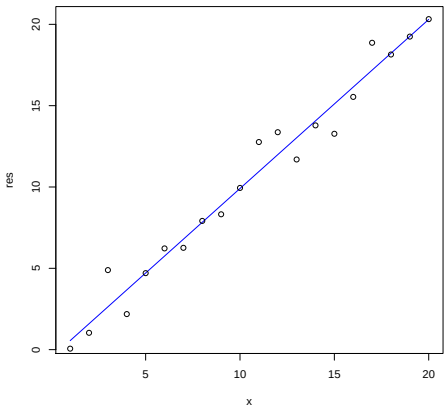


FIGURE 2.2 – *Un exemple de régression linéaire. Il s'agit de la droite qui s'ajuste le mieux au nuage de points*

Lorsque l'attribut de classe à prédire est une valeur numérique continue et que les attributs servant à la prédiction le sont également, la régression linéaire est une technique naturelle à considérer. Il ne s'agit donc pas d'une méthode classification mais bien d'une méthode de prédiction. Cependant, nous jugeons bon de l'étudier car elle nous servira de base lors de l'étude de la régression logistique. L'idée derrière cet algorithme est d'exprimer la classe comme une combinaison linéaire pondérée de la valeur des attributs :

$$y = w_0 + w_1 x_1 + \cdots + w_n x_n$$

où y est la classe ; $x_1, \ldots, x_n$ la valeur des attributs ; $w_0, \ldots, w_n$ les poids. Ces poids sont déterminés à l'aide des données de l'ensemble d'apprentissage. Soit y_i la classe du i^{eme} tuple et $x_{i1}, \ldots, x_{in}$ la valeur de ses attributs. La valeur de la classe prédite est donnée par :

$$w_0 + w_1 x_{i1} + \cdots + w_n x_{in} = w_0 + \sum_{j=1}^{n} w_j x_{ij}$$

L'attribut x_0 est appelé le biais, sa valeur est toujours égale à 1. La méthode de régression linéaire consiste à choisir les coefficients w qui minimisent la somme quadratique de la différence entre la classe réelle et la classe prédite pour tous les tuples de l'ensemble d'apprentissage. Supposons qu'il y ait k tuples, alors cette somme s'écrit :

$$\sum_{i=1}^{k} \left(y_i - \sum_{j=0}^{n} w_j x_{ij} \right)^2$$

Cette somme est la quantité que nous devons minimiser afin déterminer les coefficients adéquats. Cela peut se faire à l'aide de la méthode des moindres carrés.

Méthode des moindres carrés

La méthode des moindres carrés est une approche standard permettant d'approximer la solution d'un système d'équations. Cette solution est obtenue en minimisant la somme quadratique des erreurs de chaque équation. Dans notre cas, nous voulons minimiser la somme quadratique de la différence entre une valeur observée et une valeur ajustée fournie par un modèle. Le système d'équations étudié est de la forme :

$$\sum_{j=0}^{n} w_j x_{ij} = y_i \qquad \text{avec } i = 1, \ldots, k$$

où n est le nombre d'attributs et k le nombre de tuples de l'ensemble d'apprentissage. Il peut être réécrit sous forme matricielle $X.w = y$ avec

$$X = \begin{bmatrix} 1 & x_{11} & x_{12} & \ldots & x_{1n} \\ 1 & x_{21} & x_{22} & \ldots & x_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{k1} & x_{k2} & \ldots & x_{kn} \end{bmatrix} \qquad w = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} \qquad y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_k \end{bmatrix}$$

La solution $\hat{w}$ du problème de minimisation s'écrit donc

$$\hat{w} = \arg\min_{w} Cost(w)$$

avec $arg\min_{w}$ l'ensemble des valeurs qui minimisent $Cost$. La fonction de coût $Cost$ est donnée par

$$Cost(w) = \sum_{i=1}^{k} \left(y_i - \sum_{j=0}^{n} w_j x_{ij} \right)^2 = \|y - Xw\|^2$$

Ce problème de minimisation possède une solution unique, à condition que les n colonnes de la matrice X soient linéairement indépendantes, donnée par

$$\hat{w} = (X^T X^{-1}) X^T y$$

La FIGURE 2.2 montre un exemple de régression linéaire appliquée à un nuage de points.

La régression linéaire peut également être utilisée pour effectuer une classification multi-classe. Pour cela, il suffit de l'exécuter indépendamment pour chaque classe en posant $y = 1$ pour les tuples appartenant à celle-ci et 0 sinon. La valeur de chaque expression linéaire est alors calculée et la classe ayant obtenu la plus grande valeur est choisie. Cette méthode porte le nom de régression linéaire multiple et peut être perçue comme l'approximation d'une fonction numérique d'appartenance. Malheureusement, celle-ci comporte 2 inconvénients. Premièrement, la valeur d'appartenance qu'elle produit n'est pas une probabilité car elle peut sortir de l'intervalle $[0,1]$. Deuxièment, la méthode des moindres carrés suppose que les attributs sont linéairement indépendants mais aussi qu'ils sont distribués selon une loi normale. Cette hypothèse n'est pas respectée car les classes prennent uniquement les valeurs 0 et 1.

2.1.2 Classification linéaire : la régression logistique

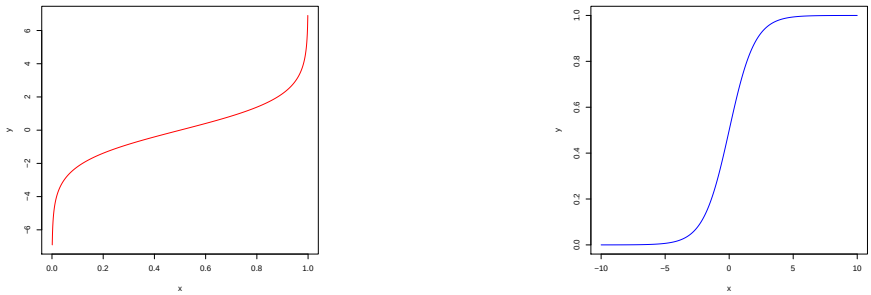


Figure 2.3 – *Graphique de la fonction logit (à gauche) et de sa réciproque, la fonction sigmoïde (à droite)*

Au lieu de travailler avec des variables quantitatives, la régression logistique s'intéresse à la description d'une variable qualitative y à deux modalités,0 ou 1, suivant une loi de Bernouilli de paramètre π. L'objectif est d'adapter le modèle linéaire vu précédemment en cherchant à expliquer les probabilités

$$\pi = P(y = 1|x) \qquad et \qquad 1 - \pi = P(y = 0|x)$$

L'idée est d'effectuer une transformation de variables en faisant intervenir une fonction $g : [0,1] \rightarrow \mathbb{R}$ monotone et donc de chercher un modèle linéaire de la forme

$$g(\pi_i) = \sum_{j=0}^{n} w_j x_{ij}$$

La fonction **logit** répond à ce critère. Elle est définie par

$$logit(\pi) = ln\frac{\pi}{1-\pi} \qquad avec \qquad logit^{-1}(z) = \frac{1}{1+e^{-z}}$$

où la quantité $\frac{\pi}{1-\pi}$ exprime un rapport de chance. Sa réciproque est appelée la fonction sigmoïde. Toutes deux sont représentées à la figure 2.3. La probabilité π_i peut être estimée par

$$\pi_i = logit^{-1}(logit(\pi_i)) = \frac{1}{1+e^{-\sum_{j=0}^{n} w_j x_{ij}}}$$

Comme pour la régression linéaire, les poids w doivent être ajustés afin d'ajuster le modèle aux données de l'ensemble d'apprentissage. La régression linéaire utilise une somme quadratique pour mesurer la qualité de l'ajustement. La régression logistique, quant à elle, utilise le logarithme de vraissemblance du modèle. La fonction de vraissemblance, notée $L(y|w)$ est une fonction de probabilité conditionnelle qui décrit les valeurs y_i d'une loi statistique en fonction des paramètres w_i. Elle s'exprime à partir de la fonction de densité $f(y, w)$ par

$$L(y|w) = \Pi_{i=1}^{k} f(y_i, w)$$

Dans notre cas, cette densité de probabilité est celle de la loi binomiale $\mathbb{B}(1, \pi)$ qui est donnée par

$$f(y_i, w) = \left(\frac{1}{1 + e^{-\sum_{j=0}^{n} w_j x_{ij}}}\right)^{y_i} \left(1 - \frac{1}{1 + e^{-\sum_{j=0}^{n} w_j x_{ij}}}\right)^{1-y_i}$$

En passant au logarithme, la fonction de vraissemblance devient

$$\sum_{i=1}^{k} (1 - y_i) \log(1 - \pi) + y_i \log(\pi)$$

Les poids w doivent être choisis de sorte à maximiser le logarithme de vraissemblance. Il n'y a pas de solution analytique à ce problème, des méthodes numériques itératives comme l'algorithme *gradient descent* doivent donc être utilisées. Une fois l'apprentissage des poids terminé, la classe y d'un tuple x peut être déterminée en examinant la valeur fournie par le modèle linéraire. Si $\sum_{i=0}^{n} w_i x_i > 0$ alors $y = 1$, sinon $y = 0$. Nous pouvons également imposer un seuil $\alpha \in [0, 1]$ tel que si $\pi > \alpha$ alors $y = 1$ et 0 sinon.

Gradient descent

Gradient descent exploite l'information donnée par la dérivée de la fonction à minimiser (resp. maximiser). Il s'agit d'une procédure itérative d'optimisation qui utilise cette information pour ajuster les paramètres de la fonction. Il prend la valeur de la dérivée, la multiplie par une petite constante appelée *taux d'apprentissage* et soustrait le résultat à la valeur actuelle du paramètre. Ce processus est répété pour la nouvelle valeur du paramètre jusqu'à ce qu'un minimum (resp. maximum) soit atteint. Le taux d'apprentissage détermine la taille du pas et par conséquent la vitesse à laquelle la recherche converge. Si sa valeur est trop grande et que la fonction possède plusieurs minimums (resp. maximums) alors l'algorithme peut rater un minimum global (resp. maximum) ou osciller. S'il elle est trop petite, la progression vers le minimum (resp.maximum) sera lente.

Algorithme 1: Gradient descent

Data : f une fonction ; θ un ensemble de paramètres ; α le taux d'apprentissage.
Result : θ ajusté.
while $\frac{\partial}{\partial_\theta} f(\theta) \neq 0$ **do**
  for $0 \leq j \leq n$ **do**
    $\theta_j := \theta_j - \alpha \frac{\partial}{\partial_{\theta_j}} f(\theta)$
return θ

2.1.3 Classification bayesienne : naive bayes

Les classificateurs bayesiens sont des classificateurs statistiques. Ils peuvent prédire la probabilité d'appartenance à une classe donnée. Ceux-ci se basent sur le théorème de Bayes.

Théorème de Bayes

En théorie des probabilités, le théorème de Bayes permet de déterminer la probabilité a posteriori de x conditionnée par y, $\mathbb{P}(y|x)$, à partir de $\mathbb{P}(x)$, $\mathbb{P}(y)$ et $\mathbb{P}(x|y)$. Ces probabilités peuvent être estimées à partir des données de l'ensemble d'apprentissage. Ce théorème est donné par la formule suivante :

$$\mathbb{P}(y|x) = \frac{\mathbb{P}(x|y)\mathbb{P}(y)}{\mathbb{P}(x)}$$

Algorithme

Soit D l'ensemble d'apprentissage et m le nombre de classes. Le classificateur va associer à chaque vecteur x, la classe y_i ayant la plus grande probabilité a posteriori conditionnée par x :

$$\mathbb{P}(y_i|x) > \mathbb{P}(y_j|x) \qquad 1 \leq j \leq m, i \neq j$$

Par le théorème de Bayes nous savons que

$$\mathbb{P}(y_i|x) = \frac{\mathbb{P}(x|y_i)\mathbb{P}(y_i)}{\mathbb{P}(x)}$$

$\mathbb{P}(x), \mathbb{P}(y_i)$ et $\mathbb{P}(x|y_i)$ peuvent être estimées à l'aide des tuples de D. Les deux premières probabilités peuvent se calculer aisément. En effet, $\mathbb{P}(x)$ est constant pour chaque classe et $\mathbb{P}(y_i)$ n'est rien d'autre que la proportion d'élements de la classe y_i présents dans D, c'est-à-dire

$$\mathbb{P}(y_i) = \frac{|\{e|e \in y_i \text{ et } e \in D\}|}{|D|}$$

Par contre, la probabilité $\mathbb{P}(x|y)$ peut s'avérer très coûteuse à évaluer lorsque le nombre d'attributs devient élevé. Afin de simplifier les calculs, nous faisons l'hypothèse (naïve) que les attributs sont indépendants entre eux. Sa valeur est alors donnée par

$$\mathbb{P}(x|y_i) = \Pi_{k=1}^{n} \mathbb{P}(x_k|y_i)$$

Les $\mathbb{P}(x_k|y_i)$ peuvent elles aussi être calculées à partir de l'ensemble d'apprentissage. Pour chaque attribut, nous devons regarder s'il est nominal ou continu :

(a) Si a_k est nominal alors

$$\mathbb{P}(x_k|y_i) = \frac{|\{e = (e_1, e_2, \ldots, e_n)|e \in y_i, e \in D \text{ et } e_k = x_k\}|}{|\{e|e \in y_i \text{ et } e \in D\}|}$$

(b) Sinon si a_k est continu nous devons effectuer un peu plus de calculs. Une valeur continue est typiquement supposée suivre une loi de distribution gaussienne de moyenne μ et d'écart-type σ défini par

$$g(x, \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Et donc

$$\mathbb{P}(x_k|y_i) = g(x_k, \mu_{y_{ik}}, \sigma_{y_{ik}})$$

Nous avons donc besoin de connaitre $\mu_{y_{ik}}$ et $\sigma_{y_{ik}}$ étant respectivement la moyenne et l'écart-type de l'attribut a_k pour la classe y_i, calculables à partir de D.

2.1.4 Apprenant tardif : les k plus proches voisins

Les apprenants tardifs sont des classificateurs qui stockent les tuples de l'ensemble d'apprentissage et utilisent une fonction de distance afin de déterminer les distances séparant les tuples de cet ensemble d'un nouveau tuple à classer. Une distance souvent utilisée est la distance euclidienne. La distance euclidienne entre un tuple x_1 et un tuple x_2 est donnée par

$$dist(x_1, x_2) = \sqrt{\sum_{i=1}^{n}(x_{1i} - x_{2i})^2}$$

Il existe des alternatives à cette distance comme les distances de Manhattan, de Chebyshev ou de Minkowski. Le choix de cette métrique est souvent dépendant de l'interprétation qu'on lui confère.

Les attributs sont généralement mesurés dans différentes grandeurs. Donc si la distance euclidienne est calculée directement, l'effet de certains attributs pourrait être minimisé par rapport à d'autres. Par conséquent, il est habituel de les normaliser afin que leurs valeurs soient comprises entre 0 et 1 en calculant

$$\bar{x_i} = \frac{x_i - \min a_i}{\max a_i - \min a_i}$$

où x_i est la valeur actuelle de l'attribut et, où le minimum et le maximum sont pris parmi tous les tuples de l'ensemble d'entrainement.

Ces formules utilisent implicitement des valeurs numériques. Pour des attributs nominaux, la différence entre des valeurs identiques est souvent évaluée à 0 et à 1 lorsqu'elles sont différentes.

Le goulot d'étranglement de cet algorithme est la recherche des k plus proches voisins. La façon la plus simple de procéder est de calculer les distances à tous les tuples de l'ensemble d'apprentissage et de choisir les k plus petites. Cette procédure est linéaire en le nombre d'instances de l'ensemble d'apprentissage. Ces voisins peuvent être trouvés plus efficacement en utilisant des structures de données plus approriées pour stocker et rechercher des instances particulières (arbres kd, arbres à boules, arbres couvrants, etc).

Une fois que les k plus proches voisins ont été trouvés, la classe représentée majoritairement parmi ceux-ci est attribuée au tuple à classer.

2.1.5 Classification par arbre de décision

Un arbre de décision est un arbre où les noeuds représentent un test sur un attribut, chaque branche un résultat du test et chaque feuille une classe (voir FIGURE 2.4). Lorsqu'un tuple x

dont la classe est inconnue est présenté à l'arbre de décision, les valeurs de ses attributs sont testées par celui-ci. Un chemin est tracé de la racine de l'arbre à une feuille, qui détient la classe prédite pour ce tuple. Avant de voir comment un tel arbre est construit, nous allons examiner deux concepts qui seront utiles à sa construction.

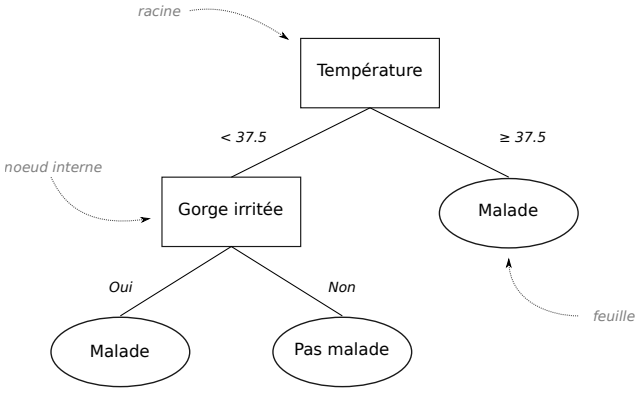


Figure 2.4 – *Un exemple d'arbre de décision comportant deux tests (température et gorge irritée) et 2 classes (malade et pas malade). Température est un attribut numérique continu et gorge irritée est un attribut nominal prenant deux valeurs : oui ou non.*

Critère de sélection des attributs

La mesure de sélection d'un attribut est une méthode heuristique permettant de sélectionner l'attribut qui séparera le mieux les tuples d'une partition D. Si nous devions séparer D en plus petites partitions conformément à ce critère, les tuples d'une partition appartiendraient idéalement à la même classe. On parle dans ce cas-là de partition pure. Cette mesure attribue à chaque attribut un score et l'attribut possédant le meilleur score est choisi pour séparer les tuples. Une mesure populaire pour la sélection des attributs est le *gain d'informations*.

Gain d'informations

Soit N le noeud contenant les tuples de la partition D. L'attribut permettant le plus grand gain d'informations est celui qui minimise l'information dont nous avons besoin pour classer les tuples dans les partitions résultantes et qui limite l'impureté dans ces partitions. L'information nécessaire pour classer un tuple de D est donnée par l'entropie :

$$Entropy(D) = \sum_{i=1}^{m} p_i \log_2(p_i)$$

où p_i est la probabilité qu'un tuple de D appartienne à la classe y_i.

Supposons que nous voudrions partitionner D selon un attribut a_i possédant k valeurs distinctes. Si a_i est discret, ses valeurs correspondent aux k résultats d'un test sur cet attribut. L'attribut a_i peut être utilisé pour séparer D en k partitions, $\{D_1, D_2, \ldots, D_k\}$, où D_j contient les tuples de D qui possèdent la valeur a_{ij} pour l'attribut a_i. Ces partitions correspondent aux branches du noeud N. Dans l'idéal, nous voudrions que chaque partition créée soit pure mais il est cependant plus probable que celle-ci soit impure. De ce fait, il nous faut calculer la quantité d'informations encore nécessaires (après partitionnement) pour arriver à une classification exacte. Cette quantité est donnée par

$$Entropy_{a_i}(D) = \sum_{j=1}^{k} \frac{|D_j|}{|D|} \times Entropy(D_j)$$

Le terme $\frac{|D_j|}{|D|}$ est le poids associé à la i^{eme} partition. $Entropy_{a_i}(D)$ est donc la quantité d'informations requises pour classer un tuple de D après avoir partitionné sur l'attribut a_i. Plus cette quantité est faible et plus les partitions seront pures. Le gain d'informations est défini comme la différence entre la quantité d'informations requises au départ et celles requises après partitionnement

$$Gain(a_i) = Entropy(D) - Entropy_{a_i}(D)$$

L'attribut a_i permettant le plus grand gain d'informations est choisi comme attribut pour N.

Lorsque a_i est un attribut numérique continu, nous devons déterminer le meilleur point de séparation. Dans un premier temps, nous devons trier les valeurs de a_i par ordre croissant. Ensuite, les valeurs intermédiaires entre chaque paire de valeurs adjacentes sont considérées comme des points de séparation potentiels. Pour chacune d'entre elles nous évaluons $Entropy_{a_i}(D)$, où le nombre de partitions k vaut 2, et sélectionons le point p pour lequel cette valeur est minimale. D_1 est alors l'ensemble des tuples qui satisfont $x_i \leq p$ et D_2 l'ensemble des tuples tels que $x_i > p$.

Élagage de l'arbre

Les arbres de décision complètement developpés contiennent souvent des structures inutiles comme des sous-arbres dupliqués et il est souvent conseillé de le simplifier avant de le déployer. Il existe deux façons d'élaguer un arbre : le pré-élagage et le post-élagage.

Le pré-élagage nécessite de décider durant le processus de construction de l'arbre quand arrêter de développer les sous-arbres. Lorsque l'arbre est en cours de construction, des mesures comme le gain d'informations peuvent être utilisées afin d'évaluer la qualité d'une séparation. Si partitionner conduit à un résultat inférieur à une certaine valeur seuil fixée, le partionnement de ce sous-arbre est stoppé. Il est cependant difficile de déterminer une valeur appropriée pour le seuil. Une grande valeur seuil peut donner un ordre trop simplifié et une valeur trop faible peut conduire à de trop petites simplifications.

La seconde approche, et la plus courante, est le post-élagage qui supprime des sous-arbres une fois l'arbre entièrement construit et les remplace par des feuilles. Cette méthode possède un avantage par rapport à la précédente. Il y a des situations où deux attributs contribuent peu individuellement mais forment un très bon prédicteur une fois combinés. Lorsque l'arbre est pré-élagué, ce type de combinaison peut être manqué. Bien que la plupart des arbres de décision soient post-élagués, le pré-élagage est une alternative viable lorsque le temps d'exécution est une mesure critique.

Construction de l'arbre

Algorithme 2: Generate_decision_tree

Data : D un ensemble d'apprentissage ; $attributes$ les attributs à considérer
Result : Un arbre de décision
Créer un noeud N;
if *les tuples de D appartiennent à la même classe* **then**
  return *N étiqueté avec cette classe*
if $attributes$ *est vide* **then**
  return *N étiqueté par la classe majoritaire dans D*
Calculer $Gain$ pour chaque attribut;
$A \leftarrow$ attribut apportant le plus d'informations;
Calculer les séparations pour A;
foreach *séparation de A* **do**
  $D_j \leftarrow$ tuples de D qui satisfont le critère de séparation;
  if D_j *est vide* **then**
    Âttacher un noeud étiqueté avec la classe majoritaire de D à N;
  else
    Attacher le noeud retourné par $Generate\_decision\_tree(D_j, attributes)$ à N;
return *Le noeud N*

2.1.6 Les forêts d'arbres décisionnels : Random Forest

L'algorithme Random Forest a été introduit en 2001 par Leo Breiman et Adele Cutler [15]. Il s'agit d'une méthode d'apprentissage par ensembles qui utilise plusieurs arbres de décision afin d'obtenir de meilleures performances. Elle s'appuie sur deux concepts importants : le bagging et les sous-espaces aléatoires.

Biais et variance

Avant de parler des concepts cités ci-dessus, nous allons introduire les notions de biais et de variance car elle interviendront par la suite. Supposons que nous puissions avoir un nombre infini d'ensembles d'entrainement et que nous les utilisions pour créer un nombre infini de classificateurs. Un tuple de test est alors traité par tous les classificateurs et sa classe est déterminée par un vote majoritaire. L'erreur induite par la combinaison de plusieurs modèles peut être décomposée en biais et en variance.

Supposons que le taux d'erreurs attendu soit évalué par la moyenne des erreurs des classificateurs combinés sur un certain nombre de tuples indépendants. Le taux d'erreurs pour un algorithme d'apprentissage particulier est appelé son **biais** et mesure à quel point la méthode correspond au problème.

Une seconde source d'erreur est issue de l'ensemble d'apprentissage utilisé, qui est inévitablement fini et donc pas complètement représentatif de la population. La valeur de cette composante de l'erreur, parmi tous les ensembles d'entrainement et de tests possibles d'une certaine taille, est appelée la **variance**. La valeur totale de l'erreur attendue est donc la somme du biais et de la variance.

Généralement, la combinaison de plusieurs classificateurs diminue l'erreur en réduisant la

composante de variance. Plus il y a de classificateurs et plus la variance est réduite. Cependant, en pratique il est difficile d'obtenir un tel schéma car habituellement nous ne disposons que d'un ensemble d'entrainement et obtenir plus de données est soit impossible, soit coûteux.

Bagging

Le bagging est une méthode qui tente de neutraliser la composante de variance en simulant le procédé décrit ci-dessus en utilisant un seul ensemble d'entrainement. Au lieu d'utiliser un ensemble d'entrainement indépendant pour chaque classificateur, l'ensemble d'entrainement de départ est modifié en supprimant certaines instances et en en dupliquant d'autres. Autrement dit, les tuples sont échantillonnés aléatoirement avec replacement à partir de l'ensemble d'entrainement original pour en créer un nouveau de la même taille. Cette technique est connue sous le nom de **bootstrap**, d'ailleurs bagging vient de bootstrap aggregating. Le bagging va donc appliquer le schéma d'apprentissage à chacun des ensembles d'apprentissage artificiels et les classificateurs générés vont voter pour élire la classe à prédire.

Sous-espaces aléatoires

Dans leur papier original sur Random Forest, les auteurs ont montré que le taux d'erreurs d'une forêt dépend de deux choses :

* des corrélations entre paires d'arbres de la forêt. Augmenter les corrélations augmente le taux d'erreurs.
* de la robustesse de chaque arbre de la forêt. Un arbre avec un faible taux d'erreur est un classificateur robuste. Augmenter la robustesse permet de diminuer le taux d'erreur de la forêt.

Partant de ce constat, ils ont eu l'idée d'introduire les sous-espaces aléatoires. Au lieu d'utiliser l'ensemble complet des attributs pour l'apprentissage, chaque arbre de la forêt est entrainé sur un sous-échantillon aléatoire d'attributs de taille m. Réduire m permet de réduire à la fois la corrélation et la robustesse. À l'inverse, augmenter m les accroit toutes les deux. Entre les deux, il existe un intervalle habituellement large de valeurs optimales pour m.

Algorithme

Les algorithmes d'apprentissage et de classification sont relativement simples et se trouvent respectivement ci-dessous.

Algorithme 3: Generate_Random_Forest

Data : D un ensemble d'apprentissage ; t le nombre d'arbres de la forêt ; m le nombre d'attributs à utiliser pour chaque arbre ; $attributes$ une liste d'attributs
Result : Une forêt d'arbres décisionnels
for $1 \leq i \leq t$ **do**
    $attributes_i \leftarrow$ choisir aléatoirement m attributs parmi $attributes$;
    $D_i \leftarrow$ construire un échantillon boostrap à partir de D;
    Stocker $Generate\_Decision\_tree(D_i, attributes_i)$;

Algorithme 4: Predict_Random_Forest

Data : Un tuple x dont nous voulons prédire la classe ; les t classificateurs
Result : La classe du tuple x
foreach *classificateur* **do**
  Prédire la classe de x;
return *La classe prédite le plus souvent*

2.1.7 Réseaux de neurones artificiels

Un réseau de neurones artificiels, souvent appelé plus simplement réseau de neurones, est un modèle mathématique inspiré du fonctionnement des réseaux de neurones biologiques. Un réseau de neurones se compose de neurones artificiels interconnectés traitant de l'information selon une approche connexioniste.

Connexionisme

Le connexionisme est un courant de recherche et les réseaux de neurones sont les modèles connexionistes les plus répandus à l'heure actuelle. Ils possèdent deux caractéristiques :

* un état peut être représenté par un vecteur à n dimensions correspondant aux valeurs d'activation des unités neuronales ;
* un réseau de neurones peut apprendre en modifiant les poids des connexions entre ses différentes unités.

Nous verrons au travers des prochaines sections que ces caractéristiques sont bien présentes.

Le neurone artificiel

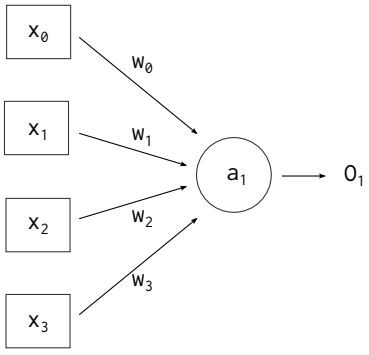


Figure 2.5 – *Un exemple de topologie à un seul noeud. Le noeud calcule son entrée sur base des valeurs qu'elle reçoit de ses connexions, calcule l'activation et la fournit en sortie*

Un neurone artificiel est composé de 3 éléments :

* des connexions pondérées ;
* une unité de calcul a ;
* une valeur de sortie ;

Lorsqu'un neurone reçoit des valeurs par ses connections, il calcule leur somme pondérée. Cette somme est appelée l'entrée du neurone. L'entrée d'un neurone pour un tuple i est donc donnée par

$$I_i = w_0 + \sum_{j=1}^{n} w_j x_j$$

où w_0 est le biais du neurone. La sortie du neurone est fonction de cette entrée. La valeur de cette fonction est appelée l'**activation** du neurone. Il existe plusieurs fonctions d'activation parmi lesquelles on retrouve :

* la fonction linéaire $f(a) = a$.
* la fonction sigmoïde $f(a) = \dfrac{1}{1 + e^{-a}}$

À lui seule, un neurone est donc capable d'effectuer une régression linéaire ou logistique. Un exemple de neurone artificiel est donné à la FIGURE 2.5. L'intérêt des neurones artificiels est qu'ils peuvent être interconnectés pour former un réseau capable de produire des modèles complexes.

Hierarchie de neurones

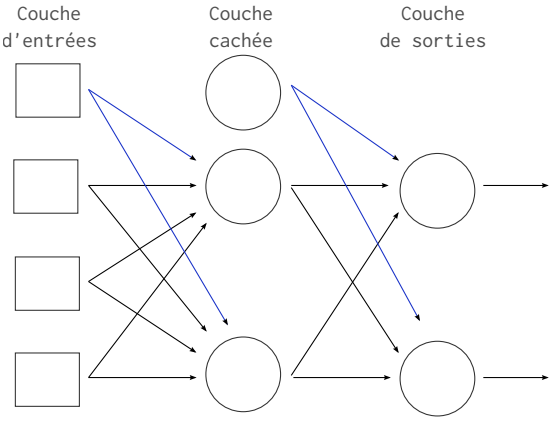


FIGURE 2.6 – *Un exemple de topologie comportant une couche d'entrées, une couche cachée et une couche de sorties. Chaque noeud est connecté à l'ensemble des noeuds de la couche suivante. Les noeuds d'où partent des connexions bleues représentent les biais. La valeur de sortie des noeuds de la couche de sorties donne la classe pour le tuple dont les valeurs des attributs ont été fournies à la couche d'entrées*

Les réseaux de neurones sont organisés en couches. La première couche est appelée la **couche d'entrées** car elle est constituée de la valeur des attributs d'un tuple à classer ; les couches intermédiaires sont appelées les **couches cachées** ; et la dernière couche est appelée la **couche de sorties** car elle fournit le résultat de la classification. Les neurones de chaque couche sont complètement reliés aux neurones de la couche suivante. L'entrée pour un neurone j d'une couche cachée ou de la couche de sorties est donnée par la somme pondérée des valeurs de sorties des neurones de la couche précédente. Autrement dit, l'entrée I_j de l'unité j est donnée par

$$I_j = w_{0j} + \sum_i w_{ij} O_i$$

où w_{ij} est le poids de la connexion entre l'unité i de la couche précédente et j ; O_i est la valeur de sortie de l'unité i de la couche précédente ; et w_{0j} le biais de l'unité j. La FIGURE 2.6 illustre un tel réseau.

Comme dans d'autres algorithmes vu précédemment, l'apprentissage d'un réseau de neurones consiste à trouver les poids w des connections qui permettent d'ajuster aux mieux le modèle aux données de l'ensemble d'apprentissage. Pour cela nous allons utiliser un algorithme nommé *Backpropagation*. Pour chaque tuple de l'ensemble d'apprentissage, l'algorithme va comparer la classe prédite par le réseau avec la classe réelle, et modifier les poids des connections afin de minimiser l'erreur quadratique. L'algorithme se déroule en 3 étapes :

Initialiser les poids. Les poids des connexions sont généralement initialisés avec des petites valeurs aléatoires (par exemple entre -1 et 1). Chaque neurone possède un biais qui est également initialisé avec une petite valeur aléatoire. Ensuite chaque tuple est traité par les deux prochaines étapes.

Propagation des entrées. Premièrement, le tuple de l'ensemble d'apprentissage est amené à la couche d'entrée. Ensuite, les neurones de chaque couche calculent leurs entrées et transmettent leurs sorties à la couche suivante jusqu'à atteindre la couche des sorties. La sortie d'un neurone est l'application de la fonction d'activation sur l'entrée calculée. Prenons comme fonction d'activation la fonction sigmoïde car elle est non-linéaire et différentiable est permet donc de modéliser des problème étant linéairement inséparables. La sortie O_j d'une unité j est donc donnée par

$$O_j = \frac{1}{1 + e^{-I_j}}$$

Rétropropagation de l'erreur. L'erreur est rétropropagée vers les premières couches en actualisant les poids et les biais afin de renvoyer l'erreur de prédiction. Pour un neurone j de la couche de sortie, l'erreur Err_j est donnée par

$$Err_j = O_j(1 - O_j)(T_j - O_j)$$

où O_j est la valeur de sortie du neurone j, et T_j est la valeur d'appartenance du tuple à la classe y_j. Si le tuple appartient à y_j alors $T_j = 1$ et 0 sinon. Notons que $O_j(1 - O_j)$ est la dérivée de la fonction sigmoïde.

Pour calculer l'erreur d'un neurone j d'une couche cachée, la somme pondérée des erreurs des unités connectées à j dans la couche suivante est calculée. L'erreur pour cette unité est alors donnée par

$$Err_j = O_j(1 - O_j) \sum_k w_{jk} Err_k$$

où w_{jk} est le poids de la connection reliant l'unité j à l'unité k de la couche suivante, et Err_k est l'erreur à l'unité k. Les poids et le biais sont actualisés afin de prendre en compte l'erreur propagée. Les poids sont mis à jour de cette façon :

$$\begin{aligned} \Delta w_{ij} &= \alpha Err_j O_i \\ w_{ij} &= w_{ij} + \Delta w_{ij} \end{aligned}$$

Les biais, quant à eux, sont mis à jour de la façon suivante :

$$\begin{aligned} \Delta w_{0j} &= \alpha Err_j \\ w_{0j} &= w_{0j} + \Delta w_{0j} \end{aligned}$$

La quantité α est le taux d'apprentissage. En fait, l'algorithme *Backpropagation* est une adaptation de l'algorithme *gradient descent* (voir section 2.1.2) aux réseaux de neurones. Les incréments pour les poids et les biais peuvent être accumulés dans une variable de façon à ce que les poids et les biais soient actualisés une fois que tous les tuples de l'ensemble d'apprentissage aient été présentés. Cette stratégie est appelée l'actualisation par époque, où une itération parmi tous les tuples de l'ensemble d'apprentissage correspond à une époque.

L'apprentissage s'arrête au moment où une certaine condition est satisfaite. Ces conditions peuvent être :

* tous les Δw_{ij} de la dernière époque sont inférieurs à une certaine valeur seuil.
* le pourcentage de tuples mal classés durant la dernière époque est inférieur à une certaine valeur seuil.
* un nombre fixé d'époques a expiré.

L'algorithme *Backpropagation* est illustré ci-dessous.

Algorithme 5: Backpropagation

Data : D l'ensemble d'apprentissage ; α le taux d'apprentissage ; *network* le réseau de neurones
Result : Le réseau de neurones entrainé
Initialiser tous les poids et biais du réseau;
while *condition d'arrêt non satisfaite* **do**
  foreach *tuple de l'ensemble d'entrainement* **do**
    foreach *unité j de la couche d'entrées* **do**
      $O_j = I_j$;
    foreach *unité j n'appartenant pas à la couche d'entrées* **do**
      $I_j = w_{0j} + \sum_i w_{ij} O_i$;
      $O_j = \frac{1}{1+e^{-I_j}}$;
    foreach *unité j de la couche de sorties* **do**
      $Err_j = O_j(1 - O_j)(T_j - O_j)$;
    foreach *unité j des couches cachées (de la dernière à la première)* **do**
      $Err_j = O_j(1 - O_j) \sum_k w_{jk} Err_k$;
    foreach *poids* w_{ij} *dans network* **do**
      $\Delta w_{ij} = \alpha Err_j O_i$;
      $w_{ij} = w_{ij} + \Delta w_{ij}$;
    foreach *biais* w_{0j} *dans network* **do**
      $\Delta w_{0j} = \alpha Err_j$;
      $w_{0j} = w_{0j} + \Delta w_{0j}$;

2.1.8 Machines à vecteurs de support

Dans cette section nous allons étudier les machines à vecteurs de support, souvent abrégées par SVM [18]. Introduite par Vladimir Vapnik en 1992, cette méthode utilise des modèles linéaires

pour implémenter des frontières de classification non linéaires. Pour cela, on applique une fonction non linéaire aux données de l'ensemble d'entrainement afin de les transposer dans un espace de plus grande dimension. Dans cette espace de plus grande dimension, l'algorithme cherche l'hyperplan séparateur optimal. Cette frontière, qui est linéaire dans cet espace, est non linéaire dans l'espace de départ. L'hyperplan est trouvé à l'aide de vecteurs de support (les tuples indispensables de l'ensemble d'apprentissage) et de marges (définies par les vecteurs de support).

Hyperplan de marge maximale et vecteurs de support

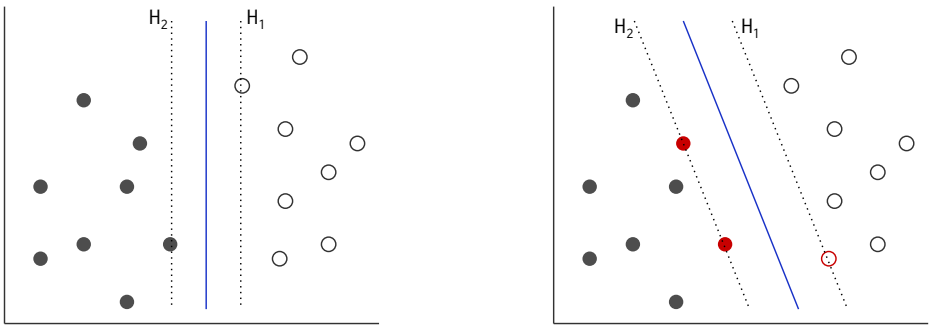


Figure 2.7 – *Une séparatrice quelconque (à gauche) et la séparatrice trouvée par SVM (à droite). La séparatrice trouvée par SVM est celle dont la marge est maximale (en pointillés). Les points en rouge représentent les vecteurs de support car ils se trouvent le long de la marge.*

Imaginons un problème à deux classes, $y \in \{-1, +1\}$, linéairement séparables. Il existe un nombre infini de droites pouvant séparer les instances des deux classes. Ce que nous voulons obtenir est la meilleure droite permettant de séparer ces points, c'est-à-dire la droite permettant d'obtenir un minimum d'erreurs de classification sur de futures données. En généralisant à n dimensions, on parle d'hyperplan séparateur.

Les SVM abordent ce problème en cherchant l'**hyperplan de marge maximale**. Examinons la figure 2.7 qui expose deux hyperplans séparateurs possibles et leurs marges respectives. Les deux hyperplans sont tous les deux capables de classer les tuples. Cependant on s'attend intuitivement à ce que l'hyperplan possédant la plus grande marge les classe mieux car la séparation entres les tuples de chaque classe est plus grande et donc il y a moins de risques de classer incorrectement les données. Un hyperplan séparateur est décrit par

$$w_0 + wx = 0$$

où $w = (w_1, w_2, \ldots, w_n)$ est un vecteur de poids et w_0 est le biais. On peut donc définir les « côtés » comme

$$\begin{aligned} H_1 &: w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n \geq +1 \qquad \text{si } y_i = +1 \\ H_2 &: w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n \leq -1 \qquad \text{si } y_i = -1 \end{aligned}$$

De ce fait, tout tuple se trouvant sur ou au-dessus de H_1 appartient à la classe +1 et tout tuple qui est sur ou en-dessous de H_2 appartient à la classe −1. En combinant ces deux inéquations on obtient

$$y_i(w_0 + w_1x_1 + w_2x_2 + \cdots + w_nx_n) \geq 1 \quad , \forall i$$

Les tuples de l'ensemble d'apprentissage se trouvant exactement sur les hyperplans H_1 ou H_2 sont appelés les **vecteurs de support**. Ces vecteurs se trouvent à égale distance de l'hyperplan séparateur. Ils sont les plus difficiles à classer et fournissent le plus d'informations concernant la classification. En effet, on peut définir l'hyperplan de marge maximale uniquement à l'aide des vecteurs de support. La distance qui les sépare de l'hyperplan séparateur est la taille de la marge et vaut $\frac{1}{||w||}$ où $||w||$ est la norme Euclidienne de w.

L'hyperplan de marge maximale est donné par

$$d(x) = w_0 + \sum_{i=1}^{l} y_i \alpha_i x_i \cdot x$$

où y_i est la classe du vecteur de support x_i ; x est un tuple de test ; l est le nombre de vecteurs de support ; α_i et w_0 sont des paramètres qui déterminent l'hyperplan. Il s'avère que trouver les vecteurs de support et déterminer les paramètres α_i et w_0 revient à résoudre un problème d'optimisation quadratique sous contrainte. Il existe plusieurs algorithmes permettant de résoudre un tel problème parmi lesquels on retrouve *minimal sequential optimization* (SMO) de John Platt [19]. Une fois le classificateur entrainé, la classe d'un tuple x est donnée par le signe de $d(x)$.

Fonction kernel

Actuellement nous sommes capable de créer des SVM pour des données linéairement séparables. Nous pouvons facilement étendre l'approche vu précédemment pour des données linéairement inséparables en utilisant une **fonction kernel**. Cette fonction est appliquée aux données d'origine et permet de chercher une séparatrice linéaire dans un espace de plus grande dimension et donc d'obtenir une séparatrice non linéaire dans l'espace d'origine. Les fonctions kernel les plus utilisées sont :

* le kernel polynomial de degré h : $K(x_i, x_j) = (x_i \cdot x_j + 1)^h$
* le kernel à base radiale gaussienne : $K(x_i, x_j) = e^{-\frac{||x_i - x_j||^2}{2\sigma^2}}$
* le kernel sigmoïde : $K(x_i, x_j) = tanh(\kappa x_i \cdot x_j - \delta)$

L'hyperplan de marge maximale est dès lors donné par

$$d(x) = w_0 + \sum_{i=0}^{l} y_i \alpha_i K(x_i, x)$$

2.2 Correction statistique de l'erreur

Dans cette section nous allons étudier une méthode de correction statistique de l'erreur mise au point par *Solow*, *Hu* et *Davis*. Celle-ci se base sur l'article [20].

Lorsqu'un ensemble de tuples est classé automatiquement et que la classification contient des erreurs, le nombre de tuples contenus dans chaque classe est une estimation biaisée du nombre

de tuples appartenant réellement à chacune d'entre elles. La méthode de *Solow et al.* propose de corriger ce biais en utilisant les probabilités de classification du classificateur.

Plus formellement, considérons le vecteur $n = (n_1, n_2, \ldots, n_s)^t$ où n_i représente le nombre de tuples appartenant réellement à la classe i. Bien que les éléments de n soient inconnus, leur total $N = \sum_{i=1}^{s} n_i$ est lui connu. Soit p_{jk} la probabilité qu'a le tuple j d'appartenir à la classe k et $P = [p_{jk}]$ la matrice de ces probabilités. Soit M_j le nombre de tuples classés dans la classe j. Cette variable aléatoire peut s'écrire comme

$$M_j = \sum_{i=1}^{s} M_{ij}$$

où M_{ij} est le nombre de tuples de la classe i classés dans j. La valeur espérée pour M_{ij} est $n_i p_{ij}$ et donc la valeur espérée pour M_j est

$$E(M_j) = \sum_{i=1}^{s} n_i p_{ij}$$

ce qui en notation matricielle s'écrit

$$E(M) = P^t n$$

où $M = (M_1, M_2, \ldots, M_n)$ et E dénote l'espérance mathématique. Un estimateur non biaisé pour n est alors

$$\hat{n} = (P^t)^{-1} m$$

où $m = (m_1, m_2, \ldots, m_s)^t$ est l'ensemble des valeurs observées pour M. Afin d'avoir une idée de la précision de l'estimation, l'écart-type peut être calculé en prenant la racine carrée des éléments sur la diagonale de la matrice de variance. Posons $Q = (P^t)^{-1}$ de sorte que $\hat{n} = Qm$. La variance pour $\hat{n}$ est alors donnée par

$$Var(\hat{n}) = QVar(M)Q^t$$

2.3 Applications

Cette section présente un état de l'art des différentes techniques de classification appliquées pour l'identification automatique de planctons.

2.3.1 Hu et Davis (2005)

Il existe beaucoup de systèmes optiques capables de produire des images digitales de planctons pouvant être classées automatiquement par ordinateur. Le problème est que ces systèmes de classification ne sont pas fiables à 100%, surtout dans les zones où l'abondance relative de certains planctons est faible. Hu et Davis [21] ont pour cela construit un système amélioré de classification utilisant une machine à vecteurs de support et des matrices de co-occurrence.

Matrice de co-occurrence

Une matrice de co-occurrence est une matrice définie par une image comme étant la distribution de valeurs concomitantes à un décalage donné. Mathématiquement, la matrice de co-occurrence U définie par une image I de taille $n \times m$ et par les décalages Δ_x et Δ_y est donnée par :

$$U_{\Delta_x,\Delta_y}(i,j) = \sum_{p=1}^{n}\sum_{q=1}^{m} \begin{cases} 1 \text{ si } I(p,q) = i \text{ et } I(p+\Delta_x, q+\Delta_y) = j \\ 0 \text{ sinon} \end{cases}$$

où i et j sont des valeurs de pixels en niveaux de gris [1]. Cette matrice permet donc d'estimer la probabilité d'avoir un pixel i à une distance d par rapport à un angle α d'un pixel j. Dans leur étude, Hu et Davis ont choisi de prendre $\alpha = 0, 45, 90, 135$ degrés et $d = 1, 4, 8, 16$ pixels. On peut en déduire des caractéristiques de textures. Un exemple est donné à la FIGURE 2.8

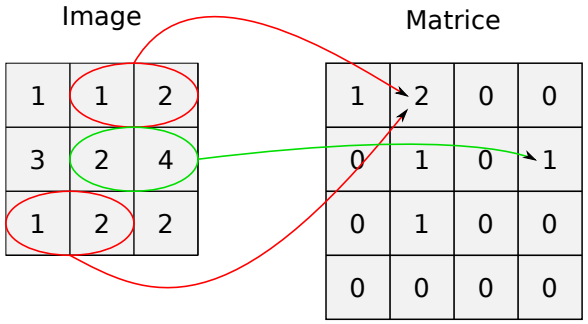


FIGURE 2.8 – *Exemple d'une matrice de co-occurrence appliquée sur une image à 4 niveaux de gris avec $\alpha = 0$ et $d = 1$.*

Machine à vecteurs de support

Etant donné qu'une machine à vecteurs de support est un classificateur binaire, Hu et Davis ont eu recours à une méthode **one-versus-one**. Elle consiste à considérer les $\frac{M(M-1)}{2}$ paires de classes possibles et à créer pour chacune d'elles une machine. Les échantillons sont alors présentés à ces machines et un vote majoritaire permet de déterminer leur classe.

Matrice de confusion

		Classes prédites	
		Classe 1	Classe 2
Classes réelles	Classe 1	vrais positifs	faux négatifs
	Classe 2	faux positifs	vrais négatifs

FIGURE 2.9 – *Une matrice de confusion pour un problème à deux classes*

Une matrice de confusion est un outil servant à mesurer la qualité d'un classificateur. Chaque colonne de cette matrice représente le nombre d'occurences d'une classe estimée tandis que chaque ligne représente le nombre d'occurences d'une classe réelle. Chaque entrée en ligne i et colonne j montre le nombre de tuples de la classe i qui ont été classés dans la classe

1. Une image en niveau de gris est une image où seule l'intensité des pixels est retenue. La valeur de l'intensité varie de 0 au niveau de gris (max. 256)

j. Idéalement, les éléments en dehors de la diagonale doivent valoir 0 ou en être proches. Plusieurs métriques peuvent être calculées à l'aide de cette matrice.

Considérons les 2 classes de la FIGURE 2.9 et désignons les tuples positifs comme étant les tuples de la classe 1 et les tuples négatifs comme étant ceux de la classe 2. Les vrais positifs sont les tuples positifs ayant été correctement classés par le clasisificateur tandis que les vrais négatifs sont les tuples négatifs ayant été correctement classés par le classificateur. De la même manière, on définit les faux positifs comme les tuples négatifs qui ont mal été étiquetés. De la même manière, les faux négatifs sont les tuples positifs ayant été incorrectement classés.

Nous pouvons également caractériser comment le classificateur reconnait les vrais positifs et les vrais négatifs. Le rappel et la spécificité peuvent être respectivement utilisés pour ces raisons. Le rappel est souvent désigné comme le taux de vrais positifs ou taux de reconnaissance :

$$recall = \frac{\#\text{ vrais positifs}}{\#\text{ tuples positifs}}$$

De la même manière, la spécificité est souvent désignée comme étant le taux de vrais négatifs :

$$specificity = \frac{\#\text{ vrais négatifs}}{\#\text{ tuples négatifs}}$$

On peut également utiliser la précision qui est le pourcentage de tuples classés dans la classe principale et qui appartiennent réellement à cette classe :

$$precision = \frac{\#\text{ vrais positifs}}{\#\text{ vrais positifs} + \#\text{ faux positifs}}$$

Plus ces valeurs sont proches de 1 et meilleure est la qualité prédictive du classificateur pour ces données. Dans certains cas, il est possible d'obtenir une classification précise (precision élevée) mais peu performante (recall élevé). Autrement dit, les tuples classés dans la classe principale sont corrects mais peu de tuples de la classe principale ont pu être identifiés. La situation inverse peut également se produire : le classificateur peut classer correctement la majorité des tuples de la classe principale mais aussi identifier à tort un grand nombre de tuples qui n'en font pas partie. Un raisonnement similaire peut être effectué pour la spécificité.

Expérimentation

Pour leur étude, Hu et Davis ont récolté plus de 20.000 images à l'aide de VPR et les ont classées manuellement dans 7 groupes taxonomiques différents (comprenant une classe « autre »). En tout, 21% des images se trouvent dans cette classe « autre ». Chaque image a ensuite été quantifiée en 16 niveaux de gris et leur matrice de co-occurrence a été calculée. Plusieurs attributs peuvent être calculés à partir de cette matrice (contraste, entropie, etc). Le kernel qu'ils ont utilisé pour SVM est le kernel gaussien. Pour chaque classe, 200 images ont été extraites afin d'entrainer le classificateur et de valider les résultats. Le reste des images ont quant à elles été utilisées pour générer la matrice de confusion et estimer la qualité de la prédiction.

Résultats

Une petite partie des résultats qu'ils ont obtenu est disponible à la TABLE 2.1. Les auteurs ont testé plusieurs kernels et ont mesuré les taux de reconnaissance du classificateur en faisant varier les paramètres (le paramètre C est une pénalité ajoutée à chaque instance mal classée

lors de la phase d'apprentissage). Concernant le kernel gaussien, le taux de reconnaissance varient très peu en fonction de la pénalité fixée et ce pour des valeurs de C allant de 10 à 500. Ce taux est cependant un peu plus sensible au paramètre σ. Concernant le kernel polynomial, le taux de reconnaissance varie entre 69 et 74% pour des degrés de polynôme allant de 1 à 6. Le taux de reconnaissance varie également peu lorsque le noyau sigmoïde est utilisé. Entre tous les kernels, les meilleurs taux de reconnaissance diffèrent seulement de 1%. Leurs résultats montrent donc que leurs SVM sont robustes à la fois au type de kernel et aux paramètres spécifiques à chacun d'eux.

Kernel gaussien ($C = 50$)							
σ	0.02	0.5	1	2	3	5	10
Taux	63	69	72	75	75	73	70
Kernel gaussien ($\sigma = 2$)							
C	10	20	50	100	200	500	
Taux	73	75	75	75	74	73	
Kernel polynomiale ($C = 5$)							
d	1	2	3	4	5	6	
Taux	74	74	74	72	69	69	
Kernel sigmoïde							
k	0.01	0.02	0.05	0.1	0.2	0.5	
Taux	69	71	73	74	72	60	

Table 2.1 – *Les performances du classificateur avec différents types de kernels et différents paramètres.*

2.3.2 Hu et Davis (2006)

Un des problèmes de la classification est que sa précision est limitée, il faut alors corriger manuellement les résultats obtenus pour obtenir une bonne estimation de l'abondance des planctons. Une solution proposée par Hu et Davis en 2006 [22] fut d'utiliser un double classificateur : un réseau de neurones utilisant des informations sur la forme des planctons et une machine à support de vecteurs se basant sur leur texture. Un plancton appartient à un groupe si les deux classificateurs sont en accord. Dans le cas contraire, il est étiqueté comme étant inconnu. Cette méthode a l'avantage de réduire grandement le nombre de faux-positifs et donne donc une meilleure estimation de leur abondance. Une matrice de confusion est ensuite calculée à partir de l'ensemble d'entrainement pour déterminer le taux de détection et de faux-positifs, ce qui permet de corriger l'estimation. Les images utilisées ont été récoltées par le VPR.

Les caractéristiques de forme

Les informations de formes extraites sont de 4 types :

* les moments invariants : il s'agit d'une mesure quantitative de la forme d'un ensemble de points. Ici, ces points sont pondérés par l'intensité des pixels de l'image. Les moments utilisés dans le cadre de l'expérience sont : le moment centré réduit, la translation, la rotation et l'invariant d'échelle.
* les mesures morphologiques : pour cette étude, 6 mesures ont été effectuées à partir des moments calculés précédemment.
* les descripteurs de Fourier : ce sont des caractéristique invariantes utilisées pour décrire les contours d'un objet.

* la granulométrie.

Au total, 7 moments invariants, 6 mesures morphologiques, 64 descripteurs de Fourier et 160 mesures granulométriques ont été calculés. Une analyse en composante principale a ensuite permis de garder les 30 attributs les plus représentatifs.

Les caractéristiques de texture

Celles-ci ont été obtenues en calculant les matrices de co-occurrence pour 4 distances (1,4,8,16 pixels) et 4 angles (0,45,90,135 degrés) différents. Sur base de ces 16 matrices, plusieurs mesures ont pu être extraites et utilisées comme attributs. En tout, pour chaque image, 64 caractéristiques de ce type ont été utilisées.

Le réseau de neurones

Ce réseau de neurones est composé de 2 couches. La première couche comprend 20 neurones et la seconde correspond aux différentes classes de planctons. Les poids des connections pour chaque classe ont été initialisés avec les valeurs moyennes des caractéristiques de celles-ci plus un petit nombre aléatoire.

Les machines à vecteurs de support

Pour cette étude, le kernel choisi est le linéaire car il ne nécessite pas d'être paramétré. De plus, lors de leur précédente étude (voir section 2.3.1), Hu et Davis ont montré que leurs SVM ne sont pas très sensibles au type de kernel. Étant donné qu'il s'agit de classificateurs binaires, une SVM a été créé pour chaque paire distincte de classes afin d'obtenir une classification multiclasses.

La correction d'abondance

Hu et Davis corrigent l'abondance à partir du recall et de la précision du classificateur pour chaque classe. Soit $\delta = (\delta_1, \delta_2, \ldots, \delta_n)$ le vecteur correcteur avec $\delta_i = \frac{precision_i}{recall_i}$ le facteur correcteur de la classe i et $B = (b_1, b_2, \ldots, b_n)$ l'abondance obtenue par le classificateur. Le résultat de la correction est donné par $\delta \cdot B$. Le principal inconvénient de cette méthode est quelle nécessite que le recall et la précision pour chaque taxon varient relativement peu.

Expérimentation

L'expérience est effectuée sur plus de 20.000 images récoltées à l'aide du VPR. Ces images ont été manuellement classées dans 7 groupes taxonomiques différents. Étant donné leurs précédentes expériences, ils ont jugé raisonnable d'utiliser 200 images par taxon pour le set d'entrainement. Les particules inconnues ont été groupées dans une catégorie "autre".

Résultats

Pour les classes ayant une abondance relative importante, les performances du double classificateur sont très similaires à celle d'un simple classificateur. Cependant, pour les classes dont l'abondance relative est faible, le double classificateur obtient de bien meilleurs résultats car le nombre de faux-positifs est beaucoup plus faible. De ce fait, le recall et la précision varient moins en fonction de l'abondance relative d'une classe rendant ainsi possible la correction automatique. Les résultats obtenus par la double classification ont été comparés avec les résultats obtenus par classification manuelle des données et sont presque semblables. Il

faut savoir qu'une étude menée par Culverhouse et al. en 2003 [23] a montré qu'un personnel entrainé est capable de classer correctement entre 67 et 83% des images lorsque la tâche est difficile (un grand nombre d'images par exemple). De plus, l'abondance est généralement mieux estimée à l'aide de ce système. En effet, dans les régions où l'abondance relative est faible, cette méthode permet de réduire entre 50 et 100% des erreurs d'estimation comparée à d'autres méthodes.

Chapitre 3| Le projet

Dans ce chapitre nous allons détailler le travail qui a du être réalisé pour ce projet. Dans un premier temps nous détaillerons les principaux aspects du projet. Ensuite, nous analyserons la structure du code source et tenterons de l'améliorer. Par la suite, nous verrons les différentes modifications et nouvelles fonctionnalités apportées au projet. Finalement, nous analyserons les résultats et discuterons de possibilités futures d'amélioration.

3.1 Les principaux aspects

Cette section a pour but de détailler les principaux aspects du projet : la classification automatique du plancton, la correction statistique des erreurs commises par le classificateur et la détection des éléments suspectés d'être mal classés.

3.1.1 Classification automatique

Nous l'avons vu tout au long de la section 2.1, la classification automatique utilise les attributs, les informations, les caractéristiques d'un élément afin de déterminer sa classe. Cette méthode machinale occupe une place centrale au sein du projet car elle permet de connaitre rapidement l'abondance du plancton dans la zone où il a été prélevé. L'inconvénient majeur de cette technique est qu'elle n'est pas fiable à 100%. En effet, le classificateur peut classer incorrectement certaines particules et biaiser le résultat final. Procéder à cette étape de façon manuelle n'est pas envisageable tant la procédure est longue, fastidieuse et fatiguante. Il est donc indispensable de trouver un compromis entre procédure automatique et manuelle, tout en garantissant la validité des résultats.

3.1.2 Correction statistique

Conscients du fait que la classification automatique n'est pas un système complètement fiable, Solow *et al* ont mis au point en 2001 une méthode de correction statistique de l'erreur permettant d'obtenir une estimation de l'abondance réelle du plancton. Ceux-ci considèrent le nombre de particules classées dans chaque groupe comme étant une estimation biaisée du nombre réellement présent dans l'échantillon [20]. Ils utilisent alors la matrice de confusion de l'ensemble d'apprentissage, pour qui les classes réelles sont connues, afin d'estimer ce biais et ainsi corriger le résultat (voir section 2.2). Cependant, cette technique pleinement automatique ne fonctionne pas comme prévu en pratique car le plancton évolue morphologiquement et, de ce fait, le modèle créé sur base de l'ensemble d'apprentissage devient rapidement obsolète.

En gardant la même idée à l'esprit, nous allons contourner le problème en utilisant la matrice de confusion de l'échantillon étudié, et plus précisément celle d'un sous-échantillon. À cet effet, nous allons procéder à la validation manuelle d'un faible pourcentage de l'échantillon (typiquement entre 10 et 20%) et pondérer sa matrice de confusion afin d'obtenir une estima-

tion de l'abondance réelle. Afin que cette estimation soit la plus fiable possible, les erreurs de classification contenues dans l'échantillon doivent être ciblées et corrigées en priorité.

3.1.3 Détection des classifications suspectes

Dans le but de cibler et de corriger en priorité les erreurs de classification, nous allons utiliser un second classificateur capable de détecter les particules pour qui la classe semble suspecte. Ces particules suspectes sont celles qui se trouvent généralement assez proche des frontières de classification. De ce fait, une faible modification du modèle pourrait très bien faire changer la classe à laquelle appartient la particule. Ce second classificateur sera entrainé à l'aide des données validées manuellement car nous pouvons désigner celles qui furent mal classées par le premier classificateur.

3.1.4 En résumé

La classification manuelle d'un échantillon de plancton est une tâche longue, fastidieuse et fatiguante. La classification automatique, quant à elle, permet d'obtenir rapidement une estimation de l'abondance du plancton mais introduit un pourcentage non négligeable d'erreurs qui peut fausser les résultats. La technique ayant été mise au point réalise un compromis entre ces deux approches. Celle-ci permet d'estimer l'abondance du plancton en ne validant qu'un sous-ensemble de l'échantillon et en ciblant en priorité les classifications suspectes.

3.1.5 Algorithme

L'algorithme est itératif et prend en entrée un tableau contenant pour chaque particule ses attributs, la classe prédite par le classificateur et des informations complémentaires.

1. Création d'un ensemble d'apprentissage Au départ, toutes les particules sont considérées comme étant suspectes. Un sous-échantillon [1] des données est extrait de façon aléatoire et est validé manuellement. Une fois validé, nous connaissons les véritables classes des particules et nous pouvons donc déterminer les erreurs de classification. Ensuite, nous créons un classificateur capable de prédire les erreurs de classification en utilisant cet échantillon validé comme ensemble d'apprentissage.

2. Détection des suspects Un classificateur est créé à partir de l'ensemble d'aprentissage. Les données n'ayant pas encore été validées sont fournies au classificateur qui les scinde en deux groupes distincts : *suspects* et *non-suspects*.

3. Création d'un ensemble test Un sous-échantillon est formé en prélevant des données dans les deux groupes. Étant donné que les particules suspectes sont ciblées en priorité, l'algorithme tente, si possible, d'en prélever une part plus importante. Cet échantillon est ensuite validé manuellement. La validation d'une petite portion de particules non-suspectes permet de corriger certaines particules erronées qui auraient échappé au classificateur.

1. La taille des sous-échantillons est déterminée par la taille de l'échantillon et d'autres paramètres entrés par l'utilisateur.

4. Estimation de l'abondance Cette estimation est réalisée à l'aide de 3 matrices de confusion : celle des données validées précédemment, celle du sous-échantillon des suspects validés et celle du sous-échantillon des non-suspects validés. La somme pondérée des 3 permet d'acquérir une estimation de l'abondance totale réelle. Il s'agit donc de l'étape de correction statistique de l'erreur.

5. Mise à jour de l'ensemble d'apprentissage Le sous-échantillon validé est ajouté à l'ensemble d'entrainement afin d'améliorer la précision du classificateur. Les étapes 2 à 5 se répètent jusqu'à ce que l'utilisateur soit satisfait du résultat.

L'algorithme suivant permet de mieux comprendre la succession des étapes citées ci-dessus :

Algorithme 6: Error correction

```
Data : L'échantillon à étudier
Result : Une estimation de l'abondance réelle du plancton
sample ← Extraire un sous-échantillon aléatoire;
validated ← Valider manuellement sample;
while non terminé do
    classif ← Créer un classificateur pour la détection des suspects à l'aide de validated;
    classif.res ← Marquer les particules suspectes et non-suspectes à l'aide de classif;
    suspect ← Prélever une portion de suspects de classif.res;
    ok ← Prélever une portion de non-suspects de classif.res;
    testset ← {suspect, ok};
    Valider testset manuellement;
    result ← Corriger l'erreur à l'aide validated et testset;
    validated ← {validated, testset}
return result
```

3.2 Analyse structurelle du code source

Dans cette partie nous allons analyser la qualité structurelle du code pour ensuite l'améliorer. Nous allons dans un premier temps décrire sa structure apparente. Ensuite, sur base d'une étude plus approndie nous créerons un premier diagramme d'activité afin de visualiser les étapes importantes du processus de correction d'erreurs. Finalement, nous verrons s'il est possible de mesurer sa qualité au travers de différentes métriques logicielles.

3.2.1 Présentation du code

Le code source fourni contient 3 fichiers *.R* : un fichier principal exécutant l'algorithme de correction d'erreurs et deux fichiers annexes contenant certaines fonctions utilisées dans le programme principal. Ce dernier se présente sous la forme d'un script d'environ 400 lignes, aucune fonction n'est définie. Il est donc difficile en le parcourant brièvement d'en dégager une certaine structure, d'identifier des relations entre différentes parties, etc. Pour cela, une lecture attentive du code en gardant à l'esprit l'algorithme vu précédemment est nécessaire afin d'identifier les parties du code réalisant une tâche commune.

3.2.2 Diagramme d'activités

Les diagrammes d'activités sont des diagrammes modélisant le comportement dynamique d'un système. Ils permettent de mieux comprendre la succession des étapes qu'un système doit accomplir afin de réaliser une certaine tâche. Sur base d'une analyse attentive du code source, voici les différentes activités ayant pu être identifiées.

Préparation des données Cette première activité prend en entrée le jeu de données contenant pour chaque particule ses caractéristiques, sa classe réelle et sa classe prédite. À partir de celui-ci, elle calcule des informations complémentaires qui seront utiles tout au long du processus de correction de l'erreur. Elle donne également la possibilité à l'utilisateur d'ajouter des caractéristiques biologiques et écologiques au jeu de données.

Création d'un ensemble d'apprentissage Cette activité consiste à créer un sous-échantillon aléatoire des données afin d'obtenir un ensemble d'apprentissage pour la prédiction des classifications suspectes.

Faux-positifs Dans cette activité, nous calculons pour chaque groupe la probabilité d'être un faux-positif à l'aide du théorème de Bayes. Nous calculons ensuite la différence qu'il y a entre les proportions prédites dans chaque groupe et cette quantité. Cette valeur sera utilisée comme attribut lors de la phase de prédiction des suspects.

Détection des particules suspectes Le classificateur créé à partir des particules validées est utilisé pour prédire les classifications qui semblent suspectes.

Création d'un ensemble test Suite au marquage des particules dont la classification semble suspecte, nous créons un ensemble de tests comprenant à la fois des particules suspectes et des particules correctes. Celui-ci est construit de telle sorte que la proportion de particules suspectes soit, si c'est possible, plus importante.

Validation Les particules de l'ensemble fourni doivent être validées avant de pouvoir procéder à la prochaine étape.

Correction statistique La correction de l'erreur est réalisée à l'aide de 3 matrices de confusion : celles des données précédemment validées, celle des données suspectes validées à l'étape courante et celle des données non-suspectes validées validées à l'étape courante. Ces 3 matrices sont aggrégées afin d'obtenir deux matrices : la matrice de confusion des suspects validés (toute étape confondue) et la matrice de confusion des non-suspects (toute étape confondue). L'estimation de l'abondance réelle est obtenue en pondérant ces 2 matrices et en les additionnant comme ceci :

$$
\begin{aligned}
suspect.conf &= \frac{\text{nombre de suspects détectés}}{\text{nombre de suspects validés}} * \text{matrice de confusion des suspects validés} \\
trusted.conf &= \frac{\text{nombre de non-suspects détectés}}{\text{nombre de non-suspects validés}} * \text{matrice de confusion des non-suspects validés} \\
abudance &= suspect.conf + trusted.conf
\end{aligned}
$$

Une illustration graphique du procédé est disponible à la FIGURE 3.1. Les données nouvellement validées sont ensuite ajoutées à l'ensemble d'apprentissage afin d'améliorer la précision du classificateur créé pour la prédiction de suspects.

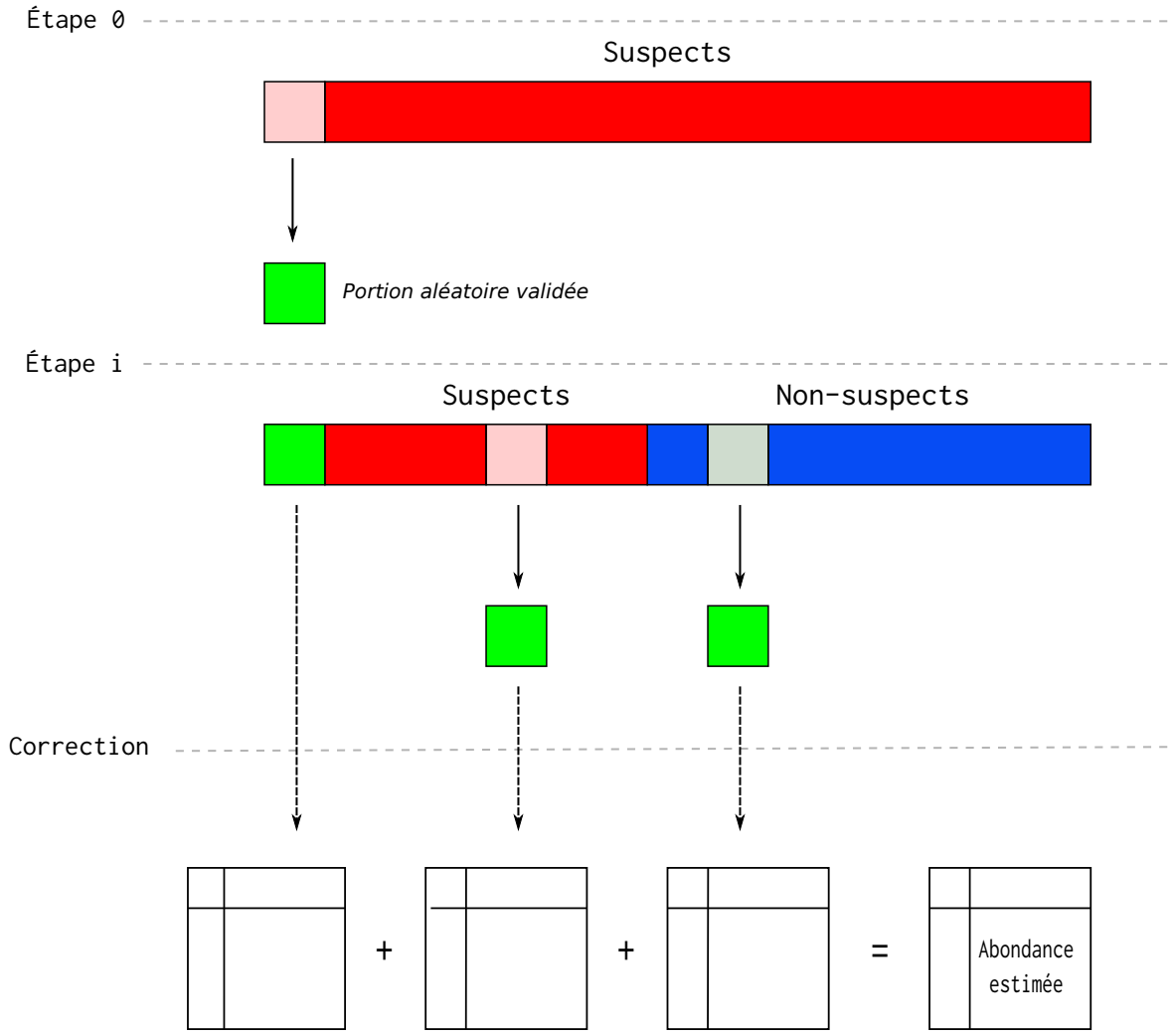


FIGURE 3.1 – *Une illustration de la détection des suspects et de la correction d'erreur. A l'étape 0 l'ensemble de l'échantillon est considéré comme suspect et une portion aléatoire est validée manuellement. Les étapes suivantes, le classificateur marque les particules suspectes et non-suspectes et une portion de chacun des deux ensemble est prélevé. Ces portions sont ensuite validées manuellement par l'utilisateur. Une fois validées, une somme pondérée des matrices de confusion des particules validées aux étapes précédentes, des suspects et des non-suspects est calculée afin de fournir une estimation de l'abondance réelle.*

À partir de cet ensemble d'activités, nous avons pu réaliser le diagramme d'activités illustré à la figure B.1 (voir annexe B). Il est important de noter que le processus décrit est complètement automatique. En effet, le code source est fourni avec un jeu de données pour lequel les classes réelles sont connues, il n'y a donc pas besoin de les inspecter manuellement pour vérifier si les classes prédites sont correctes. En pratique, ce n'est pas le cas car ne nous connaissons pas les véritables classes, elles seront prédites par un classificateur et ce sera à l'utilisateur de valider ou de corriger le résultat pour les classifications qui semblent les plus douteuses. Nous allons donc devoir modifier ce diagramme afin d'intégrer l'utilisateur dans le processus.

3.2.3 Métriques logicielles

Il existe une multitude de logiciels permettant de mesurer la qualité d'un code et de détecter les « bad smells » d'un projet. Parmi ceux-ci on compte *Sonar*, *InCode*, *Checkstyle* ou encore *PMD*. Cependant, ces logiciels ne travaillent qu'avec des langages tel que *Java*, C, C++ mais pas avec *R*. Des alternatives permettant de gérer le langage *R* n'ont malheureusement pu être trouvées.

3.3 Modification de la structure

La structure actuelle du programme ne permet pas à l'utilisateur d'intéragir avec le processus, c'est pourtant un besoin essentiel. En effet, le programme doit pouvoir être interrompu lorsque l'utilisateur doit intervenir et être repris lorsque celui-ci le demande. À cet effet, un diagramme d'activités illustrant la structure adaptée a été réalisé à la figure B.2. Les couloirs précisent quelle activité doit être effectuée par l'utilisateur et quelle activité doit être effectuée par le programme. Le processus est itératif : à chaque itération un nouveau sous-échantillon est créé, validé et utilisé pour corriger statistiquement l'erreur. Chaque itération est donc découpée en 3 phases :

1. une phase de création d'un ensemble test.
2. une phase de validation manuelle de l'ensemble test.
3. une phase de correction statistique de l'erreur.

Lorsque le processus atteint la phase 2, celui-ci doit sauver l'état dans lequel il se trouve, se stopper et permettre à l'utilisateur de valider manuellement les vignettes [2]. Une fois que l'utilisateur a terminé, il relance le processus qui vérifie alors que toutes les vignettes ont été validées. Si c'est le cas, la phase 3 s'exécute, l'état du programme est sauvegardé et le processus est stoppé. L'utilisateur peut alors soit se contenter du résultat, soit relancer une nouvelle itération. Dans l'autre cas, l'utilisateur est invité à poursuivre la validation manuelle. Lorsque toutes les données ont été validées, le processus est entièrement terminé. L'algorithme 7 décrit la nouvelle structure du programme.

2. Les vignettes contiennent l'image du plancton ainsi que ses diverses caractéristiques.

Algorithme 7: Process

```
if non initialisé then
    Initialiser;
    Créer le test set;
    étape manuelle ← true;
else if non terminé then
    if non étape manuelle then
        Calculer la probabilité d'être un faux-positif;
        Détecter les particules suspectes;
        Créer le test set;
        étape manuelle ← true;
    else
        Récupérer le test set;
        if le test set est entièrement validé then
            étape manuelle ← false ;
            Corriger l'erreur;
        else
            Inviter l'utilisateur à continuer la validation;
else
    Avertir l'utilisateur que la correction est entièrement terminée;
```

3.4 Choix de l'architecture

À la section précédente nous avons défini l'aspect fonctionnel du programme, c'est-à-dire la succession des tâches qu'il doit réaliser. Nous allons désormais nous intéresser de plus près à son architecture, autrement dit, à la manière de le concevoir. Nous allons être amenés à travailler avec de grandes quantités de données. Par exemple, le jeu de données fourni avec le code source contient environ 3000 instances du plancton caractérisées par près d'une centaine d'attributs. Il est donc nécessaire de choisir une architecture permettant de minimiser l'impact en temps et en mémoire du traitement sur ces grandes quantités de données. Afin de pouvoir justifier nos choix, une analyse du fonctionnement et des performances du langage R a été effectuée à l'annexe A. La lecture de cette section est donc vivement recommandée.

3.4.1 Architecture des données

Dans le code source nous ayant été fourni, les données opérationnelles, c'est-à-dire les données relatives à la correction de l'erreur, sont concaténées au jeu de données contenant le plancton et ses attributs. De ce fait, tout au long du processus un *data.frame* contenant des milliers de lignes et près d'une centaine de colonnes est manipulé. Il faut savoir que lorsque l'on programme en R, les arguments des fonctions sont passés par valeur et chaque modification d'un objet entraine l'affectation complète d'un nouvel objet. L'approche utilisée jusqu'ici est donc clairement inefficace, nous allons donc restructurer ces données afin de limiter leur impact sur les performances, tout en gardant une certaine cohérence vis-à-vis du traitement. Pour cela, nous allons scinder ces données en 3 *data.frame* :

1. **dataset** : il s'agit du jeu de données de départ contenant le plancton et ses attributs. Il est le plus important en terme de taille et possède l'avantage d'être rarement modifié.

2. **proba** : celui-ci contient l'ensemble des probabilités qui se rapportent aux données du *dataset*. Ces dernières sont calculées à chaque itération avant de procéder à la détection des classifications suspectes.
3. **corr** : il contient les données liées à la correction. On y retrouve les premières et secondes classes prédites, les classes validées, le marquage des particules suspectes, des particules validées et des particules erronées. C'est le plus utilisé des trois, il intervient à toutes les étapes du traitement.

3.4.2 Architecture fonctionnelle

Comme nous l'avons dit précédemment, notre programme va être amené à travailler avec une grande quantité de données et notre objectif est de minimiser leur incidence sur le temps d'exécution et l'allocation de la mémoire. Nous allons donc tenter de trouver une architecture adaptée à ces deux besoins. Pour cela, nous avons testé 5 types d'architectures différentes, certaines correspondant à des façons typiques de programmer avec R et d'autres plus techniques utilisant certaines spécificités du langage.

Architecture 1

La première architecture est la plus basique de toutes car elle correspond à la façon la plus typique de programmer en R. Dans celle-ci, les jeux de données *dataset*, *proba* et *corr* sont séparés et passés en argument des fonctions (activités) qui les requierent. Nous souhaitons mesurer avec cette architecture l'impact du passage par valeur. Une illustration de cette architecture est disponible à la FIGURE 3.2

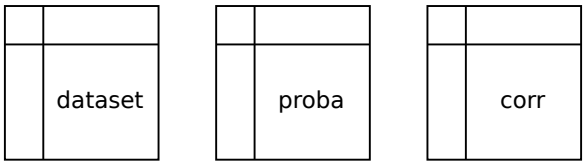


FIGURE 3.2 – *Illustration de la première architecture.*

Architecture 2

Cette seconde architecture utilise *proba* et *corr* comme attributs (voir section **??**) de *dataset*. *dataset* est ainsi donné en argument de toutes les fonctions et ses attributs sont extraits lorsqu'ils sont requis. Avec cette architecture, nous souhaitons observer si les attributs pénalisent ou non les performances. Une illustration de cette architecture est disponible à la FIGURE 3.3

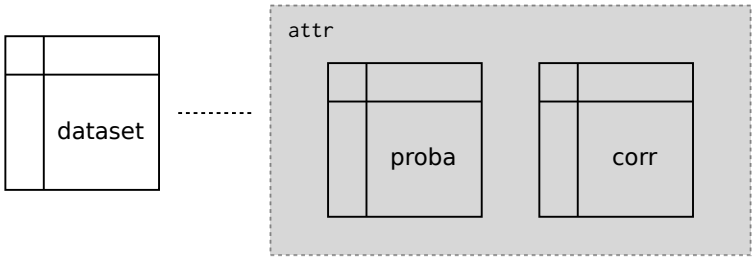


FIGURE 3.3 – *Illustration de la deuxième architecture.*

Architecture 3

La troisième architecture est similaire à la première. Cependant, cette fois-ci les données sont intégrées à un objet de type environnement (voir section A.2.2). Les fonctions manipulent alors les données à travers cet environnement. La particularité des environnements est qu'ils ne sont pas passés par valeur, nous y voyons donc un avantage substantiel par rapport à la première architecture. Une illustration de cette architecture est disponible à la FIGURE 3.4

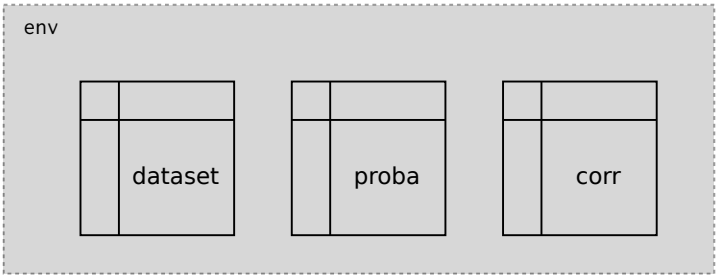


FIGURE 3.4 – *Illustration de la troisième architecture.*

Architecture 4

Cette quatrième architecture est similaire à la seconde excepté le fait que le *dataset* est intégré à un objet de type environnement. À nouveau, nous voyons un avantage substantiel à l'utilisation des environnements. De plus, cette architecture nous donne une seconde possibilité d'analyser l'impact des attributs sur les performances du système. Une illustration de cette architecture est disponible à la FIGURE 3.5

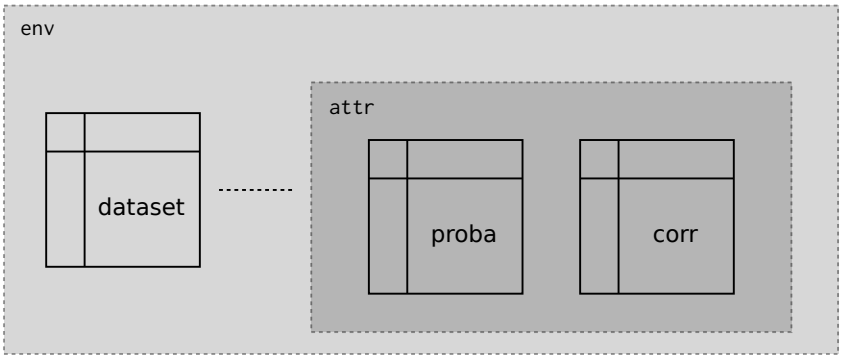


FIGURE 3.5 – *Illustration de la quatrième architecture.*

Architecture 5

La cinquième architecture est la plus technique car elle utilise le concept de fermeture et d'assignation non-locale (voir section A.5). Ce mécanisme s'inspire de la programmation orientée-objet mais n'utilise pas de définition de classe formelle. Ici, les fonctions partagent un environnement commun et mettent à jour les variables à l'aide d'assignations non-locales. Il sera intéressant de la comparer aux architectures 3 et 4 car toutes trois manipulent les données à travers un environnement mais sont conceptuellement différentes. Une illustration de cette architecture est disponible à la FIGURE 3.6

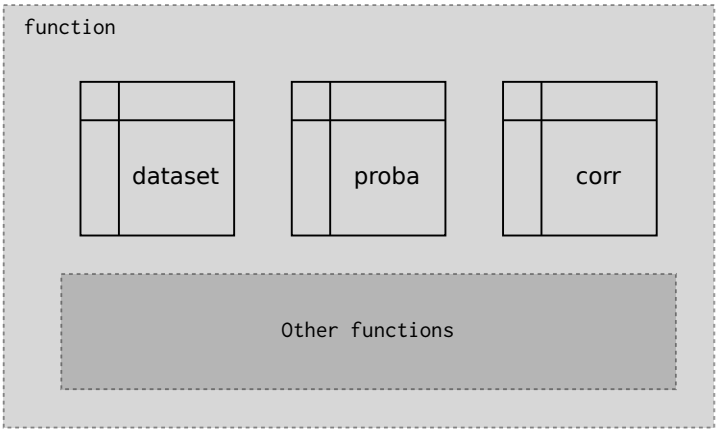


Figure 3.6 – *Illustration de la cinquième architecture.*

3.4.3 Comparaison des architectures

Les performances de ces 5 architectures ont été testées. Nous avons mesuré pour chacune d'elles le temps d'exécution, la mémoire allouée et le nombre d'objets copiés par l'évaluateur R. Celles-ci ont été exécutées 500 fois afin d'obtenir pour chaque mesure sa valeur moyenne et minimale. Le jeu de données utilisé contient 3000 instances et près d'une centaine d'attributs. La taille des échantillons prélevés est 100.

Temps d'exécution

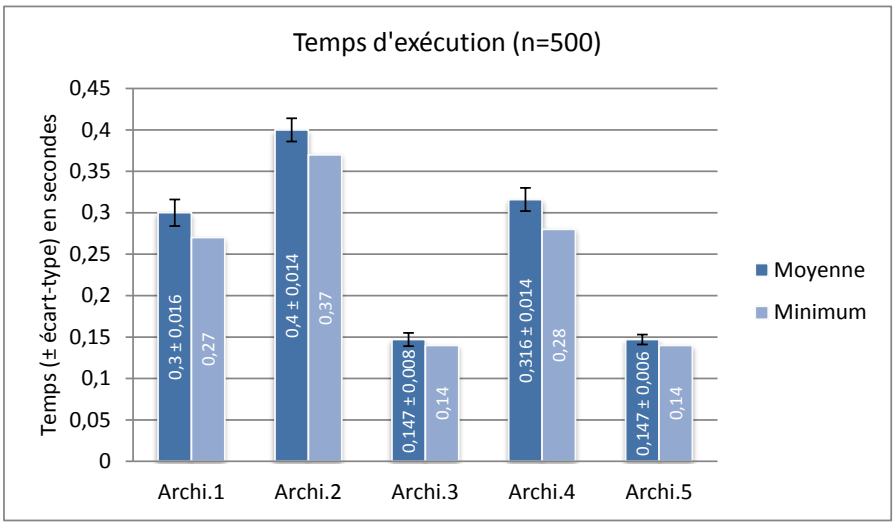


Figure 3.7 – *Graphique du temps moyen et minimum mis par chaque architecture.*

Au niveau du temps d'exécution, les architectures 3 et 5 sont de loin les plus performantes. Elles sont au minimum 2 fois plus rapides que les architectures 1, 2 et 4. Elles sont également très stables, leurs écarts-types sont faibles et sont de moitié inférieurs à ceux des trois autres. Ces meilleures performances sont dues à la manipulation des données directement dans leur environnement. En procédant de cette manière, nous économisons les copies d'arguments et limitons le travail du garbage collector. L'architecture 4 bénificie des mêmes avantages mais à

chaque fois qu'un attribut est modifié c'est tout le *dataset* qui est remplacé. Cet effet se ressent également lorsqu'on compare les architectures 1 et 2.

Allocation de mémoire

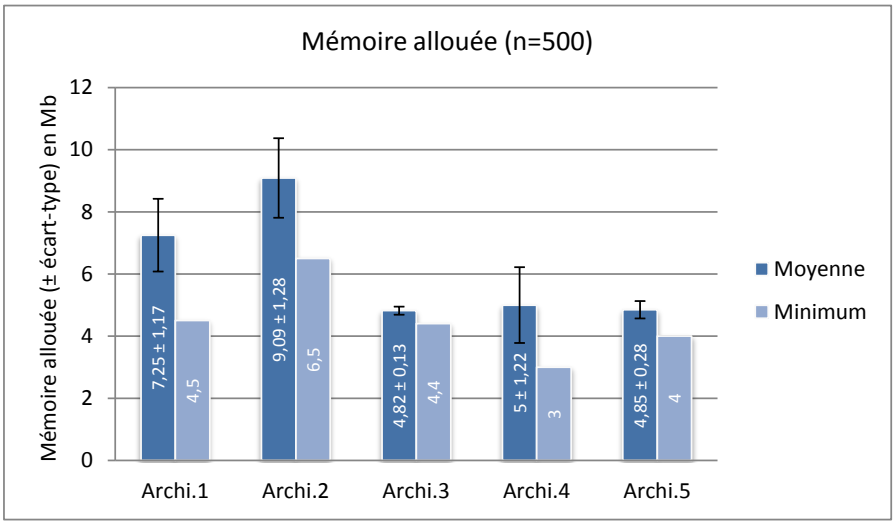


Figure 3.8 – *Graphique de la mémoire allouée en moyenne et au minimum par les différents architectures.*

Le constat est le même en ce qui concerne l'allocation de mémoire : manipuler les objets directement à travers leur environnement permet d'obtenir un gain de performance. À nouveau, les architectures 3 et 5 sont les plus performantes et les plus stables. Les capacités de l'architecture 4 sont en moyenne comparables à ces deux dernières mais elles sont moins stables.

Duplication d'objets

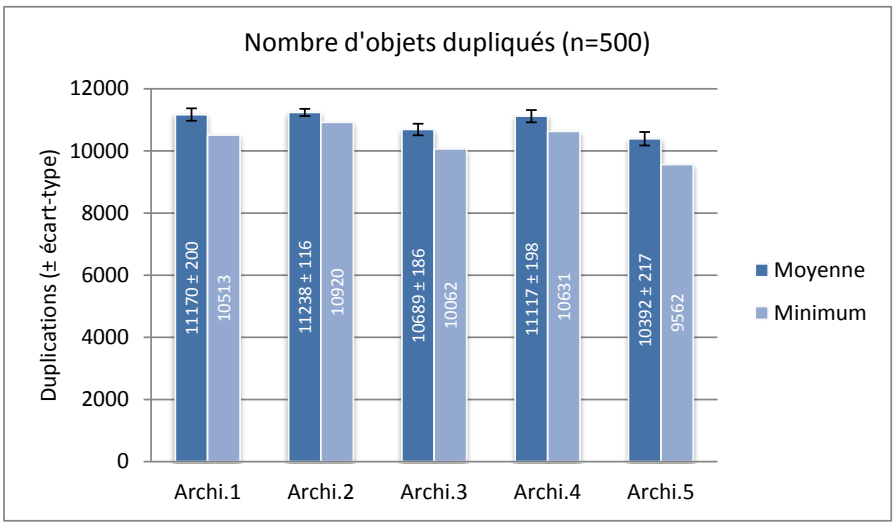


Figure 3.9 – *Graphique du nombre d'objets dupliqués en moyenne et au minimum.*

Le nombre d'objets dupliqués est le nombre de copies d'objets faites par *R* lors de l'exécution du programme. Nous constatons de manière générale que la manipulation des données directement dans leur environnement permet d'économiser quelques copies en mémoire. À nouveau, les architectures 3 et 5 sont les plus performantes à ce niveau. Elles effectuent en moyenne quelques centaines de copies de moins comparées aux autres architectures.

Choix de l'architecture

Le tableau 3.1 reprend les résultats des différentes architectures. À la vue de ceux-ci, 2 architectures se démarquent : l'architecture 3 qui manipule les objets *dataset*, *proba* et *corr* dans leur environnement et l'architecture 5 qui utilise les fermetures et les assignations non-locales. Toutes deux offrent des performances comparables, le choix de l'une ou de l'autre ne doit donc pas se baser sur ce critère. Puisqu'il faut faire un choix, nous allons décider de développer l'architecture 5 car la technique qu'elle emploie utilise un format orienté-objet : elle permet une bonne encapsulation des données et ne fournit que les interfaces nécessaires à l'utilisateur.

	Temps (s)			Mémoire (Mb)			Duplications		
Architecture	Min	Moyenne	Ecart-type	Min	Moyenne	Ecart-type	Min	Moyenne	Ecart-type
1	0.27	0.3	0.016	4.5	7.25	1.17	10513	11170	200
2	0.37	0.4	0.014	6.5	9.1	1.28	10920	11238	116
3	0.14	0.147	0.008	4.4	4.82	0.13	10062	10689	186
4	0.28	0.316	0.014	3	5	1.22	10631	11117	198
5	0.14	0.147	0.006	4	4.85	0.28	9562	10392	217

Table 3.1 – *Performances des différentes architectures.*

3.5 Nouvelles fonctionnalités

Dans cette section nous allons parler des fonctionnalités qui ont été ajoutées au programme.

3.5.1 Test unitaires

Un test unitaire est un bout de code prenant en charge la validité d'un autre morceau de code. Ces tests reçoivent des entrées et vérifient que le résultat fourni en sortie est bien celui auquel nous nous attendons. Ils permettent donc d'automatiser les tests de non-régression, autrement dit de s'assurer du bon fonctionnement du logiciel après une modification. Pour que ceux-ci soit efficaces, il faut qu'ils couvrent la majorité des fonctionnalités d'un logiciel. Le *test driven development* ou développement dirigé par les tests est une technique qui préconise d'écrire les tests unitaires avant d'écrire le code source. Le cycle de vie du test driven development comporte 5 étapes :

1. écrire un test.
2. vérifier qu'il échoue.
3. écrire le code pour ce test.
4. vérifier que le test passe.
5. améliorer le code.

Écrire les tests avant toute chose force à évaluer les différents scénarios auxquels le code sera exposé, ce qui permet d'obtenir un code valide en toutes circonstances et d'éviter des erreurs de conception. Durant le projet, nous avons utilisé cette approche dirigée par les tests. Le

framework de tests unitaires utilisé est *svUnit* [24]. Malheureusement, nous n'avons pu trouver un logiciel capable de vérifier la couverture des tests.

3.5.2 Migration de version

Le code source que nous avons reçu était conçu pour être exécuté avec la version 2 de la librairie *zooimage*. Cette librairie étant désormais dans sa version 3, il nous a fallu adapter notre code afin que le programme puisse fonctionner avec cette nouvelle version. Ce genre de migration n'est pas aisée car certaines fonctions apparaissent, d'autres disparaissent, certaines changent de nom, de signature ou de spécification. La grande partie du travail a donc été d'identifier ces différences et d'adapter notre code aux circonstances. Une fois les modifications effectuées, la validité du code a pu être vérifiée grâce aux tests unitaires préalablement écrits.

3.5.3 Intégration de différents modes

Le programme doit pouvoir être exécuté dans 3 modes différents :

* validation : il s'agit du mode de fonctionnement adapté aux situations réelles de correction. Ce mode a été implémenté mais est volontairement « incomplet » car la validation manuelle d'un échantillon de plancton requiet des compétences que nous ne possédons pas. Cependant, le code a été préparé afin de pouvoir intégrer facilement les fonctionnalités de zooimage permettant de réaliser la validation.
* stat : il s'agit d'un mode de fonctionnement automatique nécessitant de fournir un jeu de données où les classes réelles sont connues. Il permet de tester les performances de l'algorithme sur un jeu de données particulier.
* démo : il s'agit d'un mode de fonctionnement que l'on peut qualifier de semi-automatique : il prend en paramètre un jeu de données où les classes réelles sont connues et marque un arrêt à chaque fois que l'intervention de l'utilisateur est nécessaire dans le mode validation. C'est en quelque sorte un mode stats marquant une pause à chaque itération.

Les deux nouveaux modes stat et démo ont été implémentés. Leurs diagrammes d'activité se trouvent respectivement à la FIGURE B.3 et B.4 (voir annexe B).

3.6 Optimisation

Maintenant que nous avons défini la structure du programme choisi une architecture adaptée à nos besoins et ajouté les nouvelles fonctionnalités, nous allons essayer d'identifier les parties du code pouvant être optimisées.

3.6.1 Profilage

Afin de pouvoir identifier les parties du code les plus gourmandes en temps d'exécution, nous avons procédé au profilage d'une exécution complète du programme. Les résultats que nous avons obtenus se trouvent à la figure 3.10. Ceux-ci parlent d'eux-mêmes : la détection des classifications suspectes, et plus particulièrement l'algorithme *randomForest*, représente la quasi totalité du temps d'exécution.

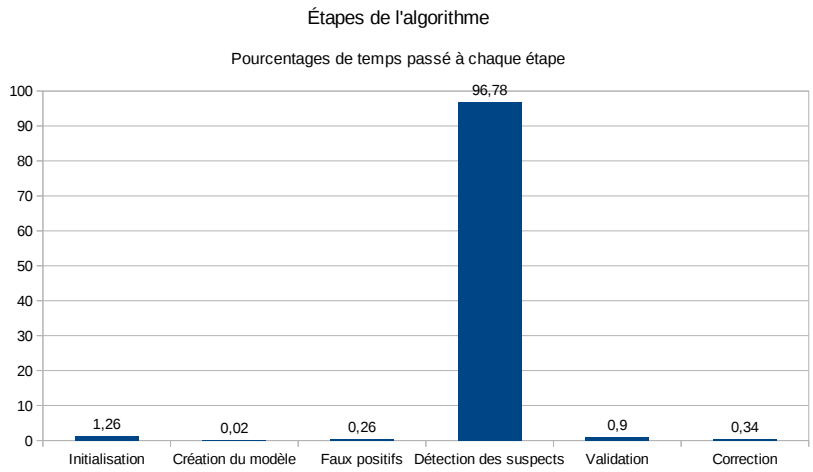


Figure 3.10 – *Pourcentage de temps passé dans chaque étape du processus de correction.*

Dans un premier temps, nous pensions réduire ce temps d'exécution en écrivant cet algorithme dans un langage de programmation plus performant comme le *C* ou le *Fortran* (voir section **??**) mais il se fait que cet algorithme est déjà écrit et optimisé en *Fortran*. Nous allons donc voir si nous pouvons trouver un autre algorithme de classification plus rapide et permettant dans la foulée d'obtenir de meilleurs résultats.

3.6.2 Algorithmes de classification

Comme nous l'avons dit précédemment, nous cherchons un algorithme de classification rapide et capable de prédire avec un maximum de certitudes les particules suspectes. Nous allons pour cela tester plusieurs algorithmes : les k plus proches voisins (voir section 2.1.4), SMO (voir section 2.1.8), naive Bayes (voir section 2.1.3) et la régression logistique (voir section 2.1.2). Le premier est disponible dans le package *e1071* et est écrit en *C* et les autres proviennent du package *RWeka* et sont écrits en *Java*. Nous allons comparer leur temps d'exécution et ensuite leur qualité de prédiction. Les tests s'effectueront sur 3 échantillons de cultures différentes : *BE.pred1* qui contient 3204 instances, *BE.pred2* qui en possède 3093 et *BE.pred3* avec 2625 exemples. Pour chaque algorithme, le programme sera exécuté 20 fois et plusieurs mesures seront relevées : l'erreur résiduelle lorsque 10% et 20% de l'échantillon ont été validés, l'aire sous les courbes d'erreur résiduelle et le temps mis pour corriger entièrement les échantillons. Les paramètres par défaut de ces algorithmes ont été utilisés sauf pour *knn* où $k = 7$.

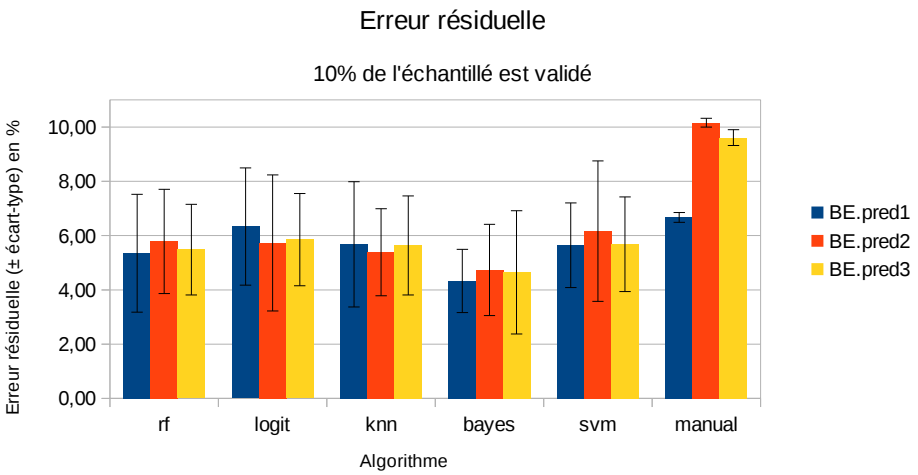


Figure 3.11 – *Erreur résiduelle moyenne pour les 3 échantillons lorsque* 10% *a été validé.*

Nous constatons premièrement que la correction statistique de l'erreur, quel que soit l'algorithme de classification utilisé, permet de mieux estimer l'abondance qu'avec une simple validation manuelle des échantillons. Ensuite, nous remarquons que les résultats varient peu d'un algorithme à l'autre. En moyenne, ils permettent d'estimer l'abondance réelle avec une précision allant de 94 à 96%. Naive Bayes est légèrement meilleur mais la différence avec les autres algorithmes est minime. Nous remarquons également que ces résultats possèdent un écart-type allant de 1 à 2.5%.

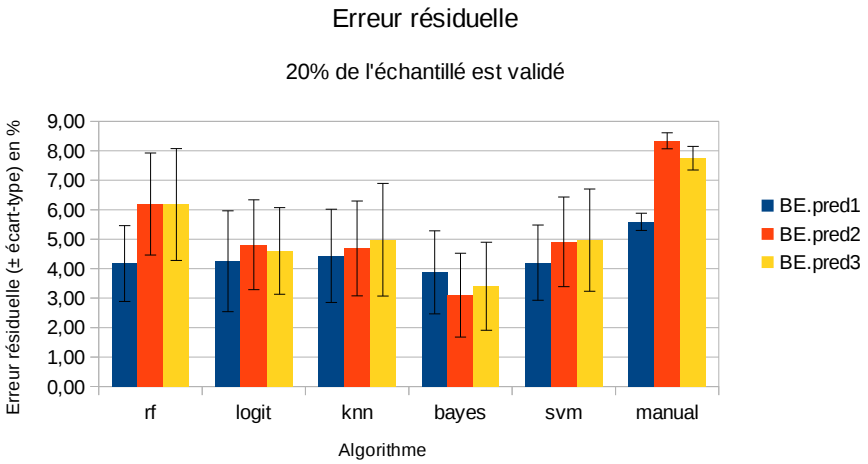


Figure 3.12 – *Erreur résiduelle moyenne pour les 3 échantillons lorsque* 20% *a été validé.*

Lorsque 20% des données ont été validées, nos algorithmes nous permettent en moyenne d'obtenir des taux de précision variant de 95 à 97%. Nous avons donc en moyenne un gain de précision de 1% par rapport aux 10% validés précédemment. Les algorithmes fournissent à nouveau une meilleur estimation de l'abondance qu'avec une simple validation manuelle. Nous pouvons également voir que nos différentes alternatives sont en moyenne meilleures

que random forest. Naive Bayes est une nouvelle fois légèrement plus performant, entre 1.5 et 3% de différence selon les algorithmes. L'écart-type se ressert quelque peu : il varient entre 1 et 2% cette fois-ci.

3.6.3 Temps d'exécution

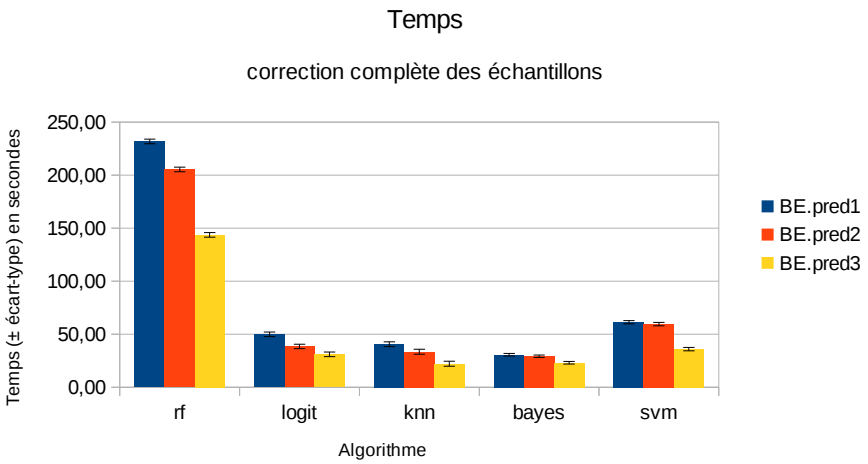


Figure 3.13 – *Temps d'exécution moyen pour corriger un échantillon complet.*

La différence se marque au niveau du temps d'éxécution. Nos alternatives peuvent être jusqu'à 5 fois plus rapides que random forest. Les deux algorithmes les plus rapides sont naive Bayes et les k plus proches voisins. Ils sont cependant suivis d'assez près par la régression logistique et les machines à vecteurs de support.

3.6.4 Performances globales

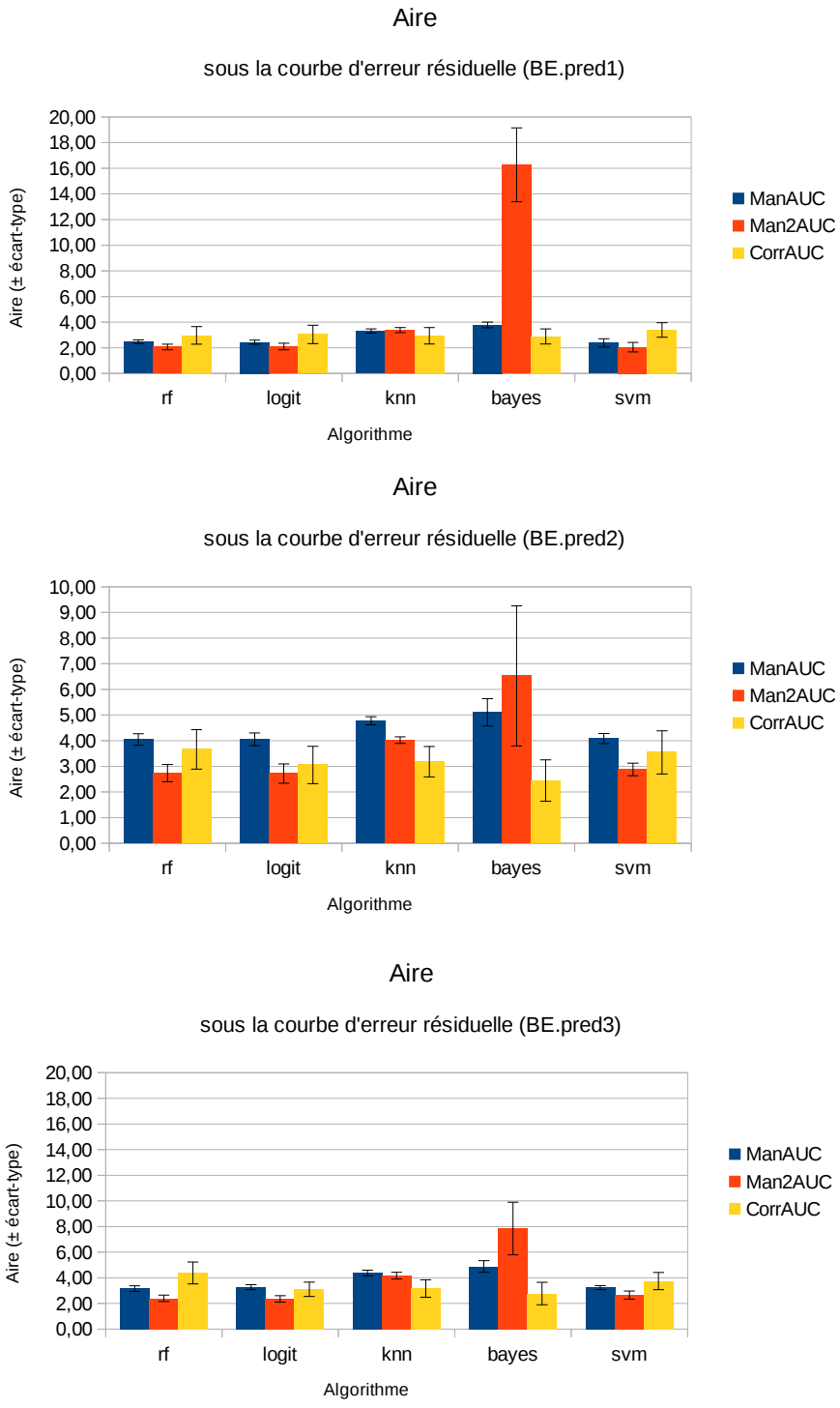


Figure 3.14 – *Aire sous les courbes d'erreur résiduelle.*

L'aire sous les courbes d'erreurs résiduelles nous permet d'avoir une idée des performances globales des différents algorithmes. *ManAUC* est l'aire pour la validation manuelle des particules, *ManAUC*2 est l'aire pour la validation manuelle avec affectation de la seconde classe

la plus probable pour les particules marquées comme suspectes et $CorrAUC$ est l'aire pour la correction statistique de l'erreur. Nous souhaitons donc avoir $ManAUC > ManAUC2 > CorrAUC$. Le seul algorithme a respecté cela est les k plus proches voisins. De manière générale, les algorithmes semblent pouvoir identifier correctement bon nombre de particules suspectes car leur $ManAUC$ est plus que grand que leur $ManAUC2$. Le seule algorithme éprouvant certaines difficultés est naive Bayes. Il marque souvent à tort des particules non-suspectes comme étant suspectes ce qui fait que son $ManAUC2$ est bien plus grand que son $ManAUC$.

Discussion

Nous avons vu que les différents algorithmes permettent d'estimer l'abondance réelle avec une précision relativement élevée allant de 94 à 97%. Il n'y a pas de différences significatives en terme de correction au niveau des classificateurs. En revanche, nos alternatives sont bien plus rapides que random forest utilisé au départ. Nous proposons donc de laisser ce choix à l'utilisateur en lui recommandant tout de même d'utiliser de préférence les k plus proches voisins, la régression logistique et les machines à vecteurs de support.

3.6.5 Utilisation

Le programme est très simple d'utilisation. Dans un premier temps, nous devons charger le code source :

```
> load("correction.final.R")
```

Ensuite, un « objet » *ErrorCorrection* doit être créé. Cet objet possède plusieurs paramètres (voir documentation) permettant de configurer le processus. Par exemple, pour effectuer une correction en mode stat en utilisant l'algorithme des k plus proches voisins nous écrirons :

```
> ec <- ErrorCorrection(dataset, classifier, mode = "stat", algorithm = "knn")
```

À chaque fois qu'un appel à la fonction *iterate* est appliqué à cet objet, le processus effectue une itération :

```
> ec$iterate()
```

L'estimation de l'abondance et l'estimation de l'erreur à l'étape courante peuvent être obtenues à l'aide des méthodes *getAbundance* et *getErrorEstimation* :

```
> ec$getAbundance()
# Result ...
> ec$getErrorEstimation()
# Result ...
```

En mode stat et démo, les résultats peuvent être comparés graphiquement à la validation manuelle de l'échantillon sans correction et à la validation manuelle de l'échantillon sans correction mais en attribuant la seconde classe la plus probable aux particules suspectes :

```
> ec$plotResult()
```

L'utilisateur peut sauver son travail et le reprendre plus tard de la manière suivante :

```
> save(file = "correction.RData", ec)
# Later ...
> load("correction.RData")
```

3.7 Conclusion

Le service d'écologie numérique des milieux aquatiques de l'UMons a developpé récemment une procédure effectuant un compromis entre les méthodes manuelles et automatiques d'étude du plancton. Cette procédure consiste, après avoir classé automatiquement les images, à valider manuellement une faible portion des données (entre 10 et 20%) et d'estimer l'abondance réelle du plancton à l'aide de celles-ci. La particularité de ce sous-échantillon est que nous essayons d'y incorporer en priorité les données que nous qualifions de suspectes, c'est-à-dire les données que nous suspectons d'avoir été mal classées par le classificateur. Ces données sont ciblées à l'aide d'un second classificateur prévu à cet effet. Une fois ces données validées, les matrices de confusion des particules suspectes et non-suspectes sont pondérées et additionnées pour former une estimation de l'abondance réelle. Cet algorithme est itératif : à chaque étape un petit échantillon est extrait (2 – 3% des données), est validé manuellement et est ajouté aux données précédemment validées pour estimer l'abondance. À chaque itération, le classificateur servant à la détection des suspects est entrainé avec l'ensemble des données validées.

Notre travail consistait à optimiser plusieurs aspects de ce programme implémenté en *R*. Dans un premier temps, nous avons examiné la structure du code nous ayant été fourni. Ce code ne possédait pas de structure clairement identifiable, nous l'avons donc étudié plus précisément afin de pouvoir identifier les différentes parties du code réalisant une tâche commune. Sur base de étude, nous avons créé un premier diagramme d'activités. Nous nous sommes alors rendus compte que la procédure était complètement automatique et ne travaillait qu'avec des données pour lesquelles les classes réelles était connues. De ce fait, nous avons adapté notre structure afin que l'utilisateur puisse intéragir avec le système et travailler dans des conditions réelles où les véritables classes ne sont pas connues.

Une fois la structure du programme clairement définie, nous nous sommes interessés de plus près à la façon dont nous allions concevoir le code. Nous avons donc étudier 5 types d'architectures différentes. Pour chacune d'elles nous avons mesuré leur impact sur le temps d'exécution, sur la consommation de mémoires et sur le nombre d'objets dupliqués. Notre choix s'est finallement porté sur une architecture utilisant les fermetures des fonctions et les assignations non-locales. Nous avons ensuite développé la structure selon cette architecture en suivant un mode de développement dirigé par les tests. Ce code fonctionnant intialement avec la version 2 de zooimage, nous avons du effectuer une migration de version afin qu'il puisse fonctionner avec la nouvelle version de zooimage. Par la suite, nous avons intégré deux nouveaux modes de fonctionnement au programme : un mode *stats* permettant de tester la procédure de façon automatique et un mode *demo* permettant de présenter son déroulement.

Subséquemment, nous avons réalisé le profilage du code afin d'identifier des parties de code pouvant être optimisées. Nous avons alors remarqué que la quasi totalité du temps d'éxécution était passée à prédire les suspects à l'aide de l'algorithme random forest. Nous nous sommes alors mis en quête d'alternatives à cet algorithme et avons pour cela testé l'algorithme des k plus proches voisins, l'algorithme SMO, l'algorithme naive Bayes et la régression logistique. Nous sommes arrivés à la conclusion que ces différentes alternatives permettent d'obtenir des résultats équivalents mais qui permettent de réduire en moyenne le temps d'exécution de plus d'un facteur 5.

Nos objectifs ont donc été atteints globalement : la structure du code a été améliorée et adaptée à des conditions réelles de traitement, son bon fonctionnement est assuré à l'aide des tests unitaires, de nouvelles fonctionnalités ont pu aisément être ajoutées, le choix de l'architecture a été fait de façon à minimiser l'impact des données en termes de temps d'exécution et de consommation de mémoires. Le seul objectif que nous n'avons pas pu atteindre est de réduire le temps d'exécution d'un facteur 10. Cependant, nous sommes tout de même arrivés à obtenir un facteur 5 ce qui n'est pas négligeable.

Annexe A| Le langage R

A.1 Introduction

R est un langage de programmation et un environnement statistique conçu en 1993 par Ross Ihaka et Robert Gentleman de l'université d'Auckland, Australie [25]. Sa conception fut fortement influencée par 2 langages : le langage *S* de John Chambers [26] et le langage *Scheme* de Steel et Sussman. Syntaxiquement très proche de *S* dont il est considéré comme le successeur, son implémentation et sa sémantique sont,elles, dérivées du *Scheme* [27]. Lancé en 1995 sous license *GNU*, *R* est rapidement devenu une référence en matière d'exploration des données. L'influence des travaux de Chambers lui ont valu de recevoir en 1998 le *ACM Software System Award* pour avoir à jamais changé la manière dont les gens analysent, visualisent et manipulent les données. Celui-ci fait partie de la *R Development Core Team* qui est atuellement responsable du développement de *R*. Ses membres ont fondé, en 1999, la *R Foundation* afin de fournir un support au projet et aux autres innovations en matière de statistiques, procurer un point de référence pour les individus, institutions ou entreprises voulant supporter ou intéragir avec la communauté de développement, mais aussi afin de détenir et administrer les copyright de *R* et de sa documentation.

Ce langage doit une partie de son succès à l'activité de sa communauté. En effet, plus de 4700 packages sont activement maintenus, sans compter les projets indépendants comme *BioConductor* [28] qui compte environ 700 packages relatifs à la bioinformatique. Réputé inefficace en tant que langage interprété [29], il dispose néanmoins d'un système d'interface permettant de coder certaines fonctions critiques en *C* tout en profitant de la syntaxe haut-niveau de *R* [30]. Ses forces sont sa portabilité, son caractère open source, la richesse du langage de base, la concision du code et la qualité de sa documentation.

Au coeur même du projet se trouve un langage de programmation relativement atypique. En effet, *R* est formé d'une combinaison assez inhabituelle de caractéristiques. En bref, il est dynamique dans le même esprit que *Scheme* mais où le type de base est le vecteur. Il est aussi fonctionnel : les fonctions sont first-class, les arguments sont passés par valeur et évalués tardivement. La récursion n'est cependant pas optimisée, la vectorisation des opérations étant encouragée. Ses fonctions ont une portée lexicale et leurs variables peuvent être mises à jour, ce qui permet un style de programmation impératif. Finalement, le langage dispose de deux systèmes d'objets : S3 basé sur les fonctions génériques et S4 fondé sur un système de classes et méthodes (voir section **??**). Les principales sources utilisées pour rédiger ce chapitre sont [27, 29, 30]

A.2 Le concept d'objet

La plupart des langages de programmation fournissent des moyens d'accès aux données stockées en mémoire, ce n'est pas le cas avec *R*. À la place, celui-ci procure de nombreuses structures de

données spécialisées que l'on qualifie d'objet. Tout élément du langage est un objet : les symboles, les fonctions, leurs arguments, etc. Cette section a pour but de fournir une description préliminaire des principales structures de données fournies dans *R*.

A.2.1 Classe et type

Tout objet en *R* possède une classe et un type. La classe d'un objet définit la façon dont les données sont structurées. De base, de nombreuses classes sont disponibles. Néanmoins, les utilisateurs sont libres d'en définir de nouvelles. C'est d'ailleurs une partie important du langage qui s'inscrit dans le concept orienté-objet. Celle-ci est accessible via la méthode *class*. Ces structures de données sont, en interne, codées en *C*. On définit donc le type d'un objet comme étant le type de données encodées par une structure. Il est accessible via la méthode *typeof*. La tableau de la FIGURE A.1 présente les principaux types d'objets présents dans le langage.

TYPE	DESCRIPTION
NULL	La valeur NULL.
symbol	Un nom de variable.
pairlist	Une paire (utilisé principalement en interne).
closure	Une fonction.
environment	Un environnement.
promise	Un objet utilisé pour implémenter l'évaluation tardive.
language	Une construction du langage.
special	Une fonction interne qui n'a pas évalué ses arguments.
builtin	Une fonction interne qui a évalué ses arguments.
logical	Un vecteur contenant des valeurs logiques.
integer	Un vecteur contenant des nombres entiers.
double	Un vecteur contenant des nombres réelles.
complex	Un vecteur contenant des nombres complexes.
character	Un vecteur contenant des caractères.
...	Les arguments de taille variable.
any	Un type spécial correspond à n'importe quel type.
expression	Une expression.
list	Une liste.
bytecode	Du byte-code (seulement en interne).
externalptr	Un pointeur externe.
weakref	Une référence faible.
raw	Un vecteur contenant des octets.
S4	Un objet S4 (différent de S3).

FIGURE A.1 – Liste des différents types existant en *R*

A.2.2 Les principales classes d'objets

Le vecteur

Le vecteur est la structure de base. Il s'agit d'une collection ordonnée d'objets d'un même type (séquence homogène). Ces types peuvent être : *logical*, *integer*, *double*, *complex*, *character* et *raw*. Une chose importante à savoir est qu'une donnée scalaire d'un de ces types est en fait considérée comme un vecteur de longueur 1. Ces données peuvent être combinées pour former un plus grand vecteur à l'aide de la fonction *c* :

```
> v <- c(1,2,3)
> v
[1] 1 2 3
> typeof(v)
[1] "double"
> u <- c("a", "b", "c")
> u
[1] "a" "b" "c"
> typeof(u)
[1] "character"
```

Les cellules d'un vecteur peuvent être indicées à l'aide de l'opérateur *[]*. Il existe plusieurs possibilités d'indiçage : à l'aide d'un vecteur d'entiers positifs, d'un vecteur d'entiers négatifs, d'un vecteur d'éléments nominatifs ou encore par un vecteur logique :

```
> constants <- c(3.1415, 2.7183, 1.6180)
> names(constants) <- c("pi", "euler", "golden")
> constants
    pi  euler golden
3.1415 2.7183 1.6180
> constants[c(1,3)]
    pi golden
3.1415 1.6180
> constants[-1]
 euler golden
2.7183 1.6180
> constants["euler"]
 euler
2.7183
> constants > 3
  pi  euler golden
TRUE  FALSE  FALSE
> constants[constants > 3]
    pi
3.1415
```

Lorsque deux vecteurs n'ayant pas la même taille sont utilisés dans une opération, le plus petit est recyclé afin d'égaler la longueur du plus long :

```
> constants * 2
    pi  euler golden
6.2830 5.4366 3.236
> constants * c(1,0)
    pi  euler golden
3.1415 0.0000 1.6180
```

Il existe également des fonctions permettant de créer des séquences (*:*, *seq*) ou encore de répéter plusieurs fois une valeur (*rep*) :

```
> 1:5
[1] 1 2 3 4 5
> seq(from = 0, to = 10, by = 2)
[1] 0 2 4 6 8 10
> seq(from = 0, to = 6, len = 3)
[1] 0 3 6
> rep(0, 3)
[1] 0 0 0
> rep(c(0,2), 2)
[1] 0 2 0 2
> rep(1:4, each = 2)
[1] 1 1 2 2 3 3 4 4
> rep(1:4, c(1,2,3,4))
[1] 1 2 2 3 3 3 4 4 4 4
```

Les listes

Les listes sont une forme générale de vecteur dans lequel les éléments n'ont pas besoin d'être du même type (séquence hétérogène) :

```
> lst <- list(name = "Pierre", age = 20)
> lst
$name
[1] "Pierre"

$age
[1] 20
```

Les éléments d'une liste sont numérotés et sont accessibles à l'aide de l'opérateur *[[]]*, à ne pas confondre avec *[]*. Le premier permet d'extraire un élément tandis que le second permet d'extraire une sous-liste. Ses composants peuvent également être nommés et accessibles par leur nom :

```
> lst[[1]]
[1] "Pierre"
> lst[1]
$name
[1] "Pierre"

> lst[["age"]]
[1] 20
> lst$age
[1] 20
```

Comme tout objet indiçable, les listes sont extensibles et concaténables :

```
> lst[3] <- list(status = "single")
> lst
$name
[1] "Pierre"

$age
[1] 20

$status
[1] "single"

> c(lst, list(country = "Belgium"))
$name
[1] "Pierre"

$age
[1] 20

$status
[1] "single"

$country
[1] "Belgium"
```

Les facteurs

Un facteur est un objet vecteur utilisé pour spécifier une classification discrète des composants d'un vecteur de même longueur. Il est accessible par l'attribut *levels* :

```
> colors <- c("Blue", "Red", "Blue", "Green", "Black", "Red", "Blue")
> fcolors <- factor(colors)
> fcolors
[1] Blue  Red   Blue  Green Black Red   Blue
Levels: Black Blue Green Red
> attr(fcolors, "levels")
[1] "Black" "Blue"  "Green" "Red"
> levels(fcolors)
[1] "Black" "Blue"  "Green" "Red"
```

Les matrices

Les matrices sont une généralisation à deux dimensions des vecteurs. Elles sont indiçables à l'aide de deux indices :

```
> m <- matrix(1:6, nrow = 3, ncol = 2, byrow = TRUE)
> m
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
> m[3,2]
[1] 6
```

Un indiçage partiel permet d'obtenir des lignes ou des colonnes complètes :

```
> m[1,]
[1] 1 2
> m[,2]
[1] 2 4 6
> m[c(1,3),]
     [,1] [,2]
[1,]    1    2
[2,]    5    6
> m[c(1,3), c(2,2)]
     [,1] [,2]
[1,]    2    2
[2,]    6    6
```

Les dimensions d'une matrice sont accessibles par son attribut *dim* :

```
> attr(m, "dim")
[1] 3 2
> dim(m)
[1] 3 2
```

Les matrices possèdent également un attribut *dimnames* permettant de nommer leurs lignes et leurs colonnes :

```
> dimnames(m) <- list(c("row1", "row2", "row3"), c("col1", "col2"))
> m
     col1 col2
row1    1    2
row2    3    4
row3    5    6
> m["row3", "col2"]
[1] 6
```

Les data.frame

Les data.frame possèdent la même structure que les matrices à la différence que les colonnes peuvent être de différents types. Il faut voir les data.frame comme étant des matrices ou chaque colonne représente une variable et chaque ligne une observation de ces variables. Cette struc-

ture est comparable à celles que l'on trouve dans les bases de données relationnelles ou encore dans les tableurs :

```
> names <- c("Pierre", "Paul", "Jacques")
> ages <- c(20, 18, 37)
> countries <- c("Belgium", "France", "Belgium")
> df <- data.frame(names, ages, countries)
> df
    names ages countries
1  Pierre   20   Belgium
2    Paul   18    France
3 Jacques   37   Belgium
```

Les data.frame sont indiçables de la même manière que le sont les matrices. De plus, chaque colonne peut être extraite individuellement à l'aide de l'opérateur *$* :

```
> df[1,3]
[1] Belgium
Levels: Belgium France
> df[c("names","countries")]
    names countries
1  Pierre   Belgium
2    Paul    France
3 Jacques   Belgium
> df$ages
[1] 20 18 37
```

Les noms des lignes et des colonnes sont accessibles via les attributs *row.names* et *names* :

```
> names(df)
[1] "names"     "ages"      "countries"
> row.names(df)
[1] "1" "2" "3"
```

Les environnements

Un environnement est une structure composée de deux choses : un ensemble de paires symbole-valeur que l'on nomme frame et un pointeur vers l'environnement qui le contient. Lorsque *R* cherche la valeur attachée à un symbole (lookup), la frame est examinée. Si le symbole s'y trouve, la valeur correspondante est retournée. Sinon, l'environnement parent est à son tour examiné et le processus se répète. Il existe 5 fonctions de base permettant de manipuler les environnements :

1. *assign(x, value, envir=,inherits=)* stocke la paire *x-value* dans l'environnement *envir*. Si *inherits* est *TRUE* les environnements parents sont également parcourus.
2. *get(x, envir=,inherits=)* retourne la valeur associée à *x* dans l'environnement *envir*. Si *inherits* est *TRUE* alors la valeur de *x* est également cherchée dans les environnements parents si elle n'est pas trouvée dans *envir*.
3. *exists(x, envir=,inherits=)* retourne un vecteur avec les noms des objets contenus dans l'environnement *envir*. Si *inherits* est *TRUE* alors *x* est également cherchée dans les environnements parents si elle n'est pas trouvée dans *envir*.
4. *objects(envir=)* retourne un vecteur avec les noms des objets contenus dans *envir*. Elle ne parcourt pas les environnements parents.
5. *remove(list =, envir=,inherits=)* supprime les objets dont le nom figure dans *list* de l'environnement *envir*. Les objets de *list* contenus dans les environnements parents peuvent également être supprimés en donnant la valeur *TRUE* au paramètre *inherits*.

Contrairement à la plupart des objets, les environnements ne sont pas copiés lorsqu'ils sont passés en arguments d'une fonction ou affectés à une variable.

```
> a <- new.env()
> b <- new.env(parent = a)
> assign("x", 1, envir = a)
> assign("y", 2, envir = a)
> objects(envir = a)
[1] "x" "y"
> exists("z", envir = a)
[1] FALSE
> c <- a
> assign("x", 10, envir = c)
> get("x", envir = c)
[1] 10
> get("x", envir = a)
[1] 10
> assign("x", 42, envir = b, inherits = TRUE)
> get("x", envir = a)
[1] 42
> objects(b)
character(0)
> assign("w", 8, envir = b, inherits = TRUE)
> w # assigne dans l'environnement global
[1] 8
```

Les fonctions

R permet à ses utilisateurs de créer leurs propres fonctions. Le langage gagne ainsi énormément en puissance, commodité et élégance. Apprendre à écrire des fonctions utiles est une des principales façons d'obtenir une utilisation de *R* confortable et productive. Une fonction est définie par une affectation de la forme :

```
> name <- function(args) body
```

Celles-ci possèdent trois composants de base : une liste d'arguments formels, un corps et un environnement. Un argument peut être soit un symbole, soit l'expression *'symbol = default'* soit l'argument spécial *'...'*. La seconde forme d'arguments est utilisée pour spécifier une valeur par défaut à un argument. Cette valeur sera utilisée si la fonction est appelée sans aucune valeur spécifiée pour cet argument :

```
> offset <- function(data, by = 1){
      data + by
  }
> data <- c(1,2,3)
> offset(data)
[1] 2 3 4
> offset(data, by = 10)
[1] 11 12 13
```

L'expression *'...'* permet d'entrer un nombre variable d'arguments. Elle est généralement utilisée lorsque le nombre d'arguments est inconnu ou quand des arguments doivent être passés à une autre fonction :

```
> fct <- function(df, graph = TRUE, ...){
    df.mean <- sapply(df, mean, na.rm = TRUE)
    if(graph){
      x <- 1:length(df)
      plot(x, df.mean, ...)
    }
    df.mean
  }
> fct(df, main = "My plot", col = "red", type = "o")
```

Lors d'un appel à une fonction, *R* tente dans un premier temps d'apparier ses arguments formels. Il créée ensuite un environnement dans lequel sera évalué le corps de celle-ci. Cet environnement a un objet assigné pour chaque argument formel, représenté par un type spécial d'objet : la promesse. Lorsque la valeur d'un symbole est recherchée, *R* examine en premier lieu l'environnement local attaché à la fonction. Si le symbole ne s'y trouve pas, il cherche dans l'environnement dans lequel la fonction fut créée, puis dans l'environnement parent, etc. Et ce jusqu'à atteindre l'environnement global. L'environnement local attaché à une fonction disparait une fois l'appel terminé.

```
> x <- 1
> y <- 7
> fct <- function(){ x <- 15; z <- 10; print(x); print(y)}
> fct() # x est locale a la fonction et y est trouve dans l'environnement parent
[1] 15
[1] 7
> x # la valeur de x dans l'environnement parent est restee inchangee
[1] 1
> z # z etait locale a la fonction et a disparu une fois l'appel termine
Erreur : objet 'z' introuvable
```

Les promesses

Les promesses font partie du méchanisme d'évaluation tardive implémenté dans *R*. Elles contiennent 3 éléments : une valeur, une expression et un environnement. Quand une fonction est appelée, les arguments sont appariés et ensuite chaque argument formel est lié à une promesse. Sont également stockés dans la promesse l'expression entrée en paramètre et un pointeur vers l'environnement où la fonction a été appelée. Tant qu'un argument n'est pas accédé, aucune valeur n'est associée à sa promesse. Par contre, lorsqu'un argument est accédé pour la première fois, l'expression stockée est évaluée dans l'environnement correspondant et le résultat est retourné. Ce dernier est également sauvé dans sa promesse.

A.2.3 Les attributs

Tous les objets, excepté NULL, peuvent avoir un ou plusieurs attributs qui leur sont attachés. Ceux-ci sont stockés dans un ensemble de paires *name* = *value*. La liste des attributs d'un objet peut être obtenue à l'aide de la fonction *attributes*. Les composants individuels sont quant à eux accédés à l'aide de la fonction *attr*. Parmi les attributs les plus souvent rencontrés on trouve *names*, *dim*, *dimnames*, *row.names*, *class*, etc. Les utilisateurs ont la possibilité de définir leurs propres attributs.

```
> df <- data.frame(names = c("Pierre", "Paul"), ages = c(18,40))
> attributes(df)
$names
[1] "names" "ages"

$row.names
[1] 1 2

$class
[1] "data.frame"

> attr(df, "status") <- c("single", "married")
> attributes(df)
$names
[1] "names" "ages"

$row.names
[1] 1 2

$class
[1] "data.frame"

$status
[1] "single"  "married"
```

A.3 Références, affectations et remplacements

Il n'existe qu'une seule manière de faire référence à un objet : par son nom. Plus précisément, par la combinaison d'un nom et d'un environnement dans lequel il est évalué. Une affectation permet de donner un nom à un objet :

```
> x <- list(name = "Pierre", age = 18)
> x
$name
[1] "Pierre"

$age
[1] 18
```

Dans notre exemple, une liste est créée et le nom *x* lui est attribué. Tant qu'aucune autre affectation n'est effectuée pour *x* dans cet environnement nous pouvons être certain que sa valeur restera inchangé. Nous verrons à la section A.5 que ce n'est pas toujours le cas. Cela a une implication importante, considérons le code suivant :

```
> y <- x$name
> y
[1] "Pierre"
> y <- "Jean"
> x$name
[1] "Pierre"
```

La première affectation extrait un composant de *x* et lui attribue le nom *y*. Nous pourrions penser que *y* n'est rien d'autre qu'une référence vers l'élément *age* de *x* mais ce n'est pas le cas. Ici, une nouvelle référence vers un objet *y* est créée avec comme valeur, l'évaluation de *x$name*. Ainsi, toute modification de *y* ne modifiera pas *x*. De la même façon, lorsque des arguments sont passés à une fonction, *R* crée un nouvelle référence locale pour chacun d'eux. Les objets créés peuvent alors être utilisés sans ambiguité au sein de la fonction. Considérons désormais la portion de code suivante :

```
> x$age <- 40
> x
$name
[1] "Pierre"

$age
[1] 40
```

Cette dernière laisse à penser que l'affectation modifie la valeur *age* référencée par l'objet *x*. Une nouvelle fois, ce n'est pas le cas. Ce type d'affectation est appelé un remplacement : un nouvel objet est créé et assigné au symbole *x*. Il est important de se souvenir de cet aspect du langage lorsque l'on remplace des portions de gros objets. En effet, chaque remplacement va effectuer l'affection complète du nouvel objet, aussi petite que soit la modification. Prenons un exemple :

```
> fct <- function(x){
    for(i in 1:length(x))
      x[i] <- sqrt(x[i])
    x
  }
> f1(1:1000)
```

Lors de l'appel à *fct*, la boucle va effectuer 1000 itérations et à chacune d'elles, un nouvel objet contenant un vecteur de 1000 éléments va être créé et affecté au symbole *x*. Ce type d'opération est clairement inefficace. Dans ce genre de cas il est préférable d'utiliser la vectorisation.

A.4 La vectorisation

Un vecteur est consistué d'une série de n éléments. Bon nombre d'applications pratiques les utilisent et possèdent des temps d'exécutions proportionnels à leur longueur (αn avec $\alpha \in \mathbb{R}$). Le but de la vectorisation est de trouver une forme de calcul capable de réduire cette constante α. La technique consiste à remplacer toute boucle par une seule expression, opérant éventuellement sur une version étendue des données, effectuant un ou plusieurs appels de fonctions. Pour que ce changement soit utile, ces fonctions doivent traiter ces données de façon relativement efficace (il est inutile de remplacer une boucle par une fonction effectuant une boucle similaire). Généralement, ce type de fonction est codé en *C* car ce langage offre d'excellentes performances. Il reste alors à espérer que le temps de traitement d'un élément en *C* soit petit comparé à l'overhead de n appels de fonctions codées en *R*. Reprenons l'exemple de tout à l'heure :

```
> fct <- function(x){
    for(i in 1:length(x))
      x[i] <- sqrt(x[i])
    x
  }
```

Cette fonction peut être remplacée par un appel à la fonction *sapply* :

```
sapply(x, FUN = sqrt)
```

Celle-ci prend en paramètre un vecteur ou un tableau et applique la fonction *FUN* à chacun de leurs éléments. Elle utilise dans sa définition une autre fonction, *lapply*, faisant appel à une routine externe codée en *C* comme le suggère l'appel à la fonction spéciale *.Internal* :

```
> lapply
function (X, FUN, ...)
{
    FUN <- match.fun(FUN)
    if (!is.vector(X) || is.object(X))
        X <- as.list(X)
    .Internal(lapply(X, FUN))
}
<bytecode: 0x9683f18>
<environment: namespace:base>
```

Comparons désormais les temps d'exécution de ces deux versions sur un vecteur de 100.000 éléments :

```
> system.time(fct(1:100000))
utilisateur     systeme      ecoule
      0.432       0.008       0.440
> system.time(sapply(1:100000,sqrt))
utilisateur     systeme      ecoule
      0.320       0.008       0.328
```

La version vectorisée s'est montrée plus performante que son homologue itérative. L'utilisation des structures itératives est donc à proscire autant que possible.

A.5 Affectations non-locales et closures

Nous avons vu précédemment que le langage R possède un mécanisme général de mises à jour locale des objets. Il est également doté d'un autre mécanisme dans lequel les fonctions partagent un environnement commun où les mises à jour se font de façon non-locale. Celui-ci est inspiré d'autres langages orientés-objets mais n'utilise pas de définition de classe formelle. Il est rendu possible grâce à deux éléments : les affections non-locales et l'environnement local créé lors d'un appel de fonction.

N'importe quelle affectation peut être rendue non-locale en utilisant l'opérateur ¡¡-. La règle avec ce type d'affectation est de chercher le symbole dans les environnements parents, en commençant par celui où l'affectation a pris place. Si un objet du même nom est trouvé, sa valeur est mise à jour. Sinon, l'objet est lié à l'environnement global.

L'autre partie de l'astuce utilise l'environnement créé lors de tout appel de fonction. En temps normal, cet environnement disparait lorsque l'appel se termine car il n'est plus utilisé. Cependant, en définissant des fonctions internes et en les retournant comme valeur de l'appel, l'environnement ne disparait pas car il est utilisé par ses fonctions. Ces dernières peuvent alors utiliser l'affectation non-locale pour mettre à jour les variables de l'environnement. Cette technique porte le nom de *closure*. Voici un exemple :

```
bankAccount <- function(initialBalance = 0){
  balance <- initialBalance

  deposit <- function(amount){
    balance <<- balance + amount
  }

  withdraw <- function(amount){
    if(amount > 0 & balance >= amount)
      balance <<- balance - amount
  }

  getBalance <- function(){
    return(balance)
  }

  return(list(deposit = deposit, withdraw = withdraw, getBalance = getBalance))
}
```

Dans cet exemple nous souhaitons créer un compte bancaire. Pour cela, nous créons une fonction *bankAccount* qui prend en paramètre un argument facultatif étant le solde de départ *initialBalance*. Cette fonction possède dans son environnement local plusieurs éléments : une variable *balance* et plusieurs fonctions l'utilisant. Les fonctions *deposit* et *withdraw* permettent de mettre à jour la valeur de *balance*. Pour cela, elles utilisent l'affectation non-locale ¡¡-. Les 3 fonctions sont ensuite retournées comme valeur ce qui permet à l'environnement créé par l'appel à *bankAccount* de ne pas disparaitre une fois celui-ci terminé :

```
> a1 <- bankAccount(1000)
> a1$getBalance()
[1] 1000
> a1$deposit(500)
> a1$getBalance()
[1] 1500
> a2 <- bankAccount()
> a2$getBalance()
[1] 0
> a1$getBalance()
[1] 1500
```

A.6 Évaluation du design

Nous avons vu précédemment que *R* est un langage fonctionnel, dynamique et orienté-objet. Cette section s'attarde à détailler chacune de ces caractéristiques.

A.6.1 Fonctionnel

Fonctions first-class

Les fonctions sont manipulées comme n'importe quel objet du langage. Elles sont créées, liées à des symboles et peuvent être passées en arguments d'autres fonctions.

Portée lexicale

Les liaisons sont effectuées dynamiquement : des symboles peuvent être ajoutés à un environnement après qu'il ait été entré. Cela force la résolution de noms à être effectuée durant l'évaluation des fonctions.

Évaluation tardive

R n'évalue pas les arguments des fonctions lors des appels. À la place, les expressions sont stockées dans des promesses contenant un pointeur vers l'environnement dans lequel elles seront évaluées. Ces promesses sont forcées lorsque leur valeur est requise. Morandat et al [29] ont montré que généralement celles-ci ne survivent pas aux fonctions auxquelles elles sont fournies. Une autre découverte surprenante est que la résolution de noms force certaines promesses afin de déterminer si certains symboles sont liés à une fonction. Par exemple : une variable *c* et la fonction *c*.

Transparence référentielle

Un langage est référentiellement transparent si toute expression peut être remplacée par sa valeur. Autrement dit, si l'évaluation d'une expression n'a pas d'effet de bord sur les autres valeurs de l'environnement. En *R*, les arguments des fonctions sont passés par valeur et donc toute mise à jour effectuée par une fonction est uniquement visible par cette fonction.

Paramètres

Les déclarations de fonctions peuvent spécifier des valeurs par défaut et un nombre variable de paramètres. Ces arguments peuvent être passés par position, par nom ou être omis dans le cas où ceux-ci possèdent une valeur par défaut. Les fonctions sont sensibles à l'ordre d'évaluation des paramètres :

```
f <- function(a, b = min(d)){
  a <- as.matrix(a)
  d <- dim(a) # creation d'une valeur d dans l'environnement local.
  l <- b + 1 # b est evalue ici avec comme parametre le d local et non global.
```

A.6.2 Dynamique

Typage dynamique

R est dynamiquement typé : les valeurs possèdent un type mais pas les variables.

Évaluation dynamique

Le langage permet au code d'être évalué dynamiquement grâce à la fonction *eval*. Une expression non-évaluée peut être créée à l'aide de la fonction *quote*. La substitution de variable (sans évaluation) est réalisée à l'aide de la fonction *substitute*.

Extension du langage

Un atout de l'évaluation tardive est qu'elle permet d'étendre le langage sans avoir recours aux macros. De nouvelles structures de contrôle peuvent par exemple être définies à l'aide des fonctions vues au point précédent :

```
myWhile <- function(cond, body){
  repeat if(!eval.parent(substitute(cond))) break
  else eval.parent(substitute(body))
}
```

Manipulation de l'environnement

Les utilisateurs peuvent créer des environnements, modifier des existants, les inclure sur la pile. Les fonctions peuvent voir leurs paramètres, leurs corps, leurs environnement changés après avoir été définis. La pile d'appels est également accessible.

A.6.3 Orienté-objet

Objets S3

Dans le modèle S3, l'orienté-objet est rendu possible grâce à l'attribut *class*. Sa seule sémantique est de spécifier l'ordre de résolution des méthodes génériques. Les méthodes S3 sont des fonctions possédant dans leur corps un appel à la fonction *UseMethod*. Celle-ci prend en paramètre une chaine de caractères *name* et recherche la fonction *name.cl* où *cl* est une des valeurs de *class* :

```
> getType() <- function(x) UseMethod("getType")
> getType.food <- function(x) print("Food")
> getType.default <- function(x) print("Unknow")
> chocolate <- "Milka"
> getType(chocolate)
[1] "Unknown"
> class(chocolate) <- "food"
> getType(chocolate)
[1] "Food"
```

Objets S4

Contrairement à S3, le modèle S4 implémente un vrai système de classes. Une classe est définie par un appel à *setClass* avec comme paramètres une liste ordonnée de classes parents (l'héritage multiple est autorisé) et une représentation. La représentation d'une classe est l'ensemble des champs qui la constituent. Les objets d'une classe sont instanciés par un appel à *new*. Un prototype est alors créé avec les arguments de *new* et passé à la fonction générique *initialize* qui copie le prototype dans les champs du nouvel objet. Les classes sans représentation ou ayant VIRTUAL dans leur représentation sont des classes abstraites qui ne peuvent être instanciées :

```
> setClass("Point", representation = (x = "numeric", y = "numeric"))
> setClass("Color", representation = (color = "character"))
> setClass("CP", contains = c("Point", "Color")) # heritage multiple
> p <- new("Point", x = 1, y = 1)
> c <- new("Color", color = "red")
> c@color
[1] "red"
```

Les méthodes sont introduites à l'extérieur de la classe par un appel à *setGeneric*. Elles sont ensuite définies par la fonction *setMethod* :

```
> setGeneric("add", function(a,b) standardGeneric("add"))
> setMethod("add", signature("Point", "Point"),
    function(a,b) new("Point", x = a@x + b@x, y = a@y + b@y))
> setMethod("add", signature("CP", "CP"),
    function(a,b) new("CP", x = a@x + b@x, y = a@y + b@y, color = a@color))
```

La combinaison des fonctions génériques et de l'évaluation tardive a comme désavantage de voir les promesses évaluées lorsque une méthode est recherchée, et ce afin de pouvoir accéder à la classe des arguments.

A.7 Évaluation de l'implémentation

Toujours sur base des travaux de Morandat et al, nous allons analyser les performances de *R* et les comparer à deux autres langages populaires : *C* et *Python*. Pour cela, les auteurs ont utilisé les benchmarks fournis par *Shootout* [31]. Ils ont également utilisé les packages de *BioConductor* [28], un projet open source dans le domaine de la bio-informatique, afin d'examiner les performances de *R* lors d'usages typiques. Leurs résultats ont été obtenus à l'aide d'un framework que les auteurs ont developpé : *TraceR* [32].

A.7.1 Temps

Les tests appliqués sur les programmes de *Shootout* ont révélé qu'en moyenne *R* est 501 fois plus lent que *C* et 43 fois plus lent que *Python*. En explorant les packages de *BioConductor*, les auteurs ont décelé qu'en moyenne 30% du temps d'exécution est dédié à la gestion de la mémoire contre 40% utilisé par les fonctions developpées. Quant à la recherche de symboles, elle occupe en moyenne 6% du temps total. Étant donné la nature du langage, de nombreuses fonctions sont codées en *C* ou en *Fortran*. Nous devrions nous attendre à ce que le temps d'exécution soit dominé par ces librairies natives. Cependant, à la vue des tests, celles-ci n'occupent que 22% du temps.

A.7.2 Mémoire

R est non seulement lent mais il consomme aussi une quantité importante de mémoires. Contrairement à *C* où les données peuvent être allouées sur la pile, toutes les données utilisateur sont placées sur le tas et collectées par le garbage collector. Les analyses ont montré que la quantité de mémoires allouées par *R* est d'un ordre de grandeur clairement supérieur à celle accaparées par *C*. De plus, dans de nombreux cas, la mémoire employée par les données internes est supérieure à celle utilisée par les données utilisateur. En moyenne, 37% des arguments finissent par être copiés. 36% des vecteurs contenus dans *BioConductor* ne contiennent qu'un seul nombre. Étant donné qu'un vecteur vide consomme 40 octets, on comprend mieux d'où vient cette grande quantité de mémoires. Cela a également un impact sur le temps d'exécution car chaque vecteur doit être déréférencé, alloué et collecté par le garbage collector.

A.7.3 Conclusion

Le langage est clairement lent et gère inefficacement la mémoire, du moins beaucoup plus que d'autres langages dynamiques. La grande diversité de caractéristiques ainsi que l'absence de types intégrés sont certainement à l'origine de cette faiblesse. Mais cette grande diversité de caractéristiques en font également un langage riche et flexible une fois maitrisé. Sa syntaxe simple et concise, son caractère open source et le nombre important de packages activement maintenus par la communauté sont autant de points positifs que l'on peut attribuer à ce langage.

Annexe B| Diagrammes et codes

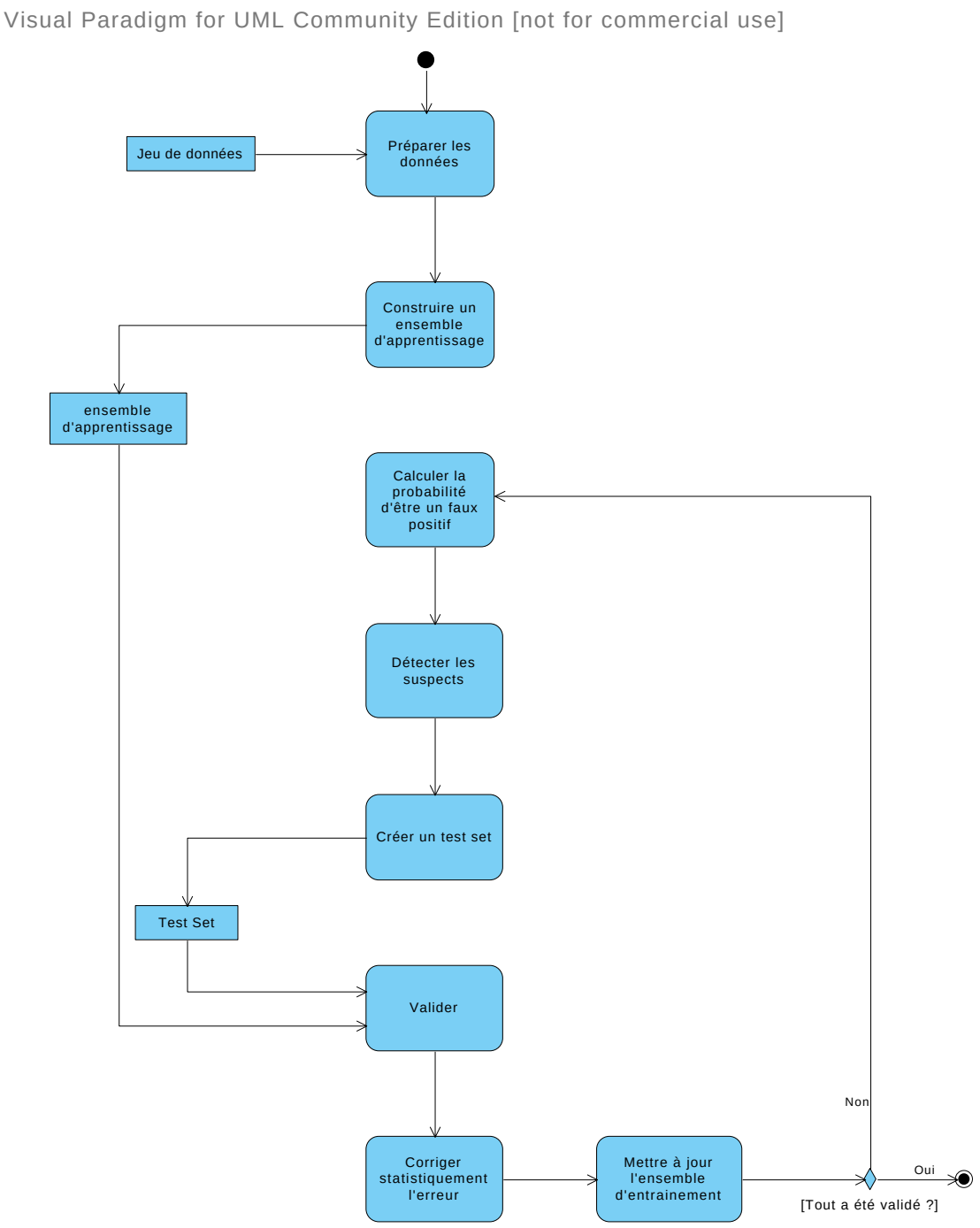


Figure B.1 – *Diagramme d'activité du code d'origine.*

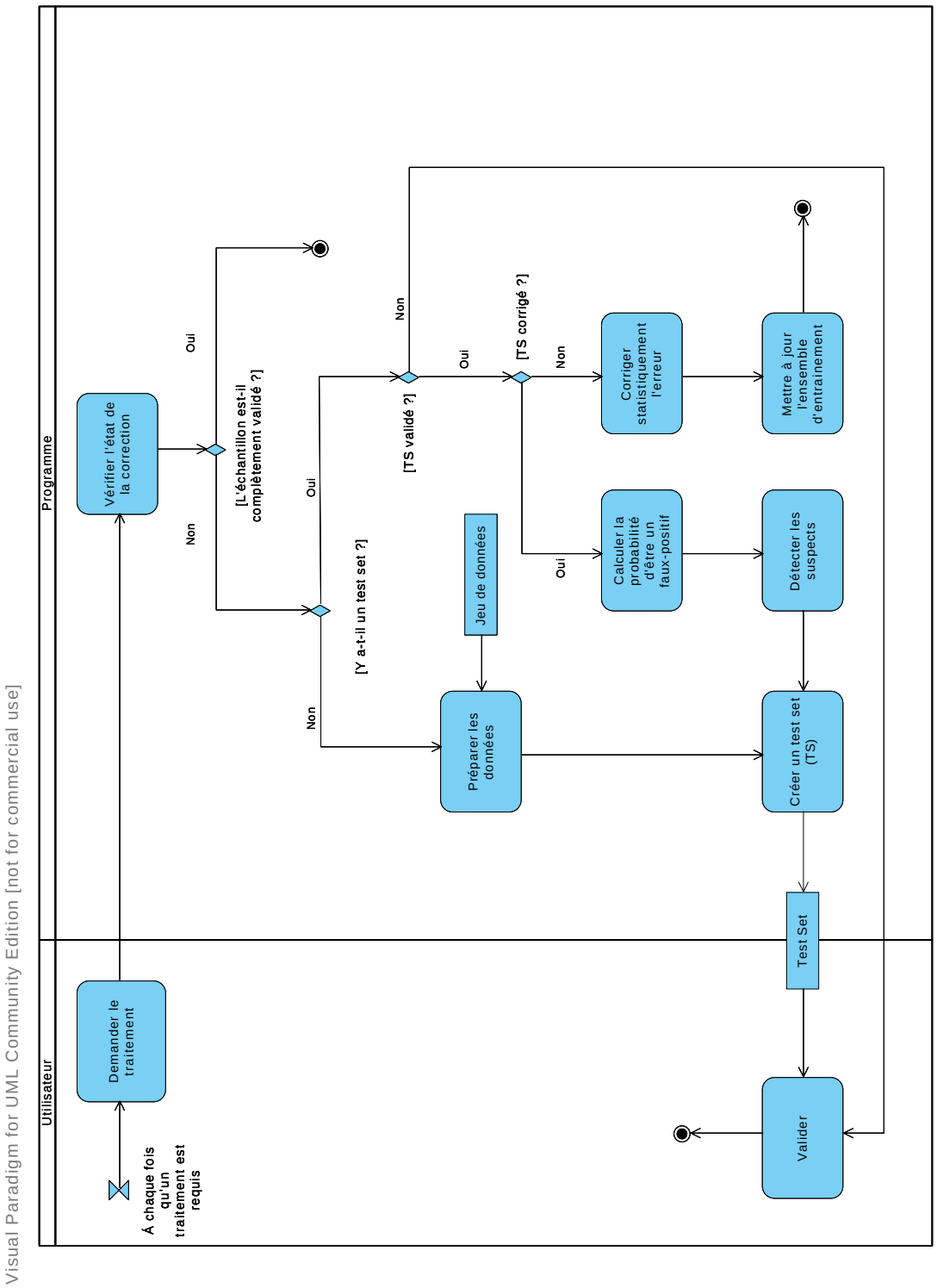


Figure B.2 – *Diagramme d'activité de la structure adaptée.*

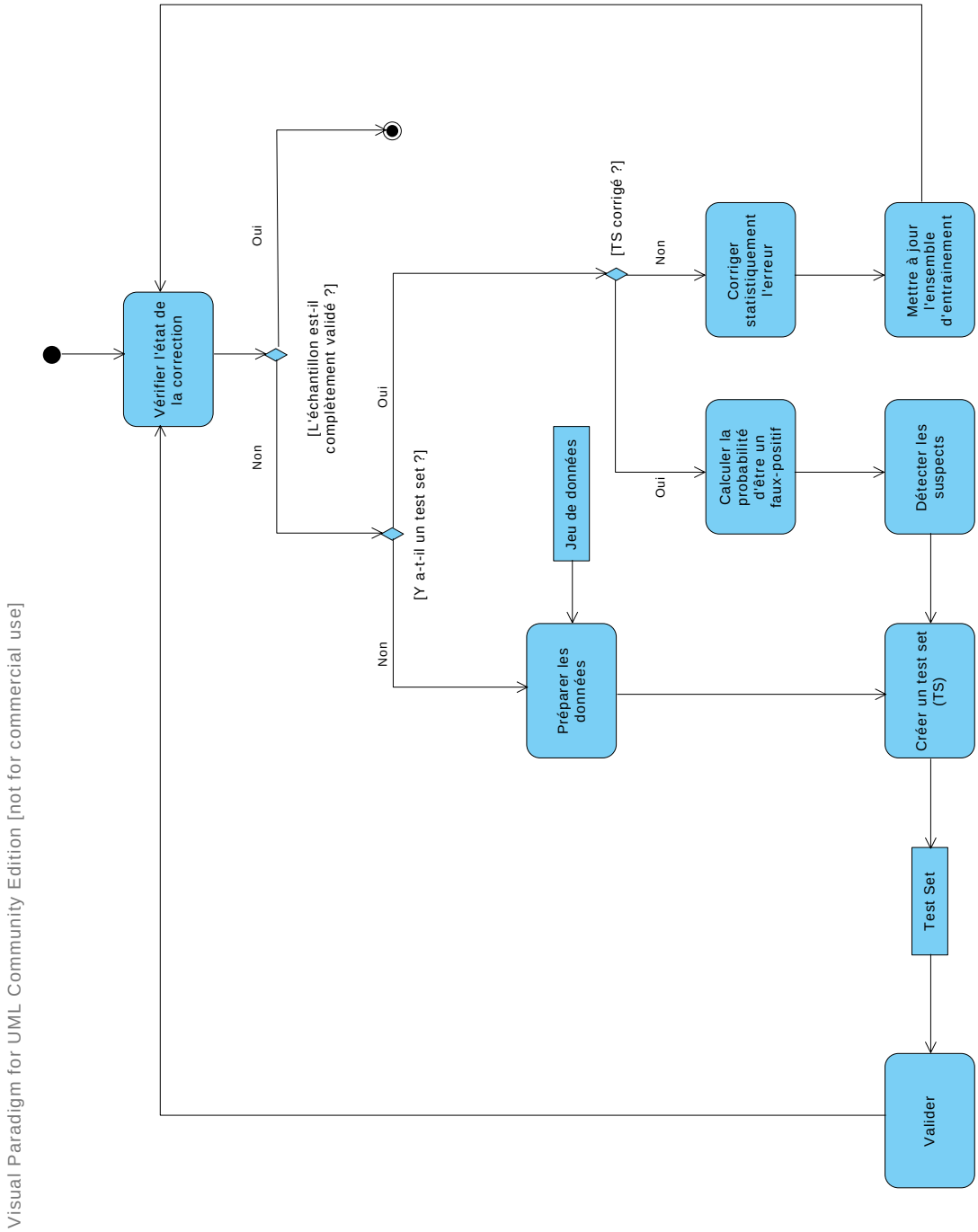


Figure B.3 – *Diagramme d'activité du mode stats.*

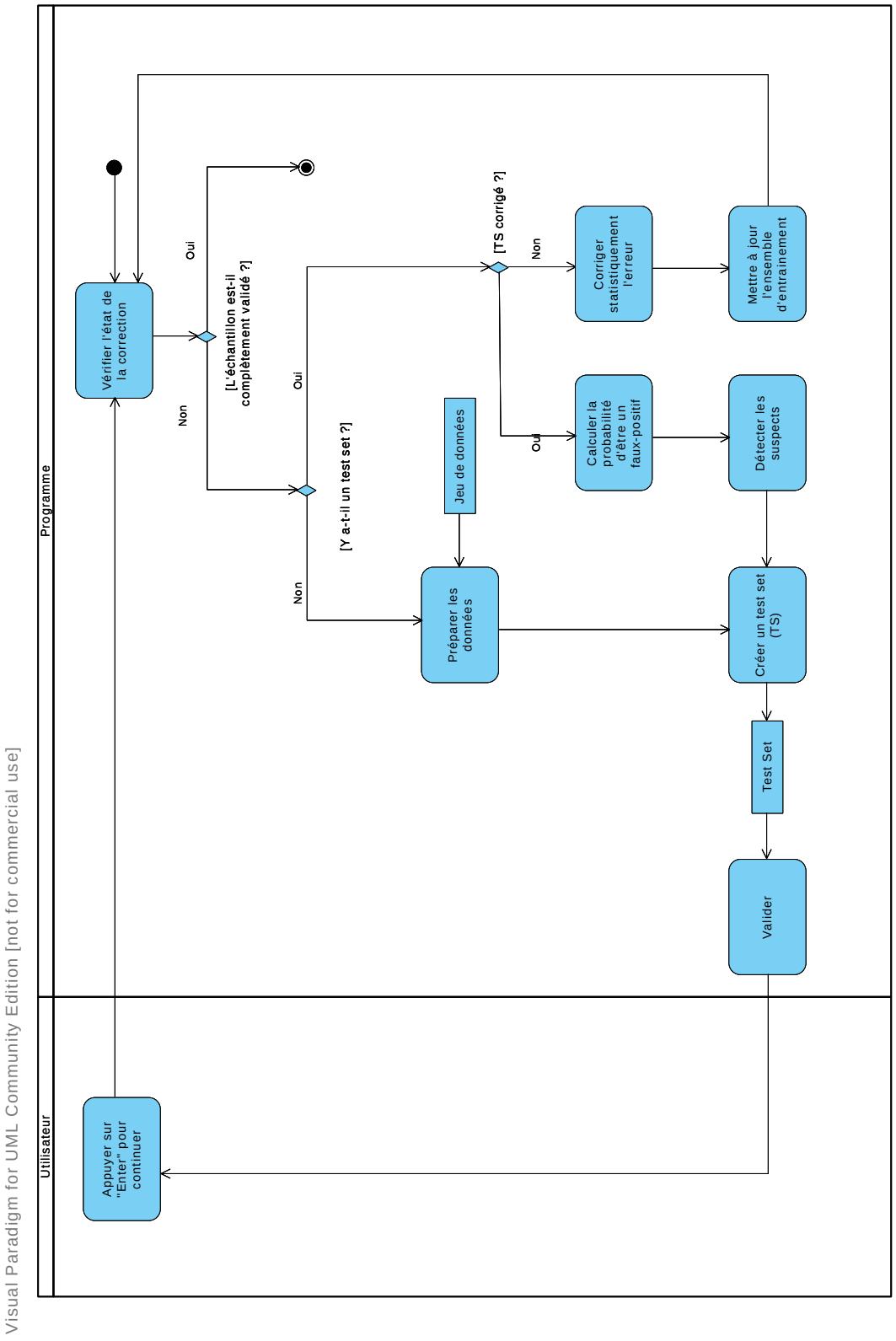


Figure B.4 – *Diagramme d'activité du mode démo.*

Bibliographie

[1] Service d'écologie numérique des milieux aquatiques. Étude du plancton marin. http://portail.umons.ac.be/FR/universite/facultes/fs/services/institut_bio/ecologie_numerique_milieux_aquatiques/Pages/PLANCTON.aspx, Janvier 2013.

[2] Philippe Grosjean. Analyze your plankton through digitized images. http://www.sciviews.org/zooimage/, Janvier 2013.

[3] M.C. Benfield, P. Grosjean, P.F. Culverhouse, X. Irigoien, M.E. Sieracki, A. Lopez-Urrutia, H.G. Dam, Q. Hu, C.S. Davis, A. Hansen, et al. Rapid : research on automated plankton identification. 2007.

[4] CS Davis, SM Gallager, MS Berman, LR Haury, and JR Strickler. The video plankton recorder (vpr) : design and initial results. *Arch. Hydrobiol. Beih*, 36 :67–81, 1992.

[5] P. Grosjean, M. Picheral, C. Warembourg, and G. Gorsky. Enumeration, measurement, and identification of net zooplankton samples using the zooscan digital imaging system. *ICES Journal of Marine Science : Journal du Conseil*, 61(4) :518–525, 2004.

[6] X. Irigoien, P. Grosjean, and A. Lopez-Urrutia. Image analysis to count and identify plankton. 2005.

[7] Coursera. Machine learning. https://class.coursera.org/ml/class/index, Janvier 2013.

[8] Ian H. Witten and Eibe Frank. *Data Mining : Practical Machine Learning Tools and Techniques*. The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann Publishers, San Francisco, CA, 2nd edition, 2005.

[9] Jiawei Han and Micheline Kamber. *Data mining : concepts and techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[10] S.J. Russell, P. Norvig, E. Davis, S.J. Russell, and S.J. Russell. *Artificial intelligence : a modern approach*. Prentice hall Upper Saddle River, NJ, 2010.

[11] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining, (First Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[12] T. Luo, K. Kramer, D.B. Goldgof, L.O. Hall, S. Samson, A. Remsen, and T. Hopkins. Active learning to recognize multiple types of plankton. In *Pattern Recognition, 2004. ICPR 2004. Proceedings of the 17th International Conference on*, volume 3, pages 478–481. IEEE, 2004.

[13] A. Tunin Ley and D. Maurer. Mise en œuvre opérationnelle d'un système couplé de numérisation (flowcam) et de traitement d'images (phytoimage) pour l'analyse automatisée, ou semi-automatisée, de la composition phytoplanctonique d'échantillons d'eau de mer-premières étapes. 2011.

[14] L. Breiman. Bagging predictors. *Machine learning*, 24(2) :123–140, 1996.

[15] L. Breiman. Random forests. *Machine learning*, 45(1) :5–32, 2001.

[16] Sebastían Maldonado and Gaston L'Huillier. Svm-based feature selection and classification for email filtering. In *Pattern Recognition-Applications and Methods*, pages 135–148. Springer, 2013.

[17] Paul Viola and Michael Jones. Rapid object detection using a boosted cascade of simple features. In *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, volume 1, pages I–511. IEEE, 2001.

[18] C. Cortes and V. Vapnik. Support-vector networks. *Machine learning*, 20(3) :273–297, 1995.

[19] John Platt et al. Sequential minimal optimization : A fast algorithm for training support vector machines. 1998.

[20] Andrew Solow, Cabell Davis, and Qiao Hu. Estimating the taxonomic composition of a sample when individuals are classified with error. *Marine Ecology Progress Series*, 216 :309–311, 2001.

[21] Q. Hu and C. Davis. Automatic plankton image recognition with co-occurrence matrices and support vector machine. *Marine Ecology Progress Series*, 295 :21–31, 2005.

[22] Q. Hu and C.S. Davis. Accurate automatic quantification of taxa-specific plankton abundance using dual classification with correction. 2006.

[23] P.F. Culverhouse, R. Williams, B. Reguera, V. Herry, and S. González-Gil. Do experts make mistakes ? a comparison of human and machine identification of dinoflagellates. *Marine Ecology Progress Series*, 247 :17–25, 2003.

[24] Philippe Grosjean. svunit. `http://cran.r-project.org/web/packages/svUnit/index.html`.

[25] Ross Ihaka and Robert Gentleman. R : A language for data analysis and graphics. *Journal of computational and graphical statistics*, 5(3) :299–314, 1996.

[26] John Chambers, Richard Becker, and Allan Wilks. *The new S language*. 1988.

[27] Gerald Jay Sussman and Guy L Steele Jr. Scheme : A interpreter for extended lambda calculus. *Higher-Order and Symbolic Computation*, 11(4) :405–439, 1998.

[28] Bioconductor Core Team. Bioconductor. `http://www.bioconductor.org/`.

[29] Floréal Morandat, Brandon Hill, Leo Osvald, and Jan Vitek. Evaluating the design of the r language. In *ECOOP 2012–Object-Oriented Programming*, pages 104–131. Springer, 2012.

[30] John M. Chambers. *Software for Data Analysis : Programming with R*. Springer, New York, 2008.

[31] Shootout. Benchmarks game. `http://benchmarksgame.alioth.debian.org/`.

[32] Purdue R Core Team. Tracer. `http://r.cs.purdue.edu/dload.php`.

Rapport préliminaire

Amélioration de la validation partielle des suspects et de la correction statistique de l'erreur dans Zoo/PhytoImage pour les échantillons REPHY

Kevin Denis & Philippe GROSJEAN

Introduction

L'identification du phytoplancton sur base de l'analyse d'images numériques et grâce aux algorithmes de classification supervisée, telle qu'implémentée dans Zoo/PhytoImage permet de discriminer plusieurs dizaines de groupes planctoniques avec un taux d'erreur total de l'ordre de 20% (voir rapports antérieurs). Lorsque l'erreur est analysée groupe par groupe, nous constatons une variation extrême avec les groupes les plus faciles à discerner qui ne montrent quasiment pas d'erreur du tout, et d'autres où l'erreur avoisine les 100%. Evidemment, nous sommes en droit de nous demander quelle est l'utilité de dénombrement de plancton ou de calcul de biomasses qui se basent sur des estimations entachées de plus de, disons, 5% d'erreur.

C'est à cause de cette limitation que l'utilisation actuelle de ces techniques en pratique passe par une étape de validation totale. Le spécialiste inspecte visuellement l'ensemble de la classification réalisée par l'ordinateur et corrige manuellement tous les cas erronés. Si cette approche donne indiscutablement un taux d'erreur plus faible et sensiblement égale à une classification purement manuelle, elle ne permet pas un gain de temps substantiel. Le gain n'est qu'occasionné par un pré-tri réalisé par l'ordinateur.

L'an dernier, nous nous sommes posés la question de savoir si une validation partielle seulement pourrait être envisagée, tout en réduisant l'erreur pour tous les groupes en dessous d'un seuil acceptable (que nous fixons subjectivement à 5 %, mais qui pourra varier, bien sûr, en pratique selon l 'objectif suivi). Nous avons constaté qu'il est possible d'identifier les particules qui ont une plus forte probabilité d'être mal classée, en recoupant plusieurs critères indicatifs :

- La probabilité associée à la classification de la particule par l'algorithme choisi (critère interne),
- La probabilité d'être un faux positif, étant donné les distribution de toutes les particules dans l'échantillon (critère bayésien),
- La superposition éventuelle des nuages de points définissant les différents groupes (zone où il est impossible de discriminer entre deux ou plusieurs groupes,
- Et enfin, un critère « biologique » éventuellement fournit par l'utilisateur, et qui quantifie la chance qu'un groupe taxonomique se rencontre dans un échantillon en fonction de sa provenance géographique, du moment où le prélèvement a été effectué, de la température, salinité, etc... voire en fonction d'un « préscreening » rapide de l'échantillon sous microscope.

Nous avons pu démonter ainsi, qu'il est possible d'indiquer un sous-ensemble des particules qui peuvent être considérées comme suspectes, et que, dans ce sous-ensemble le taux d'erreur est effectivement très nettement supérieur au taux d'erreur moyen pour toutes les particules analysées. Autrement dit, en validat les suspects en priorité, on corrige l'erreur beaucoup plus rapidement qu'en présentant les particules à valider manuellement dans un ordre queconque.

Nous avons voulu aller également plus loin. A partir du moment où nous pouvous capturer une grande partie de l'erreur dans les premiers 10 à 20 % des particules à valider, nous avons-là une information très utile pour quantifier et modéliser cette erreur. Cette modélisation est intrinsèque à l'échantillon étudié, et donc, elle s'adapte localement à ses spécificités au plus près. Les modèles testés sont le modèle linéaire généralisé, ainsi que des algorithmes de classification supervisée. Nous avons montré ainsi que cette approche permet une correction statistique de l'erreur et améliore encore la prédiction des abondances par groupes. La correction est dite statistique parce qu'ici, nous traitons le problème dans son ensemble et pouvons par exemple prédire que l'abondance dans un groupe est probablement surestimée de, disons, 8 unités, mais nous ne pouvons pas dire lesquelles

parmi toutes les particules classées dans ce groupe sont les 8 erronées. Mais à partir du moment où ce sont les statistiques générales (abondances par groupes dans l'échantillon, biomasses par groupes dans l'échantillon) qui nous intéresse, cela n'a plus d'importance.

Une difficulté avec la correction statistique de l'erreur, c'est qu'on ne sait pas exactement quel est le taux d'erreur total. Prenons un exemple simple pour illustrer ceci. Nous avons 50 particules classées dans le groupe A. La correction statistique de l'erreur nous suggère qu'il n'y a que 42 particules dans ce groupe. Cela ne signifie pas pour autant que nous ayons seulement 8 faux positifs. Nous pourrions également avoir, par exemple, 12 faux positifs, mais accompagnés de 4 faux négatifs (des particules classées erronément dans d'autres groupes que A). En effet, cela nous donne : 50 positifs -12 faux positifs + 4 faux négatifs = 42 particules dans le groupe A. L'erreur ici est donc supérieur à 8. Pour contourner cette inconnue sur l'erreur effective, nous allons comparer les dénombrements prédits avec les dénombrements réels (tels que comptabilisés lors d'une validation totale de l'échantillon) à l'aide d'un indice de dissimilarité qui va sommer les écarts entre les deux dénombrements pour chaque groupe, pondéré par le nombre de particules dans le groupe (ceci, afin d'éviter de donner trop d'importants aux groupes les plus abondants).

Faiblesses de la méthode de correction actuelle

Le livrable précédent concernant la méthode de validation des suspects et de la correction statistique de l'erreur (y compris le logiciel qui permet de le réaliser dans Zoo/PhytoImage) doit être considéré comme une première version du concept qui nécessite encore d'être peaufinée. En effet, les cinq points suivants ont été repérés :

1. La correction statistique de l'erreur ne montre pas une progression stable allant dans le sens d'une diminution monotone décroissante de la dissimilarité. Des augmentations brusques sont observées, suite à l'inclusion ou nous de particules particulièrement difficiles à classer dans des petits sous-échantillons et dont l'effet est alors amplifié localement. Sela s'observe, en pratique, le long d'un profil de correction de l'erreur, fraction par fraction. Le profil moyen de plusieurs corrections gomme ces sauts grâce au caractère aléatoire introduit dans l'ordre de présentation des particules respectivement suspectes et non suspectes. Cependant, en pratique la correction se fait seulement selon un seul profil, et il est donc souhaitable de pouvoir gommer ces « aspérités ».
2. Nous avons remarqué que la validation des suspects uniquement au début ne permet d'inclure des erreurs qui n'auraient pas pu être détectées par les règles citées plus haut. Il est donc souhaitable d'inclure d'emblée une petite fraction de particules non supectes, afin de laisser une chance de détecter de telles erreurs et de les corriger plus vite qu'après validation de tous les suspects.
3. Nous n'avons actuellement aucun indicateur de l'erreur résiduelle totale et par groupe au cours de la validation partielle. Une validation totale est nécessaire pour pouvoir calculer de tels indicateurs. Seulement, en pratique, il est souhaitable de pouvoir décider plus tôt quand l'erreur descent en dessous d'un seuil fixé à l'avance afin de permettre de décider où arrêter la validation partielle. De tels indicateurs peuvent être dérivés du modèle de correction statistique de l'erreur, mais un travail conséquent est encore nécessaire à ce stade pour y arriver.
4. Nous avons constaté que toute l'erreur n'est pas modélisable. L'erreur présente dans la classification d'un échantillon comporte en réalité deux composantes : l'une liée à biais systématique qui découle de variations (même infimes) du système entre le set d'apprentissage et le classement de l'échantillon. Il peut s'agit d'une cause liée aux particules à classer (les particules d'un groupes sont un peu plus grandes, plus transparentes ou plus

allongées, par exemple, que celles utilisées dans le set d'apprentissage, parce que les conditions écophysiologiques sont différentes). La préparation de l'échantillon peut aussi être à l'origine de variations (plus ou moins de lugol, agitation qui a cassé les chainettes les plus fragiles des colonies, …). Il peut enfin s'agir de causes liées à la prise d'image (réglage légèrement différent de l'appareil, illumination du fond qui a très légèrement changé, mise au point moins bonne ou meilleure, …). Toutes ces variations mènent à des biais systématiques matérialisés par un léger déplacement des frontières qui séparent les mesures effectuées sur les particules des différents groupes. La méthode de correction statistique de l'erreur cible ces erreurs et effectue les corrections, localement dans le contexte de l'échantillon étudié. Cette erreur est donc modélisable et corrigeable semi-automatiquement. A côté de cette erreur, on a un certain nombre de particules de formes abbérantes, superposées à d'autres, coupées, ou encore de neige marine ou de sédiment qui ont de manière fortuite une forme et une allure très proche d'une particule phytoplanctonique réelle. Cette erreur-là est bien plus aléatoire, est difficile à identifier et impossible à modéliser (en tout cas au stade actuel de nos recherches). Il serait utile de pouvoir faire la différence entre les deux types d'erreur afin de pouvoir, grâces aux indicateurs d'erreur résideulle, savoir également quelle est la part d'erreur modélisable qui est déjà corrigée. Rien ne sert, effectivement, à vouloir modéliser au delà des biais systématiques.

5. Enfin, le dernier point est également important en pratique. Dans les études présentées dans le rapport précédent, la correction d'erreur a été utilisée en mode « stat ». Dans ce model, toutes les particules sont validées manuellement à l'avance, et nous simulons la progression de l'indice de dissimilarité en fonction de la fraction validée, ayant à disposition la vérité considérée comme l'information issue de cette validation totale. Cependant, un petit pourcentage de particules sont problématiques (de 2 à 7% dans les échantillons étudiés de Mer du Nord). Il s'agit de particules non identifiables, appartenant à des groupes très rares non inclus dans le set d'apprentissage, voire des particules multiples qui se superposent, … Habituellement, ces particules sont classées manuellement dans un groupe « autres », mais l'outil de classification automatique les place dans un des groupes connus. Il s'agit allors toujours de faux positifs, mais non associés de faux négatifs pour aucun des autres groupes ciblés. Le traitement spécial du groupe « autres » est difficile dans le cade de la classification supervisée de type « machine learning » qui n'admet pas sa présence dans les hypothèses de base d'application de la technique statistique. Pour l'instant, nous avons alors éliminés ces particules « autres » de l'étude avant calculs. Cependant, ces particules sont bel et bien présentes dans l'échantillon et doivent être considérées dans le traitement lors d'une validation en routine. Une extension de la méthode de validation des suspects et de la correction d'erreur est donc nécessaire pour prendre également en compte la présence de ces particules « autres », probablement en partie dans l'erreur systématique modélisable (avec détection automatisée des particules « autres » correspondantes) et partiellement dans l'erreur aléatoire.

Dans ce rapport préliminaire, nous avons investigué quelques pistes initiales pour répondre aux points 1-5. Nous présentons ici les premiers résultats obtenu, qui montrent que leur prise en compte est réalisable dans le cadre du présent projet, donc, d'ici fin 2014.

Validation partielle et correction d'erreur – idées

Ici, nous allons comparer la méthode de validation partielle et le calcul statistique de la correction d'erreur tels qu'implémentés dans la première version remise dans le livrable du contrat précédent avec quelques idées préliminaires visant à corriger les problèmes identifiés 1-5.

Les échantillons utilisés ici sont :

- Trois échantillons provenant d'un mélange artificiel de cultures pures de *Dytilum brightwellii*, *Chaetoceros compressus* et *Thalassiosira rotula*, numérisés à l'aide d'un FlowCAM (objectif 4X, cellume de 300µm et mode autoimage) et traités à l'aide de Zoo/PhytoImage 3. Le set d'apprentissage est réalisé à partir de la numérisation des cultures pures d'origine, nommées BE1, BE2 et BE3. Ainsi, nous sommes certains qu'il n'y a pas d'erreur d'identification possible dans le set d'apprentissage. Toutefois, ce mélange artificiel est un cas simplifié qui peut très bien donner des résultats différents d'un échantillon naturel réel.
- Pour tester la méthode sur des échantillons réels, nous avons également utilisé des échantillons issues de la zone cotière belge de la Mer du Nord (échantillons W06, W07 et W08). Ceux-ci ont été numérisé avec un FlowCAM muni d'un objectif 2X et d'une cellule de 600µm, toujours en mode autoimage. Le set d'apprentissage a été réalisé au préalable sur des échantillons provenant des mêmes stations, mais collectés l'année précédente. Nous nous rapprochons donc, ici d'une situation réelle d'échantillons naturels où le set d'apprentissage provient d'analyses antérieures dans le temps. Vingt cinq groupes sont réalisés, dont 18 groupes de phytoplancton, 3 de zooplancton et 4 pour les particules inertes.

Par rapport à la version initiale de l'algorithme de validation des suspects et de correction d'erreur, nous avons progressivement introduit et testés les améliorations suivantes :

- Recalcul de la détection des suspects tranche par tranche de 5 % en 5 %, et en utilisant un nouvel outil de classification utilisant les données du set d'apprentissage initial auquel nous avons rajouté les particules validées de l'échantillon. Nous obtenons donc un algorithme qui s'adapte mieux à la situation particulière locale de l'échantillon. Cela vise à corriger partiellement le problème identifié #2 en permettant de redétecter d'autres suspects au fur et à mesure que l'identification des particules s'affine.
- Un premier sous-ensemble de 5 % des particules est sélectionne totalement aléatoirement pour estimation initiale de l'erreur commise lors de la classification automatique et une première estimation aussi pour les groupes les plus abondants. Cela correspond à une estimation précoce de l'erreur, soit le problème #3.
- Dans les ensembles suivants, nous incluons systématiquement une fraction de particules non suspectes (ici, 10%) parmi les suspects choisis en priorité. Cette pratique vise à corriger le problème #2, ainsi que d'apporter des données supplémentaires pour estimer l'erreur résiduelle (problème #3).
- La matrice de confusion totale est recalculée en considérant deux composantes complémentaires liées aux suspects corrigés et aux non suspects corrigés. La méthode précédente considérait par défaut les non suspects comme corrects. Celui induisait des comportements étranges lorsque tous les suspects ont été validés et que l'utilisateur corrige des non suspects... supposés par la méthode comme corrects. Cette nouvelle parroche semble lisser la diminution de dissimilarité au long de la fraction validée par rapport à la méthode initiale (résolution partielle -ou totale?- du problème #1).
- Nous comparons maintenant les résultats de correction statistiques avec les corrections manuelles uniquement. Lorsque les deux donnent des résultats très similaires, nous considérons que nous avons modélisé tous les biais systématiques et qu'il ne reste alors plus que de l'erreur aléatoire. Dans ce cas, nous n'avons plus besoin de la correction statistique de l'erreur et les deux courbes convergent. Cela répond, au moins partiellement, au problème #4. Nous considérons que la réponse est partielle car la part d'erreur modélisable dépend fortement de l'algorithme de correction d'erreur et il nous faudrait encore investiguer de ce côté-là pour en améliorer la puissance, si c'est possible.

- Enfin, nous n'avons pas encore eu l'occasion d'aborder le problème #5, mais il nous conduira à définir une nouvelle classe « autre » dans l'algorithme de correction d'erreur, ce qui est techniquement faisable. Une piste existe, donc.

Les Figures 1 à 6 présentent une comparaison de la correction d'erreur par validation des suspects (en vert), et par correction statistique d'erreur (en rouge). Les sous-figures de gauche correspondent à la méthode d'origine. Les sous-figures de droites implémentent toutes les modifications proposées ci-dessus. Le gain est sensible. Et nous arrivons à de très bons résultats dans tous les cas en ne validant que 10 % ou 20 % de l'échantillon.

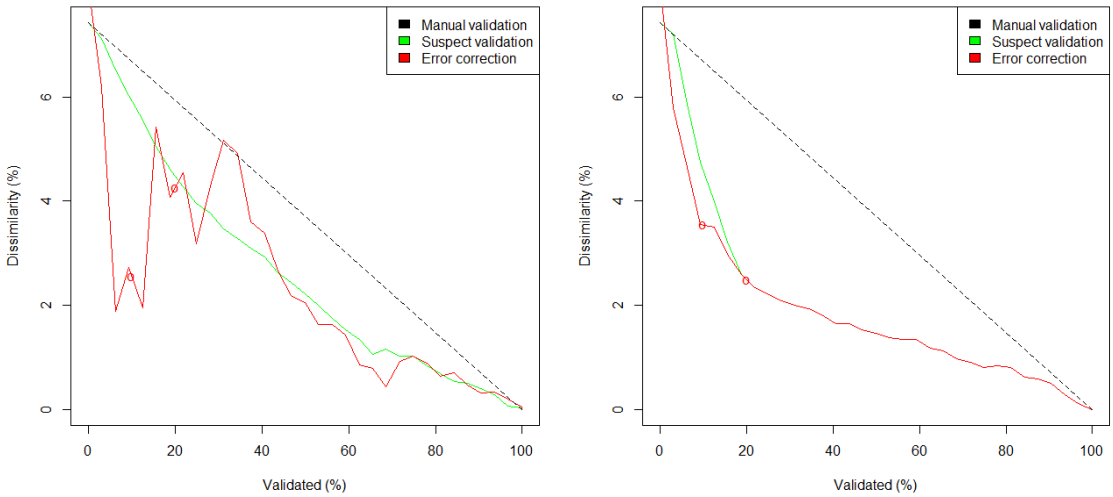


Figure 1 : *validation et correction d'erreur de l'échantillon BE1 (un seul profil). A gauche, méthode initiale et à droite méthode améliorée.*

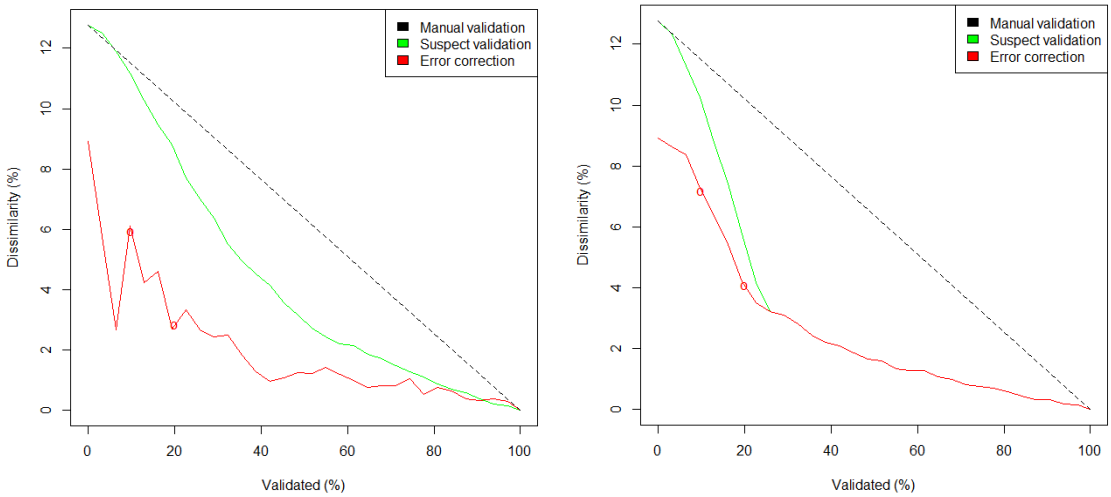


Figure 2 : *validation et correction d'erreur de l'échantillon BE2 (un seul profil). A gauche, méthode initiale et à droite méthode améliorée.*

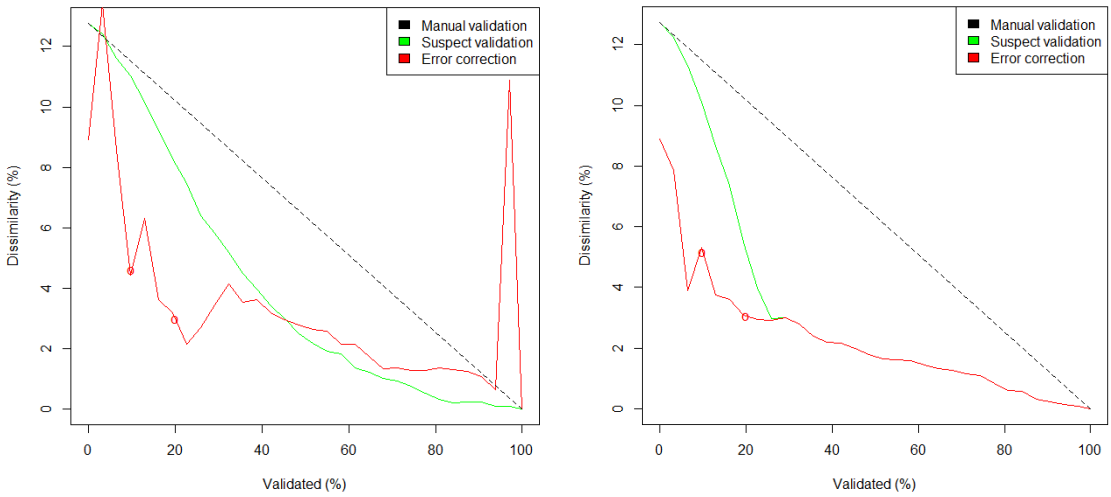


Figure 3 : *validation et correction d'erreur de l'échantillon BE3 (un seul profil). A gauche, méthode initiale et à droite méthode améliorée.*

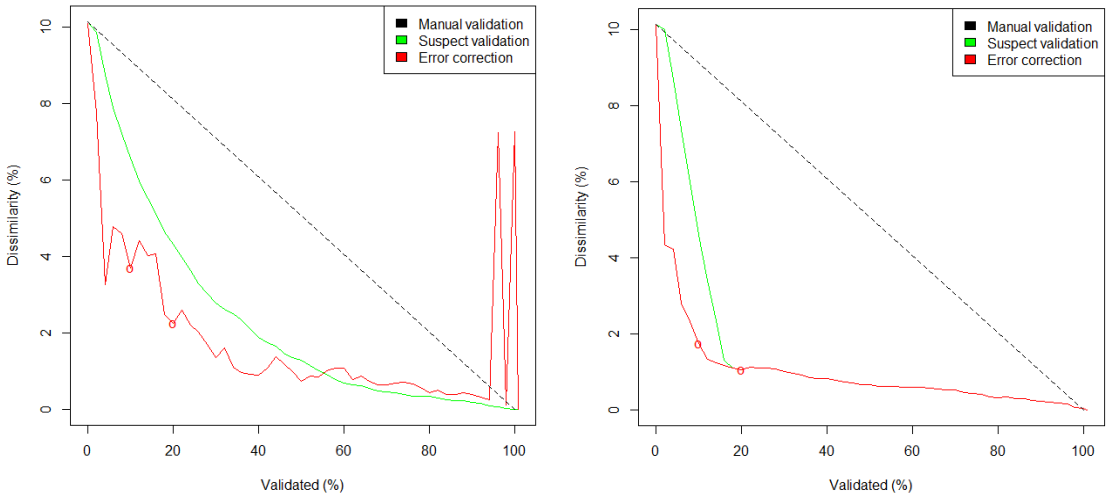


Figure 4 : *validation et correction d'erreur de l'échantillon W06 (un seul profil). A gauche, méthode initiale et à droite méthode améliorée.*

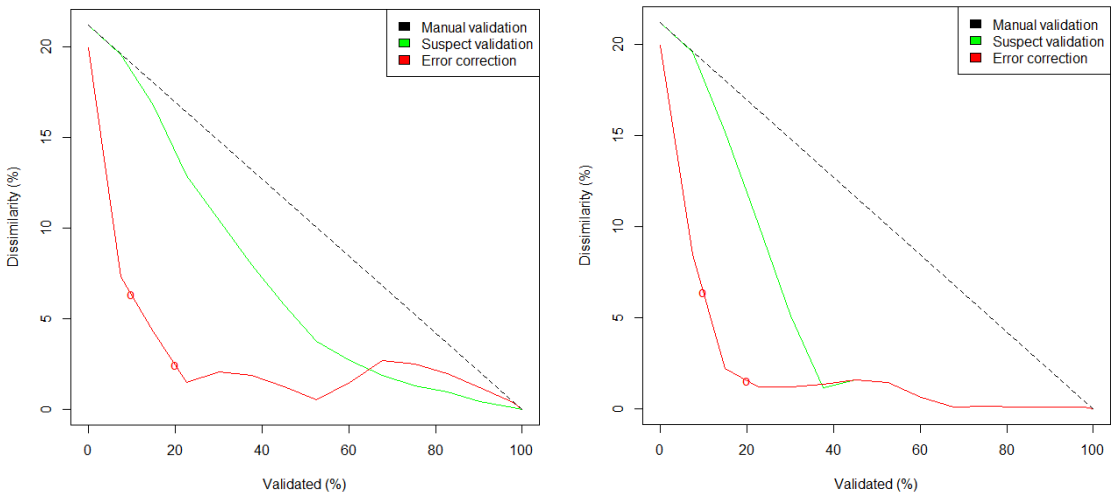


Figure 5 : *validation et correction d'erreur de l'échantillon W07 (un seul profil). A gauche, méthode initiale et à droite méthode améliorée.*

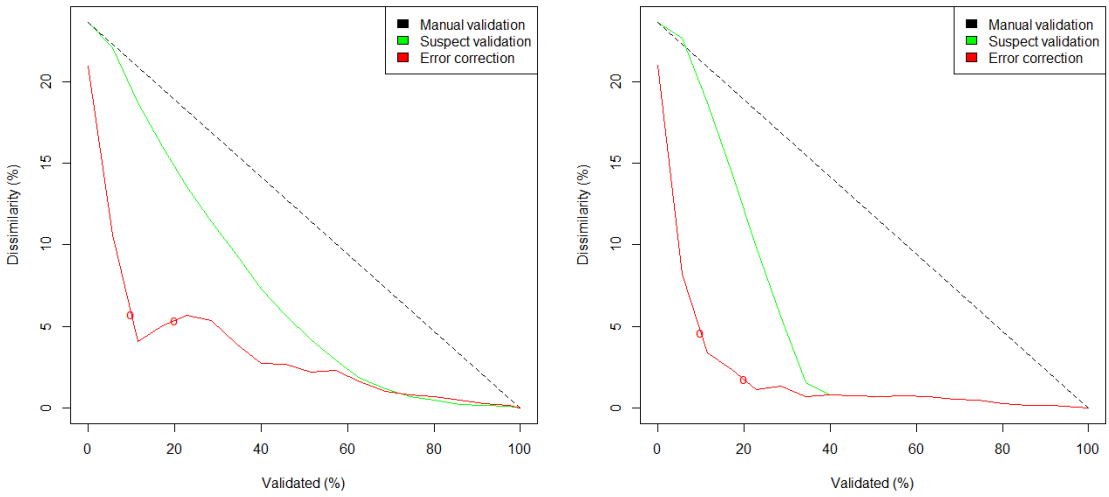


Figure 6 : *validation et correction d'erreur de l'échantillon W08 (un seul profil). A gauche, méthode initiale et à droite méthode améliorée.*

Conclusions préliminaires

La méthode de détection et de validation préférentielle des suspects, éventuellement associée à une modélisation et correction statistique de l'erreur apparaît comme une solution potentiellement intéressante pour obtenir un dénombrement de groupes phytoplanctoniques dans un échantillon en utilisant l'analyse d'image et la classification supervisée. A l'aide de cette méthode, nous sommes capables de contenir l'erreur à des niveaux acceptables de 5 % ou moins pour tous les groupes, tout en n'étant pas obligé de valider manuellement l'ensemble de l'échantillon. Ainsi, nous pouvons espérer ne devoir valider que 10 % ou 20 % de l'ensemble des particules de l'échantillon. Nous avons donc un gain substantiel de 4/5 à 9/10 du temps d'une des opérations les plus longues dans le traitement (semi)-automatisé des échantillons du REPHY, tel qu'il est envisagé à l'aide du

FlowCAM et de Zoo/PhytoImage très prochainement.

Cependant, cette méthode est encore nouvelle, et comme elle implique des calculs complexes et de jongler entre deux modèles (le premier qui classe les particules et le deuxième qui détermine qui est suspect) qui sont interdépendants, il n'est pas façile de comprendre et d'optimiser son fonctionnement. Une première version a toutefois été implémentée dans la version actuelle de Zoo/PhytoImage dans le cadre d'une collaboration antérieure IFREMER/UMONS.

Ici, nous proposons d'investiguer cinq point faibles de la première version de correction, identifiés d'après son utilisation sur des échantillons artificiels ou naturels. Dans le présent rapport, nous avons présenté ces cinq problèmes, ainsi que les pistes investiguées jusqu'ici pour les résoudre. Ben que nous avons ici des résultats préliminaires, nous pouvons observer une amélioration substance de la correction d'erreur et un comportement moins erratique dans la correction statistique de cette erreur dans la version actuelle de l'algorithme de correction.

Par la suite, nous proposons de poursuivre ces pistes, de déterminer si nous pouvons concevoir un meilleur modèle de l'erreur systématique présente dans l'échantillon (celle qui est corrigeable statistiquement), et nous proposons d'incorporer aussi la prise en compte du groupe « autre » dans cette correction. Enfin, nous élaborerons un code optimisé pour la version 4 de Zoo/PhytoImage afin que cette technique soit pleinement opérationnelle dans le cadre de la numérisation d'échantillons de phytoplancton pour le REPHY.