FACULTY OF SCIENCE AND TECHNOLOGY

MASTER'S THESIS

Study programme / specialisation:
Computational Engineering
Department of Energy Resources

The ***spring*** semester, ***2023***

Open / Confidential

Author: John Emeka Udegbunam & Reynel Isaac Villabona Gonzalez

Supervisor at UiS: Nan Zhang

Co-supervisor: Rong Chunming

External supervisor(s):

Thesis title: A Federated Computational Workflow for Analysis of DISKOS Digital Palynological Slides

Credits (ECTS):30

Keywords: Computational Workflow, Federated Computation, DISKOS, Digital Palynological Slides, Palynological Slides, Image Processing, Object detection, Classification, watershed segmentation.

Pages: **61**
+ appendix: **101 pages**

Stavanger, ***15/07/2023***

Approved by the Dean 30 Sep 21
Faculty of Science and Technology

ABSTRACT

A novel federated computational workflow for analyzing digital palynological slide images is implemented in this thesis. The slide data files, typically exceeding 3GB, present significant data mobility and computation challenges. The novel distributed computational framework is implemented to address privacy concerns and the challenges associated with moving large data. The idea is to move computational to the data location, optimally utilizing local computational capacity and reducing data movement.

Trained deep-learning models deployed in a containerized environment leveraging the Docker technology are integrated in the workflow with a user-friendly interface, and users can run processes with the trained models.
The workflow processes include reading slide image files, generating tiled images, and identifying and removing undesirable tiles such as blank tiles. Object detection with the watershed segmentation algorithm identifies tiles with potential microfossils. The identified dinoflagellates are classified with a trained convolution neural network (CNN) model. The classification results are sent to the host and shared with the users.

The federated computational approach effectively addresses the challenges related to moving and handling large palynological slide images, creating a more efficient, scalable, and distributed pipeline. Collaborative efforts involving domain experts for model training with more annotated slide images will improve the effectiveness of the workflow.

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my family, both in Colombia and the USA, for their unwavering support throughout my journey. Their constant presence and encouragement have played an instrumental role in shaping me into the person and professional I am today. I am forever grateful for their unconditional love and for instilling in me the importance of family values.

To my friends, both back in Colombia and those I have made in Stavanger, I extend my heartfelt appreciation. Your friendship has been a constant source of happiness, guidance, and joy in my life. You have become my second family, always there to offer advice and share unforgettable moments together.

I would like to extend my sincere thanks to Nan Zhang and Jungwon Seo for their invaluable guidance and mentorship. Their time and expertise have been instrumental in leading us to success in completing this thesis. Their dedication to our growth and development has been truly inspiring.

A special mention goes to Robert Williams for his kind assistance in explaining the digital paleontology process at NPD. His willingness to share information and support us in achieving our objectives has been greatly appreciated.

I am also grateful to John Emeka, my thesis partner, for his hard work and dedication in accomplishing all the objectives of this thesis. His commitment and collaboration have been invaluable, and I am fortunate to have had him by my side throughout this journey.

I would also like to express my gratitude to every person I have encountered since making the decision to live abroad. Each one of you has taught me something new and contributed to my personal and professional growth. Lastly, I extend my thanks to Norway for welcoming me with open arms and providing me with a second home over the past two years.

This thesis would not have been possible without the love, support, and guidance from my family, friends, mentors, John Emeka, and all those who have touched my life along this remarkable journey. Thank you all for your contributions, encouragement, and belief in me.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABBREVIATIONS

- **UI** User Interface
- **NPD** Norwegian Petroleum Directorate
- **DAG** Directed acyclic graph
- **CNN** Convolutional neural network
- **MRXS** An extension for digital slide image
- **NCS** Norwegian Continental Shelf
- **PY** An extension for Python based files
- **RCNN** Region-based Convolutional Neural Network
- **TXT** An extension for text files

CHAPTER ONE

INTRODUCTION

Digital paleontology is an innovative field that utilizes image processing techniques to analyze and classify palaeontological data (Livezey & Zusi, 2007). One significant source of this data is digital palynological slides. A digital slide image can exceed 3GB in size and provides valuable insights into the Earth's historical biosphere (Livezey & Zusi, 2007).

Figure 1.0.1 shows a region of interest extracted from a palynological slide image. Each slide contains valuable information that can be analyzed using advanced analytical techniques such as convolutional neural networks (CNNs) (Punyasena et al., 2022). Researchers can efficiently analyze large volumes of pollen samples by automating pollen taxa scanning and classifying microfossils using CNNs (Punyasena et al., 2022).

Digital techniques offer a more efficient and accurate approach compared to traditional manual methods (Mohammad et al., 2020). The integration of digital image processing and advanced analytical techniques has greatly enhanced the capabilities of digital paleontology in studying and understanding the morphology and diversity of microfossils (Johnson et al., 2016).

Data files used in various fields can introduce significant challenges regarding data mobil-

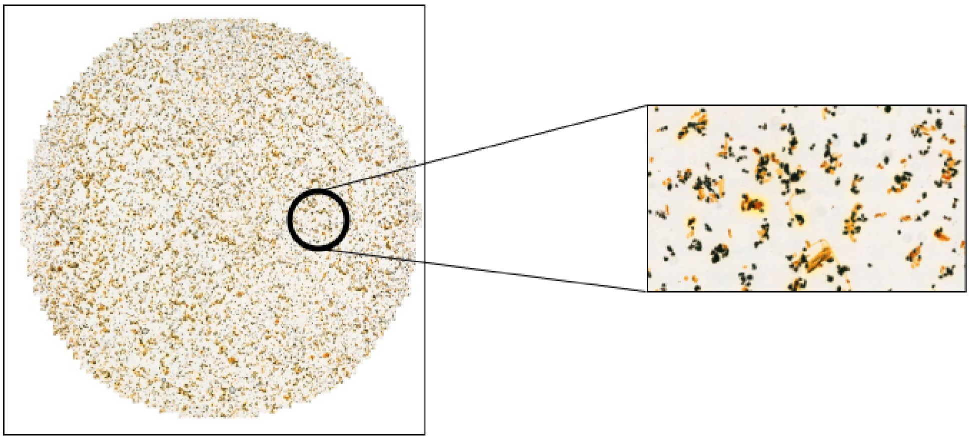


Figure 1.0.1: A digital palynological slide image

ity, computational time, and data processing. In digital paleontology, the size of digital palynological slides, which can exceed 3GB, makes it difficult to move them when analyzing hundreds of slides. Transferring whole slide images from storage to the processing location increases the computational time and associated costs (Lamani et al., 2014).
In plant omics and mass spectrometry imaging, large-scale data sets can easily reach tens of gigabytes, posing challenges in data processing (Gemperline et al., 2016).
In the life sciences, genomic data analysis involves handling datasets of variable sizes, ranging from large files (1-10 GB per file) to small files (<100 MB), presenting difficulties in data availability and management (Afgan et al., 2015).
Also, in fluid flow analysis, the large scales of spatiotemporal flow data generate massive file sizes, making data analysis, sharing, and visualization challenging (Gao, 2020). Labeling large volumes of data in sound lung analysis is labor-intensive and impedes more extensive research studies (Gemperline et al., 2016).
The challenges highlight the need for efficient data handling and processing techniques to overcome the limitations of the large data file size.

The processing of large datasets require considerable computational resources. The high resource demands can strain traditional computation systems, where data is moved to centralized processing units, causing to bottlenecks and inefficiencies (Kong et al., 2020). Traditional methods are often ineffective when applied to high-dimensional and complex data, as the data tend to be sparse and distances between points become similar (Cordeiro et al., 2013). This limitation hinders clustering large and complex datasets (Cordeiro et al., 2013).

Training computationally intensive machine-learning algorithms such as support vector machine (SVM) can be challenging when dealing with large training datasets (Guo et al., 2015). Parallel SVM algorithms such as RASMO have been developed to optimize SVM training with distributed computing resources (Guo et al., 2015). However, high-performance computing resources required for large data processing can be limited and costly, especially in resource-limited settings (Mabvakure et al., 2019).

Alternative approaches and infrastructure are needed to solve the challenges associated with handling large datasets ((Sha et al., 2020); (Mabvakure et al., 2019)). These challenges highlight the need for a more efficient method for handling and analyzing digital palynological slides.
Figure 1.0.2 illustrates the traditional data processing. In this approach, users transfer their data to a centralized computation location.

Cloud computing and data analysis have achieved considerable advancements in recent years. However, handling and processing large-scale data such as digital palynological slides remains a challenging frontier. The computational demand and data mobility challenges associated with high-resolution, large slide images pose critical challenge in digital palynology research ((Petersen et al., 1992); (K. Zhang et al., 2015)).

The present study proposes a novel federated computation workflow to address the limitations of traditional centralized approaches. The proposed workflow leverages parallel computing techniques to mitigate the high computational power requirements (Gutierrez et al., 2008). The study seeks to solve the computational and data mobility challenges. Moving the computation logic to the data location can improve data handling efficiency, minimize resource-intensive data transfers, and optimize local computational capabilities.

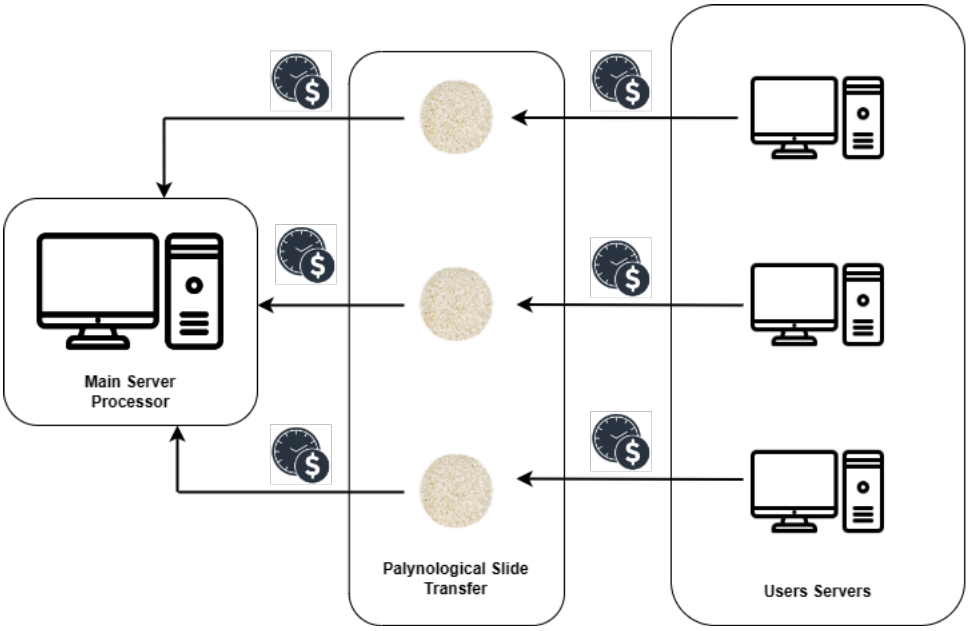


Figure 1.0.2: A centralized image processing.

The proposed innovative methodology has the potential to improve digital paleontology and inspire extensive research explorations.
Figure 1.0.3 presents a schematic of the proposed workflow for palynological slide analysis.

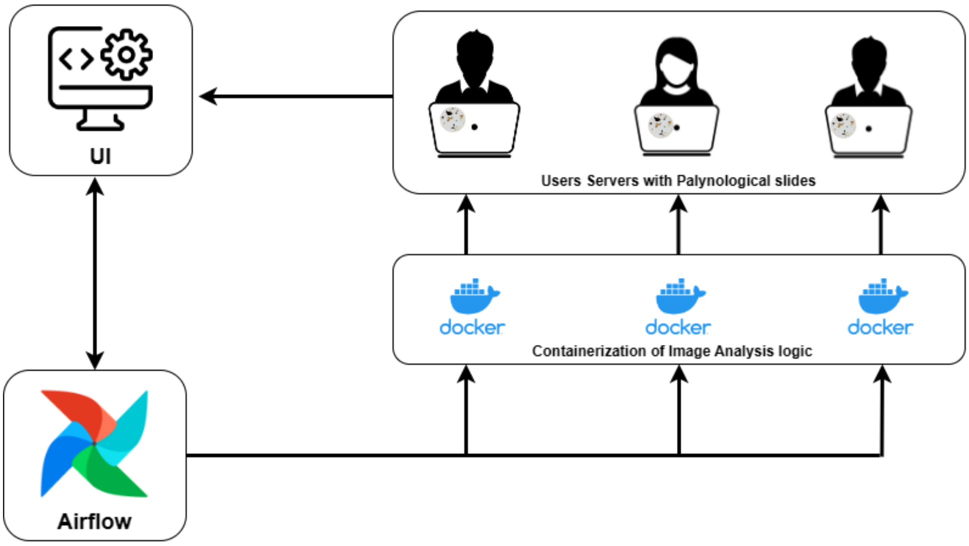

UI
Users Servers with Palynological slides
docker
docker
docker
Containerization of Image Analysis logic
Airflow

Figure 1.0.3: A schematic of federated computation.

1.1 Objectives

The main objective of this thesis is to implement a federated computational workflow for the analysis of digital palynological slide images. The innovative workflow is adapted to solve the inherent challenges associated with data mobility and expensive computational demands of centralized image processing methods. By strategically moving the computation to the data source, the goal is to maximize local computational resources and minimize the need for large-scale data transfers.

This thesis will develop a customizable workflow with a user-friendly interface. The idea is to allow users tailor the workflow to their needs, such that users can add or remove processes based on their specific data requirements. The flexibility feature simplifies user interaction and ensures the workflow remains adaptable to different analysis scenarios.

Also, the present study aims to train and deploy deep-learning models in a containerized environment leveraging the Docker technology. The models are integrated in the workflow, and users can run processes such as microfossil detection and classification task with the trained models.

1.2 Thesis Structure

A brief outline of the thesis is presented in this section. The thesis comprises preliminary pages, six chapters, a list of references, and the Appendix.

Chapter 1. presents the introduction and objectives the present study aims to achieve.

Chapter 2. presents a literature review of previous studies and discusses concepts and technologies relevant to the research.

Chapter 3. presents materials and data used in the research, including data generation and preprocessing.

Chapter 4. discusses the methodology and tools adopted in this study. This chapter comprises two parts. It presents the federated computational workflow and image processing techniques, with emphasis on deep-learning models and microfossil identification methods.

Chapter 5. presents the results of slide image analysis tasks, the workflow example cases, and discussion.

Chapter 6. presents the conclusion, summarizing findings and challenges encountered while implementing the proposed workflow for palynological slide image processing.

CHAPTER

TWO

THEORY

This chapter presents theories and concepts that form the foundation of this research. The concept of federated computing, its relevance, and its potential application in overcoming existing limitations will be discussed. Object detection and classification techniques also form a crucial part of the discussion, providing insights into current methods and their applicability to digital palynological slide analysis.

The application of Docker and containerization in modern computing practices has gained a wide acceptance. The present study will evaluate the merits of using the Docker technology in the proposed research methodology.

2.1 Digital Palynology

Palynology is the study of microfossils. A subdiscipline of geology, palynology is defined in (Jarzen, 2022) as the study of plant pollen, spores, and certain microscopic plankton organisms in both living fossilized forms.

The field has many important applications in petroleum exploration and production. It aids geoscientists in identifying potential hydrocarbon-bearing formations and evaluating the source rocks to understand the geological history of the area, hence supporting informed decision-making. Other areas where palynology is relevant to the petroleum industry include providing valuable insights for stratigraphic correlation, source rock analysis, biostratigraphy, basin analysis, and paleoenvironmental reconstruction.

Digital palynology represents a significant advancement in paleontological research. It was initially conceived to address the need for more comprehensive sampling and improved resolution in traditional palynology that requires extensive labor for microscopic analysis of plant fossils (Stillman & Flenley, 1995). This interdisciplinary field has gained substantial attention due to its ability to unlock previously inaccessible data in slide images. This enables researchers to gain better insights into past climatic conditions, ecosystems, and evolutionary patterns. The importance of digital palynology is such that it has the potential to transform how paleontological data are processed and interpreted (Holt & Bennett, 2014).

Existing techniques in digital palynology comprise several image analysis methods, from basic slide scanning to more complex procedures involving image segmentation, feature extraction, and classification (Holt & Bennett, 2014). While considerable progress has been achieved in this area, the existing techniques have some limitations, especially when handling large and complex slide images. The techniques have made it possible to conduct automated pollen identification using databases of classified spores and pollen (Holt & Bennett, 2014).
Subsequent sections will present challenges associated with traditional techniques and potential solutions, and hence the relevance of the novel methodology proposed in this thesis.

2.1.1 Digital palynology at NPD

The Norwegian Petroleum Directorate (NPD) Avatara-p project signifies significant progress in the digitalization of palynology. The initiative focuses on digitizing microfossils composed of acid-resistant carbon-hydrogen-oxygen biopolymers that are abundant and often better preserved than their contemporary counterparts in the seabed (R. Williams, 2023). Figure 2.1.1 presents a sample of a palynological slide before digitalization.

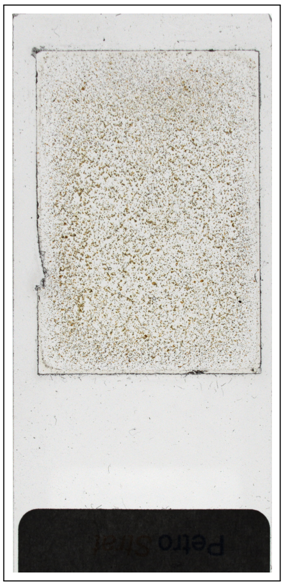


Figure 2.1.1: A sample of paleontological slide.

Palynology plays an important role in petroleum exploration by providing an analytical tool for economic risk reduction through an improved understanding of paleoenvironments, paleogeography, and basin history. The NPD has sought to transform the painstaking and slow process of palynology into a more efficient and automated procedure through digitalization (R. Williams, 2023).

The Avatara-p project employed a high-resolution slide scanner (3DHistech P1000) to digitalize palynological slides. The project has produced approximately forty thousand digital palynology slides from roughly 300 exploration and development wells. This growing dataset, currently amounting to 57 terabytes, is accessible through the Diskos platform (R. Williams, 2023).

Figure 2.1.2 shows an annotated and digitalized palynological slide image from the NPD ((The NPD, 2023).

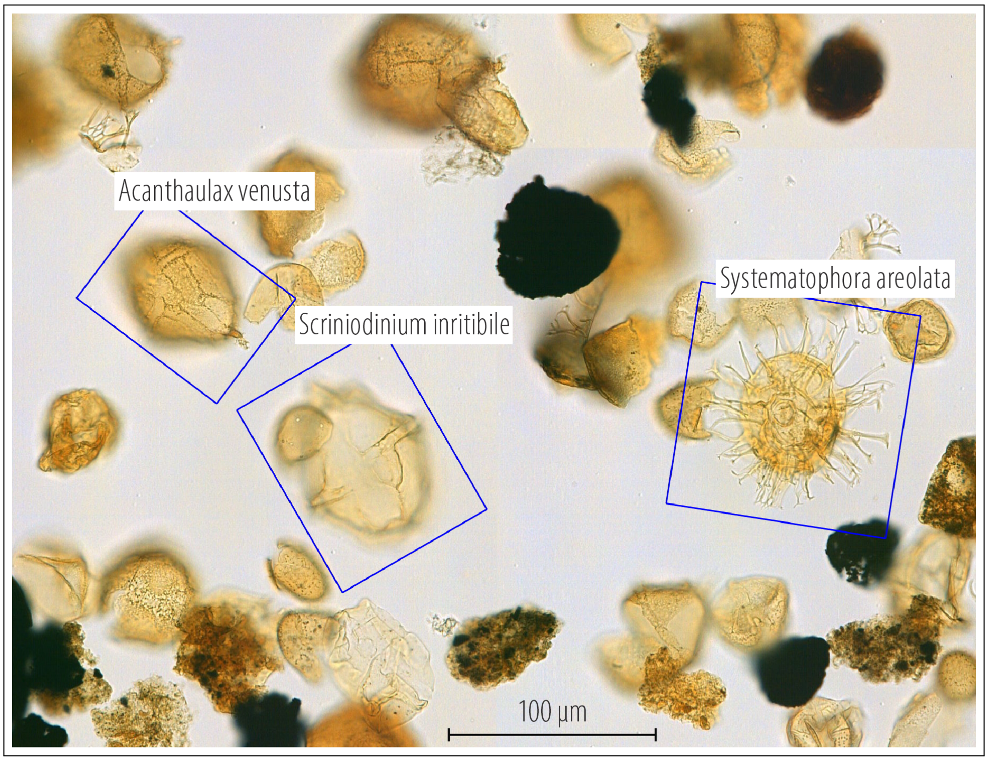


Figure 2.1.2: A digitalized and annotated paleontological slide image

The main goal of the Avatara-p project is to deploy machine learning applications to process the vast data archive of digital slides. Such a step can help gain new insights into Norway's geological history and further simplify the analysis of microplankton, pollen, and spores (R. Williams, 2023). It opens an exciting opportunity in digital palynology, leveraging advanced image processing technologies to enhance understanding of paleontological phenomena.

2.2 Large-Scale Image Processing

Large-scale image processing in digital palynology is a challenging task due Teams to the large size of palynological slides that can exceed 3GB (Wright et al., 2012). This creates significant computational demands and necessitates robust solutions for storage, transmission, and processing (Wright et al., 2012). Existing desktop-based solutions rely on local computing capabilities, limiting scalability needs and hence require the data to be stored locally (Wright et al., 2012). The use of digital slide imaging is a future solution in the field of anatomic pathology, where glass slides will be replaced by scanned images ((Wright et al., 2012); (Al-Janabi et al., 2011)).

Palynology, encompassing the study of pollen and non-pollen palynomorphs, plays an important role in academic and industrial research for correlation and interpretation of subsurface geology on local and regional scales (Stefanowicz, 2023). Non-pollen palynomorphs are often found in palynology slides (Shumilovskikh et al., 2021).

Common challenges in large-scale image processing, such as data handling and computational efficiency, are well-documented in the literature ((Z. Li et al., 2018); (N. Li et al., 2013); (W. Li et al., 2023); (Silva et al., 2022); (Xu et al., 2017)). Data handling involves issues related to the storage, retrieval, and transmission of large image files ((Z. Li et al., 2018); (Silva et al., 2022)). It is important to have storage solutions that can effectively handle the large data volumes (Silva et al., 2022). Computational efficiency refers to the time and resource-intensive nature of analyzing extensive image data (Z. Li et al., 2018).

Figure 2.2.1 shows how long will take to transfer 3GB of data at the world's average internet speed (BroadbandSearch, Accessed: July 12, 2023) using data transfer calculator (Expedient, Accessed: July 12, 2023).
Traditional image analysis algorithms often prove inadequate for large-scale image processing due to their inability to efficiently process and analyze such vast amounts of data ((Z. Li et al., 2018); (Xu et al., 2017)). Techniques such as Fourier domain structured low-rank matrix recovery (Ongie & Jacob, 2017) and parallel implementation of algorithms (N. Li et al., 2013) have been proposed as alternatives to traditional methods. By contrast, the use of high-performance computing (W. Li et al., 2023) and machine learning-based frameworks (Silva et al., 2022)) have shown promising results in addressing the challenges. These challenges are relevant in medical imaging ((Z. Li et al., 2018); (W. Li et al., 2023); (Xu et al., 2017)) and fields like cryo-electron microscopy (Cossio et al., 2017) and digital pathology (W. Li et al., 2023).

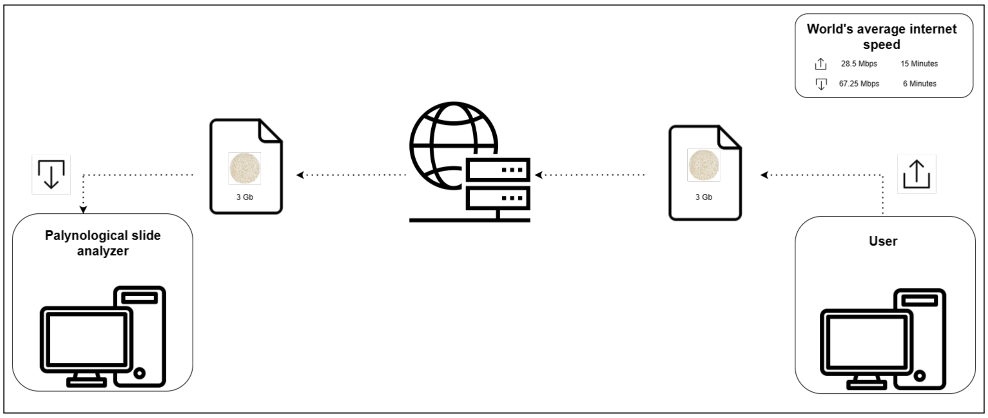


Figure 2.2.1: Transfer time in an ordinary computing transfer process.

Existing solutions to the challenges in large-scale image processing comprise a wide range of strategies including hardware advancements and software solutions ((Abudayyeh et al., 2004); (Siegel et al., 1992)). Hardware advancements involve using powerful servers and high-performance computing clusters to handle the computational demands (Abudayyeh et al., 2004). Software solutions include the implementation of parallel processing algorithms and efficient data structures for efficient data handling and analysis (Sherman et al., 2019).

Despite the solutions, limitations persist in addressing the challenges of large-scale image processing ((Abudayyeh et al., 2004); (Siegel et al., 1992)). These challenges require multidisciplinary efforts spanning software, hardware, and applications (Siegel et al., 1992). Further developments in software architecture and data structures aim to enhance efficiency and reduce memory overhead, enabling the analysis of larger datasets ((Sherman et al., 2019); (Sherman et al., 2020)). Also,studies have explored software and hardware models to address these challenges (Saito et al., 2012).

The storage and transfer of large image files continue to be time and resource-intensive processes, and image analysis algorithms still struggle with the complexity and diversity of data presented in palynological slides ((Y. Zhang et al., 2012); (George et al., 2012); (Hunt et al., 2002)). Efforts to improve the efficiency of storage and transfer methods and enhance image analysis algorithms are ongoing, recognizing the need to address these challenges (Y. Zhang et al., 2012). The complexity and diversity of data in palynological slides require advanced algorithms capable of effectively processing and analyzing such vast amounts of information ((Górka et al., 2014); (Hunt et al., 2002)). Research in various domains such radiation lung injury (Y. Zhang et al., 2012), gene analysis (Chandriani & Ganem, 2010), and paleoenvironmental studies (Hunt et al., 2002) contributes to the understanding of difficulties in storing, transferring, and analyzing large image files.

Addressing these limitations is the main focus of this thesis. By moving the computational logic closer to the data location and leveraging federated computing principles, the goal solve the prevailing challenges in large-scale image processing. The following sections will discuss proposed solutions and their implementation in the research methodology.

2.3 Federated Computing

Federated computing is an emerging computational paradigm in the era of big data. The approach presents a unique solution to address the challenges encountered in extensive scale image processing ((Ye et al., 2020);(Mohammadi & Thornburg, 2020)). At the core of federated computing is an approach that enables distributed computing nodes to collaboratively process data while ensuring data privacy and locality (Ye et al., 2020). This decentralized approach allows for efficient and secure processing of large-scale image data sets without centralizing the data (Ye et al., 2020). By leveraging intelligent sensors and advanced computing capabilities, federated computing supports the processing and analysis of data in distributed environments (Mohammadi & Thornburg, 2020). It offers a promising solution to overcome the limitations of centralized processing in large-scale image processing tasks (Ye et al., 2020).

In traditional centralized computing models, transferring data to a central location or server for processing can be resource-intensive and problematic, especially when dealing with large datasets such as those in digital palynology ((Anveden & Meding, 2007);(Oppenheimer et al., 2005))
Figure 2.3.1 illustrates the centralized computing concept.

Compared to the centralized approach, federated computing introduces a paradigm shift by bringing the computation to the data location instead of moving the data (Oppenheimer et al., 2005). By executing processing tasks at the data source itself, federated computing mitigates the need for extensive data transfer and maximizes local computational resources ((Ali et al., 2018); (Alazzam et al., 2022)).
Figure **??** illustrates the federated computing model. This approach has gained acceptance in various domains such including skin exposure assessment (Anveden & Meding, 2007), wide-area platforms (Oppenheimer et al., 2005), air traffic control communication analysis (I. Williams et al., 2002), and large-scale video stream analytics (Ali et al.,

2018). It offers a promising solution for solving data movement challenges and maximizing computational efficiency in large-scale image processing tasks (Alazzam et al., 2022).

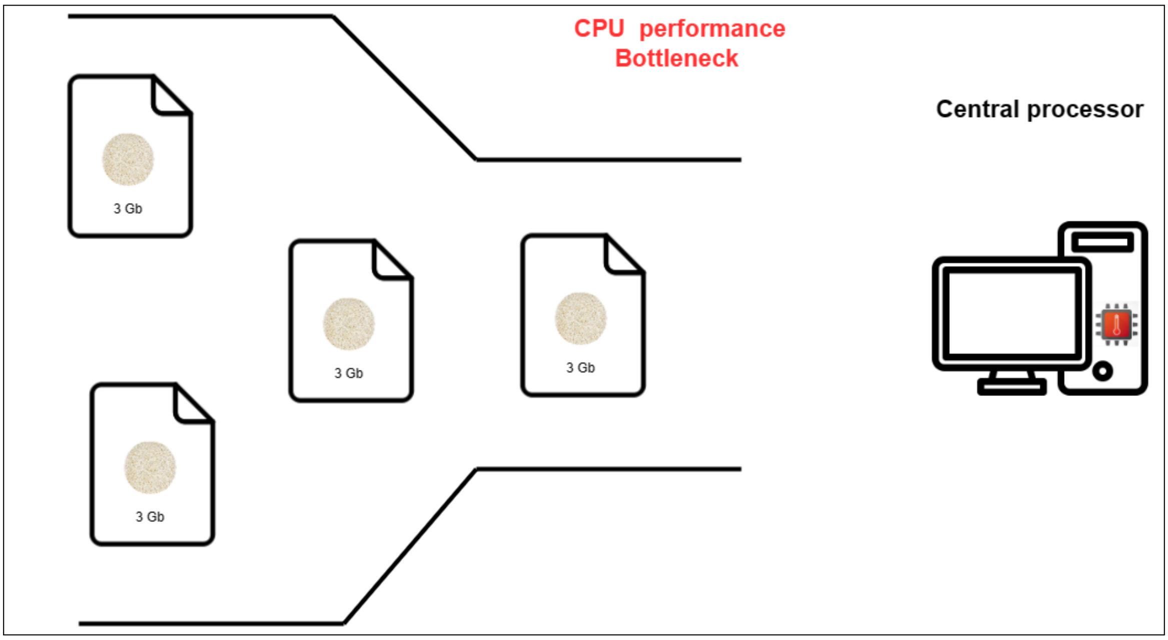


Figure 2.3.1: A centralized computing model.

Federated computing addresses issues related to data transfer and storage and enables harnessing of collective computational power from distributed nodes in a network ((Lee et al., 2012); (Xiaofeng et al., 2022a); (Bagnasco et al., 2015)). By enabling parallel processing across multiple nodes, federated computing significantly reduces the time required for analyzing large datasets ((Lee et al., 2012);(Bagnasco et al., 2015); (Shuvo et al., 2023)). This approach has been applied in estimating long-term exposure to ambient PM2.5 concentrations (Lee et al., 2012), sparse federated learning with hierarchical personalized models (Xiaofeng et al., 2022b), interoperating cloud-based virtual farms (Bagnasco et al., 2015), civil aviation passenger value analysis (S. Chen, 2023), and efficient acceleration of deep learning inference on resource-constrained edge devices (Shuvo et al., 2023).

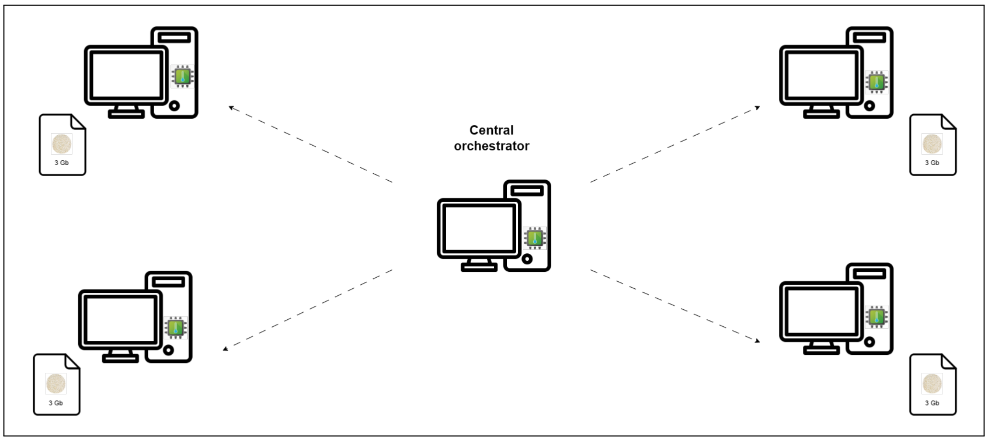


Figure 2.3.2: A federated computing model.

However, federated computing has challenges in handling heterogeneity in system architectures, ensuring synchronous computation, and managing network latency among the distributed nodes ((Brendan et al., 2016); (Cao et al., 2021); (Kimovski et al., 2018)). The challenges are attributed to the decentralized nature of federated computing and the need to coordinate and synchronize computations across multiple nodes ((Brendan et al., 2016); (Cao et al., 2021)).

To address the challenges, researchers have proposed solutions such as communication-efficient learning algorithms(Brendan et al., 2016), the use of Direct Acyclic Graph (DAG)-based blockchain for device asynchrony and anomaly detection (Cao et al., 2021), and the development of ultra-scale computing architectures for efficient deployment and management (Kimovski et al., 2018). The advancements aim to solve the challenges and harness the potential of federated computing in analyzing digital palynological slides.

The federated computational workflow proposed in this thesis will demonstrate how the challenges can be solved to leverage the benefits of federated computing in the analysis of digital palynological slide images.

2.4 Object Detection and Classification

Object detection and classification are deep-learning tools for analyzing whole slide images in digital palynology, where identifying and categorizing microfossils on slide images is of the main focus. This section reviews current algorithms and techniques such the watershed segmentation algorithm and the use of Convolutional Neural Networks (CNNs) in classifying microfossils.

The watershed algorithm (OpenCV, 2023), a classical image segmentation technique, is an effective tool for object detection. It treats the grayscale image as a topographical relief, with bright areas considered high regions and dark areas as low regions. This 'flooding' mechanism enables the algorithm to separate distinct objects in an image, even if they are touching or slightly overlapping. In digital palynology, this could mean the efficient separation of adjacent or closely located microfossils on a slide.

Compared with watershed segmentation, Region-based CNN (RCNN) algorithm offers a more advanced solution for object detection. The algorithm involves a two-step process. First, the algorithm extract bounding boxes and load masks of the objects found in an image. Then it utilizes CNN algorithm networks to identify the objects within these boxes. Faster RCNN and Mask RCNN (Gad, 2021) are two commonly used networks in this category. The adaptability and accuracy of RCNNs make them a useful tool for identifying diverse microfossils found in palynological slides.

The thesis work (Nesse, 2020) presented an extensive comparative classification study of dinoflagellates on palynological slide images with eight pre-trained CNN networks, leveraging transfer learning. The results were satisfactory.
However, the present research focuses on integrating microfossil detection and classification tasks and the proposed federated workflow for analyzing palynological slide images.

It is important to mention that while existing techniques have been successfully applied in various fields, their application in digital palynology presents diverse challenges. The complexity and diversity of palynological data and the large size of slide images require

careful calibration and optimization of the algorithms.
The methodology proposed in this thesis will discuss how the existing techniques can be adapted to solves challenges encountered in handling large digital palynological data.

2.5 Docker and Containerization

Docker and containerization have emerged as essential tools in modern computing, offering efficiency, portability, and scalability in software development and deployment ((Wu et al., 2020);(Kim et al., 2022); (Venugopal, 2017);(L. Chen et al., 2018); (Robidas & Legault, 2022)). Containers encapsulate applications into discrete units, bundling together all the necessary elements such as libraries, dependencies, and environment variables required for the application to run reliably and consistently across different computing environments ((Wu et al., 2020);(Venugopal, 2017);(Robidas & Legault, 2022)). Figure 2.5.1 illustrate an app deployment with Docker, popular containerization tool.

Docker enables packaging of software packages into containers, providing a standardized, self-contained unit that can be used for software development and running applications on any system (Wu et al., 2020). Containerization techniques such Docker and Kubernetes are employed for computation and storage resource allocation. The containerization tools offer advantages such as lower system overhead and shorter launch time compared to virtual machine technology (L. Chen et al., 2018). Containerization has also been widely used in various domains such cheminformatics, bioinformatics, and computational material science for the distribution of applications (Robidas & Legault, 2022).

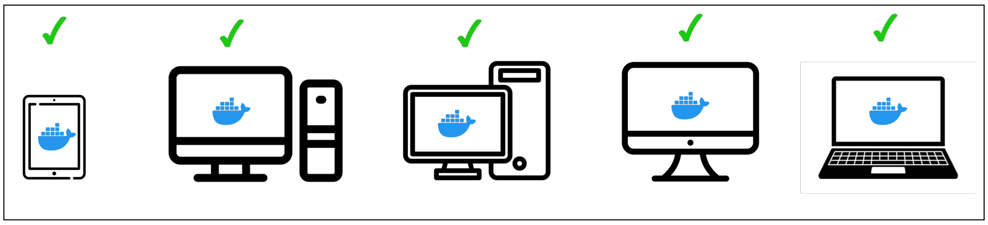


Figure 2.5.1: An application containerized and deployed with Docker

Docker offers several advantages in large-scale image processing and analysis in digital palynology. Docker containers guarantee the reproducibility of computational environments across various computing nodes (Boettiger, 2015). This is crucial in a federated computing framework where different nodes may have heterogeneous system architectures. Docker containers encapsulate all the necessary dependencies and configurations, ensuring consistent deployment and execution of applications across different environments (Boettiger, 2015).
Docker has been widely adopted in bioinformatics, demonstrating its effectiveness in providing reproducible and portable computational environments (Lindenbaum & Redon, 2017). The Docker application in bioinformatics promotes the development and sharing of tools and workflows, enabling researchers to analyze large-scale image data efficiently.

Docker containers have other interesting features that offer several advantages for large-scale image processing. The containers are lightweight and standalone, eliminating the "it works on my machine" problem ((Z. Li et al., 2017);(Alston & Rick, 2020)). This

means that the computational logic packaged into a Docker container for image processing can be reliably executed across all nodes, independent of their underlying system configurations (Z. Li et al., 2017).
Also, Docker containers provide a portable environment that contains the entire computing environment, including software, dependencies, libraries, and configuration files, bundled into one package (Alston & Rick, 2020). This ensures the reproducibility of the analysis and mitigates compatibility issues between different operating systems or program versions (Alston & Rick, 2020). The lightweight nature of Docker containers, compared to hypervisor-based virtualization, allows for efficient resource utilization and minimizes performance overhead ((Z. Li et al., 2018); (Hanussek et al., 2021)). These advantages make Docker containers valuable for large-scale image analysis, enabling consistent and reproducible execution across distributed computing nodes.

Resource isolation can be implemented with the Docker technology. This allows each Docker container to have its resource limits, enabling efficient utilization of system resources ((Keskin & Ince, 2021); (Gowri, 2019); (Al-Dhuraibi et al., 2017)). This feature is required for processing large datasets that will potentially consume significant system resources (Keskin & Ince, 2021). The ability to set container resource limits ensures that each container operates within its allocated resources and prevents resource contention, optimizing resource utilization (Al-Dhuraibi et al., 2017). This capability is beneficial in scenarios where multiple containers are running concurrently such as in cloud-based applications or high-performance computing environments ((Gowri, 2019); (Sauvanaud et al., 2020)). By providing resource isolation, Docker enhances the efficiency of large-scale image processing tasks and ensure that system resources are optimized.

The present research methodology uses Docker to build a containerized environment that hosts the object detection algorithm and classification models
Figure 2.5.2 illustrates the proposed Docker container. Each node in the federated system runs the Docker container, processing assigned palynological slide images and contributing to the overall task. The approach brings the computation to the data, reducing the need for data transfer and making efficient use of local computational resources ((Haque et al., 2020);(Massaoudi et al., 2021); (Tommasini, 2021)). By leveraging Docker containers, the computational logic can be reliably executed across different nodes, ensuring consistency and reproducibility in the analysis (Haque et al., 2020).

Working with Docker containers presents challenges such as container management and orchestration, networking between containers, and ensuring data persistence ((Haque et al., 2020); (Yamanaka et al., 2022);(Bedhief et al., 2022)).
The present research leverages the Docker technology to implement the proposed computational workflow for analyzing digital palynological slides, considering the challenges presented in (Haque et al., 2020). Networking between containers can be achieved through proper configuration and container networking features (Yamanaka et al., 2022). Ensuring data persistence can be achieved through the use of persistent volumes or other storage solutions (Haque et al., 2020).

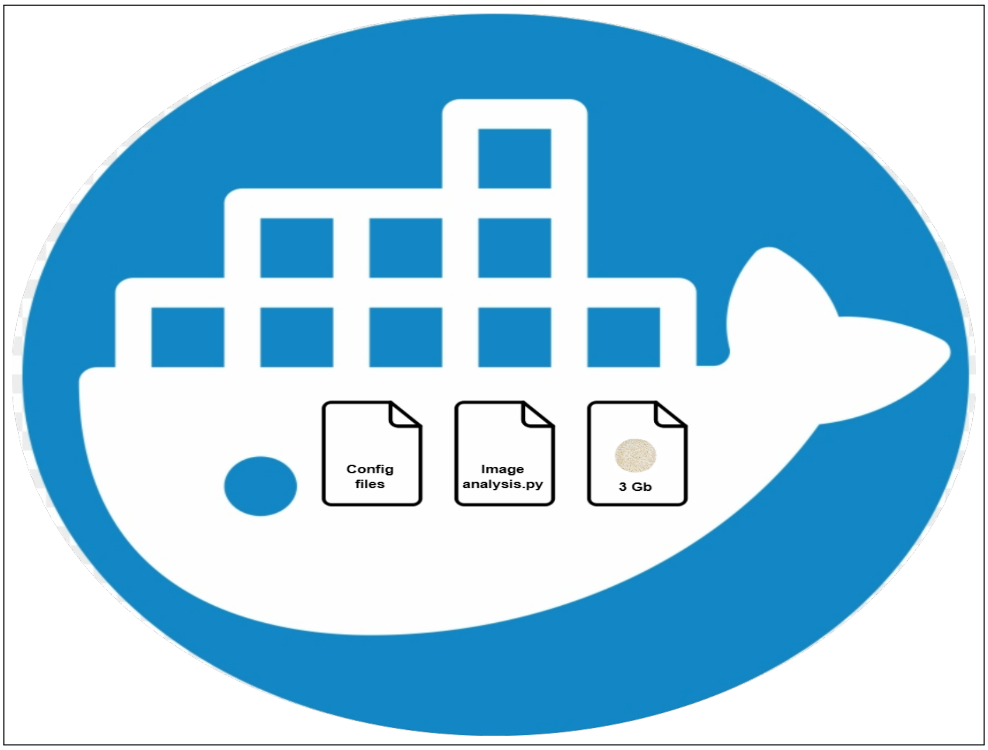


Figure 2.5.2: The proposed Docker container for palynological slide processing.

2.6 Gaps in Current Research

Existing research in digital palynology has drawbacks. One of the major concerns is the handling and processing of large image datasets. Traditional methods often rely on centralized models that require large data transfer across networks. The inefficient and resource-intensive approach poses scalability issues as data volumes increase.

While previous studies have explored object detection and classification techniques, their application in digital palynology remains relatively unexplored. Considering the complexity and diversity of palynological data, adapting and optimizing the microfossil identification techniques for a specific dataset is a challenging task.
Also, there are a few research on the application of federated computing in this domain, and applying containerized environments such as Docker to reproducibility issues and improve efficiency in distributed computations remains underexplored.

This thesis aims to address these gaps. To solve the challenges, the Docker technology is integrated in the federated computational workflow for efficient and reliable processing of digital palynological slide images. The proposed workflow implemented in a containerized reproducible environment provides a robust solution to challenges associated with large data transfers and scalability.

CHAPTER

THREE

MATERIALS AND DATA

The research materials and data used in the present study are presented in this chapter. Also, data generation and preprocessing procedures are discussed.

3.1 Digital Slide Preparations

The NPD provided the digital palynological slide images used for objection detection and classification tasks. The directorate reported it has produced more than thirty thousand digital palynological slides from 284 released wellbores drilled on the Norwegian Continental Shelf (NCS). This huge dataset of 57 terabytes is available in Diskos, a decentralized archive solution for storing and sharing seismic, wellbore, and production data(The NPD, 2023).

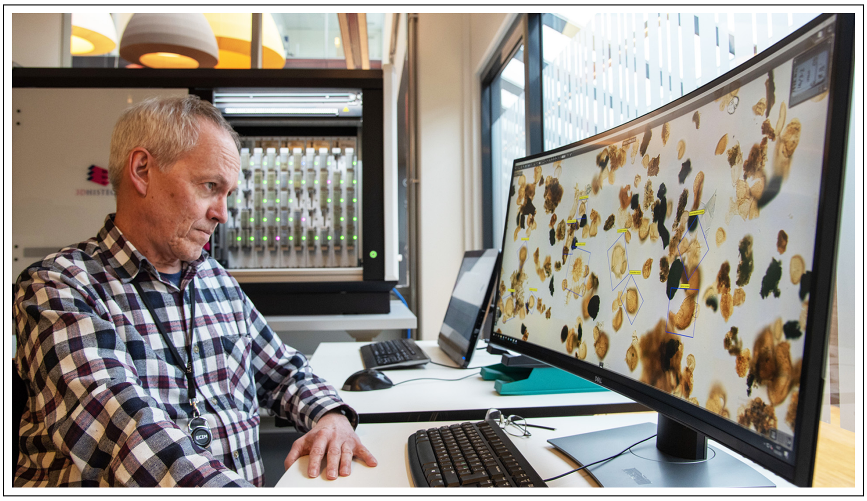


Figure 3.1.1: Palynological slide digitalization and annotation at NPD.
Geologist Robert Williams identifies and annotates dinoflagellates on a digital palynological slide. The machine behind him is a slide scanner used for scanning the NPD huge database of palynological slides derived from microplankton, pollen, and spores from wellbores drilled on the NCS. Picture: Arne Bjørøen (The NPD, 2023).

Experts at the NPD produce palynological slides with core samples from wellbores on the NCS. The wellbore depths from which the core samples were drilled are recorded and used for labeling palynological slides.
To produce a palynological slide, a small sample from a core is grounded and dissolved in an acid such as hydrochloric and hydrofluoric acids. The acid used depends on the material composition of the source formation. The sample is then washed with tap water and a few drops are placed on a microscopic slide and left overnight to dry. The last step is slide labeling. A coverslip with resin is placed on the sample, and the slide is labeled to indicate the source wellbore and procedure used in producing the slide.

Further palynological slide preparations involve scanning the slide samples with Pannoramic 1000 from 3DHistech and annotating selected digital slides with CaseViewer from the same company. 3DHistech (3DHistech, 2020) is a digital pathology company based in Budapest, Hungary.
Figure 3.1.1 shows Geologist Robert Williams identifying and annotating dinoflagellates on a digital palynological slide.
Figure 3.1.2 shows microfossils on a digital palynological slide with the wellbore information. The label indicates the core sample was drilled at a depth of 3069.3m from a wellbore named 2/4-C-11.

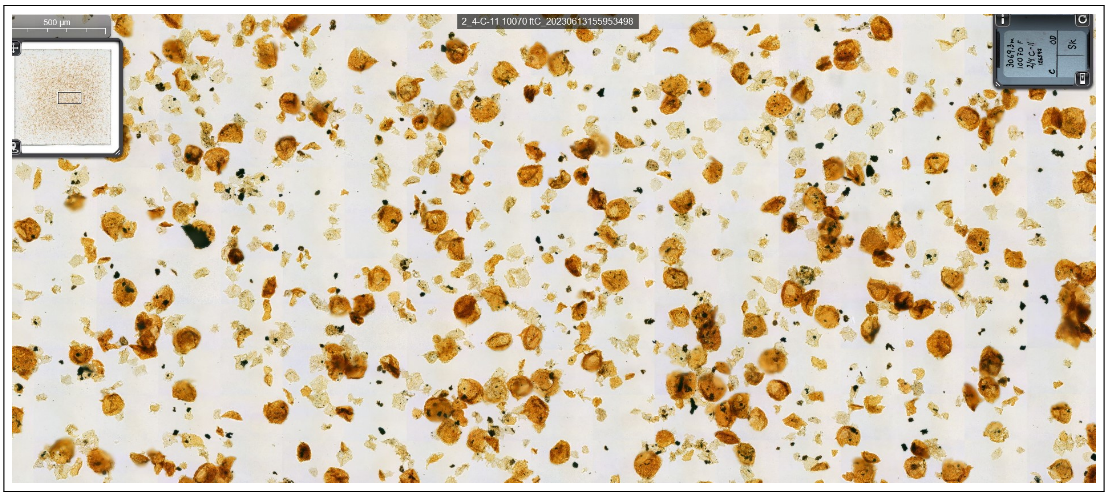


Figure 3.1.2: A digital palynological slide with the wellbore information

A research visit to the NPD and a detailed discussion with the expert geologist gave valuable insights into palynological analysis at the directorate. In collaboration with medical pathologists from Finland, leveraging improved imaging technologies, the NPD has created a huge palynological knowledge base. The knowledge database has helped increase understanding of the geological history of the NCS with the vast processing of microplankton, pollen, and spores.

3.2 Materials

Sixteen digital palynological slides from five wells on the NCS were analyzed in the preliminary study. The source rocks were of the Danian and Kimmeridgian geological ages and yielded varying distributions of microfossils depending on depositional depths.

Table 3.2.1: Digital palynological slides for image analysis and classification.

Slide Name	Well	Depth	Geological Age
2_4-C-11 10052.7 ftC.mrxs	2/4-C-11	10052.7 ft	Danian
2_4-C-11 10070 ftC.mrxs	2/4-C-11	10070 ft	Danian
*DigitalSlide*_*C*1M_4*S*_1.mrxs	2/4-C-11	10076.9 ft	Danian
*DigitalSlide*_C1M_3*S*_1.mrxs	2/4-C-11	10076.9 ft	Danian
*DigitalSlide*_C1M_3*S*_1.mrxs	2/4-C-11	10072.7 ft	Danian
25_11-5 1731.4 mC.mrxs	25/11-5	1731.4 m	Danian
25_11-5 1731.7 mC.mrxs	25/11-5	1731.7 m	Danian
25_11-5 1732 mC.mrxs	25/11-5	1732 m	Danian
2_7-14 10658 ftC.mrxs	2/7-14	10658 ft	Danian
16_3-2 1998.80 mC.mrxs	16/3-2	1998.80 m	Early Kimmeridgian
16_3-2 2000.3 mC.mrxs	16/3-2	2000.3 m	Early Kimmeridgian
1_6-6 ST2 4876 mDC.mrxs	1/6-6	4876 m	N/A
1_6-6 ST2 4990 mDC.mrxs	1/6-6	4990 m	N/A
1_6-6 ST2 5107 mDC.mrxs	1/6-6	5107 m	N/A
1_6-6 ST2 5224 mDC.mrxs	1/6-6	5524 m	N/A
1_6-6 ST2 5323 mDC.mrxs	1/6-6	5323 m	N/A

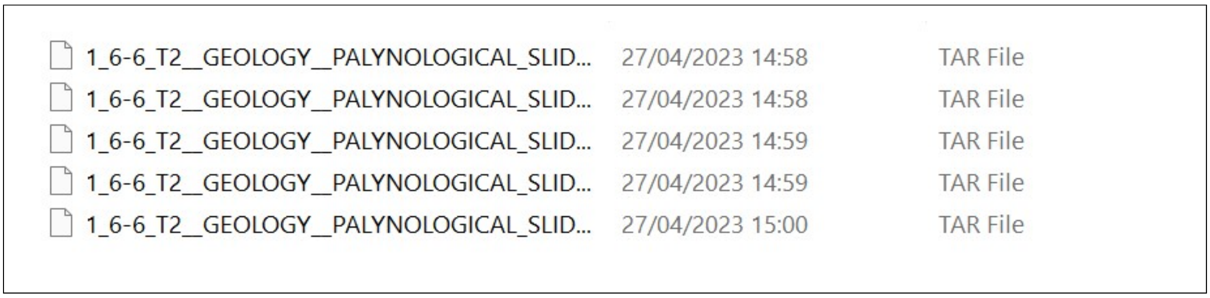
Table 3.2.1 presents a list of digital palynological slides obtained from the NPD. However, only seven selected slides that produced desired results provided datasets used for further analyses. Factors considered in the slide selection include image quality and population sizes of identified dinoflagellates. All the digital palynological slides from Well 2\4-C-11 produced good tiled images and large populations of dinoflagellates and were used for data generation. Also, the slide derived from Well 2\7-14 and the first slide from Well 16\3-2 were selected. The slide images derived from Well 1\6-6 produced the poorest quality tiles filled will dark debris, with insignificant numbers of dinoflagellates.

3.3 Data Generation

The datasets used for segmentation and classification tasks were derived from the seven selected digital palynological slides. The geologist at the NPD had annotated six slide images, while the seventh slide has no annotations.

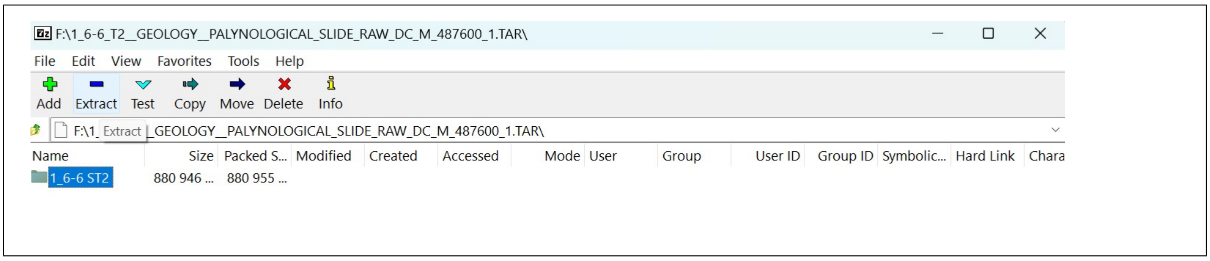
Each slide image is saved as an MRXS file, with a folder containing the slide metadata and a list of DAT files. Annotated slides have annotation details saved as XML files, in addition to metadata folders and MRXS files.

Figure 3.3.1 presents palynological slide data from Well 1/6-6 compressed into TAR files, while Figure 3.3.2 demonstrates how to extract slide data from the TAR file shown on the window of 7-Zip File Manager.



1_6-6_T2__GEOLOGY__PALYNOLOGICAL_SLID...	27/04/2023 14:58	TAR File
1_6-6_T2__GEOLOGY__PALYNOLOGICAL_SLID...	27/04/2023 14:58	TAR File
1_6-6_T2__GEOLOGY__PALYNOLOGICAL_SLID...	27/04/2023 14:59	TAR File
1_6-6_T2__GEOLOGY__PALYNOLOGICAL_SLID...	27/04/2023 14:59	TAR File
1_6-6_T2__GEOLOGY__PALYNOLOGICAL_SLID...	27/04/2023 15:00	TAR File

Figure 3.3.1: A list of slide data compressed as TAR files.



F:\1_6-6_T2__GEOLOGY__PALYNOLOGICAL_SLIDE_RAW_DC_M_487600_1.TAR\

F:\1_ Extract _GEOLOGY__PALYNOLOGICAL_SLIDE_RAW_DC_M_487600_1.TAR\

Name	Size	Packed S...	Modified	Created	Accessed	Mode	User	Group	User ID	Group ID	Symbolic...	Hard Link	Chara
1_6-6 ST2	880 946 ...	880 955 ...											

Figure 3.3.2: Extracting slide data from a TAR file with 7-Zip File Manager.

3.3.1 Slide object and visualization

The first step in data generation is the creation of a slide image object with the OpenSlide class. OpenSlide(PyPI, 2022) is a Python library used for image analysis for reading and manipulating whole slide image files. Compared with digital pathological slides used in medical research, the palynological slides analyzed are high-resolution images.
The file named **process_palyslides.ipynb** presents an algorithm for reading and manipulating slide image files. The Python code is given in the Appendix.
The slide image processing procedure adopted in this thesis followed similar steps presented in (DigitalSreeni, 2022).

To read a slide file, the MRXS file path is passed as an input argument to the OpenSlide constructor to read and load the slide image. The slide variable is instantiated as an OpenSlide object that can be used to retrieve information about the slide image and perform further analysis. The slide information that can be extracted from the slide object includes pixel data, slide metadata, dimensions, different levels of the slide image, and objective power.
Objective power refers to the magnification power of an objective lens used in capturing the slide image. The selected slide images have an objective power of 40, indicating the specimen images observed through an objective lens will appear 40 times larger than when viewed with human eyes.

3.3.2 Tiled image generation

The slide images are too larger and hence must be broken down into smaller images with desired dimensions. For example, an image size of 256 x 256 pixels is more acceptable.
Figure 3.3.3 presents a simplified palynological workflow for reading slide images and generating tiled images for deep learning tasks.

The algorithm for extracting tiled images from a palynological slide is also implemented in the code **process_palyslides.ipynb** presented in the Appendix.

The **DeepZoomGenerator** function is used to generate smaller images called *tiles* from the slide object created with the OpenSlide class. The generator accepts desired tile size and slide object as arguments. Other parameters specified are overlap and limit bounds parameters. In this case, the overlap parameter is set to 0, indicating no overlap between neighboring tiles, while the limit bounds parameter is set to FALSE. This allows tile generation to exceed the bounds of the slide image.

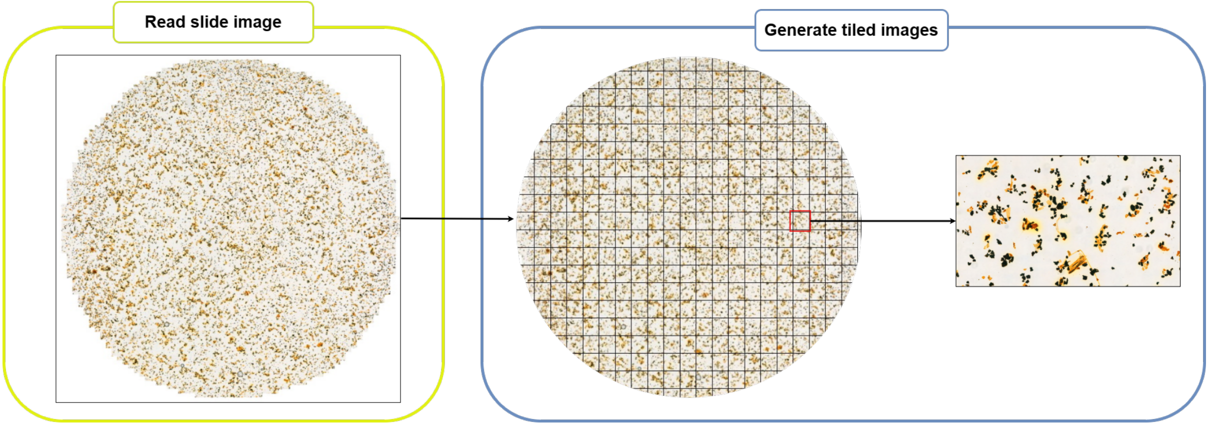


Figure 3.3.3: A simplified workflow for reading slide images and tile generation.

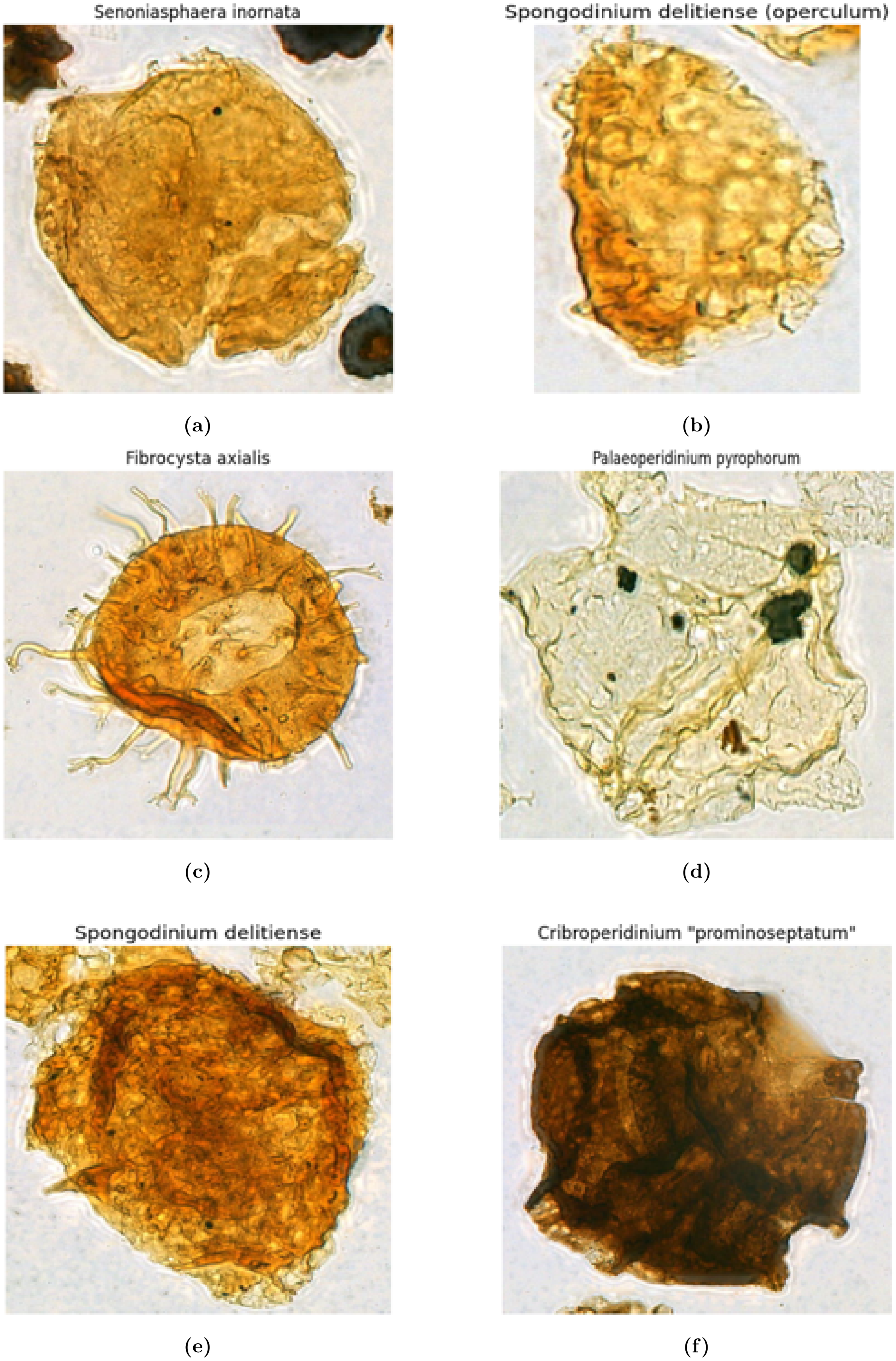
The **get_tile** method is used to extract tiles at specified coordinates and zoom levels. There are 20 zoom levels in a tile object derived from each slide image. The first zoom level has an index of 0 and the last has an index of 19. The number of tiled images generated increases with the zoom level, with Level 20 having the highest number of tiles. The choice of zoom level varies with image analysis workflows.

3.3.2.1 Data cleaning

The desired sizes of tiled images considered in this study are 256 and 512. However, tiles appearing on the bounds of a slide image may not have the desired size. Also, blank and poor-quality tiles are generated with tiles having desired microfossil images.

To generate clean tiles that passed specified thresholds, a deep learning network YOLOv4 Tiny is integrated with the tile generation algorithm. YoLOv4 Tiny is a variant of the YOLOv4 algorithm (Bochkovskiy et al., 2020) optimized for faster detection of objects in videos and images. The configuration and weights files of the YOlOv4 Tiny network are downloaded and saved in the required directory.
The setup ensures tiles generated have good quality and undesirable tiles are discarded.

3.3.3 Labeled image extraction


Senoniasphaera inornata
Spongodinium delitiense (operculum)
Fibrocysta axialis
Palaeoperidinium pyrophorum
Spongodinium delitiense
Cribroperidinium "prominoseptatum"

(a)

(b)

(c)

(d)

(e)

(f)

Figure 3.3.4: Selected classes of dinoflagellates.

Annotated images are required for training deep learning algorithms used for object identification and classification tasks. To extract labeled images from annotated slide images, the MRXS file and the corresponding annotation XML files are required.
Labeled image extraction is implemented in the algorithm **classify_microfossils.ipynb** presented in the Appendix.
Figure 3.3.4 presents selected classes of dinoflagellates from the six annotated slide images.

For a given pair of MRXS and XML files containing the annotation details of dinoflagellates present in a slide image, the algorithm iterates over the files and extracts "label", "image", and "annotation" for each microfossil and stores the data in a dictionary. A list of data stored in dictionaries for all the dinoflagellates extracted from the six annotated slides is converted to a dataframe using Pandas.

3.3.4 Data preprocessing

While only "label" and "image" data are required for training Convolutional Neural Networks (CNNs) considered in the classification workflow, "annotation" data for each labeled image is required for training object identification algorithms such as Mask RCNN and Faster RCNN.

Table 3.3.1: Dinoflagellates and counts

Class	Count
Senoniasphaera inornata	173
Fibrocysta axialis	121
Palaeoperidinium pyrophorum	117
Spongodinium delitiense	65
Cribroperidinium "prominoseptatum"	18
Spongodinium delitiense (operculum)	13
Dingodinium tuberosum	2
Sentusidinium pilosum	2
Systematophora areolata	2
Gonyaulacysta jurassica	2
Acanthaulax venusta	2
Tubotuberella apatela	2
Dingodinium tuberculosum	2
Scriniodinium inritibile	1
Thalassiphora pelagica	1
Sirmiodinium grossii	1
Cribroperidinium "prominoseptatum"	1
Leptodinium mirabile	1
Endoscrinium galeritum reticulatum	1
Danea californica	1
Cribroperidinium sp.	1
Chytroeisphaeridia cerastes	1
Total	530

The labeled data stored on the dataframe are preprocessed to filter out classes with a few

records. In the object detection and classification tasks, only dinoflagellate classes with more than two records are considered. The filtered dataset is then split into training and validation datasets for training deep learning models.
Table 3.3.1 presents the classes and number of dinoflagellates extracted from the annotated slide images before data preprocessing.
Table 3.3.2 presents selected classes of microfossils with more than two records.

Table 3.3.2: Selected classes of dinoflagellates

Class	Count
Senoniasphaera inornata	173
Fibrocysta axialis	121
Palaeoperidinium pyrophorum	117
Spongodinium delitiense	65
Cribroperidinium "prominoseptatum"	18
Spongodinium delitiense (operculum)	13
Total	507

Table 3.3.3 presents the sizes of the classification datasets used for training, validating, and evaluating the performance of the trained model.

Table 3.3.3: Classification datasets

Class	**Dataset**		
	Training	Test	Validation
Senoniasphaera inornata	138	35	9
Fibrocysta axialis	96	25	9
Palaeoperidinium pyrophorum	93	24	9
Spongodinium delitiense	52	13	8
Cribroperidinium "prominoseptatum"	14	4	5
Spongodinium delitiense (operculum)	10	3	6
Total	403	104	46

CHAPTER

FOUR

METHODOLOGY

The focus of the thesis research is the implemented methodology that combines the principles of federated computing, large-scale image processing, object detection, and classification techniques to analyze digital palynological slides. This chapter presents a comprehensive overview of the federated computational workflow's architecture and functional mechanisms with a detailed discussion of the object detection and classification methods. It is important to understand the relevance of the proposed novel computational approach and how the workflow successfully addresses the challenges associated with analyzing large-scale palynological slide images. The chapter is divided into two main sections: the federated computational workflow and the microfossil detection and classification techniques. Each section is further subdivided into several subsections detailing individual components and operations.

4.1 Federated Computational Workflow

The present research implements a novel approach to handling large-scale digital palynological slides.

Figure 1.0.3 presents a federated computational workflow. The workflow allows the computational logic to be moved to the data location as an alternative to moving the data to the computation location. This represents a significant step in overcoming the mobility challenges posed by large-size data files. The following subsections provide a detailed description of the workflow, design considerations that motivated its implementation, and operational mechanisms.

4.1.1 Workflow design

Figure 4.1.1 shows the design of federated computation workflow. The framework is based on a strategic response to two main challenges in handling large-scale digital palynology slides - computational efficiency and data mobility. Because digital palynological slides are large-size data, typically exceeding 3GB, moving them around for processing can be challenging and inefficient. For this reason, the workflow was constructed based on the idea it is more efficient to move the computation to data.

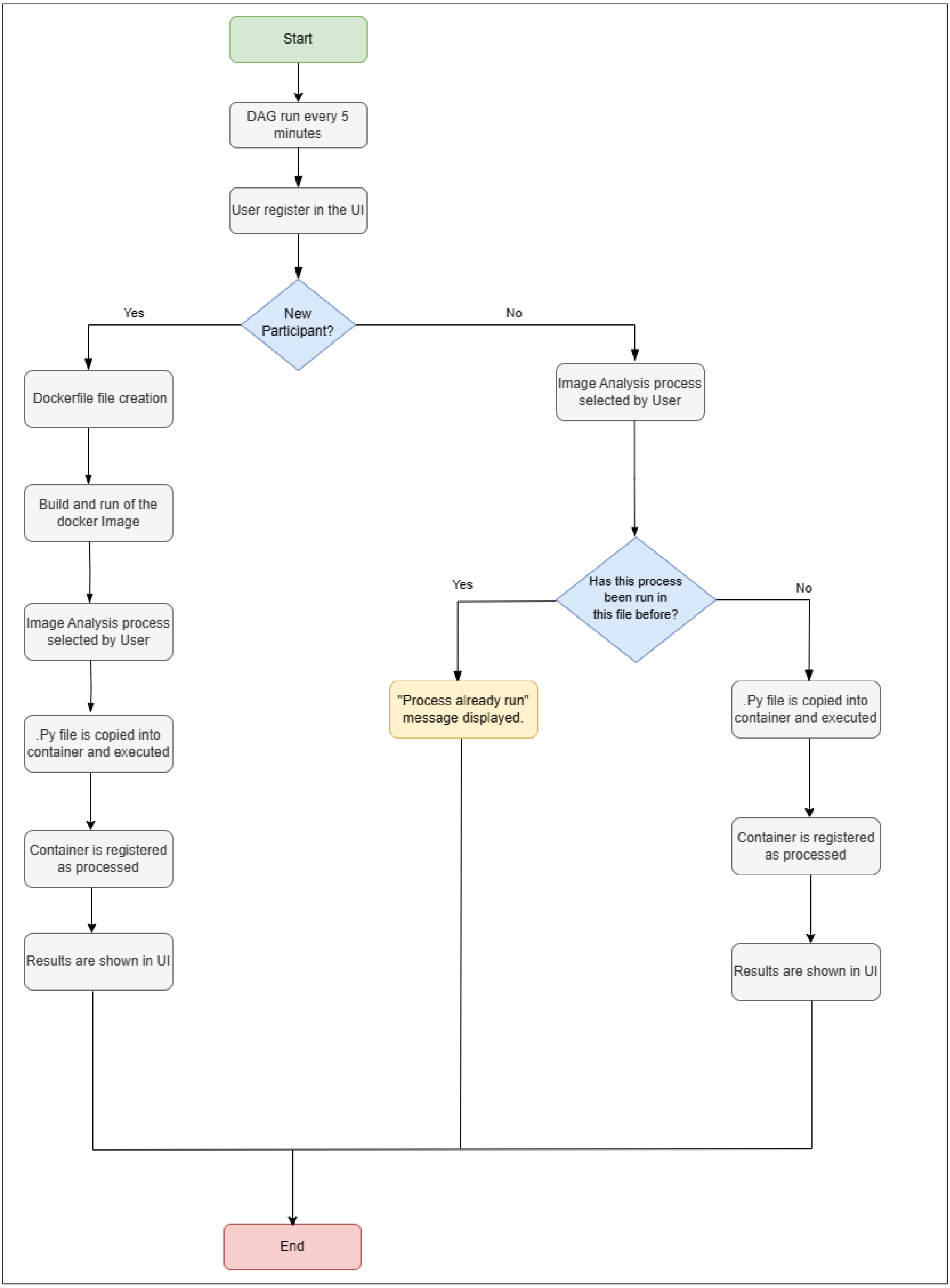
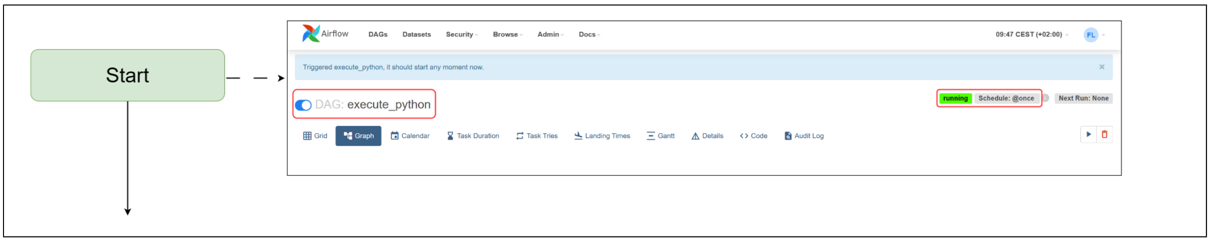

Start
DAG run every 5 minutes
User register in the UI
New Participant?
Yes
No
Dockerfile file creation
Build and run of the docker Image
Image Analysis process selected by User
.Py file is copied into container and executed
Container is registered as processed
Results are shown in UI
Image Analysis process selected by User
Has this process been run in this file before?
Yes
No
"Process already run" message displayed.
.Py file is copied into container and executed
Container is registered as processed
Results are shown in UI
End

Figure 4.1.1: A schematic of the federated computational workflow.

The workflow is activated on Airflow by switching the Boolean on-off switch shown in Figure 4.1.2 when a new participant comes onboard, registering through the specially developed user interface. During the registration, the participant writes an MRXS file path located in the user's system, containing the unprocessed digital palynological slide, and the name the user wants to be identified within the federated computation and selects a process the participant wants to run on the file.


Triggered execute_python, it should start any moment now.

DAG: execute_python

running Schedule: @once Next Run: None

Figure 4.1.2: The starting point of DAG.

Figure 4.1.3 presents the DAG system, designed to automatically scan for new participants at predefined time intervals of 5 minutes. Identifying a new participant activates the next steps of the workflow. These steps involve creating a Docker container on the participant's machine and launching the image processing workflow.

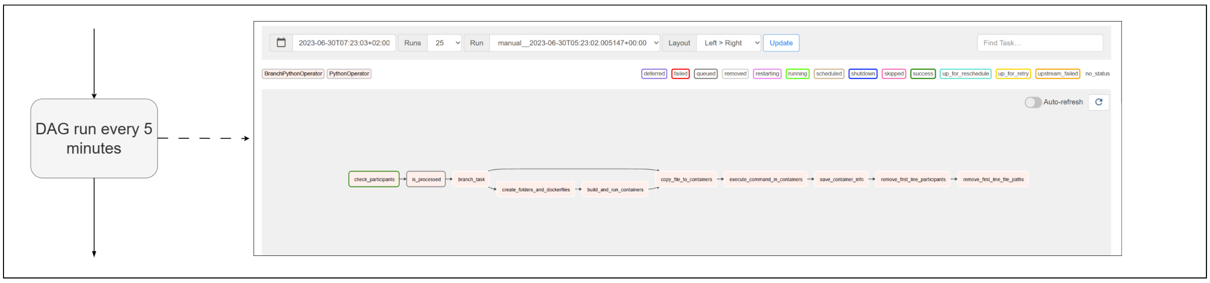

2023-06-30T07:23:03+02:00 Runs 25 Run manual__2023-06-30T05:23:02.005147+00:00 Layout Left > Right

Figure 4.1.3: Directed acyclic graph (DAG)

The Docker container's creation is a significant part of the workflow because it packages the computation logic and resources necessary for handling digital palynological slides. Once created, the Docker container operates in the local environment of the participant's machine, effectively bringing the computation logic to the data. This approach eliminates the need to move large slide MRXS files, solving data mobility issues and making the entire process significantly more efficient.

The workflow design approach allows the processing of large-scale images with maximum computational efficiency, minimal data mobility, and optimal use of local resources. It represents a breakthrough in handling large-scale digital palynological slides, creating new possibilities for further exploration and research in the field.

4.1.2 User interface

The user interface (UI) is critical to the federated computation workflow, and it acts as the primary point of interaction between users and the system. Designed with simplicity and functionality, UI aids the user registration process and initiates data entry.

4.1.2.1 Registration

During registration, users are presented with three input fields to provide the necessary information to participate in the federated computational network. These inputs include the "Username," "File path," and "Select a process".
Figure 4.1.4 shows UI and input fields.

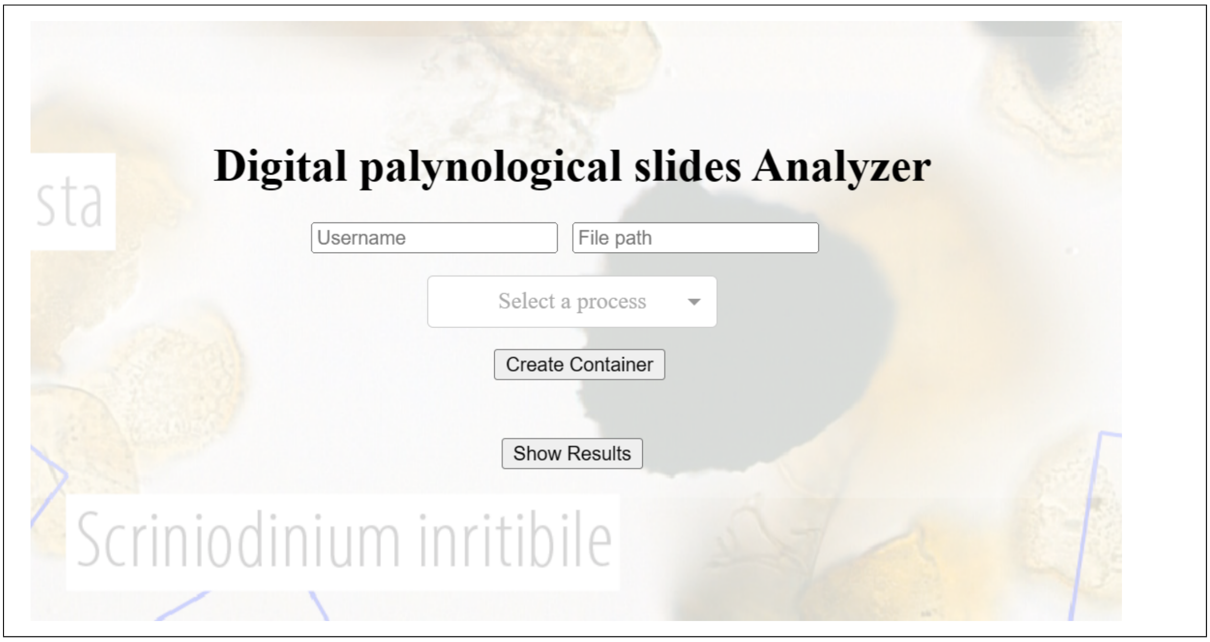

Digital palynological slides Analyzer

Figure 4.1.4: The user registration inputs.

The "Username" input allows users to enter their desired username or identifier within the federated computational network. The username will be used to recognize and differentiate users within the network.

In the "File path" input, users must specify the file path of an unprocessed slide image in their local system. This path is the location from which the system retrieves the slide data for processing. Users should provide an accurate file path to ensure successful data retrieval.

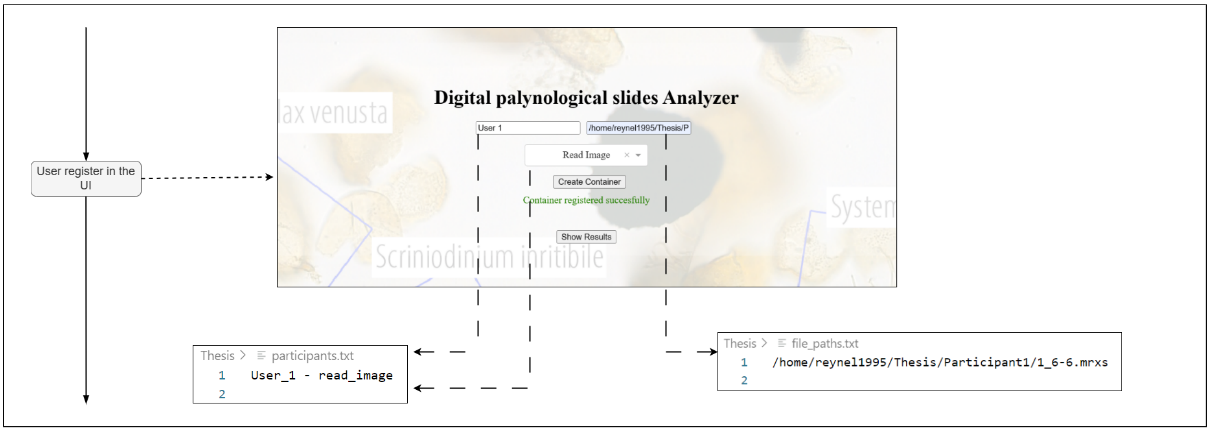
The "Select a process" input allows users to choose the specific process or processes they wish to include in the federated computational workflow. This selection empowers users to customize their analysis based on their individual requirements and preferences. Users can select one or multiple processes from the available options to tailor the workflow.

By providing these inputs during the registration phase, users establish their presence in the federated computational network and configure the initial parameters for data processing and analysis.

4.1.2.2 Create container

When users click the Create Container button, the information provided during the registration phase, including the username and the selected process, will be saved in a text file named "participants.txt". If the username contains a blank space, it will be replaced by an underscore. Also, the file path entered by the user will be stored in another text

file called file "paths.txt".
Figure 4.1.5 shows UI with the created text files. These files serve as the reference for the workflow because they are periodically checked every 5 minutes to identify any changes or updates.



User 1

/home/reynel1995/Thesis/P

Read Image

Container registered succesfully

Thesis > participants.txt

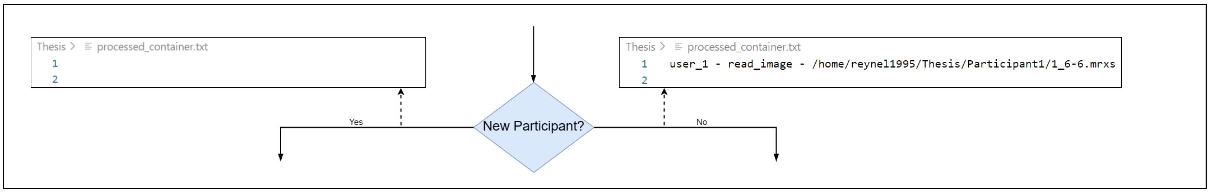
```
User_1 - read_image
```

Thesis > file_paths.txt

```
/home/reynel1995/Thesis/Participant1/1_6-6.mrxs
```

Figure 4.1.5: User Interface and created TXT files.

Before a successful registration is completed, the system performs a thorough verification process. It checks the values entered by the user against the entries in the "processed container.txt" file shown in Figure 4.1.5, which contains information about previously processed containers. The verification step helps determine whether the user is new to the system or has already registered a container, ensuring accurate handling of user data and preventing duplicate registrations.



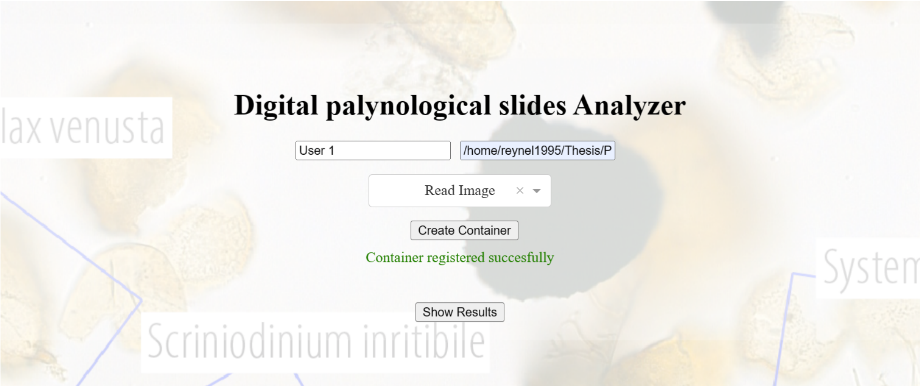
Thesis > processed_container.txt

Thesis > processed_container.txt

```
user_1 - read_image - /home/reynel1995/Thesis/Participant1/1_6-6.mrxs
```

Figure 4.1.6: Checking processed containers.

After a successful container registration, a message will be displayed indicating that the container has been registered successfully.
Figure 4.1.7 shows a successful user registration. This notification confirms that the user's container is now part of the federated computational workflow, ready to run the specified processes and contribute to the analysis of digital palynological slides.



User 1

/home/reynel1995/Thesis/P

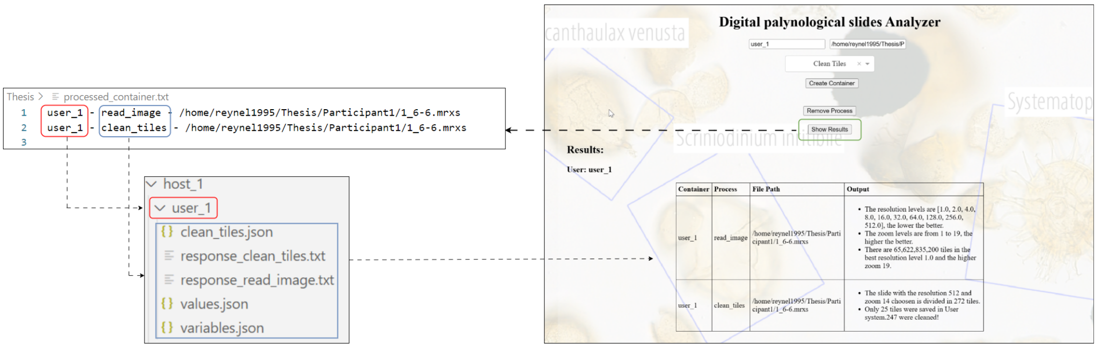
Container registered succesfully

Figure 4.1.7: A successful user registration.

4.1.2.3 Show results

When a user clicks the "Show Results" button, the system scans the "processed container.txt." Then it goes to the "Host1" system, the "Organizer" of the network, to look for the user's folder and extract details regarding the processed container, associated processes, and the corresponding files. This enables the retrieval of the generated results, which are then displayed coherently, grouped by container name.
Figure 4.1.8 shows the operation.

Figure 4.1.9 shows the UI displaying no results. If the "processed container.txt" file is either non-existent or empty, a message stating "No results available" promptly appears on the UI, ensuring users are informed about the absence of processed data. This diligent notification maintains transparency and provides clarity in the context of this thesis.



```
user_1 - read_image - /home/reynel1995/Thesis/Participant1/1_6-6.mrxs
user_1 - clean_tiles - /home/reynel1995/Thesis/Participant1/1_6-6.mrxs
```

user_1

/home/reynel1995/Thesis/P

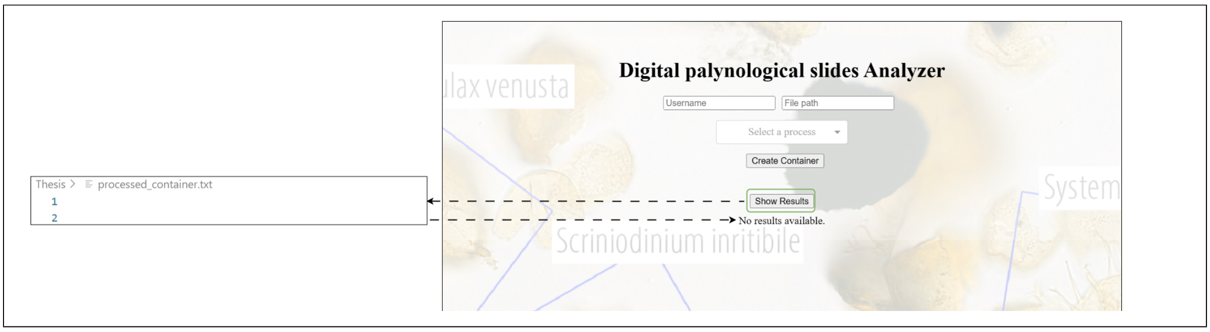
Results:

User: user_1

Container	Process	File Path	Output
user_1	read_image	/home/reynel1995/Thesis/Participant1/1_6-6.mrxs	• The resolution levels are [1.0, 2.0, 4.0, 8.0, 16.0, 32.0, 64.0, 128.0, 256.0, 512.0], the lower the better. • The zoom levels are from 1 to 19, the higher the better. • There are 65,622,835,200 tiles in the best resolution level 1.0 and the higher zoom 19.
user_1	clean_tiles	/home/reynel1995/Thesis/Participant1/1_6-6.mrxs	• The slide with the resolution 512 and zoom 14 choosen is divided in 272 tiles. • Only 25 tiles were saved in User system.247 were cleaned!

Figure 4.1.8: Show result button process.

The system functionalities also include the presence of a "Remove Process" button when the "Processed container.txt" file is updated with processes.
Figure 4.1.10 shows the remove process workflow. This feature allows users to delete previously executed processes, facilitating corrections and adjustments to the processed data. The availability of this option enhances the flexibility and integrity of the obtained results within the framework.



No results available.

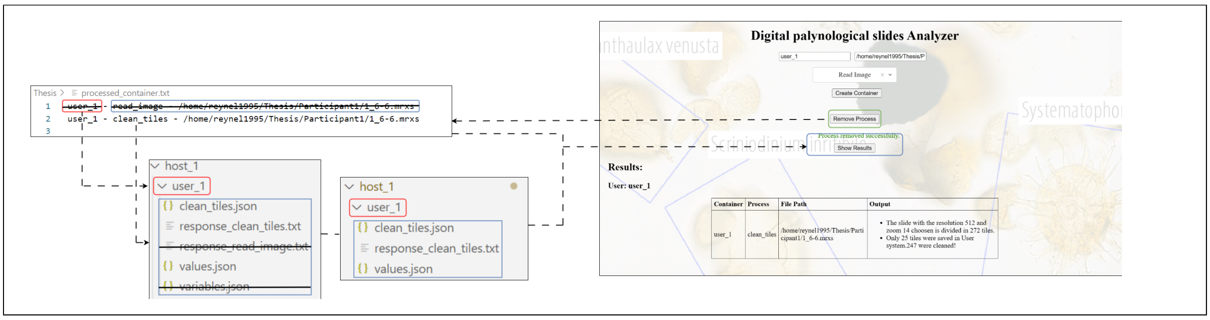
Figure 4.1.9: Show no result message.

The implemented workflow allows users to visualize processed container results. It scans the "processed container.txt" file, displays the results in an organized manner, and communicates the absence of processed data when necessary. Also, the system provides a "Remove Process" button for users to make corrections and adjustments, ensuring the accuracy and quality of the research results.

4.1.2.4 Removed process

The "Remove Process" functionality within the UI is vital to the workflow. When the button is clicked, the system initiates a series of checks and actions to ensure the proper removal of the selected process and associated files.

When the remove process is activated, the workflow first verifies the existence of the process within the "processed container.txt" file. If the process is found, the system proceeds to locate the corresponding user's folder within the host1 or organizer of the network system. The files associated with the process are identified and promptly deleted, ensuring the elimination of any remnants as is shown in Figure 4.1.10.



~~user_1 - read_image - /home/reynel1995/Thesis/Participant1/1_6-6.mrxs~~
user_1 - clean_tiles - /home/reynel1995/Thesis/Participant1/1_6-6.mrxs

Process removed successfully.

Results:

User: user_1

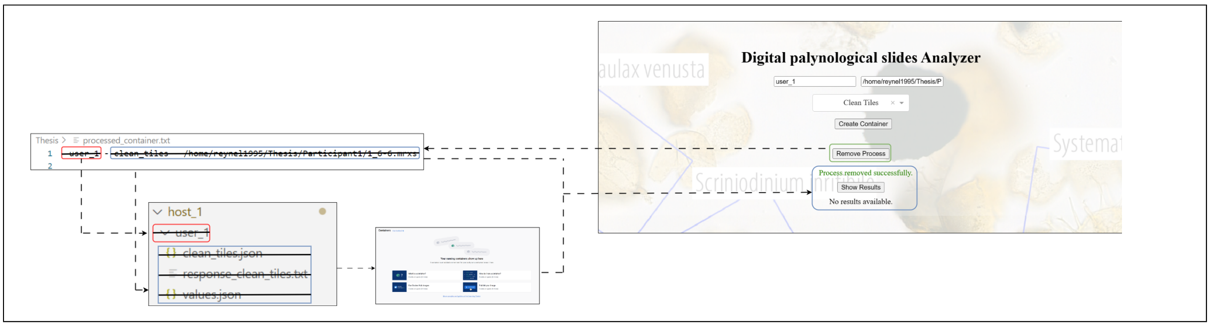
Container	Process	File Path	Output
user_1	clean_tiles	/home/reynel1995/Thesis/Participant1/1_6-6.mrxs	• The slide with the resolution 512 and zoom 14 choosen is divided in 272 tiles. • Only 25 tiles were saved in User system.247 were cleaned!

Figure 4.1.10: Removed process workflow.

The system updates the "processed container.txt" file, removing the entry of the deleted process. This ensures accurate tracking and maintains the integrity of the container processing history.

In cases where the removed process is the last one associated with a specific user, the workflow undertakes additional steps to ensure a thorough cleanup. In addition to deleting the related files and removing the process from the "processed container.txt" file, the system stops the container associated with the user. It also eliminates the Docker image

and Dockerfiles created during the process, effectively freeing up resources and minimizing unnecessary storage usage.
Figure 4.1.11 shows the UI displaying removed process Docker workflow.



~~user_1 - clean_tiles - /home/reynel1995/Thesis/Participant1/1_6-6.mrxs~~

Process removed successfully.

No results available.

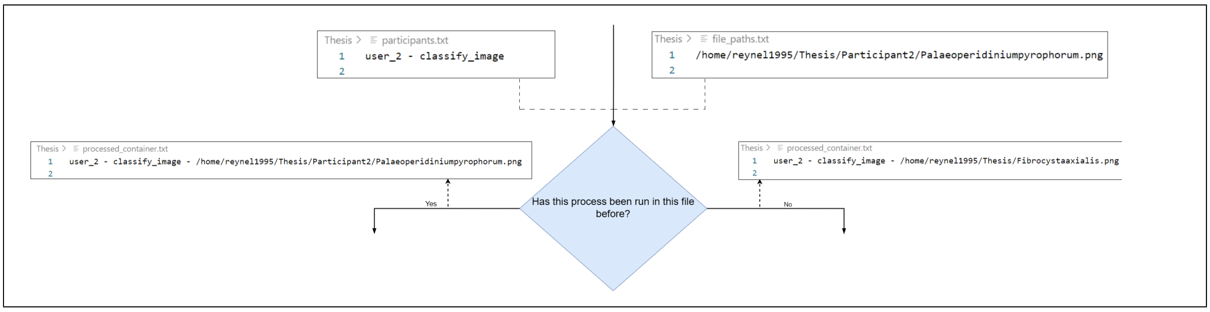
Figure 4.1.11: Removed process Docker workflow.

After successfully removing the process, the UI provides a message confirming the successful completion of the operation. This informative notification ensures transparency and clarity, keeping users informed of the status and outcome of their actions.

The "Remove Process" functionality within the UI enables users to manage and maintain their data integrity effectively. The system ensures a streamlined and efficient workflow by conducting thorough checks, removing associated files, updating the processing history, and performing container and Docker cleanup when necessary.

4.1.3 Docker implementation

Docker plays an important role in implementing the federated computational workflow. As a platform that automates the deployment of applications inside lightweight, portable containers, Docker provides an efficient means of running our image processing tasks. The implementation of Docker follows a specific condition. If there is no existing Docker container running with the same name as the registered username, the Docker image will be created. However, if a container is already running, the PY file will be copied to the existing container for execution unless the process is already run.
Figure 4.1.12 shows the Docker implementation.


Thesis > participants.txt
Thesis > file_paths.txt
Thesis > processed_container.txt
Thesis > processed_container.txt
Has this process been run in this file before?
Yes
No

```
user_2 - classify_image
```

```
/home/reynel1995/Thesis/Participant2/Palaeoperidiniumpyrophorum.png
```

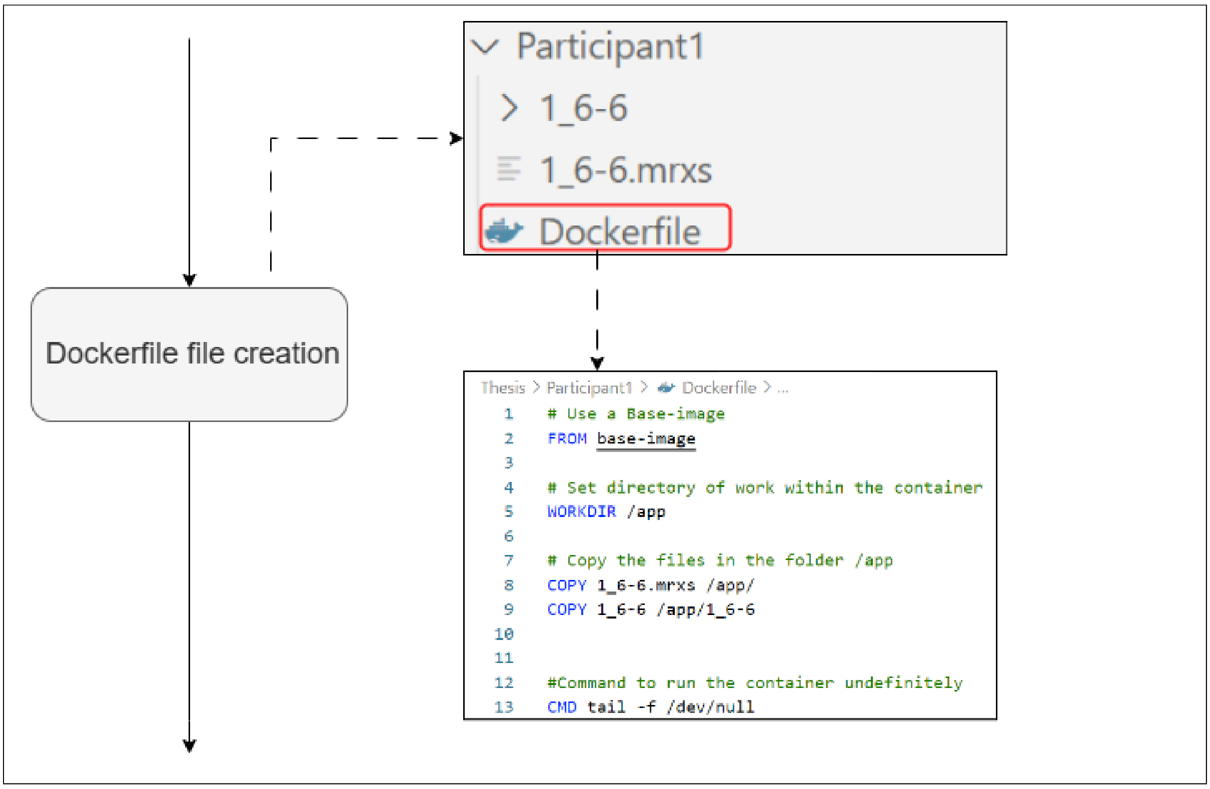
```
user_2 - classify_image - /home/reynel1995/Thesis/Participant2/Palaeoperidiniumpyrophorum.png
```

```
user_2 - classify_image - /home/reynel1995/Thesis/Fibrocystaaxialis.png
```

Figure 4.1.12: Checking processes run on already built container.

When the MRXS path file from the registered user and the username are received in the folder where the MRXS file is located, a Dockerfile is written with the instructions to

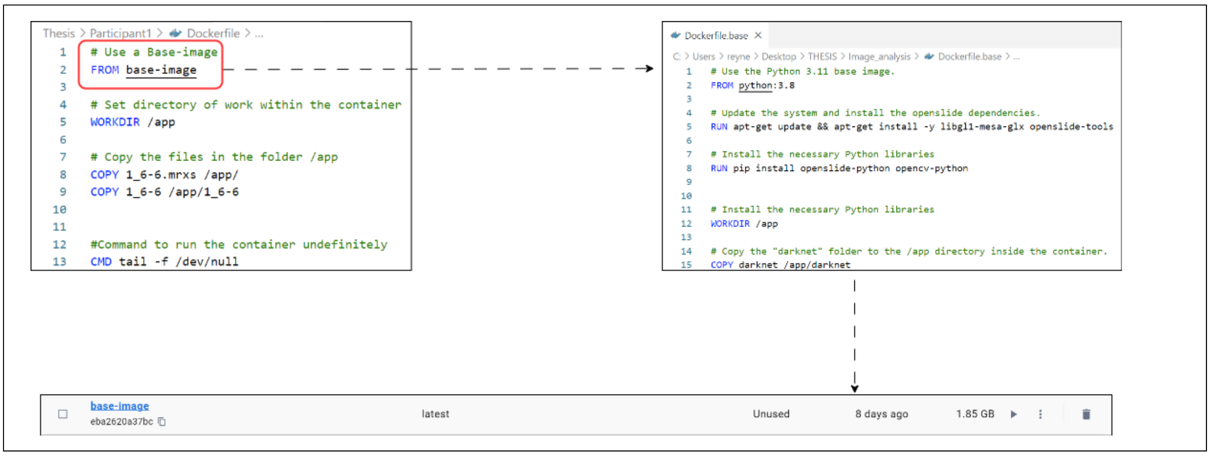
"COPY" the MRXS.
Figure 4.1.13 shows renamed folder and Docker file creation. The Dockerfile is written using a "Base Docker Image" that already contains shared dependencies and packages, and the operation is shown in Figure 4.1.14.



```
# Use a Base-image
FROM base-image

# Set directory of work within the container
WORKDIR /app

# Copy the files in the folder /app
COPY 1_6-6.mrxs /app/
COPY 1_6-6 /app/1_6-6


#Command to run the container undefinitely
CMD tail -f /dev/null
```

Figure 4.1.13: Folder renamed and Docker file creation.

The Dockerfile image is created, and the "Host1" will send an instruction to the folder where the Dockerfile is to be built and run the container on the user's machine.
Figure 4.1.15 shows the process. This container serves as a self-contained execution environment for large-scale image processing. Docker ensures the necessary dependencies, including the Python script for processing, are encapsulated within this container. This allows for seamless execution, independent of the underlying system.


Thesis > Participant1 > Dockerfile > ...
Dockerfile.base ×
C: > Users > reyne > Desktop > THESIS > Image_analysis > Dockerfile.base > ...

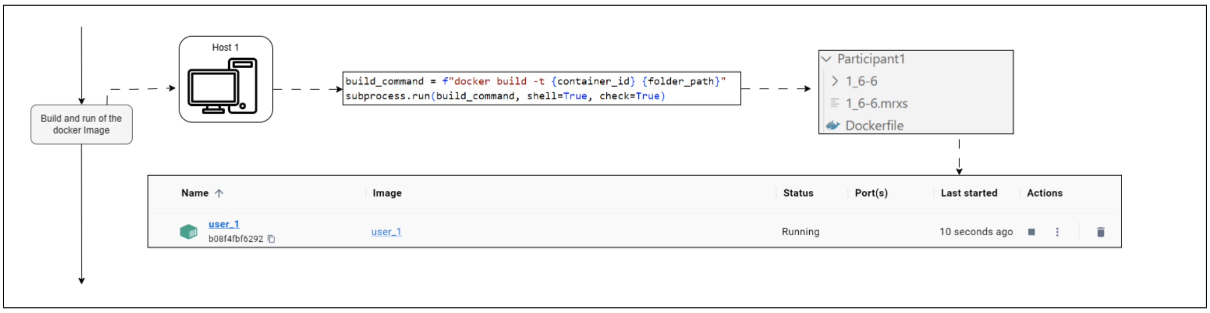
```
# Use a Base-image
FROM base-image

# Set directory of work within the container
WORKDIR /app

# Copy the files in the folder /app
COPY 1_6-6.mrxs /app/
COPY 1_6-6 /app/1_6-6


#Command to run the container undefinitely
CMD tail -f /dev/null
```

```
# Use the Python 3.11 base image.
FROM python:3.8

# Update the system and install the openslide dependencies.
RUN apt-get update && apt-get install -y libgl1-mesa-glx openslide-tools

# Install the necessary Python libraries
RUN pip install openslide-python opencv-python


# Install the necessary Python libraries
WORKDIR /app

# Copy the "darknet" folder to the /app directory inside the container.
COPY darknet /app/darknet
```

□	base-image eba2620a37bc	latest	Unused	8 days ago	1.85 GB	▶ ⋮ 🗑

Figure 4.1.14: Base image creation.

The image processing script is then transmitted to the Docker container, which the container runs on the slide MRXS file.
Figure 4.1.16 shows the transmission of the image file, This allows the computation logic to be moved to the data, in contrast to the traditional approach of transferring large datasets to the processing domain. The Docker container returns the processed data and metadata to the host system once the computation is completed.

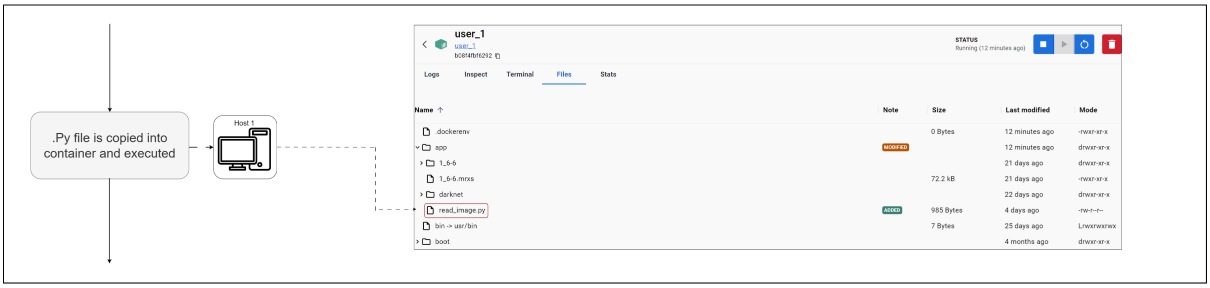


```
build_command = f"docker build -t {container_id} {folder_path}"
subprocess.run(build_command, shell=True, check=True)
```

Name	Image	Status	Port(s)	Last started	Actions
user_1 b08f4fbf6292	user_1	Running		10 seconds ago	■ ⋮

Figure 4.1.15: Build image and run container.

The application of Docker in the workflow introduces several advantages such as resource isolation and portability across different systems. Resource isolation ensures each Docker container has its resources independent of other containers. Also, Docker facilitates scale-up capabilities as additional Docker containers can be readily spun up or down depending on the computational demand. This level of flexibility and scalability would not be possible with traditional virtual machines.



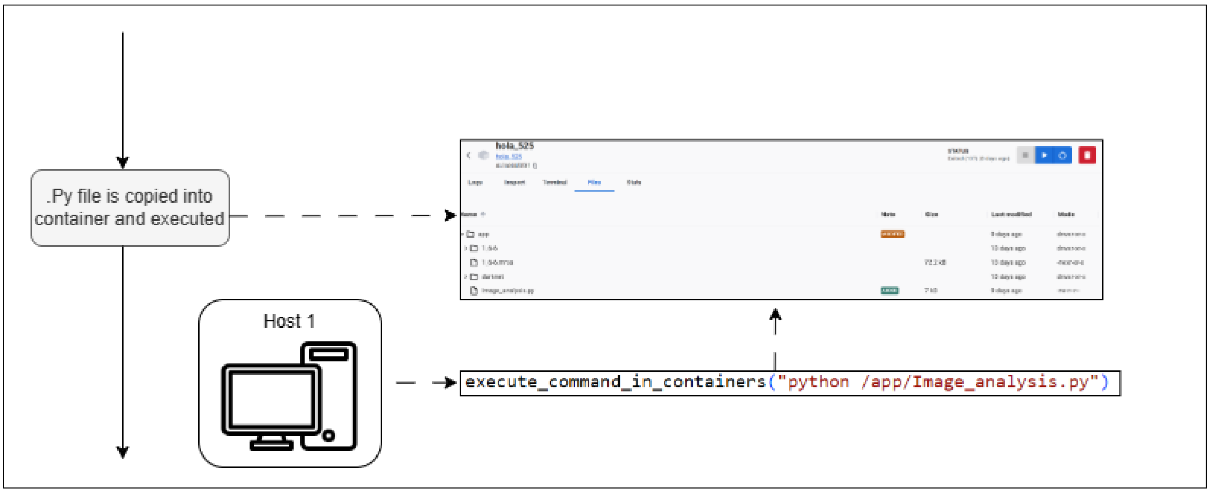
Name	Note	Size	Last modified	Mode
.dockerenv		0 Bytes	12 minutes ago	-rwxr-xr-x
app	MODIFIED		12 minutes ago	drwxr-xr-x
1_6-6			21 days ago	drwxr-xr-x
1_6-6.mrxs		72.2 kB	21 days ago	-rwxr-xr-x
darknet			22 days ago	drwxr-xr-x
read_image.py	ADDED	985 Bytes	4 days ago	-rw-r--r--
bin -> usr/bin		7 Bytes	25 days ago	Lrwxrwxrwx
boot			4 months ago	drwxr-xr-x

Figure 4.1.16: Transmission of image analysis file.

4.1.4 Large-scale image processing

In the federated computational workflow, large-scale image processing occurs within the Docker container initiated on the user's machine. The Docker container handles the large digital palynological slide files the user submits.

The Python script used for the image processing is transmitted to the Docker container, and it runs in the container.
Figure 4.1.17 shows the processing of slide image files. This script takes the necessary processing steps such as splitting the large slide images into smaller tiled images, applying required preprocessing operations, and extracting the relevant features from these tiles.



```
execute_command_in_containers("python /app/Image_analysis.py")
```

Figure 4.1.17: Execution of image analysis file.

It is important to mention that the need for large-scale data is minimized by performing these intensive processing tasks inside the Docker container and directly on the user's machine. This method substantially reduces the time and computational resources that would have been spent if these large slide files were first transferred over the network. Also, Docker provides a more scalable solution to the problem of large-scale image processing.

Once the processing is completed, the Docker container packages the processed data and metadata and returns them to the host system. The data are then available to be used in the subsequent steps of the workflow such as microfossil detection and classification task.

4.1.5 Challenges and solutions

The implementation of the federated computation workflow for processing large-scale digital palynological slides encountered some challenges. The size of the MRXS files and the computational demands of the image processing tasks posed significant hurdles. However, devised several solutions were devised to solve the issues.

One of the main challenges was managing the computational resource requirements of the workflow. Because the MRXS files are large, processing them locally can easily stretch the system's resources. To mitigate the problem, a federated computing approach that distributes the computational load across multiple machines was adopted. By leveraging the power of distributed computing, we could efficiently process large-scale image data without straining individual systems.

Another significant challenge was ensuring efficient and secure data transfer between the Docker containers and the host system. Considering the large file sizes involved, data transfers take considerable time and pose security risks. To address the issue, the workflow was designed to minimize data transfers by processing the data within the Docker containers, reducing the need for data movement. This approach expedited the processing time and enhanced data security by keeping the data within controlled containerized environments.

Challenges were encountered in creating a user-friendly and intuitive interface that could facilitate user interaction with the system. Potential mistakes and errors during the user registration and data uploading process were anticipated. A robust web-based UI using Dash, incorporating comprehensive error handling and validation mechanisms was developed to handle the issues. This ensures the interface can handle different user inputs, providing informative error messages and guiding users to correct mistakes. This user-centered approach enhances the overall user experience and minimizes frustration or confusion.

The solutions significantly improved the efficiency, security, and usability of our federated computation workflow, making it a viable method for processing large-scale digital palynological slides. By addressing the challenges related to computational resources, data transfer, and UI design, a robust framework was developed. The framework streamlines the image processing workflow while ensuring the accuracy and reliability of the results.

4.2 Palynological Image Analysis Workflow

The federated computation workflow incorporates advanced computer vision techniques for detecting and classifying microfossils on digital palynological slides. This crucial step enables the identification and classification of microfossils present in the slides, offering a powerful mechanism for analyzing vast amounts of data. The following subsections provide a detailed overview of the image analysis processes implemented in the workflow.

The image analysis workflow encompasses several interdependent processes integrated into the Dash application. The development of these processes involved meticulous planning and consideration of the dependencies between them. The first process is the slide image reading to extract the necessary data from the slides for further analysis. Also, tiled images are generated from the slide at this stage.

The data cleaning process is another important step in the image analysis workflow. Once the slide image is read and tiles are generated, cleaning eliminates unwanted tiled images such as blank tiles or tiles with undesirable images. The cleaned tiled images serve as inputs for the subsequent object detection and classification tasks.

The watershed segmentation algorithm from OpenCV (OpenCV, 2023), a popular segmentation technique, identifies and separates individual objects on the cleaned tiled images to facilitate classification. This procedure partitions the image into regions based on intensity and spatial information, facilitating the precise localization of microfossils. The results obtained from the watershed segmentation serve as inputs for the subsequent classification task.

The classification process utilizes a trained CNN model to classify microfossils identified on tiles. This trained model analyzes the features of the dinoflagellates and assigns them to predefined classes. Annotation details of labeled dinoflagellates are extracted using the provided XML and MRXS file paths. The annotation details are converted into a readable format for object detection with the Mask RCNN algorithm.

Figure 4.2 presents the workflow for palynological slide image analysis. The image analysis workflow, implemented within the Dash application, encompasses the slide image reading, data cleaning, object detection with the Watershed algorithm, and classification with a trained RCNN model. The processes are interconnected, ensuring the accurate identification and categorization of microfossils. The extraction of annotations and their integration into the classification process further enhances the accuracy of the analysis.

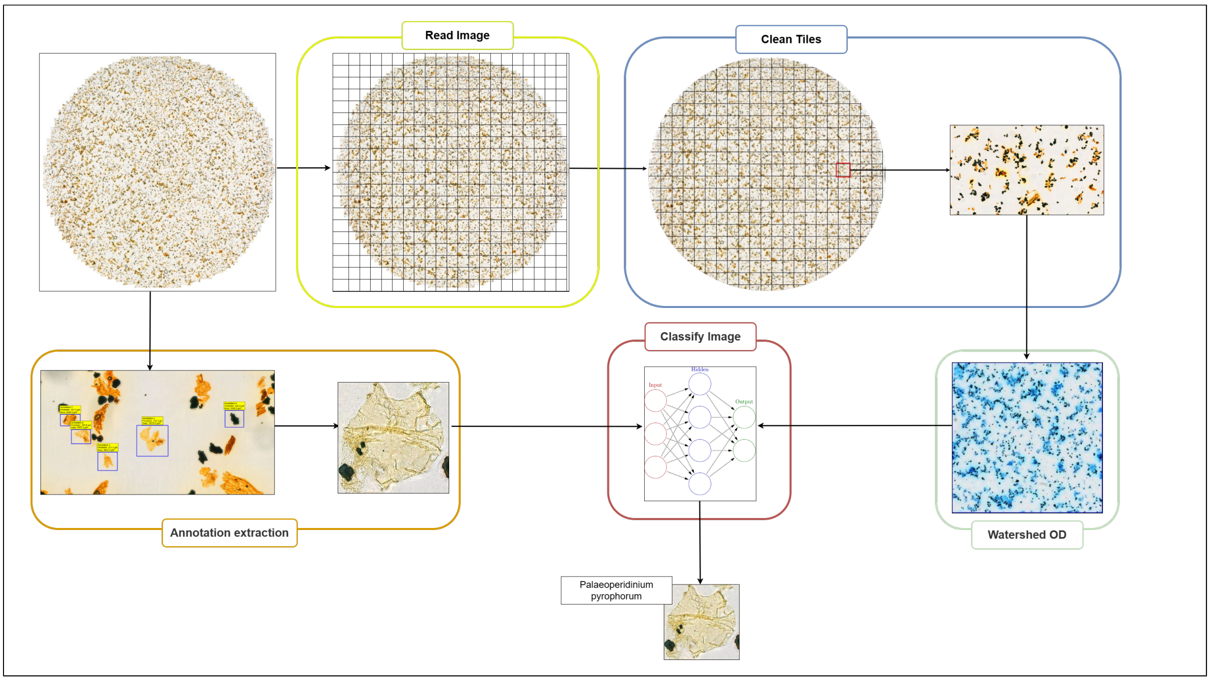


Figure 4.2.1: The palynological image analysis workflow.

4.2.1 Read image

The "Read Image" process is a crucial initial step in the image analysis workflow. It involves utilizing a Python script that employs the OpenSlide library to extract essential data from the digital palynological slide for further analysis. The script begins by importing the necessary libraries, including the OpenSlide package. It then locates the MRXS file in the specified directory.

The OpenSlide method is used to open and access the image data once the slide is located. To facilitate efficient navigation and analysis at different levels of resolution of a tile object obtained from the slide, the script utilizes the DeepZoomGenerator from OpenSlide. This generator produces tiles of varying sizes, enabling effective slide examination.

The number of deep zoom levels and downsample factors for the levels are determined from the slide metadata. These factors indicate the level of magnification or downsampling applied to the slide image. The script calculates the total number of tiles in the slide at the highest deep zoom level by analyzing the metadata.

The number of deep zoom levels in the digital palynological slide refers to the different levels of magnification available for viewing and analyzing the image. Each deep zoom level represents a specific level of detail and resolution. Higher zoom levels provide a more detailed view of the slide, allowing for a closer examination of individual components such as microfossils. By contrast, lower zoom levels offer a broader overview of the entire slide, enabling a more comprehensive understanding of its structure.
Figure 4.2.2 presents a graphical representation of zoom levels.

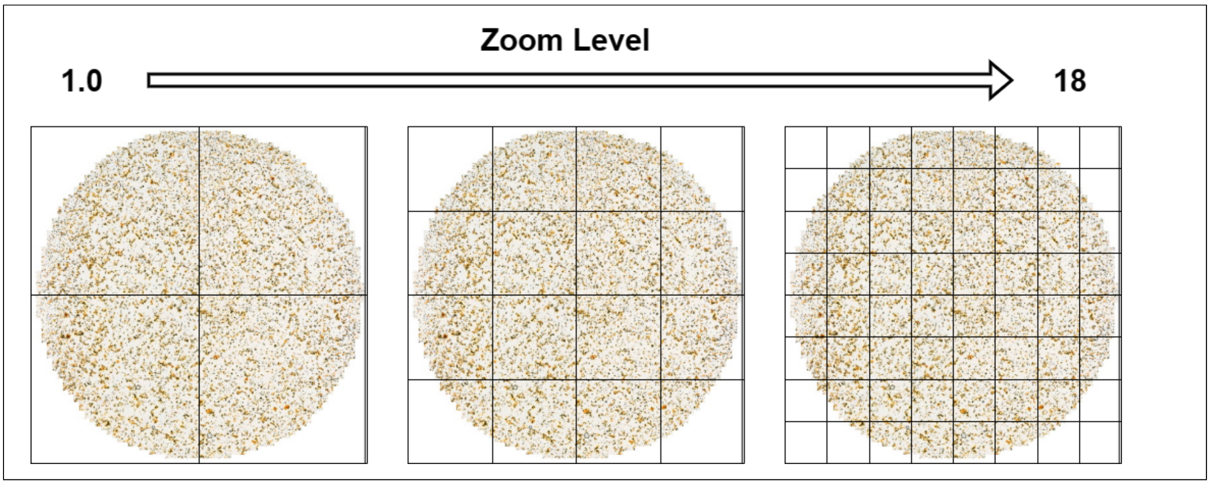


Figure 4.2.2: A graphical representation of zoom levels.

The downsample factors are values that indicate the level of downsampling applied to the image at each zoom level. Downsampling involves reducing the image resolution by discarding some fine details. The downsample factors determine the level of compression or reduction in image quality. Higher downsample factors result in lower resolution and smaller file sizes, making the image more manageable for processing and analysis. However, higher downsample factors may lead to some loss of information or fine details, particularly at lower zoom levels.
Figure 4.2.3 illustrates the downsampling effect.

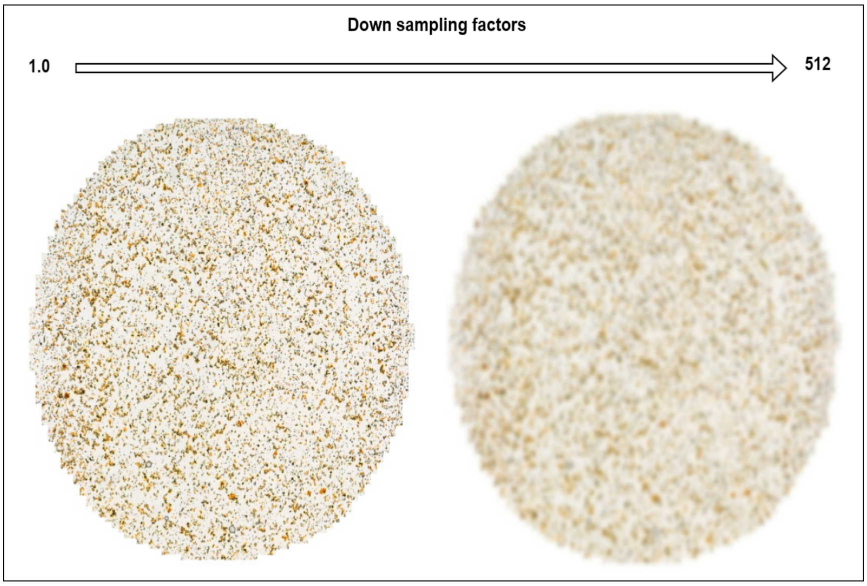
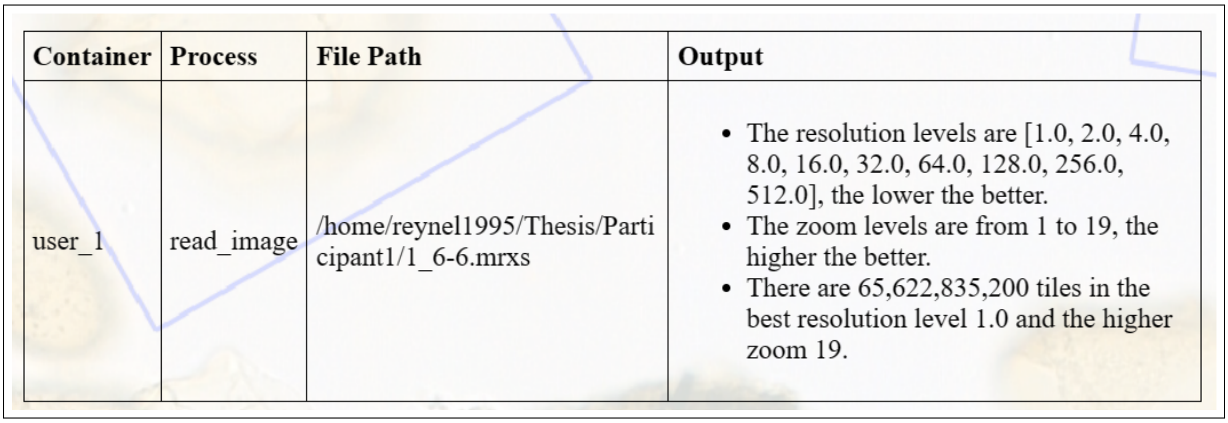


Figure 4.2.3: An illustration of downsampling effect.

The total number of tiles in the slide image at the highest deep zoom level represents the subdivision of the slide into smaller tiled images for efficient analysis and navigation. The tiles allow for a regional manipulation of the slide, enabling the detection and analysis of microfossils at a localized level. The number of tiles extracted from the slide increases with the deep zoom level. Each tile represents a specific region of the slide image with vertical and horizontal coordinates, and desirable tiles serve as inputs for further slide image analysis.

After extracting the necessary data, the script closes the slide to release system resources. The output of the "Read Image" process is a JSON object that contains the downsample factors, the number of deep zoom levels, and the total number of tiles in the slide. Figure 4.2.4 shows the process output displayed on the Dash app. The total of tiles at the highest zoom level and best resolution level indicates the space available for the user, and the number of tiles is smaller at a lower resolution.



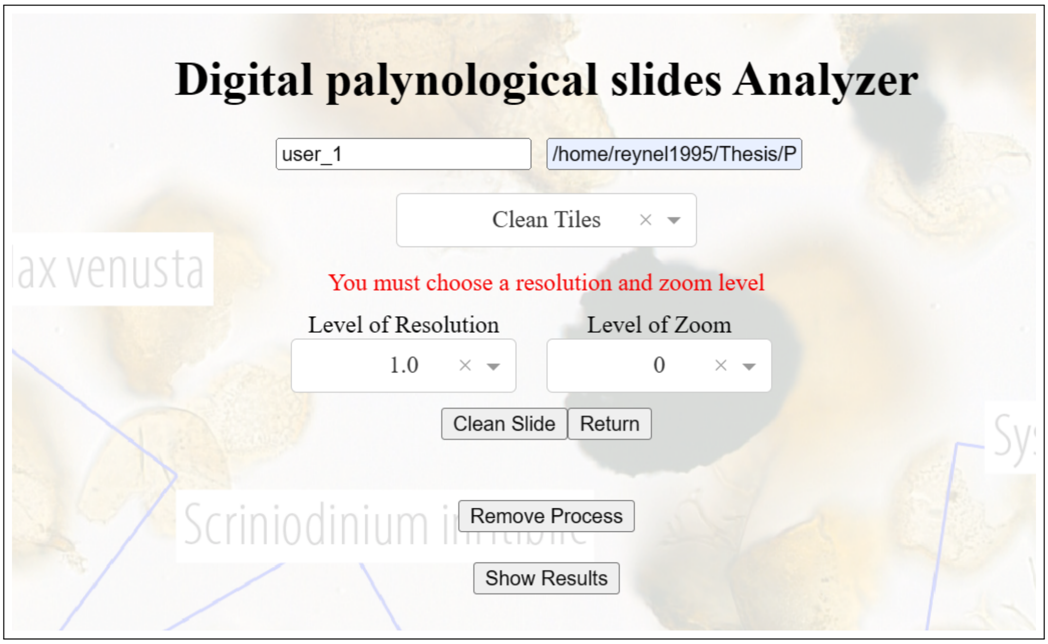
Container	Process	File Path	Output
user_1	read_image	/home/reynel1995/Thesis/Participant1/1_6-6.mrxs	• The resolution levels are [1.0, 2.0, 4.0, 8.0, 16.0, 32.0, 64.0, 128.0, 256.0, 512.0], the lower the better. • The zoom levels are from 1 to 19, the higher the better. • There are 65,622,835,200 tiles in the best resolution level 1.0 and the higher zoom 19.

Figure 4.2.4: The output of slide image reading process.

4.2.2 Clean tiles

The "Clean Tiles" process is a crucial step in the image analysis workflow that focuses on removing unwanted or blank tiles from the read image. This process ensures that subsequent analysis is performed only on meaningful and informative regions by eliminating unwanted tiles using the YOLOv4-Tiny algorithm. The following steps outline the procedure for cleaning the tiles.

After reading the slide image, the user is prompted to select a resolution and zoom levels. Figure 4.2.5 shows the UI with the zoom level and resolution selection options. The values are extracted from the JSON file generated during the "Read Image" process. The resolution level determines the size of the tiles, while the zoom level determines the level of image magnification.



Digital palynological slides Analyzer

user_1

/home/reynel1995/Thesis/P

Clean Tiles

You must choose a resolution and zoom level

Level of Resolution: 1.0

Level of Zoom: 0

Figure 4.2.5: Selection of zoom level and resolution.

The cleaning process begins with tiled image generation with the DeepZoomGenerator from the OpenSlide deepzoom module. This tool generates tiles of the specified resolution, effectively dividing the slide image into a grid of smaller regions for analysis. The division of the slide into tiles enables efficient examination and targeted evaluation.

For each tile, a series of evaluations are performed with the pre-trained deep-learning model to determine its relevance and suitability for further analysis. The tile is first converted to grayscale using the OpenCV library, simplifying subsequent analysis. The mean pixel intensity of the grayscale tile is then calculated. Tiles with a pixel intensity above a certain threshold are considered for further evaluation. In addition to pixel intensity, the percentage of white pixels within each tile is assessed. Tiles with a high percentage of white pixels, indicating blank or irrelevant regions, are discarded.
Figure 4.2.6 illustrates the data cleaning with YOlOv4-Tiny. The filtering process ensures that only tiles containing significant information are retained. The tiles that satisfy the predetermined criteria are saved in the output folder named Clean Tiles.

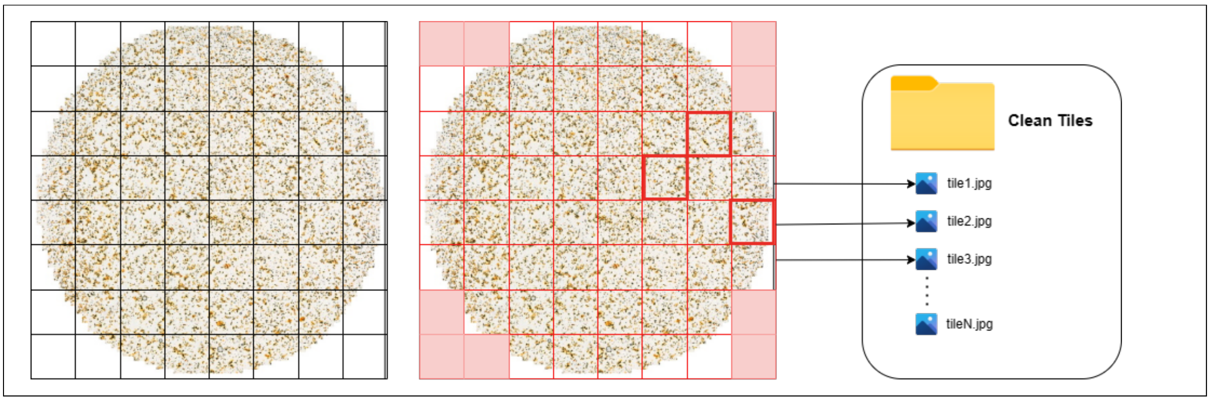
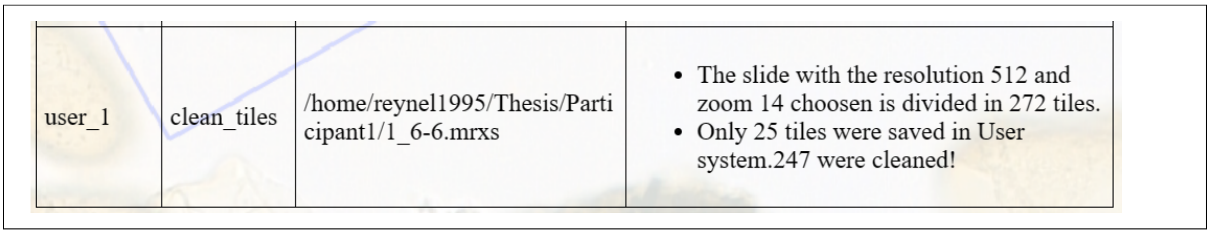


Figure 4.2.6: Selection of clean tiles for deep learning tasks.

The "Clean Tiles" process also provides valuable metadata, including the number of tiles that passed the cleaning process and the total number of tiles at the selected zoom level. This information serves as a measure of the effectiveness of the cleaning procedure. The number of tiles that successfully pass the cleaning process indicates the presence of significant regions within the digital palynological slide. By contrast, the total number of tiles at the selected zoom level provides information about the scale of the analysis.

The outputs of the "Clean Tiles" process are saved in a JSON file. This file serves as a record of the cleaning results and is later accessed by the Dash app to display the information to the user.
Figure 4.2.7 shows outputs displayed after the "Clean-Tiles" operation. The JSON file ensures that the metadata remains persistent and readily available for future reference.



user_1	clean_tiles	/home/reynel1995/Thesis/Participant1/1_6-6.mrxs	• The slide with the resolution 512 and zoom 14 choosen is divided in 272 tiles. • Only 25 tiles were saved in User system.247 were cleaned!

Figure 4.2.7: The outputs of tile cleaning process.

4.2.3 Watershed segmentation

The "Apply Watershed" process is a crucial step in the image analysis workflow. It is a preliminary object detection used to detect potential microfossils on the cleaned tiles. The process utilizes the watershed algorithm from OpenCV to separate microfossil images from the background, enhancing subsequent object detection and classification tasks. Figure 4.2.8 shows a schematic of the watershed segmentation process.

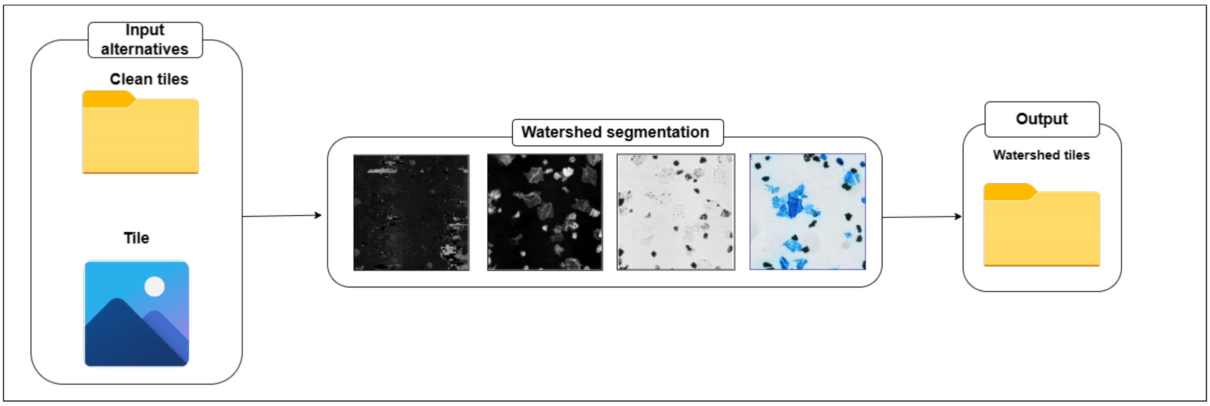


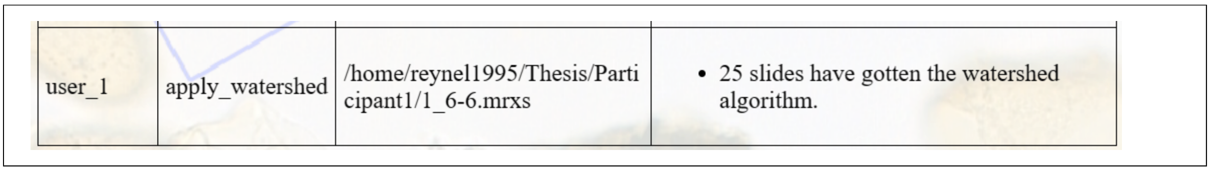
Figure 4.2.8: A schematic of the watershed segmentation process.

From Figure 4.2.8, the watershed script first defines the input folder path where the cleaned tile images are stored. The algorithm also requires an output folder path where segmented images will be segmented. If the output folder does not exist, the script creates it.

For each image in the input folder, the following prcesses are performed:

1. Load the image and convert it to grayscale.
2. Apply thresholding to create a binary image, separating the foreground (microfossils) from the background.
3. Perform morphological opening to remove small objects and noise.
4. Apply distance transform to obtain the distance map, representing the distance of each pixel from the background.
5. Apply thresholding to the distance map to obtain the foreground markers.
6. Apply the watershed algorithm to segment the foreground markers and obtain the labels.
7. Convert the grayscale image to a 3-channel format for visualization purposes.
8. Apply the watershed segmentation algorithm to the 3-channel image, using the markers to define the regions.
9. Save the segmented image to the output folder, marking the microfossils with a distinctive color.

This important information stored in a JSON file enables further analysis and identification of the segmented microfossils. The JSON file acts as a persistent and readily accessible metadata source, ensuring its availability for future reference. The stored data is later accessed by the Dash application, allowing the user to visualize and explore the segmentation results conveniently.
Figure 4.2.9 shows the output of the watershed segmentation process.



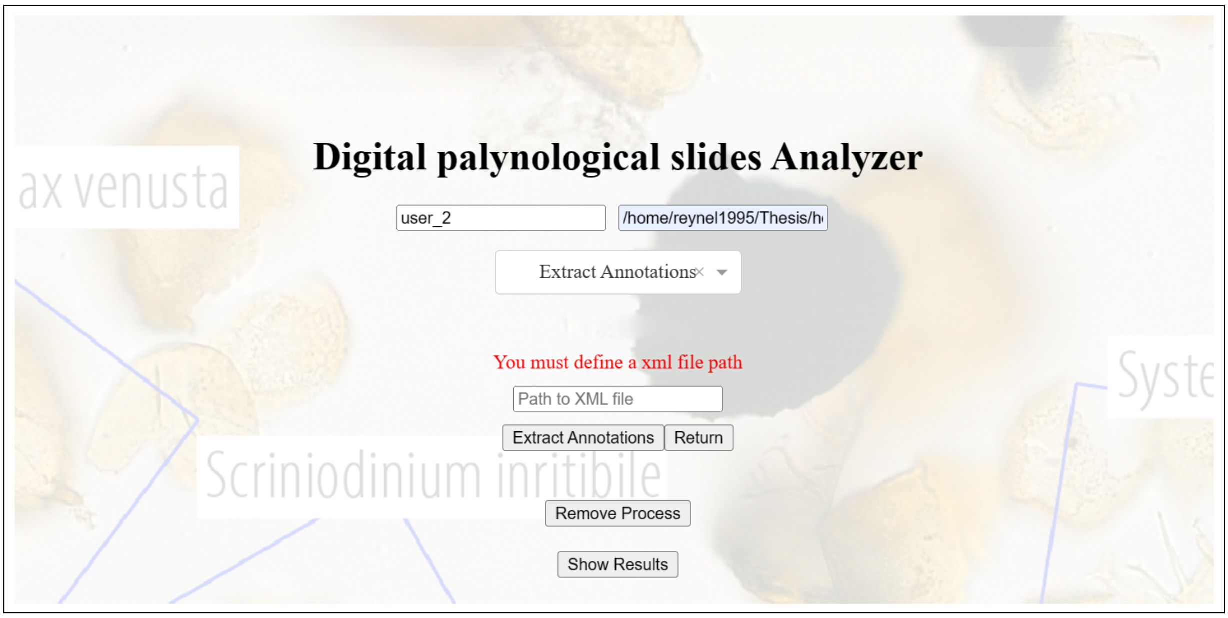
user_1	apply_watershed	/home/reynel1995/Thesis/Participant1/1_6-6.mrxs	• 25 slides have gotten the watershed algorithm.

Figure 4.2.9: The output of watershed segmentation process.

The watershed algorithm has the potential for microfossil quantification to evaluate their various characteristics. For example, microfossil size, shape, and distribution on a slide can be quantitatively analyzed. This can provide better insights into paleontological research, including species abundance, community composition, and ecological patterns.

4.2.4 Annotation extraction

The extraction of microfossil annotation details is an important process in the workflow. Figure 4.2.10 shows the UI with the process initialization.



Digital palynological slides Analyzer

user_2

/home/reynel1995/Thesis/h

You must define a xml file path

Path to XML file

Figure 4.2.10: The initialization of the annotation extraction process.

The user activates annotation extraction in the workflow through the UI by providing the XML file path and the corresponding MRXS file. The process involves locating and copying the MRXS image file and the XML file into the Docker environment or creating a Docker container with the files if necessary. The extraction process is then executed in the Docker environment with a PY file. Figure 4.2.11 shows the execution of the annotation extraction process.

The XML file is parsed using the ElementTree library to create a tree structure. The annotation sections in the XML file contain microfossil annotation details such as the name, type, and polygon points for each labeled dinoflagellate.
The polygon points are extracted and transformed into a list of coordinate tuples for each annotation. These points define the boundaries of the region of interest within the MRXS image file. A rectangular region is calculated by determining the minimum and maximum x and y values from the polygon points.

The regions of interest are extracted from the slide image based on the calculated coordinates. These regions represent the labeled images associated with the annotations. The labeled images and their corresponding annotation details are stored in dictionaries and converted to a Pandas dataframe. It is important to mention that labeled images are converted from RGBA to RGB images before further analysis.

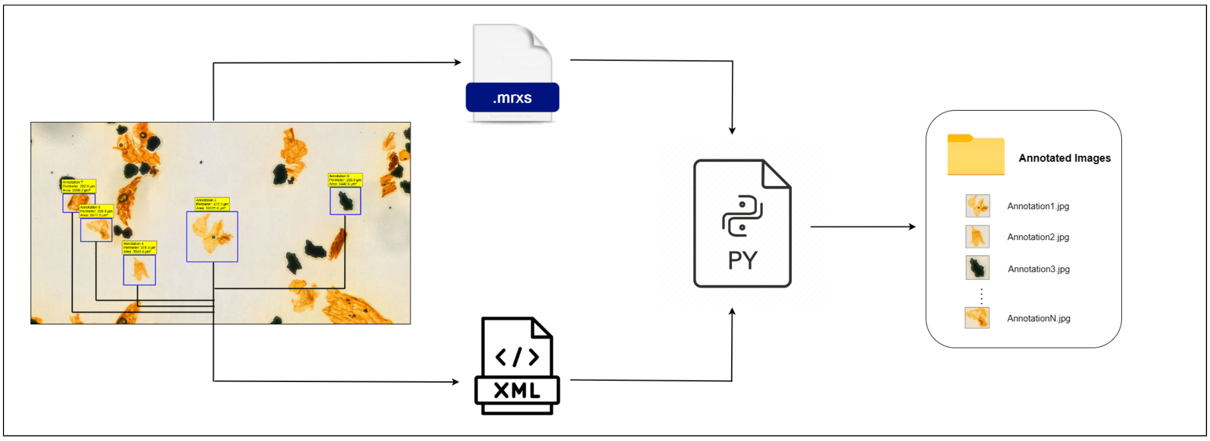
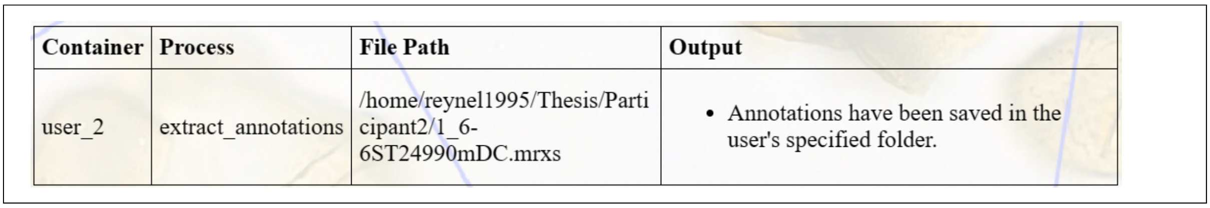


Figure 4.2.11: The annotation extraction process.

Figure 4.2.12 shows the output from the process displayed on the dash application. The annotation extraction process provides labeled images that serve as training data or references for subsequent analysis and classification tasks in the computational workflow.



Container	Process	File Path	Output
user_2	extract_annotations	/home/reynel1995/Thesis/Participant2/1_6-6ST24990mDC.mrxs	• Annotations have been saved in the user's specified folder.

Figure 4.2.12: Visualizing the annotation extraction output.

4.2.5 Image classification

This section is divided into two subsections, model training and classification. The training subsection focuses on building and training a CNN algorithm with the training and validation datasets presented in Table 3.3.3. It is important to mention that the training process occurs on the host server, and the outputs, such as the trained model and training history, are also saved there.

4.2.5.1 Model training

The model training with the definition of the input arguments to the deep-learning algorithm including the directories of the training and validation datasets, the target image size, batch size, the number of classes, and the number of epochs. Data augmentation and normalization techniques are applied to the training dataset using the ImageDataGenerator from the Keras library. This helps increase the diversity of the training data, enabling the classification model to learn characteristic features and reduce overfitting problems. The validation set is rescaled to ensure consistency.

A CNN model is built using the sequential API from Keras, consisting of convolutional layers with increasing filter sizes, max-pooling layers for downsampling, and dense layers for classification. The model is compiled with the Adam optimizer, categorical cross-entropy loss function, and accuracy metric.

The training process involves fitting the model to the training data and validating it on the validation data for a specified number of epochs. The steps per epoch and validation steps are determined based on the batch and dataset sizes. The training progress is stored in a history object.

The trained model is saved as an h5 file, and the training history is saved as a pickle file. The files serve as inputs for future use and analysis.
Figure 4.2.13 present the model training process for classification task .

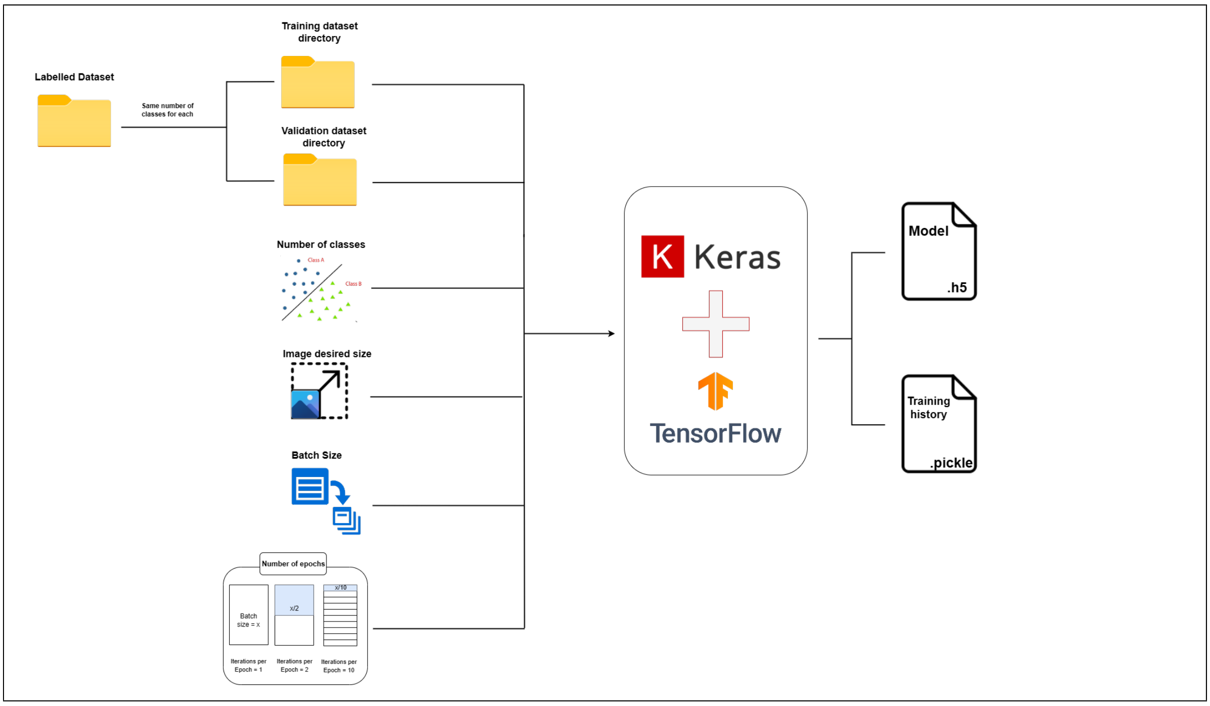


Figure 4.2.13: The training process for the classification model.

4.2.5.2 Classification

The user initiates the classification process by providing the path to the image file or the folder containing multiple image files through the UI. The system accesses the specified path and checks if a container is already created. If a container exists, the images are copied into the Docker environment with the trained model. The classification process is then executed in the Docker environment.
Figure 4.2.14 presents the classification process for predicting dinoflagellate classes with the trained CNN model. The source code is presented in **classify_microfossils.ipynb** given in the Appendix.

The image is resized using an OpenCV library to match the desired size. The model's class indices and class labels are defined. The image is preprocessed by normalizing the pixel values and expanding the dimensions. The model predicts the probabilities of the preprocessed image class.

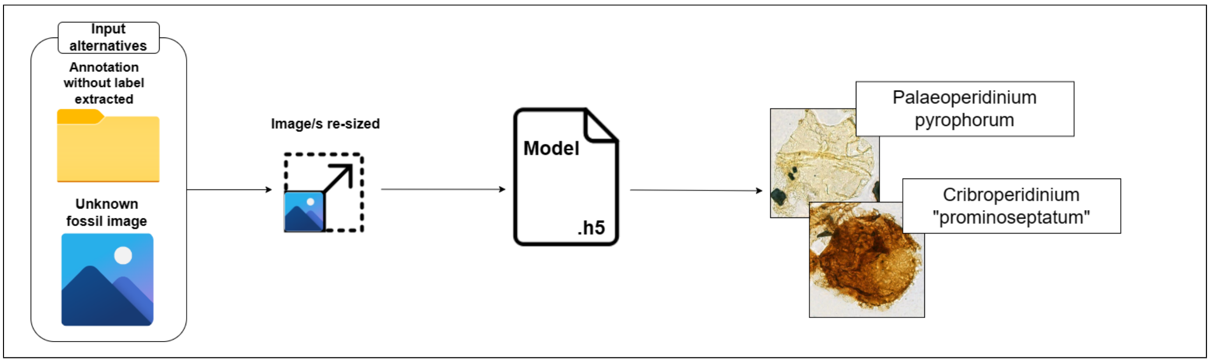
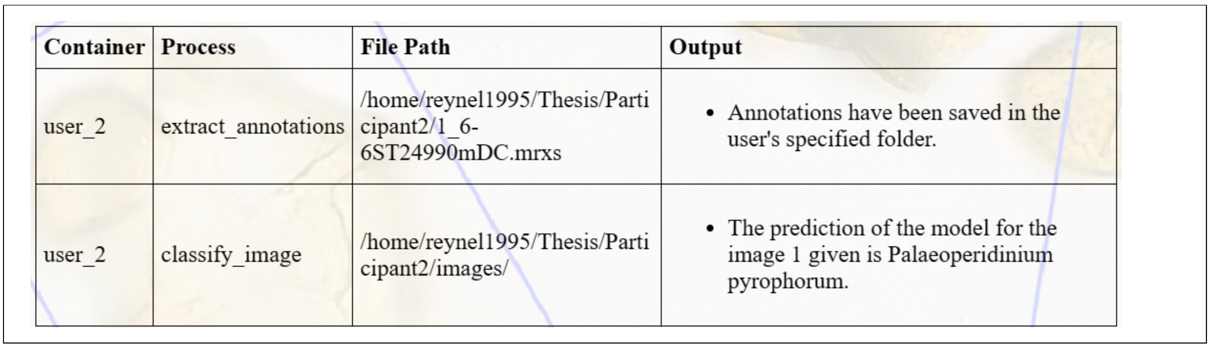

Input alternatives
Annotation without label extracted
Unknown fossil image
Image/s re-sized
Model
.h5
Palaeoperidinium pyrophorum
Cribroperidinium "prominoseptatum"

Figure 4.2.14: Dinoflagellate classification with the trained deep-learning model.

The predicted class index is obtained by selecting the class with the highest probability. The corresponding class label is retrieved from the predefined class labels list. The predicted class label is returned as the output and displayed on the Dash application (Figure 4.2.15).



Container	Process	File Path	Output
user_2	extract_annotations	/home/reynel1995/Thesis/Participant2/1_6-6ST24990mDC.mrxs	• Annotations have been saved in the user's specified folder.
user_2	classify_image	/home/reynel1995/Thesis/Participant2/images/	• The prediction of the model for the image 1 given is Palaeoperidinium pyrophorum.

Figure 4.2.15: The classification output displayed on the Dash app

The classification process can be repeated for multiple images or applied to a folder containing multiple image files.

The training subsection describes the process of building and training the CNN model using data augmentation and normalization techniques. The classification subsection explains how the pre-trained model is utilized to classify individual images by resizing, preprocessing, and predicting the class label. The processes enable the automated classification of digital palynological images, facilitating efficient analysis and interpretation.

4.2.6 Challenges and solutions

Implementing object detection and classification using the watershed algorithm and the CNN presented its challenges. The complexity and variation in the morphological features of pollen grains often resulted in overlapping regions in the watershed segmentation. The CNN model also required considerable computational resources and time for training, considering the extensive range of pollen types.

To handle the problem of overlapping regions in the watershed segmentation, a series of morphological operations such as dilation and erosion were applied to separate the overlapping regions. Also, noise filtering techniques were applied to eliminate any unwanted artifacts from the image.

This study leveraged transfer learning, using pre-trained weights, during the training of the Mask RCNN network for microfossil detection. Training the model was computationally expensive and was prone to compatibility issues. For this reason, only the trained CNN model was deployed for deep-learning tasks on the computational workflow.

The thesis work (Nesse, 2020) presented an extensive comparative classification of dinoflagellates on palynological slides with eight pre-trained CNN networks, leveraging transfer learning. However, the present research focuses on building the implemented computational workflow for palynological slide image analysis.

While the challenges were substantial, the adopted solutions successfully handled the aforementioned issues. The result was an effective and efficient system that can accurately detect and classify dinoflagellates from digital palynological slides, demonstrating the robustness of the implemented methodology.

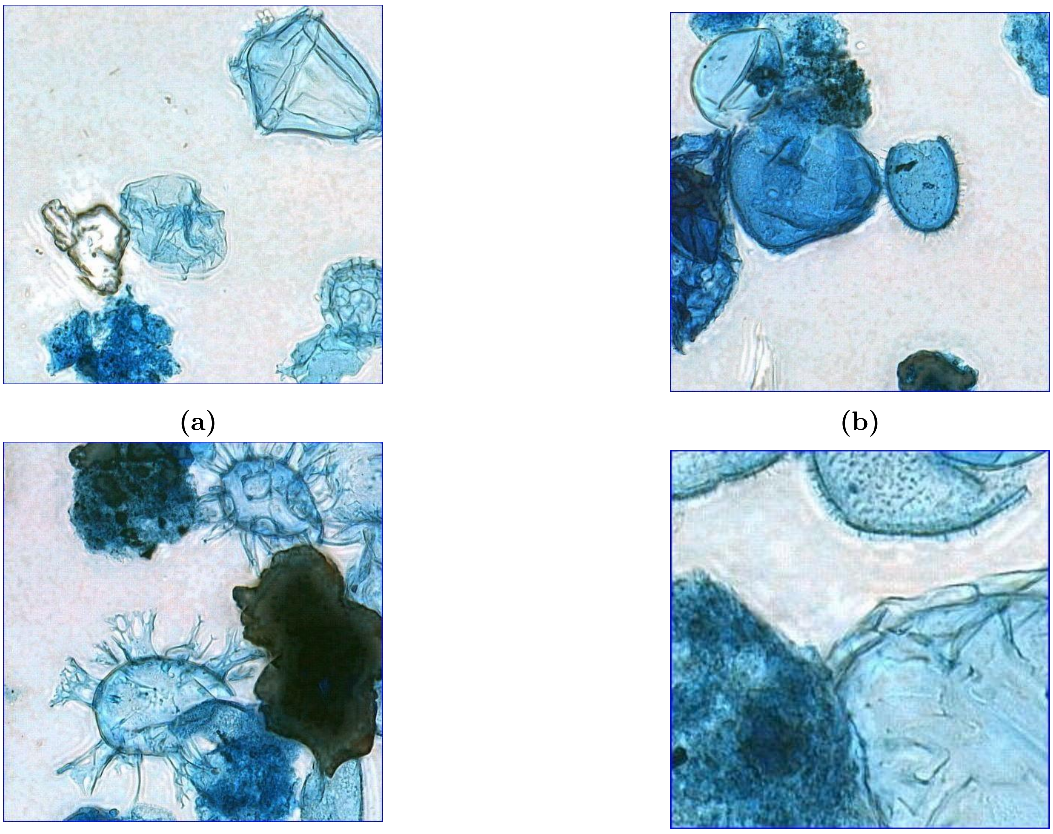
CHAPTER

FIVE

RESULTS AND DISCUSSION

5.1 Results

The results from watershed segmentation, object detection with Mask RCNN, classification task with the trained CNN model, and implemented federated computational workflow are presented in this section.



(a)

(b)

(c)

(d)

Figure 5.1.1: Watershed segmentation.

5.1.1 Implemented computational workflow

The implementation of a federated computation workflow for the analysis of large-scale digital palynological slides has proven to be successful. The developed workflow addresses the challenges posed by the size of the .mrxs files and the computational demands of the image processing tasks. By adopting a distributed computing framework, the computation logic was shifted to the data location, optimizing the utilization of local computational capacity and reducing the need for data movement.

The workflow as is shown in figure 5.1.2 begins with user registration via a Dash interface, where users input their username, the path of the .mrxs file to be processed, and the selected process. This information is stored in "participants.txt" for user management and in "file paths.txt" for easy access to the unprocessed slides. The Directed Acyclic Graph (DAG) running in Airflow regularly checks for new participants every 5 minutes, triggering the necessary processes.

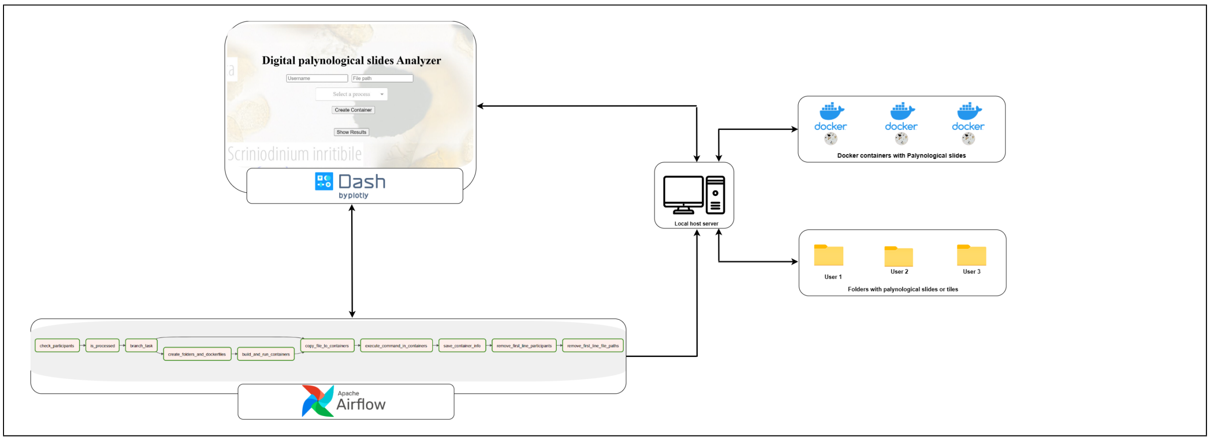


Figure 5.1.2: The implemented computational workflow.

The technology stack used in the implementation includes Dash, Airflow, and Docker. Dash provides a user-friendly interface for registration and result visualization. Airflow handles workflow management, ensuring the sequential execution of the necessary steps. Docker containerizes the processes, allowing for easy replication and scalability. Each component of the DAG performs specific tasks, such as creating Dockerfiles, building Docker images, running containers, sending PY files, and storing the process outputs.

The distributed nature of the workflow effectively addresses the challenges associated with handling large palynological slides. By processing the data within the Docker containers, data transfers are minimized, reducing processing time and improving security. The results displayed in the Dash application mainly consist of metadata from the slide analysis. The detected objects are counted and classified using a trained Convolutional Neural Network (CNN), providing valuable information to users.

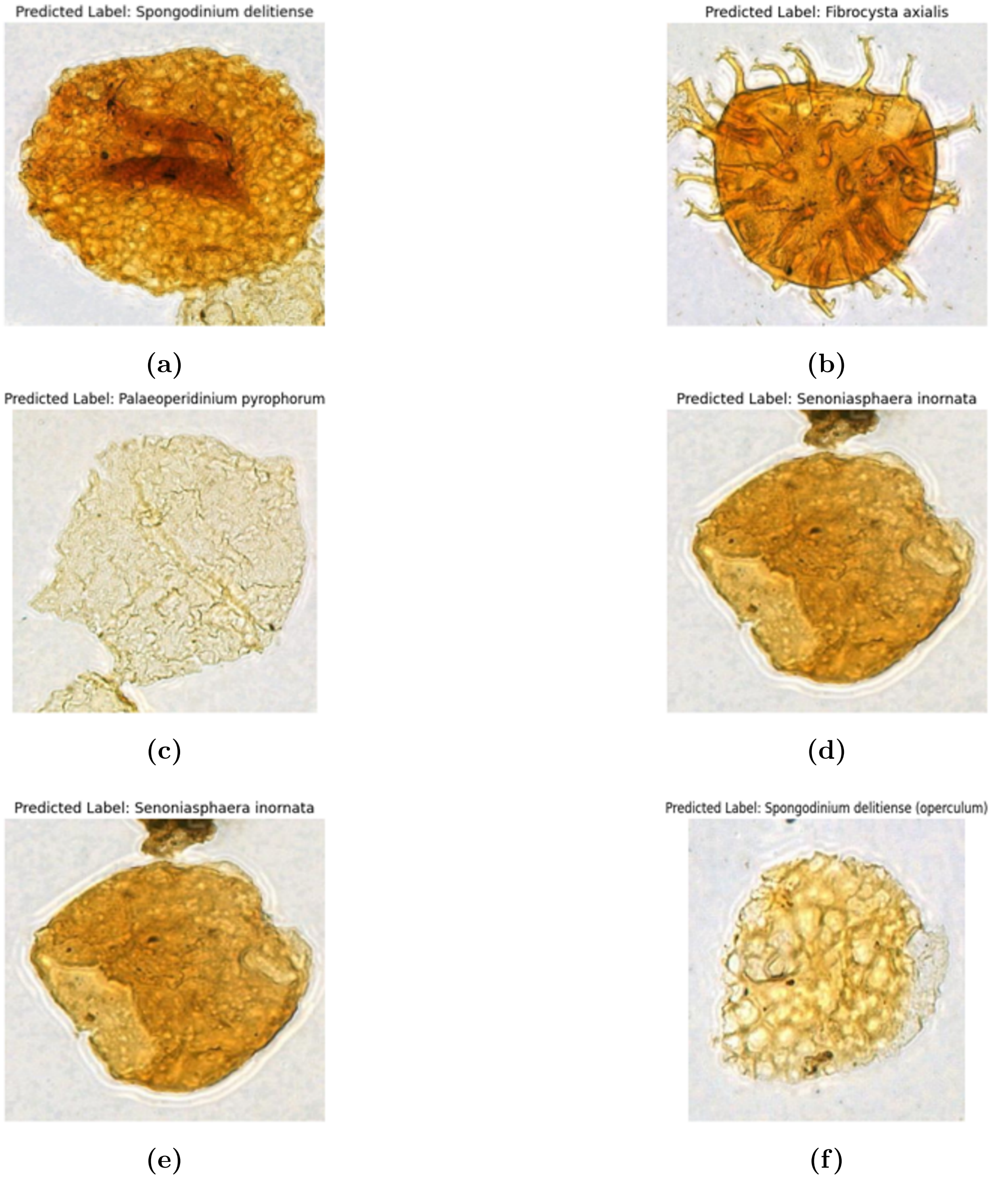
5.1.2 Palynological image analysis

5.1.2.1 Watershed segmentation

The Python code $classify\_microfossils$ presented in the Appendix implemented the watershed segmentation algorithm applied to selected tiled images during the preliminary analysis. Figure 5.1.1 presented the results, showing dinoflagellates on the tiled images correctly segmented from the foreground.

5.1.2.2 Classification

The datasets presented in Table 3.3.3 were used in training and validating the classification model and evaluating the model performance.
Figure 5.1.3 shows correctly predicted dinoflagellate classes, while Figure 5.1.4 shows two examples of dinoflagellates the classifier predicted incorrectly.


Predicted Label: Spongodinium delitiense
Predicted Label: Fibrocysta axialis
Predicted Label: Palaeoperidinium pyrophorum
Predicted Label: Senoniasphaera inornata
Predicted Label: Senoniasphaera inornata
Predicted Label: Spongodinium delitiense (operculum)

(a) (b) (c) (d) (e) (f)

Figure 5.1.3: Correct dinoflagellate class predictions.

Predicted Label: Senoniasphaera inornata
Predicted Label: Spongodinium delitiense

(a) (b)

Figure 5.1.4: Incorrect dinoflagellate class predictions.

Figure 5.1.5 shows accuracy and loss achieved during the model training and validation after 17 epochs.

Figure 5.1.6 presents the confusion matrix for the model evaluation. The matrix summarizes the performance of the classifier, showing at a glance correct and incorrect classifications. For example, the model correctly classified Cribroperidinium "prominoseptatum", Palaeoperidinium pyrophorum, and Senoniasphaera inornata dinoflagellate classes. For Fibrocysta axialis, Spongodinium delitiense, and Spongodinium delitiense (operculum) classes, the numbers of incorrect classification are three, one, and one, respectively.

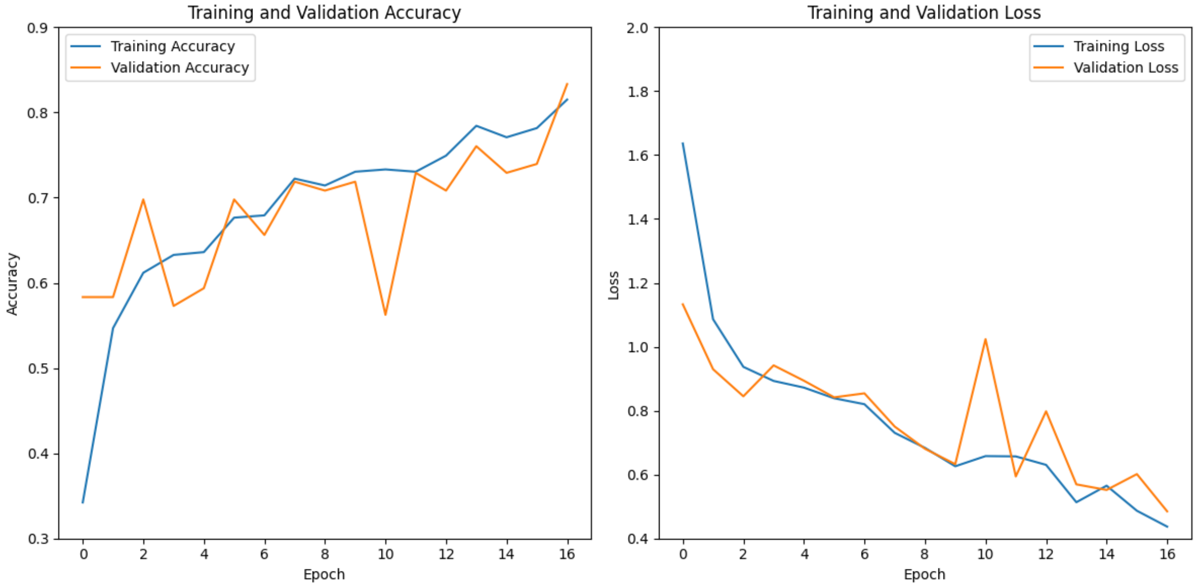


Figure 5.1.5: Training/validation loss and accuracy.

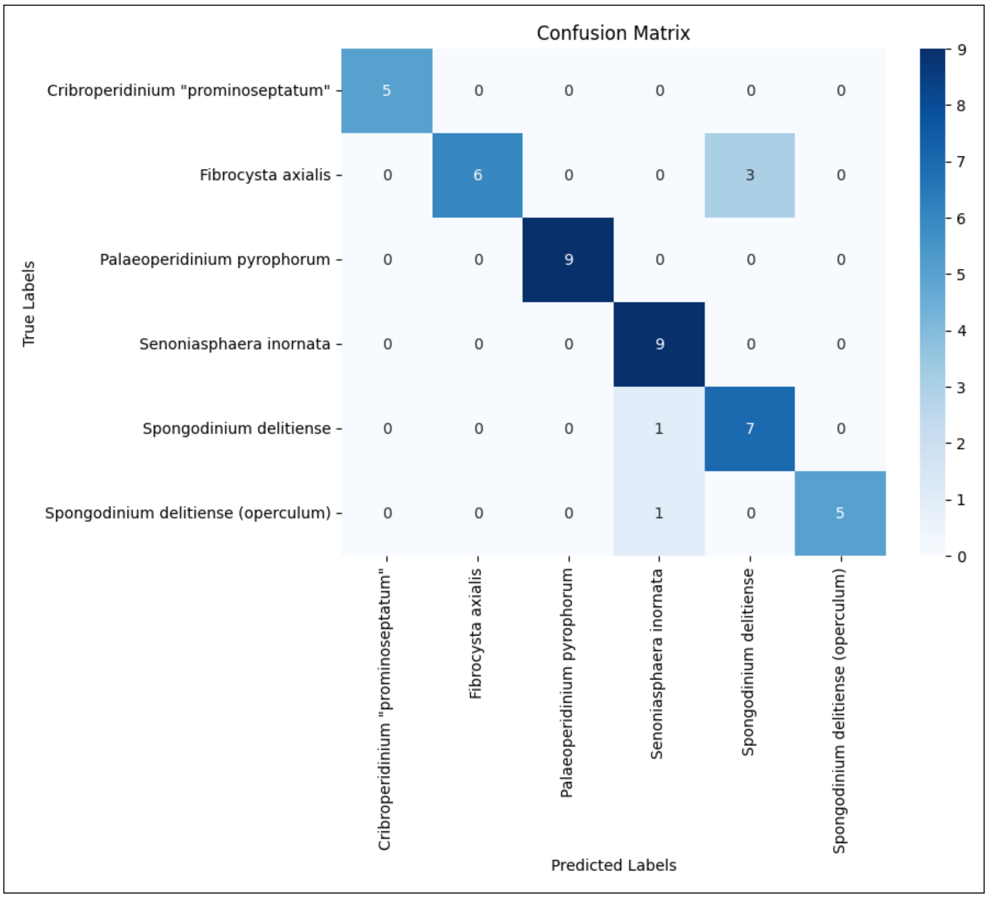
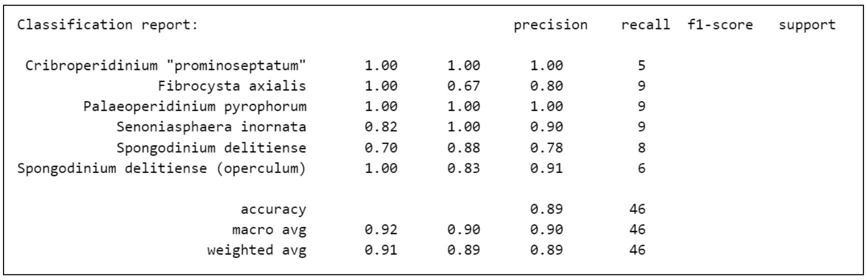


Figure 5.1.6: The confusion matrix showing correct and incorrect predictions.

Figure 5.1.7 presents a summary of evaluation metrics extracted from the confusion matrix data. Accuracy, precision, recall, and f1-score are evaluation metrics used to assess the performance of the trained deep-learning model.



```
Classification report:                                        precision    recall  f1-score   support

 Cribroperidinium "prominoseptatum"       1.00      1.00      1.00         5
                 Fibrocysta axialis       1.00      0.67      0.80         9
        Palaeoperidinium pyrophorum       1.00      1.00      1.00         9
            Senoniasphaera inornata       0.82      1.00      0.90         9
            Spongodinium delitiense       0.70      0.88      0.78         8
Spongodinium delitiense (operculum)       1.00      0.83      0.91         6

                           accuracy                           0.89        46
                          macro avg       0.92      0.90      0.90        46
                       weighted avg       0.91      0.89      0.89        46
```

Figure 5.1.7: The evaluation metrics for the classification model.

Table 5.1.1: Model Accuracy

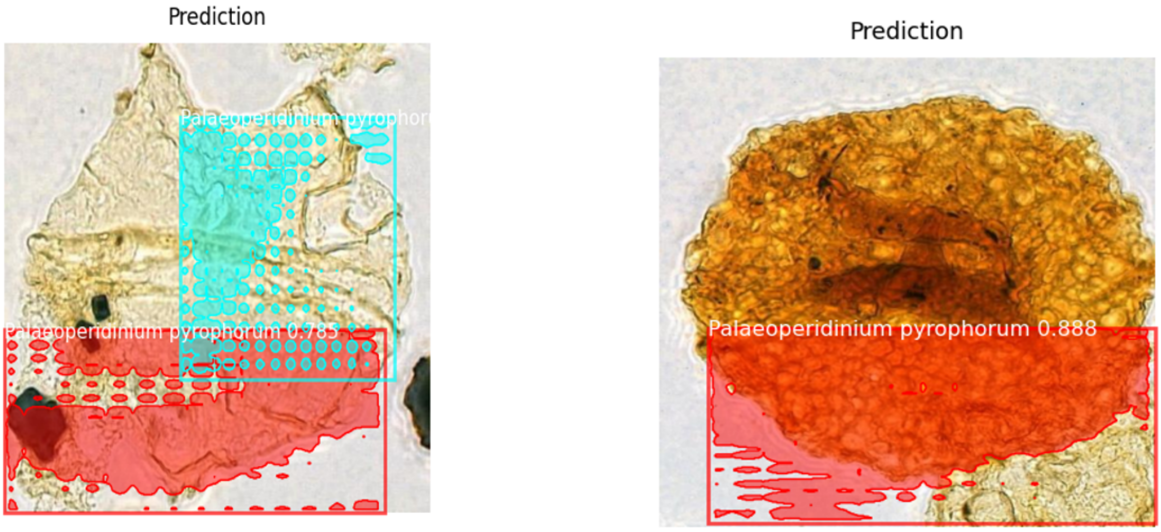
Accuracy	Value
Training	0.82
Validation	0.83
Test	0.89

Table 5.1.1 summarizes the model accuracy achieved during training, validation, and

testing. The results show the trained model generalizes well on unknown data with an accuracy of 0.89, eliminating the problem of overfitting.

5.1.2.3 Dinoflagellate detection

The Mask RCNN algorithm is trained with the dataset presented in Table 3.3.2 and the corresponding annotation detail of each microfossil record. Figure 5.1.8 presents the results of the microfossil identification task. The results show the deep-learning model correctly identified the microfossil class Palaeoperidinium pyrophorum (Figure 5.1.8a). However, the model failed to identify Spongodinium delitiense correctly, predicting the class as Palaeoperidinium pyrophorum (Figure 5.1.8b).


Prediction
Palaeoperidinium pyrophor
Palaeoperidinium pyrophorum 0.785
Prediction
Palaeoperidinium pyrophorum 0.888

(a)

(b)

Figure 5.1.8: Dinoflagellate identification with the Mask RCNN model.

5.2 Discussion

The implementation of the federated computation workflow for the analysis of large-scale digital palynological slides has proven to be successful. The developed workflow addresses the challenges posed by the size of the MRXS files and the computational demands of the image processing tasks. By adopting a distributed computing framework, the computation logic is shifted to the data location, optimizing the utilization of local computational capacity and reducing the need for data movement.

The workflow begins with user registration via a Dash interface, where users input their username, the MRXS file path to be processed, and the selected process. This information is stored in "participants.txt" for user management and in "file paths.txt" for easy access to the unprocessed slides. The Directed Acyclic Graph (DAG) running in Airflow regularly checks for new participants every 5 minutes, triggering the necessary processes.

The technology stack used in the implementation includes Dash, Airflow, and Docker. Dash provides a user-friendly interface for registration and result visualization. Airflow handles the workflow management, ensuring the sequential execution of the necessary steps. Docker is utilized to containerize the processes, allowing for easy replication and scalability. Each component of the DAG performs specific tasks, such as creating Dockerfiles, building Docker images, running containers, sending PY files, and storing the output of the processes.

The distributed nature of the workflow effectively addresses the challenges associated with handling large palynological slide images. By processing the data within the Docker containers, data transfers are minimized, reducing processing time and improving security. The results displayed in the Dash application mainly consist of metadata from the slide analysis. The workflow is designed to perform watershed segmentation, dinoflagellate identification, and classification task with trained deep-learning models such as Mask RCNN and CNN respectively, providing valuable information to users.

The performance of the trained deep-learning models in identifying and classifying microfossils has been discussed. The CNN model achieved an accuracy of 0.82 and 0.89 for training and performance evaluation, respectively. The results are satisfactory.
The dinoflagellate detection with Mask RCNN is the most challenging task implemented in this work. Because the model performance needs improvement, dinoflagellate detection with the trained Mask RCNN is not included in the implemented computational workflow. However, the results are presented in the section above.

The quality of the results obtained through the federated computation workflow has been evaluated in comparison to traditional or manual approaches. The workflow offers faster processing times and provides valuable interpretations of the slide data. The ability to determine possible microfossil types enhances the understanding of the slide's paleontological context.

The novel federated computation workflow presented in this thesis demonstrates its scalability and potential applications in analyzing digital palynological slide images. The ability to process large-scale datasets efficiently and accurately provides valuable insights to researchers. However, further improvements can be made, including implementing secure authentication, expanding the training dataset for the classification model, and developing a public web application to allow for broader accessibility.

5.3 Limitations

Various challenges were encountered during the development and implementation of the computational workflow. These included integrating multiple technologies and managing potential user errors. Robust error-handling mechanisms were implemented in the Dash application to ensure smooth user interactions and prevent invalid inputs.

Limitations and areas for improvement in the system design, performance, and functionality were identified. The sequential order of certain processes and the lack of security measures are among the areas that require attention. Also, the public release of the Dash application and the transformation of other PCs into real servers are aspects to be considered for future enhancements.

Setting up the simulation environment for the microfossil detection and training Mask RCNN were computationally challenging tasks. The deep-learning model was built on old versions of TensorFlow and Keras, with many dependencies, and hence suffered severe compatibility issues. The initial plan was to train and deploy the Mask RCNN model in the implemented computational workflow. Because the results were less satisfactory, considering the time limitation, microfossil classification with the trained CNN model was adopted as an option and was implemented in the workflow.

CHAPTER

SIX

CONCLUSION

A novel federated computational workflow for analyzing palynological slide images is developed and implemented in this thesis. Its scalability, efficiency, and automation give researchers a powerful tool for analyzing and understanding paleontological data. The workflow establishes the foundation for future advancements and new possibilities.

The workflow's contribution to digital palynological slide analysis is significant, offering an automated and scalable approach. The ability to predict the geological age of slides based on their microfossil contents opens up possibilities for stratigraphic column construction. Collaborative efforts involving domain experts for model training and providing more labeled slide images will further improve the accuracy and effectiveness of the workflow.

The novel computation workflow implemented in this thesis demonstrates its potential to revolutionize the analysis of digital palynological slide images, providing a more informed understanding of the geological history of source rocks.
Future research and development should focus on refining the system, addressing the limitations, and exploring potential extensions. The integration of secure authentication mechanisms, the enhancement of model training with a larger dataset, and the transformation of the workflow into a publicly accessible web application are key areas for improvement. Also, incorporating a machine learning model to predict the geological age of source rocks will further enhance the functionality and usefulness of the workflow.

REFERENCES

3DHistech. (2020). The digital pathology company [Last accessed 08 July 2023]. https://www.3dhistech.com/. (Cit. on p. 17)

Abudayyeh, O., Cai, H., Fenves, S., Law, K., O'Neill, R., & Rasdorf, W. (2004). Assessment of the computing component of civil engineering education. *J. Comput. Civ. Eng.*, *18*(3), 187–195. https://doi.org/10.1061/(asce)0887-3801(2004)18:3(187) (cit. on p. 9)

Afgan, E., Coraor, N., Chilton, J., Baker, D., & Taylor, J. (2015). Enabling cloud bursting for life sciences within galaxy. *Concurrency Computat.: Pract. Exper.*, *27*(16), 4330–4343. https://doi.org/10.1002/cpe.3536 (cit. on p. 3)

Alazzam, M., Alassery, F., & Almulihi, A. (2022). Federated deep learning approaches for the privacy and security of iot systems. *Wireless Communications and Mobile Computing*, 1–7. https://doi.org/10.1155/2022/1522179 (cit. on pp. 10, 11)

Al-Dhuraibi, Y., Paraiso, F., Djarallah, N., & Merle, P. (2017). Autonomic vertical elasticity of docker containers with elasticdocker. https://doi.org/10.1109/cloud.2017.67 (cit. on p. 14)

Ali, M., Anjum, A., Yaseen, M., Zamani, A., Balouek-Thomert, D., Rana, O., ..., & Parashar, M. (2018). Edge enhanced deep learning system for large-scale video stream analytics. https://doi.org/10.1109/cfec.2018.8358733 (cit. on p. 10)

Al-Janabi, S., Huisman, A., & Diest, P. (2011). Digital pathology: Current status and future perspectives. *Histopathology*, *61*(1), 1–9. https://doi.org/10.1111/j.1365-2559.2011.03814.x (cit. on p. 8)

Alston, J., & Rick, J. (2020). A beginner's guide to conducting reproducible research. https://doi.org/10.32942/osf.io/h5r6n (cit. on pp. 13, 14)

Anveden, I., & Meding, B. (2007). Skin exposure in geriatric care ? a comparison between observation and self-assessment of exposure. *Contact Dermatitis*, *57*(4), 253–258. https://doi.org/10.1111/j.1600-0536.2007.01211.x (cit. on p. 10)

Bagnasco, S., Colamaria, F., Colella, D., Casula, E., Elia, D., Franco, A., ..., & Vino, G. (2015). Interoperating cloud-based virtual farms. *J. Phys.: Conf. Ser.*, *664*(2), 022033. https://doi.org/10.1088/1742-6596/664/2/022033 (cit. on p. 11)

Bedhief, I., Kassar, M., & Aguili, T. (2022). Empowering sdn-docker based architecture for internet of things heterogeneity. *J Netw Syst Manage*, *31*(1). https://doi.org/10.1007/s10922-022-09702-3 (cit. on p. 14)

Bochkovskiy, A., Wang, C.-Y., & Liao, H.-Y. M. (2020). Yolov4: Optimal speed and accuracy of object detection. *ArXiv*, *abs/2004.10934* (cit. on p. 20).

Boettiger, C. (2015). An introduction to docker for reproducible research. *SIGOPS Oper. Syst. Rev.*, *49*(1), 71–79. https://doi.org/10.1145/2723872.2723882 (cit. on p. 13)

Brendan, M., Eider, M., Daniel, R., Seth, H., & y, A. (2016). Communication-efficient learning of deep networks from decentralized data. https://doi.org/10.48550/arxiv.1602.05629 (cit. on p. 12)

BroadbandSearch. (Accessed: July 12, 2023). Average internet speed around the world. (Cit. on p. 9).

Cao, M., Cao, B., Hong, W., Peng, M., & Bai, X. (2021). Dag-fl: Direct acyclic graph-based blockchain empowers on-device federated learning. https://doi.org/10.1109/icc42927.2021.9500737 (cit. on p. 12)

Chandriani, S., & Ganem, D. (2010). Array-based transcript profiling and limiting dilution reverse transcription-pcr analysis identify additional latent genes in kaposi's sarcoma associated herpesvirus. *J Virol*, *84*(11), 5565–5573. https://doi.org/10.1128/jvi.02723-09 (cit. on p. 10)

Chen, L., Xu, J., Ren, S., & Zhou, P. (2018). Spatio–temporal edge service placement: A bandit learning approach. *IEEE Trans. Wireless Commun.*, *17*(12), 8388–8401. https://doi.org/10.1109/twc.2018.2876823 (cit. on p. 13)

Chen, S. (2023). A federated learning-based civil aviation passenger value analysis method and maas construction considerations in the epidemic background. https://doi.org/10.5772/intechopen.107115 (cit. on p. 11)

Cordeiro, R., Faloutsos, C., & Júnior, C. (2013). Data mining in large sets of complex data. https://doi.org/10.1007/978-1-4471-4890-6 (cit. on p. 3)

Cossio, P., Rohr, D., Baruffa, F., Rampp, M., Lindenstruth, V., & Hummer, G. (2017). Bioem: Gpu-accelerated computing of bayesian inference of electron microscopy images. *Computer Physics Communications*, (210), 163–171. https://doi.org/10.1016/j.cpc.2016.09.014 (cit. on p. 9)

DigitalSreeni. (2022). *Processing whole slide images as tiles*. YouTube. https://www.youtube.com/watch?v=tNfcvgPKgyU. (Cit. on p. 19)

Expedient. (Accessed: July 12, 2023). File transfer time calculator. (Cit. on p. 9).

Gad, A. (2021). Mask r-cnn for object detection and segmentation using tensorflow 2.0 [Last accessed 10 July 2023]. https://github.com/ahmedfgad/Mask-RCNN-TF2b. (Cit. on p. 12)

Gao, H. (2020). Super-resolution and denoising of fluid flow using physics-informed convolutional neural networks without high-resolution labels. https://doi.org/10.48550/arxiv.2011.02364 (cit. on p. 3)

Gemperline, E., Keller, C., & Li, L. (2016). Mass spectrometry in plant-omics. *Anal. Chem.*, *88*(7), 3422–3434. https://doi.org/10.1021/acs.analchem.5b02938 (cit. on p. 3)

George, S., Milea, A., & Shaw, P. (2012). Proliferation in the normal fte is a hallmark of the follicular phase, not brca mutation status. *Clinical Cancer Research*, *18*(22), 6199–6207. https://doi.org/10.1158/1078-0432.ccr-12-2155 (cit. on p. 10)

Górka, P., Pietrzak, P., Kotunia, A., Zabielski, R., & Kowalski, Z. (2014). Effect of method of delivery of sodium butyrate on maturation of the small intestine in newborn calves. *Journal of Dairy Science*, *97*(2), 1026–1035. https://doi.org/10.3168/jds.2013-7251 (cit. on p. 10)

Gowri, A. (2019). Impact of virtualization technologies in the development and management of cloud applications. *ijisae*, *7*(2), 104–110. https://doi.org/10.18201/ijisae.2019252789 (cit. on p. 14)

Guo, W., Alham, N., Liu, Y., Li, M., & Qi, M. (2015). A resource aware mapreduce based parallel svm for large scale image classifications. *Neural Process Lett*, *44*(1), 161–184. https://doi.org/10.1007/s11063-015-9472-z (cit. on p. 3)

Gutierrez, E., Romero, S., Trenas, M., & Zapata, E. (2008). Parallel quantum computer simulation on the cuda architecture. *Proceedings of the International Conference on Computational Science*, 700–709. https://doi.org/10.1007/978-3-540-69384-0_75 (cit. on p. 3)

Hanussek, M., Bartusch, F., & Krüger, J. (2021). Performance and scaling behavior of bioinformatic applications in virtualization environments to create awareness for the efficient use of compute resources. *PLoS Comput Biol*, *17*(7), e1009244. https://doi.org/10.1371/journal.pcbi.1009244 (cit. on p. 14)

Haque, M., Iwaya, L., & Babar, M. (2020). Challenges in docker development. https://doi.org/10.1145/3382494.3410693 (cit. on p. 14)

Holt, K. A., & Bennett, K. D. (2014). Principles and methods for automated palynology. *New Phytologist*, *203*, 735–742. https://doi.org/10.1111/nph.12848 (cit. on pp. 6, 7)

Hunt, C., el-Rishi, H., & Hassan, A. (2002). Reconnaissance investigation of the palynology of holocene wadi deposits in cyrenaica, libya. *Libyan stud.*, (33), 1–7. https://doi.org/10.1017/s0263718900005070 (cit. on p. 10)

Jarzen, D. M. (2022). Palynology [Last accessed 07 July 2022]. https://www.floridamuseum.ufl.edu/paleobotany/palynology/. (Cit. on p. 6)

Johnson, T., Battison, L., Garwood, R., Hickman-Lewis, K., & Brasier, M. (2016). Advanced analytical techniques for studying the morphology and chemistry of proterozoic microfossils. *SP*, *448*, 81–104. https://doi.org/10.1144/sp448.4 (cit. on p. 2)

Keskin, T., & Ince, G. (2021). An ensemble learning approach for energy demand forecasting in microgrids using fog computing, 170–178. https://doi.org/10.1007/978-3-030-85577-2_20 (cit. on p. 14)

Kim, Y., Donovan, R., Ren, Y., Bian, S., Wu, T., Purawat, S., ..., & Li, G. (2022). Smart connected worker edge platform for smart manufacturing: Part 1—architecture and platform design. *J Adv Manuf Process*, *4*(4). https://doi.org/10.1002/amp2.10129 (cit. on p. 13)

Kimovski, D., Ristov, S., Matha, R., & Prodan, R. (2018). Multi-objective service oriented network provisioning in ultra-scale systems, 529–540. https://doi.org/10.1007/978-3-319-75178-8_43 (cit. on p. 12)

Kong, Q., Cao, Y., Iqbal, T., Wang, Y., Wang, W., & Plumbley, M. (2020). Panns: Large-scale pretrained audio neural networks for audio pattern recognition. *IEEE/ACM Trans. Audio Speech Lang. Process.*, *28*, 2880–2894. https://doi.org/10.1109/taslp.2020.3030497 (cit. on p. 3)

Lamani, X., Horst, S., Zimmermann, T., & Schmidt, T. (2014). Determination of aromatic amines in human urine using comprehensive multi-dimensional gas chromatography mass spectrometry (gcxgc-qms). *Anal Bioanal Chem*, *407*(1), 241–252. https://doi.org/10.1007/s00216-014-8080-5 (cit. on p. 3)

Lee, S., Serre, M., Donkelaar, A., Martin, R., Burnett, R., & Jerrett, M. (2012). Comparison of geostatistical interpolation and remote sensing techniques for estimating long-term exposure to ambient pm 2.5 concentrations across the continental united states. *Environ Health Perspect*, *120*(12), 1727–1732. https://doi.org/10.1289/ehp.1205006 (cit. on p. 11)

Li, N., Zeng, L., He, Q., & Shi, Z. (2013). Parallel implementation of apriori algorithm based on mapreduce. *IJNDC*, *2*(1), 89. https://doi.org/10.2991/ijndc.2013.1.2.3 (cit. on p. 9)

Li, W., Mikailov, M., & Chen, W. (2023). Scaling the inference of digital pathology deep learning models using cpu-based high-performance computing. *IEEE Trans. Artif. Intell.*, 1–15. https://doi.org/10.1109/tai.2023.3246032 (cit. on p. 9)

Li, Z., Kihl, M., Lu, Q., & Andersson, J. (2017). Performance overhead comparison between hypervisor and container based virtualization. https://doi.org/10.1109/aina.2017.79 (cit. on pp. 13, 14)

Li, Z., Zhang, X., Müller, H., & Zhang, S. (2018). Large-scale retrieval for medical image analytics: A comprehensive review. *Medical Image Analysis*, (43), 66–84. https://doi.org/10.1016/j.media.2017.09.007 (cit. on pp. 9, 14)

Lindenbaum, P., & Redon, R. (2017). Bioalcidae, samjs and vcffilterjs: Object-oriented formatters and filters for bioinformatics files. *Bioinformatics*, *34*(7), 1224–1225. https://doi.org/10.1093/bioinformatics/btx734 (cit. on p. 13)

Livezey, B., & Zusi, R. (2007). Higher-order phylogeny of modern birds (theropoda, aves: Neornithes) based on comparative anatomy. ii. analysis and discussion. *Zoological Journal of the Linnean Society*, *149*(1), 1–95. https://doi.org/10.1111/j.1096-3642.2006.00293.x (cit. on p. 2)

Mabvakure, B., Rott, R., Dobrowsky, L., Heusden, P., Morris, L., Scheepers, C., & Moore, P. (2019). Advancing hiv vaccine research with low-cost high-performance computing infrastructure: An alternative approach for resource-limited settings. *Bioinform Biol Insights*, (13), 117793221988234. https://doi.org/10.1177/1177932219882347 (cit. on p. 3)

Massaoudi, M., Abu-Rub, H., Refaat, S., Chihi, I., & Oueslati, F. (2021). Deep learning in smart grid technology: A review of recent advancements and future prospects. *IEEE Access*, *9*, 54558–54578. https://doi.org/10.1109/access.2021.3071269 (cit. on p. 14)

Mohammad, N., Yusof, M., Ahmad, R., & Muad, A. (2020). Region-based segmentation and classification of mandibular first molar tooth based on demirjian's method. *J. Phys.: Conf. Ser.*, *1502*(1), 012046. https://doi.org/10.1088/1742-6596/1502/1/012046 (cit. on p. 2)

Mohammadi, J., & Thornburg, J. (2020). Connecting distributed pockets of energyflexibility through federated computations:limitations and possibilities. https://doi.org/10.48550/arxiv.2009.10182 (cit. on p. 10)

Nesse, A. B. (2020). *Classifying dinoflagellates in palynological slides using convolutional neural networks* [A master thesis at the Department of Information Technology - Automation and Signal Processing]. https://uis.brage.unit.no/uis-xmlui/handle/11250/2680067. (Cit. on pp. 12, 46)

Ongie, G., & Jacob, M. (2017). A fast algorithm for convolutional structured low-rank matrix recovery. *IEEE Trans. Comput. Imaging*, *4*(3), 535–550. https://doi.org/10.1109/tci.2017.2721819 (cit. on p. 9)

OpenCV. (2023). Image segmentation with watershed algorithm [Last accessed 13 July 2023]. https://docs.opencv.org/4.x/d3/db4/tutorial_py_watershed.html. (Cit. on pp. 12, 35)

Oppenheimer, D., Chun, B., Patterson, D., Snoeren, A., & Vahdat, A. (2005). Service placement in shared wide-area platforms. https://doi.org/10.1145/1095810.1118581 (cit. on p. 10)

Petersen, K., Fuchs, M., Moreshet, S., Cohen, Y., & Sinoquet, H. (1992). Computing transpiration of sunlit and shaded cotton foliage under variable water stress. *Agronomy Journal*, *84*(1), 91–97 (cit. on p. 3).

Punyasena, S., Haselhorst, D., Kong, S., Fowlkes, C., & Moreno, J. (2022). Automated identification of diverse neotropical pollen samples using convolutional neural networks. *Methods Ecol Evol*, *13*(9), 2049–2064. https://doi.org/10.1111/2041-210x.13917 (cit. on p. 2)

PyPI. (2022). Openslide python [Last accessed 10 July 2023]. https://pypi.org/project/openslide-python/. (Cit. on p. 19)

Robidas, R., & Legault, C. (2022). Calcus: An open-source quantum chemistry web platform. *J. Chem. Inf. Model.*, *62*(5), 1147–1153. https://doi.org/10.1021/acs.jcim.1c01502 (cit. on p. 13)

Saito, K., Takato, M., Sekine, Y., & Uchikoba, F. (2012). Biomimetics micro robot with active hardware neural networks locomotion control and insect-like switching behaviour. *International Journal of Advanced Robotic Systems*, *5*(9), 226. https://doi.org/10.5772/54129 (cit. on p. 10)

Sauvanaud, C., Dholakia, A., Guitart, J., Kim, C., & Mayes, P. (2020). Big data deployment in containerized infrastructures through the interconnection of network namespaces. *Softw Pract Exper*, *50*(7), 1087–1113. https://doi.org/10.1002/spe.2793 (cit. on p. 14)

Sha, W., Guo, Y., Yuan, Q., Tang, S., Zhang, X., Lu, S., & Cheng, S. (2020). Artificial intelligence to power the future of materials science and engineering. *Advanced Intelligent Systems*, *4*(2), 1900143. https://doi.org/10.1002/aisy.201900143 (cit. on p. 3)

Sherman, T., Gao, T., & Fertig, E. (2019). Cogaps 3: Bayesian non-negative matrix factorization for single-cell analysis with asynchronous updates and sparse data structures. https://doi.org/10.1101/699041 (cit. on p. 9)

Sherman, T., Gao, T., & Fertig, E. (2020). Cogaps 3: Bayesian non-negative matrix factorization for single-cell analysis with asynchronous updates and sparse data structures. *BMC Bioinformatics*, *21*(1). https://doi.org/10.1186/s12859-020-03796-9 (cit. on p. 10)

Shumilovskikh, L., O'Keefe, J., & Marret, F. (2021). An overview of the taxonomic groups of non-pollen palynomorphs. *SP*, *511*(1), 13–61. https://doi.org/10.1144/sp511-2020-65 (cit. on p. 8)

Shuvo, M., Islam, S., Cheng, J., & Morshed, B. (2023). Efficient acceleration of deep learning inference on resource-constrained edge devices: A review. *Proc. IEEE*, *111*(1), 42–91. https://doi.org/10.1109/jproc.2022.3226481 (cit. on p. 11)

Siegel, H., Abraham, S., Bain, W., Batcher, K., Casavant, T., DeGroot, D., ..., & Wah, B. (1992). Report of the purdue workshop on grand challenges in computer architecture for the support of high performance computing. *Journal of Parallel and Distributed Computing*, *16*(3), 199–211. https://doi.org/10.1016/0743-7315(92)90033-j (cit. on p. 9)

Silva, C., Dainys, J., Simmons, S., Vienožinskis, V., & Audzijonyte, A. (2022). A scalable open-source framework for machine learning-based image collection, annotation and classification: A case study for automatic fish species identification. *Sustainability*, *21*(14), 14324. https://doi.org/10.3390/su142114324 (cit. on p. 9)

Stefanowicz, S. (2023). Detection, identification and clustering of palynomorphs using ai and machine learning. https://doi.org/10.5194/egusphere-egu23-14198 (cit. on p. 8)

Stillman, E. C., & Flenley, J. R. (1995). The needs and prospects for automation in palynology. *Unknown Journal*, *Unknown Volume*, Unknown Pages (cit. on p. 6).

The NPD. (2023). New diskos data type: Multi-gigapixel palynology slides [Last accessed 08 July 2023]. https://www.npd.no/en/news/general-news/2023/new-diskos-data-type-multi-gigapixel-palynology-slides/. (Cit. on pp. 8, 16)

Tommasini, R. (2021). Velocity on the web, 85–94. https://doi.org/10.1007/978-3-030-62476-7_8 (cit. on p. 14)

Venugopal, M. (2017). Containerized microservices architecture. *ijecs*, *11*(6). https://doi.org/10.18535/ijecs/v6i11.20 (cit. on p. 13)

Williams, I., Mills, S., Fox, C., Pfleiderer, E., & Mogilka, H. (2002). The relationship between air traffic control communication events and measures of controller taskload and workload. *Air Traffic Control Quarterly*, *10*(2), 69–83. https://doi.org/10.2514/atcq.10.2.69 (cit. on p. 10)

Williams, R. (2023). *Nature's microplastics*. Retrieved April 2, 2023, from https://geoexpro.com/natures-microplastics/. (Cit. on pp. 7, 8)

Wright, A., Smith, D., Dhurandhar, B., Fairley, T., Scheiber-Pacht, M., Chakraborty, S., ..., & Coffey, D. (2012). Digital slide imaging in cervicovaginal cytology: A pilot study. *Archives of Pathology & Laboratory Medicine*, *137*(5), 618–624. https://doi.org/10.5858/arpa.2012-0430-oa (cit. on p. 8)

Wu, Y., Zhang, Y., Wang, T., & Wang, H. (2020). Characterizing the occurrence of dockerfile smells in open-source software: An empirical study. *IEEE Access*, (8), 34127–34139. https://doi.org/10.1109/access.2020.2973750 (cit. on p. 13)

Xiaofeng, L., Yinchuan, L., Yunfeng, S., & Qing, W. (2022a). Sparse federated learning with hierarchical personalized models. https://doi.org/10.48550/arxiv.2203.13517 (cit. on p. 11)

Xiaofeng, L., Yinchuan, L., Yunfeng, S., & Qing, W. (2022b). Sparse federated learning with hierarchical personalized models. https://doi.org/10.48550/arxiv.2203.13517 (cit. on p. 11)

Xu, Y., Li, Y., Shen, Z., Wu, Z., Gao, T., Fan, Y., ..., & Chang, E. (2017). Parallel multiple instance learning for extremely large histopathology image analysis. *BMC Bioinformatics*, *18*(1). https://doi.org/10.1186/s12859-017-1768-8 (cit. on p. 9)

Yamanaka, H., Teranishi, Y., Kawai, E., Nagano, H., & Harai, H. (2022). Design and implementation of an edge computing testbed to simplify experimental environment setup. *IEICE Trans. Inf. Syst.*, *E105.D*(9), 1516–1528. https://doi.org/10.1587/transinf.2022edk0003 (cit. on p. 14)

Ye, D., Yu, R., Pan, M., & Han, Z. (2020). Federated learning in vehicular edge computing: A selective model aggregation approach. *IEEE Access*, (8), 23920–23935. https://doi.org/10.1109/access.2020.2968399 (cit. on p. 10)

Zhang, K., Wang, K., Yuan, Y., Guo, L., Lee, R., & Zhang, X. (2015). Mega-kv. *Proc. VLDB Endow.*, *8*(11), 1226–1237. https://doi.org/10.14778/2809974.2809984 (cit. on p. 3)

Zhang, Y., Zhang, X., Rabbani, Z., Jackson, I., & Vujaskovic, Z. (2012). Oxidative stress mediates radiation lung injury by inducing apoptosis. *International Journal of Radiation Oncology*Biology*Physics*, *83*(2), 740–748. https://doi.org/10.1016/j.ijrobp.2011.08.005 (cit. on p. 10)

APPENDIX

DAG .PY FILE

1. **dag:** Here we define the arguments of the dag to set up the dag to run every five minutes

2. **check_participants:** It receives as input the participant file path, and the it checks it, if it is empty then it returns an empty list, so the workflow understands that there is not participant waiting to be processed.

3. **is_participant_processed:**receives as input the username and the process selected by the user, so it can check if it was already processed.

4. **decide_next_task:** This function actually decided which way to take if should the workflow create a new docker container or should just copy the files. It receives the output from the is_participant_processed function, if it positive, then it would just copy the files, otherwise would create a container.

5. **create_folders_and_dockerfiles:** This function creates the docker files on the folder path given by the user, depends on what the file path ends with, then it would change the structure of the dockerfile.

6. **build_and_run_containers:**This function takes the username as input and then it will send an execution command to where the dockerfile is so it will spin up this process to create the image and later the container.

7. **copy_file_to_containers:**This function takes the username and process selected as input, an according to the process selected it will copy a .py file different for each selection also the username determine in which container should the .py file be copy in.

8. **execute_command_in_containers:**This function takes the username and process selected as inputs to determine how the execution command should be structured. Also according to each different process selected it takes the output of the execution command and save it in a different way for each different process.

9. **save_container_info:**This functions takes the username and the process selected, to register it as processed.

10. **remove_first_line:**This function takes the file path and participants files and cleans them.

11. **check_participants_task:**This defines the task arguments of check_participants function

12. **is_processed_task:**This defines the task arguments of is_processed function, so we can see that it takes the arguments passed from the first task which is check_participants_task

13. **branch_task:**This defines the task arguments of decide_next_task function

14. **create_folders_and_dockerfiles_task:** This defines the task arguments of create_folders_and_dockerfiles function

15. **build_and_run_containers_task:** This defines the task arguments of build_ and_run_containers function. Here also the username and process selected is taken from the firs task.

16. **copy_file_to_containers_task:** This defines the task arguments of copy_file_ to_containers function. Here also the username and process selected is taken from the firs task.

17. **execute_command_in_containers_task:** This defines the task arguments of execute_command_in_containers function. Here also the username and process selected is taken from the firs task.

18. **save_container_info_task:** This defines the task arguments of save_container _info function. Here also the username and process selected is taken from the firs task.

19. **remove_first_line_participants_task:** This defines the task arguments of remove_first_line function. Here the task takes as argument the participants file, so it cleans it.

20. **remove_first_line_file_paths_task:** This defines the task arguments of remove_first_line function. Here the task takes as argument the file paths file, so it cleans it.

21. **create_files_if_not_exist():** This function checks if two files (participants_file and file_paths_file) exist in the specified paths. If any of the files does not exist, it creates an empty file at the respective path.

APP .PY FILE

1. **show_results():**This function retrieves information from the processed_containe rs_file and generates HTML content to display the results. It reads the file line by line, extracts container, process selection, file path, and response file information. It then generates a set of unique combinations, groups the results by user, and creates a table with the relevant information.

2. **update_output(n_clicks):** This function is a callback function triggered by the show-results button click. It checks the number of clicks and displays the results accordingly. If the number of clicks is odd, it shows the results; otherwise, it displays a message indicating that no results are available.

3. **create_container(n_clicks, file_path, username, process_selection):** Thi s function is a callback function triggered by the crear-container button click. It

handles the creation of a container based on the provided input parameters. It performs various validations, such as checking if the file path and username are not empty, if the file path contains blank spaces, and if a process selection is made. It also checks if the combination of username, process selection, and file path already exists in the processed_containers_file. If the validations pass, it registers the container by adding the username and process selection to the participants_file and the file path to the file_paths_file.

4. **remove_process(n_clicks, username, file_path, process_selection):** This function is a callback function triggered by the remove-process button click. It handles the removal of a process from the processed_containers_file. It reads the file, filters the lines based on the provided username, process selection, and file path, removes the corresponding process-related files and directories, and updates the processed_containers_file by removing the filtered lines.

5. **handle_extraction_slide_button(n_clicks, xml, file_path, username, p rocess_selection):** This function is a callback function triggered by the extract-annotations -button button click. It handles the extraction of annotations from a slide. It validates the provided XML and file path, creates a directory for the user if it doesn't exist, saves the XML path and values in a JSON file, and updates the participants_file and file_paths_file.

6. **handle_clean_slide_button(n_clicks, resolution, zoom, file_path, username, process_selection):**This function is a callback function triggered by the clean-slide-button button click. It handles the cleaning of tiles from a slide. It validates the provided resolution, zoom, and file path, creates a directory for the user if it doesn't exist, saves the resolution and zoom values in a JSON file, and updates the participants_file and file_paths_file.

7. **create_dropdown_with_title(title, dropdown_id, options, value):** This function generates an HTML dropdown component with a title label. It takes a title, dropdown ID, options, and default value as parameters and returns the HTML div containing the title label and the dropdown component.

Read_image .PY file

The read_image function loads an image in the .mrxs format, extract information about zoom levels and tiles, and return this information in JSON format. The function utilizes the openslide library to handle the image loading and deep zoom generation. By providing essential details about the image, such as zoom factors and the total number of tiles, it aids in further analysis and processing of the image data.

Clean_tiles .PY file

The clean_tiles function involves the processing of images in the .mrxs format. The function utilizes the openslide and cv2 libraries to load the image, extract tiles at a specific zoom level, and perform various image processing operations. The goal is to identify and save tiles that meet certain criteria, such as having a mean pixel intensity above a threshold and a low percentage of white pixels. The function saves the selected tiles as individual .jpg images in a specified folder and returns information about the number of processed tiles and the total number of tiles at the given zoom level.

Apply_ Watershed .PY file

The apply_watershed function applies the watershed algorithm to segment and label images in the specified input folder. It reads each image file, converts it to grayscale, and performs various image processing operations such as thresholding, morphological opening, and distance transform to identify the foreground objects (microfossils) from the background. It then applies the watershed algorithm to segment the foreground markers and obtain the labels. The segmented image is visualized by assigning a specific color to the segmented regions. Finally, the segmented images are saved in the output folder, and the function returns the count of the segmented images in JSON format. This function allows for automatic segmentation and labeling of images based on the watershed algorithm, enabling further analysis and interpretation of the segmented data.

Extract Annotations .PY file The extract_labeled_images function processes MRXS and XML files to extract labeled images. It opens the MRXS slide, parses the XML file for annotations, and extracts regions of interest based on the provided polygon points. The function then returns a list of labeled data, where each data entry contains an image and its corresponding label. This function facilitates the extraction of specific image regions annotated in the palyslides, enabling further analysis and utilization of the labeled data.

Split_classes .py file

The script code counts the number of records per class in a dataset and filters out classes with only one record. It then splits the filtered data into training and validation sets, ensuring that both sets have a balanced representation of classes. The code also provides information about the number of training and validation samples, as well as the number of unique class labels in each set.

Training .py file

The build_and_train_cnn function constructs and trains a CNN model for image classification. It takes the training and validation directories, image size, batch size, number of classes, and epochs as inputs. Data augmentation and normalization are applied to the training set, while only rescaling is performed on the validation set. The CNN model consists of convolutional and pooling layers, followed by flattening and dense layers. It is compiled with an optimizer, loss function, and evaluation metric. The model is trained using the data generators created from the directories, and the training history is saved. The trained model and history are stored, and the function returns the training history.

Classify Image .PY file

The resize_image function performs image classification using a pre-trained model. It loads an image file from the specified directory and resizes it to the desired size using OpenCV. The loaded image is then converted to RGB format. The code uses a pre-trained model loaded from an H5 file to predict the class label of the resized image. The model's class indices and corresponding labels are defined in a list. The image is preprocessed by normalizing the pixel values. The preprocessed image is passed through the model to obtain predictions. The predicted class label is retrieved based on the maximum prediction value, and it is returned in JSON format. This code enables automated classification of images based on the pre-trained model, providing the predicted class label for a given image.

Model Training .PY file

The plot_training_history function is used to visualize the training and validation accuracy and loss from a training history. Given a training history object, the function extracts the epochs and metrics such as training loss, training accuracy, validation loss, and validation accuracy. It then plots the accuracy and loss curves on separate subplots using the Matplotlib library. The left subplot shows the training and validation accuracy over epochs, while the right subplot displays the training and validation loss. The function also sets appropriate titles, axis labels, and legends for the plots. Finally, it uses plt.tight_layout() to improve the spacing between the subplots and displays the plot using plt.show(). Overall, this function provides a convenient way to visualize the performance of a model during training.

Model history .PY file

In this script, a saved model and training history are loaded from their respective files. The model is loaded using the load_model function from Keras, and the history is loaded using the pickle module. The training metrics (loss and accuracy) and validation metrics (loss and accuracy) are then accessed from the history object. The code also includes commented-out lines that print the training and validation metrics. Finally, the plot_training_history function is called to visualize the training and validation accuracy and loss curves using the loaded training history.

Model evaluation .PY file

The code loads a pre-trained model and a set of class labels. It then iterates over a directory of test images, resizes each image, preprocesses it, and performs predictions using the loaded model. The true labels and predicted labels are stored in separate lists. The code calculates performance metrics such as accuracy, precision, recall, and F1-score, and displays a confusion matrix and classification report. Additionally, it visualizes the predicted images along with their corresponding predicted labels. Overall, the code enables evaluating the model's performance on the test images and provides insights into its classification accuracy.

DAG

July 10, 2023

[ ]:

```
from airflow import DAG
from airflow.operators.python_operator import PythonOperator,␣
 ↪BranchPythonOperator
from datetime import datetime, timedelta
import os
import docker
import subprocess
import cv2
import numpy as np
import sys
from io import StringIO
import json

default_args = {
    'owner': 'airflow',
    'start_date': datetime(2023, 6, 10),
    'retries': 1,
    'retry_delay': timedelta(minutes=1)
}

dag = DAG('execute_python', default_args=default_args, schedule_interval='*/5 *␣
 ↪* * *')



###########################################



def check_participants(participants_file):
    if not os.path.isfile(participants_file):
        print("Participants file not found.")
        return []  # Returns an empty list if the file doesn't exist."

    participants = []
    with open(participants_file, 'r') as f:
        lines = f.readlines()
        if not lines:  # Check if the file is empty
            print("Participants file is empty.")
            return []  # Return an empty list if the file is empty.

        for line in lines:
            line = line.strip()
            if line:
                username, process = line.split(' - ')
                participants.append((username, process))

    return participants

####################################################

#Check if a participant is already in the processed list.
def is_participant_processed(username, process_selection):
    if not os.path.isfile(processed_containers_file) or os.
 ↪stat(processed_containers_file).st_size == 0:
        return False

    with open(processed_containers_file, 'r') as f:
        for line in f:
            line = line.strip()
            if line:
                stored_username, stored_process_selection, file_path_processed␣
 ↪= line.split(' - ')
                if stored_username == username and stored_process_selection ==␣
 ↪process_selection:
                    return True
    return False
####################################################

# Check if a participant is already in the processed list.
def decide_next_task(**kwargs):
    is_processed = kwargs['ti'].xcom_pull(task_ids='is_processed')
    if is_processed:
        return 'copy_file_to_containers'
    else:
        return 'create_folders_and_dockerfiles'


####################################################

def create_folders_and_dockerfiles():
    # Read the file path
    with open('/home/reynel1995/Thesis/file_paths.txt', 'r') as f:
        file_paths = f.read().splitlines()

    if file_paths:
        file_path = file_paths[0]  # Get the first path

        if file_path.endswith('.jpg') or file_path.endswith('.png'):
                # Check if a participant is already in the processed list.
            linux_file_path = os.path.normpath(file_path).replace("\\", "/").
 ↪replace("C:", "/mnt/c")

            current_dir = os.path.dirname(linux_file_path)

            #  Get the filename without the extension.
            file_name = os.path.splitext(os.path.basename(file_path))[0]
            directory_path = os.path.join(os.path.dirname(file_path), file_name)

            dockerfile_content = f"""
            # Use a base Python image
            FROM base-image

            # Set the working directory to /app
            WORKDIR /app

            # Copy the file to the container in the /app folder
            COPY {os.path.basename(linux_file_path)} /app/


            CMD tail -f /dev/null
            """

            with open(f"{current_dir}/Dockerfile", "w") as dockerfile:
                dockerfile.write(dockerfile_content)

        else:

            #  Get the filename without the extension.
            linux_file_path = os.path.normpath(file_path).replace("\\", "/").
 ↪replace("C:", "/mnt/c")

            current_dir = os.path.dirname(linux_file_path)

            # Get the filename without the extension
            file_name = os.path.splitext(os.path.basename(file_path))[0]
            directory_path = os.path.join(os.path.dirname(file_path), file_name)

            dockerfile_content = f"""
            # Use a base Python image
            FROM base-image

            # Set the working directory to /app
            WORKDIR /app

            # Copy the file to the container in the /app folder
            COPY {os.path.basename(linux_file_path)} /app/
            COPY {os.path.basename(directory_path)} /app/{os.path.
 ↪basename(directory_path)}

            CMD tail -f /dev/null
            """

            with open(f"{current_dir}/Dockerfile", "w") as dockerfile:
                dockerfile.write(dockerfile_content)
    else:
        print("No file paths found.")



#####################################################


def build_and_run_containers(username):
    container_id = username

    # Read the file paths file.
    with open('/home/reynel1995/Thesis/file_paths.txt', 'r') as f:
        file_paths = f.read().splitlines()

    if file_paths:
        file_path = file_paths[0]  # Get the first path

        folder_path = os.path.dirname(file_path)  # Remove the file name from␣
 ↪the path

        try:
            # Build the image from the Dockerfile using the Linux shell.
            build_command = f"docker build -t {container_id} {folder_path}"
            subprocess.run(build_command, shell=True, check=True)

            print(f"Container {container_id} image built successfully.")

            # Execute the container using the docker run command and assign it␣
 ↪the name container_id
            run_command = f"docker run -d --name {container_id} {container_id}"
            subprocess.run(run_command, shell=True, check=True)
            print(f"Container {container_id} started.")

        except subprocess.CalledProcessError as e:
            print(f"Failed to build or run container {container_id}: {str(e)}")
    else:
        print("No file paths found.")

############################################################

def copy_file_to_containers(username, process_selection):
    # Determine the source file based on the process
    if process_selection == 'read_image':
        source_file = '/home/reynel1995/Thesis/host_1/read_image.py'
    elif process_selection == 'clean_tiles':
        source_file = '/home/reynel1995/Thesis/host_1/clean_tiles.py'
    elif process_selection == 'apply_watershed':
        source_file = '/home/reynel1995/Thesis/host_1/apply_watershed.py'
    elif process_selection == 'classify_image':
        source_file = '/home/reynel1995/Thesis/host_1/classify_image.py'
    else:
        raise ValueError("Invalid process.")

    try:
        # Get the container ID (participant name)
        container_id = username

        # Copy the file to the container
        copy_command = f"docker cp {source_file} {container_id}:/app/"
        os.system(copy_command)

        print(f"File {source_file} copied to container {container_id} volume.")

        # If process_selection is 'clean_tiles', also copy the values.json file
        if process_selection == 'clean_tiles':
            values_file = f'/home/reynel1995/Thesis/host_1/{username}/values.
 ↪json'
            copy_values_command = f"docker cp {values_file} {container_id}:/app/
 ↪"
            os.system(copy_values_command)

        # If process_selection is 'classify_image', also copy the values.json␣
 ↪file
        if process_selection == 'classify_image':
            model_file = f'/home/reynel1995/Thesis/host_1/model.h5'
            copy_values_command = f"docker cp {model_file} {container_id}:/app/"
            os.system(copy_values_command)
            print(f"File {model_file} copied to container {container_id} volume.
 ↪")

    except docker.errors.NotFound as e:
        print(f"Container {container_id} not found: {str(e)}")
    except docker.errors.APIError as e:
        print(f"Failed to copy file {source_file} to container {container_id}␣
 ↪volume: {str(e)}")




###########################################################

def execute_command_in_containers(username, process_selection):
    # Determine the source file based on the process selection
    if process_selection == 'read_image':
        source_file = '/home/reynel1995/Thesis/host_1/read_image.py'
    elif process_selection == 'clean_tiles':
        source_file = '/home/reynel1995/Thesis/host_1/clean_tiles.py'
    elif process_selection == 'apply_watershed':
        source_file = '/home/reynel1995/Thesis/host_1/apply_watershed.py'
    elif process_selection == 'classify_image':
        source_file = '/home/reynel1995/Thesis/host_1/classify_image.py'
    else:
        raise ValueError("Invalid process selection.")

    # Command to be executed in the containers
    command = f"python /app/{os.path.basename(source_file)}"
    print(command)

    client = docker.from_env()

    output_directory = "/home/reynel1995/Thesis/host_1"  # Output directory

    try:
        # Get the container ID (participant name)
        container_id = username

        # Execute the command in the container
        container = client.containers.get(container_id)
        print(container)
        response = container.exec_run(command)

        output = response.output.decode()
        # If process_selection is 'clean_tiles', also copy the values.json file
        if process_selection == 'read_image':

            data = json.loads(output)

            factors = data["factors"]
            num_deepzoom_levels = data["num_deepzoom_levels"]
            Tiles_totales = data["Tiles_totales"]

            Tiles_totales_float = "{:,.0f}".format(Tiles_totales)
            Tiles_totales_float

            print(f"Command '{command}' executed in container {container_id}.")

            # Create the directory for the container within the output directory
            container_directory = os.path.join(output_directory, container_id)
            os.makedirs(container_directory, exist_ok=True)

            # Copy the images from the container to the output directory on␣
↪your localhost
            #os.system(f"docker cp {container_id}:/app/watershed_images␣
↪{container_directory}")


            # Save the response to a file within the container_directory
            response_file = os.path.join(container_directory,␣
↪"response_read_image.txt")
            with open(response_file, "w") as f:
                f.write(f"The resolution levels are {factors}, the lower the␣
↪better.\n")
                f.write(f"The zoom levels are from 1 to␣
↪{num_deepzoom_levels-1}, the higher the better.\n")
                f.write(f"There are {Tiles_totales_float} tiles in the best␣
↪resolution level {factors[0]} and the higher zoom {num_deepzoom_levels-1}.
↪\n")

            # Save factors and num_deepzoom_levels to a JSON file
            variables_file = os.path.join(container_directory, "variables.json")
            with open(variables_file, "w") as f:
                json.dump({"factors": factors, "num_deepzoom_levels":␣
↪num_deepzoom_levels-1}, f)

        if process_selection == 'clean_tiles':

            data = json.loads(output)
            Tiles = data["Tiles"]
            Total_tiles = data["Total_tiles"]
            resolution = data["resolution"]
            zoom = data["zoom"]

            print(f"Command '{command}' executed in container {container_id}.")

            # Create the directory for the container within the output directory
            container_directory = os.path.join(output_directory, container_id)
            os.makedirs(container_directory, exist_ok=True)

            # Save the response to a file within the container_directory
            response_file = os.path.join(container_directory,␣
↪"response_clean_tiles.txt")
            with open(response_file, "w") as f:
                f.write(f"The slide with the resolution {resolution} and zoom␣
↪{zoom} choosen is divided in {Total_tiles} tiles.\n")
                f.write(f"Only {Tiles} tiles were saved in User system.
↪{Total_tiles - Tiles} were cleaned!\n")


            # Save factors and num_deepzoom_levels to a JSON file
            variables_file = os.path.join(container_directory, "clean_tiles.
↪json")
            with open(variables_file, "w") as f:
                json.dump({"Tiles": Tiles, "Total_tiles": Total_tiles}, f)

        if process_selection == 'apply_watershed':

            data = json.loads(output)

            count_watershed_images = data["count_watershed_images"]

            print(f"Command '{command}' executed in container {container_id}.")

            # Create the directory for the container within the output directory
            container_directory = os.path.join(output_directory, container_id)
            os.makedirs(container_directory, exist_ok=True)

            # Save the response to a file within the container_directory
            response_file = os.path.join(container_directory,␣
↪"response_watershed_tiles.txt")
            with open(response_file, "w") as f:
                f.write(f"{count_watershed_images} slides have gotten the␣
↪watershed algorithm.\n")
```

```
            # Save factors and num_deepzoom_levels to a JSON file
            variables_file = os.path.join(container_directory, "watershed_tiles.
 ↪json")
            with open(variables_file, "w") as f:
                json.dump({"count_watershed_images": count_watershed_images}, f)

        if process_selection == 'classify_image':

            output = output.strip()
            output = output[output.find('{'):output.rfind('}') + 1]

            data = json.loads(output)

            classify_image = data["predicted_class_label"]

            print(f"Command '{command}' executed in container {container_id}.")

            # Create the directory for the container within the output directory
            container_directory = os.path.join(output_directory, container_id)
            os.makedirs(container_directory, exist_ok=True)

            # Save the response to a file within the container_directory
            response_file = os.path.join(container_directory,␣
 ↪"response_classify_image.txt")
            with open(response_file, "w") as f:
                f.write(f"The prediction of the model for the image given is␣
 ↪{classify_image}.\n")




    except docker.errors.NotFound:
        print(f"Container {container_id} not found.")

    print(f"Results saved to directory {output_directory}")

####################################################

def save_container_info(username, process_selection):
    output_directory = "/home/reynel1995/Thesis"
    processed_file = os.path.join(output_directory, "processed_container.txt")

    # Read the file path
    with open('/home/reynel1995/Thesis/file_paths.txt', 'r') as f:
        file_paths = f.read().splitlines()

    if file_paths:
        file_path = file_paths[0]  # Obtener el primer path

    if not os.path.isfile(processed_file):
        open(processed_file, 'w').close()

    with open(processed_file, "a") as f:
        f.write(f"{username} - {process_selection} - {file_path}\n")

    print(f"Container info saved: Username: {username}, Process Selection:␣
 ↪{process_selection}, File Path: {file_path}")
    print(f"Results saved to directory {output_directory}")

###################################################

def remove_first_line(file_path):
    try:
        with open(file_path, 'r+') as f:
            lines = f.readlines()
            if len(lines) > 1:
                f.seek(0)
                f.writelines(lines[1:])
                f.truncate()
            else:
                f.truncate(0)
    except FileNotFoundError:
        pass


####################################################

# Path of the participants file
participants_file = '/home/reynel1995/Thesis/participants.txt'

# Path of the processed containers tracking file
processed_containers_file = '/home/reynel1995/Thesis/processed_container.txt'

# Path of the file paths
file_paths_file = "/home/reynel1995/Thesis/file_paths.txt"

#######################################################



check_participants_task = PythonOperator(
    task_id='check_participants',
    python_callable=check_participants,
    op_args=[participants_file],
    provide_context=True,
    dag=dag
)

is_processed_task = PythonOperator(
    task_id='is_processed',
    python_callable=is_participant_processed,
    op_kwargs={'username': '{{ ti.
 ↪xcom_pull(task_ids="check_participants")[0][0] }}',
                'process_selection': '{{ ti.
 ↪xcom_pull(task_ids="check_participants")[0][1] }}'},
    provide_context=True,
    dag=dag
)

# Utilizar el BranchPythonOperator para decidir el siguiente paso
branch_task = BranchPythonOperator(
    task_id='branch_task',
    python_callable=decide_next_task,
    provide_context=True,
    dag=dag
)

create_folders_and_dockerfiles_task = PythonOperator(
    task_id='create_folders_and_dockerfiles',
    python_callable=create_folders_and_dockerfiles,
    provide_context=True,
    dag=dag
)

build_and_run_containers_task = PythonOperator(
    task_id='build_and_run_containers',
    python_callable=build_and_run_containers,
    op_kwargs={'username': '{{ ti.
 ↪xcom_pull(task_ids="check_participants")[0][0] }}'},
    provide_context=True,
    dag=dag
)

copy_file_to_containers_task = PythonOperator(
    task_id='copy_file_to_containers',
    python_callable=copy_file_to_containers,
    op_kwargs={'username': '{{ ti.
 ↪xcom_pull(task_ids="check_participants")[0][0] }}',
                'process_selection': '{{ ti.
 ↪xcom_pull(task_ids="check_participants")[0][1] }}'},
    provide_context=True,
    dag=dag
)

execute_command_in_containers_task = PythonOperator(
    task_id='execute_command_in_containers',
    python_callable=execute_command_in_containers,
    op_kwargs={'username': '{{ ti.
 ↪xcom_pull(task_ids="check_participants")[0][0] }}',
                'process_selection': '{{ ti.
 ↪xcom_pull(task_ids="check_participants")[0][1] }}'},
    provide_context=True,
    dag=dag
)

save_container_info_task = PythonOperator(
    task_id='save_container_info',
    python_callable=save_container_info,
    op_kwargs={'username': '{{ ti.
 ↪xcom_pull(task_ids="check_participants")[0][0] }}',
                'process_selection': '{{ ti.
 ↪xcom_pull(task_ids="check_participants")[0][1] }}'},
    provide_context=True,
    dag=dag
)

remove_first_line_participants_task = PythonOperator(
    task_id='remove_first_line_participants',
    python_callable=remove_first_line,
    op_args=[participants_file],
    dag=dag
)

remove_first_line_file_paths_task = PythonOperator(
    task_id='remove_first_line_file_paths',
    python_callable=remove_first_line,
    op_args=[file_paths_file],
    dag=dag
)


# Definir la relación entre las tareas
check_participants_task >> is_processed_task >> branch_task
```

```
branch_task >> create_folders_and_dockerfiles_task >> build_and_run_containers_task >> copy_file_to_containers_task >> execute_command_in_containers_task
branch_task >> copy_file_to_containers_task >> execute_command_in_containers_task

execute_command_in_containers_task >> save_container_info_task >> remove_first_line_participants_task
remove_first_line_participants_task >> remove_first_line_file_paths_task
```

App

July 10, 2023

[ ]:

```
import dash
from dash import dcc, html
import os
import json
import subprocess
import docker
from dash.dependencies import Input, Output, State
from dash.exceptions import PreventUpdate
import time


processed_containers_file_path = '/home/reynel1995/Thesis/processed_container.
 ↪txt'

app = dash.Dash(__name__)








def create_files_if_not_exist():
    participants_file = '/home/reynel1995/Thesis/participants.txt'
    file_paths_file = '/home/reynel1995/Thesis/file_paths.txt'

    if not os.path.isfile(participants_file):
        with open(participants_file, 'w') as f:
            pass

    if not os.path.isfile(file_paths_file):
        with open(file_paths_file, 'w') as f:
            pass


def show_results():
    output_directory = "/home/reynel1995/Thesis/host_1"

    if not os.path.isfile(processed_containers_file_path) or os.
↪stat(processed_containers_file_path).st_size == 0:
        return html.Div("No results available.", style={'margin-top': '10px'})
    else:
        with open(processed_containers_file_path, "r") as file:
            processed_containers_file = file.readlines()

        unique_combinations = set()  # Conjunto para almacenar combinaciones␣
↪únicas de contenedor y proceso
        results_by_user = {}  # Diccionario para almacenar los resultados␣
↪agrupados por usuario

        for container in processed_containers_file:
            container_id = container.split(" - ")[0].strip()
            process_selection = container.split(" - ")[1].strip()
            file_path = container.split(" - ")[2].strip()
            combination = (container_id, process_selection, file_path)

            # Verificar si la combinación ya existe en el conjunto
            if combination in unique_combinations:
                continue

            unique_combinations.add(combination)
            response_file = os.path.join(output_directory, container_id)

            if process_selection == 'read_image':
                response_file = os.path.join(response_file,␣
↪"response_read_image.txt")
            elif process_selection == 'clean_tiles':
                response_file = os.path.join(response_file,␣
↪"response_clean_tiles.txt")
            elif process_selection == 'apply_watershed':
                response_file = os.path.join(response_file,␣
↪"response_watershed_tiles.txt")
            elif process_selection == 'extract_annotations':
                response_file = os.path.join(response_file,␣
↪"response_extraction_annotation.txt")
            elif process_selection == 'classify_image':
                response_file = os.path.join(response_file,␣
↪"response_classify_image.txt")
            else:
                continue

            if not os.path.isfile(response_file):
                continue

            if container_id not in results_by_user:
                results_by_user[container_id] = {'processes': set(), 'files':␣
↪set(), 'outputs': []}

            # Agregar la información de la fila al usuario correspondiente
            results_by_user[container_id]['processes'].add(process_selection)
            results_by_user[container_id]['files'].add(file_path)

            with open(response_file, "r") as f:
                response_content = f.read()

            response_lines = response_content.split("\n")
            response_list = html.Ul([html.Li(line) for line in response_lines[:
↪-1]])

            results_by_user[container_id]['outputs'].append(
                html.Tr([
                    html.Td(container_id, style={'border': '1px solid black',␣
↪'padding': '5px', 'text-align': 'left'}),
                    html.Td(process_selection, style={'border': '1px solid␣
↪black', 'padding': '5px', 'text-align': 'left'}),
                    html.Td(file_path, style={'border': '1px solid black',␣
↪'padding': '5px', 'text-align': 'left', 'max-width': '200px', 'word-wrap':␣
↪'break-word'}),
                    html.Td(response_list, style={'border': '1px solid black',␣
↪'padding': '5px', 'text-align': 'left', 'max-width': '300px', 'word-wrap':␣
↪'break-word'})
                ])
            )

        # Construir la lista de bloques separados por usuario
        user_blocks = []
        for user, user_results in results_by_user.items():
            processes = ', '.join(user_results['processes'])
            files = ', '.join(user_results['files'])

            user_block = [
                html.H3(f"User: {user}"),
                html.Table([
                    html.Thead(html.Tr([
                        html.Th("Container", style={'border': '1px solid␣
↪black', 'padding': '5px', 'text-align': 'left'}),
                        html.Th("Process", style={'border': '1px solid black',␣
↪'padding': '5px', 'text-align': 'left'}),
                        html.Th("File Path", style={'border': '1px solid␣
 ↪black', 'padding': '5px', 'text-align': 'left'}),
                        html.Th("Output", style={'border': '1px solid black',␣
 ↪'padding': '5px', 'text-align': 'left'})
                    ])),
                    html.Tbody(user_results['outputs'])
                ],
                style={'border': '1px solid black', 'border-collapse':␣
 ↪'collapse', 'text-align': 'left', 'margin': '10px auto'})
            ]

            user_blocks.append(html.Div(user_block))

        return html.Div([
            html.H2("Results:"),
            *user_blocks
        ],
        style={'text-align': 'left'})








@app.callback(
    dash.dependencies.Output('output-container', 'children'),
    [dash.dependencies.Input('mostrar-resultados', 'n_clicks')]
)
def update_output(n_clicks):
    if n_clicks is not None and n_clicks > 0:
        if (n_clicks - 1) % 2 == 0:
            return show_results()
        else:
            return html.Div("No results available.", style={'margin-top':␣
 ↪'10px'})
    else:
        return ''
```

```
@app.callback(
    dash.dependencies.Output('create-container-output', 'children'),
    [dash.dependencies.Input('crear-container', 'n_clicks')],
    [dash.dependencies.State('file-path', 'value'),
     dash.dependencies.State('username', 'value'),
     dash.dependencies.State('process_selection', 'value')
     ]
)

def create_container(n_clicks, file_path, username, process_selection):


    if n_clicks is not None and n_clicks > 0:
        if not file_path or not username:
            return html.Div("One field is empty", style={'color': 'red'})

        if ' ' in file_path:
                return html.Div("File path contains blank spaces",␣
 ↪style={'color': 'red'})

        if not process_selection:
            return html.Div("Must select a process from the menu",␣
 ↪style={'color': 'red'})



        username = username.lower().replace(' ', '_')  # Replacing spaces with␣
 ↪underscores and converting to lowercase




        processed_containers_file = '/home/reynel1995/Thesis/
 ↪processed_container.txt'

        # Verificar si el usuario y el proceso ya existen en el archivo
        if os.path.isfile(processed_containers_file):
            with open(processed_containers_file, "r") as file:
                processed_containers_file = file.readlines()




            for line in processed_containers_file:
                existing_username, existing_process_selection,␣
 ↪existing_file_path = line.strip().split(" - ")

```

```
                if existing_username == username and existing_process_selection␣
↪== process_selection and existing_file_path == file_path:
                    return html.Div("This user and process combination already␣
↪exists", style={'color': 'red'})

                if existing_username == username and existing_process_selection␣
↪== process_selection and existing_file_path != file_path:
                    return html.Div("You must choose a different username",␣
↪style={'color': 'red'})

                if process_selection == "classify_image":
                    if not file_path.endswith('.jpg') and not file_path.
↪endswith('.png'):
                        return html.Div("For this process, a JPG or PNG file␣
↪path must be specified", style={'color': 'red'})

                    create_files_if_not_exist()


                    participants_file = '/home/reynel1995/Thesis/participants.
↪txt'
                    file_paths_file = '/home/reynel1995/Thesis/file_paths.txt'


                    with open(participants_file, 'a') as f:
                        f.write(f'{username} - {process_selection}\n')

                    with open(file_paths_file, 'a') as f:
                        f.write(f'{file_path}\n')



                    return html.Div("Container registered successfully",␣
↪style={'color': 'green'})


                if process_selection == "extract_annotations":
                    if not file_path.endswith('.mrxs'):
                        return html.Div("For this process, a .mrxs file path␣
↪must be specified", style={'color': 'red'})

                    xml_input = dcc.Input(id='xml-input', type='text',␣
↪placeholder='Path to XML file')

                    extract_annotation_button = html.Button(
                            'Extract Annotations',
                            id='extract-annotations-button',
                            n_clicks=0,
                            style={'display': 'inline-block'}
                        )

                    return_button_extraction = html.Button(
                            'Return',
                            id='return-button-extraction',
                            n_clicks=0,
                            style={'display': 'inline-block'}
                        )

                    return html.Div(
                            id='xml-file-id',
                            children=[
                                html.Div('You must define a xml file path',␣
↪style={'color': 'red', 'margin-bottom': '10px'}),
                                xml_input,
                                html.Div([
                                    extract_annotation_button,
                                    return_button_extraction
                                ], style={'margin-top': '10px'}),
                                html.Div(id='xml-file-output',␣
↪style={'margin-top': '10px'})
                            ]
                        )


                if existing_username == username and existing_file_path ==␣
↪file_path and process_selection == 'clean_tiles':
                        # Leer el archivo variables.json
                        variables_file = os.path.join('/home/reynel1995/Thesis/
↪host_1', existing_username, 'variables.json')
                        with open(variables_file, 'r') as f:
                            variables_data = json.load(f)

                        factors = variables_data['factors']
                        num_deepzoom_levels =␣
↪variables_data['num_deepzoom_levels']

                        resolution_dropdown = create_dropdown_with_title("Level␣
↪of Resolution", "resolution-dropdown", factors, factors[0])
                        zoom_dropdown = create_dropdown_with_title("Level of␣
↪Zoom", "zoom-dropdown", range(num_deepzoom_levels + 1), 0)
```

```
                    clean_slide_button = html.Button(
                        'Clean Slide',
                        id='clean-slide-button',
                        n_clicks=0,
                        style={'display': 'inline-block'}
                    )

                    return_button = html.Button(
                        'Return',
                        id='return-button',
                        n_clicks=0,
                        style={'display': 'inline-block'}
                    )

                    return html.Div(
                        id='my-clean-tiles-id',
                        children=[
                            html.Div('You must choose a resolution and zoom␣
↪level', style={'color': 'red', 'margin-bottom': '10px'}),
                            resolution_dropdown,
                            zoom_dropdown,
                            html.Div([
                                clean_slide_button,
                                return_button
                            ], style={'margin-top': '10px'}),
                            html.Div(id='clean-slide-output',␣
↪style={'margin-top': '10px'})
                        ]
                    )


    if file_path and file_path.endswith('.mrxs'):
        create_files_if_not_exist()


        participants_file = '/home/reynel1995/Thesis/participants.txt'
        file_paths_file = '/home/reynel1995/Thesis/file_paths.txt'


        with open(participants_file, 'a') as f:
            f.write(f'{username} - {process_selection}\n')

        with open(file_paths_file, 'a') as f:
            f.write(f'{file_path}\n')
            return html.Div("Container registered successfully", style={'color':
 ↪ 'green'})

        else:
            return html.Div("File path not valid", style={'color': 'red'})




@app.callback(
    Output('remove-process-output', 'children'),
    Input('remove-process', 'n_clicks'),
    State('username', 'value'),
    State('file-path', 'value'),
    State('process_selection', 'value')
)

def remove_process(n_clicks, username, file_path, process_selection):
    print("n_clicks:", n_clicks)
    print("username:", username)
    print("file_path:", file_path)
    print("process_selection:", process_selection)

    if n_clicks > 0:
        # Ruta del archivo processed_container.txt
        container_path = '/home/reynel1995/Thesis/processed_container.txt'
        print("container_path:", container_path)

        # Verificar si el archivo processed_container.txt existe
        if os.path.isfile(container_path):
            print("processed_container.txt exists")
            # Leer el contenido del archivo
            with open(container_path, 'r') as file:
                lines = file.readlines()

            # Verificar si se encontraron líneas para eliminar
            if any(line.startswith(username) and line.
 ↪endswith(f"{process_selection} - {file_path}\n") for line in lines):
                print("Lines to remove found")
                # Obtener el ID del contenedor
                client = docker.from_env()
                container_id = client.containers.get(username)
                print("container_id:", container_id)

                # Realizar acciones adicionales según la selección del proceso
                if process_selection == 'read_image':
                    print("Process selection: read_image")
                    command = f"rm /app/read_image.py"
                    container_id.exec_run(command)

                    # Remove variables.json and read_image_response.txt from␣
↪the host1 folder
                    host1_folder = f'/home/reynel1995/Thesis/host_1/{username}'
                    os.remove(os.path.join(host1_folder, 'variables.json'))
                    os.remove(os.path.join(host1_folder, 'response_read_image.
↪txt'))

                elif process_selection == 'clean_tiles':
                    print("Process selection: clean_tiles")
                    command = f"rm -r /app/clean_tiles.py /app/paleo_images"
                    container_id.exec_run(command)

                    # Remove clean_tiles.json, values.json, and␣
↪clean_tiles_response.txt from the host1 folder
                    host1_folder = f'/home/reynel1995/Thesis/host_1/{username}'
                    os.remove(os.path.join(host1_folder, 'clean_tiles.json'))
                    os.remove(os.path.join(host1_folder, 'values.json'))
                    os.remove(os.path.join(host1_folder, 'response_clean_tiles.
↪txt'))

                elif process_selection == 'apply_watershed':
                    print("Process selection: apply_watershed")
                    command = f"rm -r /app/apply_watershed.py /app/
↪watershed_images"
                    container_id.exec_run(command)

                    # Remove clean_tiles.json, values.json, and␣
↪clean_tiles_response.txt from the host1 folder
                    host1_folder = f'/home/reynel1995/Thesis/host_1/{username}'
                    os.remove(os.path.join(host1_folder, 'watershed_tiles.
↪json'))
                    os.remove(os.path.join(host1_folder,␣
↪'response_watershed_tiles.txt'))

                elif process_selection == 'classify_image':
                    print("Process selection: classify_image")
                    command = f"rm -r /app/classify_image.py"
                    container_id.exec_run(command)

                    # Remove clean_tiles.json, values.json, and␣
↪clean_tiles_response.txt from the host1 folder
```

```
                    host1_folder = f'/home/reynel1995/Thesis/host_1/{username}'
                    os.remove(os.path.join(host1_folder,␣
↪'response_classify_image.txt'))

                # Obtener la lista de nombres de usuario únicos en el archivo␣
↪procesado
                unique_usernames = [line.split()[0] for line in lines]
                print("unique_usernames:", unique_usernames)
                count = unique_usernames.count(username)

                # Verificar si solo queda el último nombre de usuario en el␣
↪archivo procesado
                if count == 1:
                    print("Last user in processed container")
                    # Eliminar el Dockerfile del directorio file_path
                    dockerfile_path = os.path.join(os.path.dirname(file_path),␣
↪'Dockerfile')
                    os.remove(dockerfile_path)

                    container_id.stop()
                    print("Container stopped")

                    # Eliminar el contenedor Docker
                    container_id.remove()
                    print("Container removed")

                    # Eliminar la imagen Docker con el nombre de usuario
                    client.images.remove(username)
                    print("Image removed")

                # Filtrar las líneas que coinciden con el username, process␣
↪selection y file path
                filtered_lines = [line for line in lines if not (
                    line.startswith(username) and line.
↪endswith(f"{process_selection} - {file_path}\n")
                )]

                # Sobreescribir el archivo con las líneas filtradas
                with open(container_path, 'w') as file:
                    file.writelines(filtered_lines)

                return html.Div('Process removed successfully.', style={'color':
↪ 'green', 'margin-bottom': '10px'})
            else:
                return html.Div('No process was found.', style={'color': 'red'})
        else:
            return html.Div('Processed container file not found.',␣
 ↪style={'color': 'red'})

    return html.Div()

@app.callback(
    Output('xml-file-output', 'children'),
    [Input('extract-annotations-button', 'n_clicks')],
    [State('xml-input', 'value'),
     State('file-path', 'value'),
     State('username', 'value'),
     State('process_selection', 'value')]
)
def handle_extraction_slide_button(n_clicks, xml, file_path, username,␣
 ↪process_selection):
    if n_clicks is not None and n_clicks > 0:
        if not xml or not file_path:
            return html.Div("Please define an xml and .mrxs file path.",␣
 ↪style={'color': 'red'})

        if not file_path.endswith('.mrxs') or not xml.endswith('.xml'):
            return html.Div("For this process, both a .mrxs and .xml file path␣
 ↪must be specified", style={'color': 'red'})

        # Crear el directorio del usuario si no existe
        user_directory = os.path.join('/home/reynel1995/Thesis/host_1',␣
 ↪username)
        if not os.path.exists(user_directory):
            os.makedirs(user_directory)

        # Guardar los valores de resolución y zoom en un archivo JSON dentro␣
 ↪del directorio del usuario
        data = {
            'xml': xml
        }
        json_file = os.path.join(user_directory, 'values.json')

        with open(json_file, 'w') as f:
            json.dump(data, f)

        # Actualizar el archivo "participants.txt" con el usuario y el proceso␣
 ↪correspondiente
        participants_file = '/home/reynel1995/Thesis/participants.txt'
        with open(participants_file, 'a') as f:
            f.write(f'{username} - {process_selection}\n')
        # Actualizar el archivo "file_paths.txt" con el file_path␣
 ↪correspondiente
        file_paths_file = '/home/reynel1995/Thesis/file_paths.txt'
        with open(file_paths_file, 'a') as f:
            f.write(f'{file_path}\n')

        return html.Div("Values saved successfully.", style={'color': 'green'})

    return None









@app.callback(
    Output('clean-slide-output', 'children'),
    [Input('clean-slide-button', 'n_clicks')],
    [State('resolution-dropdown', 'value'),
     State('zoom-dropdown', 'value'),
     State('file-path', 'value'),
     State('username', 'value'),
     State('process_selection', 'value')]
)
def handle_clean_slide_button(n_clicks, resolution, zoom, file_path, username,␣
 ↪process_selection):
    if n_clicks is not None and n_clicks > 0:
        if not resolution or not zoom or not file_path:
            return html.Div("Please select a resolution, zoom level, and file␣
 ↪path.", style={'color': 'red'})

        # Crear el directorio del usuario si no existe
        user_directory = os.path.join('/home/reynel1995/Thesis/host_1',␣
 ↪username)
        if not os.path.exists(user_directory):
            os.makedirs(user_directory)

        # Guardar los valores de resolución y zoom en un archivo JSON dentro␣
 ↪del directorio del usuario
        data = {
            'resolution': resolution,
            'zoom': zoom
        }
        json_file = os.path.join(user_directory, 'values.json')
        with open(json_file, 'w') as f:
            json.dump(data, f)

        # Actualizar el archivo "participants.txt" con el usuario y el proceso␣
 ↪correspondiente
        participants_file = '/home/reynel1995/Thesis/participants.txt'
        with open(participants_file, 'a') as f:
            f.write(f'{username} - {process_selection}\n')

        # Actualizar el archivo "file_paths.txt" con el file_path␣
 ↪correspondiente
        file_paths_file = '/home/reynel1995/Thesis/file_paths.txt'
        with open(file_paths_file, 'a') as f:
            f.write(f'{file_path}\n')

        return html.Div("Values saved successfully.", style={'color': 'green'})

    return None


def create_dropdown_with_title(title, dropdown_id, options, value):
    dropdown = dcc.Dropdown(
        id=dropdown_id,
        options=[{'label': str(option), 'value': option} for option in options],
        value=value,
        style={'width': '150px'}  # Ajusta el ancho del dropdown
    )

    return html.Div(
        children=[
            html.Label(title),  # Título del dropdown
            dropdown
        ],
        style={'display': 'inline-block', 'margin-right': '20px', 'text-align':␣
 ↪'center'}
    )




@app.callback(
    Output('my-clean-tiles-id', 'style'),
    Output('create-container-div', 'style'),
    Input('return-button', 'n_clicks'),

)
```

```
def toggle_containers(n_clicks):
    if n_clicks and n_clicks > 0:
        return {'display': 'none'}, {'display': 'block'}
    else:
        return {'display': 'block'}, {'display': 'none'}







def generate_button():
    if not os.path.isfile(processed_containers_file_path) or os.
 ↪stat(processed_containers_file_path).st_size == 0:
        return None
    else:
        return html.Div([
            html.Button('Remove Process', id='remove-process', n_clicks=0,␣
 ↪style={'margin-bottom': '20px'}),
            html.Div(
                id='remove-process-output',
                style={'width': '100%', 'margin-left': '10px'}
            )
        ])

@app.callback(
    Output('remove-process-container', 'children'),
    [Input('remove-process', 'n_clicks')]
)
def hide_button(n_clicks):
    if n_clicks > 0:
        return ''
    else:
        return generate_button()






app.layout = html.Div(
    style={
        'position': 'relative',
        'height': '100vh',
        'overflow': 'auto',
    },
    children=[
        html.Div(
            style={
                'position': 'fixed',
                'top': 0,
                'left': 0,
                'right': 0,
                'bottom': 0,
                'background-image': 'url("/assets/kap5-mikrofossiler-2.png")',
                'background-size': 'cover',
                'background-repeat': 'repeat',
                'background-position': 'center',
                'opacity': '0.2',
                'z-index': -1,
            }
        ),
        html.Div(
            style={
                'position': 'relative',
                'margin': '20px auto',
                'width': '70%',
                'text-align': 'center',
                'background-color': 'rgba(255, 255, 255, 0.2)',
                'padding': '20px',
            },
            children=[
                html.H1("Digital palynological slides Analyzer",␣
↪style={'text-align': 'center', 'margin-top': '50px'}),
                html.Div(
                    style={'margin-bottom': '5px'},
                    children=[
                        dcc.Input(id='username', type='text',␣
↪placeholder='Username', style={'margin-right': '10px','margin-bottom':␣
↪'10px'}),
                        dcc.Input(id='file-path', type='text',␣
↪placeholder='File path', style={'margin-right': '10px'})

                    ]
                ),
                html.Div(
                    style={'margin-bottom': '15px'},
                    children=[

                        dcc.Dropdown(
                            id='process_selection',
                            options=[
                                {'label': 'Read Image', 'value': 'read_image'},
                                {'label': 'Clean Tiles', 'value':␣
 ↪'clean_tiles'},
                                {'label': 'Apply Watershed', 'value':␣
 ↪'apply_watershed'},
                                {'label': 'Extract Annotations', 'value':␣
 ↪'extract_annotations'},
                                {'label': 'Classify Image', 'value':␣
 ↪'classify_image'}
                            ],
                            placeholder='Select a process',
                            style={'width': '200px', 'margin': 'auto'}
                        )
                    ]
                ),
                html.Div(
                    id='create-container-div',
                    style={'margin-bottom': '10px'},
                    children=[
                        html.Button('Initiate process', id='crear-container',␣
 ↪n_clicks=0, style={'margin-left': '10px'}),

                    ],
                ),
                html.Div(style={'margin-bottom': '40px'},␣
 ↪id='create-container-output'),
                generate_button(),
                html.Button('Show Results', id='mostrar-resultados',␣
 ↪n_clicks=0),
                html.Div(
                    id='output-container',
                    style={'width': '100%'}
                )
            ]
        )
    ]
)



if __name__ == '__main__':
    #app.run_server(debug=True)
    app.run_server()
```

read_image

July 10, 2023

[ ]:

```python
import numpy as np
import os
import json

OPENSLIDE_PATH = "/usr/local/lib/python3.8/site-packages/openslide"

if hasattr(os, 'add_dll_directory'):
    # Python >= 3.8 on Windows
    with os.add_dll_directory(OPENSLIDE_PATH):
        import openslide
else:
    import openslide

import openslide
import glob
from openslide.deepzoom import DeepZoomGenerator

mrxs_file_path = glob.glob("/app/*.mrxs")[0]
slide = openslide.OpenSlide(mrxs_file_path)

def read_image(slide):


    factors = slide.level_downsamples



    tiles = DeepZoomGenerator(slide, tile_size=factors[0], overlap=0,␣
 ↪limit_bounds=False)
    num_deepzoom_levels = tiles.level_count

    level_num= num_deepzoom_levels-1
    level_tiles_tot = tiles.level_tiles[level_num][0]*tiles.
 ↪level_tiles[level_num][1]

    slide.close()
    return json.dumps({"factors": factors, "num_deepzoom_levels":␣
  ↪num_deepzoom_levels, "Tiles_totales": level_tiles_tot})

output = read_image(slide)
print(output)
```

apply_watershed

July 10, 2023

```
[ ]: import numpy as np
     import os
     import json

     OPENSLIDE_PATH = "/usr/local/lib/python3.8/site-packages/openslide"

     if hasattr(os, 'add_dll_directory'):
         # Python >= 3.8 on Windows
         with os.add_dll_directory(OPENSLIDE_PATH):
             import openslide
     else:
         import openslide

     import openslide
     import cv2

     input_folder_path = "/app/paleo_images"

     def apply_watershed(input_folder_path):


         output_folder_path = "/app/watershed_images"

         # Create the output folder if it does not exist
         if not os.path.exists(output_folder_path):
             os.makedirs(output_folder_path)

         count_watershed_images = 0



         # Loop through all the images in the input folder
         for filename in os.listdir(input_folder_path):
             if filename.endswith('.jpg') and not filename.endswith('_seg.jpg'):  #␣
       ↪assuming all images are JPEG files
                 # Load the image and convert it to grayscale
                 input_img_path = os.path.join(input_folder_path, filename)
```

```
            img = cv2.imread(input_img_path)
            gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

            # Apply thresholding to separate the foreground (microfossils) from␣
↪the background
            ret, thresh = cv2.threshold(gray, 0, 255, cv2.THRESH_BINARY_INV+cv2.
↪THRESH_OTSU)
                # Apply morphological opening to remove small objects and noise
            kernel = np.ones((3,3), np.uint8)
            opening = cv2.morphologyEx(thresh, cv2.MORPH_OPEN, kernel,␣
↪iterations=2)

            # Apply distance transform to obtain the distance map
            dist_transform = cv2.distanceTransform(opening, cv2.DIST_L2, 5)

                # Apply thresholding to the distance map to obtain the␣
↪foreground markers
            ret, fg_markers = cv2.threshold(dist_transform, 0.5*dist_transform.
↪max(), 255, 0)

                # Apply watershed algorithm to segment the foreground markers␣
↪and obtain the labels
            fg_markers = np.uint8(fg_markers)
            unknown = cv2.subtract(opening, fg_markers)
            ret, bg_markers = cv2.threshold(unknown, 0, 255, cv2.
↪THRESH_BINARY_INV)
            bg_markers = cv2.dilate(bg_markers, kernel, iterations=3)
            markers = cv2.add(fg_markers, bg_markers)

                # Convert the grayscale image to a 3-channel format
            img_color = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

                # Apply the watershed segmentation algorithm
            markers = markers.astype(np.int32)
            markers_copy = markers.copy()
            cv2.watershed(img_color, markers_copy)

                # Visualize the segmented image and the markers
            img_color[markers_copy == -1] = [255,0,0]

                # Save the segmented image to the output folder
            output_img_path = os.path.join(output_folder_path,␣
↪'img_'+str(filename)+'_seg.jpg')
            cv2.imwrite(output_img_path, img_color)
            count_watershed_images += 1
```

```
    return json.dumps({"count_watershed_images": count_watershed_images })

output = apply_watershed(input_folder_path)
print(output)
```

clean_tiles

July 10, 2023

[ ]:

```
import numpy as np
import os
import json

OPENSLIDE_PATH = "/usr/local/lib/python3.8/site-packages/openslide"

if hasattr(os, 'add_dll_directory'):
    # Python >= 3.8 on Windows
    with os.add_dll_directory(OPENSLIDE_PATH):
        import openslide
else:
    import openslide

import openslide
import glob
from openslide.deepzoom import DeepZoomGenerator
import cv2


mrxs_file_path = glob.glob("/app/*.mrxs")[0]
slide = openslide.OpenSlide(mrxs_file_path)




def clean_tiles(slide):


    json_file_path = '/app/values.json'
    with open(json_file_path, 'r') as f:
        data = json.load(f)
        resolution = data['resolution']
        zoom = data['zoom']

    tiles = DeepZoomGenerator(slide, tile_size=resolution, overlap=0,␣
 ↪limit_bounds=False)
```

```
    level_num= zoom
    level_tiles_tot = tiles.level_tiles[level_num][0]*tiles.
 ↪level_tiles[level_num][1]

    folder_path = "/app/paleo_images"

    if not os.path.exists(folder_path):
        os.makedirs(folder_path)

    Idx_max = zoom

    count_paleo_images = 0



    for col in range(0, tiles.level_tiles[Idx_max][0]):
        for row in range(0, tiles.level_tiles[Idx_max][1]):
            tile = tiles.get_tile(Idx_max, (col, row))
            tile_array = np.array(tile)
            gray = cv2.cvtColor(tile_array, cv2.COLOR_RGB2GRAY)
            unique_values = np.unique(gray)
            if len(unique_values) >= 2 and unique_values[0] != 255 and␣
 ↪unique_values[1] != 255:
                # Check if the mean pixel intensity is above a certain threshold
                if gray.mean() > 100:
                    # Check if the percentage of white pixels is below a␣
 ↪certain threshold
                    white_percentage = np.count_nonzero(gray == 255) / gray.size
                    if white_percentage < 0.9:
                        tile_path = os.path.join(folder_path,␣
 ↪f"tile_{col}_{row}.jpg")
                        tile.save(tile_path)
                        count_paleo_images += 1
            else:
                # print(f"Not saving blank tile at column {col} and row {row}")
                pass




    return json.dumps({"Tiles": count_paleo_images, "Total_tiles":␣
 ↪level_tiles_tot, "resolution": resolution, "zoom": zoom, })

output = clean_tiles(slide)
print(output)
```

extract_annotation

July 10, 2023

```
[ ]: # Extract labeled images from annotated palyslides using MRXS and XML files
     # Group the labeled data into 21 subfolders according to their class labels

     # MRXS and XML file paths:

     mrxs_paths = [
         'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/PalySlideImages/2_4-C-11␣
      ↪10052.7 ftC = 3064 mC.mrxs',
         'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/PalySlideImages/2_4-C-11␣
      ↪10070 ftC = 3069.3 mC.mrxs',
         'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/PalySlideImages/2_7-14␣
      ↪10658ft 8in C.mrxs',
         'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/PalySlideImages/16_3-2␣
      ↪1998.80 mC.mrxs',
         'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/PalySlideImages/
      ↪DigitalSlide_C1M_4S_1.mrxs',
         'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/PalySlideImages/
      ↪DigitalSlide_C1M_5S_1.mrxs'

     ]

     xml_paths = [
         'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/PalySlideImages/2_4-C-11␣
      ↪10052.7 ftC = 3064 mC_Annotations.xml',
         'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/PalySlideImages/2_4-C-11␣
      ↪10070 ftC = 3069.3 mC_Annotations.xml',
         'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/PalySlideImages/2_7-14␣
      ↪10658ft 8in C_Annotations.xml',
         'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/PalySlideImages/16_3-2␣
      ↪1998.80 mC_Annotations.xml',
         'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/PalySlideImages/
      ↪DigitalSlide_C1M_4S_1_Annotations.xml',
         'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/PalySlideImages/
      ↪DigitalSlide_C1M_5S_1_Annotations.xml'
     ]

     # Define a function to extract labeled images from MRXS and XML files
```

```python
def extract_labeled_images(mrxs_path, xml_path):

    slide = openslide.OpenSlide(mrxs_path)

    tree = ET.parse(xml_path)
    root = tree.getroot()

    annotations = root.find('destination/annotations')

    labeled_data = []
    for annotation in annotations:
        name = annotation.get('name')
        type = annotation.get('type')

        polygon_points = annotation.findall('p')
        points = [(int(point.get('x')), int(point.get('y'))) for point in polygon_points]

        # Extract region of interest from MRXS image based on polygon points given in the XML file.
        x_min = min([point[0] for point in points])
        x_max = max([point[0] for point in points])
        y_min = min([point[1] for point in points])
        y_max = max([point[1] for point in points])

        region = slide.read_region((x_min, y_min), 0, (x_max - x_min, y_max - y_min))

        labeled_data.append({'image': region, 'label': name})

    slide.close()

    return labeled_data

# Iterate over the MRXS and XML file paths, extract labeled images,
# and store the data in a Pandas dataframe
labeled_data_list = []
for mrxs_path, xml_path in zip(mrxs_paths, xml_paths):
    labeled_data = extract_labeled_images(mrxs_path, xml_path)
    labeled_data_list.extend(labeled_data)

data = pd.DataFrame(labeled_data_list)

data['image'] = data['image'].apply(lambda img: img.convert('RGB'))  # Convert to RGBA images to RGB
```

training

July 10, 2023

```
[ ]: def build_and_train_cnn(train_dir, val_dir, image_size, batch_size,␣
      ↪num_classes, num_epochs):

         """

         Build and train a CNN model using the provided training and validation␣
      ↪directories.



         Args:

             train_dir (str): Directory path for the training dataset.

             val_dir (str): Directory path for the validation dataset.

             image_size (tuple): Tuple specifying the target image size, e.g.,␣
      ↪(width, height).

             batch_size (int): Batch size for training.

             num_classes (int): Number of classes in the classification problem.

             num_epochs (int): Number of training epochs.



         Raises:

             ValueError: If the number of classes doesn't match the number of unique␣
      ↪labels in the datasets.

         """
```

```
# Data augmentation and normalization for training set
train_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True
)

# Only rescaling for validation set
val_datagen = ImageDataGenerator(rescale=1./255)


# Load and augment the training dataset
train_generator = train_datagen.flow_from_directory(
    train_dir,
    target_size=image_size,
    batch_size=batch_size,
    class_mode='categorical'
)
```

```python
# Load and augment the validation dataset

val_generator = val_datagen.flow_from_directory(

    val_dir,

    target_size=image_size,

    batch_size=batch_size,

    class_mode='categorical'

)



# Check if the number of classes matches the number of unique labels in the datasets

if num_classes != len(train_generator.class_indices):

    raise ValueError("Number of classes doesn't match the number of unique labels in the datasets.")




# Build the CNN model

model = tf.keras.models.Sequential([

    tf.keras.layers.Conv2D(32, (3, 3), activation='relu', input_shape=(image_size[0], image_size[1], 3)),

    tf.keras.layers.MaxPooling2D((2, 2)),

    tf.keras.layers.Conv2D(64, (3, 3), activation='relu'),

    tf.keras.layers.MaxPooling2D((2, 2)),

    tf.keras.layers.Conv2D(128, (3, 3), activation='relu'),

    tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(num_classes, activation='softmax')
])



# Compile the model
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])


# # Train the model
# model.fit(train_generator,
#           steps_per_epoch=train_generator.samples // batch_size,
#           validation_data=val_generator,
#           validation_steps=val_generator.samples // batch_size,
#           epochs=num_epochs)

# Train the model and obtain the history object
history = model.fit(train_generator,
                    steps_per_epoch=train_generator.samples // batch_size,
                    epochs=num_epochs,
                        validation_data=val_generator,
                        validation_steps=val_generator.samples // batch_size)


    # # Convert the function model to a tf.keras model object
    # model = tf.keras.Model(inputs=model_input, outputs=model_output)


    # Save the training history
    history_model_path  = 'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/
↪ImageClassification/training_history.pkl'


    with open(history_model_path, 'wb') as file:
        pickle.dump(history.history, file)


    # Save the model
    model_path  = 'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/
↪ImageClassification/model.h5'
    #save_model(model, model_path)
    model.save(model_path)


    # Return the history object
    return history
```

split_classes

July 10, 2023

[ ]:

```
# Count the number of records per class
class_counts = data['label'].value_counts()

# Filter out classes with only one record
#filtered_data = data[data['label'].map(class_counts) > 1]
filtered_data = data[data['label'].isin(class_counts[class_counts > 1].index)]

# Get the unique classes
classes = sorted(filtered_data['label'].unique().tolist())

# Split the filtered data into training and validation sets with the same␣
 ↪classes
train_data = pd.DataFrame()
val_data = pd.DataFrame()
for label in classes:

    class_data = filtered_data[filtered_data['label'] == label]
    class_train_data, class_val_data = train_test_split(class_data, test_size=0.
 ↪2, random_state=42)
    train_data = pd.concat([train_data, class_train_data])
    val_data = pd.concat([val_data, class_val_data])

# Get records using using len() function
print("Number of training image data:", len(train_data))

# Get the number of records using the .shape attribute
print("Number of validation image data:", val_data.shape[0])

# Get number of unique class labels
train_num_classes = len(train_data['label'].unique().tolist())
val_num_classes  = len(val_data['label'].unique().tolist())
print(f'Number of training class labels: {train_num_classes}')
print(f'Number of validation class labels: {val_num_classes}')

classes =  train_data['label'].unique().tolist()
classes
```

classify_image

July 10, 2023

[ ]:

```
import os
import cv2
import numpy as np
import matplotlib.pyplot as plt
from keras.models import load_model
import json
import glob
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'

# Specify the path to the image file and the desired size
image_files = glob.glob("/app/*.png") + glob.glob("/app/*.jpg")
if image_files:
    image_path = image_files[0]


h5_file_path = glob.glob("/app/*.h5")[0]
model = load_model(h5_file_path)

desired_size = (224, 224)

def resize_image(image_path, desired_size):
  # Check if the image file exists
  if not os.path.exists(image_path):
    return None

  # Load the image using OpenCV
  image = cv2.imread(image_path)

   # Check if the image is loaded successfully
  if image is None:
    return None
  # Resize the image
  resized_image = cv2.resize(image, desired_size)
  # Convert the resized image to RGB format
  resized_image = cv2.cvtColor(resized_image, cv2.COLOR_BGR2RGB)
  # Get the model's class indices
  model_class_indices = model.predict(np.zeros((1, 224, 224, 3))).argmax(axis=1)
  # Define the class labels
  class_labels = [
            'Acanthaulax venusta',
            'Cribroperidinium "prominoseptatum"',
            'Dingodinium tuberculosum',
            'Dingodinium tuberosum',
            'Fibrocysta axialis',
            'Gonyaulacysta jurassica',
            'Palaeoperidinium pyrophorum',
            'Senoniasphaera inornata',
            'Sentusidinium pilosum',
            'Spongodinium delitiense',
            'Spongodinium delitiense (operculum)',
            'Systematophora areolata',
            'Tubotuberella apatela'
        ]
  # Create a dictionary mapping the model's class indices to the class labels
  class_mapping = {index: label for index, label in enumerate(class_labels)}


  # Preprocess the image
  preprocessed_image = np.expand_dims(resized_image, axis=0)
  preprocessed_image = preprocessed_image / 255.0  # Normalize pixel values to the range [0, 1]

  # Perform the prediction
  predictions = model.predict(preprocessed_image)

  # Get the predicted class label
  predicted_class_index = np.argmax(predictions[0])

  # Print the predicted class index


  predicted_class_label = class_labels[predicted_class_index]

  return json.dumps({"predicted_class_label": predicted_class_label })



output = resize_image(image_path, desired_size)
print(output)
```

model_training

July 11, 2023

[ ]:

```python
def plot_training_history(history):
    """
    Plot the training and validation accuracy versus loss from the training history.

    Args:
        history (History): Training history object returned by model.fit().
    """

    # Extract the epochs and metrics from the history object
    epochs = range(1, len(history['loss']) + 1)

    # Access the training metrics and loss
    training_loss = history['loss']
    training_accuracy = history['accuracy']

    # Access the validation metrics and loss
    val_loss = history['val_loss']
    val_accuracy = history['val_accuracy']

    # Plot the accuracy
    plt.figure(figsize=(12, 6))
    plt.subplot(1, 2, 1)
    plt.plot(training_accuracy, label='Training Accuracy')
    plt.plot(val_accuracy, label='Validation Accuracy')
    plt.title('Training and Validation Accuracy')
    plt.ylim(0.3, 0.9)
    plt.xticks(np.arange(0, len(epochs) + 1, 2.0))
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend(loc='upper left')

    # Plot the loss
    plt.subplot(1, 2, 2)
    plt.plot(training_loss, label='Training Loss')
    plt.plot(val_loss, label='Validation Loss')
    plt.title('Training and Validation Loss')
    plt.ylim(0.4, 2)
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.xticks(np.arange(0, len(epochs)+1, 2.0))
    plt.legend(loc='upper right')

    # Show the plot
    plt.tight_layout()
    plt.show()
```

model_history

July 11, 2023

[ ]:

```
# Load the saved model
model_path = 'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/
 ↪ImageClassification/model.h5'
model = load_model(model_path)

# Load the training history
history_model_path  = 'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/
 ↪ImageClassification/training_history.pkl'
with open(history_model_path, 'rb') as file:
    history = pickle.load(file)

# Access the training metrics and loss
training_loss = history['loss']
training_accuracy = history['accuracy']

# Access the validation metrics and loss
validation_loss = history['val_loss']
validation_accuracy = history['val_accuracy']

# # Print the training and validation metrics
# print("Training Loss:", training_loss)
# print("Training Accuracy:", training_accuracy)
# print("Validation Loss:", val_loss)
# print("Validation Accuracy:", val_accuracy)

# From  the trained model and history object
plot_training_history(history)
```

Model_evaluation

July 11, 2023

[ ]:

```
def resize_image(image_path, desired_size):
    try:
        # Check if the image file exists
        if not os.path.exists(image_path):
            print(f"Image file '{image_path}' not found.")
            return None

        # Load the image using OpenCV
        image = cv2.imread(image_path)

        # Check if the image is loaded successfully
        if image is None:
            print(f"Failed to load image '{image_path}'.")
            return None

        # Resize the image
        resized_image = cv2.resize(image, desired_size)

        # Convert the resized image to RGB format
        resized_image = cv2.cvtColor(resized_image, cv2.COLOR_BGR2RGB)

        return resized_image

    except Exception as e:
        print(f"Error occurred while resizing image '{image_path}': {e}")
        return None


# Define the directory of test images and the desired size
image_dir = 'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/
 ↪ImageClassification/test_images'
desired_size = (224, 224)

# Load trained model
model_path = 'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/
 ↪ImageClassification/model.h5'
model = load_model(model_path)
```

```
# Define class labels
class_labels = [
    #'Acanthaulax venusta',
    'Cribroperidinium "prominoseptatum"',
    # 'Dingodinium tuberculosum',
    # 'Dingodinium tuberosum',
    'Fibrocysta axialis',
    # 'Gonyaulacysta jurassica',
    'Palaeoperidinium pyrophorum',
    'Senoniasphaera inornata',
    # 'Sentusidinium pilosum',
    'Spongodinium delitiense',
    'Spongodinium delitiense (operculum)'
#     'Systematophora areolata',
#     'Tubotuberella apatela'
]

# Create a dictionary to map image filenames to true labels
image_label_map = {


    '3.jpg': 'Cribroperidinium "prominoseptatum"',
    '4.jpg': 'Cribroperidinium "prominoseptatum"',
    '5.jpg': 'Cribroperidinium "prominoseptatum"',
    '6.jpg': 'Cribroperidinium "prominoseptatum"',
    '7.jpg': 'Cribroperidinium "prominoseptatum"',
    '12.jpg': 'Fibrocysta axialis',
    '13.jpg': 'Fibrocysta axialis',
    '14.jpg': 'Fibrocysta axialis',
    '15.jpg': 'Fibrocysta axialis',
    '16.jpg': 'Fibrocysta axialis',
    '17.jpg': 'Fibrocysta axialis',
    '18.jpg': 'Fibrocysta axialis',
    '19.jpg': 'Fibrocysta axialis',
    '20.jpg': 'Fibrocysta axialis',
    '23.jpg': 'Palaeoperidinium pyrophorum',
    '24.jpg': 'Palaeoperidinium pyrophorum',
    '25.jpg': 'Palaeoperidinium pyrophorum',
    '26.jpg': 'Palaeoperidinium pyrophorum',
    '27.jpg': 'Palaeoperidinium pyrophorum',
    '28.jpg': 'Palaeoperidinium pyrophorum',
    '29.jpg': 'Palaeoperidinium pyrophorum',
    '30.jpg': 'Palaeoperidinium pyrophorum',
    '31.jpg': 'Palaeoperidinium pyrophorum',
    '32.jpg': 'Senoniasphaera inornata',
    '33.jpg': 'Senoniasphaera inornata',
    '34.jpg': 'Senoniasphaera inornata',
    '35.jpg': 'Senoniasphaera inornata',
    '36.jpg': 'Senoniasphaera inornata',
    '37.jpg': 'Senoniasphaera inornata',
    '38.jpg': 'Senoniasphaera inornata',
    '39.jpg': 'Senoniasphaera inornata',
    '40.jpg': 'Senoniasphaera inornata',
    '43.jpg': 'Spongodinium delitiense',
    '44.jpg': 'Spongodinium delitiense',
    '45.jpg': 'Spongodinium delitiense',
    '46.jpg': 'Spongodinium delitiense',
    '47.jpg': 'Spongodinium delitiense',
    '48.jpg': 'Spongodinium delitiense',
    '49.jpg': 'Spongodinium delitiense',
    '50.jpg': 'Spongodinium delitiense',
    '51.jpg': 'Spongodinium delitiense (operculum)',
    '52.jpg': 'Spongodinium delitiense (operculum)',
    '53.jpg': 'Spongodinium delitiense (operculum)',
    '54.jpg': 'Spongodinium delitiense (operculum)',
    '55.jpg': 'Spongodinium delitiense (operculum)',
    '56.jpg': 'Spongodinium delitiense (operculum)',

    # Add more filename-label pairs as needed
}

# Initialize empty lists for true and predicted labels
true_labels_list = []
predicted_labels = []
predicted_images = []

# Get the file names in the directory and sort them naturally
image_filenames = natsorted(os.listdir(image_dir))

for filename in image_filenames:
    if filename.endswith('.jpg') or filename.endswith('.png'):

        # Construct the full image path
        image_path = os.path.join(image_dir, filename)
        image_path = image_path.replace('\\','/')  # Replace backslashes with␣
 ↪forward slash

        # Resize the image
        resized_image = resize_image(image_path, desired_size)

        # Check if the image was resized successfully
        if resized_image is not None:
            # Perform further processing with the resized image
            # print("Image resized successfully.")

            # Preprocess the image
            preprocessed_image = np.expand_dims(resized_image, axis=0)
            preprocessed_image = preprocessed_image / 255.0  # Normalize pixel values to the range [0, 1]

            # Perform the prediction
            predictions = model.predict(preprocessed_image, verbose=0)

            # Get the predicted class label
            predicted_class_index = np.argmax(predictions[0])
            predicted_class_label = class_labels[predicted_class_index]

            # Append the true and predicted labels
            true_label = image_label_map.get(filename, 'Unknown')  # Get the true label from the mapping, or use 'Unknown' if not found
            true_labels_list.append(true_label)
            predicted_labels.append(predicted_class_label)
            predicted_images.append(image_path)

# Convert the lists of true labels and predicted labels to numpy arrays
true_labels = np.array(true_labels_list)
predicted_labels = np.array(predicted_labels)

print("True Lables:", true_labels)
print("Predicted Labels:", predicted_labels)

# Calculate the confusion matrix
cm = confusion_matrix(true_labels, predicted_labels)

# Calculate the accuracy
accuracy = accuracy_score(true_labels, predicted_labels)

# Calculate the precision
precision = np.nan_to_num(cm.diagonal() / cm.sum(axis=0), nan=0.0)

# Calculate the recall
recall = np.nan_to_num(cm.diagonal() / cm.sum(axis=1), nan=0.0)

# Calculate the F1-score
f1 = np.nan_to_num(2 * (precision * recall) / (precision + recall), nan=0.0)

print("Confusion Matrix:")
print(cm)

# Print the accuracy, precision, recall, and F1-Score
```

```
print("Accuracy: {:.2f}".format(accuracy))
print("Precision:", end=" ")
for p in precision:
    print("{:.2f}".format(p), end=" ")
print()
print("Recall:", end=" ")
for r in recall:
    print("{:.2f}".format(r), end=" ")
print()
print("F1-Score:", end=" ")
for f in f1:
    print("{:.2f}".format(f), end=" ")
print()

# Calculate the weighted F1 score
f1 = f1_score(true_labels, predicted_labels, average='weighted')
# Print the F1 score
print("Weighted F1 Score:", round(f1, 2))


# Calculate performance metrics for each class with sklearn method
report = classification_report(true_labels, predicted_labels, zero_division=0)
# Print the classification report
print("Classification report:",report)

## Create heatmap with the seaborn method
# Create a list of class labels
labels = np.unique(true_labels)

# Plot the heatmap
plt.figure(figsize=(8,6))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=labels,␣
 ↪yticklabels=labels)
plt.title("Confusion Matrix")
plt.xlabel("Predicted Labels")
plt.ylabel("True Labels")
plt.show()

# Visualize predicted images
for i in range(len(predicted_images)):
    image_path = predicted_images[i]
    image_label = predicted_labels[i]
    image = plt.imread(image_path)

    plt.imshow(image)
    plt.title(f"Predicted Label: {image_label}")
    plt.axis("off")
    plt.show()
```

mrcnn_detect_microfossils

July 12, 2023

[ ]:

```
# Import necessary libraries

import os
import cv2
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import skimage
import glob
import pandas as pd
import hashlib
from mrcnn import utils
from mrcnn import visualize
from mrcnn.visualize import display_images
from mrcnn.model import log
from mrcnn import model as modellib
import tensorflow as tf
from tensorflow import keras
import xml.etree.ElementTree as ET
from mrcnn.utils import Dataset
from mrcnn.model import MaskRCNN
from mrcnn.config import Config
import skimage.draw
import mrcnn
from mrcnn import model as modellib

# Set the "path_to_dll" variable to a specific directory path containing DLL␣
 ↪files.
path_to_dll = "C:/Users/emmie/anacond3/Lib/site-packages/openslide/
 ↪openslide-win64-20230414/bin"
os.environ["PATH"] = path_to_dll + ";" + os.environ["PATH"]
import openslide
```

0.0.1 Create a Pandas dataframe of labeled images with their annotation details.

- The 6 pairs of slide images and corresponding XML annotation files are used for this purpose.

- The labeled images and their annotations files extracted will be used to train Mask RCNN network for object identification task.

```
[ ]: mrxs_paths = [
         'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/PalySlideImages/2_4-C-11␣
      ↪10052.7 ftC = 3064 mC.mrxs',
         'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/PalySlideImages/2_4-C-11␣
      ↪10070 ftC = 3069.3 mC.mrxs',
         'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/PalySlideImages/2_7-14␣
      ↪10658ft 8in C.mrxs',
         'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/PalySlideImages/16_3-2␣
      ↪1998.80 mC.mrxs',
         'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/PalySlideImages/
      ↪DigitalSlide_C1M_4S_1.mrxs',
         'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/PalySlideImages/
      ↪DigitalSlide_C1M_5S_1.mrxs'

     ]

     xml_paths = [
         'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/PalySlideImages/2_4-C-11␣
      ↪10052.7 ftC = 3064 mC_Annotations.xml',
         'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/PalySlideImages/2_4-C-11␣
      ↪10070 ftC = 3069.3 mC_Annotations.xml',
         'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/PalySlideImages/2_7-14␣
      ↪10658ft 8in C_Annotations.xml',
         'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/PalySlideImages/16_3-2␣
      ↪1998.80 mC_Annotations.xml',
         'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/PalySlideImages/
      ↪DigitalSlide_C1M_4S_1_Annotations.xml',
         'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/PalySlideImages/
      ↪DigitalSlide_C1M_5S_1_Annotations.xml'
     ]



     # Define a function to extract labeled data from MRXS and XML files

     def extract_labeled_data(mrxs_path, xml_path):
         slide = openslide.OpenSlide(mrxs_path)  # Open the MRXS file using openslide

         tree = ET.parse(xml_path)  # Parse the XML file using ElementTree
         root = tree.getroot()

         annotations = root.find('destination/annotations')  # Find the annotations␣
      ↪element in the XML
```

```
    labeled_data = []
    for annotation in annotations:
        name = annotation.get('name')  # Extract the name attribute of the annotation
        type = annotation.get('type')  # Extract the type attribute of the annotation

        polygon_points = annotation.findall('p')  # Find all the p elements inside the annotation
        points = [(int(point.get('x')), int(point.get('y'))) for point in polygon_points]  # Extract the x and y coordinates from the p elements

        # Extract region of interest from MRXS image based on polygon points
        x_min = min([point[0] for point in points])  # Calculate the minimum x coordinate
        x_max = max([point[0] for point in points])  # Calculate the maximum x coordinate
        y_min = min([point[1] for point in points])  # Calculate the minimum y coordinate
        y_max = max([point[1] for point in points])  # Calculate the maximum y coordinate

        region = slide.read_region((x_min, y_min), 0, (x_max - x_min, y_max - y_min))  # Read the region of interest from the MRXS image

        annotation_details = {'name': name, 'type': type, 'points': points, 'xml_file': os.path.basename(xml_path)}
        labeled_data.append({'image': region, 'label': name, 'annotations': annotation_details})  # Append the labeled data to the list

    slide.close()  # Close the MRXS slide

    return labeled_data

# Iterate over the MRXS and XML file paths, extract labeled images
# and annotations and store the data in a dataframe
labeled_data_list = []
for mrxs_path, xml_path in zip(mrxs_paths, xml_paths):
    labeled_data = extract_labeled_data(mrxs_path,xml_path)
    labeled_data_list.extend(labeled_data)

df = pd.DataFrame(labeled_data_list)  # Create a dataframe from the labeled data

df['image'] = df['image'].apply(lambda img: img.convert('RGB'))  # Convert RGBA images to RGB
```

[ ]:

```
# Count the number of records in each class
class_counts = df['label'].value_counts()

# Filter out classes with only two record
#filtered_data = data[data['label'].map(class_counts) > 1]
data = df[df['label'].isin(class_counts[class_counts > 2].index)]
```

[ ]:

```
data.head()
```

[ ]:

```
classes =  sorted(data['label'].unique().tolist())
classes
```

[ ]:

```
data['image'][29]
```

[ ]:

```
data['label'][29]
```

[ ]:

```
data['annotations'][29]
```

[ ]:

```
# Count the unique classes
class_counts = data['label'].value_counts()

# Print the unique classes and their counts
for class_name, count in class_counts.items():
    print(f"Class: {class_name}, Count: {count}")
```

0.0.2 Data Preparation

0.0.3 Saving Labeled Images and the Corresponding XML Annotation Files

- 1) The following function saves labeled images and the corresponding annotation files.
- 2) The annotation details are saved as separate XML files with the .xml extension.
- 3) The function converts the annotation detail dictionary to an XML string and write it to the XML file.

[ ]:

```
## The function saves images and corresponding annotations files into two␣
 ↪separate folders.
##  This file saving format is required for the training Mask RCNN

def save_labeledimages_with_annotations(df, output_dir):
    '''
    The function accepts dataframe df and output folder
    as arguments and extract images and correpoinding xlm files
    and saves the data in the output folder with two subfolders
    'images' and 'annotations'
    '''
    # Create the output directory if it does not exist
    if not os.path.exists(output_dir):
        os.makedirs(output_dir)

    # Create subdirectories for images and annotations
    images_dir = os.path.join(output_dir, 'images')
    annotations_dir = os.path.join(output_dir, 'annotations')
    os.makedirs(images_dir, exist_ok=True)
    os.makedirs(annotations_dir, exist_ok=True)

    # Iterate over the rows of the DataFrame
    for i, row in df.iterrows():
        image = row['image']
        annotation_details = row['annotations']

        # Generate the file name based on the DataFrame index
        file_name = str(i)

        # Generate file paths for image and annotation
        file_path_image = os.path.join(images_dir, f"{file_name}.jpg")
        file_path_annotation = os.path.join(annotations_dir, f"{file_name}.xml")

        # Save the image using the file path
        image.save(file_path_image)

        # Save the annotation details as an XML file
        annotation_xml = generate_annotation_xml(annotation_details)
        with open(file_path_annotation, 'w') as xml_file:
            xml_file.write(annotation_xml)

        # Print the file paths for reference
        # print(f"Image {i+1} saved: {file_path_image}")
        # print(f"Annotation XML for Image {i+1}: {file_path_annotation}")

def generate_annotation_xml(annotation_details):
    # Create the XML structure and populate it with annotation details
    annotation = ET.Element('annotation')

    name = ET.SubElement(annotation, 'name')
    name.text = annotation_details['name']

    type = ET.SubElement(annotation, 'type')
    type.text = annotation_details['type']

    points = ET.SubElement(annotation, 'points')
    for x, y in annotation_details['points']:
        point = ET.SubElement(points, 'point')
        point.set('x', str(x))
        point.set('y', str(y))
    # Convert the XML structure to a string
    annotation_xml = ET.tostring(annotation).decode()

    return annotation_xml

# Implementation
#  The dataframe has 'label','image' , and 'annotations' columns

output_dir = 'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/Segmentation/
 ↪dataset'

# Save the images with numbering and corresponding annotation XML files
save_labeledimages_with_annotations(data, output_dir)
```

1 Training Mask RCNN.

[ ]:

```
# Check TF version
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)

import tensorflow as tf
tf.__version__
```

[ ]:

```
class MicrofossilDataset(Dataset):

        '''''
        This classs performs the following functions:
        - Load custom dataset used for training Mask RCNN
        - Extract bounding boxes from annotation details and load masks

        '''
        def __init__(self):
                # Initialize other attributes
                self.image_info = []


                self.class_names = ['Cribroperidinium␣
 ↪"prominoseptatum"','Fibrocysta axialis',
                'Palaeoperidinium pyrophorum','Senoniasphaera inornata',
                'Spongodinium delitiense', 'Spongodinium delitiense␣
 ↪(operculum)']

                self.class_info = [
                        {"id": 1, "name": 'Cribroperidinium "prominoseptatum"',␣
 ↪"source": "NPD data"},
            {"id": 2, "name": "Fibrocysta axialis", "source": "NPD data"},
            {"id": 3, "name": "Palaeoperidinium pyrophorum", "source": "NPD␣
 ↪data"},
                        {"id": 4, "name": "Senoniasphaera inornata", "source":␣
 ↪"NPD data"},
            {"id": 5, "name": "Spongodinium delitiense", "source": "NPD data"},
            {"id": 6, "name": "Spongodinium delitiense (operculum)", "source":␣
 ↪"NPD data"}
                ]

                # Reset source_class_ids dictionary
                self.source_class_ids = {}

        def __len__(self):
                return len(self.image_info)

        def add_image(self, source, image_id, path, annotation, height, width):
                image_info = {
                        'source': 'NPD data',
                        'id': int(image_id),
                        'path': path,
                        'annotation': annotation,
                        'height': height,
                        'width': width

                }
                self.image_info.append(image_info)


        def load_dataset(self, dataset_dir, class_names):

                images_dir = os.path.join(dataset_dir, 'images')
                annotations_dir = os.path.join(dataset_dir, 'annotations')

                # Get the list of image files sorted in ascending order
                image_files = sorted(os.listdir(images_dir), key=lambda x:␣
 ↪int(os.path.splitext(x)[0]))

                for filename in image_files:
                        image_id = os.path.splitext(filename)[0]  # Get the␣
 ↪image ID from the filename
                        img_path = os.path.join(images_dir, filename)
                        ann_path = os.path.join(annotations_dir, f'{image_id}.
 ↪xml')
                        img_path = img_path.replace("\\","/")
                        ann_path = ann_path.replace("\\","/")
```

```
                        if os.path.exists(ann_path):
                                image = cv2.imread(img_path)
                                height, width, _ = image.shape

                                self.add_image(
                                        'source',
                                        image_id=image_id,
                                        path=img_path,
                                        annotation=ann_path,
                                        height=height,
                                        width=width
                                )

        def extract_boxes(self, filename):
                tree = ET.parse(filename)
                root = tree.getroot()

                image_path = filename.replace("annotations", "images")
                image_path = os.path.splitext(image_path)[0] + '.jpg'
                image = Image.open(image_path)
                width, height = image.size

                boxes = []
                for point in root.findall('.//point'):
                        x = int(point.attrib['x'])
                        y = int(point.attrib['y'])
                        boxes.append([x, y])

                return np.array(boxes), width, height

        def polygons_to_mask(self, polygons, height, width):
                mask = np.zeros((height, width), dtype=np.uint8)
                for polygon in polygons:
                        rr, cc = skimage.draw.polygon(polygon[:, 1], polygon[:, 0])
                        rr = np.clip(rr, 0, height - 1)
                        cc = np.clip(cc, 0, width - 1)
                        mask[rr, cc] = 255
                return mask

        def load_mask(self, image_id):
                image_info = self.image_info[image_id]
                annotation_path = image_info['annotation']

                # print("ImageID:", image_id)
                # print("AnnPath:", annotation_path)
                # print(f'Image info for {image_id}: {image_info}')

                tree = ET.parse(annotation_path)
                root = tree.getroot()

                masks = []
                class_ids = []

                try:
                        name_element = root.find('name')
                        class_name = name_element.text
                        # print("ClassName:", class_name)

                        if class_name not in self.class_names:
                                masks = np.zeros((image_info['height'],␣
 ↪image_info['width'], 0), dtype=np.uint8)
                                class_ids = np.array([])
                        else:
                                class_id = self.class_names.index(class_name) +␣
 ↪1
                                class_ids.append(class_id)
                                # print("ClassID:", class_id)

                                points_element = root.find('points')
                                polygon_points = []
                                for point_element in points_element.
 ↪findall('point'):
                                        x = int(point_element.get('x'))
                                        y = int(point_element.get('y'))
                                        polygon_points.append([x, y])

                                polygon = np.array(polygon_points)
                                mask = self.polygons_to_mask([polygon],␣
 ↪image_info['height'], image_info['width'])
                                masks.append(mask)
                                # print("Number of masks:", len(masks))

                except Exception as e:
                        print("Error occurred:", str(e))

                return np.stack(masks, axis=-1), np.array(class_ids)
```

Training MRCNN

[ ]:

```
# Define the path to the dataset directory
dataset_dir = 'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/Segmentation/
 ↪dataset'
```

```
class_names = ['Cribroperidinium "prominoseptatum"','Fibrocysta axialis',
                'Palaeoperidinium pyrophorum','Senoniasphaera inornata',
                'Spongodinium delitiense', 'Spongodinium delitiense␣
 ↪(operculum)']


# Initialize the dataset
dataset = MicrofossilDataset()

# Load the dataset
dataset.load_dataset(dataset_dir, class_names)

# Set the random seed for reproducibility
np.random.seed(42)

# Shuffle the dataset indices
indices = np.arange(len(dataset.image_info))
np.random.shuffle(indices)

# Split the dataset into train and validation indices
split_ratio = 0.8  # 80% for training, 20% for validation
split_index = int(split_ratio * len(indices))
train_indices = indices[:split_index]
val_indices = indices[split_index:]

# Create the training and validation datasets
train_dataset = MicrofossilDataset()
val_dataset = MicrofossilDataset()

# Add the images and annotations to the training dataset
for idx in train_indices:
    image_info = dataset.image_info[idx]
    train_dataset.add_image(
        image_info['source'],
        image_id=image_info['id'],
        path=image_info['path'],
        annotation=image_info['annotation'],
        height=image_info['height'],
        width=image_info['width']
    )

# Add the images and annotations to the validation dataset
for idx in val_indices:
    image_info = dataset.image_info[idx]
    val_dataset.add_image(
        image_info['source'],
        image_id=image_info['id'],
        path=image_info['path'],
        annotation=image_info['annotation'],
        height=image_info['height'],
        width=image_info['width']
    )

# Prepare the datasets
train_dataset.prepare()
val_dataset.prepare()

# Set the batch size
batch_size = 1

print("Train records:",len(train_dataset))
print("Validation records.", len(val_dataset))
print("Batch Size:", batch_size)
```

[ ]:

```
# Configure RCNN model

class MRCNNConfig(Config):
    GPU_COUNT = 1
    NAME = "microfossil"
    IMAGES_PER_GPU = 1
    NUM_CLASSES = 6
    STEPS_PER_EPOCH = 50

# Define the Mask R-CNN configuration
config = MRCNNConfig()
#config.display()

# Directory to save logs and trained model
# MODEL_DIR = os.path.join(os.getcwd(), "logs")
MODEL_DIR = "C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/
 ↪ImageClassification/logs"

# Create the Mask R-CNN model
model = MaskRCNN(mode="training", model_dir=MODEL_DIR, config=config)
# model = MaskRCNN(mode="training", model_dir="./", config=config)

# Load the pre-trained COCO weights (excluding top layers)
pretrained_weights_path = "C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/
 ↪mask_rcnn_weights/mask_rcnn_coco.h5"
model.load_weights(pretrained_weights_path, by_name=True,␣
 ↪exclude=["mrcnn_class_logits", "mrcnn_bbox_fc", "mrcnn_bbox", "mrcnn_mask"])

# Update the log directory path
```

```
model.log_dir =  "logs"
# model.log_dir =  "./"



# Train the model
model.train(train_dataset=train_dataset, val_dataset=val_dataset,␣
 ↪learning_rate=0.001, epochs=1, layers="heads")

# The model does not save in the target directory
# However, it is saved mannually by copying the trained model with the path.
```

[ ]:

```
# Save trained weights manually after training the MaskRCNN model
model_path = 'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/
 ↪ImageClassification/mask_rcnn_microfossil_0001.h5'
model.keras_model.save_weights(model_path)
```

1.0.1 Make Predictions with the trained Mask RCNN model

[ ]:

```
config.display
```

[ ]:

```
class MRCNNConfig(Config):
    GPU_COUNT = 1
    NAME = "microfossil"
    IMAGES_PER_GPU = 1
    NUM_CLASSES = 6
    STEPS_PER_EPOCH = 50

# Define the Mask R-CNN configuration
config = MRCNNConfig()

# Load the trained model
model = modellib.MaskRCNN(mode="inference", config=config, model_dir="logs")
model_path = 'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/
 ↪ImageClassification/mask_rcnn_microfossil_0001.h5'
model.load_weights(model_path, by_name=True)

# Load the image
image_path = "C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/
 ↪ImageClassification/images/image40.jpg"
image = skimage.io.imread(image_path)

# Run the image through the model to get predictions
results = model.detect([image])

# Get the predicted class IDs, bounding boxes, scores, and class names
r = results[0]
```

```
class_ids = r['class_ids']
print(class_ids)
rois = r['rois']
scores = r['scores']
masks = r['masks']

class_names = ['Cribroperidinium "prominoseptatum"','Fibrocysta axialis',
                'Palaeoperidinium pyrophorum','Senoniasphaera inornata',
                'Spongodinium delitiense', 'Spongodinium delitiense␣
 ↪(operculum)']

# Convert lists to numpy arrays
rois = np.array(rois)
masks = np.array(masks)
class_ids = np.array(class_ids+1)
print("Class_ids:", class_ids)

# Visualize the image with the predicted bounding boxes, scores, and class␣
 ↪labels
fig, ax = plt.subplots(1)

# visualize.display_instances(image, rois, masks, class_ids, class_names,␣
 ↪scores,min_score=0.2, ax=ax, title="Prediction")

visualize.display_instances(image, rois, masks, class_ids, class_names,␣
 ↪scores,ax=ax, title="Prediction")
plt.show()
```

process_palyslides

July 14, 2023

```
[ ]: import os
     import cv2
     import math
     import pandas as pd
     import numpy as np
     import matplotlib.pyplot as plt
     from PIL import Image
     import requests
     import urllib.request

     #Set the "path_to_dll" variable to a specific directory path containing DLL␣
      ↪files.
     path_to_dll = "C:/Users/emmie/anacond3/Lib/site-packages/openslide/
      ↪openslide-win64-20230414/bin"
     os.environ["PATH"] = path_to_dll + ";" + os.environ["PATH"]

     import openslide
     from openslide.deepzoom import DeepZoomGenerator
```

0.1 Read and Manipulate Slide Images

The procedure described in the following Youtube video was followed in reading and manipulating slide images. Link: https://www.youtube.com/watch?v=QntLBvUZR5c

```
[ ]: #Load a slide image file (mrxs) into an object

     mrxs_path = "C:/Users/emmie/Thesis2023/paleo_slides/extracted_data/1_6-6 ST2␣
      ↪4876 mDC.mrxs"
     slide = openslide.OpenSlide(mrxs_path)
```

```
[ ]: # A preliminary view of slide properties

     slide_props = slide.properties
     print(f'The slide properties: {slide_props}')
```

```
[ ]: if 'openslide.mpp-x' in slide_props:
         print("Pixel size of X in um is:", slide_props['openslide.mpp-x'])
```

```
else:
    print("Key 'openslide.mpp-x' not found in slide_props dictionary.")
```

[ ]:

```
print("The vendor is:", slide_props['openslide.vendor'])
print("Pixel size of X in um is:", slide_props['openslide.mpp-x'])
print("Pixel size of Y in um is:", slide_props['openslide.mpp-y'])
```

[ ]:

```
# Get objective power used to capture the image
# The objective power indicates the level of magnification achieved by the objective lens.
lens_objective_power = float(slide.properties[openslide.PROPERTY_NAME_OBJECTIVE_POWER])
print("The objective power is:", lens_objective_power)
```

[ ]:

```
# Get a thumbnail of the slide with dimensions of 500 x 500 pixels.
slide_thumbnail_500 = slide.get_thumbnail(size=(500,500))
slide_thumbnail_500.show()
```

[ ]:

```
# Convert the thumbnail slide image to numpy array
slide_thumbnail_500_np = np.array(slide_thumbnail_500)
plt.figure(figsize=(10,10))
plt.imshow(slide_thumbnail_500_np)
```

[ ]:

```
# Dimensions and downsampling factors of different levels in the slide image pyramid

# Get dimensions of each level in the slide image
level_dims = slide.level_dimensions
print("The dimensions of different levels in the slide image are:", level_dims)

# Get the number of levels in the slide image
num_levels = len(level_dims)
print("The number of levels in the slide image is:", num_levels)


# The downsampling factors for each level.
# Show how much the levels are downsampled compared with the original image dimensions
downsample_factors = slide.level_downsamples
print("The various levels are downsampled by factors:", downsample_factors)
```

[ ]:

```
# Return a level in the slide image given the downsample factor.
downsample_factor = 512
best_level = slide.get_best_level_for_downsample(downsample_factor)
print (f"The level number in the slide image given {downsample_factor} as the downsample factor is {best_level}.")
```

0.1.1 Generating tiles for deep learning tasks

- A read_region function can be used to slide over a slide image to extract tiles.
- An easier approach would be to use **DeepZoom based generator**.
- A tile is a single image of dimensions(x,y) extracted from a slide at a specified deep zoom level. A slide image can contain thousands of tiles.

[ ]:

```
# Create a generator object "tiles" from the slide image based on the specified tile size and overlap.

tiles = DeepZoomGenerator(slide, tile_size=256,overlap=0,limit_bounds=False) # tile_size: 256,512
```

[ ]:

```
## Number of zoom levels in the object "tiles" representing the slide image
# The index of the zoom levels start with zero to (maximum zoom level - 1)
deepzoom_levels_num = tiles.level_count # Return the number of deep zoom levels in the generated "tiles" object
print(f"The number of zoom levels present in the generated object tiles (slide image) is:{deepzoom_levels_num}")

## Dimensions of tiles at each zoom level.
dims_zoomLevel = tiles.level_dimensions
print(f"Dimensions of tiled images at each zoom level: {dims_zoomLevel}.")

## Total number of 256 x 256 tiles present in the tiled image pyramid
# For each zoom level, compute nrows and ncolumns based on the specified tile size 256,
# multiply nrows with ncolumns, and sum the products for all the zoom levels.

# total_tiles = tiles.tile_count  # This method gives the total tiles across all zoom levels

total_tiles = 0
for row, col in dims_zoomLevel:
    num_tiles_row = math.ceil(row / 256)
    num_tiles_col = math.ceil(col / 256)
    level_tiles = num_tiles_row * num_tiles_col
    total_tiles += level_tiles
print(f"The total number of tiles across all the zoom levels is: {total_tiles}.")

## Number of tiles of size 256 pixels at a specific zoom level
zoom_level = 11                                         # This level has 975 x 1027 tiled images arranged in rows and columns, respectively.
level_tiles_shape = tiles.level_tiles[zoom_level]       # Get the dimensions based on specified tile size: ceil(975/256), ceil(1027/256)
```

```
tiles_total = level_tiles_shape[0] * level_tiles_shape[1] # Mutiply Xdim with␣
 ↪Ydim
print(f"The shape of tiles at the zoom level {zoom_level} is:
 ↪{level_tiles_shape}")
print(f"This means there are {tiles_total} tiles of size 256 at the specified␣
 ↪zoom level.")

## Number of tiles at the highest zoom level in tiled image pyramid
maxzoom_IDX = deepzoom_levels_num-1
tile_shape_maxzoom = tiles.level_tiles[maxzoom_IDX]
total_tiles_maxzoom = tile_shape_maxzoom[0]*tile_shape_maxzoom[1]
print(f"The shape of tiles at the highest level containing␣
 ↪{total_tiles_maxzoom} tiles is: {tile_shape_maxzoom}.")

## Dimensions of specific tile at a specified zoom level and indices
# The height and width of each tile are 256 pixels, respectively.
# However, tiles of smaller dimensions/size can be generated and
# should be removed from the dataset during preprocessing.

zoom_level = 11
tile_IDX = (1, 1)  # The indices of the tile to be retrieved

M = level_tiles_shape[0] # Height/nrow of tiles at the specified zoom level
N = level_tiles_shape[1] #  Width/ncolums of tiles at the specified zoom level
tile_dims = tiles.get_tile_dimensions(zoom_level, tile_IDX)  # Get tile␣
 ↪dimensions at the specified zoom level and indices
tile_last_dims = tiles.get_tile_dimensions(11, ((M-1),(N-1))) # Retrieve the␣
 ↪last tile dimensions at the specified zoom level
print(f"The dimensions a tile at the zoom level {zoom_level} with indices␣
 ↪{tile_IDX} are: {tile_dims}.")
print(f"The dimensions of the last tile at the zoom level {zoom_level} are:␣
 ↪{tile_last_dims}.")

# Visualize a single tile a specified zoom level

single_tile = tiles.get_tile(zoom_level, tile_IDX) # Retrieve a tile with the␣
 ↪specified indices
single_tile_RGB = single_tile.convert('RGB')
single_tile_RGB.show()
```

0.1.2 Tile Generation with YOLOv4

- Detect images on tiles with YOLOv4-tiny to generate only desirable tiles.

[ ]:

```
# Download YOLOv4 on Jupyter by running the following command
# !git clone https://github.com/AlexeyAB/darknet.git
```

[ ]:
```
# %cd darknet
```

[ ]:
```
# Download YOLOv4-tiny configuration weights files


# # URL for YOLOv4-tiny configuration file
# url = 'https://github.com/AlexeyAB/darknet/blob/master/cfg/yolov4-tiny.cfg?
 ↪raw=true'
# # Save the weights file to the specified folder
# config_path = "C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/darknet/
 ↪yolov4-tiny.cfg"
# urllib.request.urlretrieve(url, config_path)


# # URL for YOLOv4-tiny weights file
# url = 'https://github.com/AlexeyAB/darknet/releases/download/
 ↪darknet_yolo_v4_pre/yolov4-tiny.weights'

# # Save the weights file to the specified folder
# weights_path = "C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/darknet/
 ↪yolov4-tiny.weights"
# urllib.request.urlretrieve(url, weights_path)
```

[ ]:
```
# Check if YOLOv4 network is properly installed
# if hasattr(cv2.dnn, 'DNN_BACKEND_DEFAULT'):
#     print('YOLOv4 is installed!')
# else:
#     print('YOLOv4 is not installed.')
```

0.1.3 Object Detection with YOLOv4-tiny Network

- This algorithm performs object detection using YOLOv4-tiny network.
- Unwanted tiles are removed from the dataset after preprocessing.

[ ]:
```
import cv2
import numpy as np
import os

def generate_clean_tiles(tiles, output_dir):
    '''
    The function accepts tile object "tiles" and the output flolder
    and performs object detection with YOLOv4 Tiny to remove undesireble
    tiles and generate tiled images of good quality
    '''
    # Load weights and configuration files of the YOLO4-tiny network
    weights_path = "C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/darknet/
 ↪yolov4-tiny.weights"
```

```
    config_path = "C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/darknet/
↪yolov4-tiny.cfg"
    net = cv2.dnn.readNet(weights_path, config_path)

    # Define the output layers of the network
    layer_names = net.getLayerNames()
    output_layers = [layer_names[i - 1] for i in net.getUnconnectedOutLayers()]

    # Create the output directory if it does not exist
    if not os.path.exists(output_dir):
        os.makedirs(output_dir)

    # Process and save tiles
    maxzoom_IDX = 16
    slide_code = 487

    for col in range(0, tiles.level_tiles[maxzoom_IDX][0]):
        for row in range(0, tiles.level_tiles[maxzoom_IDX][1]):
            try:
                tile = tiles.get_tile(maxzoom_IDX, (col, row))
                tile_array = np.array(tile)
                gray = cv2.cvtColor(tile_array, cv2.COLOR_RGB2GRAY)
                unique_values = np.unique(gray)

                # Verify if the tile contains colored microfossils
                if len(unique_values) >= 2 and unique_values[0] != 255 and␣
↪unique_values[1] != 255:
                    # Verify if the mean pixel intensity is above a specified␣
↪threshold
                    if gray.mean() > 100:
                        # Verify if the proportion of white pixels is below a␣
↪specified threshold
                        white_percent = np.count_nonzero(gray == 255) / gray.
↪size
                        if white_percent < 0.9:
                            # Check if the tile contains an object of interest
                            img = cv2.cvtColor(tile_array, cv2.COLOR_RGB2BGR)
                            blob = cv2.dnn.blobFromImage(img, 1 / 255, (416,␣
↪416), swapRB=True, crop=False)
                            net.setInput(blob)
                            detections = net.forward(output_layers)

                            # Check if the detected object belongs to a␣
↪specific class and meets the confidence threshold
                            for detection in detections[0]:
                                scores = detection[5:]
                                class_id = np.argmax(scores)
                                confidence = scores[class_id]

                                if class_id == 0 and confidence > 0.2:
                                    # Check if the image is dark or black
                                    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
                                    if np.mean(gray) > 10:
                                        # Save the tile as an image
                                        tile_path = os.path.join(output_dir,␣
 ↪f"tile_{slide_code}_{col}_{row}.jpg")
                                        tile.save(tile_path)
            except Exception as e:
                print(f"Error processing tile at column {col} and row {row}:␣
 ↪{str(e)}")

# Implementation

output_dir = "C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/clean_tiles"
generate_clean_tiles(tiles, output_dir)
```

0.1.4 Watershed Segmentation

[ ]:

```
# This algorithm performs watershed segmentation task based on the approach␣
 ↪oulined in the link:
#https://docs.opencv.org/4.x/d3/db4/tutorial_py_watershed.html

# Set the path to the input folder containing the tiled images
input_folder_path = "C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/
 ↪watershed_images"

# Set the path to the output folder for saving the segmented images
output_folder_path = "C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/
 ↪watershed_outputimages"

# Create the output folder if it does not exist
if not os.path.exists(output_folder_path):
    os.makedirs(output_folder_path)

# Loop through all the images in the input folder
for filename in os.listdir(input_folder_path):
    if filename.endswith('.jpg') or filename.endswith('.png'):
        # Load and convert the image to grayscale
        input_img_path = os.path.join(input_folder_path, filename)
        img = cv2.imread(input_img_path)
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        # Apply thresholding to separate the foreground (microfossils) from the␣
↪background
        ret, thresh = cv2.threshold(gray, 0, 255, cv2.THRESH_BINARY_INV+cv2.
↪THRESH_OTSU)

        # Apply morphological opening to remove small objects and noise
        kernel = np.ones((3,3), np.uint8)
        opening = cv2.morphologyEx(thresh, cv2.MORPH_OPEN, kernel, iterations=2)

        # Apply distance transform to obtain the distance map
        dist_transform = cv2.distanceTransform(opening, cv2.DIST_L2, 5)

        # Apply thresholding to the distance map to obtain the foreground␣
↪markers
        ret, fg_markers = cv2.threshold(dist_transform, 0.5*dist_transform.
↪max(), 255, 0)

        # Apply watershed algorithm to segment the foreground markers and␣
↪obtain the labels
        fg_markers = np.uint8(fg_markers)
        unknown = cv2.subtract(opening, fg_markers)
        ret, bg_markers = cv2.threshold(unknown, 0, 255, cv2.THRESH_BINARY_INV)
        bg_markers = cv2.dilate(bg_markers, kernel, iterations=3)
        markers = cv2.add(fg_markers, bg_markers)

        # Convert the grayscale image to a 3-channel format
        img_color = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

        # Apply the watershed segmentation algorithm
        markers = markers.astype(np.int32)
        markers_copy = markers.copy()
        cv2.watershed(img_color, markers_copy)

        # Visualize the segmented image and the markers
        img_color[markers_copy == -1] = [255,0,0]

        # Save the segmented image to the output folder
        output_img_path = os.path.join(output_folder_path, filename[:-4]+'_seg.
↪jpg')
        cv2.imwrite(output_img_path, img_color)
```

classify_microfossils

July 14, 2023

[ ]:

```
import os
import keras
import pandas as pd
import hashlib
import cv2
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
import tensorflow as tf
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
from keras.models import save_model, load_model
import xml.etree.ElementTree as ET
import shutil
import pickle
from pathlib import Path

# # Set the "path_to_dll" variable to a specific directory path containing DLL␣
 ↪files.
path_to_dll = 'C:/Users/emmie/anacond3/Lib/site-packages/openslide/
 ↪openslide-win64-20230414/bin'
os.add_dll_directory(path_to_dll)
import openslide
```

0.0.1 Data Preprocessing for Classification Task

- 1) Extract labeled images with the slide image MRXS files and corresponding annotation XML files.
- 2) Store the dictionary object in a pandas dataframe.
- 3) Clean the dataframe to remove classes that contain less than 2 records.
- 4) Split the filtered dataframe into training and validation datasets.
- 5) Save filtered images in the respective dataframes in output directories.

Labeled images are numbered according to their respective class counts, eg. 0,1, 2,...N-1, where N = total class records. Alternatively, labeled images can be numbered according to their dataframe indices that relate to their positions on the dataframe.

[ ]:

```
# Extract labeled images from annotated palyslides using MRXS and XML files
# Group the labeled data into 21 subfolders according to their class labels

# MRXS and XML file paths:

mrxs_paths = [
    'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/PalySlideImages/2_4-C-11␣
 ↪10052.7 ftC = 3064 mC.mrxs',
    'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/PalySlideImages/2_4-C-11␣
 ↪10070 ftC = 3069.3 mC.mrxs',
    'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/PalySlideImages/2_7-14␣
 ↪10658ft 8in C.mrxs',
    'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/PalySlideImages/16_3-2␣
 ↪1998.80 mC.mrxs',
    'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/PalySlideImages/
 ↪DigitalSlide_C1M_4S_1.mrxs',
    'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/PalySlideImages/
 ↪DigitalSlide_C1M_5S_1.mrxs'

]

xml_paths = [
    'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/PalySlideImages/2_4-C-11␣
 ↪10052.7 ftC = 3064 mC_Annotations.xml',
    'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/PalySlideImages/2_4-C-11␣
 ↪10070 ftC = 3069.3 mC_Annotations.xml',
    'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/PalySlideImages/2_7-14␣
 ↪10658ft 8in C_Annotations.xml',
    'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/PalySlideImages/16_3-2␣
 ↪1998.80 mC_Annotations.xml',
    'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/PalySlideImages/
 ↪DigitalSlide_C1M_4S_1_Annotations.xml',
    'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/PalySlideImages/
 ↪DigitalSlide_C1M_5S_1_Annotations.xml'
]

# Define a function to extract labeled images from MRXS and XML files

def extract_labeled_images(mrxs_path, xml_path):

    slide = openslide.OpenSlide(mrxs_path)

    tree = ET.parse(xml_path)
    root = tree.getroot()

    annotations = root.find('destination/annotations')

    labeled_data = []
    for annotation in annotations:
        name = annotation.get('name')
        type = annotation.get('type')

        polygon_points = annotation.findall('p')
        points = [(int(point.get('x')), int(point.get('y'))) for point in polygon_points]

        # Extract region of interest from MRXS image based on polygon points given in the XML file.
        x_min = min([point[0] for point in points])
        x_max = max([point[0] for point in points])
        y_min = min([point[1] for point in points])
        y_max = max([point[1] for point in points])

        region = slide.read_region((x_min, y_min), 0, (x_max - x_min, y_max - y_min))

        labeled_data.append({'image': region, 'label': name})

    slide.close()

    return labeled_data

# Iterate over the MRXS and XML file paths, extract labeled images,
# and store the data in a Pandas dataframe
labeled_data_list = []
for mrxs_path, xml_path in zip(mrxs_paths, xml_paths):
    labeled_data = extract_labeled_images(mrxs_path, xml_path)
    labeled_data_list.extend(labeled_data)

data = pd.DataFrame(labeled_data_list)

data['image'] = data['image'].apply(lambda img: img.convert('RGB'))  # Convert to RGBA images to RGB
```

[ ]:

```
# Display fist 5 image records in the dataframe.

data.head(10)
```

[ ]:

```
classes =  sorted(data['label'].unique().tolist())
classes
```

[ ]:

```
# Count the unique classes
class_counts = data['label'].value_counts()

# Print the unique classes and their counts
for class_name, count in class_counts.items():
    print(f"Class: {class_name}, Count: {count}")
```

[ ]:

```
data['image'][0]
```

[ ]:

```
data['label'][0]
```

Print labeled images

[ ]:

```
# Assuming the dataframe is named df with "label" and "image" columns
selected_indices = [51]  # Specify the indices of the images you want to print

for idx in selected_indices:
    row = data.iloc[idx]  # Access the row at the specified index
    label = row["label"]
    image = row["image"]

    # Display the image with its label
    plt.imshow(image)
    plt.title(label)
    plt.axis("off")
    plt.show()
```

0.0.2 Split the dataframe into training and validation datasets

[ ]:

```
# Count the number of records per class
class_counts = data['label'].value_counts()

# Filter out classes with only one record
#filtered_data = data[data['label'].map(class_counts) > 1]
filtered_data = data[data['label'].isin(class_counts[class_counts > 2].index)]

# Get the unique classes
classes = sorted(filtered_data['label'].unique().tolist())

# Split the filtered data into training and validation sets with the same␣
 ↪classes
train_data = pd.DataFrame()
val_data = pd.DataFrame()
for label in classes:

    class_data = filtered_data[filtered_data['label'] == label]
    class_train_data, class_val_data = train_test_split(class_data, test_size=0.
 ↪2, random_state=42)
    train_data = pd.concat([train_data, class_train_data])
    val_data = pd.concat([val_data, class_val_data])


# Get records using using len() function
print("Number of training image data:", len(train_data))

# Get the number of records using the .shape attribute
print("Number of validation image data:", val_data.shape[0])

# Get number of unique class labels
train_num_classes = len(train_data['label'].unique().tolist())
val_num_classes  = len(val_data['label'].unique().tolist())
print(f'Number of training class labels: {train_num_classes}')
print(f'Number of validation class labels: {val_num_classes}')

classes =  train_data['label'].unique().tolist()
classes
```

```
[ ]: # Get classes with sizes
     class_counts = val_data['label'].value_counts()
     class_counts
```

0.0.3 Save labeled images as files with unique paths

```
[ ]: def save_labeledimages(df, output_dir):

         """
         Save pandas images as files with associated labels linked with the files.

         Args:
         data (dataFrame): Dataframe containing 'image' and 'label' columns.
         output_dir (str): Output directory to save the images.

         """


         # Create the output directory if it does not exist
         if not os.path.exists(output_dir):
             os.makedirs(output_dir)

         # Create a dictionary to keep track of the numbering for each class label
         label_count = {}

         # Iterate over the rows of the DataFrame
```

```
    for i, row in df.iterrows():
        image = row['image']
        label = row['label']

        # Check if the label exists in the dictionary, if not initialize the␣
 ↪count to 0
        if label not in label_count:
            label_count[label] = 0

        # Generate a unique file path for each image based on the label and␣
 ↪count
        file_path = os.path.join(output_dir, f"{label}", f"{label_count[label]}.
 ↪jpg")

        # Replace backslashes with forward slashes in the image path
        file_path = file_path.replace("\\", "/")
        file_path = file_path.replace('"', '')

        # Create the subfolder for the label if it does nt exist
        label_dir = os.path.dirname(file_path)
        if not os.path.exists(label_dir):
            os.makedirs(label_dir)

        # Save the image using the file path
        image.save(file_path)  # The column values should be PIL Image objects

        # If the column values are numpy arrays representing images
        # cv2.imwrite(file_path, image)

        # Print the file path
        #print(f"Image {i+1} saved with label {label}: {file_path}")

        # Increment the count for the current label
        label_count[label] += 1

# Implementation:

# A dataFrame named 'df' contains 'image' and 'label' columns
train_dir = 'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/
 ↪ImageClassification/CNN_data/train_data'
val_dir = 'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/ImageClassification/
 ↪CNN_data/val_data'
labeled_data_dir = 'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/
 ↪ImageClassification/CNN_data/labeled_data'

# Save the images with numbering
```

```
save_labeledimages(train_data, train_dir) # Save only training dataset
save_labeledimages(val_data, val_dir)     # Save only validation dataset
#save_labeledimages(data, labeled_data_dir) # Save all the images in the␣
 ↪dataframe
```

0.0.4 Image Classification with CNN

Training the Deep Learning Model

- The classification algorithm CNN is trained with training and validation datasets to build a classifier (trained model)

[ ]:

```
def build_and_train_cnn(train_dir, val_dir, image_size, batch_size,␣
 ↪num_classes, num_epochs):
    """
    Build and train a CNN model with the training and validation directories.

    Args:
        train_dir (str): Directory path for the training dataset.
        val_dir (str): Directory path for the validation dataset.
        image_size (tuple): Tuple specifying the target image size, e.g.,␣
 ↪(width, height).
        batch_size (int): Batch size for training.
        num_classes (int): Number of classes in the classification task.
        num_epochs (int): Number of training epochs.

    Raises:
        ValueError: If the number of classes does not match the number of␣
 ↪unique labels in the datasets.
    """

    # Data augmentation and normalization for training set
    train_datagen = ImageDataGenerator(
        rescale=1./255,
        rotation_range=20,
        width_shift_range=0.2,
        height_shift_range=0.2,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True
    )

    # Only rescaling for validation set
    val_datagen = ImageDataGenerator(rescale=1./255)

    # Load and augment the training dataset
    train_generator = train_datagen.flow_from_directory(
        train_dir,
        target_size=image_size,
        batch_size=batch_size,
        class_mode='categorical'
    )

    # Load and augment the validation dataset
    val_generator = val_datagen.flow_from_directory(
        val_dir,
        target_size=image_size,
        batch_size=batch_size,
        class_mode='categorical'
    )

    # Check if the number of classes matches the number of unique labels in the␣
↪datasets
    if num_classes != len(train_generator.class_indices):
        raise ValueError("Number of classes doesn't match the number of unique␣
↪labels in the datasets.")


    # Build the CNN model
    model = tf.keras.models.Sequential([
        tf.keras.layers.Conv2D(32, (3, 3), activation='relu',␣
↪input_shape=(image_size[0], image_size[1], 3)),
        tf.keras.layers.MaxPooling2D((2, 2)),
        tf.keras.layers.Conv2D(64, (3, 3), activation='relu'),
        tf.keras.layers.MaxPooling2D((2, 2)),
        tf.keras.layers.Conv2D(128, (3, 3), activation='relu'),
        tf.keras.layers.MaxPooling2D((2, 2)),
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(128, activation='relu'),
        tf.keras.layers.Dense(num_classes, activation='softmax')
    ])


    # Compile the model
    model.compile(optimizer='adam',
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])

    # # Train the model
    # model.fit(train_generator,
    #           steps_per_epoch=train_generator.samples // batch_size,
    #           validation_data=val_generator,
    #           validation_steps=val_generator.samples // batch_size,
    #           epochs=num_epochs)
    # Train the model and obtain the history object
    history = model.fit(train_generator,
                        steps_per_epoch=train_generator.samples // batch_size,
                        epochs=num_epochs,
                        validation_data=val_generator,
                        validation_steps=val_generator.samples // batch_size)

    # # Convert the function model to a tf.keras model object
    # model = tf.keras.Model(inputs=model_input, outputs=model_output)

    # Save the training history
    history_model_path  = 'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/
 ↪ImageClassification/training_history.pkl'

    with open(history_model_path, 'wb') as file:
        pickle.dump(history.history, file)

    # Save the model
    model_path  = 'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/
 ↪ImageClassification/model.h5'
    #save_model(model, model_path)
    model.save(model_path)

    # Return the history object
    return history
```

[ ]:

```
# Set the paths to the train and validation directories

train_dir = "C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/
 ↪ImageClassification/CNN_data2/train_data"
val_dir = "C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/ImageClassification/
 ↪CNN_data2/val_data"

# Set the image size and batch size
image_size = (224, 224)
batch_size = 32

# Set the number of classes and epochs
num_classes = 6 #13
num_epochs = 17

# Build and train the CNN model
model_history = build_and_train_cnn(train_dir, val_dir, image_size, batch_size,␣
 ↪num_classes, num_epochs)
```

0.1 Model Training Evaluation

- Plot training/validation accuracy and loss

[ ]:

```
def plot_training_history(history):
    """
    Plot the training and validation accuracy versus loss from the training␣
 ↪history.

    Args:
        history (History): Training history object returned by model.fit().
    """

    # Extract the epochs and metrics from the history object
    epochs = range(1, len(history['loss']) + 1)

    # Access the training metrics and loss
    training_loss = history['loss']
    training_accuracy = history['accuracy']

    # Access the validation metrics and loss
    val_loss = history['val_loss']
    val_accuracy = history['val_accuracy']

    # Plot the accuracy
    plt.figure(figsize=(12, 6))
    plt.subplot(1, 2, 1)
    plt.plot(training_accuracy, label='Training Accuracy')
    plt.plot(val_accuracy, label='Validation Accuracy')
    plt.title('Training and Validation Accuracy')
    plt.ylim(0.3, 0.9)
    plt.xticks(np.arange(0, len(epochs) + 1, 2.0))
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend(loc='upper left')

    # Plot the loss
    plt.subplot(1, 2, 2)
    plt.plot(training_loss, label='Training Loss')
    plt.plot(val_loss, label='Validation Loss')
    plt.title('Training and Validation Loss')
    plt.ylim(0.4, 2)
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.xticks(np.arange(0, len(epochs)+1, 2.0))
    plt.legend(loc='upper right')

    # Show the plot
    plt.tight_layout()
    plt.show()
```

[ ]:

```
# Load the saved model
model_path = 'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/
 ↪ImageClassification/model.h5'
model = load_model(model_path)

# Load the training history
history_model_path  = 'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/
 ↪ImageClassification/training_history.pkl'
with open(history_model_path, 'rb') as file:
    history = pickle.load(file)

# Access the training metrics and loss
training_loss = history['loss']
training_accuracy = history['accuracy']

# Access the validation metrics and loss
validation_loss = history['val_loss']
validation_accuracy = history['val_accuracy']

# # Print the training and validation metrics
# print("Training Loss:", training_loss)
# print("Training Accuracy:", training_accuracy)
# print("Validation Loss:", val_loss)
# print("Validation Accuracy:", val_accuracy)

# From  the trained model and history object
plot_training_history(history)
```

Make a prediction

- Predict the class label of an unknown image.

[ ]:

```
def resize_image(image_path, desired_size):
    try:
        # Check if the image file exists
        if not os.path.exists(image_path):
            print(f"Image file '{image_path}' not found.")
            return None

        # Load the image using OpenCV
        image = cv2.imread(image_path)

        # Check if the image is loaded successfully
        if image is None:
            print(f"Failed to load image '{image_path}'.")
            return None

        # Resize the image
        resized_image = cv2.resize(image, desired_size)

        # Convert the resized image to RGB format
        resized_image = cv2.cvtColor(resized_image, cv2.COLOR_BGR2RGB)

        return resized_image

    except Exception as e:
        print(f"Error occurred while resizing image '{image_path}': {e}")
        return None

# Specify the path to the image file and the desired size
image_path = 'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/
 ↪ImageClassification/images/image24.jpg'
desired_size = (224, 224)  # Example size


# Resize the image
resized_image = resize_image(image_path, desired_size)

# Check if the image was resized successfully
if resized_image is not None:
    # Perform further processing with the resized image
    print("Image resized successfully.")

# Load the pre-trained model
model_path = 'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/
 ↪ImageClassification/model.h5'
#model_path = 'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/
 ↪ImageClassification/trained_cnn_models/model.h5'
model = load_model(model_path)

# Get the model's class indices
model_class_indices = model.predict(np.zeros((1, 224, 224, 3))).argmax(axis=1)


# Define the class labels
class_labels = [
    #'Acanthaulax venusta',
    'Cribroperidinium "prominoseptatum"',
    #'Dingodinium tuberculosum',
    #'Dingodinium tuberosum',
    'Fibrocysta axialis',
```

```python
    #'Gonyaulacysta jurassica',
    'Palaeoperidinium pyrophorum',
    'Senoniasphaera inornata',
    #'Sentusidinium pilosum',
    'Spongodinium delitiense',
    'Spongodinium delitiense (operculum)'
    #'Systematophora areolata',
    #'Tubotuberella apatela'
]

# Create a dictionary mapping the model's class indices to the class labels
class_mapping = {index: label for index, label in enumerate(class_labels)}

# print("Reordered Class Labels:")
# for index, label in class_mapping.items():
#     print(f"Index {index}: {label}")


# Preprocess the image
preprocessed_image = np.expand_dims(resized_image, axis=0)
preprocessed_image = preprocessed_image / 255.0  # Normalize pixel values to the range [0, 1]

# Perform the prediction
predictions = model.predict(preprocessed_image)

# Get the predicted class label
predicted_class_index = np.argmax(predictions[0])

# Print the predicted class index
print("Predicted Class Index:", predicted_class_index)

predicted_class_label = class_labels[predicted_class_index]
print(f'Predicted label: {predicted_class_label}')


# Reshape the image array to remove the batch dimension
image_array = np.squeeze(preprocessed_image, axis=0)

# Display the image with the predicted class label
plt.imshow(image_array)
plt.title(f'Predicted Label: {predicted_class_label}')
plt.axis('off')
plt.show()
```

0.1.1 Model Evaluation

The following algorith iterates through the folder of test images, preprocess the images, make predictions, evaluate the model perfromance using confusion matrix metrics and display images with predicted labels.

[ ]:

```
# Import necessary libraries
import os
import re
from natsort import natsorted
import seaborn as sns
import numpy as np
from PIL import Image
import cv2
from sklearn.metrics import␣
  ↪f1_score,accuracy_score,classification_report,confusion_matrix
import matplotlib.pyplot as plt
from keras.models import load_model
```

[ ]:

```
def resize_image(image_path, desired_size):
    try:
        # Check if the image file exists
        if not os.path.exists(image_path):
            print(f"Image file '{image_path}' not found.")
            return None

        # Load the image using OpenCV
        image = cv2.imread(image_path)

        # Check if the image is loaded successfully
        if image is None:
            print(f"Failed to load image '{image_path}'.")
            return None

        # Resize the image
        resized_image = cv2.resize(image, desired_size)

        # Convert the resized image to RGB format
        resized_image = cv2.cvtColor(resized_image, cv2.COLOR_BGR2RGB)

        return resized_image

    except Exception as e:
        print(f"Error occurred while resizing image '{image_path}': {e}")
        return None


# Define the directory of test images and the desired size
```

```
image_dir = 'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/
 ↪ImageClassification/test_images'
desired_size = (224, 224)

# Load trained model
model_path = 'C:/Users/emmie/OneDrive/Skrivebord/MasterThesis/
 ↪ImageClassification/model.h5'
model = load_model(model_path)

# Define class labels
class_labels = [
    #'Acanthaulax venusta',
    'Cribroperidinium "prominoseptatum"',
    # 'Dingodinium tuberculosum',
    # 'Dingodinium tuberosum',
    'Fibrocysta axialis',
    # 'Gonyaulacysta jurassica',
    'Palaeoperidinium pyrophorum',
    'Senoniasphaera inornata',
    # 'Sentusidinium pilosum',
    'Spongodinium delitiense',
    'Spongodinium delitiense (operculum)'
#     'Systematophora areolata',
#     'Tubotuberella apatela'
]

# Create a dictionary to map image filenames to true labels
image_label_map = {


    '3.jpg': 'Cribroperidinium "prominoseptatum"',
    '4.jpg': 'Cribroperidinium "prominoseptatum"',
    '5.jpg': 'Cribroperidinium "prominoseptatum"',
    '6.jpg': 'Cribroperidinium "prominoseptatum"',
    '7.jpg': 'Cribroperidinium "prominoseptatum"',
    '12.jpg': 'Fibrocysta axialis',
    '13.jpg': 'Fibrocysta axialis',
    '14.jpg': 'Fibrocysta axialis',
    '15.jpg': 'Fibrocysta axialis',
    '16.jpg': 'Fibrocysta axialis',
    '17.jpg': 'Fibrocysta axialis',
    '18.jpg': 'Fibrocysta axialis',
    '19.jpg': 'Fibrocysta axialis',
    '20.jpg': 'Fibrocysta axialis',
    '23.jpg': 'Palaeoperidinium pyrophorum',
    '24.jpg': 'Palaeoperidinium pyrophorum',
    '25.jpg': 'Palaeoperidinium pyrophorum',
    '26.jpg': 'Palaeoperidinium pyrophorum',
    '27.jpg': 'Palaeoperidinium pyrophorum',
    '28.jpg': 'Palaeoperidinium pyrophorum',
    '29.jpg': 'Palaeoperidinium pyrophorum',
    '30.jpg': 'Palaeoperidinium pyrophorum',
    '31.jpg': 'Palaeoperidinium pyrophorum',
    '32.jpg': 'Senoniasphaera inornata',
    '33.jpg': 'Senoniasphaera inornata',
    '34.jpg': 'Senoniasphaera inornata',
    '35.jpg': 'Senoniasphaera inornata',
    '36.jpg': 'Senoniasphaera inornata',
    '37.jpg': 'Senoniasphaera inornata',
    '38.jpg': 'Senoniasphaera inornata',
    '39.jpg': 'Senoniasphaera inornata',
    '40.jpg': 'Senoniasphaera inornata',
    '43.jpg': 'Spongodinium delitiense',
    '44.jpg': 'Spongodinium delitiense',
    '45.jpg': 'Spongodinium delitiense',
    '46.jpg': 'Spongodinium delitiense',
    '47.jpg': 'Spongodinium delitiense',
    '48.jpg': 'Spongodinium delitiense',
    '49.jpg': 'Spongodinium delitiense',
    '50.jpg': 'Spongodinium delitiense',
    '51.jpg': 'Spongodinium delitiense (operculum)',
    '52.jpg': 'Spongodinium delitiense (operculum)',
    '53.jpg': 'Spongodinium delitiense (operculum)',
    '54.jpg': 'Spongodinium delitiense (operculum)',
    '55.jpg': 'Spongodinium delitiense (operculum)',
    '56.jpg': 'Spongodinium delitiense (operculum)',

    # Add more filename-label pairs as needed
}

# Initialize empty lists for true and predicted labels
true_labels_list = []
predicted_labels = []
predicted_images = []

# Get the file names in the directory and sort them naturally
image_filenames = natsorted(os.listdir(image_dir))

for filename in image_filenames:
    if filename.endswith('.jpg') or filename.endswith('.png'):

        # Construct the full image path
        image_path = os.path.join(image_dir, filename)
        image_path = image_path.replace('\\','/')  # Replace backslashes with forward slash

        # Resize the image
        resized_image = resize_image(image_path, desired_size)

        # Check if the image was resized successfully
        if resized_image is not None:
            # Perform further processing with the resized image
            # print("Image resized successfully.")

            # Preprocess the image
            preprocessed_image = np.expand_dims(resized_image, axis=0)
            preprocessed_image = preprocessed_image / 255.0  # Normalize pixel values to the range [0, 1]

            # Perform the prediction
            predictions = model.predict(preprocessed_image, verbose=0)

            # Get the predicted class label
            predicted_class_index = np.argmax(predictions[0])
            predicted_class_label = class_labels[predicted_class_index]

            # Append the true and predicted labels
            true_label = image_label_map.get(filename, 'Unknown')  # Get the true label from the mapping, or use 'Unknown' if not found
            true_labels_list.append(true_label)
            predicted_labels.append(predicted_class_label)
            predicted_images.append(image_path)

# Convert the lists of true labels and predicted labels to numpy arrays
true_labels = np.array(true_labels_list)
predicted_labels = np.array(predicted_labels)

print("True Lables:", true_labels)
print("Predicted Labels:", predicted_labels)

# Calculate the confusion matrix
cm = confusion_matrix(true_labels, predicted_labels)

# Calculate the accuracy
accuracy = accuracy_score(true_labels, predicted_labels)

# Calculate the precision
precision = np.nan_to_num(cm.diagonal() / cm.sum(axis=0), nan=0.0)

# Calculate the recall
```

```
recall = np.nan_to_num(cm.diagonal() / cm.sum(axis=1), nan=0.0)

# Calculate the F1-score
f1 = np.nan_to_num(2 * (precision * recall) / (precision + recall), nan=0.0)

print("Confusion Matrix:")
print(cm)

# Print the accuracy, precision, recall, and F1-Score
print("Accuracy: {:.2f}".format(accuracy))
print("Precision:", end=" ")
for p in precision:
    print("{:.2f}".format(p), end=" ")
print()
print("Recall:", end=" ")
for r in recall:
    print("{:.2f}".format(r), end=" ")
print()
print("F1-Score:", end=" ")
for f in f1:
    print("{:.2f}".format(f), end=" ")
print()

# Calculate the weighted F1 score
f1 = f1_score(true_labels, predicted_labels, average='weighted')
# Print the F1 score
print("Weighted F1 Score:", round(f1, 2))


# Calculate performance metrics for each class with sklearn method
report = classification_report(true_labels, predicted_labels, zero_division=0)
# Print the classification report
print("Classification report:",report)

## Create heatmap with the seaborn method
# Create a list of class labels
labels = np.unique(true_labels)

# Plot the heatmap
plt.figure(figsize=(8,6))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=labels,␣
 ↪yticklabels=labels)
plt.title("Confusion Matrix")
plt.xlabel("Predicted Labels")
plt.ylabel("True Labels")
plt.show()
```

```python
# Visualize predicted images
for i in range(len(predicted_images)):
    image_path = predicted_images[i]
    image_label = predicted_labels[i]
    image = plt.imread(image_path)

    plt.imshow(image)
    plt.title(f"Predicted Label: {image_label}")
    plt.axis("off")
    plt.show()
```