Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Aksel Danielsen

Clustering and Classification of hyperspectral images on the HYPSO CubeSat

Master's thesis in Engineering Cybernetics
Supervisor: Tor Arne Johansen
Co-supervisor: Joseph Garrett
June 2021

Aksel Danielsen

Clustering and Classification of hyperspectral images on the HYPSO CubeSat

Master's thesis in Engineering Cybernetics
Supervisor: Tor Arne Johansen
Co-supervisor: Joseph Garrett
June 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

NTNU
Norwegian University of
Science and Technology

Abstract

NTNU SmallSat Lab is developing the HYPSO CubeSat, a satellite platform for observing the ocean using a hyperspectral camera payload. Processing captured hyperspectral images on board is to be done using a processing pipeline, developed prior to this thesis. Classification and clustering of hyperspectral images can reduce the need to downlink full hyperspectral images over the low capacity downlink to earth by only having to downlink pixel labels. Given the large amount of data in a hyperspectral image and the modest computational power available, on board processing requires special considerations. This thesis explores the computational limits of the payload hardware, integrates the processing pipeline in the satellite platform and develops modules for clustering, classification and dimensionality reduction subject to the computational constraints. The modules have been verified and analysed with profiling tools, and tested on real hyperspectral data.

A method for clustering is proposed using Self Organizing Maps, by creating a dense representation of hyperspectral data. It achieves partial on board clustering, where the map is trained on the ground and uploaded before being used to cluster new captured images on board HYPSO. This offloads a significant amount of computational work, resulting in a low memory usage and low computational time on the target satellite platform. Various parameters for training and clustering the Self Organizing Map are explored. It shows potential for clustering hyperspectral images with competitive results to other algorithms.

For classification, the popular Support Vector Machine with the Radial Basis Function kernel has been implemented. It is used as a benchmark against a Self Organizing Map based classifier. With good classification accuracy and low memory usage, it is a suitable classification algorithm for on board processing. Principal Component Analysis is used to speed up both classification and clustering through reducing the dimension of the input data.

Sammendrag

NTNU SmallSat Lab utvikler HYPSO, en kubesatellitt som skal overvåke havet ved bruk av et hyperspektralt kamera. Prosesseringen av hyperspektrale bilder ombord blir gjort serielt i en prosesseringskjede som ble utviklet før denne oppgaven. Klassifisering og gruppering av hyperspektrale bilder ombord kan redusere behovet for å laste ned hele hyperspektrale bilder over lavkapasitetslinken til jorden, men heller bare laste ned etiketten til hver piksel. Siden hyperspektrale bilder inneholder store mengder data og det er begrenset med prosesseringskraft tilgjengelig krever prosessering om bord spesielle hensyn. Denne oppgaven utforsker beregningsgrensene for maskinvaren om bord, integrerer prosesseringskjeden i satellitten og utvikler moduler for gruppering, klassifisering og dimensjonsreduksjon underlagt begrensningene i regnekraft. Modulene har blitt verifisert og analysert med profileringsverktøy, og testet på reell hyperspektral data.

En metode for gruppering er foreslått ved bruk av Selvorganiserende Kart, som lager en kompakt representasjon av hyperspektrale data. Det oppnår delvis gruppering ombord, hvor kartet trenes på bakken før det lastes opp og brukes til å gruppere nye bilder tatt av HYPSO. Dette avlaster store deler av prosesseringsarbeidet, som resulterer i et lavt minneforbruk og lav beregningstid ombord. Ulike parametere for å trene og klassifisere de Selvorganiserende Kartene er utforsket. Metoden viser potensiale for gruppering av hyperspektrale bilder med konkurransedyktige resultater sammenlignet med andre algoritmer.

For klassifisering er den populære Støttevektormaskinen med en radial basis funksjon som kjerne implementert. Den blir brukt som referanse til en Selvorganiserende Kart basert klassifikator. Med høy nøyaktighet og lavt minnebruk er det en passende klassifiseringalgoritme for prosessering ombord. Hovedkomponentanalyse brukes for a forbedre prosesseringshastigheten til gruppering og klassifisering ved å redusere dimensjonen til dataen.

Contents

List of Figures

List of Tables

1 Introduction

1.1 HYPSO

The HYPSO (HYPer-spectral Smallsat for ocean Observation) CubeSat is being developed by NTNU Small Satellite Lab. The mission is to launch a satellite for observing oceanographic phenomena along the Norwegian coast. Globally the ocean constitutes an important part of the ecosystem and is central to human life [2]. However with rising climate change driven by human industrial expansion, observing the ocean is of increasing importance. HYPSO is equipped with a hyperspectral imager capable of capturing rich images for analysis, and both NASA and ESA has launched hyperspectral imaging satellites in the past, with the EO-1 [3] and GomX-4B [4]. With a low earth orbit, HYPSO have fast revisit time, enabling rapid observation of the Ocean. To avoid having to downlink the full image to earth using a low capacity radio link, some of the image processing, such as sensor correction, target detection and compression, is done on board.

A specific use case of remote sensing is the observation and detection of algae blooms impacting marine life. In 2020 the Norwegian aquaculture industry exported over 13 billion meals globally [5]. High concentration of algae can be fatal for marine life [6] and in 2019 a single algae bloom killed 10 000 tons of salmon along the Norwegian coast [7].

Launching a satellite enables rapid, autonomous and continuous oceanic observation. Traditional methods with manned aircrafts and ships suffer from limited observation range, ground infrastructure requirements and manual operations. HYPSO and it's planned successor HYPSO-2 are a part of the NTNU AMOS (Autonomous Marine Operations and Systems) project. It aims to launch an interconnected network of autonomous vehicles consisting of autonomous vehicles under, on and above water illustrated in fig. 1. By employing real time processing of payload data, sensor fusion and machine learning it will monitor and observe the ocean, giving insight and data for researchers.

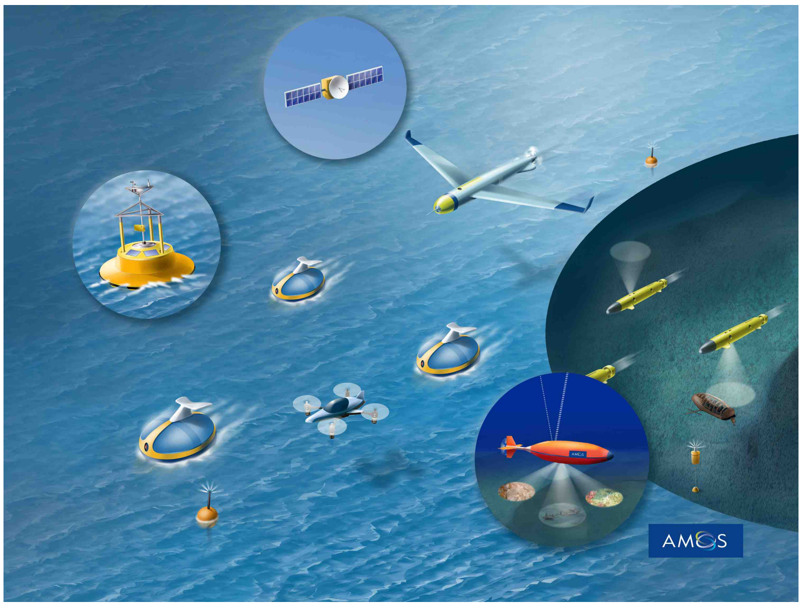


Figure 1: AMOS project for mapping and monitoring of the oceans. From [1]

1.2 Hyperspectral Data

Traditional RGB imagers such as an ordinary smartphone cameras samples wavelengths for red, green and blue light. A hyperspectral imager samples many wavelengths, often over 100 for each pixel. Instead of three discrete RGB values foe each pixel, the result is a continuous spectrum of values illustrated in fig. 2. Such a spectrum contains more information about the scene than traditional images, which makes it a powerful tool for remote sensing.

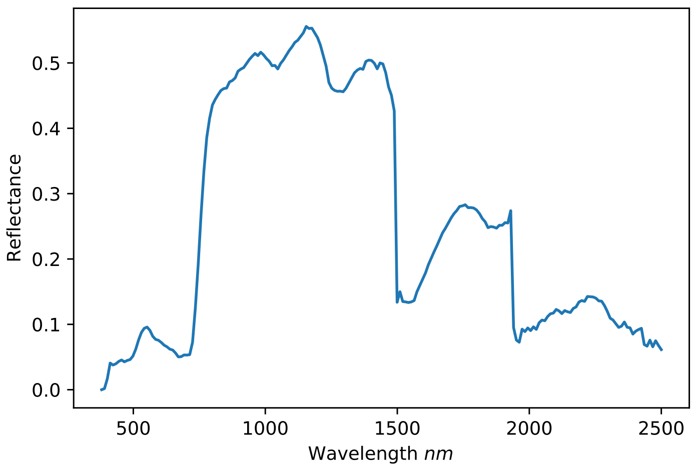


Figure 2: The spectral signature of a pixel taken by a hyperspectral imager

Unlike images taken by an ordinary digital camera which have two spatial dimensions, the HYPSO hyperspectral imager captures lines of data, where each line consists of only one spatial dimension. To capture full images with two spatial dimensions, the imager takes a series of lines, where the imager is moved slightly for each captured line. This is called the pushbroom configuration of hyperspectral imaging. They are then stacked together to form a 2D image. Since the resulting data is a 3-dimensional object with two spatial dimensions (height h and width w) and one spectral dimension (λ), it is commonly refereed to as a hyperspectral cube illustrated in fig. 3.

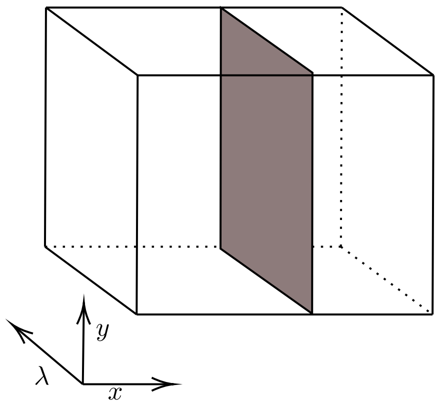


Figure 3: Hyperspectral cube. The shaded area is a single hyperspectral frame captured by the imager. The frames are then stacked along the x axis to form a 2D image

1.3 Classification and Clustering of hyperspectral images

Given the rich information available in hyperspectral cubes, they can be used for analysis of the materials in the image. One such application is classification of pixels, which is concerned with determining the material composition of each pixel, such as finding pixels of algae. A lot of work has been done on the topic of classifying hyperspectral images, such as [8], which uses a neural network to analyse the spatial and spectral information in the pixels for classification. By performing classification on board, the classification results can be downlinked to earth, saving bandwidth compared to downlinking the complete cube. However, performing it on board a satellite imposes computational constraints on the amount of memory and power usage available. A survey of the computational characteristics of on board classification methods of hyperspectral images was done by [9], while [10] used classification on board an aircraft to detect methane gas.

Clustering is a related problem to classification where the problem is to identify pixels of similar material composition, not the source material of each individual pixel. It is equivalent to classification trained in an unsupervised manner. Previous work includes [11] which performs clustering by assuming each pixel is a mixture of Gaussian distributions. An approach based on segmentation followed by KMeans [12] clustering is proposed in [13].

Clustering and classification algorithms can include techniques to reduce the data size before analysis to improve computational time, which referred to as dimensionality reduction. Selecting only a subset of the spectral bands for analysis was shown to improve results in [14]. Using the Karhunen–Loève transform [15] the authors of [16] was able to very efficiently apply a linear transformation to the pixels to reduce the spectral dimension.

1.4 Major contributions

This thesis implements algorithms for performing classification and clustering on board the HYPSO SmallSat. A summery of the key contributions is listed below.

1. A key challenge with on board clustering and classification is the amount of data to be processed: a hyperspectral cube can contain on the order of 10^7 individual sensor values, which is challenging given the computational resources available on a SmallSat platform. This thesis performs an analysis of existing algorithms and demonstrates through experiments and theoretical calculations how most existing procedures are computationally unfeasible on the HYPSO hardware.
2. A general approach using intermediate mappings is then proposed to speed up clustering computations. This approach is realized using the Self Organizing Map (SOM) [17]. It creates a compact representation of the underlying pixel distribution which can be clustered on the ground using any state of the art clustering algorithm, and then transferred to new images captured on board. The method is shown to provide competitive results with low computational time and with minimal memory usage.
3. For classification, this thesis implements the Support Vector Machine (SVM) [18] classifier, a commonly used classification algorithm for Hyperspectral Images. It is compared to a SOM based classifier. They are shown to produce good classification results and generalize well with only a small amount of labeled example data needed. They model parameters are obtained on the ground before uploading them to HYPSO were they can be used for classification.
4. Both the SOM clustering approach and the SVM classifier are further accelerated using the Principal Component Analysis (PCA) [19] dimensionality reduction to reduce the number of spectral bands to process. It is shown to only mildly affect the performance while leading to a major reduction of required computational resources.
5. Finally the clustering, classification and dimensionality reduction procedures are incorporated as modules in an image processing pipeline framework using the C programming language. They are verified using tests and tools for debugging and profiling software. The pipeline framework is then integrated with the existing payload software on HYPSO. The implemented design includes the pipeline as a separate executable program and how this increases reliability and fault tolerance for software on a satellite platform is described.

1.5 Thesis structure

This thesis is structured as follows

- Section 2 covers various background material for HYPSO, software engineering and machine learning.

- Section 3 introduces the Self Organizing Map and clustering algorithms commonly used for hyperspectral images.
- Section 4 presents common classification algorithms and the SOM classification procedure.
- Section 5 analyses the computational resources available on HYPSO. Then the chosen methods for clustering and classification and the pipeline payload integration design are proposed.
- Section 6 describes the software implementation of the modules.
- Section 7 presents and compares the computational performance and algorithm results of the chosen methods on four hyperspectral images.
- Section 8 lists some final remarks and conclusion, as well as future work to improve the methods developed in this thesis.

2 Background

2.1 Hyperspectral Imager

The on board hyperspectral imager on HYPSO is the Sony IMX249 from IDS Imaging Development Systems GmbH [20]. It supports capturing hyperspectral images with spatial width up to 1216 pixels and 1936 spectral bands. The standard cube size on HYPSO is $(h, w, d) = (956, 684, 120)$, where h is the number of captured frames, w is the number of rows in each frame while d is the number of binned spectral bands in the cube. This gives in total $n = h \times w = 6.5 \times 10^5$ pixels in the cube.

2.2 On Board System on Chip

The HYPSO payload is controlled by a on-board System on Chip (SoC) is the Xilinx Zync-7030 [21]. It consists of the Processing System (PS) and a Kinex-7 based Programmable Logic (PL) module. The main technical specifications for the PS are listed in table 1, while the main specifications of the PL are listed in table 2. The PS will handle the main payload software tasks such as image capturing, image processing, communications and file transfers. To accelerate the heavy workloads, some of the algorithms used in image processing will be synthesised and programmed to the PL.

Component	Specification
Processor	32-bit Dual Core ARM Cortex-A9 @ 667 MHz
Architecture Extensions	NEON 128b SIMD coprocessor
L1 Cache	32 KB per processor
L2 Cache	512 KB shared
On-Chip Memory	256 KB

Table 1: Zynx 7030 SoC Proccessing System technical specifications

Component	Specification
Logic Cells	125k
Look-Up Table	78.6k
Flip-Flips	157.2k
Total Block RAM	9.3Mb
DSP Slices	400

Table 2: Zynx 7030 SoC Programmable Logic technical specifications

ARM based SoCs are commonly used for low power embedded systems like mobile phones and microcontrollers due to their high performance to power usage ratio compared to other CPU designs [22]. The on board processing pipeline (section 2.5 frequently involves heavy workloads of numerical calculations like matrix multiplication and vector dot products. To accelerate these type of operations the SoC includes ARM NEON coprocessors [23]. They are capable of executing 128 bit Single Instruction, Multiple Data (SIMD) instructions which performs the same operation on multiple data blocks in parallel. The standard ARM addition instruction is

```
add r1, r2, r3
```

where r1, r2 and r3 is 32 bit registers. This will add the 32 bit values of registers r1 and r2 and store the result in register r3. In comparison the NEON cores features 128 bit registers q0 to q15. The instruction

```
VADD.I32 q1, q2, q3
```

will add the 4 32 bit values in register q1 to the 4 in q2 and store the results in q3. Utilizing the NEON engines will be core to optimize the performance of HYPSO image processing software. In example, the popular media player VLC experienced a doubling in the decoding performance when utilizing the NEON cores for the ARM based Apple A7 chip used in various iPhones and iPads [24]. This workload and SoC is similar to the processing pipeline and HYPSO SoC, so it is expected that similar performance gains can be achieved.

2.3 On Board System on Module

The SoC is located on the PicoZed [25] System on Module (SOM). It features 1 GiB of DDR3 SDRAM, which is the main working memory for the pipeline. A custom designed breakout board is connected to the PicoZed for IO management. In addition it contains temperature sensors and a heat sink.

2.4 On Board Software Infrastructure

HYPSO uses the PetaLinux Tools [26] to deploy an embedded Linux OS to the on-board SoC. It features common operating system components such as the a boot loader, kernel, applications and libraries. The payload software called `opu-system` runs on top of the embedded Linux OS and is responsible for payload tasks such as controlling the hyperspectral and RGB imager, downlinking data and handling image processing. The architecture of `opu-system` is composed of several threads called service handlers, where each service-handler is responsible for a set of tasks. Of interest for this thesis is the service handler `hsi-service` which handles requests related to hyperspectral imaging.

Communication with `opu-system` is handled from the ground using the command line interface tool `hypso-cli`. When issuing a command, it is sent to HYPSO and processed by the appropriate service handler. The communication is based on the CubeSat Space Protocol (CSP) [27], a network layer communications protocol designed for CubeSats.

2.5 On Board Processing Pipeline

The on-board processing framework is a software framework that allows processing hyperspectral cubes in a pipeline. Each module applies some algorithm to the cube in series as illustrated in fig. 4. Some modules like Smile and Keystone Correction[28] and Compression modifies the cube as it propagates through the pipeline. Other modules such as Target Detection and Classification performs some analysis of interest on the cube and saves the output of the module for downlinking.

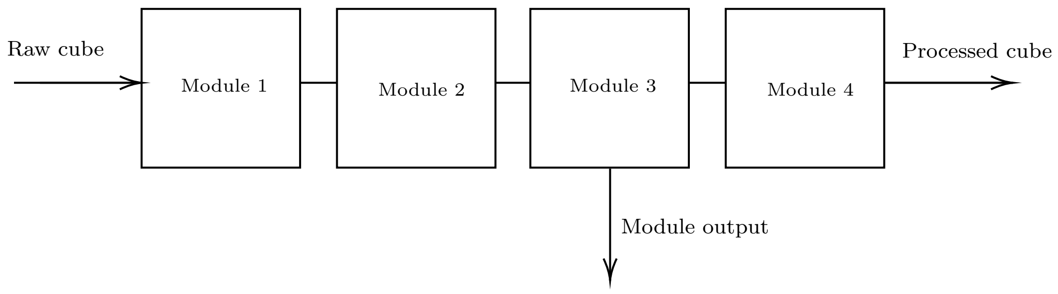


Figure 4: Pipeline modules

The pipeline framework was developed in the fall of 2020, and the current status of the pipeline modules are given in table 3.

The pipeline framework executes in the following way

Module	Implementation Status
Lossless Compression	Implemented (Hardware)
Smile and Keystone Correction	Implemented (Software)
Target Detection	Implemented (Software)
Classification	Implemented in this thesis (Software)
Clustering	Implemented in this thesis (Software)
Dimensionality Reduction	Implemented in this thesis (Software)
Geo-referencing	Not implemented
Super-resolution	Not implemented
Atmospheric Correction	Not implemented
Registration	Not implemented

Table 3: Status of HYPSO pipeline modules

1. Load the pipeline configuration file. The configuration file specifies the sequence of modules to be executed and the configuration file path for each module.
2. Load the hyperspectral cube and set the module index to 0.
3. Execute the module at the current module index. This starts with reading the configuration file for the module. Then any necessary data structures are allocated and initialized. Lastly the module algorithm is executed on the cube, before any allocated data structures are deallocated.
4. Increment the module index and go to step 2 if there are still any modules remaining.
5. Save the processed cube to disk.

Apart from implementing modules, this thesis will also integrate the pipeline in the HYPSO software payload `opu-system` for use during operations as it is currently a stand alone project.

2.6 Software Tools

Several tools has been used to develop the on board pipeline software. The following section provide a short introduction to the most frequently used tools

2.6.1 Docker

Docker [29] is a container based technology that enables consistent environments across platforms. It bundles code, libraries, tools and other dependencies together in a container and uses Linux namespaces to isolate the container process from the host OS. This allows the container to function as a standalone environment that will behave identically independent of the underlying platform. Because the container still utilizes the host OS kernel, it is computationally more lightweight than other isolation technologies such as virtual machines.

2.6.2 GNU Compiler Collection

The GNU Compiler Collection (GCC) [30] is a collection of compiler software for various languages. It features a compiler toolchain for ARM architectures, including a C compiler for the ARM Cortex-A9. Besides compiling source code to executable binaries, it is capable of providing debug information, detecting possibly harmful code and optimizing code performance. This includes generating the aforementioned SIMD NEON instructions from standard C code.

2.6.3 Valgrind

Valgrind [31] is a toolbox for analysing computer programs such as the binaries developed for HYPSO. It features a rich set of functionality such as heap and cache profilers, but the main tool is the memory analyser. By instrumenting the compiled code it is able to detect memory leaks, illegal memory access and similar memory related errors in code.

2.7 Machine Learning

Machine Learning is "to construct computer programs that automatically improve with experience" [32] or "the automatic discovery of regularities in data through the use of computer algorithms" [33]. Given some input data, a machine learning algorithm exploits the patterns in the data to solve a predefined task.

Machine Learning can be divided in two main categories: unsupervised and supervised learning. In supervised learning the algorithm needs examples of desired (input, output) pairs. It then attempts to generalize from these examples to produce output for new, unseen inputs. Unsupervised learning does not need labeled data and produces output without any prior knowledge. Classification belongs to supervised learning, while clustering and dimensionality reduction are examples of unsupervised learning [33].

Let $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$ where $\mathbf{x}_i \in \mathcal{R}^n$ be the input data. Each $\mathbf{x}_i$ is called a sample, data point or data vector.

Classification is a method in supervised learning and is concerned with assigning each sample in the input data to a value in a predefined set of discrete classes, also called labels $L = \{l_1, l_2, \ldots, l_k\}$. An example application is the classification of images of animals. Each image (input data) belongs to one class of animal (dog, cat etc). A classification model for this problem takes the image as input and outputs an estimated label for which class the image belongs to. It is common to let $y_i \in L$ denote the true (unknown) label and $\hat{y}_i \in L$ denote the estimated label of input vector $\mathbf{x}_i$.

Clustering is an unsupervised machine learning task which discovers similar groups of data points. This is done by partitioning the input data where the points in each partition have the same characteristics. The set of all pixels in a partition is referred to as a cluster. If there is c clusters the output of a clustering algorithm is $\mathbf{y} = \{y_1, y_2, \ldots, y_n\}$ where $y_i \in \{1, 2, \ldots, c\}$ denotes the assigned cluster of sample $\mathbf{x}_i$.

Dimensionality Reduction aims to transform the data to a lower dimension. Each data vector is mapped to a vector in a lower dimensional space. Often this is done as a preprocessing step before other algorithms to speed up computations or transform the data to extract desirable features. Given some input data, the algorithm outputs the new data vectors for each point in the new space $\mathbf{X}' = \{\mathbf{x}'_1, \mathbf{x}'_2, \ldots, \mathbf{x}'_n\}$ where $\mathbf{x}'_i \in \mathcal{R}^m$ with $m < d$.

Machine Learning models commonly require a configuration, in practice a set of parameters called hyperparameters and denoted by $\boldsymbol{\lambda}$. An example would be the target dimension m of dimensionality reduction. Hyperparameters is not learned from the data directly, but given as an input to the algorithm along with the input data $\mathbf{X}$.

2.8 Asymptotic Notation

This thesis will use asymptotic notation to denote the computational efficiency of algorithms in terms of memory usage and time usage. For a given input with n elements, if the scaling of the worst case scenario of the algorithm is dominated by a term $f(n)$, the asymptotic complexity is $O(f(n))$. Formally it is defined as a set of functions given by

$$O(f(n)) = \{g(n) : 0 \leq g(n) \leq \alpha f(n)\} \tag{1}$$

for some constants α, n_0 and for all $n > n_0$ [34]. Note that this ignores any constants in the driving terms, for example $3n^2 + 10000n = O(n^2)$ and $n^3 + 2^n = O(2^n)$.

3 Clustering

This section will first give an overview of clustering and it's applications. Then a selection of clustering algorithms is presented and some external metrics for measuring clustering performance. Finally the Principal Component Analysis (PCA) is introduced as a tool for speeding up computations.

Data clustering is a procedure in which we make clusters of entities based on their similar features [35]. It is also worth noting that clustering algorithms are a form of unsupervised machine learning. They require no *a-priori* knowledge, such as existing data with known clusters or domain knowledge from human experts.

A clustering algorithm partitions a set of data points into a set of disjoint sets. Each data point is assigned a label, typically a number, indicating that this data point is a member of that set, called a cluster. The set of these labels is referred to as a clustering of the data. Let $\mathbf{X} = \{\mathbf{x_1}, \mathbf{x_2} \dots \mathbf{x_n}\}$ be a set of n data points where each point $\mathbf{x_i} = \begin{bmatrix} x_{1,i} & x_{2,i} & \dots & x_{d,i} \end{bmatrix}^T$ consists of d features. Then a clustering algorithm is a function f that takes the input data $\mathbf{X}$ and a set of hyperparameters $\boldsymbol{\lambda}$ such that

$$f(\mathbf{X}, \boldsymbol{\lambda}) = \begin{bmatrix} y_1 & y_2 & \dots & y_n \end{bmatrix} \tag{2}$$

where $y_i \in \{1, 2, \dots, c\}$ is the assigned label to data point i and c is the number of clusters to partition the data into.

Clustering algorithms aim to retain the distribution properties of the input data, such that similar points are grouped in the same cluster. The similarity of points within a cluster should be high, while the similarity to points from other clusters should be low. Exactly how similarity is defined varies, and is often a crucial parameter to determine in order to achieve good clustering results.

The objective of clustering the data is to extract semantically meaningful insight and patterns from the data itself. Possible use cases are

- **Detection of outliers or anomalies**. If a clustering is good and a data point has low similarity to the rest of the points in their assigned clusters, the point is "rare" or uncharacteristic with respects to the rest of the data. Some clustering algorithms also manually detect outliers, like the DBSCAN clustering algorithm [36].
- **Understanding and visualizing data.** By clustering the data, the characteristics of each cluster (segment) can be analysed to understand the data source better, emphasizing subtle patterns in the data such as pixel correlation in a hyperspectral image.
- **Contextualizing new data points with old data.** If a new data point is obtained, then a clustering of the complete data set will reveal how the new point relates to the existing data.

With regards to hyperspectral images, the data points to be clustered are typically the pixels in an image. The features are the pixel's value in the various spectral bands. A good clustering algorithm would group pixels that have similar spectral signatures, in other words pixels which are likely the same material. For a remote hyperspectral imager such as HYPSO, a special use case for on board clustering is to reduce the amount of data to be downlinked. Instead of downlinking the whole cube, it would be possible to to cluster the pixels and then estimate the spectral endmember (true spectral signature of the pictured material) of each identified cluster. By only downlinking the clustering labels and their corresponding endmembers, the total amount of data is reduced approximately proportional to the number of spectral bands, which for hyperspectral images can be on the order of 100x.

The notation in table 4 will be used for the presented algorithms, and the algorithms will be demonstrated on the synthetic datasets in fig. 5.

n	Number of pixels
d	Number of spectral bands
c	Number of clusters
$\mathbf{y}$	The clustering labels
y_i	Label for pixel i in $\mathbf{y}$
$\mathbf{X}$	Input image
$\mathbf{x_i}$	Pixel i in $\mathbf{X}$
C_j	The set pixels in cluster j
C	$\{C_1, C_2, \dots, C_c\}$ the partitioned set of pixels
t	Number of algorithm iterations

Table 4: Clustering Notation

There clustering labels $\mathbf{y}$ and pixel partitions C is interchangeable as $C_j = \{\mathbf{x}_i \in \mathbf{X} : y_i = j\}$. If an image consists of three pixels $\begin{bmatrix}\mathbf{x}_1 & \mathbf{x}_2 & \mathbf{x}_3\end{bmatrix}$, then $\mathbf{y} = \begin{bmatrix}1 & 2 & 1\end{bmatrix}$ or $C = \{C_1, C_2\}$ where $C_1 = \{\mathbf{x}_1, \mathbf{x}_3\}, C_2 = \{\mathbf{x}_2\}$ are equivalent partitions.

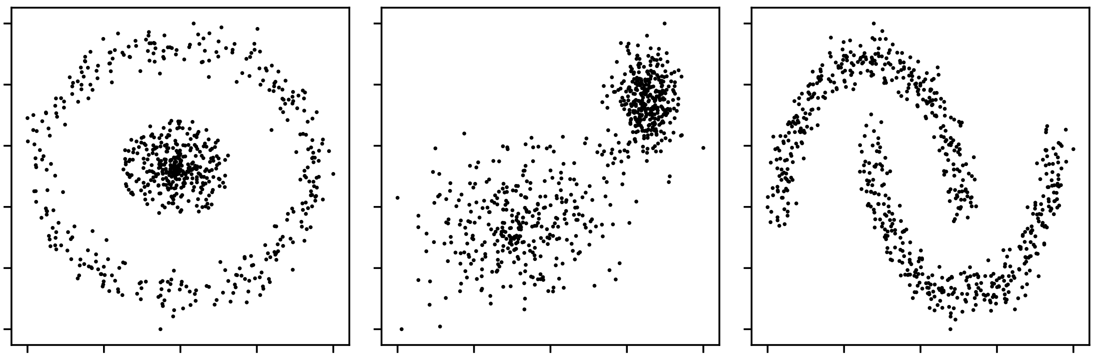


Figure 5: Three 2D dimensional synthetic datasets. They will be used to illustrate the various clustering algorithms presented in this section with $c = 2$ clusters.

3.1 Centroid based Clustering

Centroid based clustering is a clustering technique where each cluster is represented by a center, and each data point $\mathbf{x}_i$ is assigned to a center. KMeans [37] is one of the most popular centroid based clustering methods, known for it's simplicity and computational efficiency. The algorithm aims to find a set of centroids $M = \{\boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \dots, \boldsymbol{\mu}_c\}$ where $\boldsymbol{\mu}_i \in \mathcal{R}^d$ such that each centroid is a prototype or mean member of a cluster. The formal objective is find M and cluster assignments $C = \{C_1, C_2, \dots, C_c\}$ that minimizes the sum of distances from points in the cluster to the cluster centroid

$$\min_{M,C} \sum_{j=1}^{c} \sum_{x_i \in C_j} k(\mathbf{x}_i, \boldsymbol{\mu}_j) \tag{3}$$

where k is the euclidean distance (L_2 norm) as defined in appendix A. To find the global minimum of the objective function in eq. (3) is a known NP-hard problem [38]. The standard heuristic algorithm for minimizing eq. (3) is a generalized version of Lloyds Algorithm [12]. It converges to a local minimum in an iterative fashion as follows:

1. Initialize M

2. **E-Step:** Assign each point $\mathbf{x}_i$ to the cluster C_j that minimizes $k(\mathbf{x_i}, \boldsymbol{\mu}_j)$:
$$C_j = \{\mathbf{x_i} \in \mathbf{X} : j = \operatorname*{argmin}_v k(\mathbf{x_i}, \boldsymbol{\mu}_v)\}$$
3. **M-Step:** Compute the new cluster centroids as the mean of it's assigned points
$$\boldsymbol{\mu}_j = \frac{1}{|C_j|} \sum_{x_i \in C_j} \mathbf{x}_i$$
4. Repeat steps 2-3 until convergence. This occurs when the cluster assignments C_j does not change during the E-step, or equivalently the estimated centroids $\boldsymbol{\mu}_j$ does not change during the M-step.
5. Output C as the final clustering assignments

Since the algorithm only converges to a local minimum of eq. (3), an important choice is how to choose the initial centroids M in step 1. The baseline approach is to choose c random samples from $\mathbf{X}$. There are however more appealing choices, like the kmeans++ algorithm proposed in [39], which is currently the default initialization method in the popular machine learning library scikit-learn[40]. It chooses the the first centroid uniformly from samples in $\mathbf{X}$. The next centroids are then chosen with probability weighted according to the distance to the closest already chosen centroid:

$$\begin{aligned} p(\boldsymbol{\mu}_1 = \mathbf{x}_i) &= \frac{1}{n} && (4) \\ p(\boldsymbol{\mu}_j = \mathbf{x}_i) &= \frac{D(\mathbf{x}_i)^2}{\sum_{\mathbf{x} \in \mathbf{X}} D(\mathbf{x})^2} \quad \text{for } j \in \{2, 3, \ldots, c\} && (5) \end{aligned}$$

where $D(\mathbf{x})$ is the lowest euclidean distance to an already selected centroid. This is shown to speed up convergence and get solutions closer to the global minimum. It is also common to run the KMeans algorithm several times and output the best run as the final clustering output since the found local minimum is heavily dependent on the initialization.

The objective function in eq. (3) is equivalent to minimizing the sum of variances within each cluster

$$\min_C \sum_{j=1}^{c} \sum_{\mathbf{x}_v, \mathbf{x}_u \in C_j} k(\mathbf{x}_v, \mathbf{x}_u) \tag{6}$$

This illustrates a key assumption of Kmeans. Clusters are formed from points that are close together in euclidean space which assumes that the clusters are spherical. Each data point will be assigned to the closest centroid, and by considering the boundary of two clusters, i.e. where the distance to the centroids are equal

$$\begin{aligned} ||\mathbf{x} - \boldsymbol{\mu}_i||_2 &= ||\mathbf{x} - \boldsymbol{\mu}_j||_2 \\ \sum_{k=1}^{d} (x_k - \mu_{ik})^2 &= \sum_{k=1}^{d} (x_k - \mu_{jk})^2 \end{aligned} \tag{7}$$

After subtracting the the quadratic terms of $\mathbf{x}$ on both sides the resulting equation is on the form

$$\sum_{k=1}^{d} \alpha_k x_k = \alpha_0 \tag{8}$$

where α_k are constants. This is the equation for a d dimensional hyperplane. So the separating boundary for KMeans is linear. Figure 6 shows the obtained clusters on the synthetic datasets. As expected the algorithm does not perform well on non linearly separable data.

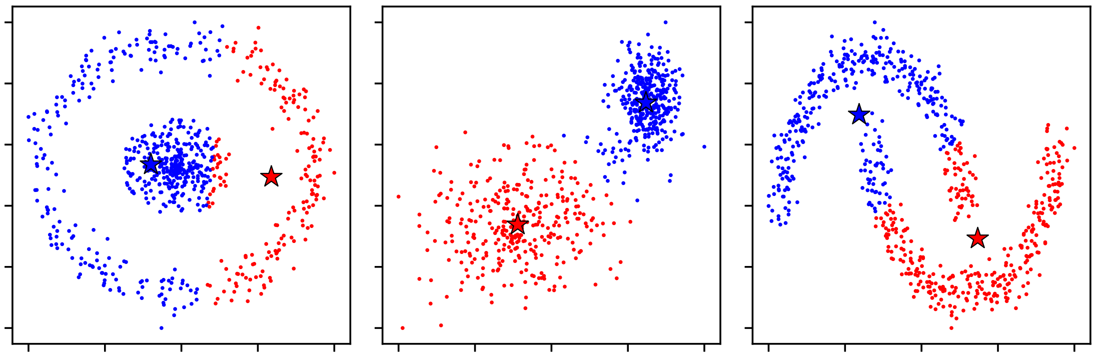


Figure 6: KMeans clustering on synthetic datasets. The star marked indicates the final centroids M

The time complexity of the standard KMeans algorithm is $O(cdnt)$, while the extra memory complexity is $O(cd)$ or $O(cd) + O(n)$ if KMeans++ initialization is used.

3.2 Spectral Clustering

Spectral Clustering (SC) [41] uses eigenvectors to nonlinearly transform the data into a lower dimensional space in which clusters are more prominent. It then performs clustering in the lower dimension using other clustering algorithms like KMeans.

An affinity matrix $\mathbf{A} \in \mathcal{R}^{n \times n}$ for a set of vectors $\mathbf{X} = \{\mathbf{x_1}, \mathbf{x_2} \dots \mathbf{x_n}\}$ is a symmetric matrix with positive elements. Each element A_{ij} denotes the similarity between $\mathbf{x}_i$ and $\mathbf{x}_j$, and is typically in the range $[0, 1]$ where 1 is most similar, and 0 is least similar.

The affinity matrix should be though of as a graph, where each index is a node and each element A_{ij} denotes how strongly linked node i and node j is. The clustering problem can then be solved as a graph partition problem, where the graph encoded by $\mathbf{A}$ should be partitioned into c subgraphs.

A common way to construct a similarity matrix is using a pairwise similarity measure k

$$A_{ij} = k(\mathbf{x_i}, \mathbf{x_j}) \tag{9}$$

such as the radial basis function (RBF) kernel

$$k(\mathbf{x_i}, \mathbf{x_j}) = \exp(-\gamma ||\mathbf{x_i} - \mathbf{x_j}||_2^2) \tag{10}$$

where γ is a positive constant. A degree matrix $\mathbf{D} \in \mathcal{R}^{n \times n}$ for the matrix $\mathbf{X} \in \mathcal{R}^{n \times n}$ is a diagonal matrix defined as

$$\mathbf{D}_{ii} = \sum_{j=1}^{n} \mathbf{X}_{ij} \tag{11}$$

It is used for computing the graph Laplacian matrix $\mathbf{L} \in \mathcal{R}^{n \times n}$ as

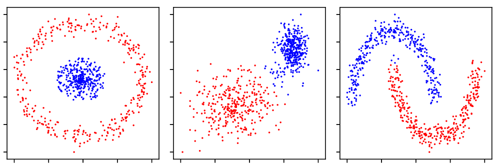
$$\mathbf{L} = \mathbf{D} - \mathbf{A} \tag{12}$$

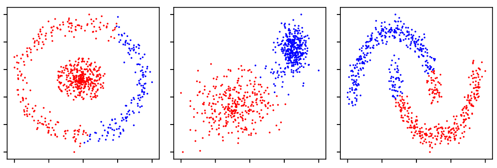
where $\mathbf{A}$ is an affinity matrix of $\mathbf{X}$ and $\mathbf{D}$ is the degree matrix of $\mathbf{A}$.

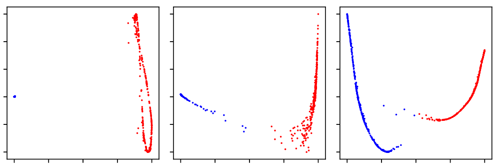
Spectral Clustering computes the Laplacian graph matrix of $\mathbf{X}$ and then exploits spectral properties of the graph to solve the clustering problem. It can be summarized in the following steps

1. Compute the graph Laplacian matrix $\mathbf{L}$ from $\mathbf{X}$ using eq. (12).
2. Find the top c eigenvalues and the corresponding eigenvectors $\{\mathbf{z}_1, \mathbf{z}_2, \dots, \mathbf{z}_c\}$ of $\mathbf{L}$. Construct the matrix $\mathbf{Z} \in \mathcal{R}^{n \times c}$ as $\mathbf{Z} = \begin{bmatrix} \mathbf{z}_1 & \mathbf{z}_2 \dots & \mathbf{z}_c \end{bmatrix}$.
3. Consider the n rows of $\mathbf{Z}$ as data points and cluster them using some other clustering algorithm, typically KMeans.
4. Assign the clustering labels y_i for each pixel i the clustering label of row i of $\mathbf{Z}$

To obtain a good clustering with SC a good choice of affinity matrix is critical [42]. There can be several sources of features for affinity matrix construction. Clustering of hyperspectral images can for example exploit both the spectral signature of each pixel, in addition to spatial information in the image. An algorithm for aggregating different affinity matrices constructed from the same data was proposed in [43]. Figure 7 illustrates the different clustering obtained by using the affinity matrix from eq. (9) with the RBF kernel from eq. (10) with $\gamma = 100$ (fig. 7a) and with $\gamma = 10$ (fig. 7b). The difference in clustering quality can explained from the quality of the spectral embedding, which is the rows of $\mathbf{Z}$, in fig. 7c and fig. 7d.


(a) Clustering for $\gamma = 100$


(b) Clustering for $\gamma = 10$


(c) Embedding for $\gamma = 100$

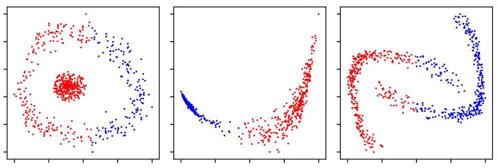

(d) Embedding for $\gamma = 10$

Figure 7: Spectral Clustering and Spectral Embeddings on the synthethic datasets using the RBF kernel

A key challenge with Spectral Clustering for HSI is the computational complexity. The time complexity is $O(dn^2)$ and $O(cn^2)$ for constructing the Laplacian matrix $\mathbf{L}$ and the eigenvalue decomposition of $\mathbf{L}$ respectively. The extra memory complexity is $O(n^2)$ for storing the Laplacian matrix $\mathbf{L}$.

To alleviate the the computational complexity of spectral clustering for hyperspectral images, [44] proposed a variant of Spectral Clustering for HSI called Fast Spectral Clustering With Anchor Graph (FSCAG). This variant improves both time and memory usage by using a set of m anchor points $\mathbf{U} = \{\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u_m}\}$ where $\mathbf{u}_i \in \mathcal{R}^d$. The idea is to construct a matrix $\mathbf{W} \in \mathcal{R}^{n \times m}$ instead of the full affinity matrix $\mathbf{A}$. Each element w_{ij} measures the similarity between point i and anchor

j. They then show that the the affinity matrix $\mathbf{A}$ used for calculating the Laplacian $\mathbf{L}$ can be approximated as $\mathbf{W}\mathbf{\Lambda}^{-1}\mathbf{W}^T$ where $\mathbf{\Lambda}$ is a diagonal matrix with $\Lambda_{ii} = \sum_{j=1}^{n} w_{ji}$.

This work is further improved in [45] which introduces an optimized variant called Fast Spectral Clustering (FSC) by including the work of [46] on Non-negative Matrix Factorization (NMF), which allows $\mathbf{Z}$ to be obtained from the affinity matrix approximation $\mathbf{W}\mathbf{\Lambda}^{-1}\mathbf{W}^T$ through a multiplicative update rule instead of an eigenvalue decomposition.

The time complexity of FSC is $O(dnm)$ for constructing $\mathbf{W}$, $O(nm)$ for approximating $\mathbf{Z}$ by NMF and $O(cmnt)$ for the final KMeans clustering. Since typically $ct < d$, the overall complexity is $O(dnm)$. With respects to memory, the complexity of FSC is $O(nm)$ for storing $\mathbf{Z}$. However, [45] observes that the clustering result is heavily dependent on the number of anchors m, with larger m improving the results. They recommend using $m = 0.1n$, which means that in reality both the time and memory complexity of FSC is quadratic in the number of pixels n, although with one order of magnitude lower coefficients than SC.

3.3 Model Based Clustering

Model based clustering is based on forming a probabilistic model for how the pixels $\mathbf{X}$ are generated. This typically involves considering the pixels as random variables drawn from some distribution that depends on the latent, unobserved variables $\mathbf{y}$. The clustering problem is then transformed into an inference problem of estimating the latent labels $\mathbf{y}$ from the observed data $\mathbf{X}$.

3.3.1 Gaussian Mixture Model

Model based clustering for Hyperspectral Images is typically based on the Gaussian mixture model solved by the Expectation Maximization (EM) algorithm [47]. The Gaussian mixture model is a probabilistic model that assumes that each pixel is drawn from one of c multivariate Gaussian distributions. The intuition behind this model is that each pixel comes from some spectral end-member, with the addition of some noise.

$$\mathbf{x}_i = \boldsymbol{\mu}_{\boldsymbol{j}} + \mathbf{z}_{\mathbf{j}} \tag{13}$$

for some $j \in \{1, 2, \ldots, c\}$ and where $\mathbf{z}_{\mathbf{j}}$ is Gaussian noise, i.e. $p(\mathbf{z}_j) = \mathcal{N}(\mathbf{z}; \mathbf{0}, \mathbf{\Sigma}_j)$.

Every pixel $\mathbf{x}$ can therefore be though of as a independently drawn random variable with probability density function

$$p(\mathbf{x}) = \sum_{j=1}^{c} \pi_j \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_j, \mathbf{\Sigma}_j) \tag{14}$$

where $\pi_i \in [0, 1] = p(y = j)$ is the mixing variable with the constraint $\sum_{j=1}^{c} \pi_j = 1$ and $\boldsymbol{\mu}_j$ and $\mathbf{\Sigma}_j$ is the mean and covariance matrix for the multivariate distribution of cluster j. The conditional distribution of $\mathbf{x}$ then becomes the Gaussian

$$p(\mathbf{x}_i | y_i) = \mathcal{N}(\mathbf{x}_\mathbf{i}; \boldsymbol{\mu}_{y_i}, \mathbf{\Sigma}_{y_i}) \tag{15}$$

The observed data in this model is the pixels $\mathbf{X}$, while the latent variables to be estimated are $\mathbf{y} = \{y_1, y_2, \ldots, y_n\}$, i.e. the clustering labels for each pixel. Let $\boldsymbol{\theta}$ be the model parameters $\{\boldsymbol{\mu}_1, \mathbf{\Sigma}_1, \pi_1 \ldots, \boldsymbol{\mu}_c, \mathbf{\Sigma}_c, \pi_c\}$. To solve the clustering problem, the objective is to find the Maximum A Posteriori (MAP) estimate (appendix A) of $\mathbf{y}$. The posterior distribution is given by

$$\begin{aligned}
p(\mathbf{y}|\mathbf{X}) &= \prod_{i=1}^{n} p(y_i|\mathbf{x}_i) \\
&= \prod_{i=1}^{n} \frac{p(y_i)p(\mathbf{x}_i|y_i)}{p(\mathbf{x}_i)} \\
&= \prod_{i=1}^{n} \frac{\pi_{y_i}\mathcal{N}(\mathbf{x_i};\boldsymbol{\mu}_{y_i},\boldsymbol{\Sigma}_{y_i})}{\sum_{j=1}^{c}\pi_j\mathcal{N}(\mathbf{x_i};\boldsymbol{\mu}_j,\boldsymbol{\Sigma}_j)} \\
&\propto \prod_{i=1}^{n} \pi_{y_i}\mathcal{N}(\mathbf{x_i};\boldsymbol{\mu}_{y_i},\boldsymbol{\Sigma}_{y_i}) \qquad (16)
\end{aligned}$$

Since each y_i only appears once in this product, it is straightforward to maximise the value of this probability by maximising $p(y_i|\mathbf{x}_i)$

$$\begin{aligned}
y_i &= \operatorname*{argmax}_k p(y_i = k|\mathbf{x}_i) \\
&= \operatorname*{argmax}_k \pi_k\mathcal{N}(\mathbf{x_i};\boldsymbol{\mu}_k,\boldsymbol{\Sigma}_k) \quad \text{for } k \in \{1,2,\ldots,c\} \qquad (17)
\end{aligned}$$

where the proportionality in the last step of eq. (16) is exploited to simplify the equation. To evaluate this posterior distribution, the model parameters $\boldsymbol{\theta}$ must be known. They can be estimated using the Maximum Likelihood Estimator (MLE) from the log likelihood function l (see (appendix A)). The likelihood of the observed data $\mathbf{X}$ is given by $l(\boldsymbol{\theta}|\mathbf{X}) = \ln p(\mathbf{X}|\boldsymbol{\theta})$. However, it is typically computationally intractable to find maximum likelihood estimates of $\boldsymbol{\theta}$ from this distribution [33].

Instead consider the joint log likelihood over all the variables $(\mathbf{X}, \mathbf{y})$

$$\begin{aligned}
l(\boldsymbol{\theta}|\mathbf{X},\mathbf{y}) &= \ln p(\mathbf{X},\mathbf{y}|\boldsymbol{\theta}) \\
&= \sum_{i=1}^{n} \ln p(\mathbf{x_i}, y_i|\boldsymbol{\theta}) \\
&= \sum_{i=1}^{n} \ln p(y_i|\boldsymbol{\theta}) + \ln p(\mathbf{x}_i|y_i,\boldsymbol{\theta}) \\
&= \sum_{i=1}^{n} \ln \pi_{y_i} + \ln \mathcal{N}(\mathbf{x}_i;\boldsymbol{\mu}_{y_i},\boldsymbol{\Sigma}_{y_i}) \qquad (18)
\end{aligned}$$

From this distribution it is computationally tractable to obtain MLE estimates of $\boldsymbol{\theta}$, but it requires knowing the latent variables $\mathbf{y}$. This means there is a circular dependency, where estimating $\mathbf{y}$ requires knowing $\boldsymbol{\theta}$ and estimating $\boldsymbol{\theta}$ requires knowing $\mathbf{y}$.

Now assume there is an estimate of the posterior distribution of $\mathbf{y}$ from eq. (16) available with some approximate parameters $\hat{\boldsymbol{\theta}}$, i.e. $p(\mathbf{y}|\mathbf{X},\hat{\boldsymbol{\theta}})$. The expected value of eq. (18) with respect to this distribution is then

$$\begin{aligned}
\mathbb{E}_{p(\mathbf{y}|\mathbf{X},\hat{\boldsymbol{\theta}})}[\ln p(\mathbf{X},\mathbf{y}|\boldsymbol{\theta})] &= \sum_{\mathbf{y}} p(\mathbf{y}|\mathbf{X},\hat{\boldsymbol{\theta}}) \ln p(\mathbf{X},\mathbf{y}|\boldsymbol{\theta}) \\
&= \sum_{i=1}^{n}\sum_{k=1}^{c} p(y_i = k|\mathbf{x}_i,\hat{\boldsymbol{\theta}})(\ln p(\mathbf{x}_i, y_i|\boldsymbol{\theta})) \\
&= \sum_{i=1}^{n}\sum_{k=1}^{c} p(y_i = k|\mathbf{x}_i,\hat{\boldsymbol{\theta}})(\ln p(y_i|\boldsymbol{\theta}) + \ln p(\mathbf{x}_i|y_i,\boldsymbol{\theta})) \\
&= \sum_{i=1}^{n}\sum_{k=1}^{c} \frac{\hat{\pi}_k \mathcal{N}(\mathbf{x_i};\hat{\boldsymbol{\mu}}_k,\hat{\boldsymbol{\Sigma}}_k)}{\sum_{j=1}^{c}\hat{\pi}_j \mathcal{N}(\mathbf{x_i};\hat{\boldsymbol{\mu}}_j,\hat{\boldsymbol{\Sigma}}_j)}(\ln \pi_k + \ln \mathcal{N}(\mathbf{x}_i;\boldsymbol{\mu}_k,\boldsymbol{\Sigma}_k)) \qquad (19)
\end{aligned}$$

By taking the expected value with respect to the posterior distribution of $\mathbf{y}$ using $\hat{\boldsymbol{\theta}}$, the dependency on $\mathbf{y}$ is removed from the likelihood function, and now MLE estimates for $\boldsymbol{\theta}$ can be obtained.

The idea behind EM is therefore to alternately improve the estimates of the latent variables $\mathbf{y}$ and model parameters $\boldsymbol{\theta}$ by the using the current estimate of one to improve the estimate of the other. This is done by generating a sequence of estimates $\{\boldsymbol{\theta}^0, \boldsymbol{\theta}^1, \dots\}$ where the value of the likelihood function will increase monotonically each iteration to some local minimum l^* [48]. It can be summarized in the following procedure

1. Initialize the model parameters $\boldsymbol{\theta}^0$.
2. **E-Step:** Evaluate the posterior distribution $p(\mathbf{y}|\mathbf{X},\boldsymbol{\theta}^t)$ from eq. (16) using the current estimated model parameters $\boldsymbol{\theta}^t$. Use this distribution to evaluate the expected value of the joint likelihood function in eq. (19).
3. **M-Step:** Obtain new MLE estimates of the model parameters $\boldsymbol{\theta}^{t+1}$ from the expected value of the joint log likelihood function.
4. Repeat steps 2-3 until the parameters converge, i.e. $|\boldsymbol{\theta}^{t+1} - \boldsymbol{\theta}^t| < \epsilon$.
5. Output the final clustering labels $\mathbf{y}$ by maximising the posterior distribution using eq. (17).

The update rules for the MLE estimates in step 3 is given by [33]

$$N_k = \sum_{i=1}^{n} p(y_i = k|\mathbf{x}_i) \qquad (20)$$

$$\boldsymbol{\mu}_k = \frac{1}{N_k}\sum_{i=1}^{n} p(y_i = k|\mathbf{x_i})\mathbf{x}_i \qquad (21)$$

$$\boldsymbol{\Sigma_k} = \frac{1}{N_k}\sum_{i=1}^{n} p(y_i = k|\mathbf{x}_i)(\mathbf{x}_i - \boldsymbol{\mu}_k)(\mathbf{x}_i - \boldsymbol{\mu}_k)^T \qquad (22)$$

$$\pi_k = \frac{N_k}{n} \qquad (23)$$

Figure 8 shows the GMM clustering results on the synthetic datasets. GMM has an implicit assumption that the data clusters are higher dimensional ellipsoids, similar to how KMeans from section 3.1 assumes that clusters are hyper-spheres. The plotted ellipses are contours of the estimated Gaussian distributions where $p(\mathbf{x}|y_i)$ is constant.

3.3.2 Label-dependent Spectral Mixture Model with Markov Random Field

An extension of this algorithm called Label-dependent Spectral Mixture Model with Markov Random Field (LSMM-MRF) is proposed in [11]. It introduces two changes, and the first is to modify

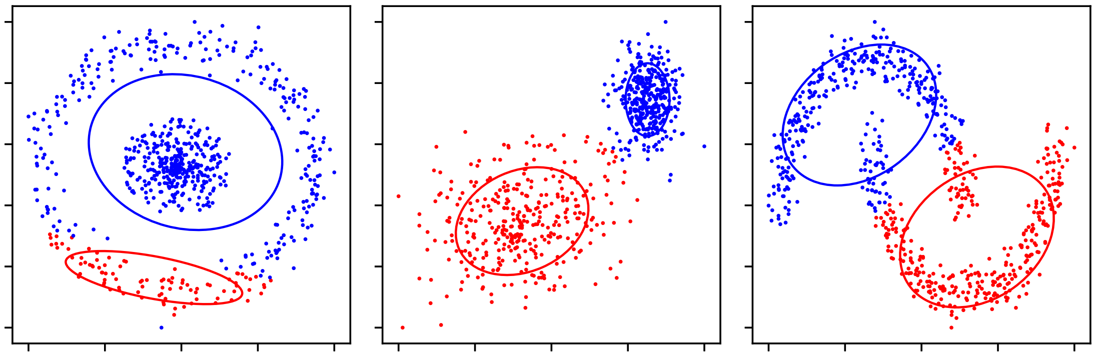


Figure 8: GMM clustering on synthetic datasets.

the generative model in eq. (14) to instead consider every pixel $\mathbf{x}_i$ as a mix of spectral endmembers given by

$$\mathbf{x}_i = \sum_{j=1}^{c} s_i^j \boldsymbol{\mu}_j + \mathbf{z} \tag{24}$$

where $s_i^j > 0$ is the abundance of endmember j in pixel i and $\mathbf{z}$ is uncorrelated Gaussian noise. They argue that this model better captures the inner-class variation, as well as modeling mixed pixels resulting from limited spatial resolution. Let $\mathbf{M} = \begin{bmatrix}\boldsymbol{\mu}_1 & \boldsymbol{\mu}_2 & \dots & \boldsymbol{\mu}_c\end{bmatrix}$ and $\mathbf{s}_i = \begin{bmatrix}s_i^1 & s_i^2 & \dots & s_i^c\end{bmatrix}^T$ and $\mathbf{S} = \{\mathbf{s}_1, \mathbf{s}_2, \dots, \mathbf{s}_n\}$. The conditional distribution of $\mathbf{x}$ from eq. (15) then becomes

$$p(\mathbf{x}_i | y_i = k) = \mathcal{N}(\mathbf{x}_i; \mathbf{M}\mathbf{s}_i, \boldsymbol{\Lambda}), \quad s_i^k > s_i^{j \neq k} > 0 \tag{25}$$

Note that the constraint imposed on the abundances $\mathbf{s}_i$ makes them coupled with y_i in a convoluted way. The second difference is to introduce a Markov Random Field (MRF) prior on the clustering labels $\mathbf{y}$. They consider every pixel as a node which is connected to its surrounding pixels as in fig. 9.

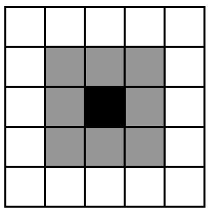


Figure 9: Neighborhood system for the Markov Random Field prior. Every pixel i (black) is connected to its spatially surrounding pixels (grey)

The energy function between two neighboring pixels $\{i, j\}$ is given by $\beta(1 - \delta(y_i, y_j))$, where δ is the Kronecker function such that $\delta(y_i, y_j) = 1$ if $y_i = y_j$ and 0 otherwise and β is a positive constant. This promotes a lower energy, i.e. a higher probability when $y_i = y_j$, and therefore biases the labels towards to be spatially smooth. The full energy function for the prior is then

$$E(\mathbf{y}) = \beta \sum_{\{i,j\} \in \mathcal{N}} 1 - \delta(y_i, y_j) \tag{26}$$

where $\mathcal{N}$ is the set of all neighboring pixels. This gives the joint Gibbs distribution prior over all the labels as

$$p(\mathbf{y}) = \frac{1}{V}\exp\{-E(\mathbf{y})\} \tag{27}$$

where $V = \sum_{\mathbf{y}} p(\mathbf{y})$ is the normalizing constant. Note that this distribution is computationally difficult to even evaluate as it requires summing over the c^n different possible values of $\mathbf{y}$. Let $\boldsymbol{\theta} = \{\mathbf{M}, \mathbf{S}\}$, i.e. the model parameters as before. To optimize this model and solve the Maximum A Posteriori problem required for estimating the labels $\mathbf{y}$ they propose the following EM like algorithm

1. Initialize the model parameters $\boldsymbol{\theta}^0$.
2. **E-Step** Find $\hat{\mathbf{y}} = \text{argmax}_{\mathbf{y}}\, p(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta}^t)$, the MAP estimate of $\mathbf{y}$, using *energy minimization.*
3. **M-Step** Given $\hat{\mathbf{y}}$, obtain new estimates $\mathbf{M}^{t+1}$ from *purified means* and new estimates $\mathbf{S}^{t+1}$ using *non negative least squares* (NNLS).
4. Repeat steps 2-3 until the parameters converge, i.e. $|\boldsymbol{\theta}^{t+1} - \boldsymbol{\theta}^t| < \epsilon$
5. Output $\hat{\mathbf{y}}$ as the final clustering assignment

Note the difference compared to the true EM algorithm presented in section 3.3.1. Instead of finding the posterior $p(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta})$, [11] insert the MAP estimate $\hat{\mathbf{y}}$ directly in the joint likelihood from eq. (18). This is equivalent to assigning all the probability to the mode of the posterior as

$$p(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta}) = \begin{cases} 1, & \text{if } \mathbf{y} = \text{argmax}_{\mathbf{y}'}\, p(\mathbf{y}'|\mathbf{X}, \boldsymbol{\theta}) \\ 0, & \text{otherwise} \end{cases} \tag{28}$$

The authors of [11] make no comment on, or justification for, this simplification. Presumably it is intended, and a strategy to simplify calculations. It is however unclear how this affects the correctness guarantee of EM and convergence properties as given in [48].

Energy Minimization

Energy Minimization formulates the MAP estimation problem as a minimization problem given by minimizing the negative log posterior

$$\begin{aligned} \hat{\mathbf{y}} &= \underset{\mathbf{y}}{\text{argmax}}\, p(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta}) \\ &= \underset{\mathbf{y}}{\text{argmin}} - \ln p(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta}) \\ &= \underset{\mathbf{y}}{\text{argmin}} - \ln p(\mathbf{X}|\mathbf{y}, \boldsymbol{\theta}) - \ln p(\mathbf{y}) \end{aligned} \tag{29}$$

where the last equality comes from Bayes Theorem.

There are two competing terms in this minimization problem. The first term in eq. (25) is the data energy $E_{\text{data}} = -\ln p(\mathbf{X}|\mathbf{y}, \boldsymbol{\theta})$. It will decrease for choices of $\mathbf{y}$ that makes the observed data $\mathbf{X}$ more likely based on the model in eq. (25). This biases the solution towards clusters that have similar spectral signatures. If pixel i belongs to cluster j, it is expected that $s_i^j > s_i^{k \neq j}$ for $k \in \{1, 2, \ldots, c\}$ since the abundance of endmember $\boldsymbol{\mu}_j$ should be large. The likelihood that pixel i belongs to cluster j is therefore expected to be inversely proportional to s_i^j. For computationally efficiency and simplicity [11] therefore uses the following approximation as the energy function for E_{data}

$$E_{\text{data}} = \sum_{i=1}^{n} \frac{1}{s_i^{y_i}} \tag{30}$$

The last term in eq. (25) is called the smoothness energy $E_{\text{smooth}} = -\ln p(\mathbf{y})$ and it will decrease for choices of $\mathbf{y}$ that are spatially smooth. So this term biases the solution towards clusters that are spatially connected

$$\begin{aligned} \underset{\mathbf{y}}{\operatorname{argmin}}\, E_{\text{smooth}} &= \underset{\mathbf{y}}{\operatorname{argmin}} -\ln\left\{\frac{1}{V}\exp(-E(\mathbf{y}))\right\} \\ &= \underset{\mathbf{y}}{\operatorname{argmin}}\, E(\mathbf{y}) \\ &= \underset{\mathbf{y}}{\operatorname{argmin}}\, \beta \sum_{\{i,j\}\in\mathcal{N}} 1 - \delta(y_i, y_j) \end{aligned} \tag{31}$$

The resulting minimization problem is

$$\hat{\mathbf{y}} = \underset{\mathbf{y}}{\operatorname{argmin}} \left\{ \sum_{i=1}^{n} \frac{1}{s_i^{y_i}} + \beta \sum_{\{i,j\}\in\mathcal{N}} 1 - \delta(y_i, y_j) \right\} \tag{32}$$

The constant β can therefore be thought of as a hyperparameter that weights the relative importance of spatial smoothness and fitting the observed pixels.

There exists many techniques for solving energy minimization problems and the authors of [11] uses the α-expansion grid algorithm from [49]. This requires constructing a graph with vertices for each pixel and vertices between each neighboring pixel that does not have the same label, and then solving a min cut problem.

Purified Means

To estimate the endmembers $\mathbf{M}$, the authors of [11] adopt the purified means approach from [50]. The idea is to consider all the pixels currently assigned to a cluster j and estimate the endmember $\boldsymbol{\mu}_j$ by removing the abundance of other endmembers in the pixels as

$$\boldsymbol{\mu}_j = \frac{1}{|C_j|}\Big(\sum_{\mathbf{x}_i \in C_j} \mathbf{x}_i - \sum_{k\neq j}^{c} s_i^k \boldsymbol{\mu}_k \Big) \tag{33}$$

Non Negative Least Squares

To find the abundances $\mathbf{S}$ that best explains the observed data given the endmembers $\mathbf{M}$, [11] adopts the non negative least squares (NNLS) method from [51].

$$\mathbf{s}_i = \underset{\mathbf{s'}_i}{\operatorname{argmin}}\, ||\mathbf{x}_i - \mathbf{M}\mathbf{s'}_i||_2 \quad \mathbf{s'}_i > 0 \tag{34}$$

Sub algorithm	Time Complexity	Space Complexity
Energy Minimization using α expansion [49]	$O(cn^2)$	$O(cn)$
Purified Means	$O(cnd)$	$O(1)$
Non Negative Least Squares using fast NNLS [52]	$O(cnd)$	$O(1)$

Table 5: Computational complexity for the different steps in the LSMM-MRF algorithm

Table 5 shows the asymptotic computational complexity for the sub algorithms for LSMM-MRF. In addition there is an additional memory complexity of $O(nc)$ for storing $\mathbf{S}$. Running the algorithm for t iterations and noting that $d < n$ gives a total time complexity of $O(cn^2t)$ and memory complexity of $O(cn)$.

@Joe This is very long compared to the other ones, do you think I should try to make it shorter?

3.4 Density Based Clustering

Density based clustering is a clustering technique where the idea is to identify regions in the input data space with high density of data points. The assumption is that high density regions consist of similar points and that clusters should be formed from those regions.

DBSCAN [36] is a popular density based clustering algorithm. It aims to partition the data points $\mathbf{X}$ into three sets

- **Core Points** P_c: Points in regions with high density
- **Border Points** P_b: Points on the edge of regions with high density
- **Noise Points** P_n: Points in regions with low density

and then form clusters from continuous regions of core and border points. Let $\mathcal{N}_i$ be the set of neighboring points to $\mathbf{x}_i$ given by

$$\mathcal{N}_i = \{\mathbf{x} \in \mathbf{X} : k(\mathbf{x}_i, \mathbf{x}) < \epsilon\} \tag{35}$$

where k is a metric such as the euclidian distance and ϵ is a positive constant. Note that $\mathbf{x}_i \in \mathcal{N}_i$.

The number of neighbors $|\mathcal{N}_i|$ is an indication of density, where $|\mathcal{N}_i|$ is expected to be higher in high density regions. Set membership is determined from a threshold parameter M where $\mathbf{x}_i$ is a considered a core point if it has at least M neighbors

$$P_c = \{\mathbf{x}_i \in \mathbf{X} : M \leq |\mathcal{N}_i|\} \tag{36}$$

A point $\mathbf{x}_j$ is *reachable* from point $\mathbf{x}_i$ if $\mathbf{x}_j \in \mathcal{N}_i$. Since metrics are symmetric, $k(\mathbf{x}_i, \mathbf{x}_j) = k(\mathbf{x}_j, \mathbf{x}_i)$, reachability is a symmetric relationship. Border points P_b are points that are not core points themselves, but still reachable from a core point

$$P_b = \{\mathbf{x}_i \in \mathbf{X} : \mathbf{x}_i \notin P_c \wedge P_c \cap \mathcal{N}_i \neq \emptyset\} \tag{37}$$

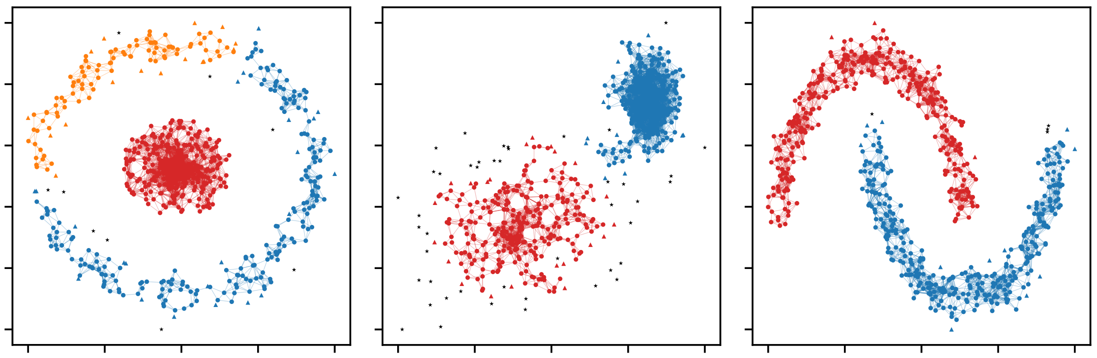
where $\emptyset$ is the empty set. The noise points are then points that are not core points, and not reachable from a core point

$$P_n = \{\mathbf{x}_i \in \mathbf{X} : \mathbf{x}_i \notin P_c \wedge P_c \cap \mathcal{N}_i = \emptyset\} \tag{38}$$

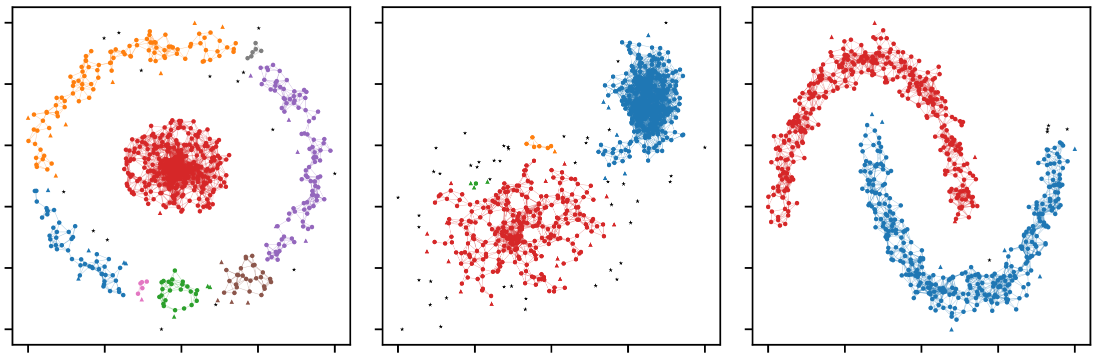
After the points is partitioned, clusters are formed from the core points P_c and border points P_b. The clusters are core points that are reachable from each other, plus any border points reachable from those core points. Construct an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with the points as nodes and edges between reachable points where at least one of the points is a core point

$$\mathcal{V} = P_c \cup P_b \tag{39}$$
$$\mathcal{E} = \{(i, j) : \mathbf{x}_i \in P_c \wedge \mathbf{x}_j \in \mathcal{N}_i\} \tag{40}$$

Clusters are then the set of nodes in every connected subgraph. The number of clusters are in other words determined by the algorithm itself. The key challenge with DBSCAN it to tune the hyperparameters ϵ and M as the clustering results are very sensitive to their values. Figure 10 shows the DBSCAN algorithm on the synthetic datasets. Figure 10a uses $\epsilon = 0.05$ and $M = 5$, while fig. 10b uses $\epsilon = 0.045$ and $M = 4$. With this small perturbation in hyperparameters, the clustering of the circles dataset changes significantly. There has been significant work done on automatic hyperparameter tuning for DBSCAN, such as [53] which uses an evolutionary algorithm to generate good candidates forϵ and M.



(a) DBSCAN clustering on synthetic datasets with $\epsilon = 0.05$ and $M = 5$



(b) DBSCAN clustering on synthetic datasets with $\epsilon = 0.045$ and $M = 4$.

Figure 10: DBSCAN clustering with euclidean metric. The connected graphs are the core points, the triangles the border points and the black stars are the noise points. The edges between core points and border points are not drawn to help differentiate between the two sets

The time complexity of DBSCAN is claimed to be $O(dn \log n)$ in the original paper [36] since each point would on average only calculate the similarity to $O(\log n)$ other points. However [54] proved that it is really $O(dn^2)$ for $d > 2$. The memory complexity is potentially $O(n)$ for storing the set membership and $\mathcal{N}_i$ for each pixel $\mathbf{x}_i$. However this requires repeatedly computing the distance metric for the same points, inducing a severe computational speed penalty. However most implementations utilize some data structure to speed up computations in order to reuse the calculations. The naive solution is to store the full $O(n^2)$ similarity matrix similar to spectral clustering in section 3.2. As noted in [55], a more efficient approach is to store only the distance to the neighboring points $\mathcal{N}_i$, reducing the memory complexity to $O(n|\mathcal{N}_{\max}|)$ where $|\mathcal{N}_{\max}|$ is the average number of adjacent vertices for each vertex in the graph.

3.5 Self Organizing Map

The self-organizing map (SOM) [17] is an unsupervised machine learning (section 2.7) model that learns a dense representation of the input data. The input data is mapped to a low dimensional space, typically consisting of far fewer data points than the original data. This is done by creating a map of nodes that preserves the distribution of the input data while also representing the full dataset well. While the map can have any structure, the most common is a finite two dimensional rectangular grid as in fig. 11.

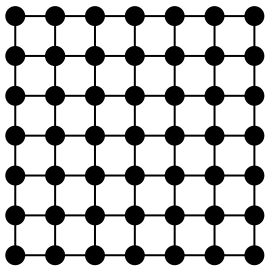


Figure 11: A 2D grid SOM with dimension 7

The number of rows and columns in the grid is called the dimension of the SOM. Each node is represented by vector $\mathbf{z}$ with the same dimension as the input data points to be mapped. Let $\mathbf{X} = \begin{bmatrix}\mathbf{x}_1 & \mathbf{x}_2 & \dots & \mathbf{x}_n\end{bmatrix}$ where $\mathbf{x}_i \in \mathcal{R}^d$ be the input data and $\mathbf{Z} = \begin{bmatrix}\mathbf{z}_{1,1} & \mathbf{z}_{1,2} & \dots & \mathbf{z}_{m,m}\end{bmatrix}$ where $\mathbf{z}_i \in \mathcal{R}^d$ be the $m \times m$ grid of SOM nodes. The map revolves around the concept of a best matching unit (BMU) given by

$$\mathrm{BMU}(\mathbf{x}_i) = \underset{\mathbf{z} \in \mathbf{Z}}{\operatorname{argmin}}\, k(\mathbf{x}_i, \mathbf{z}) \tag{41}$$

for some similarity metric k like the euclidean distance. The BMU for a sample point $\mathbf{x}_i$ can be thought of as the representative node or encoding for that point. When the map is trained, any two points that are close (in the sense of k) in the input data space will have their BMUs be identical or spatially close in the SOM grid, therefore the grid preserves the topology of the input space.

Training the map is done by moving the nodes closer to the input data while preserving the relative topology of the nodes as illustrated in fig. 12.

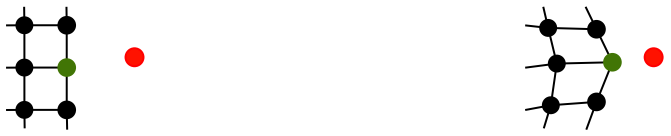

(a) Before training the nodes on the new sample (b) After training the nodes on the new sample

Figure 12: Training the SOM on a new input sample (red). The best matching unit (green) is affected the most, while the neighboring nodes are less influenced. The distance between the nodes is proportional to the similarity in the data space.

Training the map to learn the input data can be summarized in the following procedure

1. Initialize the nodes $\mathbf{Z}$.
2. For each input sample $\mathbf{x}_i$ in the input data $\mathbf{X}$: Train the map nodes $\mathbf{Z}$ on $\mathbf{x}_i$.
3. Repeat step 2 for T iterations.

The core idea when training the map on an input sample $\mathbf{x}_i$ is to update $\mathrm{BMU}(\mathbf{x}_i)$ to decrease $k(\mathbf{x}_i, \mathrm{BMU}(\mathbf{x}_i))$ to improve the BMUs representation of the input sample. The neighboring nodes of the BMU in the SOM grid should also be updated, but with a smaller magnitude than the

BMU. To decrease $k(\mathbf{x}_i, \mathbf{z}_j)$ a numerical minimization algorithm can be used. Gradient Descent [56] is a commonly used algorithm that reaches local minimums by using the local gradient of the function. Since the gradient of a function gives the local direction of steepest ascent, gradient descent performs minimization by stepping in the opposite direction of the gradient. Let $f(\mathbf{x})$ be the objective function to minimize. Gradient Descent then iteratively updates $\mathbf{x}$ according to

$$\mathbf{x} \longleftarrow \mathbf{x} - \alpha \nabla_{\mathbf{x}} f(\mathbf{x}) \tag{42}$$

where α is a positive constant often referred to as the *learning rate* and $\nabla_{\mathbf{x}} f = \begin{bmatrix} \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_2} & \dots & \frac{\partial f}{\partial x_d} \end{bmatrix}^T$ is the gradient.

Let $\mathbf{z}_i^* = \text{BMU}(\mathbf{x}_i)$ denote the (current) BMU for input vector $\mathbf{x}_i$ from eq. (41). The gradient descent update step for each SOM node $\mathbf{z}_j$ is then given by

$$\mathbf{z}_j \longleftarrow \mathbf{z}_j - \alpha(t)\beta(t, \mathbf{z}_j, \mathbf{z}_i^*)\nabla_{\mathbf{z}_i} k(\mathbf{x}_i, \mathbf{z}_j) \tag{43}$$

where $\alpha(t) > 0$ is the learning rate for iteration t, $\beta(t, \mathbf{z}_j, \mathbf{z}_i^*) > 0$ is the neighborhood function.

The learning rate $\alpha(t)$ determines how fast the map learns the input data. It is a common strategy to let $\alpha(t)$ be a monotonically decreasing function in order to first learn rough features, then fine tune towards the end of training. Common choices for $\alpha(t)$ are

$$\alpha(t) = \alpha_0 e^{-t/\tau} \tag{44}$$

$$\alpha(t) = \alpha_0 - \alpha_1 t \tag{45}$$

where α_0, α_1 and τ are positive constants.

The influence $\beta(t, \mathbf{z}_j, \mathbf{z}_i^*)$ preserves the topology by training nodes spatially far from $\mathbf{z}_i^*$ with a lower magnitude. It usually involves some neighborhood function $\sigma(t)$ which decreases in a similar fashion to $\alpha(t)$ to initially learn the rough distribution topology, then fine tune the local areas of the distribution towards the end. Common choices for β are

$$\beta(t, \mathbf{z}_j, \mathbf{z}_i^*) = e^{-d(\mathbf{z}_j, \mathbf{z}_i^*)/\sigma(t)^2} \tag{46}$$

$$\beta(t, \mathbf{z}_j, \mathbf{z}_i^*) = \begin{cases} 1, & \text{if } d(\mathbf{z}_j, \mathbf{z}_i^*) < \sigma(t)) \\ 0, & \text{otherwise} \end{cases} \tag{47}$$

where $d(\mathbf{z}_j, \mathbf{z}_i^*)$ is the euclidean distance between the nodes positions in the grid and $\sigma(t)$ is the neighborhood function commonly taken to be the an exponential decaying function similar to eq. (44).

Training the map itself is quite slow. For each training iteration t the each sample $\mathbf{x}_i$ must apply eq. (43) to each node $\mathbf{z}_j$ resulting in $O(dnz^2t)$ with fairly large coefficients since it requires finding the BMU, calculating the gradient for each node and multiplicating it with both the learning rate and neighborhood function. The memory usage is on the other hand low since it only requires storing the nodes $\mathbf{Z}$.

The SOM can be used as a clustering algorithm where each SOM node is a cluster. This naturally leads to a high number of clusters depending on the map size. Sensible map sizes could be $16 \times 16 = 256$ nodes in total, resulting in a one byte encoding of the data. In section 5 a method to use the SOM as an intermediate stage for clustering is proposed as a means for speeding up clustering computations.

Algorithm	Time Complexity	Space Complexity
KMeans	$O(cdnt)$	$O(cd) + O(n)$
FSC	$O(dn^2)$	$O(n^2)$
LSMM-MRF	$O(cn^2t)$	$O(cn)$
DBSCAN	$O(dn^2)$	$O(n)$
SOM	$O(dnz^2t)$	$O(z^2d)$

Table 6: Computational complexity for various clustering algorithms

3.6 Summary

A summary of the presented clustering algorithms and their respective computational complexity can be seen in table 6.

3.7 Evaluation Metrics

For evaluating the output of a clustering algorithm there exists several objective metrics as presented below. They require knowing the ground truth (real) clustering assignments, which will often have to be created manually. One clustering can be preferred over another despite having an equal metric based on the use case, but this is often a subjective judgment to be done by a human. As before, let $C = \{C_1, C_2, \dots, C_c\}$ be the clustered sets of n pixels and let $T = \{T_1, T_2, \dots, T_t\}$ be the ground truth clustering.

3.7.1 Purity

As the name suggests, purity measures the inter cluster coherency and is given by

$$\text{purity}(C, T) = \frac{1}{n} \sum_{i=1}^{c} \max_j |C_i \cap T_j| \tag{48}$$

it simply assigns each cluster the class of the most common ground truth class and calculates the percentage of points that were assigned to the correct cluster.

3.7.2 Normalized Mutual Information

Normalized Mutual Information (NMI) measures the amount of information extracted by the clustering algorithm [57] and is given by

$$\text{NMI}(C, T) = \frac{2I(C, T)}{H(C) + H(T)} \tag{49}$$

where I is the mutual information

$$I(C, T) = \sum_{i=1}^{c} \sum_{j=1}^{t} \frac{|C_i \cap T_j|}{n} \log_2 \frac{n|C_i \cap T_j|}{|C_i||T_j|} \tag{50}$$

and H is the entropy

$$H(C) = -\sum_{i=1}^{c} \frac{|C_i|}{n} \log_2 \frac{|C_i|}{n} \tag{51}$$

3.8 Dimensionality Reduction for faster clustering

Principal Component Analysis (PCA) is a fast and powerful dimensionality reduction algorithm that can reduce a complex set of data to a lower dimension to reveal simplified structures [19]. While reducing the data to a lower dimension potentially discards some information in the data, it allows for faster computations by decreasing the number of bands d in the input image. PCA exploits eigendecomposition to identify the principle components in the data and projects the data down into a lower dimensional subspace. Let $\mathbf{X} = \begin{bmatrix}\mathbf{x}_1 & \mathbf{x}_2 & \dots & \mathbf{x}_n\end{bmatrix}$ where $\mathbf{x}_i = \begin{bmatrix}x_{i1} & x_{i2} & \dots & x_{id}\end{bmatrix}$ be the input data. The objective of PCA is to find some matrix $\mathbf{Z} = \begin{bmatrix}\mathbf{z}_1 & \mathbf{z}_2 & \dots & \mathbf{z}_n\end{bmatrix} \in \mathcal{R}^{n\times m}$ where each $\mathbf{x}_i \in \mathcal{R}^d$ is transformed to some $\mathbf{z}_i \in \mathcal{R}^m$ where $m < d$ in a manner which preserves the information of interest in the data. Specifically consider the set of linear transformations

$$\mathbf{z}_i = \mathbf{P}\mathbf{x}_i \tag{52}$$

where $\mathbf{P} \in \mathcal{R}^{m\times d}$ is some projection matrix. Let $\widetilde{\mathbf{x}}_i$ denote the centered vector $\mathbf{x}_i$ as

$$\widetilde{\mathbf{x}}_i = \mathbf{x}_i - \overline{\mathbf{x}} \tag{53}$$

where $\overline{\mathbf{x}}$ is the mean. Let $\widetilde{\mathbf{X}}$ denote the matrix of cenetered data points. The covariance of the matrix $\mathbf{X}$ is then the symmetric matrix given by

$$\Sigma_{\mathbf{X}} = \frac{1}{n}\widetilde{\mathbf{X}}\widetilde{\mathbf{X}}^T \tag{54}$$

where $(\Sigma_{\mathbf{X}})_{ij}$ denotes the covariance between feature i and j. PCA finds the projection $\mathbf{P}$ that maximises the kept variance of $\mathbf{Z} = \mathbf{PX}$ along each dimension, which is the trace of the covariance matrix of $\mathbf{Z}$ and it can be found by considering the eigenvalue decomposition of $\Sigma_{\mathbf{X}}$ [33]

$$\Sigma_{\mathbf{X}} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^{-1} \tag{55}$$

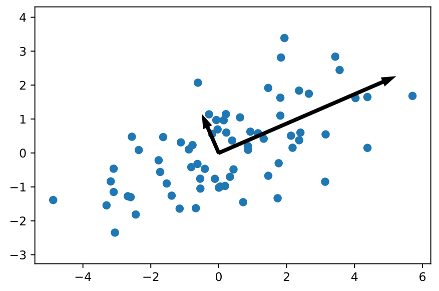
where Λ is a diagonal matrix with eigenvalues $\begin{bmatrix}\lambda_1 & \lambda_2 & \dots\lambda_d\end{bmatrix}$ along the diagonal and $\mathbf{Q} = \begin{bmatrix}\mathbf{q}_1 & \mathbf{q}_2 & \dots & \mathbf{q}_d\end{bmatrix}$ are the corresponding orthogonal eigenvectors. Assume the eigenvalues are sorted such that $\lambda_1 > \lambda_2 > \dots > \lambda_d$. Then the optimal projection matrix is

$$\mathbf{P} = \begin{bmatrix}\mathbf{q}_1 & \mathbf{q}_2 & \dots\mathbf{q}_m\end{bmatrix}^T \tag{56}$$

where $\mathbf{q}_i$ is called the i-th principal component of $\mathbf{X}$. The amount of variance explained by each component is given by its corresponding eigenvalue λ_i. Increasing the number of components therefore increases the kept variance in the data, or equivalently minimizes the amount of lost information in the data from using $\mathbf{z}$ instead of $\mathbf{x}$. Figure 13 shows an example of the how the data is projected with $m = 1$ component. The input data and principal components are shown in fig. 13a where the principal components is scaled with the eigenvalues. The data is then projected on the first (largest eigenvalue) principal component and the result is fig. 13b.

Choosing the number of components m to project on is inherently a trade off between reducing the dimension and preserving information of value in the data. The amount of variance explained by the components typically drops in an exponential manner as illustrated in fig. 14, and thus there are diminishing returns for each additional component kept.

Calculating the covariance matrix is computationally expensive with $O(dn^2)$ operations. However once $\mathbf{P}$ is found the projection itself is quite fast and is done in $O(dmn)$. Given the projection matrix $\mathbf{P}$ and the mean $\overline{\mathbf{x}}$, projection is just a vector subtraction to center the vector (eq. (53)) and a matrix multiplication (eq. (52)). The memory complexity is also low with $O(md)$ for storing the projection matrix $\mathbf{P}$.


(a) Input data with principal components

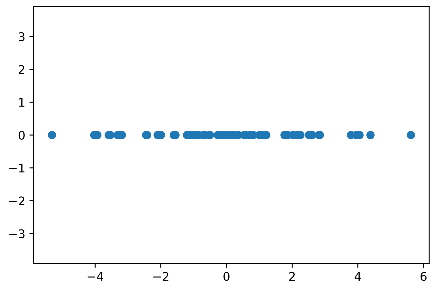

(b) Projected data with $m = 1$ components

Figure 13: PCA dimensionality reduction from $\mathcal{R}^2$ to $\mathcal{R}$

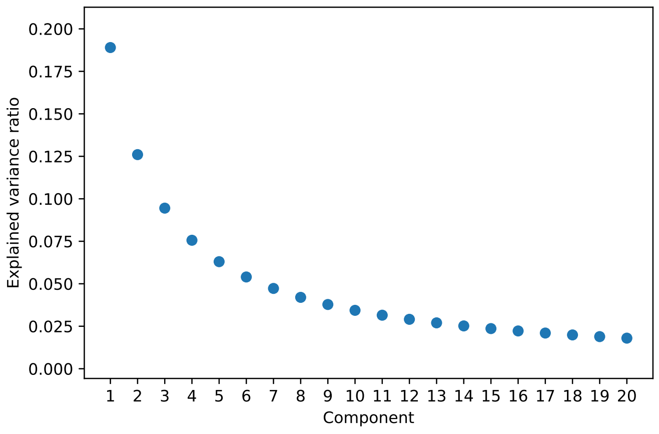


Figure 14: Typical ratio of total variance explained by each PCA component

4 Classification

This section will give a brief overview of classification, before classification algorithms is presented. Lastly the Overall Accuracy evaluation metric is introduced.

Classification algorithms aims to assign each input vector in a dataset to one of a finite number of discrete categories or *classes* [33]. Each datapoint is a pair $(\mathbf{x}_i, y_i)$ where y_i is the label associated with the input vector $\mathbf{x}_i$. In general the label y_i is unknown and the objective is to correctly estimate y_i given $\mathbf{x}_i$. It is a form of supervised learning (section 2.7), meaning it requires a set of known $(y, \mathbf{x})$ pairs to be used as examples for classifying new input vectors $\mathbf{x}$ with unknown labels. The labeled set of example data is called the *training data.* For hyperspectral images, the input vectors are the spectra of the individual pixels, while the class to be estimated is typically the material of which the pixel originates.

n_t	Number of pixels in the training set
d	Number of spectral bands
k	Number of classes
$\mathbf{X}_t$	Training data samples
$\mathbf{y}_t$	The class labels for the training data

Table 7: Classification Notation

Many classification algorithms are separated into two phases. First comes the *training* (or *learning*) phase in which a model is learned from the training data. The training procedure aims to find a model that will learn the latent patterns in the data such that it can generalize the information to correctly classify new, unseen samples. What constitutes the best model is typically decided by some objective function, and the learning phase therefore involves searching the space of models for parameters that maximize the objective function. Let $\boldsymbol{\theta}$ be the set of all model parameters and $\mathbf{X}_t = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{n_t}\}$, $\mathbf{y}_t = \{y_1, y_2, \ldots, y_{n_t}\}$ be the training data where n_t is the number of training samples. The learning phase can then be considered as a function g that takes the training data and hyperparameters as input and outputs the best set of model parameters $\boldsymbol{\theta}^*$

$$\boldsymbol{\theta}^* = g(\mathbf{X}_t, \mathbf{y}_t, \boldsymbol{\lambda}) \tag{57}$$

The second phase is the classification itself. Given a new data sample $\mathbf{x}$, the objective is to estimate the unknown class y. Using the learned parameters $\boldsymbol{\theta}$ a classifier is a function h that outputs the estimate $\hat{y}$

$$\hat{y} = h(\mathbf{x}, \boldsymbol{\theta}^*) \tag{58}$$

How computational efficient the training phase is does not have a large impact for the HYPSO mission as it is done on the ground. The resulting model parameters $\boldsymbol{\theta}^*$ can then be stored on the satellite to be used by the classifier. Naturally the computational efficiency of the classifier will be important, both the time used for each point as well as the memory usage.

4.1 K Nearest Neighbors

K Nearest Neighbors (KNN) [58] is a simple algorithm which performs classification of a new input $\mathbf{x}$ by directly identifying similar input vectors in the training data. This is based on the assumption that if two input vectors are similar based on some distance metric , then it is likely that their labels are identical. The idea is then to compare the new sample to the existing samples in the training set and find the most similar points. The labels of these points are known and the label of the new point it taken to be the most frequent label among these similar points. KNN therefore makes no assumption of the functional relation between $\mathbf{x}$ and y.

Let $(\mathbf{X}_t, \mathbf{y}_t)$ be the set of training data and $\mathbf{x}$ be the new datapoint to classify. Given the training data $(\mathbf{X}_t, \mathbf{y}_t)$, the number of neighbors k, the new datapoint $\mathbf{x}$ and the similarity metric K the KNN algorithm is given by

1. Compute $d_t = K(\mathbf{x}, \mathbf{x}_t)$ for each $\mathbf{x}_t \in \mathbf{X}_t$.
2. Find the top k training samples with the highest similarity score d_t.
3. Output $\hat{y}$ as most common class among these points.

The hyperparameters to choose in KNN are the number of neighbors k to be considered during the label voting and how to determine the similarity between a training sample and the new point to classify. While the similarity metric can be any valid metric (see appendix A) it is most commonly taken to be the euclidean distance.

There are no model parameters $\boldsymbol{\theta}$ in KNN, meaning there are no prior training phase before classification. It is however common to arrange the training samples in a data structure such as a K-D tree [59] in order to faster find the k most similar training samples during classification. The naive implementation of KNN takes $O(dn_t)$ to classify each sample since it has to compute the similarity for each training sample, while K-D trees reduces this to $O(d \log n_t)$ [59] by only having to consider $O(\log n_t)$ samples on average. Since all the training samples have to be available during classification, the memory complexity is $O(dn_t)$.

4.2 Support Vector Machines

Support Vector Machines (SVM) [18] are a type of classification algorithm which is common to apply to hyperspectral images. SVMs generally achieve a high classification accuracy even with relative small amounts of training data, and the classification is done at competitive speeds compared with other methods [60]. SVMs find a hyperplane that separates two classes during the learning phase. Classification is then done by computing on which side of the resulting hyperplane the new sample is located. Note that the SVM is therefore a binary classifier, it solves classification problems involving only two classes. Methods for extending binary classifiers to multiclass problems is discussed in section 4.3.

The equation for a d dimensional hyperplane that partitions $\mathcal{R}^d$ in two is given by

$$\mathbf{w}^T\mathbf{x} + b = w_1x_1 + w_2x_2 + \cdots + w_dx_d + b = 0 \tag{59}$$

where $\mathbf{w}$ is the normal vector of the hyperplane and $\mathbf{x}$ is an arbitrary point on the hyperplane. Let $h(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b$. If $\mathbf{x}_0$ is a point in the space above the hyperplane (in the sense that the normal vector $\mathbf{w}$ points into the space) there must exist an $\mathbf{x}$ on the hyperplane such that $\mathbf{x}_0 = \mathbf{x} + k\mathbf{w}$ where $k > 0$. Then

$$\begin{aligned} h(\mathbf{x}_0) &= h(\mathbf{x}) + \mathbf{w}^T(k\mathbf{w}) \\ &= k\mathbf{w}^T\mathbf{w} > 0 \end{aligned} \tag{60}$$

and inversely for any point below the hyperplane. Meaning that for any point the sign of $h(\mathbf{x})$ indicates which side of the hyperplane $\mathbf{x}$ is. Let the two classes in the binary classification problem be encoded by $\{-1, +1\}$ such that $y_i \in \{-1, +1\}$ for each $y_i \in \mathbf{y}$. The SVM classifier is then given by

$$h(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b \tag{61}$$

$$\hat{y} = \begin{cases} +1, & h(x) > 0 \\ -1, & h(x) \leq 0 \end{cases} \tag{62}$$

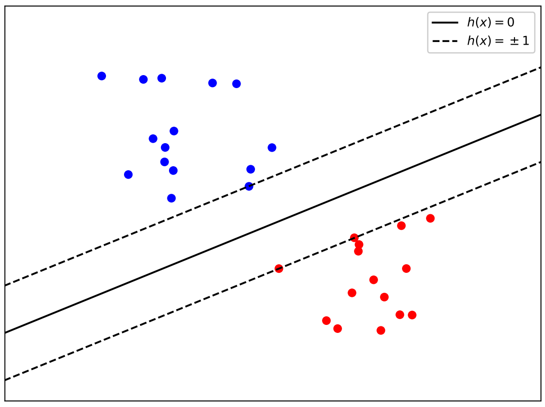


Figure 15: Maximum margin separating hyperplane

The classification problem is transformed into finding a suitable hyperplane that separates the two classes such that the points belonging to different classes is placed on opposite sides. The SVM finds the hyperplane that maximizes the separating margin between the two classes which is the shortest distance between the separating hyperplane and any point. Consider the parallel hyperplanes to $h(\mathbf{x}) = 0$ given by $h(\mathbf{x}) = -1$ and $h(\mathbf{x}) = 1$ such that the planes intersect the closest points as illustrated in fig. 15, which requires that $y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1$ for all training samples $\mathbf{x}_i$. Then maximizing the separation margin is equivalent to maximizing the distance between these two hyperplanes. Let $\mathbf{x}_0$ be a point on $h(\mathbf{x}) = -1$ such that $\mathbf{w}^T\mathbf{x}_0 + b = -1$. The line orthogonal to the plane is then given by $\mathbf{x}_0 + k\mathbf{w}$ for $k \in \mathcal{R}$. At the intersection with $h(\mathbf{x}) = 1$ the value of k is

$$\begin{aligned}
\mathbf{w}^T(\mathbf{x}_0 + k\mathbf{w}) + b &= 1 \\
\mathbf{w}^T\mathbf{x}_0 + b + k\mathbf{w}^T\mathbf{w} &= 1 \\
k\mathbf{w}^T\mathbf{w} &= 2 \\
k &= \frac{2}{\mathbf{w}^T\mathbf{w}} \qquad (63)
\end{aligned}$$

and the length of the line segment between the planes is $||k\mathbf{w}||_2$. Since an orthogonal line is the shortest distance between two parallel planes, the margin is given by the length

$$\begin{aligned}
||k\mathbf{w}||_2 &= k\sqrt{\mathbf{w}^T\mathbf{w}} \\
&= \frac{2}{\sqrt{\mathbf{w}^T\mathbf{w}}} \qquad (64)
\end{aligned}$$

and the margin is therefore maximized by minimizing the quantity $\mathbf{w}^T\mathbf{w}$. The resulting optimization problem is given by

$$\min_{\mathbf{w},b} \frac{1}{2}\mathbf{w}^T\mathbf{w} \qquad (65)$$

$$y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1, \quad i \in \{1, 2, \dots, n_t\} \qquad (66)$$

This is a convex optimization problem belonging to the class of Quadratic Programming (QP) problems for which there exists algorithms to obtain the global minimum directly like active set methods[61]. The training samples in fig. 15 is separable since there exists a hyperplane that can segment the two groups. If this is not the case, then the optimization problem in eq. (65) is infeasible. It is therefore common to relax the requirement of separating the data by introducing a soft margin, originally proposed in [62]. The optimization problem to be solved is then

$$\begin{aligned} \min_{\mathbf{w},b,\boldsymbol{\zeta}} \frac{1}{2}\mathbf{w}^T\mathbf{w}+C\sum_{i=1}^{n_t}\zeta_i \\ y_i(\mathbf{w}^T\mathbf{x}_i+b) \geq 1-\zeta_i, \quad i \in \{1,2,\ldots,n_t\} \\ \zeta_i \geq 0, \quad i \in \{1,2,\ldots,n_t\} \end{aligned} \tag{67}$$

where ζ_i are slack variables and $C > 0$ scales the penalty for violating the margin. Since the separating boundary is still a linear hyperplane, classification is still based on the assumption that the underlying relation between the data vector $\mathbf{x}$ and the corresponding label y_i is indeed linear. To classify non linear data, [63] proposed a nonlinear version of SVM. It introduces a nonlinear transformation ϕ that maps the data into a higher dimensional space in which the data is separated by a hyperplane. When the hyperplane is projected back into the input space it is no longer constrained to a linear boundary. The resulting optimization problem is then

$$\begin{aligned} \min_{\mathbf{w},b,\boldsymbol{\zeta}} \frac{1}{2}\mathbf{w}^T\mathbf{w}+C\sum_{i=1}^{n_t}\zeta_i \\ y_i(\mathbf{w}^T\phi(\mathbf{x}_i)+b) \geq 1-\zeta_i, \quad i \in \{1,2,\ldots,n_t\} \\ \zeta_i \geq 0, \quad i \in \{1,2,\ldots,n_t\} \end{aligned} \tag{68}$$

Note that $\mathbf{w}$ is no longer a d dimensional vector, but rather a vector in the same dimension as $\phi(\mathbf{x})$. If there exists a function K (referred to as a *kernel*) such that $K(\mathbf{x},\mathbf{y}) = \phi(\mathbf{x})^T\phi(\mathbf{y})$ then it is not necessary to compute the projection, but rather evaluate K. This follows from the observation that if K exists then the solution to the optimization problem in eq. (68) is given by [63]

$$\mathbf{w} = \sum_{i=1}^{n_t} \alpha_i y_i \phi(\mathbf{x}_i) \tag{69}$$

where α_i are the solution parameters from the dual form of eq. (68). The classifier h is then

$$\begin{aligned} h(\mathbf{x}) &= \mathbf{w}^T\phi(\mathbf{x})+b \\ &= \sum_{i=1}^{n_t} \alpha_i y_i \phi(\mathbf{x}_i)^T\phi(\mathbf{x})+b \\ &= \sum_{i=1}^{n_t} \alpha_i y_i K(\mathbf{x}_i,\mathbf{x}) \end{aligned} \tag{70}$$

The training samples for which $\alpha_i \neq 0$ is refereed to as the support vectors and $(\alpha y)_i$ as the coefficients for the support vectors. Common choices for the kernel K are [64]

$$K(\mathbf{x},\mathbf{y}) = (1+\mathbf{x}^T\mathbf{y})^d \quad \text{(Polynomial)} \tag{71}$$

$$K(\mathbf{x},\mathbf{y}) = \exp(-\gamma||\mathbf{x}-\mathbf{y}||_2^2) \quad \text{(RBF)} \tag{72}$$

$$K(\mathbf{x},\mathbf{y}) = \tanh(\gamma\mathbf{x}^T\mathbf{y}+r) \quad \text{(Sigmoid)} \tag{73}$$

where γ, d, r are kernel hyperparameters. Let $\mathbf{S} = \{\mathbf{s}_1,\mathbf{s}_2,\ldots,\mathbf{s}_{n_s}\}$ denote the set of n_s support vectors and $\boldsymbol{\alpha}\mathbf{y}$ the set of learned coefficients. The full algorithm for the binary classification SVM is then given by

1. Solve the QP optimization problem in eq. (68) using the training data $(\mathbf{X}_t, \mathbf{y}_t)$ to obtain the support vectors $\mathbf{S}$ and the coefficients $\boldsymbol{\alpha}\mathbf{y}$.
2. Given a new sample $\mathbf{x}$, compute the classifier $h(\mathbf{x})$ using eq. (70).
3. Obtain the class estimate $\hat{y}$ using eq. (62)

The time complexity of the SVM is $O(n_s d)$ for evaluating the kernel between the input sample and each support vector $\mathbf{s}_i$. The memory complexity is driven by storing the support vectors and is therefore $O(n_s d)$.

4.3 Multiclass classification from binary classifiers

There are two general strategies when extending a binary classification algorithm to problems with $k > 2$ classes, *one versus one*(OVO) and *one versus the rest*(OVR)[65]. In an OVO classifier, a binary classifier is trained to separate each pair of classes. For each class pair (i, j) the classifier is trained with the training data where $y \in \{i, j\}$. In total there are trained $k(k-1)/2$ classifiers. With the OVR strategy each classifier is trained to separate one class from the rest, resulting in k classifiers.

For the OVR case, the classifiers naturally assigns a probability or score to each class. Let h_i denote the classifier that is trained to classify class i versus the remaining classes. The final output can be taken as the maximum output over the k classifiers

$$\hat{y} = \operatorname*{argmax}_i h_i(\mathbf{x}) \tag{74}$$

However in the case of OVO, there are multiple strategies for deciding on a final output. Let h_{ij} denote the classifier trained to separate classes i and j. A class i wins the classification against class j if the classifier h_{ij} favors i over j. A proposed method from [40] seen in algorithm 1 is to assign a score to each class, where the score is the based on the sum of the output from each classifier, as well as a bonus score of +1 for each classifier it wins. The final output is taken as the class with the highest score.

Algorithm 1: Multiclass classification using OVO classifiers

```
Input: $h_{ij}(\mathbf{x})$ trained classifiers
$\mathbf{x}$ New datapoint
Output: The estimated label $\hat{y}$ of $\mathbf{x}$
1  $CS_i \leftarrow 0, \quad i \in \{1, 2, \ldots, k\}$
2  $score_i \leftarrow 0, \quad i \in \{1, 2, \ldots, k\}$
3  for $i \in \{1, 2, \ldots, k\}$ do
4      for $j \in \{i+1, i+2, \ldots, k\}$ do
5          $CS_i \leftarrow CS_i + h_{ij}(\mathbf{x})$
6          $CS_j \leftarrow CS_j - h_{ij}(\mathbf{x})$
7          if $h_{ij} > 0$ then
8              $score_i \leftarrow score_i + 1$
9          else
10             $score_j \leftarrow score_j + 1$
11         end
12     end
13 end
14 for $i \in \{1, 2, \ldots, k\}$ do
15     $score_i \leftarrow score_i + \frac{CS_i}{3|CS_i|+1}$
16 end
17 $\hat{y} \leftarrow \operatorname{argmax}_i score_i$
```

As noted in [65], the OVO strategy generally slightly outperforms OVR in terms of accuracy for SVM, and the methods tends to have the same number of support vectors in total.

4.4 Convolutional Neural Networks

Neural networks are a broad set of machine learning models capable of solving various tasks, including classification. Especially a specific architecture called Convolutional Neural Networks (CNNs), originally proposed to recognize handwritten digits [66], has been successful in image classification. Even a simple CNN model was shown to achieve higher accuracy than SVM on popular datasets for hyperspectral image classification [67]. These networks have also been shown to exploit the correlation of spatially adjacent pixels for increasing classification accuracy [8].

The general idea behind a neural network used for classification is to assume some functional relation between y and $\mathbf{x}$ as

$$\mathbf{p} = f(\mathbf{x}, \boldsymbol{\theta}) \tag{75}$$

where $\boldsymbol{\theta}$ is a set of parameters, $\mathbf{p} = \{p_1, p_2, \dots, p_k\}$ and $p_i = P(y = i)$, the estimated probability that $\mathbf{x}$ belongs to class i. Then the final estimated label is taken as

$$\hat{y} = \operatorname*{argmax}_i p_i \tag{76}$$

The functional form of f is build from layers, where each layer f_i applies some nonlinear transform to the output from the previous layer f_{i-1} using a set of parameters $\boldsymbol{\theta}_i$

$$\begin{aligned}
\mathbf{x}_1 &= f_1(\mathbf{x}, \boldsymbol{\theta}_1) \\
\mathbf{x}_2 &= f_2(\mathbf{x}_1, \boldsymbol{\theta}_2) \\
&\dots \\
\mathbf{x}_{n-1} &= f_{n-1}(\mathbf{x}_{n-2}, \boldsymbol{\theta}_{n-1}) \\
f(\mathbf{x}, \boldsymbol{\theta}) &= f_n(\mathbf{x}_{n-1}, \boldsymbol{\theta}_n)
\end{aligned} \tag{77}$$

where $\boldsymbol{\theta} = \{\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \dots, \boldsymbol{\theta}_n\}$. The total number of parameters $|\boldsymbol{\theta}|$ can be on the order 10^7 or more [68]. The large model space makes the neural network capable of approximating highly complex functions, and was shown to be able to approximate any function arbitrarily well even with a simple structure [69].

The network is trained in the learning phase by minimizing a loss function for each training sample on the form $L(\mathbf{p}, y)$ such as the cross entropy loss [70]

$$l(\mathbf{p}, y) = -\log p_y \tag{78}$$

which is 0 if the classifier assigns all the probability to the correct class ($p_y = 1$), and induces an increasing loss the further p_y is from 1. The objective function to minimize is then given as the sum of losses over all the training data

$$L(\boldsymbol{\theta}) = \sum_{i=1}^{n_t} l(f(\mathbf{x}_i, \boldsymbol{\theta}), y_i) \tag{79}$$

In general this nonlinear objective function is not convex, and optimizing it directly like the SVM objective in eq. (68) is therefore not feasible [61]. The parameters $\boldsymbol{\theta}$ are trained to minimize the objective function eq. (79) using gradient descent (see appendix A) like numerical optimization algorithms such as the Adam optimizer [71].

A challenge with neural networks is the computational efficiency. Due to the high number of parameters and matrix multiplications involved in the classifier f, they typically require specialized hardware such as Graphical Processing Units (GPUs) to accelerate the computation.

4.5 Self Organizing Maps for Classification

The Self Organizing Map introduced in section 3.5 has been used for supervised classification for hyperspectral images. A Massive Self Organizing Map (MSOM) was proposed in [72], where SOM maps with $> 100,000$ nodes are trained as described in section 3.5. After training the BMU of each pixel in the training set is calculated. Each node is labeled with the most frequent class among the pixels with that node as the BMU. Classifying new samples is then done by finding the label of the BMU, similar to the KNN classifier from section 4.1 with $k = 1$.

As an alternative [73] proposed training several smaller SOMs in a supervised manner. Their method is to train one SOM for each class using the labeled training data. As a result, the SOM trained on class i would then use the examples $(\mathbf{x}_t, y_t)$ where $y_t = i$ to learn a representation of the conditional distribution $P(\mathbf{x}|y = i)$. The intuitive idea is that how well a new data point belongs to a class i is given by how well it fits in the map trained on known members of class i. Let $\text{BMU}_i(\mathbf{x})$ be the similarity between $\mathbf{x}$ and the best matching unit in the SOM trained on class i. The proposed classifier is then

$$\hat{y} = \underset{i}{\operatorname{argmax}}\, \text{BMU}_i(\mathbf{x}) \tag{80}$$

The estimated label for pixel $\mathbf{x}$ is the class specific SOM where the BMU best represents the candidate pixel.

If z is the number of nodes in each SOM and k is the number of classes, the resulting time complexity to classify a pixel is $O(dkz)$ for computing the BMU for each of the k maps. And the extra memory usage is simply the self organizing maps, resulting in $O(dkz)$ for memory as well.

4.6 Evaluation Metrics

Given an estimated set of class labels $\hat{\mathbf{y}}$ and the ground truth classes $\mathbf{y}$, the overall accuracy is the percentage of correctly classified samples given by

$$\text{OA}(\hat{\mathbf{y}}) = \frac{1}{|\hat{\mathbf{y}}|} \sum_{i=1}^{|\hat{\mathbf{y}}|} \delta(\hat{y}_i, y_i) \tag{81}$$

where δ is the Kronecker delta function.

5 Method and Design

This chapter details the design choices for implementing Clustering and Classification modules on HYPSO, as well as the proposed design for integration of the pipeline framework.

5.1 Computational Limits On-board

5.1.1 Central Processing Unit

The specifications and features of the on-board ARM CPU are described in section 2.2. It is, however, often difficult to assess the computational power of a CPU purely based on specifications such as clock frequency and core count due to the complexity of modern CPUs. A more common method is to run a benchmark workload on the CPU, which in return provides a score which is comparable across systems.

Dhrystone [74] is a computing benchmark that aims to mimic real CPU workloads like function calls, variable assignments, arithmetic operations etc. The ARM Cortex-A9 achieves a Dhrystone score (DMIPS) of 1667.5 [21]. In comparison, when running the benchmark on an Intel i5-6200U which is a 6 year old medium end laptop CPU, the obtained result was a score of 18566. This is about 11x the score of the Cortex-A9. The source code of the Dhrystone Benchmark is listed in appendix B.6. However the workload in the on board processing pipeline will mainly consist of memory I/O and various arithmetic operations on the hyperspectral pixels. To better understand the processing power available, consider the toy problem of computing the vector dot product between each pixel $\mathbf{x}_i$ and some vector $\mathbf{s}$

$$r_i = \mathbf{x}_i^T \mathbf{s} = x_{i1}s_1 + x_{i2}s_2 + \dots x_{id}s_d \tag{82}$$

In C this computation can be done in a loop as

```c
float r = 0;
for (int i = 0; i < BANDS; ++i)
{
    r += x[i] * s[i];
}
```

where `x` is the pixel array $\mathbf{x}_i$ and `s` is an array corresponding to the vector $\mathbf{s}$. The full test program for the dummy example can be seen in appendix B.5. When compiling with optimization GCC (section 2.6.2) is able to generate highly efficient assembly code for the target hardware where the computation inside the loop consists of three SIMD instructions, fully utilizing the NEON cores

```asm
vldmia      r0!, s0      ; Load 4 32 bit float values to s0
vldmia      r1!, s1      ; Load 4 32 bit float values to s1
vmla.f32    s2, s0, s1   ; Multiply the floats in s0 and s1 and add the sum to s2
```

meaning the loop processes 4 values from $\mathbf{x}_i$ and $\mathbf{s}$ at a time. For the standard cube of dimensions $(h, w, d) = (956, 684, 120)$ this corresponds roughly to $120/4 = 30$ loop iterations per pixel, and therefore $956 \times 684 \times 30 = 1.96 \times 10^7$ loop iterations in total. The target hardware takes 1.42 seconds to execute it. The Intel Laptop uses only 0.1 seconds, about 14 times faster, which is similar to the difference in the Dhrystone score.

This is in the best case scenario where the algorithm involves few calculations and the computational features on board is optimally utilized. Most algorithms of interest is considerably more complex and computationally difficult than a simple dot product, and therefore considerably slower. The number of sensor values in the standard cube is on the order of 7.8×10^7, meaning any extra computation for each value induces a considerable workload. Take the on board compression which is based on the CCSDS-123 standard [75]. The software implementation on the target hardware is only able to process about 2.5 frames a second for the standard cube size, which results in about 330 seconds or 6.5 minutes for the whole cube. For a complete orbit, there is allocated power for 380 seconds of image processing (the full HYPSO power budget can be seen in appendix C). Meaning any image processing pipeline should be completed in less than that, and therefore any single module can not exceed this limit. Ideally each module takes considerable less time, i.e. less than half, such that it is possible to apply several modules in a single orbit. As a result, the HYPSO team decided to implemented the compression algorithm in hardware on the on-board FPGA [76] which could process the same cube in about 0.2 seconds.

5.1.2 Memory

The available on board memory is 1 GiB of DDR3 RAM (section 2.3). Of the 1024 MiB available, 512 MiB is reserved for Direct Memory Access for the FPGA. The PetaLinux OS and various other software components will typically use about 100 MiB, so there will be about 400 MiB free for the processing pipeline. When using 4 byte single precision floating point numbers, the standard hyperspectral cube itself is $956 \times 684 \times 120 \times 4 = 299$ MiB. Therefore, any extra memory allocated by the pipeline can at most be on the order of 100 MiB. Moreover, any algorithm that aims to process a standard cube on board will have a hard requirement of only allocating an average of under $100\text{MiB}/(956 \times 684) = 160$ bytes for each pixel in the input cube. This is the theoretical maximum, but due to real world issues such as memory fragmentation [77], the available memory during operations might be even lower. It is therefore ideal that the memory usage is well below 100 MiB to be safe, such that the pipeline is not forced to abort mid run.

The 100 MiB available could potentially be increased to 250 MiB by doing calculations using half-precision (2 byte) floating point numbers. However this comes at the cost of decreased numerical accuracy and a substantial risk of overflows. By the IEEE 754 standard [78], a half-precision float only have 10 bit fractional, giving about 3 decimals in base 10. This is compared to 23 bit fractional for single precision giving about 7 decimals in base 10. Furthermore, half-precision can only store values with absolute value up to 65504 compared to 3.4×10^{38} for single precision. The cube pixel values themselves could have slightly higher values (up to 65536), but most algorithms of interest commonly have the need to compute values that are significantly higher. Resolving this issue would require storing any data that could potentially require numbers above this limit with at least 4 bytes. This induces computational overhead in the form of casting values between various data types and also makes development more error prone.

Max computational time (hard)	380 seconds
Max computational time (ideal)	190 seconds
Max memory usage (hard)	100 MiB
Max memory usage (ideal)	50 MiB

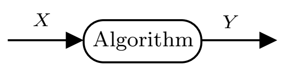
Table 8: Design requirements for HYPSO pipeline modules

The above considerations can be summed up to a list of design requirements in table 8.

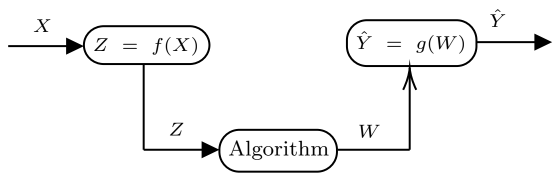
5.2 Intermediate Variables

Intermediate variables can lower the computational demands of an algorithm. Consider the general class of algorithms which take some input X and produce some output Y. To increase the computational efficiency, the following general procedure for obtaining an approximate solution $\hat{Y}$ using intermediate variables is proposed as illustrated in fig. 16.

1. Map the input X to some intermediate variables Z by computing $Z = f(X)$
2. Apply the algorithm to Z and obtain output W
3. Map the intermediate output W to the final output $\hat{Y}$ by computing $\hat{Y} = g(W)$



(a) Original procedure



(b) Modified procedure

Figure 16

If the computational savings of applying the algorithm to Z instead of X is more than the computational cost of mapping X to Z and W to $\hat{Y}$, the overall computational efficiency is increased at the cost of only having the approximate solution $\hat{Y}$ instead of the true solution Y. Note that the computational cost of the mapping involves both finding f and g, as well as evaluating $f(X)$ and $g(W)$. Dimensionality reduction (section 2.7) is an example of such a mapping f in which the number of intermediate variables $\mathbf{Z}$ is the same as the number of input variables $\mathbf{X}$, but the size of each input vector is reduced.

In clustering the input is a set of data vectors $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n\}$, while the output is a vector $\mathbf{y} \in \mathcal{R}^n$ with cluster assignments. As argued in section 5.1, the key computational challenge with on board processing algorithms is that the number of pixels n is typically large. Thus reducing n can have a significant impact on performance optimization, especially for algorithms that scale more with n than d. A natural candidate for the intermediate variable mapping is therefore to reduce the number of data vectors by taking $\mathbf{Z} = \{\mathbf{z}_1, \mathbf{z}_2, \dots, \mathbf{z}_m\}$ where $m \ll n$, with f being a many to one mapping

$$f(\mathbf{x}_i) = \mathbf{z}_j \tag{83}$$

for some $j \in \{1, 2, \dots, m\}$. Let $\mathbf{w} = \{w_1, w_2, \dots, w_m\}$ be the output from the algorithm on the intermediate variables $\mathbf{Z}$. The natural candidate for g is then to assign

$$y_i = w_j \text{ if } f(\mathbf{x}_i) = \mathbf{z}_j. \tag{84}$$

This reduces the problem of finding the mappings g and f to finding a suitable set of intermediate variables $\mathbf{Z}$ and how to determine which $\mathbf{z}_j$ each input vector $\mathbf{x}_i$ should be mapped to. The SOM (section 3.5) provides an intuitive solution to both, where the mapping is the best matching unit (BMU) and the set of intermediate variables is the SOM nodes learned during training.

Using the SOM is fundamentally different from dimensionality reduction, since it does not reduce the number of dimensions d of each data sample, but rather the number of samples n, since multiple samples are represented by the same BMU. It does however achieve a the same goal as reducing the amount of data to be processed typically increases the computational efficiency of an algorithm.

5.3 Clustering

As noted in the toy example of the dot product from section 5.1.1, a simple linear $O(dn)$ algorithm runs at the order of 1 second on the target hardware. This is a challenge for clustering as most algorithms intrinsically requires $O(dn^2)$ operations since each input vector is in some way compared against every other input vector as noted in table 6. In example, Spectral Clustering (section 3.2) requires constructing the affinity matrix, which is precisely a pairwise comparison between each sample. This gives an expected runtime on the order of 10^5 seconds for such algorithms, which will obviously never be feasible within the constraints. Of the presented algorithms in section 3, this includes all except KMeans. This can be partially alleviated with dimensionality reduction such as the PCA algorithm (section 3.8), which will typically speed up an algorithm proportional to the reduction in the number of bands d.

However, most algorithms are also intrinsically linear $O(n)$ or higher in terms of memory as they require storing some information for each pixel. As noted in section 5.1.2, the available memory for each pixel is theoretically at most 160 bytes, meaning any linear memory usage is borderline possible and will depend on the coefficients of the peak memory usage of the algorithm. In order to conservatively estimate the memory usage of each algorithm, consider the memory usage of the major data structures of each algorithm assuming 4 byte floating points and $c = 5$ clusters

- **KMeans** Storing the probability array of choosing each point as the next cluster during initialization gives $4 \times n = 2.6$ MB.
- **FSC** The affinity matrix approximation $\mathbf{Z}$ is $0.1n^2$, which is $4 \times 0.1n^2 = 171$ GB.
- **LSMM-MRF** The largest data structure is the graph used during energy minimization. A conservative estimate is that this graph has $1.5n$ vertices and an average of 3 edges for each vertex. Assuming that a 4 bit integer for each vertex or edge is enough to store all information needed about the graph, the memory usage is $1.5n \times 4 + 1.5n \times 4 \times 3 = 156$ MB.
- **DBSCAN** Assuming the distance matrix only stores the neighboring vertices and not the full $n \times n$ matrix, the memory usage is $4 \times n \times |\mathcal{N}_{\max}| = |\mathcal{N}_{\max}| \times 2.6$ MB. Thus the limit of the maximum number of average neighboring pixels $|\mathcal{N}_{\max}|$ in order to keep the total below 100 MB is about 38. However the actual amount of neighbors is given by the properties of the data and is not something that can be decided beforehand. If the resulting graph from the data does not adhere to this constraint, the algorithm will fail.

Again, the only algorithm well within the constraints is KMeans. Note that the analysis of the memory consumption of the above algorithms does not depend on the number of features, i.e. spectral bands d, meaning dimensionality reduction will not alleviate their challenges apart from reducing the memory usage of the cube itself. To do clustering within the constraints and simultaneously leverage more powerful clustering algorithms such as Spectral Clustering, it is therefore proposed to use the self organizing map (SOM) for clustering by the use of intermediate variables to reduce the number of pixels n to be processed. HYPSO is set to be operational for 5 years, and it therefore reasonable to assume that data from captured images will be available on the ground soon after launch. Assuming that newly captured images follows the same distribution as existing images, the following procedure can be applied

1. Train a self organizing map on the existing data as detailed in section 3.5.
2. Consider the nodes of the SOM as the intermediate variables $\mathbf{Z}$ introduced in section 5.2 and cluster them using any clustering algorithm, leveraging any programming language and machine learning library, to obtain the clustering assignments $\mathbf{w}$.
3. Upload the resulting map nodes $\mathbf{Z}$ with corresponding labels $\mathbf{w}$ to HYPSO.
4. Cluster newly captured images through eq. (84).

This can be thought of as a two stage clustering procedure. First the SOM is used to obtain a rough clustering where each SOM node is its own cluster. It is then clustered again for the final output.

The memory usage of this approach is trivial as if the dimension of the map is 20×20, there are 400 nodes in the map, giving a memory usage of only $4 \times 400 \times 120 = 0.19$ MB. The upper limit on the dimension of the map to stay within the memory constraints is 456 which would use 99.8 MB.

The execution time scales as $O(dnz^2)$ where z is the dimension of the SOM, meaning the number of nodes z^2 will eventually become too large to handle within the computational constraints. The main benefit of the resulting strategy it that it offloads most complex computational work of obtaining the initial rough clustering to the ground. Besides making algorithms computationally feasible, it also enables faster development as each algorithm does not have to be implemented in C.

5.4 Classification

The support vector machine (SVM) with the RBF kernel has been popular for classification of hyperspectral images and is widely used as the benchmark when proposing new algorithms. It typically has good classification performance, is relatively fast and has low memory usage. Even with 10^5 support vectors the memory overhead is only about $4 \times 10^5 \times d = 4.8$ MB for the standard cube with $d = 120$. Moreover, the use of the kernel provides and easy way to modify and experiment with the algorithm without much coding since the rest of the algorithm does not change. Another benefit of the SVM is that it allows direct tuning of the trade off between computational performance and classification accuracy. The driving computational cost is computing the kernel for each support vector, so reducing the number of support vectors will lead to an (approximately) proportional reduction in computational cost, at the potential expense of reduced accuracy. It is therefore chosen as the classification algorithm to implement on HYPSO.

The multiple SOM classifier presented in section 4.5 is very similar to the proposed clustering strategy. They can in fact be made identical except for the learning phase. By combining the nodes of the maps for each class into a single large map as illustrated in fig. 17, the resulting map can be treated as a map for clustering. However the cluster labels are in fact found in a supervised manner using labeled data.

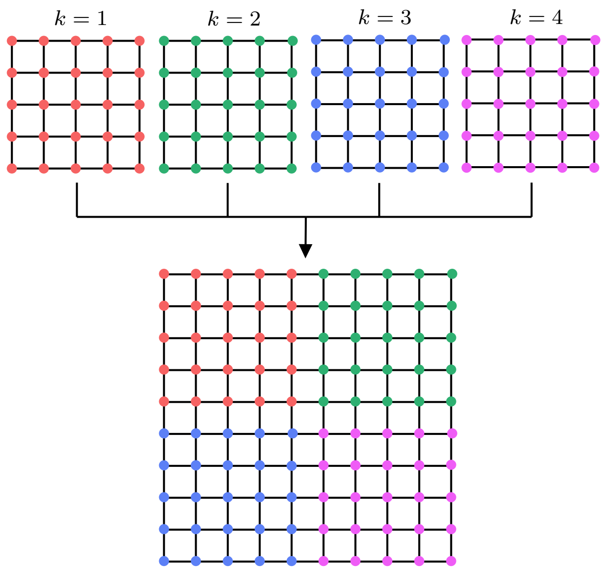


Figure 17: Combining a SOM classifier with $k = 4$ classes to a single map

Thus the code for clustering on HYPSO can be re-used to do classification using this method. It is therefore included for comparison to the SVM and for evaluating how access to the ground truth labels improves the capabilities of the SOM.

During operations, the model will be trained on the ground using available labeled data before the trained parameters is uploaded to HYPSO for classification. This will of course require some labeled data to use for training. Meaning the classification module will either require using labeled data from another source or some initial manual labor of labeling images with classes before the models can be trained.

5.5 Dimensionality Reduction

The dimensionality reduction module will implement the PCA algorithm from section 3.8. Since computing the full symmetric covariance matrix $\Sigma_{\mathbf{X}}$ requires $0.5n^2$ dot products, it is from the toy example in section 5.1.1 expected to take on the order of 3×10^5 seconds, and therefore not feasible to learn the components on board. However by the same argument as for the SOM, if the underlying distribution does not change significantly, the projection can be learned on the ground and uploaded to HYPSO for use on board. This has the added benefit of not having to downlink the projection as side channel data in addition to the dimensionality reduced cube since it will already be known.

In order to speed up classification and clustering, PCA will be used as a preprocessing module. This requires the classification and clustering models to be learned on PCA embedded data. The amount of components m will represent a trade off between keeping information in the data for the algorithms to exploit, and reducing the amount of data to a computationally feasible level.

5.6 Pipeline Integration

Regarding the on board processing pipeline presented in section 2.5, it is proposed to integrate it on HYPSO as a separate separate binary executable. It should then be possible to issue a new command called `hsi pipeline`, which in turn makes the service handler `hsi-service` in `opu-system` apply the pipeline to a captured cube.

There are several benefits of such a modular design.

1. It increases the fault tolerance of the system because in the case that the pipeline should have a flaw, like a module not handling an error correctly or a memory leak, the rest of the system is not affected. This is especially important for a space mission where debugging and redesigning components is costly and time consuming.
2. Updating or patching the pipeline is done by uploading a new executable, it is not necessary to restart the payload system or configure anything else.
3. With separate code bases there is also less code bloating and easier to test and verify individual software components for correctness.

The downsides are duplication of code, and there is also a slight performance overhead of starting a new process compared to including the pipeline directly in the `opu-system` binary.

Figure 18 shows the proposed execution flow when executing the on board pipeline from the ground. When the `hsi pipeline` command is executed the following steps are taken:

1. The ground command line interface tool `hypso-cli` parses the given pipeline arguments. The proposed data fields to construct a valid configuration can be seen in table 9.
2. Next, the pipeline configuration parameters is serialized into a CSP packet and sent to the satellite using the CSP protocol.
3. On board, the service handler `hsi-service` attempts to deserialize the CSP packet and read the specified configuration file. If this is not successful, an appropriate error message is sent back. If it is successful, it will initialize the pipeline data structure `PipelineContext` and populate the data fields.
4. Before starting execution, the validity of the pipeline data structure is checked. This includes checking that numerical fields have sensible values (e.g. the cube dimensions are within reasonable limits), path fields point to files that exists and have necessary read permissions and that each pipeline module entry corresponds to a known module. If any check should fail, the execution is aborted and an error message is sent back with a descriptive error code.
5. To execute the pipeline, the service task uses `fork`, a Linux system call which spawns a new child process by duplicating the current process. The child process then uses `exec`, another Linux system call which replaces the current process with the pipeline executable binary. The parent process on the other hand waits for the child process to terminate, either by a normal system call to `exit` or from receiving a signal from the OS kernel.
6. Upon termination of the child process, the parent process sends the cause of termination and corresponding value (exit code or signal) back to the ground using the CSP protocol.

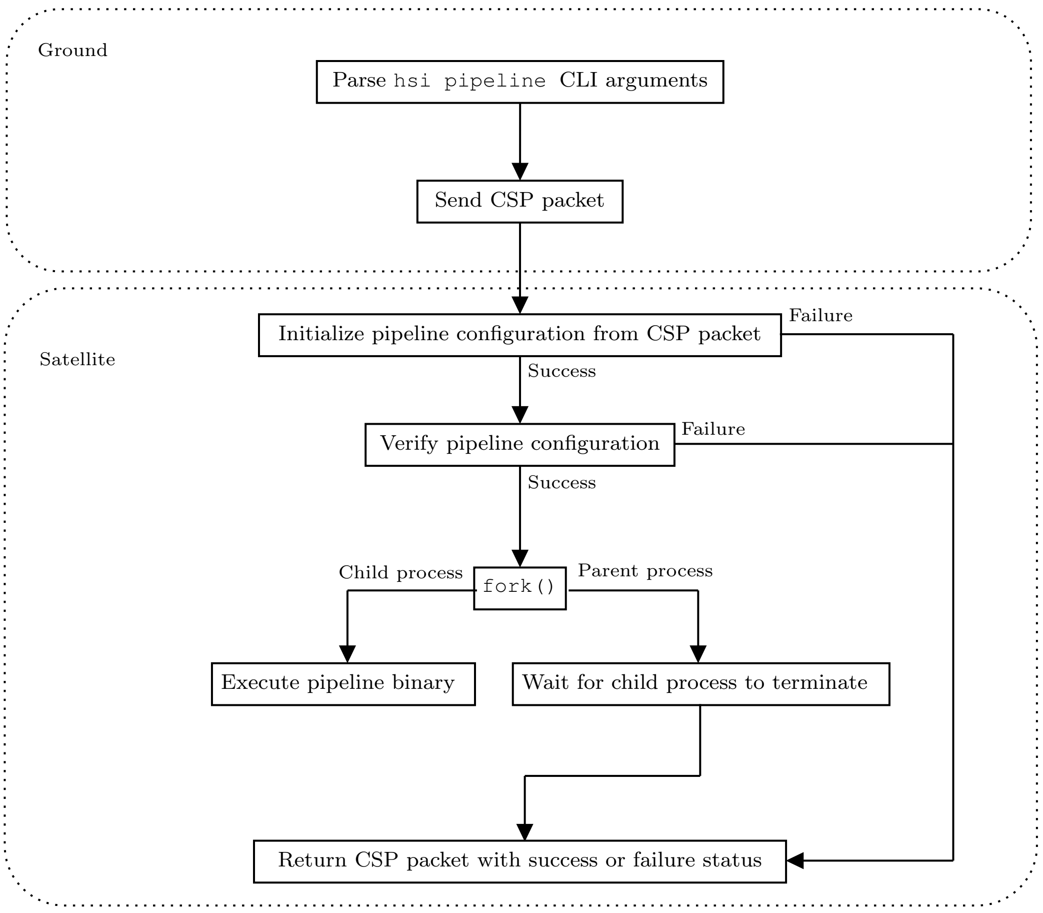

Ground
Parse hsi pipeline CLI arguments
Send CSP packet
Satellite
Initialize pipeline configuration from CSP packet
Failure
Success
Verify pipeline configuration
Failure
Success
Child process
fork()
Parent process
Execute pipeline binary
Wait for child process to terminate
Return CSP packet with success or failure status

Figure 18: Flowchart for issuing a new pipeline execution command

Argument	Expected Value
`-c`	Path to pipeline configuration file on board
`-i`	Path to cube the pipeline should process
`-o`	Path to write the processed cube
`-b`	Number of spectral bands in the input cube
`-y`	Spatial height of the cube (number of frames)
`-x`	Spatial width of the cube (number of rows in each frame)
`-n`	Process the cube in n partitions (optional)

Table 9: Arguments needed to initialize a pipeline execution on the HYPSO satellite

6 Implementation

This chapter will present the implementation of the modules in the pipeline framework. For each module, the configuration parameters and the code for the core execution algorithm will be presented. Utility functionality like allocating and freeing memory, IO and error handling are mostly left out. The full source code can be seen in appendix B.

The three modules (dimensionality reduction, Clustering and Classification) have the same general flow:

1. Parse the module configuration file
2. Allocate memory for the necessary datastructures and populate them with the model binary data from the disk as specified by the configuration file.
3. Apply the algorithm to the input cube.
4. Save any results and free up the allocated memory.

The configuration files consists of (field, value) pairs on the form

```
field = value
```

and is parsed using the Libconfig [79] library. This provides functionality for reading a configuration file and looking up specific fields in the parsed file. Reading an integer value called `example_field` in the configuration file `/configurations/example_config_file.config` can be done as

```c
config_t cfg;
config_init(&cfg);

// Attempt to read configuration file "config_file"
if (!config_read_file(&cfg, "/configurations/example_config_file.config"))
{
    // handle configuration file error
}

int value;

// Attempt to get field "example_field"
if (!config_lookup_int(&cfg, "example_field", &value))
{
    // handle missing field error
}
```

The model binary data files and the configuration files are automatically generated with the Python scripts used for training the models.

6.1 Dimensionality Reduction Module

The configuration file of the dimensionality reduction module specifies the number of components m as well as the location of the binary data for the PCA projection. A valid configuration could be the following:

```
n_components = 10
basis_binary_file = "./pca/binary_10.bin"
```

After parsing the configuration file, the binary file basis_binary_file is read. The standard PCA implementation presented in section 3.8 is to first center the data and then calculate the projection. However observe that

$$\begin{aligned}\mathbf{z} = \mathbf{P}\widetilde{\mathbf{x}} &= \mathbf{P}(\mathbf{x} - \overline{\mathbf{x}}) \\ &= \mathbf{P}\mathbf{x} - \mathbf{P}\overline{\mathbf{x}} \\ &= \mathbf{P}\mathbf{x} - \overline{\mathbf{z}} \end{aligned} \tag{85}$$

so it is possible to first project, then subtract the projected mean. This is computationally cheaper since the dimension of $\mathbf{z}$ is lower than the dimension of $\mathbf{x}$. Therefore, the binary file needs to contain the projected mean $\overline{\mathbf{z}}$ and the $m \times d$ projection matrix $\mathbf{P}$. The first $4 \times m$ bytes of the binary file is $\overline{\mathbf{z}}$ and the remaining $4 \times m \times d$ bytes is $\mathbf{P}$.

The code for the core algorithm can be seen below while the complete source code is listed in appendix B.1. The first step is to allocate new memory for the the dimensionality reduced cube $\mathbf{Z}$ called dr_cube. If the allocation fails, an error code is returned. Next, the GNU Scientific Library (GSL)[80] is used to compute $\mathbf{Z} = \mathbf{PX}$ before the projected mean is subtracted. Finally the old cube $\mathbf{X}$ is deallocated and the cube pointer is updated to the new cube $\mathbf{Z}$.

```c
int PCA_transform(PCA_Context* ctx, float** cube)
{
    // Allocate the new cube Z
    float* dr_cube = malloc(sizeof(float) * ctx->n_pixels * ctx->n_components);

    if (!dr_cube)
    {
        printf("PCA: Could not allocate space for transformed cube\n");
        return -1;
    }

    gsl_matrix_float_view cube_matrix =
        gsl_matrix_float_view_array(*cube, ctx->n_pixels, ctx->n_bands);

    gsl_matrix_float_view dr_matrix =
        gsl_matrix_float_view_array(dr_cube, ctx->n_pixels, ctx->n_components);

    // Calculate Z = PX
    int ret = gsl_blas_sgemm(CblasNoTrans, CblasTrans, 1, &cube_matrix.matrix,
                             ctx->basis, 0, &dr_matrix.matrix);

    if (ret != GSL_SUCCESS)
    {
        printf("Failed to do PCA computation: %d\n", ret);
        free(dr_cube);
        return ret;
    }

    // Subtract projected mean z_i = z_i - z_bar
    for (size_t i = 0; i < ctx->n_pixels; ++i)
    {
        gsl_vector_float_view pix = gsl_matrix_float_row(&dr_matrix.matrix, i);
        gsl_vector_float_sub(&pix.vector, ctx->means);
    }

    // Free the old cube X and update the pointer to the newly constructed Z
    free(*cube);
    *cube = dr_cube;

    return 0;
}
```

6.2 SOM Method

The SOM method uses the following configuration file fields

```
SOM_data_file = "./clustering/som.bin"
results_file = "results.bin"
SOM_dim = 7
```

- The binary file determined by the `SOM_data_file` field contains the SOM nodes and the corresponding labels. The first $4 \times d \times z^2$ bytes are the floating point node values, while the remaining z^2 are the integer labels. Each label is only stored as a single unsigned byte since the number of unique classes or clusters are typically far below 256, which is the maximum number of an unsigned byte.
- `results_file` specifies the file path the cluster assignments or classification labels should be saved to. After successful execution, the module will create a new file and write the output to this file.
- The `SOM_dim` field specifies z^2, the size of the SOM grid

The core execution of the module is to compute the BMU using eq. (41) for each pixel, then cluster or classify it using the label of the BMU by eq. (80). The outermost loop steps through all the pixels in the input cube. For each pixel, it enumerates all the SOM nodes and computes the distance to the node. If this distance is lower than the current lowest distance, it is set as the current lowest distance and the current label estimate is updated. After all nodes have been checked, the label is assigned. The full source code can be seen in appendix B.2.

```
int cls_som_classify(SomClassification* ctx)
{
    // Iterate over all pixels
    for (size_t p = 0; p < ctx->n_pixels; ++p)
    {
        uint8_t bmu_lbl = ctx->SOM_labels[0];
        float min_bmu = FLT_MAX;

        float* pixel = &(ctx->hsi_cube_f[p * ctx->n_bands]);

        // Iterate over all SOM nodes and find best match
        for (size_t i = 0; i < ctx->SOM_dim; ++i)
        {
            float bmu_tot = 0;
            float* som_node = &(ctx->SOM_data[i * ctx->n_bands]);

            // Calculate distance between node and pixel
            for (size_t j = 0; j < ctx->n_bands; ++j)
            {
                bmu_tot += (pixel[j] - som_node[j]) * (pixel[j] - som_node[j]);
            }

            if (bmu_tot < min_bmu)
            {
                min_bmu = bmu_tot;
                bmu_lbl = ctx->SOM_labels[i];
            }
        }

        // Assign the label/cluster of the best matching unit
        ctx->cube_labels[p] = bmu_lbl;
    }

    return 0;
}
```

6.3 Support Vector Machine classifier

A valid configuration file for the SVM module can be

```
SVM_binary_file = "./classification/svm_data.bin"
results_file = "results_svm.bin"
n_classes = 4
gamma = 5.0505050505e-03
n_SV0 = 346
n_SV1 = 53
n_SV2 = 502
n_SV3 = 195
```

- The `SVM_binary_file` is the model file with the trained classifiers. It contains the support vectors $\mathbf{S}$ and coefficients $\boldsymbol{\alpha}\mathbf{y}$ used in the classifier from eq. (70). The first $4 \times d \times n_s$ bytes are the support vectors, where the support vectors arranged in increasing order by the class. The next are the coefficients in the one versus one classifiers. There are $k(k-1)/2$ classifiers, each requiring coefficients for each support vector for a total of $4 \times k(k-1)/2 \times n_s$ bytes, also sorted by class.

  Associated with each class i is the following: A set of n_{si} support vectors $\mathbf{S}_i = \{\mathbf{s}_{i1}, \mathbf{s}_{i2}, \dots, \mathbf{s}_{in_{si}}\}$ and the set of coefficients for the classifier against every other class j: $\boldsymbol{\alpha}\mathbf{y}_{ij} = \{\alpha y_{ij1}, \alpha y_{ij2}, \dots, \alpha y_{ijn_{si}}\}$ for $i \neq j$.
- `results_file` specifies the file path the classification results should be saved to, similar to the SOM method.
- `n_classes` specifies the number of classes k.
- `gamma` gives the γ parameters for the RBF kernel in eq. (72).
- `n_SVi` is n_{si}, the number of support vectors for class i.

Classifying a new sample is done as follows:

First the kernel K is evaluated between the new pixel and the support vectors $\mathbf{S}$. The results are stored in the array `kernel_results` where the i-th entry in the array corresponds to $K(\mathbf{x}, \mathbf{s}_i)$. Precomputing the kernel to a temporary array results in about a 2x speedup against computing them when needed since the kernel computation can be reused in the case that a support vector is used in more than one classifier. Since the support vectors are stored sorted on class, the kernel values for the support vectors of class k starts at the offset $\sum_{i=1}^{k} n_{si}$ from the start at the array, where n_{si} is the number of support vectors for class i.

```
static void precompute_kernel(const SVM_Classification* ctx, const float* x)
{
    size_t idx = 0;
    for (size_t i = 0; i < ctx->n_classes; ++i)
    {
        for (size_t j = 0; j < ctx->n_support[i]; ++j)
        {
            kernel_results[idx] = 0;
            const float* sv = ctx->classifiers[i].support_vectors[j];
            for (size_t n = 0; n < ctx->n_bands; ++n)
            {
                kernel_results[idx] -= (x[n] - sv[n]) * (x[n] - sv[n]);
            }

            kernel_results[idx] = expf(ctx->gamma * kernel_results[idx]);
            ++idx;
        }
    }
}
```

With the kernel K computed, each one versus one classifier h_{ij} is evaluated according to eq. (70) for each (i, j) class pair where $i \neq j$. Evaluating the support vectors of class i against class j is done by finding the location of the precomputed kernel for class i in `kernel_results` by adding the array offset. Then the coefficients $\boldsymbol{\alpha}\mathbf{y}_{ij}$ are located based on class j and the series is computed. Evaluating the support vectors of j against i is equivalent with i and j swapped. The final output $h_{ij}(\mathbf{x})$ is returned.

```
static float ovo_classifier(const float* x, int class_i, int class_j,
                            const SVM_Classification* ctx)
{
    float value = 0;
    for (int k = 0; k < 2; ++k)
    {
        int c1, c2;

        // On first iteration evaluate the support vectors of class i
        if (k == 0)
        {
            c1 = class_i;
            c2 = class_j;
        }

        // Second iteration is support vectors of class j
        else
        {
            c1 = class_j;
            c2 = class_i;
        }

        const SVM_data* c1_data = &(ctx->classifiers[c1]);


        // Find kernel computations for c1
        const float* c1_kernel_results = kernel_results;
        for (int i = 0; i < c1; ++i)
        {
            c1_kernel_results += ctx->n_support[i];
        }

        // Find coefficients for c1 trained against c2
        int c1_coeffs_idx = c1 < c2 ? c2 - 1 : c2;
        const float* c1_coeffs = c1_data->coefficients[c1_coeffs_idx];


        // Evaluate c1 trained against c2
        for (int i = 0; i < c1_data->n_sv; ++i)
        {
            value += c1_coeffs[i] * c1_kernel_results[i];
        }
    }

    return value;
}
```

Lastly the *one versus one* to *one versus all* procedure introduced in algorithm 1 is used to convert the evaluated one versus one classifiers h_{ij} to a final score for each class stored in the array `ovr`. The i-th entry in the array corresponds to the score for class i.

```
static void ovo_to_ovr(float* ovo, float* ovr, int n_classes)
{
    // Reset arrays
    for (int i = 0; i < n_classes; ++i)
    {
        confidence_sums[i] = 0;
        ovr[i] = 0;
    }

    int k = 0;
    for (int i = 0; i < n_classes; ++i)
    {
        for (int j = i + 1; j < n_classes; ++j)
        {
            confidence_sums[i] += ovo[k];
            confidence_sums[j] -= ovo[k];

            ovr[i] += (ovo[k] > 0);
            ovr[j] += (ovo[k] < 0);

            ++k;
        }
    }

    for (int i = 0; i < n_classes; ++i)
    {
        ovr[i] += confidence_sums[i] / (3 * (fabsf(confidence_sums[i]) + 1));
    }
}
```

The final class label is then the argmax of the `ovr` array. After applying the procedure to every pixel in the input cube, the result is saved to the file specified by the `results_file` field. The full source code for the SVM module can be seen in appendix B.3.

6.4 Executing a new pipeline

Summing it up, the following steps is needed for executing a new pipeline consisting of one or more modules

1. Train the module models on the ground. The scripts will automatically output the configuration files and the learned binary model data.
2. Upload the configuration files and the model binary data to HYPSO.
3. Issue the `hsi pipeline` command from the ground, starting the execution flow of the pipeline as described in fig. 18.

Naturally the configuration files and model data can be reused for processing subsequent cubes. If a single pipeline should exceed the computational time allocated for processing, it would have to be separated in two pipelines and applied in steps during several orbits.

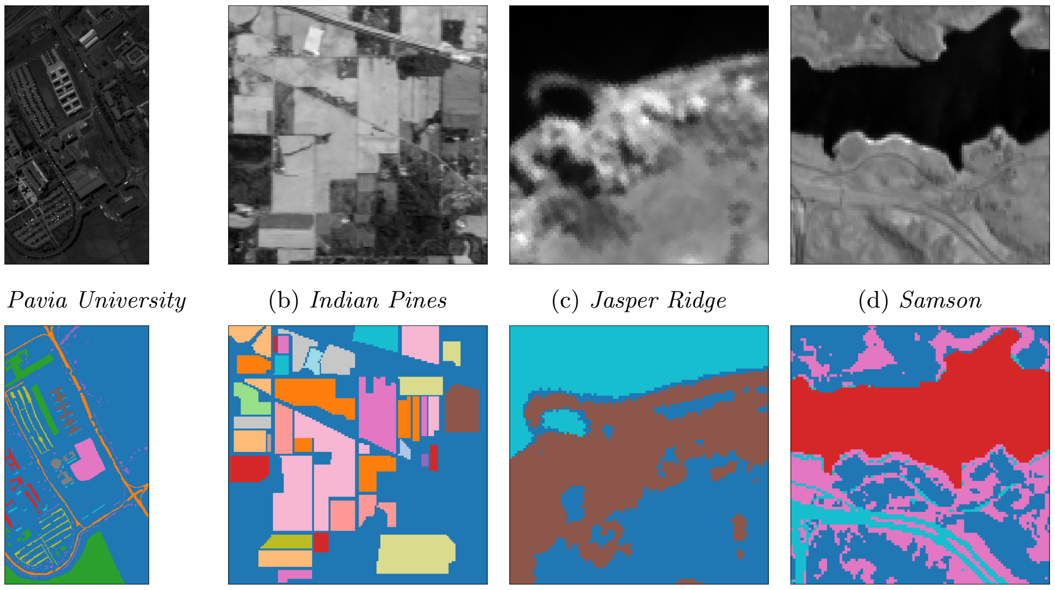
7 Results

The following popular hyperspectral datasets will be used for testing the proposed algorithms

- *Pavia University*: This scene pictures the Pavia University in Italy and was captured by the ROSIS-03 sensor. It is 610 × 304 pixels with 103 spectral bands. In the scene there are 9 targets available for classification given by
  1. Asphalt
  2. Meadows
  3. Gravel
  4. Trees
  5. Painted metal sheets
  6. Bare Soil
  7. Bitumen
  8. Self-Blocking Bricks
  9. Shadows
- *Indian Pines* [81]: This dataset was captured by the AVRIS sensor and is 145 × 145 pixels with 200 spectral bands. The scene pictures farm land in Indiana, USA. There are 16 ground truth classes available for classification given by
  1. Alfalfa
  2. Corn-notill
  3. Corn-mintill
  4. Corn
  5. Grass-pasture
  6. Grass-trees
  7. Grass-pasture-mowed
  8. Hay-windrowed
  9. Oats
  10. Soybean-notill
  11. Soybean-mintill
  12. Soybean-clean
  13. Wheat
  14. Woods
  15. Buildings-Grass-Trees-Drives
  16. Stone-Steel-Towers
- *Samson* [82]: A simple scene from a coastline environment. It is 95 × 95 pixels with 156 spectral bands for each pixel. The 3 targets available are
  1. Soil
  2. Tree
  3. Water
- *Jasper Ridge* [82]: Also captured by the AVRIS sensor, this scene is 100 × 100 pixels with 198 spectral bands. The 4 ground truth classes are
  1. Road
  2. Soil

3. Water
4. Tree

Grayscale image approximations of the datasets and the corresponding ground truth pixel classes can be seen in fig. 19. These hyperspectral datasets presents a varied set of testing data. *Jasper* and *Samson* are both simple scenes with few and distinct classes. The number of samples from each class is approximately balanced. On the other hand, the *Pavia University* and *Indian Pines* sets are larger and more complex scenes. The number of samples available from some of the classes are low, and the classes are partially overlapping such as 5. Grass-pasture and 7. Grass-pasture-mowed in *Indian Pines*.


(a) *Pavia University* (b) *Indian Pines* (c) *Jasper Ridge* (d) *Samson*

(e) *Pavia University* (f) *Indian Pines* (g) *Samson* (h) *Jasper Ridge*

Figure 19: Grayscale images and pixel labels

7.1 Execution time analysis

The computational time of the models scale with the parameters of the model, e.g. the SVM scales with the number of support vectors. In order to know under what parameters the models are computationally tractable within the constraints from table 8, they were tested on the target hardware. Then based on the measurements, runtime models are interpolated. They are interpolated with only the highest order term in the asymptotic runtime to provide a simple and intuitive equation. The runtime analysis is based on the standard cube size of $(956, 684, 120)$.

First consider the PCA dimensionality reduction. As noted in section 3.8 the computational time of projecting a cube is expected to scale linearly with the number of projected components m. Figure 20a shows measurements of the computational time against the number of components, as well as the interpolation. The resulting expected computational time in seconds is

$$E_{\text{PCA}} = 0.8858m \tag{86}$$

The driver of the computational time of the SVM is the RBF kernel calculation which is $O(dn_s)$ where n_s is the number of support vectors. From the measurements and corresponding interpolated model of the expected computational time is

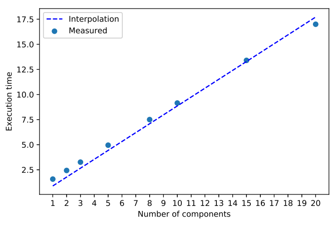
$$E_{\text{SVM}} = 0.0235dn_s \tag{87}$$

fig. 20b show the interpolated model for measurements with $d = 10$ and $d = 120$ bands. The measurements support the linear model for the most part, however there are some deviance for large number of support vectors, probably caused by increased amount of cache misses. For the ideal requirement of < 190 seconds of computation and a cube with full dimension $d = 120$, the upper limit on the number of support vectors is about 70, which is typically far below the number of support vectors in a model. Reducing the dimension to $d = 10$ gives time for about 750 support vectors.

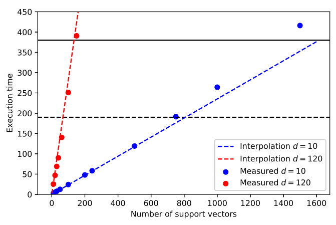
The SOM method has a quadratic runtime in the dimension z and the driving quadratic term also scales with the number of bands d. The interpolated model is as seen in fig. 20c.

$$E_{\text{SOM}} = 0.0104dz^2 \tag{88}$$

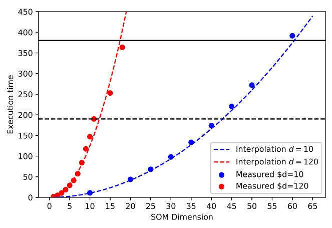
With $d = 120$, the upper limit on z in the ideal case is about 11. Taking $d = 10$ allows for z to be about 40.



(a) Execution time for PCA with $d = 120$ bands



(b) Execution time for SVM



(c) Execution time for the SOM method

Figure 20: Execution time for PCA, SVM and SOM for a standard image with $(h, w) = (956, 684)$. The ideal requirement and hard requirement on computational time are shown with dashed and solid lines respectively.

Comparing with the memory usage analysis from section 5.3 and section 5.4, it is clear that the limiting factor for both methods will be the computational time, and not the memory usage.

7.2 Clustering

The testing images are significantly smaller than the HYPSO cube of $(956, 684, 120)$. Meaning they would allow for a larger SOM map and still be within the HYPSO computational requirements. While this could result in increased clustering performance, the models are restricted to sizes that would be computationally feasible on HYPSO for a realistic evaluation. The SOM sizes and corresponding computational times are therefore reported as if a hyperspectral image of the size of a HYPSO cube was clustered, as opposed to the actual testing images.

Based on the runtime analysis in section 7.1, the SOM is set to have a dimension of $z = 11$ for the full dimensional cube, resulting in about 190 seconds of computational time. To assess the viability of a faster clustering, it will be compared to PCA reduction followed by clustering with $z = 20$ and $d = 10$, which have an combined expected runtime of $9 + 42 = 51$ seconds. Both sets will train the SOM on the testing image, then attempt to cluster the same image. On HYPSO, the training image and image to be clustered will however not be the same. Therefore a third set of models are trained with the full dimension and $z = 11$ where the SOM is trained on half the image, then attempt to cluster the other half. This verifies if the model is able to generalize the learned embedding to new data. The three data sources for training will hereby be refereed to as Full, PCA and Half.

For the SOM clustering method, the hyperparameters that must be chosen are the following

- Influence function and influence radius

- Learning rate function
- Similarity function
- SOM topology

To restrict the number of models under consideration, the model space is limited to the most common found in existing SOM literature. The influence radius and learning rate were both set to the exponential decaying form of eq. (44), with the same time constant. The similarity function is taken to be the euclidean distance, and the SOM topology as a rectangular grid as illustrated in fig. 11. The parameters under consideration are therefore the influence function, as well as the initial values and time constant of the learning rate and influence radius. The candidate values explored are listed in table 10. Each SOM is trained for 50 epochs, at which point the learning rate and update radius are both so low that further training will barely change the map.

Parameter	Values
Influence function (β)	Gaussian (eq. (46)), Bubble (eq. (47))
Learning rate (α_0)	0.05, 0.1, 0.5, 1, 2
Influence radius (σ_0)	2, 3, 4, 7, 10
Time constant (τ)	1, 2, 3, 4, 5, 6, 8, 10

Table 10: Hyperparameters for SOM classification

After training the map, it is clustered using each of the following clustering algorithms

- KMeans (section 3.1)
- Spectral Clustering with RBF affinity matrix (section 3.2)
- Spectral Clustering with nearest neighbor affinity matrix (section 3.2)
- Gaussian Mixture Model (section 3.3.1).

The evaluation metrics used will be the standard set of purity (section 3.7.1), normalized mutual information (section 3.7.2) and overall accuracy (section 4.6). The best results for each algorithm can be seen in table 11, table 12, table 13, table 14 for each of the four training images respectively.

Data	SOM Cluster Algorithm	Purity	NMI	OA
Full	KMeans	0.688	**0.558**	0.588
	SC-RBF	0.661	0.497	0.541
	SC-NN	0.480	0.506	0.434
	GMM	**0.707**	0.539	**0.597**
PCA	KMeans	0.638	**0.560**	0.567
	SC-RBF	**0.968**	0.533	**0.599**
	SC-NN	0.502	0.540	0.462
	GMM	0.667	0.551	0.518
Half	KMeans	0.612	0.546	**0.540**
	SC-RBF	0.614	0.487	0.514
	SC-NN	0.495	0.497	0.439
	GMM	**0.657**	**0.562**	0.535

Table 11: SOM clustering results for *Pavia University*

The results show that the performance are not reduced dramatically when attempting clustering on the other half of the image, indicating that the maps are actually learning some information about the underlying distribution of the pixels. In addition, the combination of PCA reduction and increasing the dimension of the MAP provides competitive results to clustering with the full dimension at only 22% of the computational time. The results for the two simple scenes *Samson* and *Jasper* are significantly better than for *Pavia University* and *Indian Pines*:

Data	SOM Cluster Algorithm	Purity	NMI	OA
Full	KMeans	0.411	0.435	0.366
	SC-RBF	**0.550**	0.380	**0.411**
	SC-NN	0.394	0.419	0.337
	GMM	0.404	**0.439**	0.373
PCA	KMeans	0.383	0.432	0.355
	SC-RBF	**0.497**	**0.432**	**0.385**
	SC-NN	0.347	0.426	0.337
	GMM	0.392	0.427	0.358
Half	KMeans	0.394	**0.432**	0.365
	SC-RBF	**0.570**	0.387	**0.385**
	SC-NN	0.402	0.431	0.355
	GMM	0.461	0.421	0.381

Table 12: SOM clustering results for *Indian Pines*

Data	SOM Cluster Algorithm	Purity	NMI	OA
Full	KMeans	0.867	0.623	0.867
	SC-RBF	0.757	0.502	0.757
	SC-NN	**0.880**	**0.670**	**0.880**
	GMM	0.810	0.536	0.810
PCA	KMeans	0.770	0.433	0.662
	SC-RBF	0.868	0.634	0.868
	SC-NN	**0.874**	**0.653**	**0.874**
	GMM	0.729	0.431	0.729
Half	KMeans	0.794	0.455	0.647
	SC-RBF	0.781	0.557	0.781
	SC-NN	**0.874**	**0.631**	**0.874**
	GMM	0.747	0.514	0.726

Table 13: SOM clustering results for *Samson*

- The clustering problem with more clusters is intrinsically a harder problem since it is less probable to choose the correct cluster.
- Since the SOM dimension is fixed, the average number of SOM nodes for each cluster is lower. Thus there are fewer nodes for learning the underlying cluster distribution.
- The simple scenes have a balanced amount of samples for each class, while the complex scenes do not. As a consequence the SOM is trained more often on samples from some clusters which skews the map.

KMeans on the full image will be used as a benchmark since it is a fast algorithm with low memory usage, making it a realistic candidate to run on HYPSO. The SOM will also be compared to Cluster - Binary Partition Trees (CLUS-BPT), a hyperspectral image classification procedure proposed in [13]. Table 15 shows the best SOM cluster results (for the Full data source) for each image compared against standard KMeans without the SOM and CLUS-BPT. For fair competition the KMeans algorithm was run 10 times with kmeans++ initialization and the best result is reported.

Comparing to CLUS-BPT the SOM is performing competitively on the simpler scenes, but the shortcomings when the number of clusters grows are again apparent. Compared to KMeans, the SOM performs well for all images. While it is difficult to directly compare computational times without controlled tests on identical hardware, the computational times and hardware specifications reported in [13] suggests that CLUS-BPT is 10x - 100x slower.

There is unrealised potential in the SOM. Since each SOM node is assigned one cluster label, it will not be possible to a perfect clustering if two pixels from different classes have the same BMU. The upper theoretical limit of clustering results would be achieved if each SOM node was assigned

Data	SOM Cluster Algorithm	Purity	NMI	OA
Full	KMeans	0.877	0.705	0.877
	SC-RBF	0.899	0.734	0.899
	SC-NN	0.797	0.668	0.797
	GMM	**0.901**	**0.734**	**0.901**
PCA	KMeans	**0.910**	**0.726**	**0.856**
	SC-RBF	0.865	0.681	0.800
	SC-NN	0.797	0.657	0.733
	GMM	0.717	0.620	0.707
Half	KMeans	0.847	0.692	0.847
	SC-RBF	**0.903**	**0.746**	**0.903**
	SC-NN	0.732	0.648	0.732
	GMM	0.770	0.650	0.770

Table 14: SOM clustering results for *Jasper Ridge*

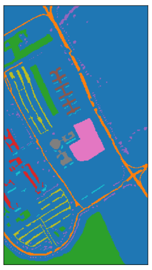
	SOM				KMeans			CLUS-BPT [13]		
Image	SOM alg.	Pur	NMI	OA	Pur	NMI	OA	Pur	NMI	OA
Pavia	GMM	**0.707**	0.539	0.597	0.578	0.537	0.510	0.700	**0.581**	**0.824**
Pines	SC-RBF	0.497	0.432	0.385	0.355	0.343	0.429	**0.576**	**0.603**	**0.578**
Samson	SC-NN	**0.880**	**0.670**	**0.880**	0.847	0.591	0.847	0.690	**0.670**	0.879
Jasper	GMM	**0.901**	**0.734**	**0.901**	0.886	0.712	0.886	0.766	0.566	0.767

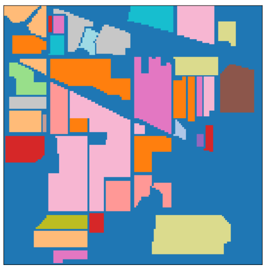
Table 15: Comparison of clustering performance

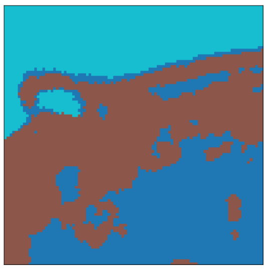
the cluster which maximised the accuracy. This can be thought of as the embedding quality of the SOM, i.e. how well the SOM is able to represent the data. Table 16 shows the clustering results if the maximum of the maps from table 15 is obtained. Comparing with table 15, the SOM image embedding is not fully exploited by the clustering algorithms. The labels of the testing images for the best SOM clustering, KMeans and maximum theoretical clustering is shown in fig. 21.

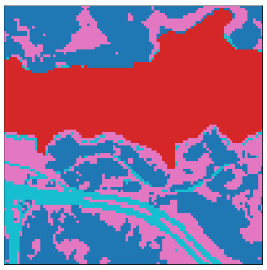
Image	Pur	NMI	OA
Pavia	0.871	0.788	0.689
Pines	0.672	0.541	0.616
Samson	0.962	0.842	0.962
Jasper	0.947	0.832	0.947

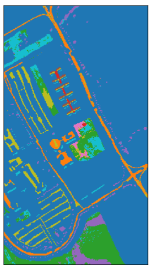
Table 16: Maximum theoretical clustering results for the SOMs in table 15

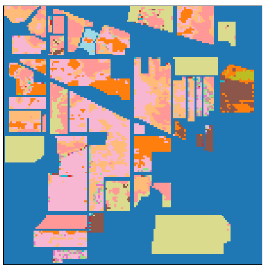

(a) Reality, *Pavia*

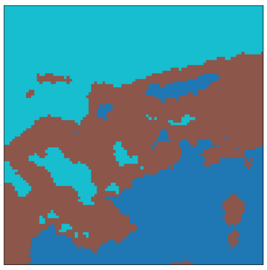

(b) Reality, *Pines*

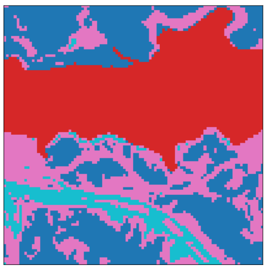

(c) Reality, *Samson*

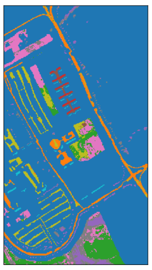

(d) Reality, *Jasper*

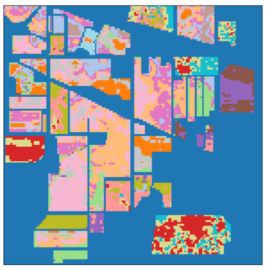

(e) SOM GMM, *Pavia*

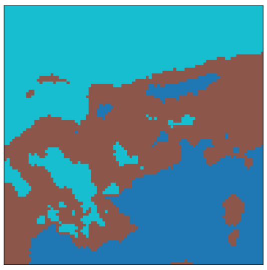

(f) SOM SC-RBF,*Pines*

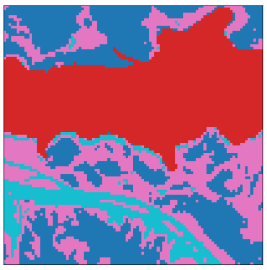

(g) SOM SC-NN,*Samson*

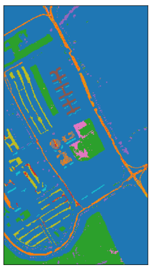

(h) SOM GMM, *Jasper*

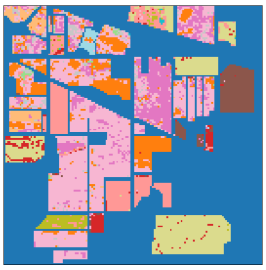

(i) KMeans, *Pavia*

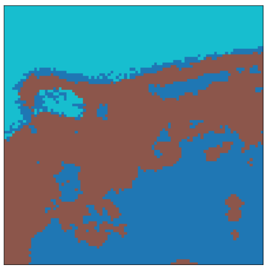

(j) KMeans, *Pines*


(k) KMeans, *Samson*


(l) KMeans, *Jasper*


(m) SOM Max, *Pavia*


(n) SOM Max, *Pines*


(o) SOM Max, *Samson*

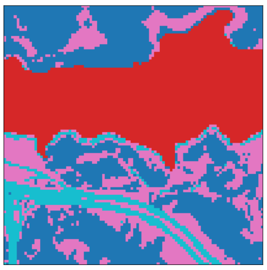

(p) SOM Max, *Jasper*

Figure 21: Clustering results of hyperspectral images for the four testing datasets.

7.3 Classification

The methods are evaluated as follows. In the training phase 10% of the pixels are given as a training set. After the model has been trained, it is tasked to classify the remaining 90% of the pixels as an unseen test set. All models are given the same split between training and testing data and the reported overall accuracy (using eq. (81)) is on the test set.

For obtaining the best results, the set optimal of hyperparameters for each method must be obtained. This was done by searching every combination of the candidate values and using the testing set for measuring accuracy. The hyperparameters for the SOM are the same as in for clustering in table 10. The two hyperparameters to choose for the SVM classifier are the margin violation parameter C and the γ parameter in the RBF kernel. For easier tuning of the γ parameter, it was computed as

$$\gamma = \frac{\gamma_0}{d\text{var}(\mathbf{X}_t)} \tag{89}$$

where d is the number of spectral bands in the image and $\text{var}(\mathbf{X}_t)$ is the variance of the input training data. The values of γ_0 and C explored in the models are given by table 17.

Parameter	Values
C	0.01, 0.1, 1, 10, 100, 500, 1000
γ_0	0.1 0.5, 1, 2, 5

Table 17: Hyperparameters for SVM classification

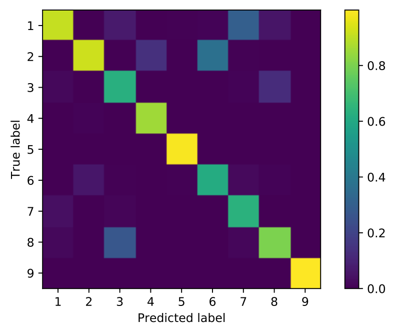
Table 18 shows the best hyperparameters found and the corresponding accuracy for each of the four datasets without any dimensionality reduction. The pixel labels can be seen in fig. 24e-fig. 24l.

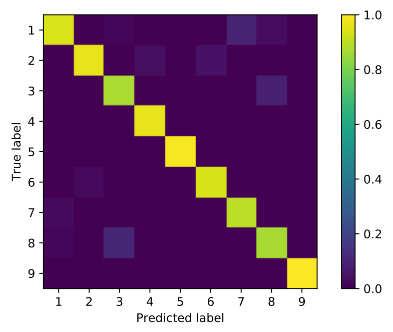
Dataset	SOM					SVM			
	Influence	α_0	σ_0	τ	Accuracy	C	γ_0	n_s	Accuracy
Pavia	gaussian	0.5	3	6	0.862	100	2	1023	0.946
Pines	gaussian	0.5	7	6	0.713	100	1	773	0.819
Samson	bubble	0.5	3	6	0.978	500	0.5	49	0.985
Jasper	bubble	0.5	3	4	0.974	500	0.5	62	0.977

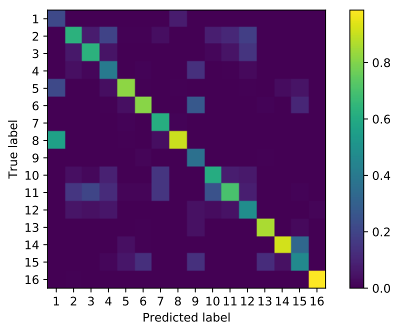
Table 18: Optimal hyperparameters for classification for full cube

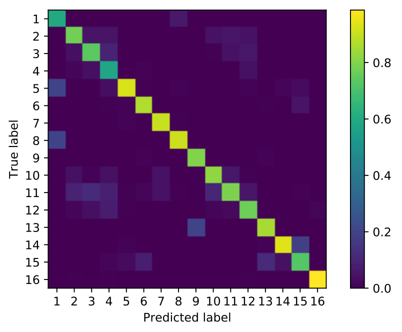
The confusion matrices for the testing datasets and the two classifiers can be seen in fig. 22. The predicted labels are on the x-axis, while the true labels are on the y-axis. The columns of each matrix has been normalized to 1. A perfect classification would therefore be the identity matrix. When comparing the two classifiers, the general trend seems to be that they both behave equally well for each class, relative to the overall performance. In other words, for the classes where the SVM mixes some classes, the SOM mixes the same classes, only slightly more.

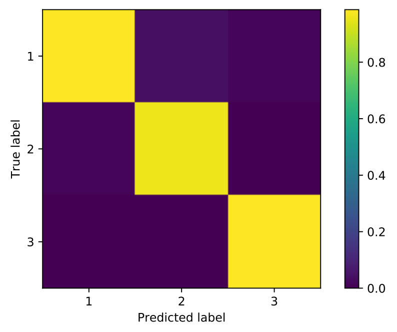
In *Pavia University*, the two most difficult classes to separate are 2. Meadows versus 6. Bare Soil and 1. Asphalt versus 7. Bitumen (a type of hydrocarbons commonly used when constructing asphalt). These classes are similar in material composition, so it is as expected as they are also the most difficult classes to correctly classify. For *Indian Pines* the most difficult targets to classify are also the classes of similar material. Fore example, the classifiers struggle with class 10. Soybean-notill versus class 11. Soybean-mintill, and classes 2,3 and 4 (the three types of corn) also gets mixed up by both classifiers. For both *Jasper Ridge* and *Indian Pines*, the overall performance is good, although class 3. Water sometimes gets incorrectly labeled as class 4. Tree in *Jasper Ridge*.

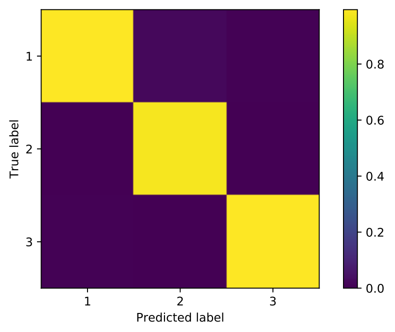

(a) SOM Classifier, *Pavia University*

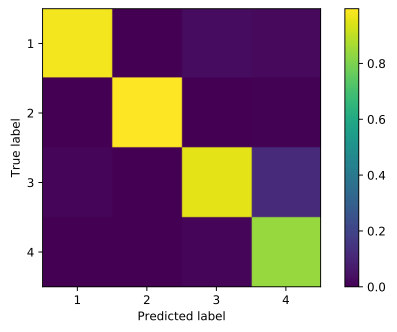

(b) SVM, *Pavia University*


(c) SOM Classifier, *Indian Pines*


(d) SVM, *Indian Pines*


(e) SOM Classifier, *Samson*


(f) SVM, *Samson*


(g) SOM Classifier, *Jasper Ridge*

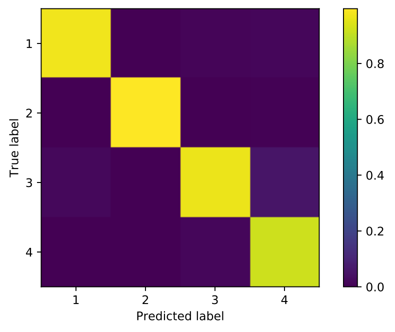
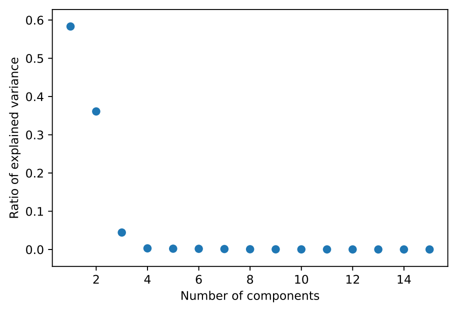
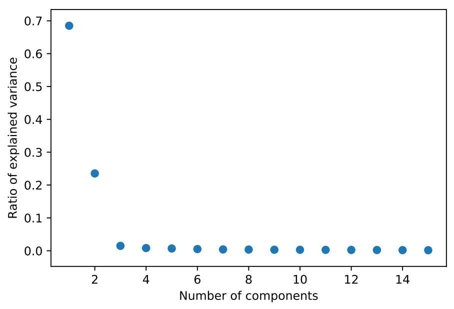

(h) SVM, *Jasper Ridge*

Figure 22: Confusion matrices for the SOM classifier and SVM on the four test datasets

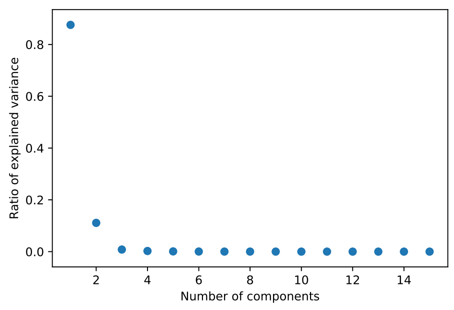
However as noted in section 7.1, the upper limit on the number of support vectors for a full cube is about 75 in the ideal case and about 150 for the hard requirement. Both the complex scenes *Pavia University* and *Indian Pines* greatly exceed this limit. And the self organizing maps are also too large since there are one map for each class. Since the images captured by HYPSO is larger than the testing datasets, it seems reasonable to assume that the models will be at least as large as the models for the test sets. Models of these sizes would not be computationally feasible for a full cube captured by HYPSO.



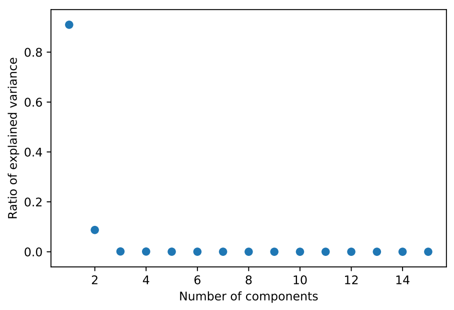
(a) *Pavia University*



(b) *Indian Pines*



(c) *Jasper Ridge*



(d) *Samson*

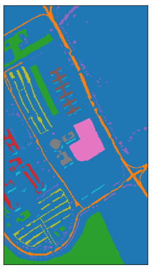
Figure 23: Ratio of explained variance of the first 20 PCA components for the 4 testing datasets

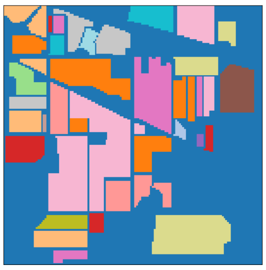
Looking at the explained variance for the PCA components of each datasets as seen in fig. 23 reveals that most of the variance is in the first few components, which suggests that a fairly aggressive PCA dimensionality reduction can be done without losing much accuracy. Taking the number of components to be $m = 6$ which reduces the computational time with a factor of about 25 with the same hyperparameters give the results in table 19 where the estimated computational time is computed using the interpolated models in section 7.1. The pixel labels are shown in fig. 24m-fig. 24t

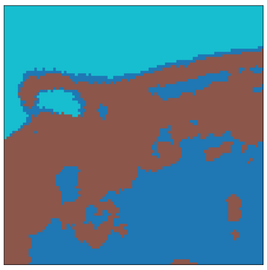
	SOM			SVM			
Dataset	oa	oa change	est. time	oa	oa change	est. time	n_s
Pavia	0.802	-0.060	126	0.878	-0.068	208	1477
Pines	0.638	-0.075	224	0.699	-0.120	100	708
Samson	0.950	-0.028	42	0.980	-0.005	7	55
Jasper	0.942	-0.032	56	0.967	-0.010	13	96

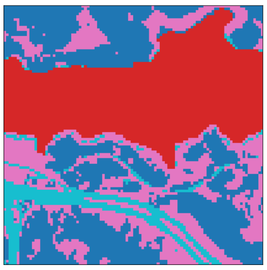
Table 19: Results with PCA dimensionality reduction with $m = 6$ and estimated computational time in seconds

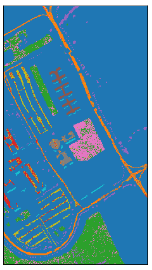
Comparing with table 18 the accuracy is about $2 - 7\%$ lower for all models and datasets. It could be possible that the optimal hyperparameters for the models when using the PCA transformed data is not equal to the optimal parameters for the full datasets. However it will always be the case that exploring more candidate values for hyperparameters results in potentially improved accuracy. The obtained results verifies that a significant PCA reduction can be taken without severe loss off accuracy.

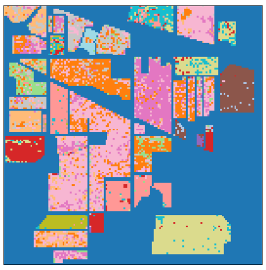

(a) Reality, *Pavia*

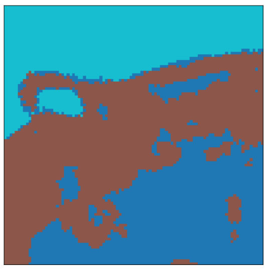

(b) Reality, *Pines*

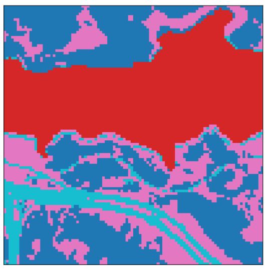

(c) Reality, *Samson*

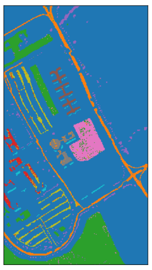

(d) Reality, *Jasper*

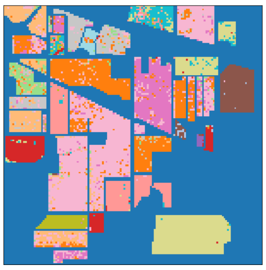

(e) SOM, *Pavia*

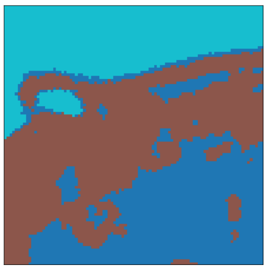

(f) SOM, *Pines*

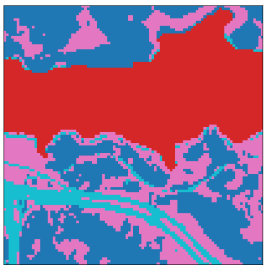

(g) SOM, *Samson*

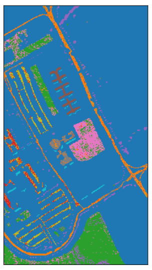

(h) SOM, *Jasper*

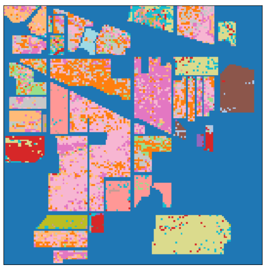

(i) SVM, *Pavia*

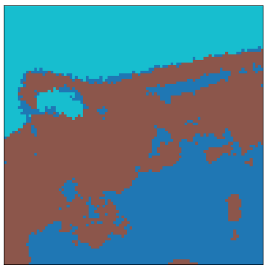

(j) SVM, *Pines*

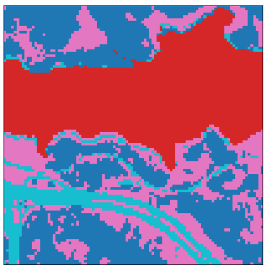

(k) SVM, *Samson*

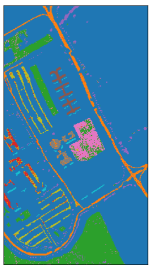

(l) SVM, *Jasper*

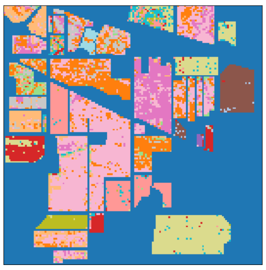

(m) PCA+SOM, *Pavia*

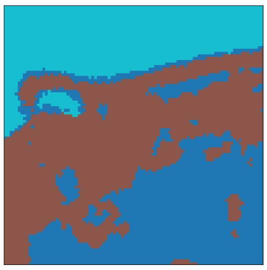

(n) PCA+SOM, *Pines*


(o) PCA+SOM, *Samson*


(p) PCA+SOM, *Jasper*


(q) PCA+SVM, *Pavia*


(r) PCA+SVM, *Pines*


(s) PCA+SVM, *Samson*

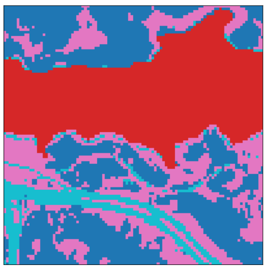

(t) PCA+SVM, *Jasper*

Figure 24: Classification labels for the four testing datasets and SOM and SVM classification algorithms, with and without PCA dimentionality reduction with $m = 6$ components

7.4 Development Methodology

Going into development, a key design goal was designing code with high reliability. If it is going to run on a satellite, it should be trusted to behave as desired. A way to increase reliability is to write unit tests for individual functions and modules, as well as larger integration tests for the pipeline framework. An excerpt of the pipeline tests is in appendix B.4. The standard tests gives valid input to a function where the expected output is known, and verify that the output actually is as expected. Another approach is to give invalid input, like a file path that does not exist, and make sure that the code actually detects that the input is indeed invalid. Especially writing the last class of tests were useful for finding errors in the code, several which could easily have occurred in flight during the HYPSO mission. In addition to testing the developed modules and framework has been analysed using compiler instrumentation and Valgrind, and no dangerous memory operations or memory leaks were found.

It is however difficult to formally verify the correctness of software using testing since it requires exhaustively testing every combination of input. Testing can therefore in practice only verify the presence of bugs, not the absence of them. In an effort to minimize the amount of software bugs, development has generally followed a fail fast principle. Errors, such as failing to allocate sufficient memory or a binary file with model parameters not having the correct size, causes the pipeline to output an error code signaling the source of the error and subsequently crash. This is opposed to attempt error recovery methods such as falling back to default values or repeating the operation. Recovery induces more complexity and software states to the code, increasing the likelihood that some combination of input will make the module behave in an undesirable manner. On the other hand, a crash immediately exposes the problem to the user or developer, effectively creating a faster feedback loop to more quickly identify and fix errors.

8 Conclusion and Future work

This section lists some conclusions and reflections from the work and results presented in this thesis, as well as recommendations for future work.

8.1 Processing Pipeline

The pipeline framework provides a flexible and easy platform for processing hyperspectral images on board HYPSO. With the integration in the on board payload software and ground control system, it is easy to define pipelines for execution in space.

The separation of processing software and payload control software creates a more modular system where the consequences of failures and software errors are reduced. Utilizing tests and profiling tools were key for fixing bugs from early stages of development, as opposed to discovering them during flight in space.

The hardware specifications of the payload SoC on HYPSO does limit the range of algorithms available. HYPSO-2 is expected to have improved hardware, which will improve the viability of further processing modules. However hardware synthesizing will still provide unparalleled computational speeds and power usage and future work should include placing more modules on the on board FPGA.

There are still multiple opportunities for optimizing the software performance of the on board pipeline. Many modules potentially compute similar intermediate values, such as the pixel correlation matrix. Currently it is not easy to reuse such intermediate computations between modules. Furthermore, the on board SoC has two cores which could be utilized to process pixels in parallel, but there is currently no support for multi-threaded operations.

8.2 Clustering

Clustering large hyperspectral images in software with the limited resources available on a small satellite is a challenging problem. An analysis of existing algorithms showed how both memory usage and computation time severely limits the availability of traditional clustering algorithms used for hyperspectral images such as Spectral Clustering. While the number of spectral bands can be reduced fairly efficiently using dimensionality reduction techniques, the large number of pixels n is still an obstacle.

The Self Organizing Map provides a solution to do partial on board clustering. It is computationally efficient, particularly in terms of memory usage since is uses a dense representation of the input image. The current limiting factor is the computational time, which restricts how large the SOM can be.

While the main computational work still have do be done on existing data on the ground, the clustering can be transferred to new images on board. Learning the map on board is currently not feasible. Since it has to be trained on the ground using existing images, it assumes that the pixel distribution of new images does not differ significantly. The results obtained shows that it can generalize to new data, but did not account for images taken under varying weather conditions and similar effects.

The flexibility of the SOM has both advantages and disadvantages: With a rich model space for how the SOM is trained and configured it is able to capture and express a dense representation of the underlying distribution. On the other hand it requires choosing a lot of parameters for optimal results, as opposed to more simplistic algorithms like KMeans or GMM. As a result the SOM models under consideration in this thesis had to be limited to a more narrow set of candidates. However since the training is done on the ground, future operations can test a wider range of SOMs. Furthermore, different SOMs could be used in different circumstances.

SOM clustering shows promising results for simple scenes, but performance is reduced when the number of ground truth clusters grows and sample balance is skewed. This is an inherent weakness of the SOM since the number of nodes for each cluster is proportional to the number of clusters for a SOM of fixed size. Analysing the SOM shows that it is not able to realize the full potential of the embedding during clustering.

PCA is an efficient and powerful method for dimensionality reduction. The reduced number of bands allows for larger maps to be trained and still maintain a significantly lower computational time. This at minimal cost to the clustering results.

Future work in using the SOM for on board clustering could explore other choices of SOM topology, similarity functions, learning rate and influence functions. Ideally a robust set of parameters would be found that worked well for a wide variety of images. An alternative is to find heuristics for automatically selecting parameters based on the input data.

The procedure for clustering a new image using the SOM could also be improved. A natural enhancement is to exploit spatial information from the input image, since the material composition of adjacent pixels tend to strongly correlate.

There might very well be better sources of dense representations and intermediate variables than the SOM for solving the clustering problem. An approach could be to consider it as an optimization problem by explicitly formulating an objective, e.g. minimizing some representation error. Another potential technique is to replace or combine the SOM with density estimation models. Ideally the solution should allow for the computation to be done on board in order to enable full on board clustering.

8.3 Classification

An *one versus one* based SVM classifier was implemented. It provides high classification accuracy with low memory usage. Utilizing a kernel it provides some flexibility in how the classifier behaves. The commonly used RBF kernel was shown to provide good classification results, even for the complex scenes, and outperforming the SOM based classifier.

Using the SVM on HYPSO requires utilizing dimensionality reduction to speed up the computations. PCA provides the necessary reduction in dimensionality in a an efficient manner, at the cost of a small reduction in accuracy.

PCA is a generic dimensionality reduction technique. Future work in improving classification accuracy with the SVM could therefore consider dimensionality reduction methods to produce features geared towards classification. As with clustering, there is also potential for exploiting spatial information in the correlation of adjacent pixels, for example by introducing a prior on the label distribution.

The state of the art models in classification is deep learning bases models such as convolutional neural networks. It is a rapidly developing field, and future work in on board classification should explore the viability of employing light weight neural networks for classification. This would be further enabled by the improved hardware specifications of HYPSO-2.

Bibliography

[1] NTNU AMOS. Research at ntnu amos. https://www.ntnu.edu/amos/research, 2020.

[2] FAO. The State of World Fisheries and Aquaculture 2020. Sustainability in action, 2020.

[3] Elizabeth Middleton, Stephen Ungar, Daniel Mandl, Lawrence Ong, Stuart Frye, Petya Campbell, David Landis, Joseph Young, and Nathan Pollack. The Earth Observing One (EO-1) Satellite Mission: Over a Decade in Space. *IEEE JOURNAL OF SELECTED TOPICS IN APPLIED EARTH OBSERVATIONS AND REMOTE SENSING*, 6:243, 04 2013. doi: 10.1109/JSTARS.2013.2249496.

[4] Michael Soukup, Janis Gailis, Daniele Fantin, Arnoud Jochemsen, Christina Aas, Pieter-Jan Baeck, Iskander Benhadj, Stefan Livens, B. Delauré, Massimo Menenti, Ben Gorte, S.Enayat Hosseini aria, Marco Esposito, and Chris Dijk. Hyperscout: Onboard processing of hyperspectral imaging data on a nanosatellite. In *Small Satellites, System and Services Symposium*, 05 2016.

[5] fisk.no. Stabil sjømateksport i 2020 til tross for koronapandemien. https://fisk.no/fiskeri/7301-stabil-sjomateksport-i-2020-til-tross-for-koronapandemien, 1 2021.

[6] Havforskningsinstituttet. Dette vet vi om den såkalte ≪dødsalgen≫ i Nord-Norge. https://www.hi.no/hi/nyheter/2019/mai/dette-vet-vi-om-den-sakalte-dodsalgen-i-nord-norge, 5 2019.

[7] The Norwegian Directorate of Fisheries. 10 000 tonn død laks til nå. https://www.fiskeridir.no/Akvakultur/Nyheter/2019/0519/10-000-tonn-doed-laks-til-naa, 5 2019.

[8] Ying Li, Haokui Zhang, and Qiang Shen. Spectral–spatial classification of hyperspectral imagery with 3d convolutional neural network. *Remote Sensing*, 9(1), 2017. ISSN 2072-4292. doi: 10.3390/rs9010067. URL https://www.mdpi.com/2072-4292/9/1/67.

[9] Adrián Alcolea, Mercedes E. Paoletti, Juan M. Haut, Javier Resano, and Antonio Plaza. Inference in supervised spectral classifiers for on-board hyperspectral imaging: An overview. *Remote Sensing*, 12(3), 2020. ISSN 2072-4292. doi: 10.3390/rs12030534. URL https://www.mdpi.com/2072-4292/12/3/534.

[10] D. R. Thompson, I. Leifer, H. Bovensmann, M. Eastwood, M. Fladeland, C. Frankenberg, K. Gerilowski, R. O. Green, S. Kratwurst, T. Krings, B. Luna, and A. K. Thorpe. Real-time remote detection and measurement for airborne imaging spectroscopy: a case study with methane. *Atmospheric Measurement Techniques*, 8(10):4383–4397, 2015. doi: 10.5194/amt-8-4383-2015. URL https://amt.copernicus.org/articles/8/4383/2015/.

[11] Yuan Fang, Linlin Xu, Junhuan Peng, Honglei Yang, Alexander Wong, and David A. Clausi. Unsupervised bayesian classification of a hyperspectral image based on the spectral mixture model and markov random field. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 11(9):3325–3337, 2018. doi: 10.1109/JSTARS.2018.2858008.

[12] S. P. Lloyd. Least squares quantization in PCM. *IEEE Trans. Inf. Theory*, 28:129–136, 1982.

[13] Mohamed Ismail and Milica Orlandic. Segment-based clustering of hyperspectral images using tree-based data partitioning structures. *Algorithms*, 13, 12 2020. doi: 10.3390/a13120330.

[14] Sen Jia, Zhen Ji, Yuntao Qian, and Linlin Shen. Unsupervised band selection for hyperspectral imagery classification without manual band removal. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 5(2):531–543, 2012. doi: 10.1109/JSTARS.2012.2187434.

[15] K. Karhunen and Selin. I. *On Linear Methods in Probability Theory.* RAND, 1960.

[16] Ian Blanes and Joan Serra-Sagristà. Pairwise orthogonal transform for spectral image coding. *IEEE Transactions on Geoscience and Remote Sensing*, 49(3):961–972, 2011. doi: 10.1109/TGRS.2010.2071880.

[17] T. Kohonen. The self-organizing map. *Proceedings of the IEEE*, 78(9):1464–1480, 1990. doi: 10.1109/5.58325.

[18] Vladimir Vapnik. *Estimation of Dependences Based on Empirical Data: Springer Series in Statistics (Springer Series in Statistics)*. Springer-Verlag, Berlin, Heidelberg, 1982. ISBN 0387907335.

[19] Jonathon Shlens. A tutorial on principal component analysis, 2014.

[20] E. Prentice, M. Grøtte, F. Sigernes, and T. A. Johansen. Design of hyperspectral imager using COTS optics for small satellite applications. In *Int. Conf. Space Optics (ICSO)*, 2021.

[21] Xilinx. Zynq-7000 SoC, 2018. URL https://www.xilinx.com/support/documentation/selection-guides/zynq-7000-product-selection-guide.pdf.

[22] Teodor Neagoe, Ernest Karjala, and Logica Banica. Why ARM processors are the best choice for embedded low-power applications? In *2010 IEEE 16th International Symposium for Design and Technology in Electronic Packaging (SIITME)*, pages 253–258, 2010. doi: 10.1109/SIITME.2010.5650194.

[23] ARM. Neon, 2020. URL https://developer.arm.com/architectures/instruction-sets/simd-isas/neon.

[24] Ewout ter Hoeven. How Arm's NEON assembly enables efficient AV1 decoding on mobile , 2019. URL https://medium.com/@ewoutterhoeven/how-arms-neon-enables-efficient-av1-decoding-on-mobile-5fcb3a4f6e7f.

[25] Avnet. PicoZed 7030 SoM, 2021. URL https://www.avnet.com/shop/us/products/avnet-engineering-services/aes-z7pz-7z030-som-i-g-rev-e-3074457345635303359/.

[26] Xilinx. Petalinux tools, 2020. URL https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html.

[27] Libscp. Cubesat Space Protocol - A small network-layer delivery protocol designed for Cubesats. https://github.com/libcsp/libcsp, 2021.

[28] M. B. Henriksen, J. L. Garrett, E. F. Prentice, A. Stahl, T. A. Johansen, and F. Sigernes. Real-time corrections for a low-cost hyperspectral instrument. In *2019 10th Workshop on Hyperspectral Imaging and Signal Processing: Evolution in Remote Sensing (WHISPERS)*, pages 1–5, 2019. doi: 10.1109/WHISPERS.2019.8921350.

[29] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

[30] Free Software Foundation, Inc. GCC, the GNU Compiler Collection, 2021. URL https://gcc.gnu.org/.

[31] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Conference on Programming Language Design and Implementation*, 2007.

[32] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997. ISBN 978-0-07-042807-2.

[33] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[34] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2001.

[35] Saima Bano and Naeem Khan. A survey of data clustering methods. *International Journal of Advanced Science and Technology*, 113, 04 2018. doi: 10.14257/ijast.2018.113.14.

[36] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, page 226–231. AAAI Press, 1996.

[37] Sameer Ranjan, Deepak Ranjan Nayak, Kallepalli Satish Kumar, Ratnakar Dash, and Banshidhar Majhi. Hyperspectral image classification: A k-means clustering based approach. In *2017 4th International Conference on Advanced Computing and Communication Systems (ICACCS)*, pages 1–7, 2017. doi: 10.1109/ICACCS.2017.8014707.

[38] Meena Mahajan, Prajakta Nimbhorkar, and Kasturi Varadarajan. The planar k-means problem is np-hard. *Theoretical Computer Science*, pages 274–285, 02 2009. doi: 10.1007/978-3-642-00202-1_24.

[39] David Arthur and Sergei Vassilvitskii. K-means++: The advantages of careful seeding. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '07, page 1027–1035, USA, 2007. Society for Industrial and Applied Mathematics. ISBN 9780898716245.

[40] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[41] Andrew Ng, Michael Jordan, and Yair Weiss. On spectral clustering: Analysis and an algorithm. In T. Dietterich, S. Becker, and Z. Ghahramani, editors, *Advances in Neural Information Processing Systems*, volume 14. MIT Press, 2002. URL https://proceedings.neurips.cc/paper/2001/file/801272ee79cfde7fa5960571fee36b9b-Paper.pdf.

[42] Francis Bach and Michael Jordan. Learning spectral clustering. *In Proceedings of 17th Advances in Neural Information Processing Systems*, 16, 03 2004.

[43] Hsin-Chien Huang, Yung-Yu Chuang, and Chu-Song Chen. Affinity aggregation for spectral clustering. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 773–780, 2012. doi: 10.1109/CVPR.2012.6247748.

[44] Rong Wang, Feiping Nie, and Weizhong Yu. Fast spectral clustering with anchor graph for large hyperspectral images. *IEEE Geoscience and Remote Sensing Letters*, 14(11):2003–2007, 2017. doi: 10.1109/LGRS.2017.2746625.

[45] Yang Zhao, Yuan Yuan, and Qi Wang. Fast spectral clustering for unsupervised hyperspectral image classification. *Remote Sensing*, 11(4), 2019. ISSN 2072-4292. doi: 10.3390/rs11040399. URL https://www.mdpi.com/2072-4292/11/4/399.

[46] Ali Caner Türkmen. A review of nonnegative matrix factorization methods for clustering, 2015.

[47] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society: Series B*, 39:1–38, 1977. URL http://web.mit.edu/6.435/www/Dempster77.pdf.

[48] C. F. Jeff Wu. On the convergence properties of the em algorithm. *The Annals of Statistics*, 11(1):95–103, 1983. ISSN 00905364. URL http://www.jstor.org/stable/2240463.

[49] Y. Boykov, O. Veksler, and R. Zabih. Fast approximate energy minimization via graph cuts. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23(11):1222–1239, 2001. doi: 10.1109/34.969114.

[50] Linlin xu, Alexander Wong, Fan Li, and David Clausi. Extraction of endmembers from hyperspectral images using a weighted fuzzy purified-means clustering model. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 9:1–13, 08 2015. doi: 10.1109/JSTARS.2015.2450499.

[51] C. Lawson and R. Hanson. *Solving least squares problems*. SIAM, 1995.

[52] Rasmus Bro and Sijmen Jong. A fast non-negativity-constrained least squares algorithm. *Journal of Chemometrics*, 11:393–401, 09 1997. doi: 10.1002/(SICI)1099-128X(199709/10)11:53.0.CO;2-L.

[53] Amin Karami and Ronnie Johansson. Choosing dbscan parameters automatically using differential evolution. *Int. J. Comput. Appl.*, 91, 03 2014. doi: 10.5120/15890-5059.

[54] Junhao Gan and Y. Tao. Dbscan revisited: Mis-claim, un-fixability, and approximation. *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015.

[55] Ole Ekseth and Svein-Olaf Hvasshovd. How an optimized dbscan implementation reduces execution time and memory requirements for large data sets. In *Patterns 2019, IARIA*, 02 2018.

[56] Sebastian Ruder. An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747, 2016. URL http://arxiv.org/abs/1609.04747.

[57] L. Danon, A. Díaz-Guilera, J. Duch, and A. Arenas. Comparing community structure identification. *Journal of Statistical Mechanics: Theory and Experiment*, 2005:P09008, 2005.

[58] Max Bramer. *Principles of Data Mining*. Springer, 01 2007. ISBN 978-1-84628-765-7. doi: 10.1007/978-1-84628-766-4.

[59] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975. ISSN 0001-0782. doi: 10.1145/361002.361007. URL https://doi.org/10.1145/361002.361007.

[60] F. Melgani and L. Bruzzone. Classification of hyperspectral remote sensing images with support vector machines. *IEEE Transactions on Geoscience and Remote Sensing*, 42(8): 1778–1790, 2004. doi: 10.1109/TGRS.2004.831865.

[61] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer, New York, NY, USA, second edition, 2006.

[62] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Mach. Learn.*, 20(3):273–297, September 1995. ISSN 0885-6125. doi: 10.1023/A:1022627411411. URL https://doi.org/10.1023/A:1022627411411.

[63] Bernhard E. Boser, Isabelle M. Guyon, and Vladimir N. Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, COLT '92, page 144–152, New York, NY, USA, 1992. Association for Computing Machinery. ISBN 089791497X. doi: 10.1145/130385.130401. URL https://doi.org/10.1145/130385.130401.

[64] Arti Patle and Deepak Singh Chouhan. Svm kernel functions for classification. In *2013 International Conference on Advances in Technology and Engineering (ICATE)*, pages 1–9, 2013. doi: 10.1109/ICAdTE.2013.6524743.

[65] C. Hsu and C. Lin. A comparison of methods for multiclass support vector machines. *IEEE transactions on neural networks*, 13 2:415–25, 2002.

[66] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551, 1989. doi: 10.1162/neco.1989.1.4.541.

[67] Wei Hu, Yangyu Huang, Li Wei, Fan Zhang, and Hengchao Li. Deep convolutional neural networks for hyperspectral image classification. *Journal of Sensors*, 2015:1–12, 07 2015. doi: 10.1155/2015/258619.

[68] Asifullah Khan, Anabia Sohail, Umme Zahoora, and Aqsa Saeed Qureshi. A survey of the recent architectures of deep convolutional neural networks. *Artificial Intelligence Review*, 53 (8):5455–5516, Apr 2020. ISSN 1573-7462. doi: 10.1007/s10462-020-09825-6. URL http://dx.doi.org/10.1007/s10462-020-09825-6.

[69] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2:303–314, 1989.

[70] Hichame Yessou, Gencer Sumbul, and Begüm Demir. A comparative study of deep learning loss functions for multi-label remote sensing image classification, 2020.

[71] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[72] Michael Wong, Wajira Abeysinghe, and Chih-Cheng Hung. A massive self-organizing map for hyperspectral image classification. In *IEEE Whispers*, 09 2019. doi: 10.1109/WHISPERS.2019.8921093.

[73] May Stéphane. Classification of hyperspectral images with self organizing map. In *2013 5th Workshop on Hyperspectral Image and Signal Processing: Evolution in Remote Sensing (WHISPERS)*, pages 1–4, 2013. doi: 10.1109/WHISPERS.2013.8080616.

[74] Reinhold P. Weicker. Dhrystone benchmark, 1988. URL https://www.netlib.org/benchmark/dhry-c.

[75] Consultative Committee for Space Data Systems. Lossless multispectral and hyperspectral image compression-ccsds 123.0-b-1. *Blue Book*, 2012.

[76] Milica Orlandić, Johan Fjeldtvedt, and Tor Arne Johansen. A parallel fpga implementation of the ccsds-123 compression algorithm. *Remote Sensing*, 11(6), 2019. ISSN 2072-4292. doi: 10.3390/rs11060673. URL https://www.mdpi.com/2072-4292/11/6/673.

[77] Nikola Zlatanov. Dynamic memory allocation and fragmentation, 07 2015.

[78] Institute of Electrical and Electronics Engineers. *IEEE standard for binary floating-point arithmetic*. Institute of Electrical and Electronics Engineers, New York, 1985. Note: Standard 754–1985.

[79] Mark Lindner. libconfig - C/C++ library for processing configuration files. https://hyperrealm.github.io/libconfig/, 2021.

[80] GNU Project. GSL - GNU Scientific Library. https://www.gnu.org/software/gsl/, 2021.

[81] Marion F. Baumgardner, Larry L. Biehl, and David A. Landgrebe. 220 band aviris hyperspectral image data set: June 12, 1992 indian pine test site 3, Sep 2015. URL https://purr.purdue.edu/publications/1947/1.

[82] Feiyun Zhu, Ying Wang, Bin Fan, Gaofeng Meng, and Chunhong Pan. Effective spectral unmixing via robust representation and learning-based sparsity. *CoRR*, abs/1409.0685, 2014. URL http://arxiv.org/abs/1409.0685.

Appendix

A Mathematical Background

Metric

A metric is defined as a function $k : \mathcal{R}^n \times \mathcal{R}^n \longrightarrow [0, \infty)$ for which the following properties hold

1. $k(\mathbf{x}, \mathbf{y}) = 0 \Longleftrightarrow \mathbf{x} = \mathbf{y}$
2. $k(\mathbf{x}, \mathbf{y}) = k(\mathbf{y}, \mathbf{x})$
3. $k(\mathbf{x}, \mathbf{y}) \leq k(\mathbf{x}, \mathbf{z}) + k(\mathbf{z}, \mathbf{y})$

L_p Norm

The L_p norm of a vector $\mathbf{x} \in \mathcal{R}^n$ is defined as

$$||\mathbf{x}||_p = (|x_1|^p + |x_2|^p + \cdots + |x_n|^p)^{\frac{1}{p}}$$

Manhattan Distance

The Manhattan distance between two vectors $\mathbf{x}$ and $\mathbf{y}$ is defined as $||\mathbf{x} - \mathbf{y}||_1$, i.e. the L_1 norm of $\mathbf{x} - \mathbf{y}$.

Euclidean Distance

The euclidean distance between two vectors $\mathbf{x}$ and $\mathbf{y}$ is defined as $||\mathbf{x} - \mathbf{y}||_2$, i.e. the L_2 norm of $\mathbf{x} - \mathbf{y}$.

Cosine Similarity

The cosine similarity between two vectors $\mathbf{x}$ and $\mathbf{y}$ is defined as

$$\frac{\mathbf{x}\mathbf{y}^T}{||\mathbf{x}||_2||\mathbf{y}||_2}$$

Radial Basis Function Kernel

The radial basis function (RBF) kernel between two vectors $\mathbf{x}$ and $\mathbf{y}$ is defined as

$$\exp(-\gamma||\mathbf{x} - \mathbf{y}||_2)$$

for some positive real number γ. Is is also equivalently defined with $2\sigma^2 = \gamma^{-1}$ where σ^2 is referred to as the bandwidth or variance.

Affinity Matrix

An affinity matrix $\mathbf{A} \in \mathcal{R}^{n \times n}$ for a set of vectors $\mathbf{X} = \{\mathbf{x_1}, \mathbf{x_2} \ldots \mathbf{x_n}\}$ is defined as

$$A_{ij} = k(\mathbf{x_i}, \mathbf{x_j})$$

where k is a similarity measure such as the RBF kernel.

Degree Matrix

A degree matrix $\mathbf{D} \in \mathcal{R}^{n \times n}$ for the matrix $\mathbf{X} \in \mathcal{R}^{n \times n}$ is a diagonal matrix defined as

$$\mathbf{D}_{ii} = \sum_{j=1}^{n} \mathbf{X}_{ij}$$

Laplacian Matrix

The Laplacian matrix $\mathbf{L} \in \mathcal{R}^{n \times n}$ for the matrix $\mathbf{X} \in \mathcal{R}^{n \times n}$ is

$$\mathbf{L} = \mathbf{D} - \mathbf{A}$$

where $\mathbf{A}$ is a similarity matrix of $\mathbf{X}$ and $\mathbf{D}$ is the degree matrix of $\mathbf{A}$.

Likelihood Function

Assume that a set of random variables $X = \{x_1, x_2, \ldots, x_n\}$ is drawn from a probability distribution for which probability density function is parameterized by a set of variables $\theta = \{\theta_1, \theta_2, \ldots, \theta_k\}$. The likelihood function L is then given as the joint probability density f over X

$$L(\theta) = p(X|\theta) = f(x_1, x_2, \ldots, x_n; \theta)$$

Maximum Likelihood Estimator

The Maximum Likelihood Estimator (MLE) aims to find the parameter values θ that will maximize the probability of observing the data X. It given by

$$\text{MLE}(X) = \underset{\theta}{\text{argmax}}\, L(\theta)$$

where L is the likelihood function.

Maximum A-posteriori Estimator

The Maximum A-posteriori Estimator (MAP) considers the parameter values θ as random variables of their own, and aims to find the value of θ that is most probable given the observed data X. Let g be the prior distribution over θ, i.e. the marginal distribution. Then the MAP estimator is given by

$$\text{MAP}(x) = \underset{\theta}{\operatorname{argmax}}\, p(\theta|X) = \underset{\theta}{\operatorname{argmax}}\, L(\theta)g(\theta)$$

where the last equality comes from Bayes Theorem.

Gradient Descent

Gradient Descent is a method for numerical minimization of some objective function $f(\mathbf{x})$ [61]. For non convex functions, it steps iteratively in the direction of steepest descent towards a local minimum by the update rule

$$\mathbf{x} \leftarrow \mathbf{x} - \alpha \nabla_x f(\mathbf{x}) \tag{90}$$

where $\alpha > 0$ is a parameter regulating the step size.

B Code

B.1 Principle Component Analysis

pca.h

```c
#ifndef PCA_H_INCLUDED
#define PCA_H_INCLUDED
#include "gsl/gsl_matrix.h"
#include <stddef.h>

typedef struct
{
    size_t n_components;
    size_t n_bands;
    size_t n_pixels;

    gsl_matrix_float* basis;
    gsl_vector_float* means;

    char* binary_file;

} PCA_Context;

int PCA_Context_init(PCA_Context* ctx, size_t n_pixels, size_t n_bands,
                     const char* config_file);

void PCA_Context_free(PCA_Context* ctx);

int PCA_transform(PCA_Context* ctx, float** cube);

#endif
```

pca.c

```c
#include "dr/pca.h"
#include "gsl/gsl_blas.h"
#include <gsl/gsl_matrix.h>
#include <stddef.h>
#include <string.h>
#include <libconfig.h>

static int allocate(PCA_Context* ctx)
{
    ctx->basis = gsl_matrix_float_alloc(ctx->n_components, ctx->n_bands);
    ctx->means = gsl_vector_float_alloc(ctx->n_components);

    if (!ctx->basis || !ctx->means)
    {
        return -1;
    }

    return 0;
}

static int read_config(PCA_Context* ctx, const char* config_file)
{
    config_t cfg;
    config_init(&cfg);
    int ret = 0;
    ctx->binary_file = NULL;

    if (!config_read_file(&cfg, config_file))
    {
        printf("PCA: Could not read config file %s\n", config_file);
        ret = -1;
        goto free_cfg;
    }

    int n_comps;

    if (!config_lookup_int(&cfg, "n_components", &n_comps))
    {
        printf("PCA: Could not read number of components in file %s\n",
               config_file);
        ret = -2;
        goto free_cfg;
    }

    ctx->n_components = (size_t)n_comps;

    const char* tmp;

    if (!config_lookup_string(&cfg, "basis_binary_file", &tmp))
    {
        printf("PCA: Could not read basis binary file path in %s\n",
               config_file);
        ret = -3;
        goto free_cfg;
    }

    ctx->binary_file = strdup(tmp);

free_cfg:
    config_destroy(&cfg);
    return ret;
}

static size_t get_file_size(FILE* file)
{
    fseek(file, 0L, SEEK_END);
    size_t sz = ftell(file);
    rewind(file);
    return sz;
}

static int read_binary_data(PCA_Context* ctx)
{
    FILE* fp = fopen(ctx->binary_file, "rb");

    if (!fp)
    {
        printf("PCA: Cant open file %s\n", ctx->binary_file);
        return -1;
    }
    // Basis and means
    size_t n_basis = ctx->n_components * ctx->n_bands;
    size_t n_means = ctx->n_components;

    size_t e_size = (n_basis + n_means) * sizeof(float);
    size_t size = get_file_size(fp);

    if (size != e_size)
    {
        printf("PCA: File size not correct\n");
        printf("Expected %d, got %d\n", (int)e_size, (int)size);
    }

    // Read PCA basis vectors
    size_t nread = fread(ctx->basis->data, sizeof(float), n_basis, fp);
    nread += fread(ctx->means->data, sizeof(float), n_means, fp);

    int ret = 0;
    if (nread != n_basis + n_means)
    {
        printf("PCA: Read unexpected number of elements: %d\n", (int)nread);
        ret = -1;
    }

    fclose(fp);
    return ret;
}

int PCA_Context_init(PCA_Context* ctx, size_t n_pixels, size_t n_bands,
                     const char* config_file)
{
    ctx->n_pixels = n_pixels;
    ctx->n_bands = n_bands;

    int ret = read_config(ctx, config_file);
    if (ret)
    {
        printf("Failed to parse PCA config: %d\n", ret);
        return -1;
    }

    ret = allocate(ctx);

    if (ret)
    {
        printf("Failed to allocate PCA Context: %d\n", ret);
        goto free_ctx;
    }

    ret = read_binary_data(ctx);

    if (ret)
    {
        printf("Failed to read PCA binary data: %d\n", ret);
        goto free_ctx;
    }
```

```c
    return 0;

free_ctx:
    PCA_Context_free(ctx);

    return ret;
}

void PCA_Context_free(PCA_Context* ctx)
{
    gsl_matrix_float_free(ctx->basis);
    gsl_vector_float_free(ctx->means);
    free(ctx->binary_file);
}

int PCA_transform(PCA_Context* ctx, float** cube)
{
    float* dr_cube = malloc(sizeof(float) * ctx->n_pixels * ctx->n_components);

    if (!dr_cube)
    {
        printf("PCA: Could not allocate space for transformed cube\n");
        return -1;
    }

    gsl_matrix_float_view cube_matrix =
        gsl_matrix_float_view_array(*cube, ctx->n_pixels, ctx->n_bands);

    gsl_matrix_float_view dr_matrix =
        gsl_matrix_float_view_array(dr_cube, ctx->n_pixels, ctx->n_components);

    int ret = gsl_blas_sgemm(CblasNoTrans, CblasTrans, 1, &cube_matrix.matrix,
                             ctx->basis, 0, &dr_matrix.matrix);

    if (ret != GSL_SUCCESS)
    {
        printf("Failed to do PCA computation: %d\n", ret);
        free(dr_cube);
        return ret;
    }

    // Subtract projected mean
    for (size_t i = 0; i < ctx->n_pixels; ++i)
    {
        gsl_vector_float_view pix = gsl_matrix_float_row(&dr_matrix.matrix, i);
        gsl_vector_float_sub(&pix.vector, ctx->means);
    }

    free(*cube);
    *cube = dr_cube;

    return 0;
}
```

B.2 Self Organizing Map Clustering

cls_som.h

```c
#ifndef CLS_SOM_H_INCLUDED
#define CLS_SOM_H_INCLUDED
#include <stdint.h>
#include <stddef.h>

typedef struct
{
    size_t n_pixels;
    size_t n_bands;

    size_t SOM_dim;

    char* SOM_data_file;
    char* SOM_label_file;
    char* results_file;

    float* SOM_data;
    uint8_t* SOM_labels;

    float* hsi_cube_f;
    uint8_t* cube_labels;

} SomClassification;

int cls_som_allocate(SomClassification* ctx);
void cls_som_free(SomClassification* ctx);

int cls_som_classify(SomClassification* ctx);

int cls_som_read_config(SomClassification* ctx, const char* config_file);

int cls_som_read_SOM_data(SomClassification* ctx);
int cls_som_read_SOM_labels(SomClassification* ctx);

#endif
```

cls_som.c

```c
#include <stdint.h>
#include <math.h>
#include <stdio.h>
#include <float.h>
#include <libconfig.h>
#include <string.h>
#include <stdlib.h>

#include "classification/cls_som.h"

int cls_som_allocate(SomClassification* ctx)
{
    ctx->SOM_data =
        malloc(sizeof(float) * ctx->n_bands * ctx->SOM_dim * ctx->SOM_dim);
    ctx->SOM_labels = malloc(sizeof(uint8_t) * ctx->SOM_dim * ctx->SOM_dim);

    ctx->hsi_cube_f = malloc(sizeof(float) * ctx->n_bands * ctx->n_pixels);
    ctx->cube_labels = malloc(sizeof(uint8_t) * ctx->n_pixels);

    if (ctx->hsi_cube_f && ctx->SOM_data && ctx->SOM_labels && ctx->cube_labels)
    {
        return 0;
    }

    free(ctx->SOM_data);
    free(ctx->SOM_labels);
    free(ctx->hsi_cube_f);
    free(ctx->cube_labels);
    return -1;
}

void cls_som_free(SomClassification* ctx)
{
    free(ctx->SOM_data);
    free(ctx->SOM_labels);

    free(ctx->hsi_cube_f);
    free(ctx->cube_labels);

    free(ctx->SOM_data_file);
    free(ctx->SOM_label_file);
    free(ctx->results_file);
}

int cls_som_classify(SomClassification* ctx)
{
    // Iterate over all pixels
    for (size_t p = 0; p < ctx->n_pixels; ++p)
    {
        uint8_t bmu_lbl = ctx->SOM_labels[0];
        float min_bmu = FLT_MAX;

        float* pixel = &(ctx->hsi_cube_f[p * ctx->n_bands]);

        // Iterate over all SOM nodes and find best match
        for (size_t i = 0; i < ctx->SOM_dim * ctx->SOM_dim; ++i)
        {
            float bmu_tot = 0;
            float* som_node = &(ctx->SOM_data[i * ctx->n_bands]);

            // Calculate euclidian distance between node and pixel
            for (size_t j = 0; j < ctx->n_bands; ++j)
            {
                bmu_tot += (pixel[j] - som_node[j]) * (pixel[j] - som_node[j]);
            }

            if (bmu_tot < min_bmu)
            {
                min_bmu = bmu_tot;
                bmu_lbl = ctx->SOM_labels[i];
            }
        }

        ctx->cube_labels[p] = bmu_lbl;
    }

    return 0;
}
```

```c
int cls_som_read_config(SomClassification* ctx, const char* config_file)
{
    config_t cfg;
    config_init(&cfg);
    int ret = 0;

    if (!config_read_file(&cfg, config_file))
    {
        printf("Could not read config file %s\n", config_file);
        ret = -1;
        goto free_cfg;
    }

    const char* som_data_file;

    if (!(config_lookup_string(&cfg, "SOM_data_file", &som_data_file)))
    {
        printf("Could not read SOM_data_file path config in %s\n", config_file);
        ret = -2;
        goto free_cfg;
    }

    ctx->SOM_data_file = strdup(som_data_file);

    const char* som_label_file;

    if (!(config_lookup_string(&cfg, "SOM_label_file", &som_label_file)))
    {
        printf("Could not read SOM_label_file path config in %s\n",
               config_file);
        ret = -3;
        goto free_cfg;
    }

    ctx->SOM_label_file = strdup(som_label_file);

    const char* results_file;

    if (!(config_lookup_string(&cfg, "results_file", &results_file)))
    {
        printf("Could not read results_file path config in %s\n", config_file);
        ret = -4;
        goto free_cfg;
    }

    ctx->results_file = strdup(results_file);

    int SOM_dim;

    if (!config_lookup_int(&cfg, "SOM_dim", &SOM_dim))
    {
        printf("Could not read SOM_dim config in %s\n", config_file);
        ret = -5;
        goto free_cfg;
    }

    ctx->SOM_dim = SOM_dim;
    int n_bands;

    if (!config_lookup_int(&cfg, "n_bands", &n_bands))
    {
        printf("Could not read n_bands config in %s\n", config_file);
        ret = -6;
        goto free_cfg;
    }

    ctx->n_bands = n_bands;

free_cfg:
    config_destroy(&cfg);
    return ret;
}

int cls_som_read_SOM_data(SomClassification* ctx)
{
    FILE* fp = fopen(ctx->SOM_data_file, "rb");

    if (!fp)
    {
        printf("Cant open file %s\n", ctx->SOM_data_file);
        return -1;
    }

    size_t n_elems = ctx->SOM_dim * ctx->SOM_dim * ctx->n_bands;

    size_t nread = fread(ctx->SOM_data, sizeof(float), n_elems, fp);

    if (nread != n_elems)
    {
        printf("Failed to read correct number of elements: %d != %d\n",
               (int)n_elems, (int)nread);
        return -2;
    }

    fclose(fp);
    return 0;
}

int cls_som_read_SOM_labels(SomClassification* ctx)
{
    FILE* fp = fopen(ctx->SOM_label_file, "rb");

    if (!fp)
    {
        printf("Cant open file %s\n", ctx->SOM_label_file);
        return -1;
    }

    size_t n_elems = ctx->SOM_dim * ctx->SOM_dim;

    size_t nread = fread(ctx->SOM_labels, sizeof(uint8_t), n_elems, fp);

    if (nread != n_elems)
    {
        printf("Failed to read correct number of elements: %d != %d\n",
               (int)n_elems, (int)nread);
        return -2;
    }

    fclose(fp);
    return 0;
}
```

B.3 Support Vector Machine

cls_svm.h

```
#ifndef CLS_SVM_H_INCLUDED
#define CLS_SVM_H_INCLUDED
#include <stdint.h>
#include <stddef.h>

#define SVM_MAX_CLASSES 16

/* Holds data for the support vectors and coeffcients for each class */
typedef struct
{
    // Support vectors: array[n_sv][n_bands]
    float** support_vectors;

    // Kernel coefficients for each support vector
    // array[n_classes - 1][n_sv]
    float** coefficients;

    size_t n_sv;
} SVM_data;

typedef struct
{
    size_t n_pixels;
    size_t n_bands;

    char* SVM_binary_file;

    size_t n_classes;
    size_t n_support[SVM_MAX_CLASSES];
    SVM_data classifiers[SVM_MAX_CLASSES];

    float* intercepts;

    float gamma;

    char* scaler_file;
    float* means;
    float* stds;

    char* results_file;
    uint8_t* labels;

    int is_allocated;
} SVM_Classification;

int cls_svm_init(SVM_Classification* ctx, const char* config_file,
                 size_t n_pixels);
void cls_svm_free(SVM_Classification* ctx);

int cls_svm_classify(SVM_Classification* ctx, const float* hsi_cube);

#endif
```

cls_svm.c

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <libconfig.h>
#include <math.h>

#include "classification/cls_svm.h"

static int cls_svm_read_config(SVM_Classification* ctx, const char* config_file)
{
    config_t cfg;
    config_init(&cfg);
    int ret = 0;

    ctx->scaler_file = NULL;
    ctx->results_file = NULL;
    ctx->SVM_binary_file = NULL;

    if (!config_read_file(&cfg, config_file))
    {
        printf("Could not read config file %s\n", config_file);
        ret = -1;
        goto free_cfg;
    }

    const char* svm_bin;
    if (!(config_lookup_string(&cfg, "SVM_binary_file", &svm_bin)))
    {
        printf("Could not read SVM_binary_file path config in %s\n",
               config_file);
        ret = -2;
        goto free_cfg;
    }
    ctx->SVM_binary_file = strdup(svm_bin);

    const char* scaler_file;
    if (!(config_lookup_string(&cfg, "scaler_file", &scaler_file)))
    {
        printf("Could not read scaler_file path config in %s\n", config_file);
        printf("Skipping scaling");
    }
    else
    {
        ctx->scaler_file = strdup(scaler_file);
    }
    const char* results_file;
    if (!(config_lookup_string(&cfg, "results_file", &results_file)))
    {
        printf("Could not read results_file path config in %s\n", config_file);
        ret = -4;
        goto free_cfg;
    }
    ctx->results_file = strdup(results_file);

    int n_bands;
    if (!config_lookup_int(&cfg, "n_bands", &n_bands))
    {
        printf("Could not read n_bands config in %s\n", config_file);
        ret = -5;
        goto free_cfg;
    }
    ctx->n_bands = n_bands;

    int n_classes;
    if (!config_lookup_int(&cfg, "n_classes", &n_classes))
    {
        printf("Could not read n_classes config in %s\n", config_file);
        ret = -6;
        goto free_cfg;
    }
    ctx->n_classes = n_classes;

    char buf[16];

    for (int i = 0; i < ctx->n_classes; ++i)
    {
        sprintf(buf, "n_SV%d", i);
        int nsv;
        if (!config_lookup_int(&cfg, buf, &nsv))
        {
            printf("Could not read %s config in %s\n", buf, config_file);
            ret = -7;
            goto free_cfg;
        }

        ctx->n_support[i] = nsv;
    }

    double gamma;

    if (!config_lookup_float(&cfg, "gamma", &gamma))
    {
        printf("Could not read rbf kernel gamma param from %s\n", config_file);
        ret = -1;
        goto free_cfg;
    }
    ctx->gamma = (float)gamma;

free_cfg:
    config_destroy(&cfg);
    return ret;
}
```

```c
static int cls_svm_check_config(SVM_Classification* ctx)
{
    if (ctx->n_pixels > 1500 * 1500 || ctx->n_pixels == 0)
    {
        printf("Unexpected number of pixels %d\n", (int)ctx->n_pixels);
        return -1;
    }

    if (ctx->n_bands > 2000 || ctx->n_bands == 0)
    {
        printf("Unexpected number of bands %d\n", (int)ctx->n_bands);
        return -2;
    }

    if (ctx->n_classes > SVM_MAX_CLASSES || ctx->n_classes < 2)
    {
        printf("Unexpected number of classes %d\n", (int)ctx->n_classes);
        return -3;
    }

    for (int i = 0; i < ctx->n_classes; ++i)
    {
        if (ctx->n_support[i] == 0 || ctx->n_support[i] > 20000)
        {
            printf("Unexpected number of support vectors for class %d: %d\n",
                   (int)ctx->n_support[i], i);
            return -4;
        }
    }

    return 0;
}

static int get_number_of_classifiers(const SVM_Classification* ctx)
{
    return (ctx->n_classes * (ctx->n_classes - 1)) / 2;
}

static int cls_svm_allocate(SVM_Classification* ctx)
{
    /* Allocate and check rows of support vectors */
    for (int i = 0; i < ctx->n_classes; ++i)
    {
        size_t nsv = ctx->n_support[i];
        ctx->classifiers[i].support_vectors = malloc(sizeof(float*) * nsv);
        ctx->classifiers[i].n_sv = nsv;
    }

    for (int i = 0; i < ctx->n_classes; ++i)
    {
        if (!ctx->classifiers[i].support_vectors)
        {
            goto sv_malloc_error;
        }
    }

    /* Allocate and check rows coefficients */
```

```c
for (int i = 0; i < ctx->n_classes; ++i)
{
    ctx->classifiers[i].coefficients =
        malloc(sizeof(float*) * (ctx->n_classes - 1));
}

for (int i = 0; i < ctx->n_classes; ++i)
{
    if (!ctx->classifiers[i].coefficients)
    {
        goto coeff_malloc_error;
    }
}

/* If rows of both support vectors and
coefficients are OK, then allocate columns
*/

for (int i = 0; i < ctx->n_classes; ++i)
{
    int nsv = ctx->n_support[i];
    for (int j = 0; j < nsv; ++j)
    {
        ctx->classifiers[i].support_vectors[j] =
            malloc(sizeof(float) * ctx->n_bands);
    }

    for (int j = 0; j < (ctx->n_classes - 1); ++j)
    {
        ctx->classifiers[i].coefficients[j] = malloc(sizeof(float) * nsv);
    }
}

ctx->intercepts = malloc(sizeof(float) * get_number_of_classifiers(ctx));

if (ctx->scaler_file != NULL)
{
    ctx->means = malloc(sizeof(float) * ctx->n_bands);
    ctx->stds = malloc(sizeof(float) * ctx->n_bands);
}
else
{
    ctx->means = NULL;
    ctx->stds = NULL;
}

ctx->labels = malloc(sizeof(uint8_t) * ctx->n_pixels);

// Now check everything

for (int i = 0; i < ctx->n_classes; ++i)
{
    SVM_data* d = &(ctx->classifiers[i]);

    for (int j = 0; j < d->n_sv; ++j)
    {
        if (!d->support_vectors[j])
        {
                goto malloc_error;
            }
        }

        for (int j = 0; j < ctx->n_classes - 1; ++j)
        {
            if (!d->coefficients[j])
            {
                goto malloc_error;
            }
        }
    }

    if (!ctx->intercepts || !ctx->labels)
    {
        goto malloc_error;
    }

    if (ctx->scaler_file != NULL)
    {
        if (!ctx->means || !ctx->stds)
        {
            goto malloc_error;
        }
    }

    return 0;

malloc_error:
    cls_svm_free(ctx);
    return -1;

coeff_malloc_error:
    for (size_t i = 0; i < ctx->n_classes - 1; ++i)
    {
        free(ctx->classifiers[i].coefficients);
    }

sv_malloc_error:
    for (size_t i = 0; i < ctx->n_classes; ++i)
    {
        free(ctx->classifiers[i].support_vectors);
    }

    return -2;
}

static size_t get_file_size(FILE* file)
{
    fseek(file, 0L, SEEK_END);
    size_t sz = ftell(file);
    rewind(file);
    return sz;
}

static int cls_svm_read_binary_data(SVM_Classification* ctx)
{
    FILE* fptr = fopen(ctx->SVM_binary_file, "rb");
```

```
if (!fptr)
{
    return -1;
}

size_t sz = get_file_size(fptr);

size_t expected_size = 0;

for (int i = 0; i < ctx->n_classes; ++i)
{
    SVM_data* d = &(ctx->classifiers[i]);
    // Support vectors
    expected_size += d->n_sv * ctx->n_bands * sizeof(float);

    // Coefficients
    expected_size += d->n_sv * (ctx->n_classes - 1) * sizeof(float);
}

// Intercepts
expected_size += sizeof(float) * get_number_of_classifiers(ctx);

if (expected_size != sz)
{
    printf("Wrong file size for %s. Expected %d got %d\n",
           ctx->SVM_binary_file, (int)expected_size, (int)sz);
}

int unused;

// First comes support vectors
for (int i = 0; i < ctx->n_classes; ++i)
{
    SVM_data* d = &(ctx->classifiers[i]);

    for (int j = 0; j < d->n_sv; ++j)
    {
        unused =
            fread(d->support_vectors[j], sizeof(float), ctx->n_bands, fptr);
    }
}

// Now Coefficients
for (int i = 0; i < ctx->n_classes; ++i)
{
    SVM_data* d = &(ctx->classifiers[i]);

    for (int j = 0; j < ctx->n_classes - 1; ++j)
    {
        unused = fread(d->coefficients[j], sizeof(float), d->n_sv, fptr);
    }
}

// Intercepts
unused = fread(ctx->intercepts, sizeof(float),
               get_number_of_classifiers(ctx), fptr);
    (void)unused;

    fclose(fptr);

    return 0;
}

static int cls_svm_read_scaler_data(SVM_Classification* ctx)
{
    FILE* fptr = fopen(ctx->scaler_file, "rb");
    if (!fptr)
    {
        return -1;
    }

    size_t file_size = get_file_size(fptr);

    size_t expected_size = 2 * sizeof(float) * ctx->n_bands;
    if (expected_size != file_size)
    {
        printf("Wrong file size for %s. Expected %d got %d\n", ctx->scaler_file,
               (int)expected_size, (int)file_size);
    }

    int unused = fread(ctx->means, sizeof(float), ctx->n_bands, fptr);
    unused = fread(ctx->stds, sizeof(float), ctx->n_bands, fptr);
    (void)unused;

    fclose(fptr);
    return 0;
}

int cls_svm_init(SVM_Classification* ctx, const char* config_file,
                 size_t n_pixels)
{
    ctx->n_pixels = n_pixels;
    ctx->is_allocated = 0;

    int ret = cls_svm_read_config(ctx, config_file);
    if (ret)
    {
        printf("Parsing configuration file %s failed\n", config_file);
        return ret;
    }

    ret = cls_svm_check_config(ctx);
    if (ret)
    {
        printf("Invalid configuration file %s: %d\n", config_file, ret);
    }

    ret = cls_svm_allocate(ctx);
    if (ret)
    {
        printf("Failed to allocate SVM buffers: %d\n", ret);
        return ret;
    }
    ctx->is_allocated = 1;

    ret = cls_svm_read_binary_data(ctx);
    if (ret)
    {
        printf("Failed to SVM binary data from %s\n", ctx->SVM_binary_file);
        return ret;
    }

    if (ctx->scaler_file != NULL)
    {
        ret = cls_svm_read_scaler_data(ctx);
    }

    if (ret)
    {
        printf("Failed to read SVM scaling data from %s\n", ctx->scaler_file);
        return ret;
    }

    return ret;
}

void cls_svm_free(SVM_Classification* ctx)
{
    free(ctx->SVM_binary_file);
    free(ctx->scaler_file);
    free(ctx->results_file);

    if (ctx->is_allocated == 0)
    {
        return;
    }

    for (int i = 0; i < ctx->n_classes; ++i)
    {
        SVM_data* d = &(ctx->classifiers[i]);

        for (int j = 0; j < d->n_sv; ++j)
        {
            free(d->support_vectors[j]);
        }

        free(d->support_vectors);

        for (int j = 0; j < ctx->n_classes - 1; ++j)
        {
            free(d->coefficients[j]);
        }

        free(d->coefficients);
    }

    free(ctx->intercepts);

    free(ctx->means);
    free(ctx->stds);
```

```c
    free(ctx->labels);
}

// Temporary arrays
// Allocated and freed in cls_svm_classify
static float* confidence_sums;
static float* ovo_results;
static float* ovr_results;
static float* scaled_pix;
static float* kernel_results;

// Compute the learned kernel for pixel x in the ovo classifier for i vs j
static float rbf_kernel(const float* x, int class_i, int class_j,
                        const SVM_Classification* ctx)
{
    float value = 0;
    for (int k = 0; k < 2; ++k)
    {
        int c1, c2;

        // On first iteration evaluate the support vectors of class i
        if (k == 0)
        {
            c1 = class_i;
            c2 = class_j;
        }

        // Second iteration is support vectors of class j
        else
        {
            c1 = class_j;
            c2 = class_i;
        }

        const SVM_data* c1_data = &(ctx->classifiers[c1]);

        // Find coefficients for c1 trained against c2
        int c1_coeffs_idx = c1 < c2 ? c2 - 1 : c2;
        const float* c1_coeffs = c1_data->coefficients[c1_coeffs_idx];

        // Find support vectors for c1
        const float* c1_kernel_results = kernel_results;
        for (int i = 0; i < c1; ++i)
        {
            c1_kernel_results += ctx->n_support[i];
        }

        // Evaluate c1 trained against c2
        for (int i = 0; i < c1_data->n_sv; ++i)
        {
            value += c1_coeffs[i] * c1_kernel_results[i];
        }
    }

    return value;
}

static void precompute_kernel(const SVM_Classification* ctx, const float* x)
```

```
{
    size_t idx = 0;
    for (size_t i = 0; i < ctx->n_classes; ++i)
    {
        for (size_t j = 0; j < ctx->n_support[i]; ++j)
        {
            kernel_results[idx] = 0;
            const float* sv = ctx->classifiers[i].support_vectors[j];
            for (size_t n = 0; n < ctx->n_bands; ++n)
            {
                kernel_results[idx] -= (x[n] - sv[n]) * (x[n] - sv[n]);
            }

            kernel_results[idx] = expf(ctx->gamma * kernel_results[idx]);
            ++idx;
        }
    }
}

// Compupte the SVM decicion function for each OvO classifier
// https://scikit-learn.org/stable/modules/svm.html#svc
static void decicion_function(const SVM_Classification* ctx, const float* x,
                              float* results)
{
    int idx = 0;

    precompute_kernel(ctx, x);

    for (size_t i = 0; i < ctx->n_classes; ++i)
    {
        for (size_t j = i + 1; j < ctx->n_classes; ++j)
        {
            results[idx] = rbf_kernel(x, i, j, ctx) + ctx->intercepts[idx];
            ++idx;
        }
    }
}

static void ovo_to_ovr(float* ovo, float* ovr, int n_classes)
{
    // Reset arrays
    for (int i = 0; i < n_classes; ++i)
    {
        confidence_sums[i] = 0;
        ovr[i] = 0;
    }

    int k = 0;
    for (int i = 0; i < n_classes; ++i)
    {
        for (int j = i + 1; j < n_classes; ++j)
        {
            confidence_sums[i] += ovo[k];
            confidence_sums[j] -= ovo[k];

            ovr[i] += (ovo[k] > 0);
            ovr[j] += (ovo[k] < 0);
```

```c
                ++k;
            }
        }

        for (int i = 0; i < n_classes; ++i)
        {
            ovr[i] += confidence_sums[i] / (3 * (fabsf(confidence_sums[i]) + 1));
        }
}

static void scale_pixel(const float* pixel, float* scaled, const float* means,
                        const float* stds, size_t length)
{
    for (size_t i = 0; i < length; ++i)
    {
        scaled[i] = (pixel[i] - means[i]) / (stds[i]);
    }
}

static uint8_t classify_single(SVM_Classification* ctx, const float* pix)
{
    if (ctx->scaler_file == NULL)
    {
        decicion_function(ctx, pix, ovo_results);
    }

    else
    {
        scale_pixel(pix, scaled_pix, ctx->means, ctx->stds, ctx->n_bands);
        decicion_function(ctx, scaled_pix, ovo_results);
    }

    ovo_to_ovr(ovo_results, ovr_results, ctx->n_classes);

    // Label is argmax of ovr
    uint8_t label = 0;
    float max_val = ovr_results[0];
    for (int j = 1; j < ctx->n_classes; ++j)
    {
        if (ovr_results[j] > max_val)
        {
            max_val = ovr_results[j];
            label = j;
        }
    }

    return label;
}

int cls_svm_classify(SVM_Classification* ctx, const float* hsi_cube)
{
    scaled_pix = NULL;
    if (ctx->scaler_file != NULL)
    {
        scaled_pix = malloc(sizeof(float) * ctx->n_bands);
    }

    // One vs one
    ovo_results = malloc(sizeof(float) * get_number_of_classifiers(ctx));

    // One vs rest
    ovr_results = malloc(sizeof(float) * ctx->n_classes);

    confidence_sums = malloc(sizeof(float) * ctx->n_classes);

    size_t n_support_total = 0;

    for (int i = 0; i < ctx->n_classes; ++i)
    {
        n_support_total += ctx->n_support[i];
    }

    kernel_results = malloc(sizeof(float) * n_support_total);

    for (size_t i = 0; i < ctx->n_pixels; ++i)
    {
        const float* pix = &(hsi_cube[i * ctx->n_bands]);
        ctx->labels[i] = classify_single(ctx, pix);
    }

    free(scaled_pix);
    free(ovo_results);
    free(ovr_results);
    free(confidence_sums);
    free(kernel_results);

    return 0;
}
```

B.4 Tests

test_classification.c

```
#include <check.h>
#include <math.h>
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>

#include "pipeline/PL_runner.h"
#include "pipeline/PL_builder.h"
#include "classification/cls_som.h"
#include "classification/cls_svm.h"

#define TEST_DATA_DIR "./tests/test_data/classification/"

#define SOM_CLASS_CONFIG_FILE_PATH TEST_DATA_DIR "class_som.config"
#define SVM_CLASS_CONFIG_FILE_PATH TEST_DATA_DIR "class_svm.config"

#define SOM_DATA_FILE TEST_DATA_DIR "jasper_som.bin"
#define SOM_LABEL_FILE TEST_DATA_DIR "jasper_som_labels.bin"
#define RESULTS_FILE TEST_DATA_DIR "results"

#define CLASS_TEST_CUBE TEST_DATA_DIR "jasper.bip"
```

```
#define CLASS_GT TEST_DATA_DIR "jasper_gt.bip"

#define CUBE_OUTPUT_PATH TEST_DATA_DIR "NOT_USED"

// Test cube dims
static int n_bands = 198;
static int h = 100;
static int w = 100;

static void setup(void)
{
}

static void teardown(void)
{
    remove(CUBE_OUTPUT_PATH);
    remove(RESULTS_FILE);
}

static int read_results(uint8_t* buf, char* file)
{
    FILE* fp = fopen(file, "rb");

    if (!fp)
    {
        return -1;
    }
    size_t nread = fread(buf, sizeof(uint8_t), h * w, fp);

    if (nread != h * w)
    {
        printf("Did not read expected number of elements: %d - %d\n",
               (int)nread, (int)(h * w));
        return -2;
    }

    return 0;
}

START_TEST(test_som_classification_config)
{
    SomClassification ctx;
    cls_som_read_config(&ctx, SOM_CLASS_CONFIG_FILE_PATH);

    ck_assert_msg(ctx.SOM_dim == 7, "Wrong som dimension");
    ck_assert_msg(ctx.n_bands == n_bands, "Wrong number of bands");

    ck_assert_str_eq(ctx.SOM_data_file, SOM_DATA_FILE);
    ck_assert_str_eq(ctx.SOM_label_file, SOM_LABEL_FILE);
    ck_assert_str_eq(ctx.results_file, RESULTS_FILE);

    free(ctx.SOM_data_file);
    free(ctx.SOM_label_file);
    free(ctx.results_file);
}
END_TEST

START_TEST(test_som_classification_module)
```

```
{
    PipelineContext pl_ctx;
    pl_ctx.input_path = strdup(CLASS_TEST_CUBE);
    pl_ctx.output_path = strdup(CUBE_OUTPUT_PATH);
    pl_ctx.n_bands = n_bands;
    pl_ctx.spatial_height = h;
    pl_ctx.spatial_width = w;
    pl_ctx.num_partitions = 1;

    pl_ctx.pl = PL_builder_new_pipeline();
    PL_builder_add_module(pl_ctx.pl, PM_CLASSIFICATION_SOM,
                          SOM_CLASS_CONFIG_FILE_PATH);

    int ret = PL_runner_execute(&pl_ctx);

    ck_assert_msg(ret == 0, "Failed to apply classification pipeline");

    uint8_t* results = malloc(sizeof(uint8_t) * h * w);
    uint8_t* gt = malloc(sizeof(uint8_t) * h * w);

    ret = read_results(results, RESULTS_FILE);
    ck_assert_msg(ret == 0, "Failed to read results file correctly %d", ret);

    ret = read_results(gt, CLASS_GT);
    ck_assert_msg(ret == 0, "Failed to ground truth file correctly %d", ret);

    float n_corr = 0;
    for (size_t i = 0; i < h * w; ++i)
    {
        results[i] == gt[i] ? n_corr += 1.0f : 0;
    }

    float tot_acc = n_corr / (h * w);

    printf("SOM Classification accuracy: %.2f\n", tot_acc);
    ck_assert_msg(tot_acc > 0.5, "Classification accuracy unexpecdedly low\n");

    free(results);
    free(gt);
    free(pl_ctx.input_path);
    free(pl_ctx.output_path);
    PL_builder_free_pipeline(pl_ctx.pl);
}
END_TEST

START_TEST(test_svm_classification_module)
{
    PipelineContext pl_ctx;
    pl_ctx.input_path = strdup(CLASS_TEST_CUBE);
    pl_ctx.output_path = strdup(CUBE_OUTPUT_PATH);
    pl_ctx.n_bands = n_bands;
    pl_ctx.spatial_height = h;
    pl_ctx.spatial_width = w;
    pl_ctx.num_partitions = 1;

    pl_ctx.pl = PL_builder_new_pipeline();
    PL_builder_add_module(pl_ctx.pl, PM_CLASSIFICATION_SVM,
                          SVM_CLASS_CONFIG_FILE_PATH);
    int ret = PL_runner_execute(&pl_ctx);

    ck_assert_msg(ret == 0, "Failed to apply classification pipeline");

    uint8_t* results = malloc(sizeof(uint8_t) * h * w);
    uint8_t* gt = malloc(sizeof(uint8_t) * h * w);

    ret = read_results(results, RESULTS_FILE);
    ck_assert_msg(ret == 0, "Failed to read results file correctly %d", ret);

    ret = read_results(gt, CLASS_GT);
    ck_assert_msg(ret == 0, "Failed to ground truth file correctly %d", ret);

    float n_corr = 0;
    for (size_t i = 0; i < h * w; ++i)
    {
        results[i] == gt[i] ? n_corr += 1.0f : 0;
    }

    float tot_acc = n_corr / (h * w);

    printf("SVM Classification accuracy: %.2f\n", tot_acc);
    ck_assert_msg(tot_acc > 0.5, "Classification accuracy unexpecdedly low\n");

    free(results);
    free(gt);
    free(pl_ctx.input_path);
    free(pl_ctx.output_path);
    PL_builder_free_pipeline(pl_ctx.pl);
}
END_TEST

Suite* make_classification_suite(void)
{
    Suite* suite = suite_create("classification_suite");
    TCase* tcase = tcase_create("classification_case");

    tcase_set_timeout(tcase, 100);

    tcase_add_test(tcase, test_som_classification_config);
    tcase_add_test(tcase, test_som_classification_module);
    tcase_add_test(tcase, test_svm_classification_module);

    tcase_add_checked_fixture(tcase, setup, teardown);

    suite_add_tcase(suite, tcase);

    return suite;
}
```

test_pca.c

```
#include <math.h>
#include <check.h>
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
```

```c
#include <string.h>

#include "dr/pca.h"
#include "pipeline/PL_builder.h"
#include "pipeline/PL_runner.h"
#include "pipeline/PL_utils.h"

#define TEST_DATA_DIR "./tests/test_data/"

#define PCA_CONFIG TEST_DATA_DIR "pca/jasper_pca.config"

#define TEST_CUBE TEST_DATA_DIR "classification/jasper.bip"
#define CUBE_OUTPUT_PATH TEST_DATA_DIR "out.bip"

#define JASPER_EXPECTED_CUBE TEST_DATA_DIR "pca/jasper_pca.bip"

static int n_bands = 198;
static int h = 100;
static int w = 100;
static int nc = 10;

static void setup(void)
{
}

static void teardown(void)
{
    remove(CUBE_OUTPUT_PATH);
}

START_TEST(test_pca_module)
{
    PipelineContext pl_ctx;
    pl_ctx.input_path = strdup(TEST_CUBE);
    pl_ctx.output_path = strdup(CUBE_OUTPUT_PATH);
    pl_ctx.n_bands = n_bands;
    pl_ctx.spatial_height = h;
    pl_ctx.spatial_width = w;
    pl_ctx.num_partitions = 1;

    pl_ctx.pl = PL_builder_new_pipeline();
    PL_builder_add_module(pl_ctx.pl, PM_DIMENSIONALITY_REDUCTION, PCA_CONFIG);

    pl_ctx.hsi_cube = malloc(sizeof(cube_t) * h * w * n_bands);
    PL_utils_read_cube(TEST_CUBE, h * w * n_bands, pl_ctx.hsi_cube);

    int ret = PL_runner_apply_pipeline(&pl_ctx);

    ck_assert_msg(ret == 0, "Failed to apply pca pipeline");
    ck_assert_msg(pl_ctx.n_bands == nc, "Incorrect number of dimensions");

    float* expected = malloc(sizeof(float) * h * w * nc);

    FILE* fd = fopen(JASPER_EXPECTED_CUBE, "rb");
    ck_assert_msg(fd, "Failed to read cube file correctly");

    int unused = fread(expected, sizeof(float), w * h * nc, fd);
    ++unused;
    int error = 0;
    for (size_t i = 0; i < h * w * pl_ctx.n_bands; ++i)
    {
        if (fabsf(pl_ctx.hsi_cube[i] - expected[i]) > 1e-1)
        {
            error = 1;
        }
    }

    ck_assert_msg(error == 0, "Error in PCA tranformation");

    free(expected);
    free(pl_ctx.input_path);
    free(pl_ctx.output_path);
    PL_builder_free_pipeline(pl_ctx.pl);
    free(pl_ctx.hsi_cube);
}
END_TEST

Suite* make_pca_suite(void)
{
    Suite* suite = suite_create("pca_suite");
    TCase* tcase = tcase_create("case");

    tcase_set_timeout(tcase, 100);

    tcase_add_test(tcase, test_pca_module);

    tcase_add_checked_fixture(tcase, setup, teardown);

    suite_add_tcase(suite, tcase);

    return suite;
}
```

test_pl_runner.c

```
#include <check.h>
#include <stdint.h>
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

#include "common/types.h"
#include "pipeline/PL_builder.h"
#include "pipeline/PL_runner.h"
#include "pipeline/PL_utils.h"

static char test_cube_path[] = "./tests/test_data/TEMP_CUBE.bip";
static char test_cube_out_path1[] = "./tests/test_data/TEMP_CUBE_OUT1.bip";
static char test_cube_out_path2[] = "./tests/test_data/TEMP_CUBE_OUT2.bip";
static char test_config[] = "./tests/test_config1.config";

static void setup(void)
{
}
```

```
static void teardown(void)
{
    remove(test_cube_path);
    remove(test_cube_out_path1);
    remove(test_cube_out_path2);
}

// Test that an invalid pipeline is not accepted
START_TEST(test_invalid_pipeline_module)
{
    Pipeline* pl = PL_builder_new_pipeline();
    PL_builder_add_module(pl, 1000, "");

    PipelineContext ctx;
    ctx.pl = pl;
    ctx.input_path = "NOT_A_CUBE";
    ctx.output_path = "UNUSED";
    cube_t tmp[1] = { 0 };
    ctx.hsi_cube = tmp;
    ctx.n_bands = 1;
    ctx.spatial_height = 1;
    ctx.spatial_width = 1;
    ctx.num_partitions = 1;

    ck_assert_msg(PL_runner_execute(&ctx) != 0, "Zero return code on invalid "
                                                "cube\n");
    ck_assert_msg(PL_runner_apply_pipeline(&ctx) != 0, "Zero return code on "
                                                       "invalid "
                                                       "pipeline\n");

    PL_builder_free_pipeline(pl);
}
END_TEST

START_TEST(test_pipeline_partitioning)
{
    PipelineContext ctx1;

    ctx1.pl = PL_builder_from_config_file(test_config);
    ck_assert_msg(ctx1.pl != NULL, "Config is NULL\n");

    // Generate some random data for testing
    srand(time(NULL));

    size_t h = 100;
    size_t w = 50;
    size_t nb = 100;

    cube_t* buf = malloc(sizeof(cube_t) * h * w * nb);

    for (size_t i = 0; i < h * w * nb; ++i)
    {
        buf[i] = rand() % (UINT16_MAX);
    }

    PL_utils_write_cube(test_cube_path, w * h * nb, buf);

    ctx1.spatial_height = h;
```

```c
    ctx1.spatial_width = w;
    ctx1.n_bands = nb;
    ctx1.input_path = test_cube_path;
    ctx1.output_path = test_cube_out_path1;
    ctx1.num_partitions = 1;

    PipelineContext ctx2 = ctx1;
    ctx2.num_partitions = 3;
    ctx2.output_path = test_cube_out_path2;

    int ret = PL_runner_execute(&ctx1);

    ck_assert_msg(ret == 0, "Pipeline failed to execute\n");

    cube_t* out_buf = malloc(sizeof(cube_t) * h * w * nb);
    PL_utils_read_cube(test_cube_out_path1, w * h * nb, out_buf);

    int is_eq = memcmp(buf, out_buf, w * h * nb * sizeof(cube_t));

    ck_assert_msg(is_eq != 0, "Processed cube should not be equal to input "
                              "cube\n");

    ret = PL_runner_execute(&ctx2);
    ck_assert_msg(ret == 0, "Pipeline failed to execute with multiple "
                            "partitions\n");

    // Reuse buf
    PL_utils_read_cube(test_cube_out_path2, w * h * nb, buf);

    is_eq = memcmp(buf, out_buf, w * h * nb * sizeof(cube_t));
    ck_assert_msg(is_eq == 0, "Processed cubes should be equal\n");

    free(buf);
    free(out_buf);
    PL_builder_free_pipeline(ctx1.pl);
}
END_TEST

Suite* make_pl_runner_suite(void)
{
    Suite* suite = suite_create("pl_runner_suite");
    TCase* tcase = tcase_create("pl_runner_case");

    tcase_add_test(tcase, test_invalid_pipeline_module);
    tcase_add_test(tcase, test_pipeline_partitioning);

    tcase_add_checked_fixture(tcase, setup, teardown);

    suite_add_tcase(suite, tcase);

    return suite;
}
```

test_pl_builder.c

```c
#include <check.h>
#include <malloc.h>
#include <stdio.h>
```

```
#include "pipeline/PL_builder.h"
#include "pipeline/PL_utils.h"

static char* dummy_path1 = "/tmp/config/snk";
static char* dummy_path2 = "/tmp/config/td";
static char* test_config_file = "./tests/test_config2.config";

START_TEST(empty_pipeline)
{
    Pipeline* pl = PL_builder_new_pipeline();

    ck_assert(pl->num_modules == 0);
    ck_assert(pl->modules == NULL);
    PL_builder_free_pipeline(pl);
}
END_TEST

static void add_dummy_modules(Pipeline* pl)
{
    PL_builder_add_module(pl, PM_SMILE_AND_KEYSTONE, dummy_path1);
    PL_builder_add_module(pl, PM_TARGET_DETECTION, dummy_path2);

    char* dummy_path3 = "/tmp/config/compression";
    PL_builder_add_module(pl, PM_COMPRESSION, dummy_path3);
}

static int is_pipeline_eq(Pipeline* pl1, Pipeline* pl2)
{
    if (pl1->num_modules != pl2->num_modules)
    {
        return 0;
    }

    for (size_t i = 0; i < pl1->num_modules; ++i)
    {
        if (pl1->modules[i].module_type != pl2->modules[i].module_type)
        {
            return 0;
        }

        if (strcmp(pl1->modules[i].config_file_path,
                   pl2->modules[i].config_file_path))
        {
            return 0;
        }
    }

    return 1;
}

START_TEST(add_module)
{
    Pipeline* pl = PL_builder_new_pipeline();
    add_dummy_modules(pl);

    ck_assert(pl->num_modules == 3);
    ck_assert(pl->modules[0].module_type == PM_SMILE_AND_KEYSTONE);
    ck_assert(pl->modules[2].module_type == PM_COMPRESSION);
    ck_assert_str_eq(pl->modules[0].config_file_path, dummy_path1);
    ck_assert_str_eq(pl->modules[2].config_file_path, "/tmp/config/"
                                                      "compression");

    PL_builder_free_pipeline(pl);
}
END_TEST

START_TEST(pipeline_size)
{
    Pipeline* pl = PL_builder_new_pipeline();
    add_dummy_modules(pl);

    size_t sz = sizeof(size_t) + 3 * sizeof(PipelineModuleTypes) +
                3 * sizeof(size_t) + 15 + 14 + 23;

    ck_assert_msg(sz == PL_builder_get_pipeline_size(pl), "Incorrect pipeline "
                                                          "size\n");
    PL_builder_free_pipeline(pl);
}
END_TEST

START_TEST(serialize_pipeline)
{
    Pipeline* pl = PL_builder_new_pipeline();
    add_dummy_modules(pl);

    size_t sz = PL_builder_get_pipeline_size(pl);

    size_t buf_size = sz + 1;

    char* buf = malloc(buf_size);
    memset(buf, 0, buf_size);

    size_t num_written = PL_builder_serialize_pipeline(buf, pl);
    buf[buf_size - 1] = '\0';

    ck_assert_msg(num_written == sz, "Did not write expected number of "
                                     "bytes\n");

    Pipeline* pl_deserialized = PL_builder_deserialize_pipeline(buf);

    ck_assert_msg(is_pipeline_eq(pl, pl_deserialized), "Pipelines is not "
                                                       "equal\n");

    ck_assert_msg(sz == PL_builder_get_pipeline_size(pl_deserialized),
                  "Pipeline sizes does not match\n");

    ck_assert_msg(pl_deserialized->modules[2].module_type == PM_COMPRESSION,
                  "Wrong "
                  "expected "
                  "pipeline "
                  "type\n");

    ck_assert_str_eq(pl_deserialized->modules[1].config_file_path, dummy_path2);
    ck_assert_str_eq(pl_deserialized->modules[2].config_file_path, "/tmp/"
                                                                   "config/"
                                                                   "compressio"
```

```
                                                                           "n");

    PL_builder_free_pipeline(pl);
    PL_builder_free_pipeline(pl_deserialized);
    free(buf);
}
END_TEST

START_TEST(test_pl_config_reader)
{
    printf("Attempting to read config file %s\n", test_config_file);
    Pipeline* pl = PL_builder_from_config_file(test_config_file);

    ck_assert_msg(pl != NULL, "Could not read pipeline config file\n");

    int expected_num_modules = 6;

    char error_msg[100];

    sprintf(error_msg, "Did not get expected number of modules: %d\n",
            (int)pl->num_modules);

    ck_assert_msg(pl->num_modules == expected_num_modules, error_msg);

    PipelineModuleTypes expected_modules[] = { PM_SMILE_AND_KEYSTONE,
                                               PM_COMPRESSION,
                                               PM_TARGET_DETECTION,
                                               PM_DIMENSIONALITY_REDUCTION,
                                               PM_CLASSIFICATION_SOM,
                                               PM_CLASSIFICATION_SVM };

    for (int i = 0; i < expected_num_modules; ++i)
    {
        const char* expected_mod_string =
            PL_utils_module_to_string(expected_modules[i]);
        const char* actual_mod_string =
            PL_utils_module_to_string(pl->modules[i].module_type);

        ck_assert_msg(expected_mod_string != NULL, "Could not get module "
                                                   "name\n");
        ck_assert_msg(actual_mod_string != NULL, "Could not get module name\n");

        sprintf(error_msg, "Expected module %s (%d), got %s (%d)\n",
                expected_mod_string, expected_modules[i], actual_mod_string,
                pl->modules[i].module_type);

        ck_assert_msg(expected_modules[i] == pl->modules[i].module_type,
                      error_msg);
    }

    PL_builder_free_pipeline(pl);
}
END_TEST

Suite* make_pl_builder_suite(void)
{
    Suite* suite = suite_create("pl_builder");
    TCase* tcase = tcase_create("case");
    /* Tests go here */
    tcase_add_test(tcase, empty_pipeline);
    tcase_add_test(tcase, add_module);
    tcase_add_test(tcase, pipeline_size);
    tcase_add_test(tcase, serialize_pipeline);
    tcase_add_test(tcase, test_pl_config_reader);

    suite_add_tcase(suite, tcase);
    return suite;
}
```

B.5 Dot product toy experiment

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

#define B 120
#define W 684
#define H 956

float vec_prod(float* a, float* b, size_t size)
{
    float res = 0;

    for (size_t i = 0; i < size; ++i)
    {
        res += a[i] * b[i];
    }

    return res;
}

int main()
{
    float* s = malloc(sizeof(float) * B);

    float* img = malloc(sizeof(float) * B * W * H);

    float* res = malloc(sizeof(float) * H * W);
    srand(time(NULL));

    if (img != NULL && s != NULL && res != NULL)
    {
        printf("Malloc OK\n");
    }
    else
    {
        printf("Malloc Failed\n");
        goto free;
    }

    for (size_t i = 0; i < B; ++i)
    {
        s[i] = rand() % 30000;
    }
    for (size_t i = 0; i < B * W * H; ++i)
    {
        img[i] = rand() % 30000;
    }

    clock_t start, end;
    start = clock();

    for (size_t i = 0; i < H * W; ++i)
    {
        res[i] = vec_prod(s, &img[i * B], B);
    }

    end = clock();

    double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC; // in seconds

    printf("fun() took %f seconds to execute \n", time_taken);

free:
    free(s);
    free(img);
    free(res);
    return 0;
}
```

B.6 Dhrystone Benchmark

dhry.h

```
/*
 ****************************************************************************
 *
 *                   "DHRYSTONE" Benchmark Program
 *                   -----------------------------
 *
 *  Version:    C, Version 2.1
 *
 *  File:       dhry.h (part 1 of 3)
 *
 *  Date:       May 25, 1988
 *
 *  Author:     Reinhold P. Weicker
 *
 */

/* Compiler and system dependent definitions: */

#ifndef TIME
#undef TIMES
#define TIMES
#endif
/* Use times(2) time function unless     */
/* explicitly defined otherwise          */

#ifdef MSC_CLOCK
```

```
#undef HZ
#undef TIMES
#include <time.h>
#define HZ CLK_TCK
#endif
/* Use Microsoft C hi-res clock */

#ifdef TIMES
#include <sys/times.h>
#include <sys/types.h>
/* for "times" */
#endif

#define Mic_secs_Per_Second 1000000.0
/* Berkeley UNIX C returns process times in seconds/HZ */

#ifdef NOSTRUCTASSIGN
#define structassign(d, s) memcpy(&(d), &(s), sizeof(d))
#else
#define structassign(d, s) d = s
#endif

#ifdef NOENUM
#define Ident_1 0
#define Ident_2 1
#define Ident_3 2
#define Ident_4 3
#define Ident_5 4
typedef int Enumeration;
#else
typedef enum { Ident_1, Ident_2, Ident_3, Ident_4, Ident_5 } Enumeration;
#endif
/* for boolean and enumeration types in Ada, Pascal */

/* General definitions: */

#include <stdio.h>
/* for strcpy, strcmp */

#define Null 0
/* Value of a Null pointer */
#define true 1
#define false 0

typedef int One_Thirty;
typedef int One_Fifty;
typedef char Capital_Letter;
typedef int Boolean;
typedef char Str_30[31];
typedef int Arr_1_Dim[50];
typedef int Arr_2_Dim[50][50];

typedef struct record {
  struct record *Ptr_Comp;
  Enumeration Discr;
  union {
    struct {
      Enumeration Enum_Comp;
      int Int_Comp;
      char Str_Comp[31];
    } var_1;
    struct {
      Enumeration E_Comp_2;
      char Str_2_Comp[31];
    } var_2;
    struct {
      char Ch_1_Comp;
      char Ch_2_Comp;
    } var_3;
  } variant;
} Rec_Type, *Rec_Pointer;
```

dhry_1.c

```
/*
 ****************************************************************************
 *
 *                   "DHRYSTONE" Benchmark Program
 *                   -----------------------------
 *
 *  Version:    C, Version 2.1
 *
 *  File:       dhry_1.c (part 2 of 3)
 *
 *  Date:       May 25, 1988
 *
 *  Author:     Reinhold P. Weicker
 *
 ****************************************************************************
 */

#include "dhry.h"

/* Global Variables: */

Rec_Pointer Ptr_Glob, Next_Ptr_Glob;
int Int_Glob;
Boolean Bool_Glob;
char Ch_1_Glob, Ch_2_Glob;
int Arr_1_Glob[50];
int Arr_2_Glob[50][50];

extern char *malloc();
Enumeration Func_1();
/* forward declaration necessary since Enumeration may not simply be int */

#ifndef REG
Boolean Reg = false;
#define REG
/* REG becomes defined as empty */
/* i.e. no register variables   */
#else
Boolean Reg = true;
#endif

/* variables for time measurement: */
```

```
#ifdef TIMES
struct tms time_info;
/* see library function "times" */
#define Too_Small_Time (2 * HZ)
/* Measurements should last at least about 2 seconds */
#endif
#ifdef TIME
extern long time();
/* see library function "time"  */
#define Too_Small_Time 2
/* Measurements should last at least 2 seconds */
#endif
#ifdef MSC_CLOCK
extern clock_t clock();
#define Too_Small_Time (2 * HZ)
#endif

long Begin_Time, End_Time, User_Time;
float Microseconds, Dhrystones_Per_Second;

/* end of variables for time measurement */

main()
/*****/

/* main program, corresponds to procedures        */
/* Main and Proc_0 in the Ada version             */
{
  One_Fifty Int_1_Loc;
  REG One_Fifty Int_2_Loc;
  One_Fifty Int_3_Loc;
  REG char Ch_Index;
  Enumeration Enum_Loc;
  Str_30 Str_1_Loc;
  Str_30 Str_2_Loc;
  REG int Run_Index;
  REG int Number_Of_Runs;

  /* Initializations */

  Next_Ptr_Glob = (Rec_Pointer)malloc(sizeof(Rec_Type));
  Ptr_Glob = (Rec_Pointer)malloc(sizeof(Rec_Type));

  Ptr_Glob->Ptr_Comp = Next_Ptr_Glob;
  Ptr_Glob->Discr = Ident_1;
  Ptr_Glob->variant.var_1.Enum_Comp = Ident_3;
  Ptr_Glob->variant.var_1.Int_Comp = 40;
  strcpy(Ptr_Glob->variant.var_1.Str_Comp, "DHRYSTONE PROGRAM, SOME STRING");
  strcpy(Str_1_Loc, "DHRYSTONE PROGRAM, 1'ST STRING");

  Arr_2_Glob[8][7] = 10;
  /* Was missing in published program. Without this statement,    */
  /* Arr_2_Glob [8][7] would have an undefined value.             */
  /* Warning: With 16-Bit processors and Number_Of_Runs > 32000,  */
  /* overflow may occur for this array element.                   */

  printf("\n");
  printf("Dhrystone Benchmark, Version 2.1 (Language: C)\n");
  printf("\n");
  if (Reg) {
    printf("Program compiled with 'register' attribute\n");
    printf("\n");
  } else {
    printf("Program compiled without 'register' attribute\n");
    printf("\n");
  }
  printf("Please give the number of runs through the benchmark: ");
  {
    int n;
    scanf("%d", &n);
    Number_Of_Runs = n;
  }
  printf("\n");

  printf("Execution starts, %d runs through Dhrystone\n", Number_Of_Runs);

  /***************/
  /* Start timer */
  /***************/

#ifdef TIMES
  times(&time_info);
  Begin_Time = (long)time_info.tms_utime;
#endif
#ifdef TIME
  Begin_Time = time((long *)0);
#endif
#ifdef MSC_CLOCK
  Begin_Time = clock();
#endif

  for (Run_Index = 1; Run_Index <= Number_Of_Runs; ++Run_Index) {

    Proc_5();
    Proc_4();
    /* Ch_1_Glob == 'A', Ch_2_Glob == 'B', Bool_Glob == true */
    Int_1_Loc = 2;
    Int_2_Loc = 3;
    strcpy(Str_2_Loc, "DHRYSTONE PROGRAM, 2'ND STRING");
    Enum_Loc = Ident_2;
    Bool_Glob = !Func_2(Str_1_Loc, Str_2_Loc);
    /* Bool_Glob == 1 */
    while (Int_1_Loc < Int_2_Loc) /* loop body executed once */
    {
      Int_3_Loc = 5 * Int_1_Loc - Int_2_Loc;
      /* Int_3_Loc == 7 */
      Proc_7(Int_1_Loc, Int_2_Loc, &Int_3_Loc);
      /* Int_3_Loc == 7 */
      Int_1_Loc += 1;
    } /* while */
    /* Int_1_Loc == 3, Int_2_Loc == 3, Int_3_Loc == 7 */
    Proc_8(Arr_1_Glob, Arr_2_Glob, Int_1_Loc, Int_3_Loc);
    /* Int_Glob == 5 */
    Proc_1(Ptr_Glob);
    for (Ch_Index = 'A'; Ch_Index <= Ch_2_Glob; ++Ch_Index)
    /* loop body executed twice */
    {
      if (Enum_Loc == Func_1(Ch_Index, 'C'))
      /* then, not executed */
      {
        Proc_6(Ident_1, &Enum_Loc);
        strcpy(Str_2_Loc, "DHRYSTONE PROGRAM, 3'RD STRING");
        Int_2_Loc = Run_Index;
        Int_Glob = Run_Index;
      }
    }
    /* Int_1_Loc == 3, Int_2_Loc == 3, Int_3_Loc == 7 */
    Int_2_Loc = Int_2_Loc * Int_1_Loc;
    Int_1_Loc = Int_2_Loc / Int_3_Loc;
    Int_2_Loc = 7 * (Int_2_Loc - Int_3_Loc) - Int_1_Loc;
    /* Int_1_Loc == 1, Int_2_Loc == 13, Int_3_Loc == 7 */
    Proc_2(&Int_1_Loc);
    /* Int_1_Loc == 5 */

  } /* loop "for Run_Index" */

  /**************/
  /* Stop timer */
  /**************/

#ifdef TIMES
  times(&time_info);
  End_Time = (long)time_info.tms_utime;
#endif
#ifdef TIME
  End_Time = time((long *)0);
#endif
#ifdef MSC_CLOCK
  End_Time = clock();
#endif

  printf("Execution ends\n");
  printf("\n");
  printf("Final values of the variables used in the benchmark:\n");
  printf("\n");
  printf("Int_Glob:            %d\n", Int_Glob);
  printf("        should be:   %d\n", 5);
  printf("Bool_Glob:           %d\n", Bool_Glob);
  printf("        should be:   %d\n", 1);
  printf("Ch_1_Glob:           %c\n", Ch_1_Glob);
  printf("        should be:   %c\n", 'A');
  printf("Ch_2_Glob:           %c\n", Ch_2_Glob);
  printf("        should be:   %c\n", 'B');
  printf("Arr_1_Glob[8]:       %d\n", Arr_1_Glob[8]);
  printf("        should be:   %d\n", 7);
  printf("Arr_2_Glob[8][7]:    %d\n", Arr_2_Glob[8][7]);
  printf("        should be:   Number_Of_Runs + 10\n");
  printf("Ptr_Glob->\n");
  printf("  Ptr_Comp:          %d\n", (int)Ptr_Glob->Ptr_Comp);
  printf("        should be:   (implementation-dependent)\n");
  printf("  Discr:             %d\n", Ptr_Glob->Discr);
  printf("        should be:   %d\n", 0);
  printf("  Enum_Comp:         %d\n", Ptr_Glob->variant.var_1.Enum_Comp);
  printf("        should be:   %d\n", 2);
  printf("  Int_Comp:          %d\n", Ptr_Glob->variant.var_1.Int_Comp);
  printf("        should be:   %d\n", 17);
  printf("  Str_Comp:          %s\n", Ptr_Glob->variant.var_1.Str_Comp);
  printf("        should be:   DHRYSTONE PROGRAM, SOME STRING\n");
  printf("Next_Ptr_Glob->\n");
  printf("  Ptr_Comp:          %d\n", (int)Next_Ptr_Glob->Ptr_Comp);
  printf("        should be:   (implementation-dependent), same as above\n");
  printf("  Discr:             %d\n", Next_Ptr_Glob->Discr);
  printf("        should be:   %d\n", 0);
  printf("  Enum_Comp:         %d\n", Next_Ptr_Glob->variant.var_1.Enum_Comp);
  printf("        should be:   %d\n", 1);
  printf("  Int_Comp:          %d\n", Next_Ptr_Glob->variant.var_1.Int_Comp);
  printf("        should be:   %d\n", 18);
  printf("  Str_Comp:          %s\n", Next_Ptr_Glob->variant.var_1.Str_Comp);
  printf("        should be:   DHRYSTONE PROGRAM, SOME STRING\n");
  printf("Int_1_Loc:           %d\n", Int_1_Loc);
  printf("        should be:   %d\n", 5);
  printf("Int_2_Loc:           %d\n", Int_2_Loc);
  printf("        should be:   %d\n", 13);
  printf("Int_3_Loc:           %d\n", Int_3_Loc);
  printf("        should be:   %d\n", 7);
  printf("Enum_Loc:            %d\n", Enum_Loc);
  printf("        should be:   %d\n", 1);
  printf("Str_1_Loc:           %s\n", Str_1_Loc);
  printf("        should be:   DHRYSTONE PROGRAM, 1'ST STRING\n");
  printf("Str_2_Loc:           %s\n", Str_2_Loc);
  printf("        should be:   DHRYSTONE PROGRAM, 2'ND STRING\n");
  printf("\n");

  User_Time = End_Time - Begin_Time;

  if (User_Time < Too_Small_Time) {
    printf("Measured time too small to obtain meaningful results\n");
    printf("Please increase number of runs\n");
    printf("\n");
  } else {
#ifdef TIME
    Microseconds =
        (float)User_Time * Mic_secs_Per_Second / (float)Number_Of_Runs;
    Dhrystones_Per_Second = (float)Number_Of_Runs / (float)User_Time;
#else
    Microseconds = (float)User_Time * Mic_secs_Per_Second /
                   ((float)HZ * ((float)Number_Of_Runs));
    Dhrystones_Per_Second =
        ((float)HZ * (float)Number_Of_Runs) / (float)User_Time;
#endif
    printf("Microseconds for one run through Dhrystone: ");
    printf("%6.1f \n", Microseconds);
    printf("Dhrystones per Second:                      ");
    printf("%6.1f \n", Dhrystones_Per_Second);
    printf("\n");
  }
}

Proc_1(Ptr_Val_Par)
    /******************/
    REG Rec_Pointer Ptr_Val_Par;
/* executed once */
{
  REG Rec_Pointer Next_Record = Ptr_Val_Par->Ptr_Comp;
  /* == Ptr_Glob_Next */
  /* Local variable, initialized with Ptr_Val_Par->Ptr_Comp,    */
  /* corresponds to "rename" in Ada, "with" in Pascal           */

  structassign(*Ptr_Val_Par->Ptr_Comp, *Ptr_Glob);
  Ptr_Val_Par->variant.var_1.Int_Comp = 5;
  Next_Record->variant.var_1.Int_Comp = Ptr_Val_Par->variant.var_1.Int_Comp;
  Next_Record->Ptr_Comp = Ptr_Val_Par->Ptr_Comp;
  Proc_3(&Next_Record->Ptr_Comp);
  /* Ptr_Val_Par->Ptr_Comp->Ptr_Comp
                      == Ptr_Glob->Ptr_Comp */
  if (Next_Record->Discr == Ident_1)
  /* then, executed */
  {
    Next_Record->variant.var_1.Int_Comp = 6;
    Proc_6(Ptr_Val_Par->variant.var_1.Enum_Comp,
           &Next_Record->variant.var_1.Enum_Comp);
    Next_Record->Ptr_Comp = Ptr_Glob->Ptr_Comp;
    Proc_7(Next_Record->variant.var_1.Int_Comp, 10,
           &Next_Record->variant.var_1.Int_Comp);
  } else /* not executed */
    structassign(*Ptr_Val_Par, *Ptr_Val_Par->Ptr_Comp);
} /* Proc_1 */

Proc_2(Int_Par_Ref)
    /******************/
    /* executed once */
    /* *Int_Par_Ref == 1, becomes 4 */

    One_Fifty *Int_Par_Ref;
{
  One_Fifty Int_Loc;
  Enumeration Enum_Loc;

  Int_Loc = *Int_Par_Ref + 10;
  do /* executed once */
    if (Ch_1_Glob == 'A')
    /* then, executed */
    {
      Int_Loc -= 1;
      *Int_Par_Ref = Int_Loc - Int_Glob;
      Enum_Loc = Ident_1;
    }                           /* if */
  while (Enum_Loc != Ident_1); /* true */
} /* Proc_2 */

Proc_3(Ptr_Ref_Par)
    /******************/
    /* executed once */
    /* Ptr_Ref_Par becomes Ptr_Glob */

    Rec_Pointer *Ptr_Ref_Par;

{
  if (Ptr_Glob != Null)
    /* then, executed */
    *Ptr_Ref_Par = Ptr_Glob->Ptr_Comp;
  Proc_7(10, Int_Glob, &Ptr_Glob->variant.var_1.Int_Comp);
} /* Proc_3 */

Proc_4() /* without parameters */
/*******/
/* executed once */
{
  Boolean Bool_Loc;

  Bool_Loc = Ch_1_Glob == 'A';
  Bool_Glob = Bool_Loc | Bool_Glob;
  Ch_2_Glob = 'B';
} /* Proc_4 */

Proc_5() /* without parameters */
/*******/
/* executed once */
{
  Ch_1_Glob = 'A';
  Bool_Glob = false;
} /* Proc_5 */

/* Procedure for the assignment of structures,          */
/* if the C compiler doesn't support this feature       */
#ifdef NOSTRUCTASSIGN
memcpy(d, s, l) register char *d;
register char *s;
register int l;
{
  while (l--)
    *d++ = *s++;
}
#endif
```

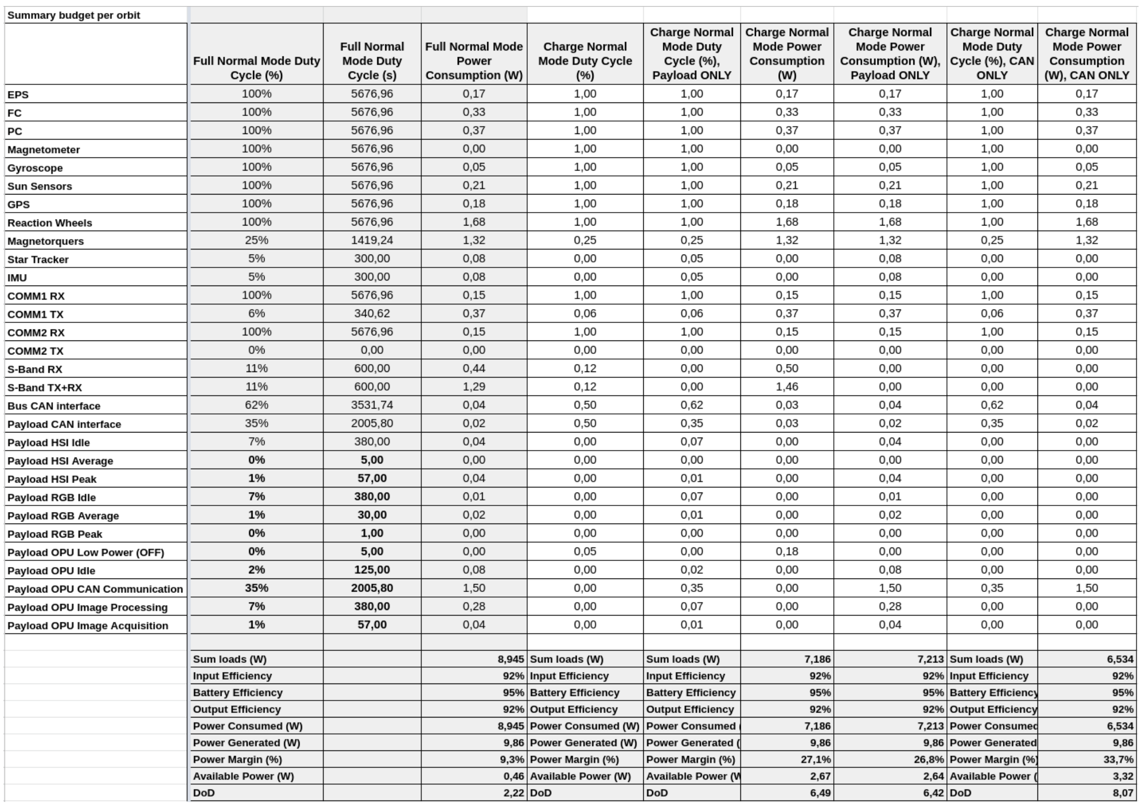
dhry_2.c

```
/*
 ****************************************************************************
 *
 *                   "DHRYSTONE" Benchmark Program
 *                   -----------------------------
 *
 *  Version:    C, Version 2.1
 *
 *  File:       dhry_2.c (part 3 of 3)
 *
 *  Date:       May 25, 1988
 *
 *  Author:     Reinhold P. Weicker
 *
 ****************************************************************************
 */

#include "dhry.h"
```

```
#ifndef REG
#define REG
        /* REG becomes defined as empty */
        /* i.e. no register variables   */
#endif

extern  int     Int_Glob;
extern  char    Ch_1_Glob;


Proc_6 (Enum_Val_Par, Enum_Ref_Par)
/*********************************/
    /* executed once */
    /* Enum_Val_Par == Ident_3, Enum_Ref_Par becomes Ident_2 */

Enumeration  Enum_Val_Par;
Enumeration *Enum_Ref_Par;
{
  *Enum_Ref_Par = Enum_Val_Par;
  if (! Func_3 (Enum_Val_Par))
    /* then, not executed */
    *Enum_Ref_Par = Ident_4;
  switch (Enum_Val_Par)
  {
    case Ident_1:
      *Enum_Ref_Par = Ident_1;
      break;
    case Ident_2:
      if (Int_Glob > 100)
        /* then */
      *Enum_Ref_Par = Ident_1;
      else *Enum_Ref_Par = Ident_4;
      break;
    case Ident_3: /* executed */
      *Enum_Ref_Par = Ident_2;
      break;
    case Ident_4: break;
    case Ident_5:
      *Enum_Ref_Par = Ident_3;
      break;
  } /* switch */
} /* Proc_6 */


Proc_7 (Int_1_Par_Val, Int_2_Par_Val, Int_Par_Ref)
/**********************************************/
    /* executed three times                                      */
    /* first call:      Int_1_Par_Val == 2, Int_2_Par_Val == 3,  */
    /*                  Int_Par_Ref becomes 7                    */
    /* second call:     Int_1_Par_Val == 10, Int_2_Par_Val == 5, */
    /*                  Int_Par_Ref becomes 17                   */
    /* third call:      Int_1_Par_Val == 6, Int_2_Par_Val == 10, */
    /*                  Int_Par_Ref becomes 18                   */
One_Fifty       Int_1_Par_Val;
One_Fifty       Int_2_Par_Val;
One_Fifty      *Int_Par_Ref;
{
  One_Fifty Int_Loc;
  Int_Loc = Int_1_Par_Val + 2;
  *Int_Par_Ref = Int_2_Par_Val + Int_Loc;
} /* Proc_7 */


Proc_8 (Arr_1_Par_Ref, Arr_2_Par_Ref, Int_1_Par_Val, Int_2_Par_Val)
/*********************************************************************/
    /* executed once      */
    /* Int_Par_Val_1 == 3 */
    /* Int_Par_Val_2 == 7 */
Arr_1_Dim       Arr_1_Par_Ref;
Arr_2_Dim       Arr_2_Par_Ref;
int             Int_1_Par_Val;
int             Int_2_Par_Val;
{
  REG One_Fifty Int_Index;
  REG One_Fifty Int_Loc;

  Int_Loc = Int_1_Par_Val + 5;
  Arr_1_Par_Ref [Int_Loc] = Int_2_Par_Val;
  Arr_1_Par_Ref [Int_Loc+1] = Arr_1_Par_Ref [Int_Loc];
  Arr_1_Par_Ref [Int_Loc+30] = Int_Loc;
  for (Int_Index = Int_Loc; Int_Index <= Int_Loc+1; ++Int_Index)
    Arr_2_Par_Ref [Int_Loc] [Int_Index] = Int_Loc;
  Arr_2_Par_Ref [Int_Loc] [Int_Loc-1] += 1;
  Arr_2_Par_Ref [Int_Loc+20] [Int_Loc] = Arr_1_Par_Ref [Int_Loc];
  Int_Glob = 5;
} /* Proc_8 */


Enumeration Func_1 (Ch_1_Par_Val, Ch_2_Par_Val)
/*************************************************/
    /* executed three times                                         */
    /* first call:      Ch_1_Par_Val == 'H', Ch_2_Par_Val == 'R'    */
    /* second call:     Ch_1_Par_Val == 'A', Ch_2_Par_Val == 'C'    */
    /* third call:      Ch_1_Par_Val == 'B', Ch_2_Par_Val == 'C'    */

Capital_Letter   Ch_1_Par_Val;
Capital_Letter   Ch_2_Par_Val;
{
  Capital_Letter        Ch_1_Loc;
  Capital_Letter        Ch_2_Loc;

  Ch_1_Loc = Ch_1_Par_Val;
  Ch_2_Loc = Ch_1_Loc;
  if (Ch_2_Loc != Ch_2_Par_Val)
    /* then, executed */
    return (Ident_1);
  else  /* not executed */
  {
    Ch_1_Glob = Ch_1_Loc;
    return (Ident_2);
   }
} /* Func_1 */


Boolean Func_2 (Str_1_Par_Ref, Str_2_Par_Ref)
```

```
/*************************************************/
    /* executed once */
    /* Str_1_Par_Ref == "DHRYSTONE PROGRAM, 1'ST STRING" */
    /* Str_2_Par_Ref == "DHRYSTONE PROGRAM, 2'ND STRING" */

Str_30  Str_1_Par_Ref;
Str_30  Str_2_Par_Ref;
{
  REG One_Thirty        Int_Loc;
      Capital_Letter    Ch_Loc;

  Int_Loc = 2;
  while (Int_Loc <= 2) /* loop body executed once */
    if (Func_1 (Str_1_Par_Ref[Int_Loc],
                Str_2_Par_Ref[Int_Loc+1]) == Ident_1)
      /* then, executed */
    {
      Ch_Loc = 'A';
      Int_Loc += 1;
    } /* if, while */
  if (Ch_Loc >= 'W' && Ch_Loc < 'Z')
    /* then, not executed */
    Int_Loc = 7;
  if (Ch_Loc == 'R')
    /* then, not executed */
    return (true);
  else /* executed */
  {
    if (strcmp (Str_1_Par_Ref, Str_2_Par_Ref) > 0)
      /* then, not executed */
    {
      Int_Loc += 7;
      Int_Glob = Int_Loc;
      return (true);
    }
    else /* executed */
      return (false);
  } /* if Ch_Loc */
} /* Func_2 */


Boolean Func_3 (Enum_Par_Val)
/***************************/
    /* executed once        */
    /* Enum_Par_Val == Ident_3 */
Enumeration Enum_Par_Val;
{
  Enumeration Enum_Loc;

  Enum_Loc = Enum_Par_Val;
  if (Enum_Loc == Ident_3)
    /* then, executed */
    return (true);
  else /* not executed */
    return (false);
} /* Func_3 */
```

C Power Budget



Summary budget per orbit									
	Full Normal Mode Duty Cycle (%)	Full Normal Mode Duty Cycle (s)	Full Normal Mode Power Consumption (W)	Charge Normal Mode Duty Cycle (%)	Charge Normal Mode Duty Cycle (%), Payload ONLY	Charge Normal Mode Power Consumption (W)	Charge Normal Mode Power Consumption (W), Payload ONLY	Charge Normal Mode Duty Cycle (%), CAN ONLY	Charge Normal Mode Power Consumption (W), CAN ONLY
EPS	100%	5676,96	0,17	1,00	1,00	0,17	0,17	1,00	0,17
FC	100%	5676,96	0,33	1,00	1,00	0,33	0,33	1,00	0,33
PC	100%	5676,96	0,37	1,00	1,00	0,37	0,37	1,00	0,37
Magnetometer	100%	5676,96	0,00	1,00	1,00	0,00	0,00	1,00	0,00
Gyroscope	100%	5676,96	0,05	1,00	1,00	0,05	0,05	1,00	0,05
Sun Sensors	100%	5676,96	0,21	1,00	1,00	0,21	0,21	1,00	0,21
GPS	100%	5676,96	0,18	1,00	1,00	0,18	0,18	1,00	0,18
Reaction Wheels	100%	5676,96	1,68	1,00	1,00	1,68	1,68	1,00	1,68
Magnetorquers	25%	1419,24	1,32	0,25	0,25	1,32	1,32	0,25	1,32
Star Tracker	5%	300,00	0,08	0,00	0,05	0,00	0,08	0,00	0,00
IMU	5%	300,00	0,08	0,00	0,05	0,00	0,08	0,00	0,00
COMM1 RX	100%	5676,96	0,15	1,00	1,00	0,15	0,15	1,00	0,15
COMM1 TX	6%	340,62	0,37	0,06	0,06	0,37	0,37	0,06	0,37
COMM2 RX	100%	5676,96	0,15	1,00	1,00	0,15	0,15	1,00	0,15
COMM2 TX	0%	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
S-Band RX	11%	600,00	0,44	0,12	0,00	0,50	0,00	0,00	0,00
S-Band TX+RX	11%	600,00	1,29	0,12	0,00	1,46	0,00	0,00	0,00
Bus CAN interface	62%	3531,74	0,04	0,50	0,62	0,03	0,04	0,62	0,04
Payload CAN interface	35%	2005,80	0,02	0,50	0,35	0,03	0,02	0,35	0,02
Payload HSI Idle	7%	380,00	0,04	0,00	0,07	0,00	0,04	0,00	0,00
Payload HSI Average	**0%**	**5,00**	0,00	0,00	0,00	0,00	0,00	0,00	0,00
Payload HSI Peak	**1%**	**57,00**	0,04	0,00	0,01	0,00	0,04	0,00	0,00
Payload RGB Idle	**7%**	**380,00**	0,01	0,00	0,07	0,00	0,01	0,00	0,00
Payload RGB Average	**1%**	**30,00**	0,02	0,00	0,01	0,00	0,02	0,00	0,00
Payload RGB Peak	**0%**	**1,00**	0,00	0,00	0,00	0,00	0,00	0,00	0,00
Payload OPU Low Power (OFF)	**0%**	**5,00**	0,00	0,05	0,00	0,18	0,00	0,00	0,00
Payload OPU Idle	**2%**	**125,00**	0,08	0,00	0,02	0,00	0,08	0,00	0,00
Payload OPU CAN Communication	**35%**	**2005,80**	1,50	0,00	0,35	0,00	1,50	0,35	1,50
Payload OPU Image Processing	**7%**	**380,00**	0,28	0,00	0,07	0,00	0,28	0,00	0,00
Payload OPU Image Acquisition	**1%**	**57,00**	0,04	0,00	0,01	0,00	0,04	0,00	0,00
	Sum loads (W)		8,945	Sum loads (W)	Sum loads (W)	7,186	7,213	Sum loads (W)	6,534
	Input Efficiency		92%	Input Efficiency	Input Efficiency	92%	92%	Input Efficiency	92%
	Battery Efficiency		95%	Battery Efficiency	Battery Efficiency	95%	95%	Battery Efficiency	95%
	Output Efficiency		92%	Output Efficiency	Output Efficiency	92%	92%	Output Efficiency	92%
	Power Consumed (W)		8,945	Power Consumed (W)	Power Consumed (	7,186	7,213	Power Consumed	6,534
	Power Generated (W)		9,86	Power Generated (W)	Power Generated (	9,86	9,86	Power Generated	9,86
	Power Margin (%)		9,3%	Power Margin (%)	Power Margin (%)	27,1%	26,8%	Power Margin (%)	33,7%
	Available Power (W)		0,46	Available Power (W)	Available Power (W	2,67	2,64	Available Power (	3,32
	DoD		2,22	DoD	DoD	6,49	6,42	DoD	8,07

Figure 25: Power Budget for HYPSO

Aksel Danielsen

Clustering and Classification of hyperspectral images on the HYPSO CubeSat