UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Efficient computational analysis of biomedical images with applications in diagnostics

Diploma Thesis

Athanasia Despoina Sapountzi

Supervisor: Christos Antonopoulos

September 2023

UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Efficient computational analysis of biomedical images with applications in diagnostics

Diploma Thesis

Athanasia Despoina Sapountzi

Supervisor: Christos Antonopoulos

September 2023

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Αποδοτική υπολογιστική ανάλυση βιοϊατρικών εικόνων με εφαρμογές στη διαγνωστική

Διπλωματική Εργασία

Αθανασία Δέσποινα Σαπουντζή

Επιβλέπων/πουσα: Χρήστος Αντωνόπουλος

Σεπτέμβριος 2023

Approved by the Examination Committee:

Supervisor **Christos Antonopoulos**
Professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **Dimitrios Iakovidis**
Professor, Department of Computer Science and Biomedical Informatics, University of Thessaly

Member **Nikolaos Bellas**
Professor, Department of Electrical and Computer Engineering, University of Thessaly

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my supervisor, Professor Christos Antonopoulos, for his unwavering support, guidance, and encouragement throughout the development of this Thesis. I feel fortunate to have had such a dedicated supervisor who was always available to answer my questions. The completion of this Thesis is primarily due to his generosity in providing me with so much of his time and insight.

I am profoundly grateful to Professor Dimitrios Iakovidis for his collaboration as well as allowing me to work on this subject. I want to thank Postdoctoral Researcher Michael Vasilakakis for his constant guidance and support throughout this Thesis. Also, I am grateful to Professor Nikolaos Bellas for being a member of the examination committee for my Thesis.

I want to thank my friends for making these academic years much more enjoyable and encouraging me to pursue my dreams.

Most importantly, I want to express my gratitude to my family, who has always supported me and helped me to accomplish my goals.

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

«Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I also declare that the results of the work have not been used to obtain another degree. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism».

The declarant

Athanasia Despoina Sapountzi

Diploma Thesis

Efficient computational analysis of biomedical images with applications in diagnostics

Athanasia Despoina Sapountzi

Abstract

The field of Artificial Intelligence is rapidly expanding with numerous potential applications in healthcare. As patient data becomes increasingly accessible, healthcare professionals and systems rely more on Artificial Intelligence algorithms to extract valuable insights from biomedical data. However, it is concerning that the applications that use such algorithms often take several hours to days to generate an accurate result. This slow execution time is particularly problematic in the medical field, where timely analysis and decision-making are crucial to patient outcomes. Thus, it is essential to decrease the execution time of such applications while maintaining their performance. In this Thesis, we examine an application that does interpretable data classification. The application detects pathologies in biomedical images while explaining why it produces specific results with the help of fuzzy logic. The goal of this Thesis is to reduce the application's execution time. To achieve this, we follow a systematic, iterative approach to optimize the code of the application while ensuring that we do not negatively affect the functionality and efficiency of the application.

Keywords:

Interpretability, Biomedical Images, Classification, Interpretable fuzzy rules, Fuzzy similarity phrases, Algorithmic optimization, Parallelization, Compiler optimization, GPU

Διπλωματική Εργασία

Αποδοτική υπολογιστική ανάλυση βιοϊατρικών εικόνων με εφαρμογές στη διαγνωστική

Αθανασία Δέσποινα Σαπουντζή

Περίληψη

Ο τομέας της Τεχνητής Νοημοσύνης επεκτείνεται ταχύτατα με πολυάριθμες πιθανές εφαρμογές στην υγειονομική περίθαλψη. Καθώς τα δεδομένα των ασθενών γίνονται όλο και πιο προσβάσιμα, οι επαγγελματίες και τα συστήματα υγειονομικής περίθαλψης βασίζονται όλο και περισσότερο στους αλγορίθμους Τεχνητής Νοημοσύνης για την εξαγωγή πολύτιμων πληροφοριών από τα βιοϊατρικά δεδομένα. Ωστόσο, είναι ανησυχητικό το γεγονός ότι οι εφαρμογές που χρησιμοποιούν τέτοιους αλγορίθμους συχνά χρειάζονται αρκετές ώρες έως ημέρες για να παράγουν ένα ακριβές αποτέλεσμα. Αυτός ο αργός χρόνος εκτέλεσης είναι ιδιαίτερα προβληματικός στον ιατρικό τομέα, όπου η έγκαιρη ανάλυση και η λήψη αποφάσεων είναι ζωτικής σημασίας για τα αποτελέσματα των ασθενών. Συνεπώς, είναι απαραίτητο να μειωθεί ο χρόνος εκτέλεσης τέτοιων εφαρμογών, διατηρώντας παράλληλα την απόδοσή τους. Στην παρούσα Διπλωματική Εργασία, εξετάζουμε μια εφαρμογή που κάνει ερμηνεύσιμη ταξινόμηση δεδομένων. Η εφαρμογή ανιχνεύει παθολογίες σε βιοϊατρικές εικόνες, ενώ εξηγεί γιατί παράγει συγκεκριμένα αποτελέσματα με τη βοήθεια της ασαφούς λογικής. Στόχος είναι η μείωση του χρόνου εκτέλεσης της εφαρμογής. Για να το πετύχουμε αυτό, ακολουθούμε μια συστηματική, επαναληπτική προσέγγιση για τη βελτιστοποίηση του κώδικα της εφαρμογής, φροντίζοντας παράλληλα να μην επηρεάσουμε αρνητικά τη λειτουργικότητα και την αποδοτικότητα της εφαρμογής.

Λέξεις-κλειδιά:

Ερμηνευσιμότητα, Βιοϊατρικές εικόνες, Ταξινόμηση, Ερμηνεύσιμοι ασαφείς κανόνες, Ασαφείς φράσεις ομοιότητας, Αλγοριθμική βελτιστοποίηση, Παραλληλοποίηση, Βελτιστοποίηση μεταγλωττιστή, Μονάδα επεξεργασίας γραφικών

Table of contents

List of figures

List of tables

Abbreviations

AI	Artificial Intelligence
API	Application Programming Interface
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
FSP	Fuzzy Similarity Phrase
GD	Gradient Descent
GI	Gastrointestinal
GIL	Global Interpreter Lock
GPU	Graphical Processing Unit
JIT	Just In Time
ML	Machine Learning
OS	Operating System
RAM	Random Access Memory
RB	Rule Base
VCE	Video Capsule Endoscopy

Chapter 1

Introduction

Medical diagnosis refers to the systematic process of identifying a patient's specific condition and its associated symptoms. Medical diagnosis provides essential information about the disease or condition, which is necessary for the appropriate treatment. This information is collected by a comprehensive assessment of the patient's medical history, physical examination, and surveys. Biomedical images are considered invaluable assets for medical diagnostics [2, 3, 4].

The field of medicine is seeing a progressive transformation due to the integration of artificial intelligence (AI) technology. The primary goal of AI research in the field of biomedical imaging is to develop and refine technologies that improve patient outcomes. Typically, AI tools take the form of imaging decision support systems that offer practical suggestions to imaging professionals [5, 6, 7].

Machine learning (ML) is a more precise designation to characterize the prevailing expression of artificial intelligence. In order to construct machine learning systems, numerous positive and negative examples are fed to an algorithm that modifies itself based on its response to these examples [8]. Machine learning techniques may be categorized into two main classes, based on the training approach: supervised and unsupervised ones. Supervised learning is an approach to learning that relies on labeled data, which provide the correct answer. Unsupervised methods refer to a set of techniques used to automatically cluster comparable data by identifying shared characteristics. In some instances, unsupervised learning is employed as a principal approach to identify important features that can be subsequently used in supervised learning.

A significant barrier to the clinical use and potential regulatory endorsement of AI algo-

rithms is the lack of knowledge regarding the decision-making process employed by these algorithms. Deep learning networks that have undergone training have been commonly referred to as ”black boxes” in the academic literature. Deep-learning models exhibit vulnerability to bias and are sensitive to adversarial attacks. These obstacles are not exclusive to the field of medical imaging; similar issues have emerged in domains such as finance, military, and autonomous vehicles. Hence, the objective of interpretability is to explain certain characteristics of an ML model in a way that can be understood by a human [9]. Consequently, there are increasing efforts to develop methodologies for explainable artificial intelligence [10].

1.1 Motivation

An important challenge with models that employ interpretable ML is that there is an increase in overall execution time compared to ”black box” models [11]. There are several reasons why this is a problem, especially with medical field applications. First, quick / real-time detection of possible anomalies enables early interventions and could potentially prevent serious complications. In other words, whenever a timely and accurate diagnosis is made, the patient has the greatest chance for a positive health outcome [12, 13].Moreover, lower image analysis turnaround times improve the capacity of doctors and medical / computational equipment.

One additional factor to consider is the impact on research and development. The use of faster image analysis tools can expedite research in the field of medical imaging. This enables researchers to process large datasets more efficiently, leading to the identification of novel patterns and insights [14]. In particular, medical practitioners frequently run the same program several times until convergence in the context of simulation-based iterative approaches [15], so any delay in the execution of the application will be amplified.

Therefore, it is necessary to optimize [16] the execution time of applications with interpretable models, while maintaining the accuracy scores.

1.2 Contribution

The aim of the Thesis is to optimize the execution time of an application that employs an interpretable fuzzy classification framework based on Fuzzy Similarity Phrases (FSPs) to

detect abnormalities in biomedical images [1]. The optimized implementation is 250 times faster than the initial implementation. We approached this optimization process iteratively, guided by profiling before and after each optimization step. We first targeted the method at the algorithmic level, and then focused on refining the implementation, while also enabling compatibility with Numba, an optimizing python compiler, for additional optimization. We exploited parallelization, by first implementing multiprocessing at the CPU level, and then by using a GPU to further improve the execution time of the classification algorithm and of additional computations which are amenable to GPU acceleration.

1.3 Thesis Structure

The rest of the thesis is structured as follows:

- Chapter 2 provides background information presenting the main tools used in this Thesis.
- Chapter 3 briefly describes the application which is our optimization target.
- In Chapter 4 we discuss the methodology we used to optimize the code which forms the topic of this thesis.
- Chapter 5 demonstrates the application of the methodology and all the techniques that were used for the optimization of the initial implementation.
- In Chapter 6 we illustrate the interpretability of application decisions using examples from the medical domain.
- Finally, Chapter 7 concludes this Thesis.

Chapter 2

Background – Profiling

A profile is a collection of statistics that describe the execution of a program. Profiling, and the understanding of the intrinsic behavior of the program is the first step towards code optimization. Particularly, identifying bottlenecks in the code enables focusing optimization efforts on specific, problematic code snippets, expected to yield high performance gains when optimized.

The code of the target application is implemented in Python. The version that we use is Python 3.9[17]. In the following sections we briefly outline the characteristics of the profiling toolset we used.

2.1 cProfiler

cProfile is a profiling module in Python's standard library. It provides deterministic profiling of Python programs; all function call, function return, and exception events are monitored, and precise durations are recorded for the intervals between these events (during which the user's code executes). It is a C extension with "reasonable overhead" [18], that makes it suitable for profiling long-running programs, such as our target application in this Thesis.

2.2 Vizualization of profiling results: snakeviz

The application produces a complex dynamic call graph with long branches, including both custom functions and functions provided by Python libraries. Therefore, it is impractical to read and analyze the textual cProfile output provided by the standard library module. As

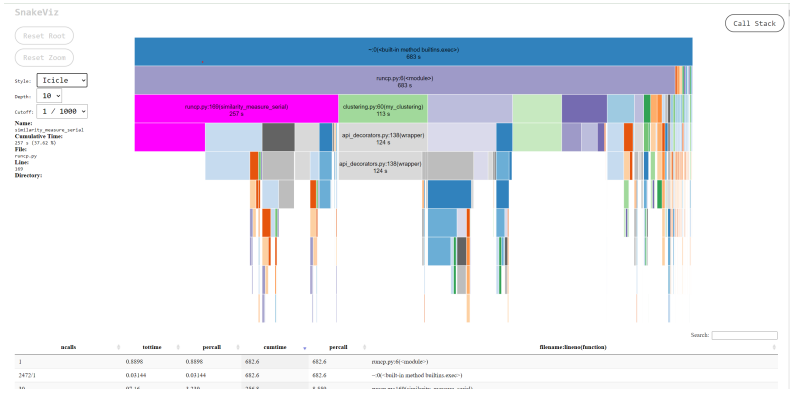


Figure 2.1: Icicle visualization

an alternative, we use Snakeviz[19]. SnakeViz is a browser-based graphical viewer for the output of Python's cProfile module.

SnakeViz supports two visualization styles: icicle (the default) and sunburst. In both, the fraction of time spent in a function is represented by the extent of a visualization element, either the width of a rectangle or the angular extent of an arc. More specifically:

- Icicle visualization: The icicle visualization technique is shown in Figure 2.1. Rectangles represent style functions within the icicle visualization. A root function is the uppermost rectangle, followed, in a recursive pattern, by the functions it calls, then the functions they call, etc. The width of the rectangle represents the total time spent within a function. A rectangle that spans the majority of the visualization represents a function that is consuming the majority of its calling function's time, whereas a slender rectangle represents a low time-consuming function.

- Sunburst visualization: The sunburst visualization technique, which is displayed in Figure 2.2, represents functions as arcs. A function at the root of a call tree (or a subtree) is represented as the center of a layered disk composed, recursively, of a layer of functions that it calls, followed by a layer of functions that those functions call, etc. The angular extent of the arc represents the time spent within a function. An arc that spans a large angle represents a function that consumes the majority of its calling function's time. In contrast, a narrow arc represents a function that barely contributes to the execution time.

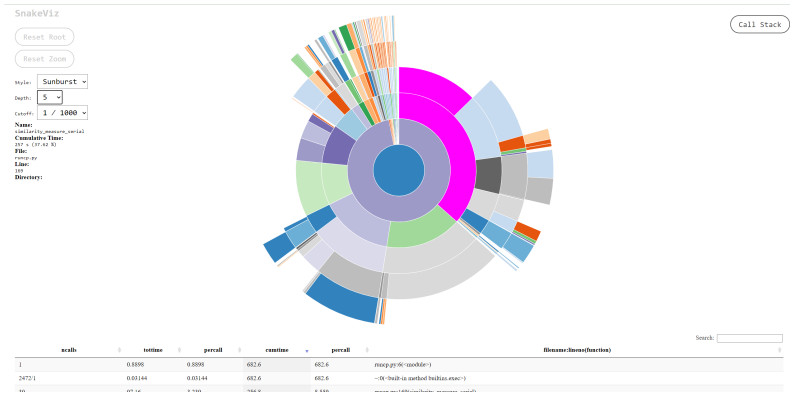


Figure 2.2: Sunburst visualization

Apart from visualization, Snakeviz also exports a table with a row corresponding to each unique function called. The table is illustrated in Figure 2.3. Calls to the same function from different places are all grouped into the corresponding row. Each column, in turn, corresponds to a metric collected during profiling. Those metrics are described in Section 4.1.1. The GUI allows sorting the table in ascending/descending order according to any column of choice.

Search:

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.8898	0.8898	682.6	682.6	runcp.py:6(<module>)
2472/1	0.03144	0.03144	682.6	682.6	~:0(<built-in method builtins.exec>)
30	97.16	3.239	256.8	8.559	runcp.py:169(similarity_measure_serial)
3040	116.4	0.03831	123.6	0.04066	api_decorators.py:138(wrapper)
10	0.0006802	6.802e-05	112.6	11.26	clustering.py:60(my_clustering)
10	0.8251	0.08251	106.8	10.68	gradient_descent.py:116(update_weights_gd)
2842120	86.77	3.053e-05	101.5	3.573e-05	cupy.py:23(rmm_cupy_allocator)
163	20.59	0.1263	85.43	0.5241	gradient_descent.py:15(return_equal_sample)
12066294/11338501	7.618	6.719e-07	76.58	6.754e-06	~:0(<built-in method numpy.core._multiarray_umath.implement_array_function>)
20	61.78	3.089	61.78	3.089	my_fuzzification.py:189(vectorization_similarities)
10302218	5.345	5.189e-07	58.67	5.695e-06	<__array_function__ internals>:177(delete)
20	11.23	0.5615	57.28	2.864	features.py:69(feature_selection)
10302218	39.57	3.841e-06	47.06	4.568e-06	function_base.py:5054(delete)
471640	0.5462	1.158e-06	39.15	8.301e-05	sumprod.py:9(sum)
471640	18.47	3.917e-05	38.27	8.115e-05	~:0(<method 'sum' of 'cupy._core.core._ndarray_base' objects>)
10	2.963	0.2963	34.04	3.404	gradient_descent.py:181(test_classification)
471640	18.17	3.852e-05	29.12	6.173e-05	_norms.py:31(norm)
221688	23.62	0.0001065	23.62	0.0001065	features.py:6(class)
159326	4.2	2.636e-05	20.49	0.0001286	gradient_descent.py:55(error_calc)
1015883	19.87	1.956e-05	19.87	1.956e-05	~:0(<built-in method builtins.sum>)
221688	18.93	8.538e-05	18.93	8.538e-05	features.py:150(<listcomp>)
474680	0.4823	1.016e-06	15.79	3.327e-05	__init__.py:795(asnumpy)
159570	0.1118	7.003e-07	15.45	9.68e-05	<__array_function__ internals>:177(unique)
474680	13.6	2.866e-05	15.24	3.211e-05	~:0(<method 'get' of 'cupy._core.core._ndarray_base' objects>)
159570	0.1856	1.163e-06	15.19	9.516e-05	arraysetops.py:138(unique)

Figure 2.3: cProfile statistic results

Chapter 3

Background – Fuzzy similarity phrases (FSP) for interpretable data classification

3.1 FSP Application Structure

This thesis investigates methods to improve the time efficiency of an application implementing a framework for interpretable data classification. This framework is based on the concept of Fuzzy Similarity Phrases, recently proposed in [1]. This framework [1] aims to enhance the tradeoff between interpretability and classification performance.

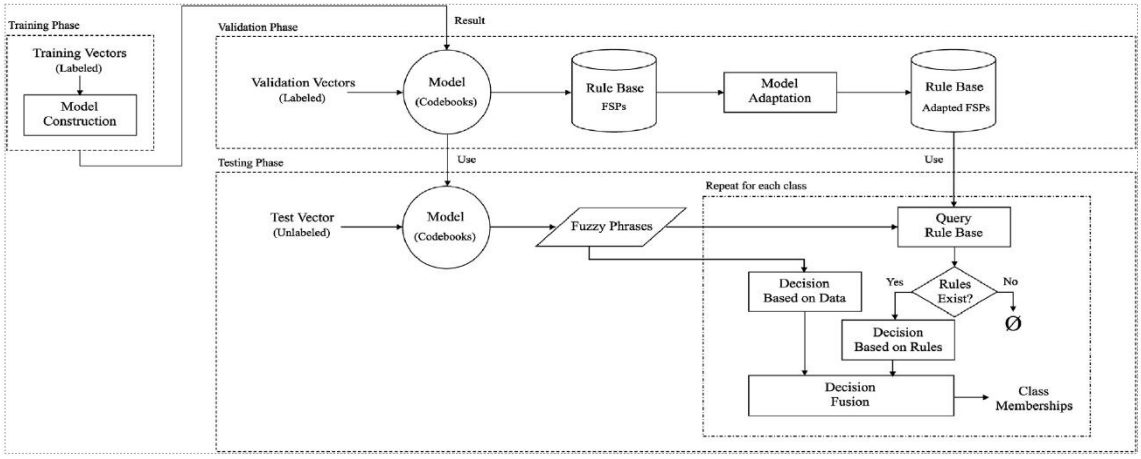


Figure 3.1: Outline of the FSP data classification framework [1]

Figure 3.1 illustrates the framework [1] the application we optimize is based upon. The

application is described in the section that follows.

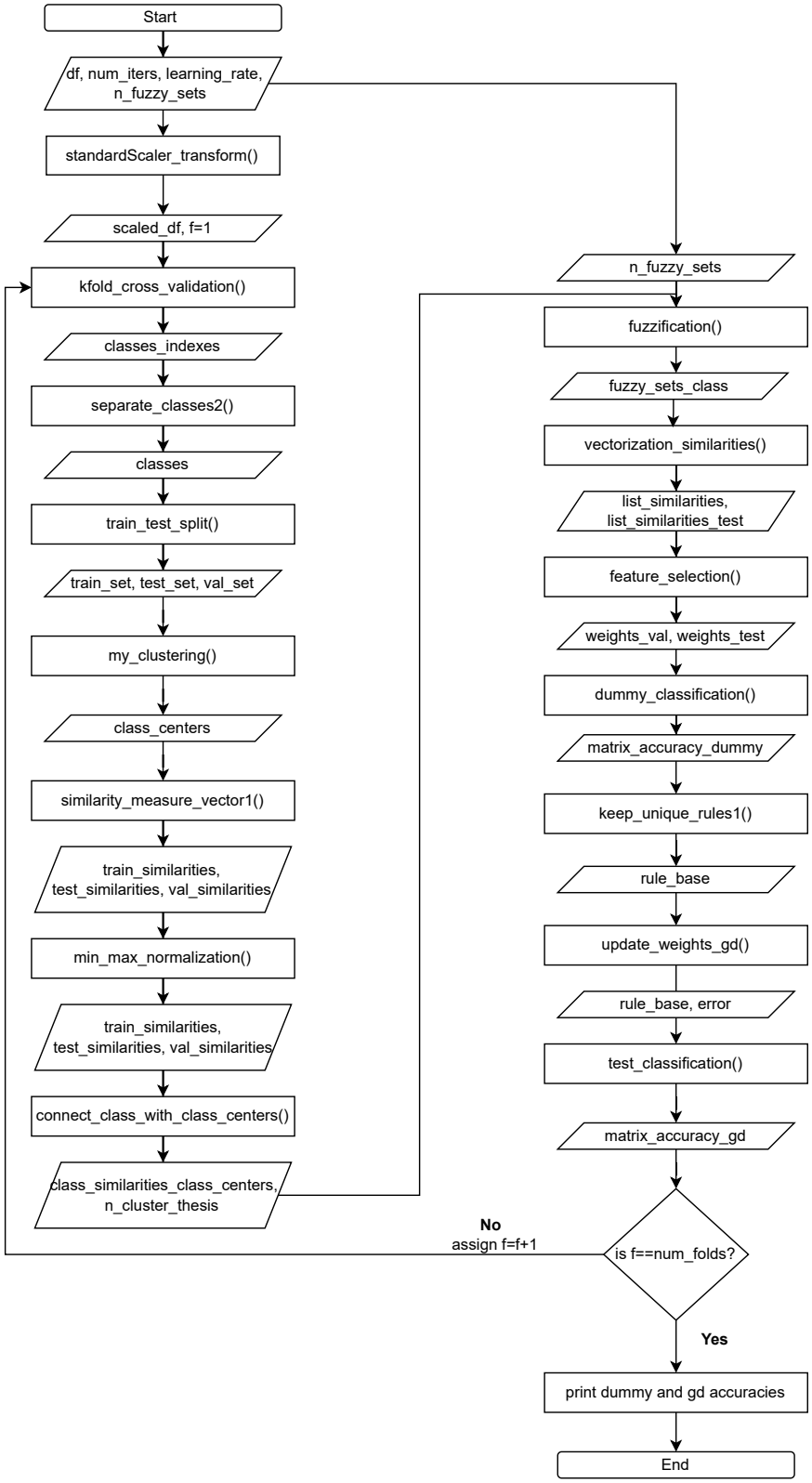


Figure 3.2: FSP flowchart

The flowchart illustrated in Figure 3.2 provides a visual representation of the initial appli-

cation, before any optimizations. It is a graphical representation of the application's essential functions. In addition to illustrating the sequential execution of functions, the flowchart shows the input and output of each function. An extensive analysis of the application depicted in the flowchart is provided in the following subsections.

The application consists of 5 modules:

3.1.1 FSP Model Construction

First, the application processes the data appropriately to create the FSPs. It organizes and partitions the dataset according to the class label of each vector. The algorithm iterates ten times using a smaller segment of the dataset. Each segment contains distinct data (so the mean accuracy is computed by averaging over all iterations). The implementation splits the data into training, validation, and testing sets. The K-means clustering algorithm utilizes the training set to create cluster centroids for vocabulary construction. Next, the application computes the similarities 5.1 between each vector (in the training, validation, and testing set) and centroid in the set created during the previous step. A detailed explanation of how similarities are calculated is provided in Section 5.5.3. Those similarities are normalized via the min-max normalization method [20], and their values are limited in the range [0,1].

Next, the application proceeds to construct the fuzzy sets. Once the similarity matrix is calculated, a clustering algorithm is applied to its elements per column to group the set of similarities into a total of L_c 1D clusters. A fuzzy set is defined for each cluster of elements of the similarity matrix of the training set. Particularly, a triangular membership function is created to represent the fuzzy set A for each cluster l of column m. The computed centroids are sorted in ascending order. The centroid of each cluster can be used as the peak of the triangular function. The base of the function extends from the adjacent smaller centroid to the left to the adjacent larger centroid to the right. This way, the application creates a matrix of fuzzy sets for elements of all classes.

3.1.2 Creation of Phrases and Phrases Reduction

A Fuzzy Similarity Phrase (FSP) is defined as a set of fuzzy sets explaining the belongingness of a vector to a class with respect to a set of words (centroids from K-Means clustering). Next, the application initiates the classification of a vector with unknown class belongingness (e.g. vectors from the validation and testing set) into the "normal" or the "abnormal" class. For each vector similarity to each word, the fuzzy set to which the similarity has the maximum membership is selected to characterize the respective similarity degree. This pro-

cess creates two matrices, one for the vectors of the validation set and one for the vectors of the testing set. Each matrix consists of the similarities, the max membership value for each similarity, and the index that indicates which fuzzy set is assigned to that membership value.

With the aim of simplifying the classification process, the application employs a method for phrase reduction of the FSP, where only one of the fuzzy sets is chosen to represent a particular linguistic value, thereby reducing the number of words used. Following that, a word is inserted in each similarity that describes the class, as well as the weight value.

3.1.3 Rule Generation

The application then proceeds to generate the Rule Base (RB). Pairs of FSPs that describe the belongingness of the same input vector to two different classes are combined to create binary classification rules. The application checks each new rule against all existing ones to identify duplicates. If the new rule is indeed duplicate, it is not added to the rule base.

3.1.4 Model Validation

The set of all weighted rules constitutes an RB, which is then refined and adapted by gradient descent to minimize classification error on the validation set. In particular, the application modifies each rule in RB so that each validation vector is categorized according to its class. Gradient Descent (GD)[21] is used to update the weight vectors of each rule in RB to maximize relative certainty for the correct classification of vectors generating the same FSP rule.

3.1.5 Classification of testing set

Next, the application proceeds to classify vectors from the test set to the appropriate class. Given an unlabeled input vector $\mathbf{x}^u$, which is the feature vector that is retrieved from an input image that belongs to the testing set, an FSP model and an adapted RB, the classification of $\mathbf{x}^u$ to a class during the testing phase is based on the FSP that will be generated using the FSP model. This FSP explains why the input vector belongs to the specific class and with what certainty, based on the weight value. Also, this FSP can be used as a query to find relevant classification rules in RB. The classification of such a vector is based on the result of equation 3.1. If $y^{c,c'}(\mathbf{x}^u) > 0$, then $\mathbf{x}^u > 0$ is assigned to class c.

$$y^{c,c'}(\mathbf{x}^u) = \begin{cases} \frac{y^{R^{c,c'}}(\mathbf{x}^u) + y^{c,c'}(\mathbf{x}^u)}{2}, & \text{if any rules are retrieved from RB} \\ y^{c,c'}(\mathbf{x}^u), & \text{if no rules are retrieved from RB} \end{cases} \tag{3.1}$$

where:

$y^{c,c'}(\mathbf{x}^u)$ is called the basic rule,

$y^{R^{c,c'}}(\mathbf{x}^u)$ is called relative certainty of the retrieved rule(s)

The application calculates the basic rule as:

$$y^{c,c'}(\mathbf{x}^u) = \sum_m s_m^c * \mu_m^c * q_m^c - \sum_m s_m^{c'} * \mu_m^{c'} * q_m^{c'} \tag{3.2}$$

where:

$s_m$is the similarity value between input vector and word m

$\mu_m$is the membership value

q_m is the initial weight value set by the application.

The application calculates the relative certainty of the retrieved rule(s) as:

$$y^{R^{c,c'}}(\mathbf{x}^u) = \sum_m s_m^c * \mu_m^c * q_{gd_m}^c - \sum_m s_m^{c'} * \mu_m^{c'} * q_{gd_m}^{c'} \tag{3.3}$$

where:

$s_m$is the similarity value between input vector and word m

$\mu_m$is the membership value

$q_{gd_m}$is updated weight value calculated by the gradient descent.

The main output of the testing phase is the percentage of vectors from the testing set that were accurately labeled by calculating, namely the accuracy metric [22]. As we discussed earlier, the testing phase is repeated on ten partitions of the test data. The accuracy attained on each iteration is stored in a matrix, Then, the application computes the mean accuracy and the standard deviation (SD) across those 10 values.

3.2 Dataset

For all experiments in the rest of this Thesis, we use input feature vectors derived from a modified version of a Video Capsule Endoscopy (VCE) dataset, called Kvasir-Capsule [23]. A VCE is a compact capsule containing a wide-angle camera, illumination sources, batteries, and additional electronic components. The patient consumes the capsule, which records video

as it passively travels through the digestive tract. A recorder carried by the patient or included in the capsule archives the video prior to its examination by a medical professional following the procedure. The latter contains over 47000 images from inside the gastrointestinal (GI) tract.

The Kvasir-Capsule comprises 117 videos containing 4,741,504 frames and 14 categories of findings. The dataset includes both labeled and unlabeled video files. The VCE recordings were collected using the Olympus Endocapsule 10 System45 from consecutive clinical examinations conducted at the Department of Medicine, Baerum Hospital, Vestre Viken Hospital Trust in Norway. In addition, they exported around 47000 images with labels from these recordings, which we used as input for conducting these experiments.

The assortment of images is originally categorized into three crucial anatomical landmarks and three clinically noteworthy discoveries. Furthermore, the dataset contains two separate categories of images related to endoscopic polyp removal. The dataset is sorted and annotated by medical doctors with expertise in endoscopy.

The altered version comprises the images into two classes: one that contains normal images and one that contains images with pathologies. These images are converted to feature vectors [24] with the aid of VGG-16 [25], a convolutional neural network (CNN) architecture that is capable of extracting feature vectors from images. The feature vectors are, in turn, used as the training, testing and validation input to the FSP application.

Chapter 4

Methodology

In this chapter, we outline the methodology we used for the optimization of the code which is the target of this Thesis. More specifically, we discuss: (i) the profiling-guided optimization loop, (ii) the validation process after each optimization step, (iii) the characteristics of the hardware infrastructure, (iv) the characteristics of the software stack, including the application code, and (v) the inputs used as benchmarks.

4.1 Profile-guided optimization

We employ a systematic, iterative approach, depicted in Figure 4.1 that involves the steps described in this section.

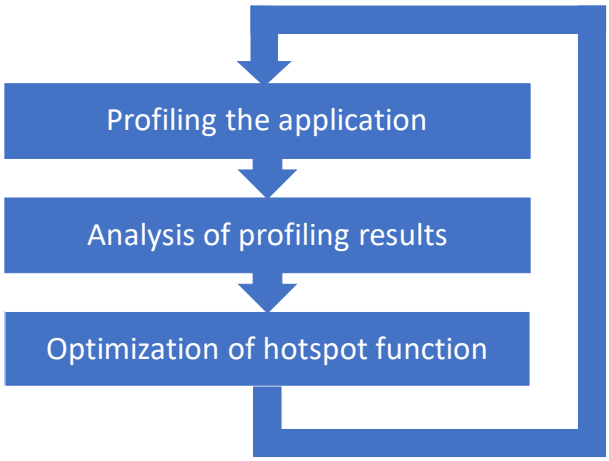


Figure 4.1: Iterative profile-guided optimization pipeline

4.1.1 Application profiling

We execute the code using cProfile to collect statistics that quantify the time spent in each part of the program, the number of calls and to identify the respective call paths. Next, we visualize the collected statistics with SnakeViz[19], as described in detail in section 2.2. SnakeViz generates a report with the time spent in each function and its sub-calls, via a hierarchical view of the dynamic function call graph. The visualization provided by SnakeViz aids in identifying the functions which contribute the most to the overall execution time and in locating performance bottlenecks.

After post-processing the profiler offers the following metrics for each function called:

- ncalls: Number of calls. We ought to try to optimize functions that acquire multiple calls or consume an excessive amount of time per call.
- tottime: Total time consumed within the function, excluding any sub-calls.
- cumtime: Cumulative time, that incorporates sub-calls.
- tottime percall: Total net time of each call.
- cumtime percall: Total cumulative time of each call (including sub-calls).

4.1.2 Analysis of profiling results

According to Amdahl's Law[26], the maximum potential improvement to the performance of a system is limited by the contribution of parts of the system that cannot be improved to the baseline execution time. Thus, we sort the statistics based on the cumulative time and consider the functions with the highest cumulative time as possible hotspot functions. This way, we maximize the fraction of the execution time of the application that the hotspot function represents. Another parameter that we take into consideration is how many times the application calls the specific hotspot function. In the spirit of the discussion earlier, in cases the hotspot function is called a significant amount of times the function call overhead (which can not be optimized) may dominate the function's execution time, outweighing potential optimizations to the function code. Therefore, such functions are not necessarily prime targets for optimization and one should move higher in the hierarchy of the dynamic call graph.

4.1.3 Optimization of hotspot functions

After we have identified a hotspot function as a good potential target for optimization, we perform a code review in the context of the target function to identify concrete optimization opportunities. Such opportunities may present in the direction of making algorithmic changes, using the functionality offered by libraries, refactoring the code, parallelizing parts of the code, or even changing the data layout.

4.2 Hardware and Software infrastructure

Table 4.1 displays the hardware and software characteristics of the system that runs the application. In more detail:

- CPU model: The CPU model is Intel(R) Xeon(R) Gold 6330.
- Number of CPUs: The number of CPUs is 2.
- Number of cores per CPUs: For each CPU there are 28 cores and 2-way hyperthreading support.
- Total number of supported threads: As mentioned above there are 28 cores * 2 threads per core, which equals 56 logical threads per CPU. The system has 2 CPUs, so the total number of threads is 112.
- RAM: The system's RAM capacity is 192 gigabytes. .
- GPU model: The GPU model is NVIDIA A40.
- OS: The operating system is Ubuntu 22.04.2 LTS.
- Programming language: The programming language used is Python 3.9.
- CUDA Version: The version of CUDA is 12.1.

CPU model	Intel(R) Xeon(R) Gold 6330
Number of CPUs	2
Number of cores per CPUs	28, 2-way hyperthreaded
Total number of supported threads	112
RAM	192 GBytes
GPU model	NVIDIA A40
OS	Ubuntu 22.04.2 LTS
Programming language	Python 3.9
CUDA Version	12.1

Table 4.1: Hardware and Software infrastructure

4.3 Baseline application

The baseline implementation of the application we optimize is based on the framework described in detail in Chapter 3. Firstly, it uses a vocabulary-based feature extraction scheme that incorporates similarity features and fuzzy sets to create Fuzzy Similarity Phrases (FSPs). Additionally, it provides a feature selection methodology that works within the FSP classifier to choose the most representative similarity features for dimensionality reduction. Next, pairs of FSPs are combined to create the RB. The application refines the weight vectors of each rule with the aid of GD. Finally, the implementation offers an interpretable classification mechanism, based on the interpretation of the FSPs.

4.4 Validation

After every modification to the original implementation, we validate that the accuracy of the application's predictions remains over 0.75, which is considered as an acceptable accuracy threshold [24] Wherever this is practically feasible, we additionally perform a localized validation step, by directly comparing the function output from the original and the optimized version of the optimized function.

Chapter 5

Performance Optimization

In this chapter, we apply the methodology that was presented in Chapter 4. For each of the profiling-optimization iterations, we discuss the profiling results, the respective optimization technique we employ, the performance improvement attained, and the main take-home points.

5.1 Algorithmic optimization

5.1.1 Profiling results

The profiling results of the original implementation reveal that the execution time of the application is 172000 seconds, which is equivalent to approximately 2 days.

Figure 5.1 illustrates the contribution of the most time-consuming functions to the total execution time of the application. We consider these functions as potential hotspots and further analyze them. For each candidate function, Table 5.1 summarizes the number of times the potential hotspot function is called, along with the execution time per call and the cumulative execution time.

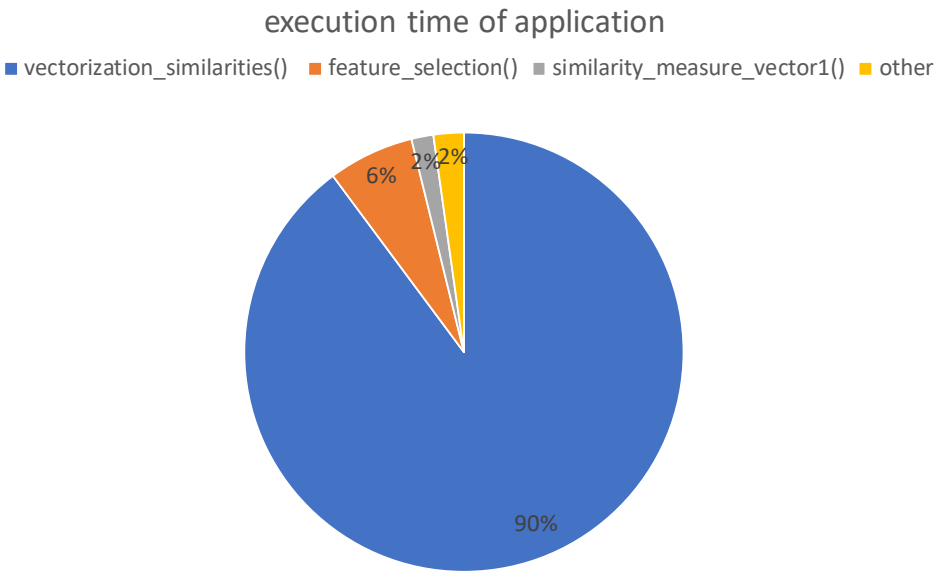


Figure 5.1: Distribution of execution time

	Number of calls	Cumulative time (sec)	Time per call (sec)
`vectorization_similarities`	20	154500	7825
`feature_selection`	20	10886	544
`similarity_measure_vector1`	30	2783	93

Table 5.1: Calling statistics of top hotspot functions

5.1.2 Analysis of profiling results

`vectorization_similarities()` is the function that practically monopolizes the execution time of the application. Further analyzing the profiling results and the `vectorization_similarities()` code, we find that `trimf()` offered by SciKit-Fuzzy [27]is the primary contributor to this delay, being responsible for 90% of the cumulative execution time of `vectorization_similarities()`.

5.1.3 Optimization of hotspot function

`trimf()` computes fuzzy membership values using a triangular function. It is given a set of parameters to define the points of the triangle (fuzzy set), and a vector that defines the sampling points. Then, the original code employs the interp_membership() function from SciKit-Fuzzy using the membership values computed by `trimf()` at the sampling point,

to estimate – via interpolation – the fuzzy membership degree of a value of interest. The value does not necessarily coincide with one of the sampling points in the vector originally provided to `trimf()`. It may, instead, lie between two sampling points.

`trimf()` overcomputes, as it calculates the membership value for a large vector (in the order of 100000 elements) to use, at the end of the day, the values computed for one or two of those elements. Our optimization approach involves skipping the intermediate step of creating a triangular function. We, instead, directly calculate the membership degree of the value of interest. We use equation 5.2 [28] as the function to calculate the membership degree, given a fuzzy set and a value of interest.

$$f(x,a,b,c) = \begin{cases} 0, & x \leq a \text{ or } x \geq c \\ \frac{x-a}{b-a}, & a \leq x < b \\ \frac{c-x}{c-b}, & b \leq x \leq c \end{cases} \quad \text{where } a, b, \text{ and } c \text{ are the values of the given fuzzy set}$$

Figure 5.2: Membership value estimation, based on a triangular membership function

The execution time analysis after the optimization indicates that the application now finishes in 9454 seconds, which is 18.2 times faster than the original implementation.

5.1.4 Validation

Apart from verifying that the mean value of the accuracy is over 0.75, we further evaluate our implementation by comparing the results of membership degrees of the original and the updated version. The results are equal in all significant digits offered by the float type representation of Python.

5.1.5 Discussion

Languages for rapid prototyping and libraries offering rich functionality for different domains significantly enhance programming productivity. However, using off-the-self components without a deep understanding of the functionality offered and the associated computational overhead may come at the cost of paying additional execution-time, should the respective components not be a perfect match for the required functionality.

5.2 Compiler Optimization - `feature_selection()`

5.2.1 Analysis of profiling results

Figure 5.3 illustrates the primary hotspot functions following the previous optimization round, while Table 5.2 provides further information regarding the number of calls for each prospective hotspot function and the corresponding execution time (both total and per call).

As the reader can observe in Figure 5.3, the previous hotspot is now the third slowest function. Therefore, we focus on `feature_selection()` as the new hotspot function, since it has the highest cumulative execution time. `feature_selection()` function takes about 34% of the overall run-time of the application.

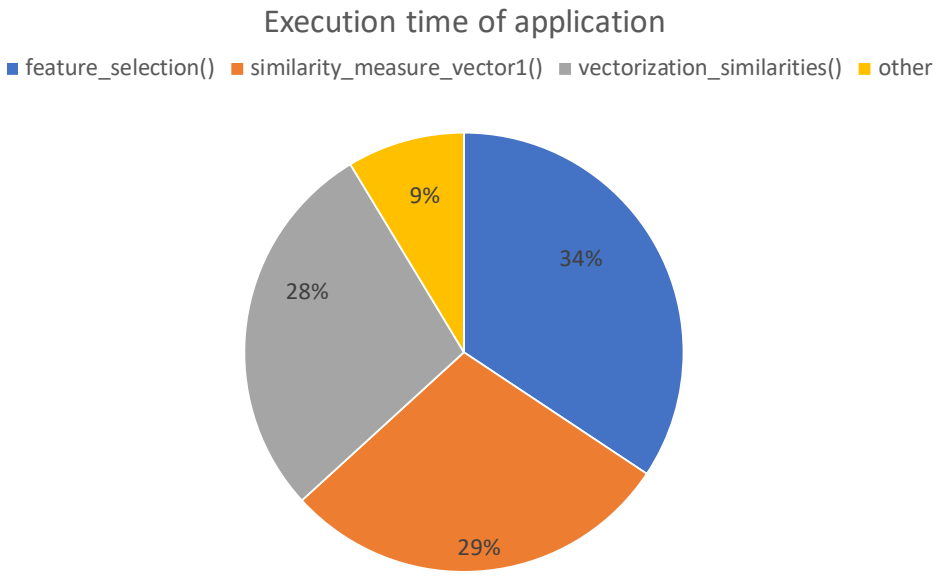


Figure 5.3: Distribution of execution time

	Number of calls	Cumulative time (sec)	Time per call (sec)
`feature_selection`	20	3246	162
`similarity_measure_vector1`	30	2730	90
`vectorization_similarities`	20	2660	132

Table 5.2: Calling statistics of top hotspot functions

5.2.2 Optimization of hotspot function

The goal of `feature_selection()` is to reduce the amount of fuzzy sets of the FSP. Specifically, before the execution of the hotspot function, several fuzzy sets represent the same linguistic value. `feature_selection()` selects the most appropriate fuzzy set for each linguistic value. The selected fuzzy set maximizes the product $similarity * membership$. The calculation of similarity is discussed in detail in Section 5.5.3 and the computation of membership is discussed in Section 5.1.3. We reduce the execution time of the current hotspot function by using Numba[29, 30], a just-in-time (JIT) compiler for Python that specializes in optimizing numerical computations. The main limitation of Numba is that it only supports specific Python libraries and data structures. To enable optimiztions, we transferred the main computational part of the hotspot function in a separate function, called `max_calculations()`.`max_calculations()` returns a list of indexes of the fuzzy sets that did not produce the maximum product and consequently will be deleted. Therefore, the compiler is able to analyze and optimize the – much simpler – `max_calculations()` function and, in turn, to reduce the execution time of `feature_selection()`.

We use the @jit decorator offered by Numba on `max_calculations()` function and specify the additional compilation option nopython=True. The nopython=True argument, used with the @jit decorator, enforces strict type inference during compilation, aiming to compile the function without depending on Python objects or the interpreter. If the compilation process is successful, it generates highly efficient machine code that operates on statically typed variables, like the NumPy arrays, typically resulting in significant performance gain.

The aforementioned code refactoring, the use of Numba, and the use of specific compilation options resulted in a notable decrease in the application's run-time to 6300 seconds. This means that the optimized implementation is 1.5 times faster than our previous optimized version of the application and 27.3 times faster than the original application.

5.2.3 Validation

The newly implemented system has been confirmed to be accurate since the mean accuracy value exceeds 0.75. Another validation step involved comparing the returned indexes of the optimized and non-optimized implementations. The indexes were found to be identical.

5.2.4 Discussion

As it is the case with many optimizing compilers, the code must conform to several requirements / limitations, otherwise Numba is not capable of assessing the safety of several optimizations, and thus it does not apply them. Therefore, refactoring the code (or providing hints to the compiler) is often a necessary step to assist code analysis and enable optimizations.

5.3 Optimization of Data Representation

5.3.1 Profiling results

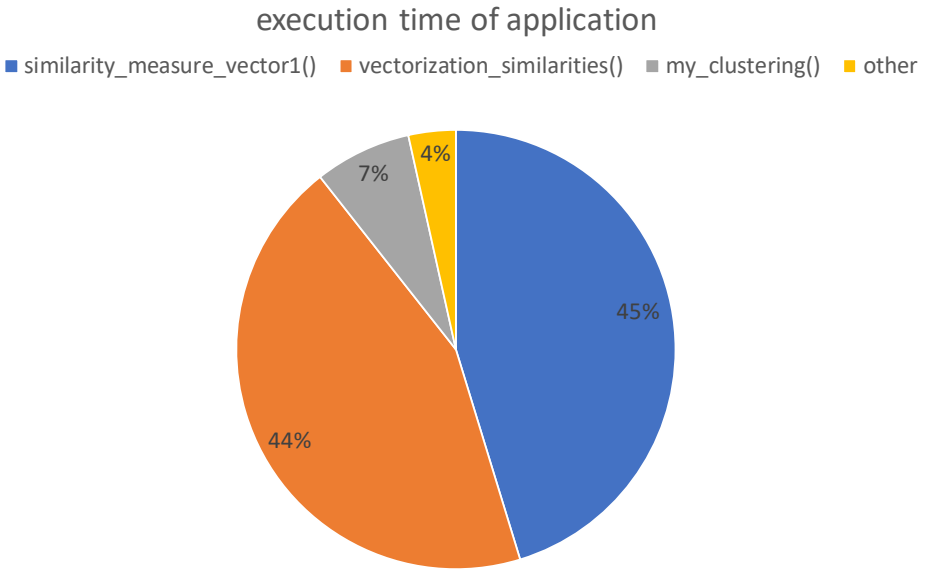


Figure 5.4: Distribution of execution time

	Number of calls	Cumulative time (sec)	Time per call (sec)
`similarity_measure_vector1`	30	2730	90
`vectorization_similarities`	20	2660	132
`my_clustering`	10	430	43

Table 5.3: Calling statistics of top hotspot functions

As discussed earlier, the execution time of the application after the optimization step

described in Section 5.1 is 6300 seconds. Figure 5.4 displays the new top contributors to the execution time of the application, as determined by a new profiling step. As we can observe, the previous hotspot function remains a hotspot with a considerable contribution. Table 5.3 summarizes profiling results in terms of the number of calls of the top hotspot functions, the execution time per call, and the cumulative execution time.

5.3.2 Analysis of profiling results

According to the profiling results in Table 5.3, functions `vectorization_similarities()` and `similarity_measure_vector1()` account for a similar fraction of the application's execution time at around 45%. However, this execution time is amortized to more calls for `similarity_measure_vector1()`. In terms of the cumulative time per call, `vectorization_similarities()` takes 132 seconds compared to the other function `similarity_measure_vector1()`, which takes 90 seconds. Therefore function `vectorization_similarities()` appears as a more promising optimization target.

5.3.3 Optimization of hotspot function

Following a careful code review, we observed that the original implementation creates a variety of different intermediate data structures in order to calculate an array, which serves as the return value of the function. More specifically, it sequentially creates a dictionary, a Pandas Dataframe based on that dictionary, and a list of the numerical values from the data frame. Finally, it produces a final list with the flattened (serialized) elements of those intermediate lists, using list comprehension. Converting data structures from one type to another can introduce significant performance penalties, particularly when dealing with large datasets. It should be noted that conversion between types is pure overhead, as it is related to development and execution logistics and does not directly contribute towards the solution of the problem at hand.

We mitigate this problem by using NumPy arrays as the main data structures, as we observe that only numerical values contribute to the return output of the function. As a result, there is no need to create dictionaries. To implement this solution we replace Python function list.index() provided only for list data structures with a custom function that identifies and returns the max membership value in an array, as well as the corresponding index in the array. Then we directly collect the desired results in Numpy arrays. Finally, we concatenate the transposed version of those arrays. This way, we produce the function output without utilizing any of the previously mentioned diverse data types and structures.

After these modifications, the execution time of the application decreases to 4110 seconds, which is 1.5 faster than our previous optimized implementation and 41.85 times faster than the original implementation.

5.3.4 Validation

We employ a validation process similar to the one we used in the previous optimization step to test the accuracy of the numerical values computed by the function. It should be noted that both implementations yield identical results to the extent of precision supported by the native float type.

5.3.5 Discussion

This optimization highlights that in an effort to achieve optimal performance, it is crucial to carefully consider and choose the most suitable data structures that align with the specific requirements of the code. Also, conversions between different data structures and data types should be minimized, as they may introduce overhead – significant for programs manipulating large datasets.

5.4 Compiler optimization - `vectorization_similarities()`

5.4.1 Profiling results and analysis

Figure 5.5 depicts the top hotspot functions after the previous optimization step, whereas Table 5.4 provides more details on the number of calls of each candidate hotspot and the execution time (total and per call).

The results indicate `similarity_measure_vector1()` as the top hotspot. However, an earlier optimization step, namely turning to NumPy arrays as the main data structure, paves the way to further optimization opportunities. Therefore, we opt to further optimize function `vectorization_similarities()` using Numba.The function `vectorization_similarities()` uses NumPy arrays, which are well supported by Numba, and is adequately numerically intensive.

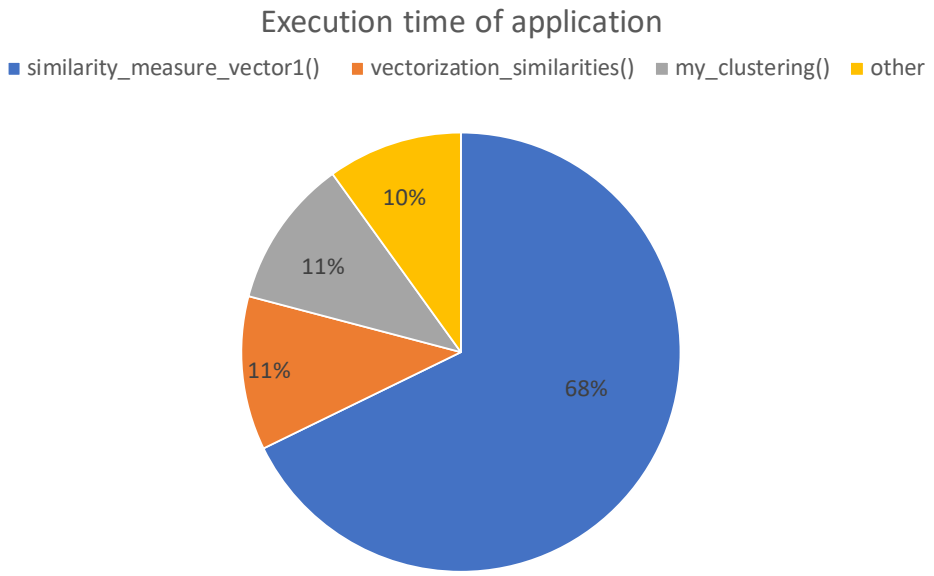


Figure 5.5: Distribution of execution time

	Number of calls	Cumulative Time	time per call
`similarity_measure_vector1`	30	2783s	90s
`vectorization_similarities`	20	465s	23s
`my_clustering`	10	408s	45s

Table 5.4: Calling statistics of top hotspot functions

5.4.2 Optimization of hotspot function

We directly use the @jit decorator on the function `vectorization_similarities()` and we again specify the nopython=True compilation option.

The `jit` decoration with the additional compilation option nopython=True reduced the application time to 3770 seconds, which is 1.1 times faster than the previous implementation and 45.6 times faster than the original implementation.

5.4.3 Validation

The mean value of accuracy of the altered implementation remains higher than 0.75. Thus, the new implementation validates as correct. Moreover, we compare the similarity values of the sequential and parallelized versions of the hotspot function. The outcomes are identical to the extent of the precision of the Python float type.

5.4.4 Discussion

We should point out that Python is an interpreted language, which is not designed for high performance. Using just-in-time compilers – such as Numba – for selected parts of supported code is a straightforward way to achieve measurable performance gains.

5.5 Computation reuse

5.5.1 Profiling results

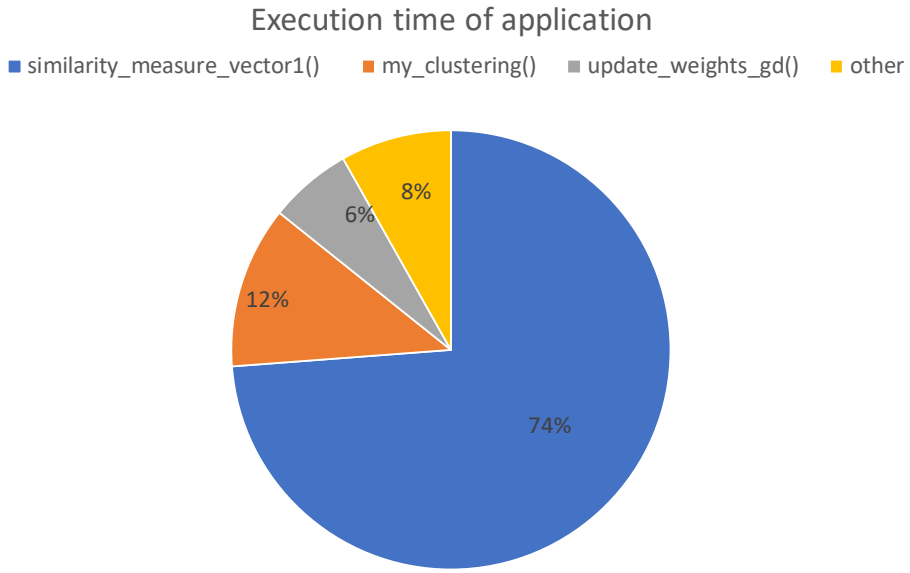


Figure 5.6: Distribution of execution time

	Number of calls	Cumulative Time	time per call
`similarity_measure_vector1`	30	2783s	90s
`my_clustering`	10	450s	45s
`update_weights_gd`	10	230s	23s

Table 5.5: Calling statistics of top hotspot functions

As mentioned earlier, the application's execution time after the previous optimization steps has been reduced to 3770 seconds. After a new profiling round, Figure 5.6 shows the top contributors to the application's execution time. As we can observe, the previous hotspot function `vectorization_similarities()` no longer appears among the top contrib-

utors to execution time. The profiling results are summarized in Table 5.5, which includes details such as the number of calls, execution time per call, and cumulative execution time for each of the top hotspot functions.

5.5.2 Analysis of profiling results

The `similarity_measure_vector1()` function practically monopolizes the execution time of the application. Further analyzing the profiling results and the code of the function, we find that the `linalg.norm()`[31] function offered by NumPy[32] is the primary contributor to this delay, responsible for 68% of the cumulative execution time of `similarity_measure_vector1()`. The number of times `linalg.norm()` is being called is high, at around 284,000,000 times, and each call takes around just 5.6 μsec.

5.5.3 Optimization of hotspot function

According to [1] the formula that computes the similarity of a feature vector $\mathbf{x}_n^c$, to a word $\mathbf{w}_m^{c'}$, $m = 1, \ldots, M$, $c' = 1, \ldots, C$ is:

$$\mathrm{s}_{n,m}^{c,c'} = 1 - \frac{\|\mathbf{x}_n^c - \mathbf{w}_m^{c'}\|}{\sum_{j=1}^{C} \sum_{i=1}^{M^j} \|\mathbf{x}_n^c - \mathbf{w}_i^j\|} \tag{5.1}$$

where $s_{n,m}^{c,c'} \in [0, 1]$ where:

n is the index of feature vector, $n = 1, \ldots, N^c$

$\mathbf{x}_n^c$ is the n[th] feature vector

$\mathbf{w}_i^j$ is the i[th] word from j class

The hotspot function `similarity_measure_vector1()` computes similarities according to Equation 5.1. The initial implementation involved a loop that repeatedly called the norm function to calculate the total distance, then another loop that recalculated and normalized each distance. This caused the `linalg.norm()` function to be invoked redundantly. In the new implementation, we calculate the distance between each feature vector and word in each class, and store the results in a NumPy array. To compute the total distance, we perform a sum reduction across all values in the array.

The optimized application now takes 2402 seconds, which is 1.6 times faster than the previous optimized implementation and 71.6 times faster than the original one.

The optimized `similarity_measure_vector1()` function's cumulative execution time is 1429s, which is 1.9 times faster than its original implementation.

5.5.4 Validation

In addition to checking that the mean value of accuracy remains higher than 0.75, we evaluate our implementation by comparing the similarity values of the results. Using Python's float type representation, the outcomes are identical to all significant digits.

5.5.5 Discussion

This optimization highlights the trade off between memory space and execution time: quite often it is preferable to store intermediate computations in order to reuse them, at the expense of an increased memory footprint of the application.

5.6 Parallelization

5.6.1 Profiling results and analysis

Figure 5.7 reveals that `similarity_measure_vector1()` function still accounts for 60% of the execution time. After drilling down into the profile, we find that 50% of the hotspot function's execution time is still spent in `linalg.norm()`. `linalg.norm()` is called around 142,000,000 times after the previous optimization step, which corresponds to half the number of calls in the original implementation.

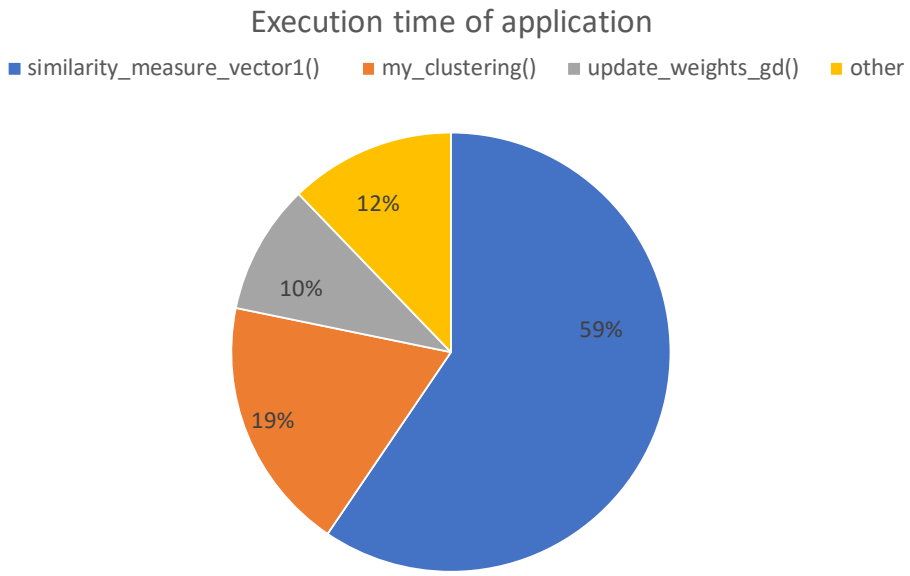


Figure 5.7: Distribution of execution time

	Number of calls	Cumulative Time	time per call
`similarity_measure_vector1`	30	1429s	48s
`my_clustering`	10	450s	45s
`update_weights_gd`	10	230s	23s

Table 5.6: Calling statistics of top hotspot functions

Algorithm 1 Algorithm of hotspot function `similarity_measure_vector1()`

for $c = 1, 2, \ldots, C$ **do**
  for $n = 1, 2, \ldots, N$ **do**
    for $j = 1, 2, \ldots, C$ **do**
      for $i = 1, 2, \ldots, M^j$ **do**
        Calculate norms $\|x_n^c - w_i^j\|$
      end for
    end for
    Calculate similarities between feature vector x_n^c and every word w
    $1 - \frac{\|x_n^c - w\|}{\sum_{j=1}^{C} \sum_{i=1}^{M^j} \|x_n^c - w_i^j\|}$
  end for
end for

5.6.2 Optimization of hotspot function

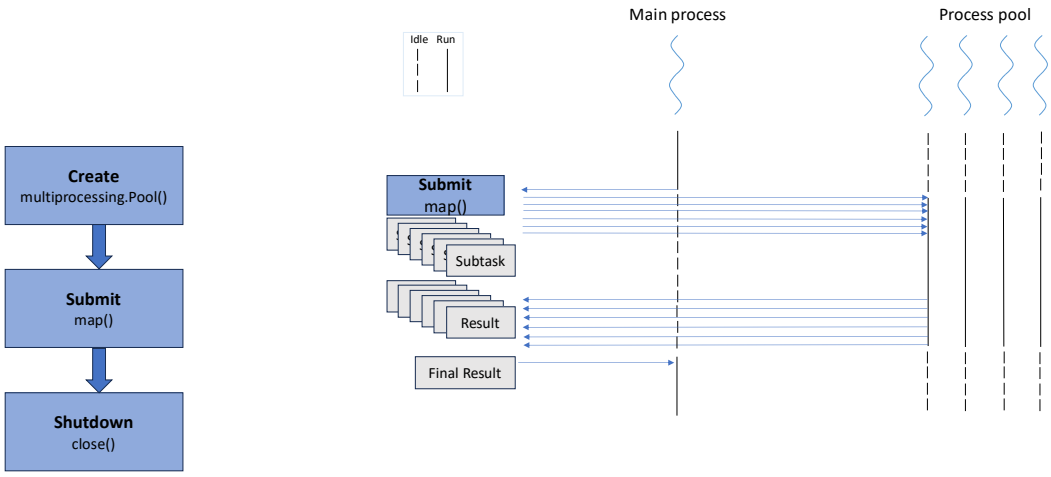
From Equation 5.1, we can observe that there is no dependency between the calculation of similarities for different pairs of words / feature vectors. The aforementioned computation is embarrassingly parallel, therefore the respective loops in `similarity_measure_vector1()` can be parallelized. We opted to parallelize the second level of the loop nest as demonstrated in Algorithm 5.6.1. The reason is that the outermost loop iterates across classes and typically has a low iteration count (2 in the case of binary classification). We tried different alternatives for the parallelization of the loop. The technique that worked best was to create one pool of processes, provided by the multiprocessing.pool library[33]. Figure 5.8 illustrates this technique and Figure 5.8α′ illustrates the life-cycle of the pool. We discuss alternative approaches we experimented with in Section 5.6.5.

We create the pool of processes at the beginning of the application, and we submit tasks to those processes whenever we want to achieve multiprocessing throughout the execution of the application. We terminate the process pool when the application finishes performing its multiprocessing tasks.

Figure 5.8 illustrates this technique in more detail. The key is maintaining these processes throughout the execution and activating/reusing them each time the hotspot function is called. More specifically, there are the main and pool processes, which are idle while executing serial code. The main process becomes inactive as soon as tasks are submitted to the pool, enabling the pool processes to carry out the tasks. When the pool processes have completed all duties, the main process collects their results, and the pool processes revert to an idle state.

We replace the loop that calculates similarities with the parallel version of pool.map(), a Python construct that allows the parallel runtime to assign the work corresponding to different items in the iterable to different – potentially multiple – executors. Each item in the iterable is added to the process pool as a distinct task. Similar to the built-in map[34] function, the returned vector of similarity values is ordered according to the supplied iterable. This means that the runtime supporting work distribution guarantees the ordering of the output to be equivalent to that of the sequential implementation.

One of the parameters when experimenting with parallelism is the number of execution vehicles (processor cores, virtually represented by processes) to use. We performed a performance analysis with different numbers of processes. Figure 5.9 illustrates the experimental results. We find that, for the particular dataset, the optimal number is 6 processes.



(α′) Pool life-cycle (β′) map() visualization

Figure 5.8: Multiprocessing Pool/map illustration

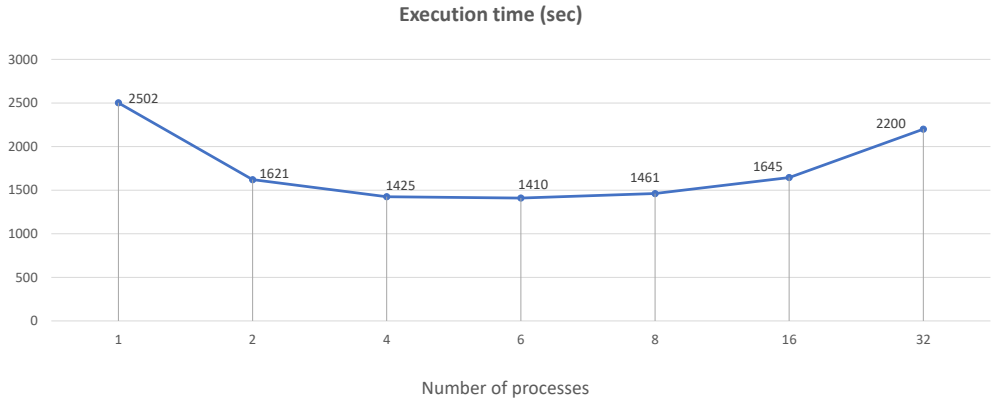


Figure 5.9: Execution time achieved vs. number of processes

The slowdown when using more than six processes may be because optimal multiprocessing performance requires maximizing the workload designated to each worker while minimizing interprocess communication. Notably, the multiprocessing package utilizes queues to facilitate interprocess communication. However, this procedure can be time-consuming when dealing with large arguments, as pickling and interprocess communication require time. This can decrease the positive aspects of multiprocessing. Moreover, increasing the number of processes reduces the workload for each process, and the overhead of interprocess com-

munication becomes disproportionately high compared to the speedup gained from utilizing multiple processes. Thus, we observe from the figure that for a more significant number of processes than 6 the execution time of the application increases.

The optimized implementation has an execution time of 1410 seconds, which is 1.7 times faster than the previous optimization and 122 times faster than the original implementation. The targeted function's cumulative execution time is 418 seconds, which is 6.7 times faster than the original version of the function and 3.4 times faster than the previous optimized version of the hotspot function.

5.6.3 Validation

Apart from validating that the mean value of accuracy is greater than 0.75, we again evaluate our implementation by comparing the similarity values of the sequential and parallelized versions of the hotspot function. The results are identical, to the extent of the accuracy of the float type provided by Python.

5.6.4 Discussion

The exploitation of parallelism is a prerequisite for achieving high performance in modern computing systems, which scale to multiple cores. The exploitation of parallelism may be implicit (through libraries). For instance, KMeans [35] from the `ScikitLearn` [36] library benefits from OpenMP parallelism through Cython[37]. Alternatively, parallelism exploitation may require explicit changes to the code or even the algorithm. In any case, the exploitation of parallelism is typically associated with some overhead which increases with the number of processes and the granularity of the tasks (the finer the granularity, the higher the overhead). Figure 5.9 clearly illustrates this phenomenon, as, with more than 6 processes the execution time increases, despite the target loop offering ample parallelism. The overhead for the exploitation of parallelism depends on the implementation of the runtime system supporting parallel execution, in our case offered by Python.

5.6.5 Unsuccessful optimization efforts

Multithreading

In an effort to reduce the execution time of the application, we tried thread-based parallelism via the Python threading library [38]. A thread is a sequence of instructions executed within the context of a process. Multiple threads can be spawned from a single process but share the same memory. Typically, when using multi-threaded code on a machine with

multiple cores, the available cores are expected to be utilized, resulting in improved overall performance. However, the execution time of the multi-threaded implementation was higher than the one of the sequential version.

After further investigation, we found that the Global Interpreter Lock (GIL) [39] is the reason why this optimization was not successful. More specifically, Python GIL is a mutex that allows a single thread to have control of the Python interpreter at a time. This ensures that only one thread is running at any given time, which prevents races. Therefore, GIL introduces a bottleneck by preventing threads to be executed concurrently within the same process context, making it challenging to exploit CPU parallelism through threading. The execution time of the application with just 2 threads was 2798, which is 1.17 times slower than the previous implementation of the application.

Process pool creation strategy

Due to the GIL bottleneck, we experimented with multiprocessing (using multiple processes rather than threads). In the initial attempt of multiprocessing, we create the pool as a context manager using the `with` clause in place of the for loop that we aim to optimize. This means that the manager automatically creates the pool of processes, executes the code inside the block, and destroys the pool at block exit, by terminating the worker processes. Therefore, for each iteration of the outer loop, a pool was created and destroyed, resulting in unnecessary overhead. The execution time of the application was 2002 seconds, which is faster but not as fast as the implementation discussed in Section 5.6.2.

Using the pool as a context manager proved impractical. Instead, we create a pool of workers only once, in main, execute the code of the application with the same pool of workers, and manually close the pool. When the code is not in a multiprocessing phase, the pool remains idle.

Creation of individual processes

We also experimented with creating and using individual processes, instead of pool of workers. To assign work to these processes, we developed a custom implementation that works similarly to pool/map. Although the execution time was higher the sequential version of the hotspot function, it led to the idea of manually creating/destroying the pool of workers outside the loop and consequently outside the hotspot function. The application's run-time was 6966 seconds, which is 2.9 times slower than the previous implementation.

Map chunksize

To further optimize the application, we experimented with the chunk size. Specifically, we set chunk size=1 and implemented the 2nd for loop inside the `calculate_distances()` function, which is the function that the workers execute. This loop iterates as many times as the value of the previous chunk size, namely is $\frac{loop\_iteration\_count}{number\_of\_pool\_workers}$. The rationale was that the `calculate_distances()` function would be called fewer times and would create less contention among pool workers. The execution time of the application decreased to 1798 seconds. However, with the implementation mentioned in Section 5.6.2 the application runtime dropped to 1410 seconds.

5.7 Classification algorithm GPU acceleration

5.7.1 Profiling results

We have successfully reduced the application's runtime to 1410 seconds via multi-processing, using 6 processes. Figure **??** displays the sections of the application that were the most time-consuming in the next profiling round. Notably, the hotspot function that previously caused the most delay now ranks as the second most time-consuming. Table **??** summarizes the profiling results, including crucial metrics such as the number of calls, execution time per call, and cumulative execution time for each of the top hotspot functions.

5.7.2 Analysis of profiling results

Figure **??** reveals that `my_clustering()` function now accounts for 30% of the application time. After analyzing the profiling results, we identify the execution of the clustering algorithm as the main time-consuming computation of the hotspot function. The clustering algorithm is K-means from the Scikit-Learn [35] library, an unsupervised learning algorithm that partitions a data set into K distinct, non-overlapping clusters, and is based on Lloyd's implementation [40].

The k-means algorithm takes, as input, a set of data points and the number of clusters, k. K-means selects a group of cluster centers, referred to as centroids, minimizing a criterion known as the inertia or within-cluster sum-of-squares. The inertia quantifies the sum of the squared distances of the points to their closest centroid in each cluster, as described by formula 5.2.

$$\sum_{i=1}^{n} \min_{\mu_j \in C}(||x_i - \mu_j||^2) \tag{5.2}$$

where:

n is the total number of data points in the dataset,

C is the set of clusters,

μ_j is the centroid (mean) of cluster j,

x_i is the i-th data point in the dataset, and

$\|\cdot\|^2$ represents the squared Euclidean distance.

5.7.3 Optimization of hotspot function

As previously stated in Section 5.6.4, KMeans from Scikit-Learn[41] library leverages OpenMP-based parallelism when possible [42]. In an effort to further accelerate Kmeans, we aim to execute it using a GPU, whenever such an accelerator is available. Generally, GPUs are well-suited for parallel processing duties and can greatly accelerate specific patterns of computation in comparison to CPUs. Utilizing the enormous power of GPUs can substantially enhance performance in numerical computations. The effectiveness of using a GPU to execute k-means is dependent on the size of the dataset and the efficiency of the GPU implementation. Rapids [43] is a collection of GPU-accelerated open-source Python libraries. These libraries aim to improve data science and analytics pipelines. The libraries offer a familiar Python interface but leverage NVIDIA CUDA primitives to exploit GPUs as accelerators [44].

Rapids includes a machine learning library called cuML. cuML supports machine learning tasks without the need to write code that is specifically developed and optimized for a GPU. In other words, it offers a GPU implementation of Scikit-learn's Kmeans functionality[45]. Therefore, we replaced the functions from Scikit-learn with functions from cuML.

After these modifications, the execution time of the application decreases to 943 seconds, which is 1.5 times faster than our previous optimized implementation and 182 times faster than the original implementation. In addition, the hotspot function's execution time is now 122 seconds, so the hotspot function has been accelerated by a factor of 3.

5.7.4 Validation

The mean value of the updated implementation's accuracy is greater than 0.75. Therefore, this optimization does not affect the precision of the application.

5.7.5 Clustering acceleration - `fuzification()`

Analysis of profiling results

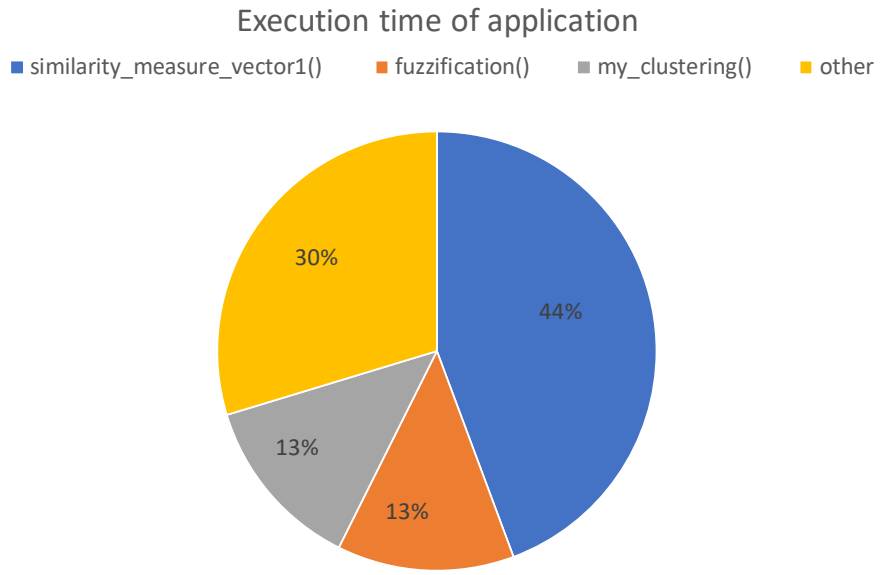


Figure 5.10: Distribution of execution time

	Number of calls	Cumulative Time	time per call
`similarity_measure_vector1`	30	418s	14s
`fuzzification`	10	123s	12s
`my_clustering`	10	122s	12s

Table 5.7: Statistics of possible hotspot functions

The application's execution time has been reduced to 943 seconds following the previous optimization steps. Figure 5.10 displays the main contributors to the application's execution time following a new profiling round. The previous hotspot function now ranks third. Table 5.7 summarizes the profiling results. The functions that are considered as potential hotspot functions are `similarity_measure_vector1()` which has already been optimized using multiprocessing as described in Section 5.6.2, and `fuzzification()` which accounts for 13% of the application's execution time.

Optimization of the hotspot function

Let $\overline{X}^c = \{\mathbf{x}_1^c, \mathbf{x}_2^c, \ldots, \mathbf{x}_{N^c}^c\}$ be a set of N_c training feature vectors $\mathbf{x}_n^c = \left(x_{n,1}^c, x_{n,1}^c, \ldots, x_{n,D}^c\right)$, $n = 1, \ldots, N^c$ belonging to class $c = 1, \ldots, C$. The `fuzzification()` function's purpose is to create fuzzy sets. Initially, the hotspot function executes the K-means clustering

algorithm to group the feature vectors of each set Xc into $M_c < N_c$ clusters.

The clustering algorithm is the main time-consuming task of the hotspot function. The code uses the K-means implementation from Scikit-Learn [35]. We replace the Scikit-Learn K-Means implementation with the CuML K-Means implementation offered by RAPIDS, similarly to the approach described in Section 5.7.3

The application now runs in 864 seconds, which is 1.1 times faster than the optimized version and 199 times faster than the original implementation. Particularly, the cumulative time of the hotspot function is 122 sec, which is 10 times faster than the previous version of the function `fuzzification()`.

Validation

After this optimization step, we validated that mean accuracy remains over 0.75. Therefore, this optimization preserves the quality of application results.

5.7.6 Discussion

The K-means algorithm is sensitive to the initial selection of cluster centroids[46]. Our experiments showed that the cuML K-Means implementation with the initialization flag 'scalable-k-means++' or 'k-means||' (using scalable K-Means++ or parallel K-Means reslectively for centroid initialization), had much lower mean accuracy at $0.68 < 0.75$ compared with the implementation using Scikit-Learn K-Means, despite being faster. On the other hand, when using the 'random' flag to initialize the centroids with the same additional parameters, the application achieved a mean classification accuracy of 0.81, which exceeds the validation criterion of "accuracy >= 0.75". It should be noted that different initialization methods may be more / less appropriate for different datasets. As a consequence, evaluating different centroid initialization methods may be necessary.

5.7.7 Unsuccessful attempts

FAISS

Apart from RAPIDS, we experimented with FAISS [47] library. We developed an implementation that performs K-Means clustering with functions provided by Faiss on the CPU. The execution time of `fuzification()` was higher. The execution time of the application with the FAISS implementation is 1737.7 seconds which is 1.23 times slower than the previous implementation of the application. Particularly, the hotspot function is 1.7 times slower than the initial SciKit-Learn implementation.

Mini-Batch K-Means

Another optimization direction was to replace K-Means from SciKit-Learn with Mini-Batch K-Means [48]. Mini-Batch K-Means, provided by SciKit-Learn, is a variant of the K-Means algorithm that uses mini-batches to reduce computation time. In each training iteration, mini-batches are used, which are randomly sampled subsets of the input data. The execution time of the hotspot function was reduced, however Mini-Batch had inferior performance compared with the cuML implementation. More specifically, using Mini-Batch K-Means reduced the cumulative run-time to 17.4s for each call of the hotspot function, which is 2.47 times faster than the initial K-Means by SciKit-Learn. However, with cuML K-Means, run-time per call was reduced to 12s, which is 3.6 times faster than the initial implementation.

5.8 GPU acceleration

5.8.1 Profiling results

Figure 5.11 highlights the most notable hotspot functions following the previous optimization rounds. Further information on each candidate hotspot, including the number of calls and their respective execution times, is provided in Table 5.8.

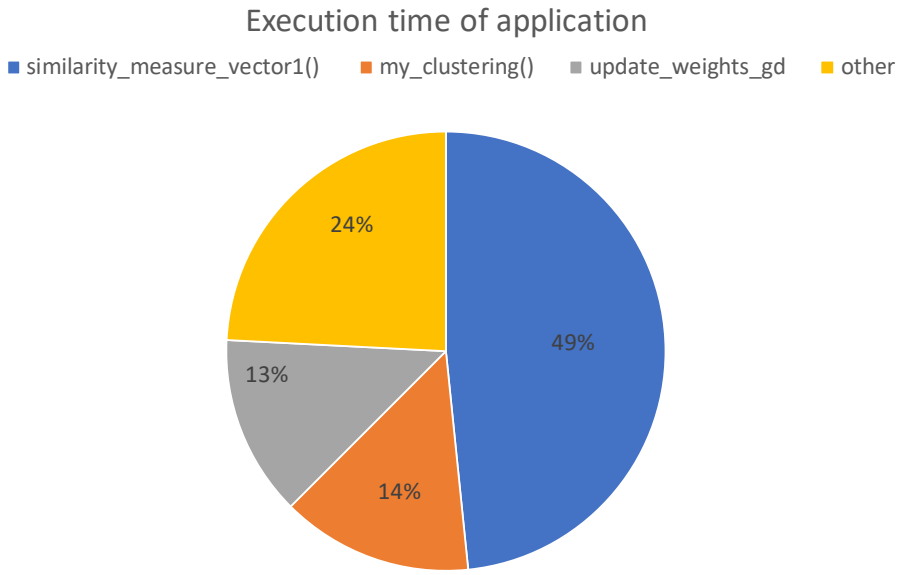


Figure 5.11: Distribution of execution time

	Number of calls	Cumulative Time	time per call
`similarity_measure_vector1`	30	418s	14s
`my_clustering`	10	122s	12s
`update_weights_gd`	10	115s	11.5s

Table 5.8: Statistics of possible hotspot functions

5.8.2 Analysis of profiling results

After optimizing the K-Means algorithm, the application execution time decreases to 864 seconds. `fuzification()`, the previous hotspot function, is no longer among the slowest ones and does not appear in Table 5.8. Although we have optimized `similarity_measure_vector1()` using multiprocessing, as described in Section 5.6.2, it still accounts for 49% of the application's runtime. Therefore, taking into account Amdahl's Law, we focus again on `similarity_measure_vector1()` as a target hotspot function.

5.8.3 Optimization of hotspot function

We previously optimized the hotspot function using Python's multiprocessing capabilities to exploit multiple CPU cores. We will now focus on leveraging the computational power provided by the GPU.

NumPy has been the standard library for array manipulation and numerical operations in Python for many years, but it lacks native GPU acceleration.

cuPy [49, 50] is a GPU-Accelerated NumPy Replacement. cuPy is a high-performance library that provides GPU acceleration while conforming to the NumPy API. It facilitates transparently transferring existing code using NumPy to the GPU, thereby achieving significant performance improvements for computationally intensive tasks. Using the parallel processing capabilities of NVIDIA GPUs, cuPy enables massively parallel array operations and mathematical computations.

CuPy supports all NumPy functions that were utilized in the original implementation, such as linalg.norm. The challenging task is transferring as few and as small arrays as possible to the GPU and back to the CPU [51]. The reason is that GPUs tyically have limited memory space (compared to the amount of RAM available on a modern compute node). Moreover, data transfers between the GPU and CPU are particularly time consuming.

The new implementation demonstrated a significant performance improvement, reducing the execution time to 685 seconds. This is 1.3 times faster than the optimized version produced in the previous round and an impressive 251 times faster than the original implementation. Notably, the cumulative time of the hotspot function is 257 seconds, which is 1.6 times faster than the previous optimized version that uses multiprocessing, and 5.6 times faster than the sequential implementation of the hotspot function. The performance of the different implementations of the hotspot are summarized in Table 5.9 and Figure 5.12.

5.8.4 Validation

In addition to ensuring that the mean value of the accuracy is higher than 0.75, we evaluate our implementation by comparing the sequential and GPU accelerated versions' similarity results. Similarity values are identical in all significant digits.

Implementation	Cumulative Time of hotspot function
`Serial version`	1429s
`Multiprocessing`	418s
`GPU`	257s

Table 5.9: Cumulative Time of hotspot function for different implementations

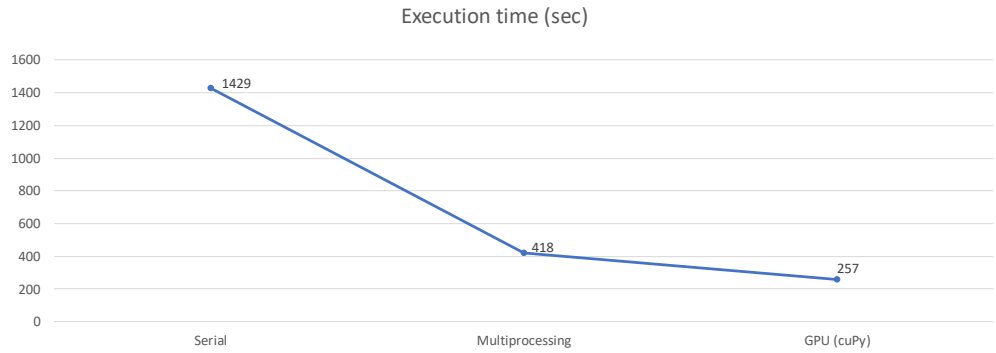


Figure 5.12: Comparison of different implementations of hotspot function

5.8.5 Discussion

The CPU and GPU are distinct components with separate memory. A NumPy-created array is physically located in the host's main memory (RAM) and is accessible to the CPU but not the GPU. Before executing any operations on it, the data must be copied to the GPU memory. Additionally, the computed results need to be transfered back to the host (CPU). This procedure is time-consuming, both due to latency and bandwidth limitations. Optimal exploitation of the CPU-GPU interconnect is achieved with few transfers of large, consecutive data chinks. Therefore, we copy the whole array of each class once, instead of copying the segment used for each iteration.

An important consideration is the limited memory space of our GPU. If transferring the full array is not possible, the implementation may fall-back to copying the array in multiple chunks (with each chunk being as close as possible to the capacity of the GPU memory).

5.9 Post-optimization profiling

We were able to significantly improve the overall execution time of the application by performing the sequence of optimizations discussed earlier in this chapter. Figure 5.13 illustrates the heaviest functions and Table 5.10 provides more details about the cumulative and per-call time of these functions after all optimizations. The function that previously was the main bottleneck is now the third slowest.

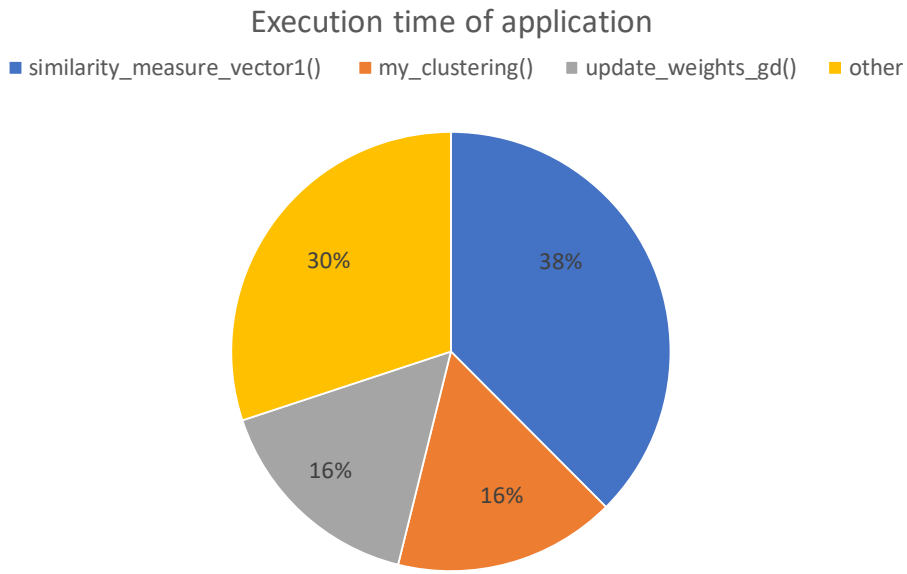


Figure 5.13: Distribution of execution time

	Number of calls	Cumulative Time	time per call
`similarity_measure_vector1`	30	257s	8.6s
`my_clustering`	10	112s	11.2s
`update_weights_gd`	10	110s	11s

Table 5.10: Statistics of possible hotspot functions

Chapter 6

Interpretable classification case study

In this chapter, we demonstrate how the optimized application detects possible pathologies in biomedical images and how the application explains the methodology that led to a particular decision outcome. As we previously mentioned in the Thesis, we use the kvasir dataset that comprises two classes: one consists of images with some type of abnormality, and one contains normal images.For each image the application extracts a feature vector with the aid of CNN model VGG-16. The vector of the unlabeled image from the testing set is defined as $\mathbf{x}^u$.

As mentioned in Section 3.1.5, the application uses the value computed using Equation 3.1 to classify the input feature vector $\mathbf{x}^u$ to the appropriate class.

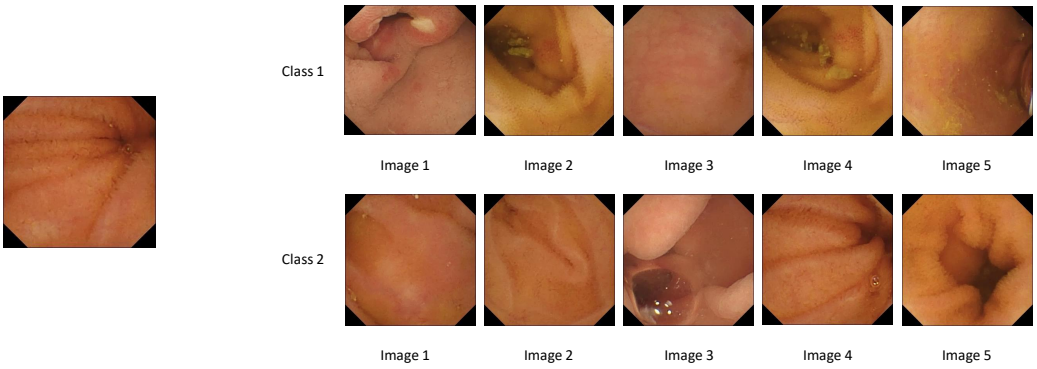
In the following we present 2 successful classification cases. The first one classifies an unlabeled vector to the class that does not contain pathologies and the second case classifies another unlabeled vector to the class with pathologies.

6.1 Classification of image without abnormality

Figure 6.1α′ is the input image from which the sample vector $\mathbf{x}^u$ is produced. Figure 6.1β′ consists of the images whose vectors are closest to the centroids that are selected after the phrase reduction. Specifically the top row consists of images from the first class with pathologies, and the second row consists of images from the second class without any abnormality. For input vector $\mathbf{x}^u$, the application acquired 2 relevant rules from RB. Table 6.1 provides comprehensive details in IF-THEN format for each rule. The application calculates $y^{c,c'}(\mathbf{x}^u)$ for each rule using formula 3.1. Since we have more than 1 rule, the total $y^{c,c'}$ (the result in

the third column of Table 6.1) is calculated as:

$$total\_y^{c,c'}(\mathbf{x}^u) = \frac{\sum\limits_{\forall \text{Rule}} y^{c,c'}(\mathbf{x}^u)}{\text{number of relevant Rules}} \tag{6.1}$$



(α′) Input (β′) Vocabulary

Figure 6.1: Classification of Image without abnormality

RULE	IF	THEN	RESULT
1	If the similarity of the sample $\mathbf{x}^u$ with w_5^1 is Very Low and with w_1^1 is Low and with w_4^1 is Medium and with w_3^1 is High and with w_2^1 is Very High and with w_4^2 is Very Low and with w_2^2 is Low and with w_1^2 is Medium and with w_5^2 is High and with w_4^2 is Very High	Then Class 1	$total\_y^{c,c'}(\mathbf{x}^u) = 0.88$
2	If the similarity of the sample $\mathbf{x}^u$ with w_2^1 is Very Low and with w_1^1 is Low and with w_3^1 is Medium and with w_4^1 is High and with w_5^1 is Very High and with w_3^2 is Very Low and with w_2^2 is Low and with w_1^2 is Medium and with w_4^2 is High and with w_5^2 is Very High	Then Class 2	

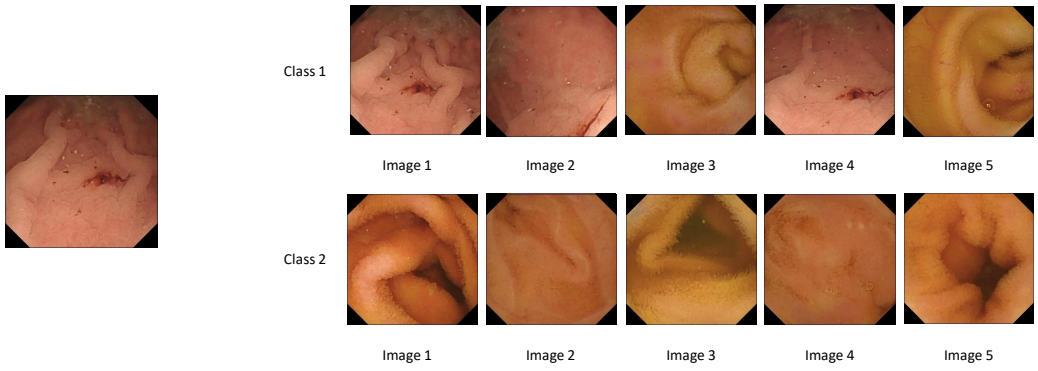
Table 6.1: FSP Classification Rules – normal image

The classification results in Table 6.1 show that the input vector is classified in class 2. Particularly, the result>0 means that the score for class 2 is higher than the score of class 1, which is correct since the image does not illustrate any abnormality.

6.2 Classification of a normal endoscopic image

Figure 6.2α′ is the input image from which the sample vector $\mathbf{x}^u$ is obtained. Figure 6.2β′ consists of the images whose vectors are closest to the centroids that are selected after the phrase reduction. Specifically, the top row consists of images from the first class with abnormalities, and the second row consists of images from the second class without any abnormality. The application obtained input vector $\mathbf{x}^u$ from image 6.2α′ and acquired 2 relevant rules

from the RB. Table 6.2 provides the details of each rule in IF-THEN format. The application calculates $y^{c,c'}(\mathbf{x}^u)$ for each rule using formula 3.1. The result in the third column in Table 6.2 is again computed using equation 6.1.



(α′) Input (β′) Images closest to centroids

Figure 6.2: Classification of Image with abnormality

RULE	IF	THEN	RESULT
1	If the similarity of the sample $\mathbf{x}^u$ with w_5^1 is Very Low and with w_2^1 is Low and with w_4^1 is Medium and with w_1^1 is High and with w_3^1 is Very High and with w_4^2 is Very Low and with w_5^2 is Low and with w_1^2 is Medium and with w_2^2 is High and with w_1^1 is Very High	Then Class 1	$total\_y^{c,c'}(\mathbf{x}^u) = -3.1$
2	If the similarity of the sample $\mathbf{x}^u$ with w_3^1 is Very Low and with w_4^1 is Low and with w_5^1 is Medium and with w_2^1 is High and with w_1^1 is Very High and with w_3^2 is Very Low and with w_1^2 is Low and with w_5^2 is Medium and with w_3^2 is High and with w_2^2 is Very High	Then Class 2	

Table 6.2: FSP Classification Rules – abnormal image

The classification results in Table 6.2 show that the input vector is classified in class 1. Particularly, result<0 means that the score for class 1 is larger than the score of class 2, which is correct since the image displays an abnormality.

Chapter 7

Conclusions

7.1 Conclusions and future work

We successfully optimized the execution time of an application that is responsible for classifying biomedical images in an effort to detect possible abnormalities. More specifically, we improved the application from an algorithmic approach without affecting the accurate results that the application produced in its original form. We have also adjusted parts of the code to be recognizable by the Numba compiler for further optimization. We achieved parallelization with the aid of multiple processes. We accelerated the classification algorithm with the use of GPU. Finally, we achieved optimization by executing operations on the GPU. For each experiment, we profiled the application to detect which part of the code is highly prioritized for optimization. The optimized FSP application has a 251000% speedup.

There are several important lessons we learned from this Thesis. Firstly, many standard Python libraries provide functions that are mostly constructed with a focus on user-friendliness and adaptability, rather than performance. Third-party libraries may provide functions that sacrifice optimality for generality and execute redundant computations. Secondly, Python is an interpreted language, which means that code is executed line by line by the Python interpreter. The interpreter adds additional performance overhead that compiled code would not have. Furthermore, there is Python's Global Interpreter Lock (GIL) that allows only one thread to execute in the interpreter at any time. Thus, multithreading is not a realistic option to improve in Python.

Regarding future work, it is possible to further optimize the parts of the code where Numba JIT is used, by employing CUDA JIT, a low-level entry point to the CUDA fea-

tures in Numba. Another direction is to identify appropriate libraries and/or develop code that can exploit multiple GPUs, a common configuration of modern heterogeneous systems.

Bibliography

[1] Michael D. Vasilakakis and Dimitris K. Iakovidis. Fuzzy similarity phrases for interpretable data classification. *Information Sciences*, 624:881–907, 2023.

[2] Shah Hussain, Iqra Mubeen, Niamat Ullah, Syed Shah, Bakhtawar Khan, Muhammad Zahoor, R. Ullah, Farhat Khan, and Mujeeb Sultan. Modern diagnostic imaging technique applications and risk factors in the medical field: A review. *BioMed Research International*, 2022, 06 2022.

[3] Zeeshan Ahmed, Saman Zeeshan, and Thomas Dandekar. Mining biomedical images towards valuable information retrieval in biomedical and life sciences. *Database*, 2016:baw118, 08 2016.

[4] Julian Varghese. Artificial intelligence in medicine: Chances and challenges for wide clinical adoption. *Visceral Medicine*, 36:1–7, 10 2020.

[5] Curtis Langlotz, Bibb Allen, Bradley Erickson, Jayashree Kalpathy-Cramer, Keith Bigelow, Tessa Cook, Adam Flanders, Matthew Lungren, David Mendelson, Jeffrey Rudie, Ge Wang, and Krishna Kandarpa. A roadmap for foundational research on artificial intelligence in medical imaging: From the 2018 nih/rsna/acr/the academy workshop. *Radiology*, 291:190613, 04 2019.

[6] Sang Kim and Yun Jeong Lim. Artificial intelligence in capsule endoscopy: A practical guide to its past and future challenges. *Diagnostics*, 11:1722, 09 2021.

[7] Gian Tontini, Alessandro Rimondi, Marta Vernero, Helmut Neumann, Maurizio Vecchi, Cristina Bezzio, and Flaminia Cavallaro. Artificial intelligence in gastrointestinal endoscopy for inflammatory bowel disease: a systematic review and new horizons. *Therapeutic Advances in Gastroenterology*, 14:175628482110177, 06 2021.

[8] Kun-Hsing Yu, Andrew L Beam, and Isaac S Kohane. Artificial intelligence in healthcare. *Nat Biomed Eng*, 2(10):719–731, 2018 10 2018.

[9] Ribana Roscher, Bastian Bohn, Marco F. Duarte, and Jochen Garcke. Explainable machine learning for scientific insights and discoveries. *IEEE Access*, 8:42200–42216, 2020.

[10] Andreas Holzinger. From machine learning to explainable ai. pages 55–66, 08 2018.

[11] Owen Shen. Interpretability in ml: A broad overview. *The Gradient*, 2020.

[12] Noha Ossama El-Ganainy, Ilangko Balasingham, Per Steinar Halvorsen, and Leiv Arne Rosseland. A new real time clinical decision support system using machine learning for critical care units. *IEEE Access*, 8:185676–185687, 2020.

[13] Institute of Medicine, National Academies of Sciences, Engineering, and Medicine. *Improving Diagnosis in Health Care*. The National Academies Press, Washington, DC, 2015.

[14] Eyad Elyan, Pattaramon Vuttipittayamongkol, Pamela Johnston, Kyle Martin, Kyle McPherson, Carlos Moreno-García, Chrisina Jayne, and Md. Mostafa Kamal Sarker. Computer vision and machine learning for medical image analysis: recent advances, challenges, and way forward. *Artificial Intelligence Surgery*, 2, 03 2022.

[15] Nick Holford, Holly Kimko, Jonathan Monteleone, and Carl Peck. Simulation of clinical trials. *Annual review of pharmacology and toxicology*, 40:209–34, 02 2000.

[16] Eric Topol. High-performance medicine: the convergence of human and artificial intelligence. *Nature Medicine*, 25, 01 2019.

[17] Python 3.9 documentation. `https://docs.python.org/3.9/contents.html`.

[18] cprofile documentation. `https://docs.python.org/3/library/profile.html`.

[19] Snakeviz documentation. `https://jiffyclub.github.io/snakeviz`.

[20] Devore J.L. and Peck L.E. *Pattern Recognition Principles*. Addison Wesley Publishing Company, 1983.

[21] P. Baldi. Gradient descent learning algorithm overview: a general dynamical systems perspective. *IEEE Transactions on Neural Networks*, 6(1):182–195, 1995.

[22] Marina Sokolova and Guy Lapalme. A systematic analysis of performance measures for classification tasks. *Information Processing and Management*, 45:427–437, 07 2009.

[23] Pia H. Smedsrud, Vajira Thambawita, Steven A. Hicks, Henrik Gjestang, Oda Olsen Nedrejord, Espen Næss, Hanna Borgli, Debesh Jha, Tor Jan Derek Berstad, Sigrun L. Eskeland, Mathias Lux, Håvard Espeland, Andreas Petlund, Duc Tien Dang Nguyen, Enrique Garcia-Ceja, Dag Johansen, Peter T. Schmidt, Ervin Toth, Hugo L. Hammer, Thomas de Lange, Michael A. Riegler, and Pål Halvorsen. Kvasir-capsule, a video capsule endoscopy dataset. volume 8. Nature Publishing Group, Dec 2021.

[24] Michael Vasilakakis. Personal communication.

[25] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv 1409.1556*, 09 2014.

[26] Afips '67 (spring): Proceedings of the april 18-20, 1967, spring joint computer conference. New York, NY, USA, 1967. Association for Computing Machinery.

[27] Scikit fuzzy trimf. `https://pythonhosted.org/scikit-fuzzy/api/api.html`.

[28] Matlab documentation. `https://www.mathworks.com/help/fuzzy/trimf.html`.

[29] Numba documentation. `https://numba.pydata.org/numba-doc/latest/index.html`.

[30] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, New York, NY, USA, 2015. Association for Computing Machinery.

[31] Norm documentation. `https://numpy.org/doc/stable/reference/generated/numpy.linalg.norm.html`.

[32] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[33] multiprocessing documentation. `https://docs.python.org/3/library/multiprocessing.html`.

[34] built-in map documentation. `https://docs.python.org/3/library/functions.html#map`.

[35] Scikit-learn kmeans documentation. `https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html#sklearn.cluster.KMeans`.

[36] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine learning in python. *J. Mach. Learn. Res.*, 12(null):2825–2830, Nov 2011.

[37] Cython documentation. `https://cython.readthedocs.io/en/latest/`.

[38] Multi-threading. `https://docs.python.org/3/library/threading.html`.

[39] Global interpreter lock. `https://wiki.python.org/moin/GlobalInterpreterLock`.

[40] S. Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28(2):129–137, 1982.

[41] Scikit-learn documentation. `https://scikit-learn.org/stable/modules/clustering.html`.

[42] Scikit-learn documentation. `https://scikit-learn.org/stable/modules/clustering.html#low-level-parallelism`.

[43] Rapids documentation. `https://docs.rapids.ai/`.

[44] Sebastian Raschka, Joshua Patterson, and Corey Nolet. Machine learning in python: Main developments and technology trends in data science, machine learning, and artificial intelligence. *Information*, 11(4):193, Apr 2020.

[45] Corey Nolet. Combining speed and scale to accelerate k-means in rapids cuml. *Medium RAPIDS AI*, 2019.

[46] M. Emre Celebi, Hassan A. Kingravi, and Patricio A. Vela. A comparative study of efficient initialization methods for the k-means clustering algorithm. *Expert Systems with Applications*, 40(1):200–210, 2013.

[47] Faiss documentation. `https://faiss.ai/index.html`.

[48] Mini-batch k-means documentation. `https://scikit-learn.org/stable/modules/generated/sklearn.cluster.MiniBatchKMeans.html`.

[49] Ryosuke Okuta, Yuya Unno, Daisuke Nishino, Shohei Hido, and Crissman Loomis. Cupy: A numpy-compatible library for nvidia gpu calculations. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*, 2017.

[50] Matthew Rocklin. High performance python components. In *Proceedings of the Platform for Advanced Scientific Computing (PASC) Conference*, Zurich, Switzerland, June 2019.

[51] Yusuke Fujii, Takuya Azumi, Nobuhiko Nishio, Shinpei Kato, and Masato Edahiro. Data transfer matters for gpu computing. In *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS*, pages 275–282, Seoul, Korea (South), 12 2013.