SAPIENZA
UNIVERSITÀ DI ROMA

DEPARTMENT OF INFORMATION ENGINEERING, ELECTRONICS AND TELECOMMUNICATIONS

PHD THESIS
IN
INFORMATION AND COMMUNICATIONS TECHNOLOGIES

Design of a Multi-Agent Classification System

Supervisor:
Prof. Antonello Rizzi

Reviewers:
Prof. Salima Hassas
Prof. Sean Luke

PhD Student:
Mauro Giampieri[1]
Serial Number 1191355

A. Y. 2018 - 2019

[1]mauro.giampieri@uniroma1.it

“Torture numbers and they will confess to anything”
— Gregg Easterbrook

Contents

1 Introduction **3**

2 Requirements Elicitation **15**
2.1 High-level Data Set Properties . . . 16
2.1.1 High Data Set Cardinality . . . 16
2.1.2 Uneven Features . . . 18
2.1.3 Feature Selection and Local Metric Learning . . . 20

3 Algorithms Design **25**
3.1 Algorithm Properties . . . 25
3.2 Multi-Agent Framing . . . 29
3.3 Clustering: E-ABC . . . 30
3.3.1 Agent Behavior . . . 30
3.3.2 Inter-Agent Fusions . . . 38
3.3.3 Evolutionary Procedure . . . 39
3.3.4 E-ABC Main Loop . . . 44
3.4 Classification: E-ABC^2 . . . 47
3.4.1 Classification Model: Decision Clusters . . . 48
3.4.2 Pattern Classification Procedure . . . 49
3.4.3 Model Synthesis . . . 50
3.4.4 Parallel Implementation . . . 65

4 Preliminary Results **67**
4.1 E-ABC . . . 67
4.1.1 Data Set Description . . . 68
4.1.2 Evaluation Metrics . . . 70
4.1.3 Tests Results . . . 71
4.2 E-ABC^2 . . . 73
4.2.1 Evaluation Metrics . . . 74
4.2.2 Test On Iris Data Set . . . 75
4.2.3 Test On Acea Data Set . . . 78

5 Experimental Results **87**
5.1 Data Set Description . . . 87
5.2 Evaluation Metrics . . . 89
5.3 Vanilla E-ABC^2 . . . 90
5.4 Fitness Sharing . . . 98
5.5 Model Pruning . . . 106
5.6 Multimodal E-ABC^2 Scalability . . . 110

6 Conclusions **115**

A Software Design **119**

B Classes Declarations **123**
B.1 E-ABC^2 . . . 123
B.2 Hyperpoint . . . 127
B.3 RL-BSAS . . . 129

Bibliography **132**

Abstract

In the era of information, emergent paradigms such as the Internet of Things with billions of devices constantly connected to the Internet exchanging heterogeneous data, demand new computing approaches for intelligent big data processing. Given this technological scenario, a suitable approach leverages on many computational entities, each of which performs tasks with low computational burden, conceived to be executed on multi-core and many-core architectures.

To this aim, in this thesis, we propose Evolutive Agent Based Clustering (E-ABC) as promising framing reference. E-ABC is conceived to orchestrate a swarm of intelligent agents acting as individuals of an evolving population, each performing a random walk on a different subset of patterns. Each agent is in charge of discovering well-formed (compact and populated) clusters and, at the same time, a suitable subset of features corresponding to the subspace where such clusters lie, following a local metric learning approach, where each cluster is characterized by its own subset of relevant features. E-ABC is able to process data belonging to structured and possibly non-metric spaces, relying on custom parametric dissimilarity measures.

Specifically, two variants are investigated. A first variant, namely E-

ABC, aims at solving unsupervised problems, where agents' task is to find well-formed clusters lying in suitable subspaces. A second variant, E-ABC^2, aims at solving classification problems by synthesizing a classification system on the top of the clusters discovered by the swarm.

In particular, as a practical and real-world application, this novel classification system has been employed for recognizing and predicting localized faults on the electric distribution network of Rome, managed by the Italian utility company ACEA. Tests results show that E-ABC is able to synthesize classification models characterized by a remarkable generalization capability, with adequate performances to be employed in Smart Grids condition-based management systems. Moreover, the feature subsets where most of the meaningful clusters have been discovered can be used to better understand sub-classes of failures, each identified by a set of related causes.

Chapter 1

Introduction

During the last decades, the Information and Communication Technology area (also known as ICT) has experienced an enormous growth. Just think about the Moore law [1] which has driven the computer industry development between mid 1960s and late 2010s. For about half a century integrated circuits complexity (in terms of transistor density) has doubled each 18 months. This has brought from 66 MHz single core – single thread CPU, 8 MB RAM, 1 GB hard-drive personal computer at half 1990s, to 3.7 GHz 6 cores – 12 threads CPU, 32 GB RAM, 4 TB hard-drive personal computer at the second half of 2010s (some raw data just to have an hint without taking into account the actual detailed performance) for about the same discounted price. Performance improvement is not the only consequence of this process. The miniaturization itself has opened up lots of new possibilities. It has dramatically drop down the price for lower performance electronics making it possible to have a fully functional computer in a single 30 by 65 mm board, whose power consumption is lower than 2 W, for about 5 € as it is the case

for Raspberry Pi Zero (Figure 1.1). This has encouraged the spread of this

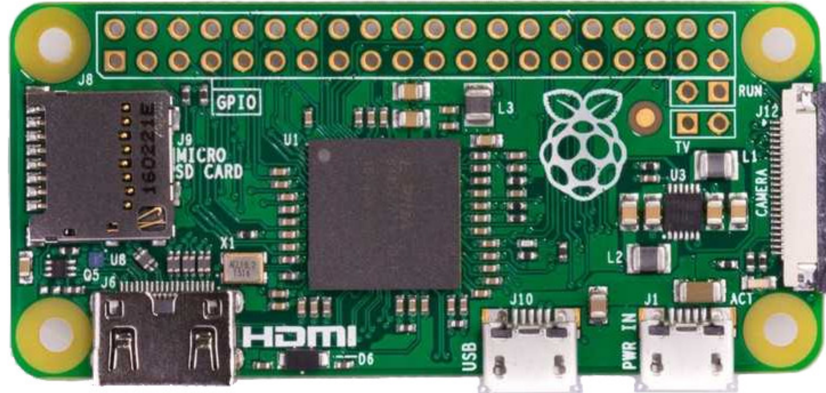

Figure 1.1: Photo shot of a Raspberry Pi Zero.

kind of devices. Just think about the success and spread of smartphones at the consumer level and sensors and robotics in industrial environment. A huge amount of data is poured every day into the Internet by end-users through social networks, bots analyzing the Internet itself, sensor networks and the introduction of 5G networks [2] will give a further boost to interconnectivity and sensorization. The direction traced is that of the Internet of Things (IoT) where all electric devices will have a direct connection with the network and they will pour an always growing amount of sensored data into the Internet. All of these data are nowadays universally known to be an economical resource for user profiling, diagnostics and prediction in the most diverse fields of application, but they will be unusable if we will not have efficient and effective tools to extract intelligible information from them.

In the last decade, machine learning and computational intelligence emerged as breakthrough disciplines in order to analyze and extract knowledge from data [3, 4, 5]. Machine learning algorithms are usually based on biologically-inspired concepts and are able to learn from data and experience without

being explicitly programmed to do so [3, 6]. In other words, given a set of input–output pairs pertaining to suitable domain and codomain, machine learning algorithms are able to learning the underlying mapping of the process that generated these input–output pairs.

Broadly speaking, machine learning algorithms can be divided in three main families, depending on the nature of the input space and the nature of the output space:

- classification: in classification problems, the output space contains a suitable finite set of problem-related classes pertaining to a categorical domain. For example, one might train a classification system in order to distinguish between 'normal' or 'abnormal', given the state of a physical system

- function approximation: in function approximation problems such as regression, the output space is a normed space, usually $\mathbb{R}$, and aim of the learning system is to estimate as accurately as possible the output value for a given input pattern. For example, one might train a function approximation system in order to predict tomorrow's temperature or chance of raining.

- clustering: in clustering problems there are no output values and regularities have to be discovered only by considering mutual relationships between patterns lying in the input space.

In the literature, problems like classification and function approximation are usually referred to as *supervised learning* problems since the learning system

can rely on ground-truth values, whereas clustering problems usually fall under the *unsupervised learning* umbrella. However, as will be thoroughly stressed during this thesis, clustering and classification shall not be considered as two diametrically opposed techniques and might as well cooperate. Indeed, the so-called *decision clusters* are well-known in the literature: according to the decision clusters model, one can cluster the input space without considering any output value and then infer a labelling over the resulting clusters. Every new pattern can be classified according to the nearest cluster, or the K nearest clusters or using a fuzzy assignment procedure [7, 8, 9].

Specifically, all clustering algorithms such as k-means [10], DBSCAN [11] and OPTICS [12] share the same approach of finding groups (clusters) of similar data according to some predefined criteria (density, homogeneity and the like) and according to a given dissimilarity measure which, as its name suggests, quantifies the distance between any two patterns. Figure 1.2 sketches one of the possible outcomes for a clustering problem.

The employment of a classification system can be divided in two independent phases: training and usage. During the training phase, a classification algorithm observes pattern–class label pairs drawn from the data set under analysis and exploits them to synthesize a classification model. To simplify, the classification model is a domain space segmentation where each area is associated with a class label. Once the model is synthesized, it can be used to predict and assign a class label, out of the given set of classes, to unknown elements based on its position in the domain space. Figure 1.3 sketches the decision boundary for a binary classification problem, whereas Figure 1.4 shows the same problem, but likely solved by the aforementioned decision

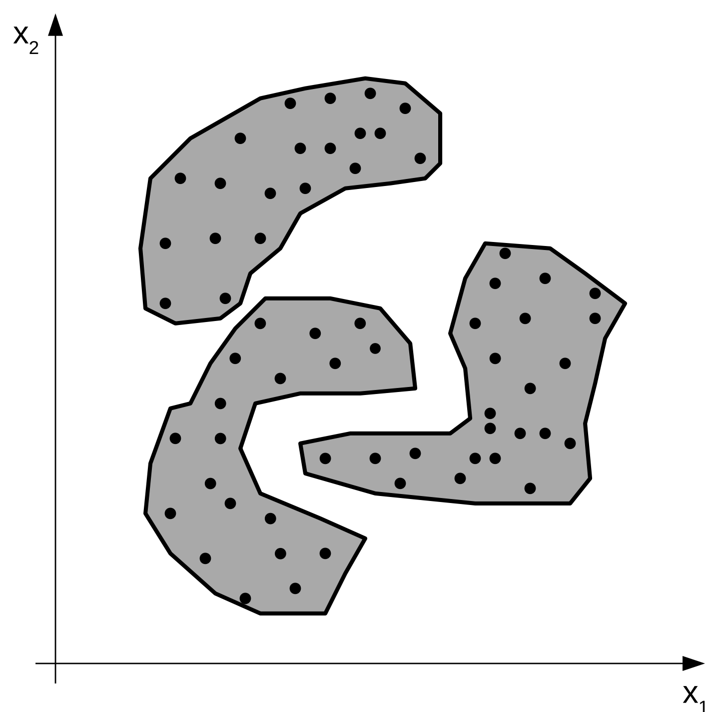

Figure 1.2: A possible clustering solution.

clusters approach.

In the recent years, the multiple agent paradigm emerged in order to perform both supervised and unsupervised learning [13, 14, 15] and has a tight link to the well-established reinforcement learning [16, 17, 18, 19] and is rapidly expanding towards deep reinforcement learning [20, 21, 22]. According to the multi-agent framework, a set of elementary units called *agents* operate in either synchronous or asynchronous matter in order to process the data set at hand. Its intrinsic setup makes this paradigm very appealing in big data contexts, since agents usually operate independently one another and can be embarrassingly parallelized across several computational units [20, 23].

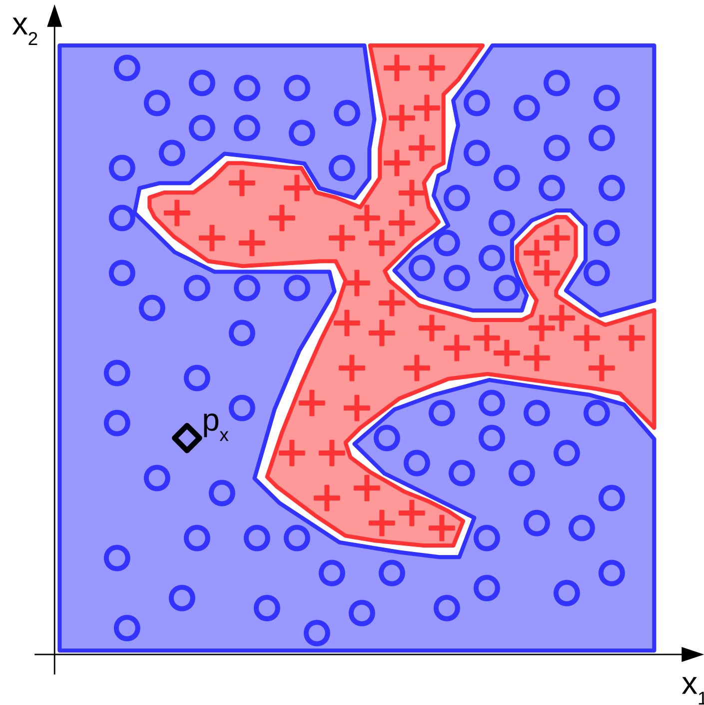

Figure 1.3: Decision boundary for a binary classification problem.

For example, as clustering procedures are concerned, in [24] a multi-agent approach has been used for local graph clustering in which each agent performs a random walk on a graph with the main constraint that such agents are "tied" together by a rope, forcing them to be close to each other. In [25] a set of self-organising agents by means of Ant Colony Optimisation has been applied to anomaly detection and network control. In [26] each agent runs a different clustering algorithm in order to return the best one for the data set at hand. In [27] agents negotiate one another rather than being governed by a master/wrapper process (e.g. evolutive algorithm). In [28] ant colony optimisation has been used in order to organise agents, where each ant "walks" on the data set, building connections amongst points. In

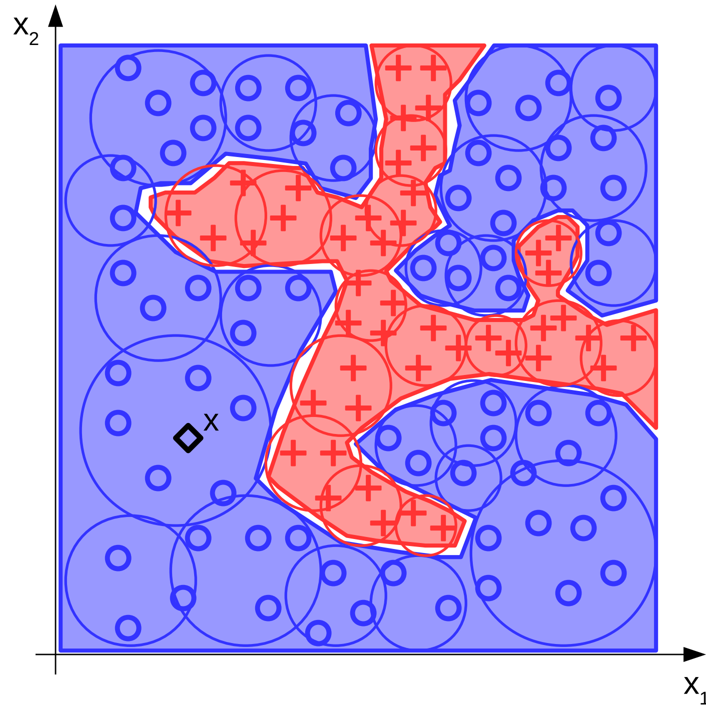

Figure 1.4: Decision boundary for the same classification problem, solved by means of decision clusters.

[29] each agent consists in a set of data points and agents link to each other, thus leading to clustering. In [30] a genetic algorithm has been used where the agents' genetic code is connection-based: each agent is a clustering result whose genetic code builds a (sub)graph and, finally, such subgraphs can be interpreted as clusters. In [31] the multi-agent approach collapses into two agents: a first agent runs a cascade of principal component analysis, self organizing maps and k-means in order to cluster data and a second agent validates such results: the two agents interactively communicate with each other. Finally, in [32] a multi-agent algorithm has been proposed in which agents perform a Markovian random walk on a weighted graph representation of the input data set. Each agent builds its own graph connection matrix

amongst data points, weighting the edges according to the selected distance measure parameters, and performs a random walk on such graph in order to discover clusters. This algorithm has been employed in [33] to identify frequent behaviours of mobile network subscribers starting from a set of call data records.

An interesting aspect of clustering techniques is that they allow synthesizing data-driven models of real-world phenomena acting as white boxes: indeed, it is quite easy to analyze clusters' contents. This kind of knowledge discovery is of paramount importance for specific applications, in which field-experts (personnel with likely no expertise in machine learning) may gather further insights on the modelled system. Examples of real-world applications in which this kind of knowledge discovery is crucial include predictive medicine [9], device profiling and anomaly detection in IoT networks [34] and the failure prediction in modern Smart Grids [35] equipped with smart sensors and powerful data centers.

In fact, the availability of smarter and low-cost technology in the power grid industry is changing forever the way to approach the underlying problems arising in the world-wide power network. Following this line, the Smart Grids concept is becoming a reality. It is the case of the Advanced Metering Infrastructure (AMI) in Smart Grids with the capability to facilitate two-way communication between a "smart" meter and the grid operator's control center, as well as between the smart meter and consumer appliances [36, 37]. Hence, ICTs are providing the enabling technologies to transform the actual power grid in the future Smart Grids. The communication infrastructure equipped with suitable protocols stack together with Smart Sensors

scattered in the power grid and suitable data centers allow collecting and processing a huge amount of data. The nature of data that can be extracted is highly heterogeneous and depends also on the application. By the way, the lowering of the costs of smart sensors and the increase in the transport capacity of the communication network together with ever high-performance computation machines allow collecting raw data related to both: the power grid status and the environment where the power grid is immersed. Power grid and environment data can be integrated in order to extract, through Knowledge Discovery from Data (KDD) mechanisms and Computational Intelligence (CI) techniques, useful information for the managing of the power network infrastructure. It is the case of Condition Based Maintenance (CBM) programs that can be automated and rationalized using a Decision Support System (DSS) capable of carrying on a decision-making process based on data collected from Smart Sensors and related to faults and outages that can happen during the normal grid operations. The current study born from the need of synthesizing an ad-hoc DSS for CBM capable of mining on fault states within the real-world power grid managed by Azienda Comunale Energia Ambiente (ACEA), the company in charge of managing the electric power grid of Rome, Italy. It is well-known that the quick and accurate diagnosis and restoration of power grid faults are extremely important for keeping the power grid operational and minimizing losses caused by power failure. Moreover, for a power grid early warning system, evaluating, warning against, diagnosing and automatically controlling faults to avoid hidden troubles or limit fault-related losses to the lowest level is critical to ensure the healthy and safe service of the power grid [38, 39].

We can define a DSS as an expert system that provides decision support for the commanding and dispatching systems of the power grid. Such a system analyzes the risk for damage of crucial types of equipment, assesses the power grid security, forecasts and provides warnings about the magnitude and location of possible faults, and timely broadcasts the early-warning signals through suitable communication networks [40]. The information provided by the DSS can be used also for CBM in the power grid [41].
CBM is defined as “a philosophy that posits repair or replacement decisions on the current or future condition of assets”. The objective of CBM is thus to minimize the total cost of inspection and repair by collecting and interpreting (heterogeneous) data related to the operating condition of critical components. Through the use of CBM, advanced smart sensor technologies have the potential to help utilities to improve the power grid reliability by avoiding unexpected outages. A discussion on how the changes in modern power grids have affected the maintenance procedures can be found in [42]; the importance of modern diagnostic techniques is treated in [43].

P. J. Werbos argues that the main paradigm around the “injection” of intelligence in the actual technology ecosystem is the *Computational System Thinking Machine* (CSTM) [44]. Through methods and techniques related to the Computational Intelligence (CI) framework and more in general to Machine Learning and Artificial Intelligence techniques, they allow large systems, such as Smart Grids, to carry out important functions through high-performance algorithms, such as advanced monitoring, forecasting, decision-making, control and optimization. These algorithms must be fast, scalable and dynamic. The main CI paradigms for Smart Grids related to problem so-

lutions are Neurofuzzy, Neuroswarm, Fuzzy-PSO, Fuzzy-GA, Neuro-Genetic [45]. Such paradigms often well developed in a theoretical framework with promising results needs further research and analysis effort in order to be considered useful in real-world applications. This is the case of clustering or classification techniques for fault recognition in a Smart Grids – the main application of this thesis – based on real-world data, coming from smart sensors, describing, through suitable features, the Smart Grid states, i.e. faults or standard functioning states. As we will see in the remainder of the thesis, real-world applications in Data Science, such as data classification and Data Mining can see many difficulties in applying standard procedures that are developed for vector-based feature space. Moreover, real applications rely so often on non-metric spaces in which standard Machine Learning procedure can fail. Hence, CI techniques hybridized with Soft Computing, data-driven modeling techniques and Data Mining algorithms have been widely adopted in the Smart Grid [46, 47, 48, 49]. On the other hand as a highly application-driven discipline, data mining has seen great successes in many applications [50].

The algorithm developed in this thesis constitutes a building block of an advanced DSS with the aim of predicting faults and outages occurring in the power grid of Rome offering even a modeling tool of failures. Fault states are described by a complex and heterogeneous pattern related to both the network devices (constitutive parameters) and the environmental conditions such as weather conditions, load, etc. (external causes). In fact, the state of the power grid consists of real-valued numbers, integers, categorical variables and unevenly time-spaced sequences of events. From the classifier point

of view, it is worth to note that the starting feature space is non-metric. Furthermore, the causes of a fault during the normal power grid operations can be heterogeneous and it is likely that it exists several sub-classes of fault and, for each one of them, a different set of features can be correlated with the fault occurrence. Moreover, in a large power grid feeding millions of users, the number of faults can be very large. For this purpose, a multi-agent clustering paradigm [51, 52] together with an evolutionary metric learning technique [53] is adopted, with the objective of discovering local sub-classes of faults stored in a huge real-world data set. Detailed requirements and considerations leading to our algorithm design are described in next chapter 2.

Chapter 2

Requirements Elicitation

As we said in chapter 1, the algorithm we have developed and we are going to describe and analyze in depth, takes its moves from the necessity to face data sets with specific properties. The actual data set which has inspired this algorithm is the one from Acea company about power grid faults. Acea is the company which is in charge of managing the electric power grid in Rome. Each single element of the data set (pattern) they were interested to deal with is an aggregation of uneven data which represents an operational condition of the power grid. Each of these is associated to a state regular functioning or fault situation occurred few time later. The target was to develop a tool to predict faults of the power grid. Each pattern contains information about geographical coordinates of the event, weather conditions and intrinsic grid properties where the event occurred. At this stage we are not interested in details about this data set (described in §4.2.3). Here we are only interested in some high-level properties of this data set, which are common to be found in real world data sets, so as to justify the development

of an algorithm to face them.

2.1 High-level Data Set Properties

In the following we shortly summarize the main Acea data set properties which have inspired the design of the algorithm described and developed in this thesis.

2.1.1 High Data Set Cardinality

From the starting of digital age, data sets availability has increased dramatically. That is likely because companies, noting information has a commercial value, have started to collect data through more and more sensor networks about their technological equipment and human users for profiling. At the same time, data sets cardinality has increased due to growing amount of information technologies users, posing additional challenges when it comes to synthesize machine learning models. In most clustering and classification algorithms, the most atomic task is pattern-to-pattern dissimilarity computation. For some clustering algorithms (including, but not limited to, k-medoids, DBSCAN and linkage clustering) a straightforward and time efficient solution would be to compute all possible pairwise dissimilarities and keep them in memory [54]. Let us suppose we have a dissimilarity function $d(\cdot,\cdot)$ and it is symmetric so that, given two patterns p_1 and p_2, we have

$$d(p_1, p_2) = d(p_2, p_1) \tag{2.1}$$

Table 2.1: Memory required to store pattern-to-pattern distances for all patterns in data set.

Patterns quantity	Memory occupation [GB]
126492	32
178886	64
252983	128

For a data set with n patterns the number of relevant elements to be stored

$$\frac{n \cdot (n-1)}{2} \tag{2.2}$$

would result in an asymptotic time and space complexity of

$$O(n^2) \tag{2.3}$$

For a practical analysis, this means that, under the operational hypothesis we use 32-bit floating point variables for distance storage, with a modern top consumer level computer we can deal with data sets containing no more than about 250000 patterns, as shown in table 2.1 (an Intel® Core™ i9-9900K supports at most 128 GB RAM) The opposite alternative is computing the dissimilarity between two patterns p_1, p_2 each time it is necessary. This requires no additional memory beyond what is necessary to store the data set itself. On the other hand, it requires additional time because $d(p_1, p_2)$ must be evaluated again each time it is needed.

2.1.2 Uneven Features

Real world patterns not always may be represented by means of a real valued vector where each component x_j (feature) of each pattern p_i is a real-valued scalar:

$$p_i = (x_1, x_2, ..., x_m), \quad x_j \in \mathbb{R}, \quad x_j \in [0, 1] \tag{2.4}$$

For example, this is a common scenario as heterogeneous (structured) patterns are concerned, where each feature may be a data type way more complex than a plain real valued scalar. For instance, they can be time series, texts, graphs, audio tracks, images, videos or whatever we are able to store in a digital data type. In information science all of these elements are represented by means of sequences of numbers, but can not be compared with a simple Euclidean distance. For example, time series will likely require Dynamic Time Warping; sequences will likely require an edit distance (e.g., the Levenshtein distance in case each atomic element of the sequence is a plain character); text mining and sentiment analysis usually rely on embedding techniques such as bag-of-words and TF-IDF or on neural approaches in order to consider also words semantic; how to compare graphs depends on their topology and edge/node properties, if present; audio tracks can again be treated as sequences or they can be analyzed in frequency domain; images comparison is strongly application dependent: many optical flow algorithms, such as Lucas-Kanade [55] or Brox [56], are available for motion detection, or feature extraction algorithms, such as SIFT [57] or SURF [58], may fit better if the task is object recognition or scene interpretation and reconstruction; for videos we have the same possibilities we have for images but, in addition,

we have to take into account they are sequences of images.

Data embedding is a common approach to face learning from complex data. Embedding methodologies map original data to real valued vectors to overcome the comparison problem. In this way patterns representations are obtained which can easily be compared by means of a simple Euclidean distance. However, designing this mapping function is a critical issue since it must fill the semantic gap between the two domains as much as possible. In any case, once a dissimilarity measure is given in every feature space, the overall dissimilarity measure can always be defined as a convex linear combination of the elementary dissimilarities, where weights can be learned during the training procedure, in order to maximize system performance and at the same time giving insights about the relative importance of each feature. Moreover, improved knowledge discovery capabilities can be obtained relying on algorithms yielding explicit clusters of patterns, since simple statistical representations of clusters (such as the average vector or variability ranges) can be easily interpreted in natural language, conversely to other black box models (e.g., artificial neural networks and support vector machines). Under this light, patterns are properly compared feature by feature in their original domain and the results are further normalized in $[0, 1]$ range in order to avoid implicit weighting of different components. These dissimilarities between homologous features are finally combined in order to obtain a final dissimilarity value.

2.1.3 Feature Selection and Local Metric Learning

As we have seen in §2.1.2, patterns may be constituted of many independent features, each one with its own data type. Not all of the features describing a pattern are strictly necessary to determine its membership to a given class. We may be interested in discovering which components are useful to discriminate different classes. In other words we want to determine which features are informative for a classification task and which ones can be dropped. This task is known in literature as metric learning. Field experts may be able to provide some knowledge about which features are supposed to be more relevant and which others are likely useless. Moreover, different classes may be visible in different subspaces, increasing the complexity of this task, which leads to the local metric learning. It is a more specific version of metric learning where different groups of classes result to be discernible in different subspaces. Metric learning and local metric learning help to simplify synthesized models, making them easier to be interpreted and understood by field experts.

Specifically, the *metric learning* problem is concerned with learning a (parametric) distance function tuned to a particular task and has been shown to be useful when used in conjunction with techniques that rely on distances or dissimilarities such as clustering algorithms, nearest-neighbor classifiers and the like [59]. For example, if the task is to asses the similarity (or dissimilarity) between two images with the aim of finding a match, for example in face recognition, we would discover a proper distance function that emphasizes appropriate features (hair color, ratios of distances between facial

key-points, etc). Although this task can be performed by hand, a tool for automatic learning important features capable to learn task-specific distance functions in a supervised manner can be assessed. Many declinations of metric learning are available, besides, according to Fig. 2.1, they can be resumed in three principal paradigms: *fully supervised*, *weakly supervised* and *semi supervised*. An informal formulation of the supervised metric learning task is as follows: given an input distance function $d(\vec{x}, \vec{y})$ between objects $\vec{x}$ and $\vec{y}$ (for example, the Euclidean distance), along with supervised information regarding an ideal distance, construct a new distance function $\hat{d}(\vec{x}, \vec{y})$ which is "better" than the original distance function [60]. Normally, fully supervised paradigms have access to a set of labeled training instances, whose labels are used to generate a set of constraints. In other words, supervised distance metric learning is cast into pairwise constraints: the equivalence constraints where pairs of data points belong to the same class, and inequivalence constraints where pairs of data points belong to different classes [61]. In weakly supervised learning algorithms we do not have access to the label of individual training data and learning constraints are given in a different form as side information, while semi-supervised paradigm does not use either labeled samples or side information. Some authors (e.g. Liu Yang in [61]) deals with unsupervised metric learning paradigms, sometimes called also *manifold learning* referring to the idea of learning an underlying low-dimensional manifold[1] where geometric relationships (e.g. distance) between most of the observed data are preserved. Often this paradigm co-

[1]A manifold is a topological space that resembles Euclidean space near each point. Hence a n-dimensional manifold has a neighborhood that is homeomorphic to the Euclidean space of dimension n.

incides with the *dimensionality reduction* paradigm of which examples are the well-known Principal Component Analysis [62] and the Classical Scaling (CS), that are based on linear transformations. As non-linear counterpart it is worth it to take note of Embedding methods such as ISOMAP [63], Locally Linear Embedding [64] and Laplacian Eigenmap [65]. Other methods are based on information-theoretic relations such as Mutual Information. Hence, the form or structure of the learned metric can be *linear*, *non-linear*, *local*. Linear metric learning paradigms are based on the learning of a metric in the form of a generalized Mahalanobis distance between data objects, i.e. $\mathcal{D}_{ij}^{\vec{W}} = \sqrt{(\vec{x}_i - \vec{x}_j)^T \vec{W}^T \vec{W} (\vec{x}_i - \vec{x}_j)} = \sqrt{(\vec{x}_i - \vec{x}_j)^T M (\vec{x}_i - \vec{x}_j)}$, where $M = \vec{W}^T \vec{W}$ is a matrix with suitable properties that needs to be learned. In other words, the learning algorithm learns a linear transformation $\vec{x} \rightarrow \vec{W}\vec{x}$ that better represents similarity in the target domain. Sometimes in available data there are some non-linear structures that linear algorithms are unable to capture. This limitation leads to a non-linear metric learning paradigm, that can be based on the *kernelization* of linear methods or direct non-linear mapping methods. The last cases lead, for the Euclidean distance, to a kernelized version combining the learned transformation $\phi(\vec{x}) : \mathbb{R}^m \rightarrow \mathbb{R}^m$ with a Euclidean distance function with the capability to capture highly non-linear similarity relations, that is $\mathcal{D}_{ij}^{\phi} = \|(\phi(\vec{x}_i) - \phi(\vec{x}_j))\|_2$ [66].

Local metric refers to a problem where multiple local metrics are learned and often relies on heterogeneous data objects. In the last setting, algorithms learn using only local pairwise constraints. According to the scheme depicted in Fig. 2.1 the *scalability* of the solution is a challenging task, considering the growing of the availability of data in the Big Data context. The scalability

could be important both considering the data set dimension n or the dimensionality of data m. Finally, the intrinsic optimization task underlying the metric learning paradigm makes important the optimality of the solution depending on the structure of the optimization scheme, that is, if the problem is convex or not. In fact for convex formulations it is guaranteed the attaining of a global maximum. On the contrary, for non-convex formulations, the solution may only be a local optimum.

As clustering algorithms are concerned, local metric learning paved the way towards the development of *subspace clustering* algorithms, whose capabilities of finding clusters in specific subspaces (namely, subspaces in which clusters themselves are better defined) have been widely affirmed in the literature, especially as Euclidean spaces are of interest [67, 68].

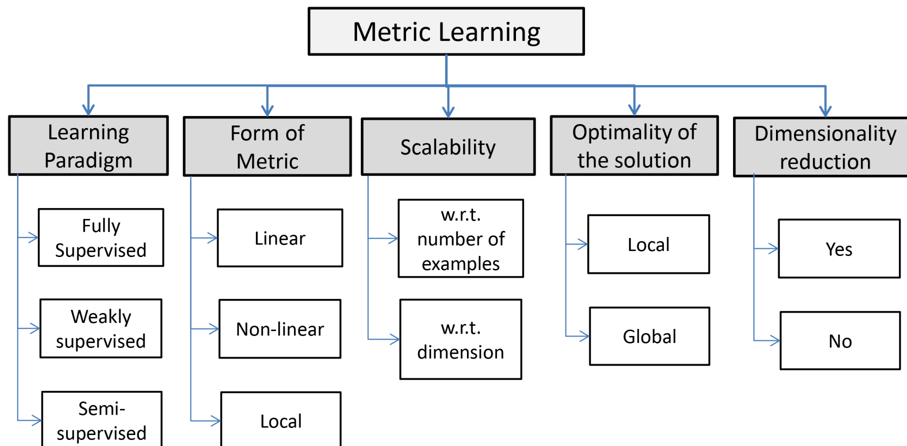

Figure 2.1: Five key properties of ML algorithms [69].

Chapter 3

Algorithms Design

In chapter 2 we described the principles at the base of this work inspired by some common properties observed in data sets. Here in the following we are going to analyze in details how these observations have been implemented in the algorithm design process. We will see how the problem has been split in two simpler sub-problems: one about how to discover clusters, possibly lying in different subspaces, in an efficient and scalable manner, the other on using these clusters to build a classification model.

3.1 Algorithm Properties

A real world data set may be very large. It may be so huge to make it unfeasible analyzing the entire data set in a centralized way. It can even be an infinite data stream generated during system at hand operation. Before to go on describing our solution to face this problem, we need some considerations about this kind of data sets and mostly their sub-sampling properties. Sup-

pose we have a data set as shown in figure 3.1[1] and suppose it is a data set with so many patterns that it is impossible to analyze it all at one time within a reasonable amount of time. For the sake of representation here we used a bi-dimensional Cartesian plot with patterns normalized in unitary hyper-cube, but as it is shown in the following it is not mandatory for our purpose. A necessary but realistic condition for the approach we are describing is that a given problem-related class is represented by one or more groups of homogeneous patterns (clusters): in light of this hypothesis, patterns that belong to the same class might not be necessarily similar one another. Now suppose

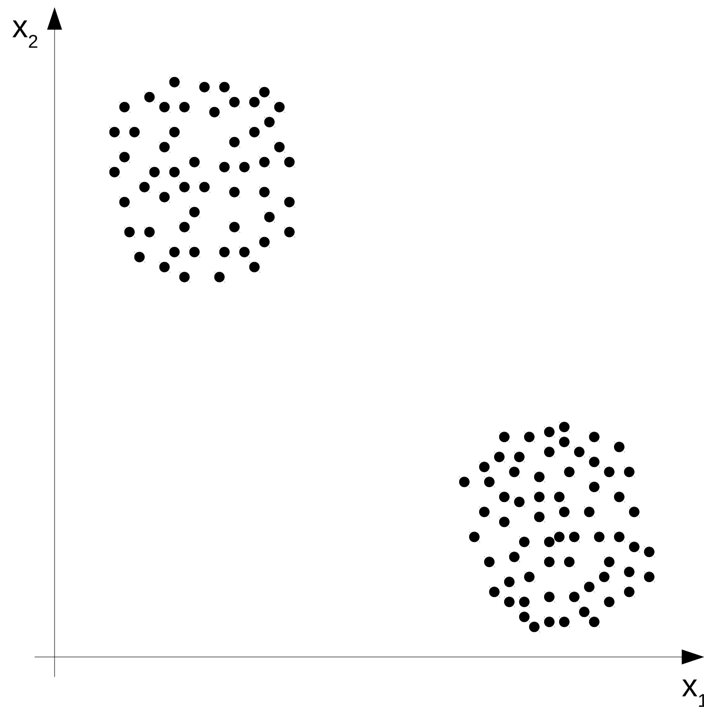

Figure 3.1: Two well-formed clusters in a bi-dimensional Euclidean space.

we do not consider the entire data set but only a small sub-sampling as shown

[1]Many images in this first part are not technical accurate pictures, they are intended for narrative purposes only.

in figure 3.2. Despite the lower data density (see Figure 3.1 vs. Figure 3.2), it is still possible to recognize portions of the input space where data is concentrated. This means clustering algorithms are still valid candidates when

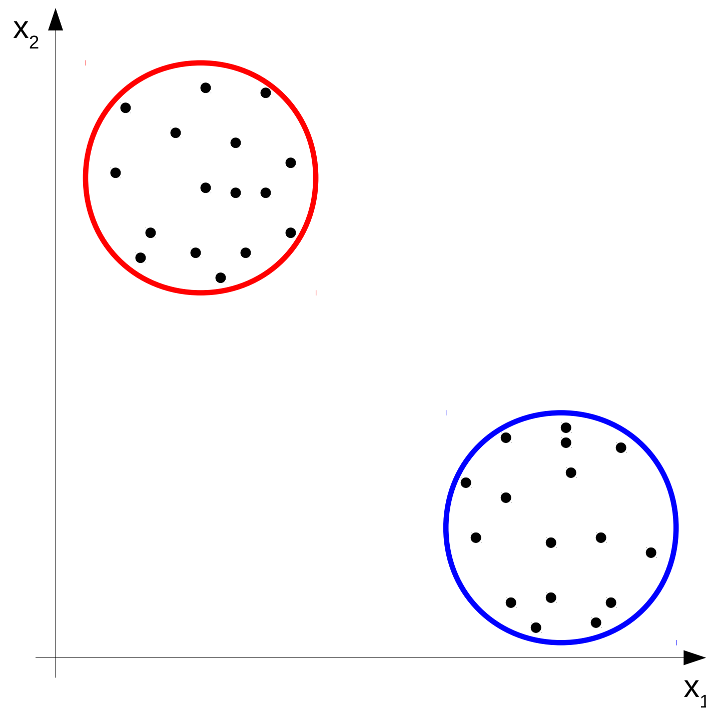

Figure 3.2: By performing a random sub-sampling, the two clusters from Figure 3.1 are still visible.

it comes to perform space segmentation, where each segment is associated with a class label. For this reason Decision Clusters have been chosen for our classification model: they suit for necessity of working with small data sub-sample and it allows to split the problem in two simpler tasks. The first task is to find groups of homogeneous patterns (clusters), the second is to use these clusters to build a classification model. Moreover Decision Clusters also satisfy the requirement of dealing with structured data or, more generally, non-metric data which can not be represented with a Cartesian plot.

In fact, several clustering algorithms such as the aforementioned k-medoids, DBSCAN and hierarchical clustering rely only on explicit pairwise dissimilarities and do not rely on any algebraic structures which turn out to be nonsensical as non-metric spaces are concerned [54, 70, 71, 72].
Under consideration of fast algorithm requirement, the choice for clustering algorithm at the basis of the proposed system fell on RL-BSAS [73], which adds some Reinforcement Learning-like behaviour to the standard BSAS algorithm [4]. RL-BSAS is not the strongest clustering algorithm in terms of performances because can only discover clusters with hyper-spherical shape, but it is capable of running in $O(n)$, where n is the number of patterns in the data set, because it needs to iterate over input data only once.
Feature selection may be a very challenging problem in high dimensionality spaces. In case of binary feature activation, where each feature can only be active or inactive (as it is the case in this work), it is a combinatorial problem whose complexity grows as $O(2^n)$ where n is the number of features in patterns domain. Our choice fell on entrusting this task to a metaheuristic optimization. The choice of a Genetic Algorithm [74] to face feature selection [75] is justified by the difficulty in describing the problem in closed form. In this sense our system acts as a black box: given a subspace, we are able to evaluate easily the quality of the results generated by its usage (fitness), but we can not define a procedure to modify it for certainly improve the performance. A Genetic Algorithm is instead capable of driving the optimization only relying on the fitness value associated to a given input variables configuration (i.e., the genetic code). Moreover, the intrinsic Genetic Algorithm population structure makes the design of a parallelizable multi-agent sys-

tem very appealing, due to the intrinsic agents' independence: indeed, each agent is in charge of investigating a different data sub-sample in a different subspace by means of a simple clustering algorithm with low resources requirements. The final result derives from inter-agent sharing and gathering together individual agents results obtained generation by generation. It is in a sense an emergent result generated by a plethora of elementary agents solving very simple tasks.

The development of this algorithm has been divided in two phases. In the first one we developed a system capable of performing clustering with simultaneous local metric learning, in the second we adapted it to create a classification model by collecting clusters, useful for this task, generation by generation. These two algorithms are described in details respectively in following §3.3 and §3.4. The former part has been investigated in [76] and [77], whereas the latter results have been published in [78].

3.2 Multi-Agent Framing

Referring to the taxonomy presented in [14] our algorithms can be classified as reward-based (specifically stochastic search) homogeneous team learning. These algorithms consist of multi-agent systems which globally aim in the first case to cluster the data set and, in the second one, to build a classification model. Despite each agent has its own parameter configuration and its own data set sub-sample to be analyzed, each one performs an instance of the same clustering algorithm. The most of inter-agent communication occurs by the Genetic Algorithm. On the basis of the achieved results each agent

is given a fitness value which results in its reward. The agent reward is in the form of a higher probability of sharing its configuration and propagating it to next generation. The higher is the agent fitness the the higher is the probability of propagating its genetic code to the next generation.

3.3 Clustering: E-ABC

As already mentioned above, this Evolutive Agent Based Clustering algorithm (E-ABC) consists of a population composed of simple agents which perform trivial tasks. Agents live in an environment promoting those which get better results and the population evolve generation by generation. During the evolution the result is built step by step to be returned at the end of the procedure. In the following we are going to walk through the entire structure with a bottom-up approach. We will start from the clustering algorithm performed by each agent to reach the evolutive over-structure going through the inter-agent interactions. The whole E-ABC structure is schematized in figure 3.3 which acts as an atlas of the algorithm.

3.3.1 Agent Behavior

Basic Sequential Algorithm Scheme

Basic Sequential Algorithm Scheme (BSAS) is a clustering algorithm which cluster the given data set with a single scan. As already discussed above, the choice of this algorithm is due to its simplicity and the consequent low system requirements for a single agent executing it. In its basic implementa-

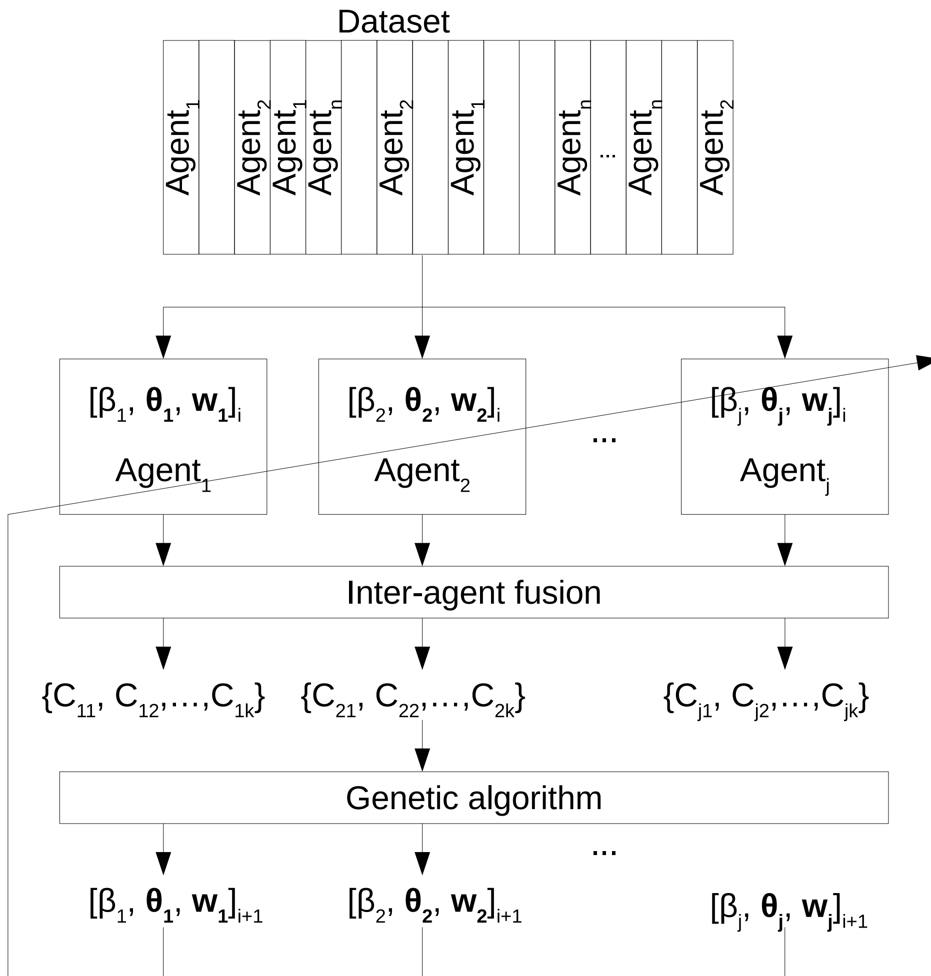

Figure 3.3: Block diagram of the E-ABC algorithm.

tion, BSAS only requires a single configuration parameter θ representing the maximum distance of a pattern from the centroid of the cluster it belongs to. BSAS scans all input data pattern by pattern. For each pattern p_x it computes the distance $D_k = d(p_x, C_k)$ from all of the clusters C_k already created and selects the one minimizing the distance D. If $D < \theta$, then p_x is added to the corresponding C_k cluster, otherwise a new cluster is created with p_x inside[2]. After each new pattern-to-cluster assignment, the centroid of the cluster is updated. Trivially, for very small θ values, BSAS is likely to return a huge number of clusters: in order to avoid this scenario, a new parameter K^* can be introduced and it indicates the maximum number of allowed clusters. If K^* is considered, new clusters are generated if and only if the number of clusters discovered so far is below K^*.

Let us suppose BSAS is analyzing the data set in figure 3.4 following the p_x index order. When BSAS analyzes the first pattern p_1 there is still no cluster so it just create C_1 and put p_1 inside. When p_2 is analyzed its distance from C_1 is computed, but, because $d(p_2, C_1) > \theta$, cluster C_2 with it is created. Distance of p_3 from C_1 and C_2 is computed. Because C_2 is the closets cluster and it is close enough, p_3 is added to it. And so on and so forth for the whole data set. Notice that each pattern resulting in an outlier with the given θ will remain alone or grouped with few other patterns in its cluster.

Pseudocode

[2]From here on each section describing an algorithm have an ending paragraph with its pseudocode adherent as much as possible to the implementation. Refer to it for details.

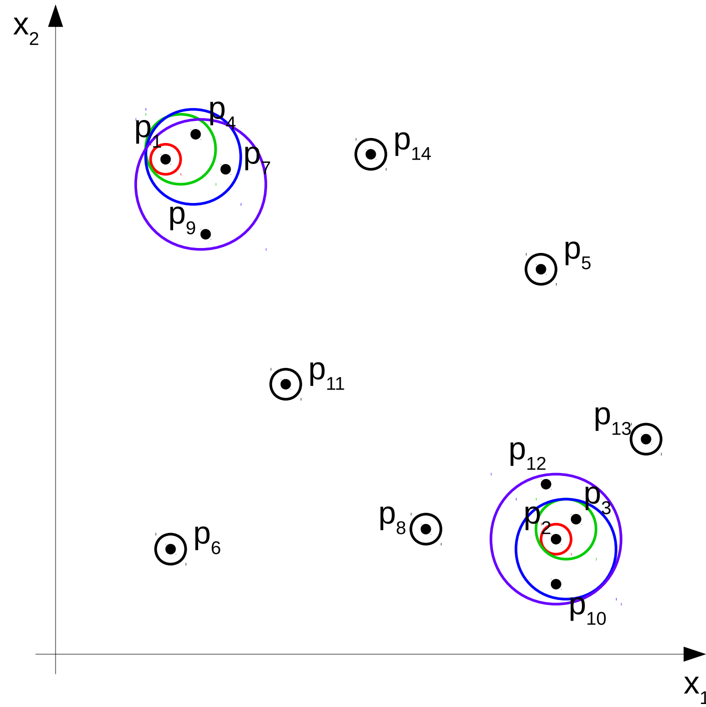

Figure 3.4: Schematic behaviour of the BSAS algorithm.

Symbols

C_k - k^{th} cluster

θ - cluster radius

D - distance between two entities (cluster - cluster, cluster - pattern, pattern - pattern...)

$\overline{D}$ - normalized D

p - single pattern in S_{tr}, S_v or S_{ts}

K^* - maximum number of clusters RL-BSAS is allowed to create

BSAS

Input: dataset S, θ, K^*

Output: C_k

1: Initialization: let the first pattern of the set S be the centroid of the first cluster C_k with θ radius
2: **for** each p in S **do**
3: **for** each C_k cluster **do**
4: compute pattern - cluster distance D (with respect to the metric configuration associated to C_k)
5: normalize D as $\overline{D} = \frac{D}{\theta}$
6: **end for**
7: select C_k for which $\overline{D}$ is minimized
8: **if** $\overline{D} < 1$ **then**
9: insert p in C_k
10: **else if** number of cluster $C_k \leq K^*$ **then**
11: create new C_k
12: insert p in C_k
13: **end if**
14: **end for**

Basic Sequential Algorithm Scheme with Reinforcement Learning

As we may observe in the above description, BSAS in its basic implementation is very sensitive to data ordering and, especially for low θ values, might as well create a huge number of clusters with very few patterns due to its partitional-like nature. This is the reason for which we adopted a modified version of it that at the same time reduce the number of returned clusters and only keeps the most relevant ones. It is named Reinforcement Learning BSAS (RL-BSAS): a Reinforcement Learning mechanism is used to keep track of the most relevant clusters which must be kept and those which instead deserve to be deleted. For this purpose each new cluster is associated with an initial energy value. When a new pattern p_x is associated to an existing cluster, the energy of this cluster is increased, whereas the energy

of all the others is reduced. If p_x can not be associated with any existing cluster, all their energies are decreased. If the energy of a cluster vanishes it is deleted. Moreover the maximum amount of returned clusters is bounded to a user defined number. When a pattern can not be associated with any existing cluster, a new one is created only is a free slot is available.

Pseudocode

Symbols

β - RL-BSAS penalty parameter

IE - initial RL-BSAS clusters energy

RL-BSAS

Input: dataset S, θ, K^*, β

Output: C_k

1: Initialization: let the first pattern of the set S be the centroid of the first cluster C_k with θ radius
2: **for** each p in S **do**
3: **for** each C_k cluster **do**
4: compute pattern - cluster distance D (with respect to the metric configuration associated to C_k)
5: normalize D as $\overline{D} = \frac{D}{\theta}$
6: **end for**
7: select C_k for which $\overline{D}$ is minimized
8: **if** $\overline{D} < 1$ **then**
9: insert p in C_k
10: increase C_k energy by 1
11: decrease all others C_k energy by β
12: **else**
13: decrease all C_k energy by β
14: **if** number of clusters $C_k < K^*$ **then**

15: create new C_k
16: initialize C_k energy to IE
17: insert p in C_k
18: **end if**
19: **end if**
20: **for** each C_k cluster **do**
21: **if** C_k energy ≤ 0 **then**
22: delete C_k
23: **end if**
24: **end for**
25: **end for**

Agent Characterization

What distinguish two agents is the set of parameters which condition their behaviour. Specifically these parameters are β, $\boldsymbol{\theta}$ and $\mathbf{w}$ highlighted in figure 3.5. β is the death rate of clusters in RL-BSAS reinforcement learning approach, $\boldsymbol{\theta}$ is a vector storing the actual radii of clusters discovered by the agent and initialized with θ and $\mathbf{w}$ describes the subspace where the agent is acting.

Intra-Agent Fusions

In E-ABC, agents have memory of discovered clusters as generations go by. In order to shrink the bucket of clusters belonging to a given agent, an intra-agent fusion procedure is triggered right after the RL-BSAS execution (see figure 3.5). The end-user defines a threshold parameter θ_{fus} and two clusters are merged if the distance between their respective centroids is below this threshold. After merging, the centroid of the resulting cluster is updated and

its strength is restored to the default value.

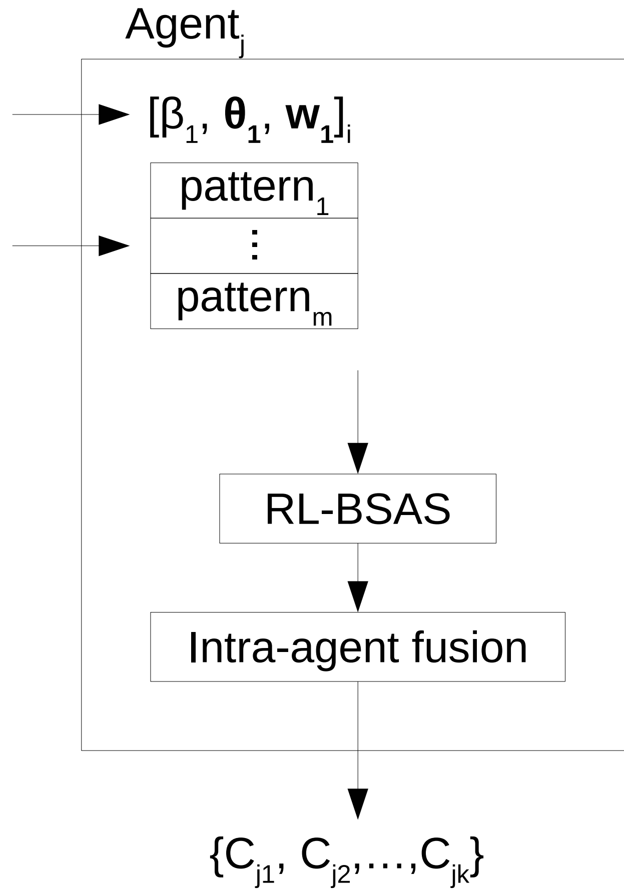

Figure 3.5: Block diagram of the agent behaviour in E-ABC.

Pseudocode

Symbols

P_i	-	population of ith generation
a_{ij}	-	j^{th} agent of P_i
C_{ijk}	-	k^{th} cluster discovered by a_{ij}
θ_{fus}	-	distance threshold for cluster fusion

E-ABC Agent

Input: a_{ij}, S^*_{tr}

Output: C_{ijk} (set in a_{ij})

1: CALL rlbsas with a_{ij}, S^*_{tr} RETURNING C_{ijk}
2: **repeat**
3: **for** each pair C_{ijk^1}, C_{ijk^2} in C_{ijk} **do**
4: compute D between C_{ijk^1} and C_{ijk^2}
5: **if** $D < \theta_{fus}$ **then**
6: merge C_{ijk^1} and C_{ijk^2}
7: **end if**
8: **end for**
9: **until** a merge occurs
10: set resulting C_{ijk} clusters in a_{ij}

3.3.2 Inter-Agent Fusions

When each agent of the current population has completed its task of running the clustering algorithm on the given data set sub-sample, an additional fusion step occurs. For various reasons (the same parents in the previous generation, evolutionary convergence or just accidentally) different agents may share the same search subspace. With this premise, different agents may discover similar clusters, namely clusters either very close to each other or with non-negligible overlap. In order to shrink the set of clusters discovered by the swarm during the evolution, agents are grouped by subspace and their clusters are merged together if they are closer then θ_{fus} parameter. After this step, genetic operators may take place. The inter-agent fusion procedure is quite similar to that above described for intra-agent fusion, except an additional check which is required to ensure clusters to be merged

were found in the same subspace.

Pseudocode

E-ABC Inter-Agent Fusion

Input: a_{ij}, S^*_{tr}

Output: C_{ijk} (set in a_{ij})

```
1: repeat
2:    for each pair C_{ij^1k^1}, C_{ij^2k^2} in C_{ijk} do
3:       if C_{ij^1k^1} subspace match C_{ij^2k^2} subspace then
4:          compute D between C_{ij^1k^1} and C_{ij^2k^2}
5:          if D < θ_{fus} then
6:             merge C_{ij^1k^1} and C_{ij^2k^2}
7:          end if
8:       end if
9:    end for
10: until a merge occurs
11: set resulting C_{ijk} clusters in a_{ij}
```

3.3.3 Evolutionary Procedure

Fitness Function

The core of any evolutionary optimization procedure is the evaluation of each individual with a given fitness function value. Once this have been done, they can be ordered before to be given to the genetic operators. In E-ABC we are just performing a clustering, so the only information about the performance of a single agent is contained in the quality of the clusters themselves. There are two main properties for a cluster to be desired: compactness and cardinality. Given a cluster C with centroid c containing a set of patterns p_x as

shown in figure 3.6, cardinality is the number of patterns contained in the

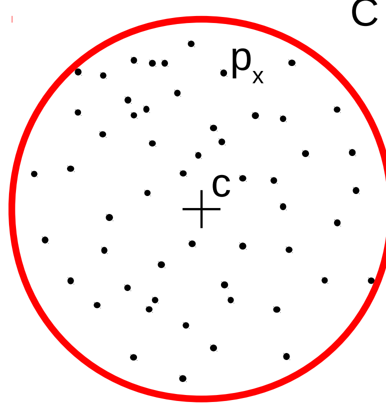

Figure 3.6: Given a cluster C centered in c, desired C properties are high cardinality (number of patterns p_x it contains) and high compactness (complement of average distance $p_x - c$).

cluster:

$$ca = |C| \tag{3.1}$$

and compactness co is defined as the complement of dispersion. Dispersion is in turn the average distance of p_x patterns from the centroid c of the cluster itself:

$$co = 1 - \frac{\sum_{p_x \in C} d(p_x, c)}{|C|} \tag{3.2}$$

to make them comparable, compactness and cardinality both have been normalized in $[0, 1]$ range by properly scaling according to the data set shard (the former) and the square root of the number of features (the latter). The normalization of the two terms allows a straightforward and unbiased combination: the final fitness (to be maximized) for a single cluster is then

$$F_{cc}(C) = \lambda \cdot f_{co}(C) + (1 - \lambda) \cdot f_{ca}(C) \tag{3.3}$$

where $\lambda \in [0, 1]$ is a user defined trade off parameter. So far we have defined how to compute the fitness value associated to a single cluster. The evolutionary procedure will not deal directly with clusters, but with the agents which output them. Each agent, by running RL-BSAS, may find multiple clusters and the fitness function associated to the agent is evaluated as the mean value among the fitness values of its clusters (see Eq. 3.3).

Genetic Operators

Once each individual fitness value has been evaluated, the current population can be sorted and the genetic operators may be applied to build the individuals for the next population. Parameters optimized by means of the genetic algorithm are: the penalty β for the RL-BSAS behavior, the clusters radii $\boldsymbol{\theta}$ and the metric representing the agent search space $\mathbf{w}$. The evolutionary procedure we use is a custom implementation and composition of well known genetic operators elitism, tournament selection, mutation and single-point crossover. One last note, it is worth to mention before the operators description, is that, where it is possible, inheritance does not only involve the agent configuration, but also the discovered clusters. In other words, if the configuration changes occurred in the evolutionary process do not invalidate the clusters discovered so far, the agent created for the next generation is not empty, instead it already contains the clusters found by the agents it is derived from.

Overall Structure This implementation of the genetic algorithm makes use of four standard genetic operators: elitism, selection, mutation and

crossover. The population of the next generation is composed of two blocks. The former is the result of the elitism and it is meant to preserve the best results obtained so far, while the latter, obtained through selection, mutation and crossover, is meant to explore the parameters space looking for better solutions. Specifically, the best individual so far is preserved by means of elitism and the end-user can specify the percentage of individuals obtained by means of crossover and mutation. If the percentage does not add up to the number of individuals, new individuals will be randomly spawned.

Elitism The elitism just copies the best agent (with respect to the given fitness values) from the current generation to the next one with all their configuration parameters, the discovered clusters and the set of patterns constituting the clusters themselves.

Selection The selection process chooses the individuals to be employed in the mutation and the crossover operators. 50% of the population is preliminarily selected through a uniform random extraction. Two lists of agents are then composed, one with the individuals to be modified by the mutation operator, and the other one with the individuals to be mated by the crossover operator. Each list is built by an additional random extraction step of the individuals from the subsampled population. The probability of an individual to be selected by the second extraction is proportional to its fitness value. Each agent can be selected only once for each list, but it may belong to both of them.

Mutation Depending on the fitness of the selected individual, the mutation may be "small" (local search) or "big" (exploration). It is small if the fitness value is high, big otherwise. This is because if the agent has gained a good fitness it is supposed to be near to a locally optimal solution in the search space, otherwise it is worth to try a much different parameters configuration. As mentioned above each agent configuration is composed of RL-BSAS penalty, clusters radii and the metric. Only one of these three components is mutated at a time. The first step is to extract a uniformly distributed value in $\{0, 1, 2\}$ to select which component to mutate. If 0 is extracted the RL-BSAS penalty value is mutated. The new value is extracted with uniform distribution. In case of small mutation the range to extract the new value is built as the old one $\pm 5\%$ of the admissible range size, if the mutation is big the extraction range is the old value $\pm 25\%$ of the admissible range size. When 1 is extracted by the block selection all the clusters radii of the agent are mutated by extracting the new value in the range centred on the old value $\pm 5\%$ (small mutation) or $\pm 25\%$ (big mutation) of the admissible range size. If the selected block is 2 (agent metric), in case of small mutation only one weight of the metric is flipped, otherwise a set of weights is randomly selected and flipped.

Crossover As described above, the section of the genetic code to apply the operator to is preliminarily selected by means of a random extraction. Because the list of radii has a link with the clusters each agent has found so far, it make no sense to exchange two cluster radii between two different agents. For this reason the crossover operator only applies to the RL-BSAS

penalty value and to the metric. If the former is extracted the value is exchanged between the two selected agents, otherwise one crossover point is extracted from the indices of the metric weights and they are exchanged from that index to the end of the metric weights list.

3.3.4 E-ABC Main Loop

Putting together the components described so far, the main E-ABC procedure structure may be summarized as it follows: one first population is created and randomly initialized. Each agent executes its clustering procedure for a different random data set sub-sample and its results are evaluated in terms of fitness value. At this stage the main loop of the evolutionary procedure is entered. Each loop execution corresponds to a different generation of the genetic algorithm. At each iteration the evolutionary procedure creates a new population, each agent performs the clustering of a new random data set sub-sample (possibly starting with the inherited clusters) and its fitness is evaluated. There are two possible stop criteria: the first one checks if the average population fitness has converged, the second one checks if the maximum number of iterations has been performed. The convergence test compares the current average fitness with the value of the previous generation. If the difference does not exceeds the given threshold a counter is incremented, otherwise it is set to 0. If the counter reaches the set threshold the main loop is interrupted. When one of the stop criteria occurs all the clusters discovered by all the agents in the last generation are collected all together and returned as the final result.

Pseudocode

Symbols

P_i	-	population of ith generation
$\lvert P_* \rvert$	-	population size
a_{ij}	-	j^{th} agent of P_i
i_{max}	-	max number of generations
S_{tr}	-	training set
S_v	-	validation set
S_{ts}	-	test set
M	-	classification model
Acc_{gl}	-	M accuracy over S_v or S_{ts}
θ	-	cluster radius
$\mathbf{w}$	-	binary vector for subspace search
C_{ijk}	-	k^{th} cluster discovered by a_{ij}
$\lvert C_{ijk} \rvert$	-	C_{ijk} normalized cardinality
$\langle C_{ijk} \rangle$	-	C_{ijk} normalized compactness
$Acc_{C_{ijk}}$	-	accuracy over S_v of the classification model only containing C_{ijk}
E	-	elite pool containing good agent configurations
$\delta_{F_{avg}}$	-	minimum F_{avg} increment for generation
$\sigma_{F_{avg}}$	-	$\delta_{F_{avg}}$ violations counter
$\overline{\sigma_{F_{avg}}}$	-	maximum $\delta_{F_{avg}}$ violations count
$Fcc_{C_{ijk}}$	-	C_{ijk} fitness based on $\lvert C_{ijk} \rvert$ and $\langle C_{ijk} \rangle$
$F_{C_{ijk}}$	-	C_{ijk} fitness based on $Fcc_{C_{ijk}}$, $Acc_{C_{ijk}}$ and Acc_{gl}
λ	-	Tradeoff parameter between $\lvert C_{ijk} \rvert$ and $\langle C_{ijk} \rangle$

E-ABC

Input: S_{tr}, S_v, S_{ts}

Output: M

1: $\sigma_{F_{avg}} = 0$
2: **for** $j = 1$ to $|P_*|$ **do**
3: set a_{0j} in P_0 to random configuration
4: **end for**
5: CALL evaluatePopulation with P_0
6: **for** $i = 1$ to i_{max} **do**
7: CALL evolvePopulation with P_{i-1} RETURNING P_i
8: CALL evaluatePopulation with P_i
9: compute F_{avg}
10: **if** F_{avg} increment $< \delta_{F_{avg}}$ **then**
11: INCREMENT $\sigma_{F_{avg}}$
12: **else**
13: $\sigma_{F_{avg}} = 0$
14: **if** $\sigma_{F_{avg}} > \overline{\sigma_{F_{avg}}}$ **then**
15: BREAK
16: **end if**
17: **end if**
18: **end for**
19: **return** C_{ijk} {for last i run}

Evaluate Population

Input: P_i

Output: $F_{C_{ijk}}$ (set in a_{ij})

1: **for** each a_{ij} in P_i **do**
2: set S^*_{tr} to S_{tr} random subsample
3: CALL rlbsas with a_{ij}, S^*_{tr}
4: **end for**
5: **for** each a_{ij} in P_i **do**
6: **for** each C_{ijk} in a_{ij} **do**

7: compute $Fcc_{C_{ijk}} = \lambda \cdot \langle C_{ijk} \rangle + (1 - \lambda) \cdot |C_{ijk}|$
8: **end for**
9: set C_{ij^*} as C_{ijk} with the highest $Fcc_{C_{ijk}}$
10: set a_{ij} fitness value to $Fcc_{C_{ij^*}}$
11: **end for**

Agents Evolution

Input: P_{i-1}

Output: P_i

1: copy GC with best fitness from P_{i-1} to P_i {elitism}
2: **for** 40% $|P_*|$ **do**
3: select GC_m with deterministic tournament over P_{i-1} {mutation}
4: mutate θ of GC_m with a scale factor
5: mutate $\mathbf{w}$ by flipping a random number of bit
6: add mutated GC_m to P_i
7: **end for**
8: **for** 40% $|P_*|$ **do**
9: select GC_{co1} with deterministic tournament over P_{i-1} {crossover}
10: select GC_{co2} with deterministic tournament over P_{i-1}
11: assign $\theta_{GC_{co1}}$ to GC_{co2} and $\theta_{GC_{co2}}$ to GC_{co1}
12: perform single-point crossover to $\mathbf{w}_{GC_{co1}}$ and $\mathbf{w}_{GC_{co2}}$
13: add crossovered GC_{co1}, GC_{co2} to P_i
14: **end for**
15: add random GC_r to P_i to fill population size {randomize}
16: **return** P_i

3.4 Classification: E-ABC^2

Evolutive Agent Based Clustering Classifier (E-ABC^2) is a classification algorithm which takes advantage of the approach proposed for E-ABC algorithm and uses it to extract meaningful clusters from the data set, which will later

be used for building the classification model.

3.4.1 Classification Model: Decision Clusters

For requirements described in chapter 2 and technical solutions chosen in §3.1, where motivations are described in details, the classification model used in E-ABC^2 is inspired by Decision Clusters model. Decision Cluster models are composed by a set of clusters induced over the training data. When an unknown pattern p_x is required to be classified, clusters containing the pattern itself are selected, and information about clusters composition in terms of their class partition is used to determine p_x membership (detailed classification procedure in §3.4.2).
Each cluster C included in model $M_C = \{C_1, C_2, ...\}$ is a set of patterns. Since BSAS has been considered as the core clustering algorithm in both E-ABC and E-ABC^2, clusters are well described by an hypersphere. Moreover, because of local metric learning assumption, each cluster may lie in a different subspace. To keep track of the subspace where C has been found, it is associated with a binary vector $\mathbf{w}$ storing which features have been taken into account to compute pattern-to-pattern dissimilarity. In summary: a classification model M_C is a set of clusters C; each cluster is a set of patterns p_x and is characterized by a subspace described by vector $\mathbf{w}$ where it lies, its centroid and its radius.

3.4.2 Pattern Classification Procedure

When a pattern p_x needs to be classified, all clusters C stored in model M_C are initially considered as prospective actors for classification. The pattern-to-cluster dissimilarity $D = d(p_x, c)$ for each C is computed by obviously considering the subspace $\mathbf{w}$ in which C lies. Only clusters for which D is smaller than C radius take part to the process. Selected clusters are used to determine the label to be associated with p_x by means of a majority vote mechanism. As highlighted in §3.4.3, some low quality clusters may be included in M_C, mostly in the early stage of the evolutionary procedure and for clusters close to the decision boundary. "Low quality" refers to clusters that, although they have a clear most voted class, contain a non-negligible percentage of patterns belonging to other classes, increasing the uncertainty for the classification proposed by the considered low quality cluster. To mitigate this problem we do not use crisp voting: to keep as much information as possible each cluster does not vote $+1$ for its most voted class. It instead express a vote in $[0, 1]$ range for each class, proportionally to its class composition. Votes of each cluster for all possible classes sum to 1. All contributions are then summed by class and p_x is associated with the most globally voted class. If p_x does not lie in any cluster from M_C, it is marked by the classifier as *unclassified*. As we will show in chapter 5, this, together with considering each unclassified pattern as a classification mistake, helps to avoid considering the classifier performance undeservedly high by chance.

Pseudocode

Symbols

L_p - class label assigned to p

V - class by class votes

E-ABC^2 Classification

Input: p

Output: L_p

1: initialize V to 0 vote for each class
2: **for** each C_k in M **do**
3: compute D between p and C_k
4: **if** $D < \theta$ **then**
5: sum C_k votes to V
6: **end if**
7: **end for**
8: **if** exists vote in $V \neq 0$ **then**
9: set L_p to most voted class in V
10: **return** L_p
11: **else**
12: **return** *unclassified*
13: **end if**

3.4.3 Model Synthesis

E-ABC adaptation for classification required modifications and additions to the procedure described in §3.3. We also tested a preliminary E-ABC^2 Python implementation whose results are in §4.2. For shorthand we do not report details about the initial Python implementation. The algorithm here described is already the version obtained taking into account what we observed in preliminary tests. Performance of the procedure described in these paragraphs is shown in detail in chapter 5.

With the same bottom-up order used in §3.3, in the following we expose E-ABC^2, mainly focusing to the differences with respect to E-ABC. Almost all modifications can be divided in three categories:

- parameters simplification, to improve the software usability by making it simpler to be configured
- introduction of structures and procedures required for classification task

E-ABC^2 Agent

The core clustering algorithm RL-BSAS used in E-ABC^2 agents is the same described above for E-ABC agents. At this stage we skip its description and we refer to 3.3.1 for details. The entire E-ABC^2 agent behavior is almost unchanged with respect to E-ABC agent. The only difference concerns intra-agent fusions dealt in 3.3.1. E-ABC version of this step required a user defined θ_{fus} parameter. In E-ABC^2 we decided to remove this parameter to simplify the setup and to substitute it with a procedure which makes more reliable this step. The decision if performing or not a fusion is not related to an a priori fixed threshold. It depends instead on clusters to be merged properties. If two clusters contain each other centroid or, in other words, if their distance is smaller then their minimum radius, they are merged together.
Agent characterization is slightly different. We observed how inheriting clusters together with the parents configuration from the previous generation worsen the classification model synthesis. It was imputed to a lower exploration capability: if all agents from the same parents start from the same

previous cluster, their configuration will never be used to investigate a different area and will never be able to discover other clusters visible in the same subspace. For this reason clusters inheritance has been removed and with that the agent is no more characterized by a vector of radii $\boldsymbol{\theta}$. Instead it is characterized by a single value θ which the agent uses for all clusters it creates.

Pseudocode

Symbols

θ_{ijk} - radius of *k-th* cluster found by agent a_{ij}

E-ABC^2 Agent

Input: a_{ij}, S^*_{tr}

Output: C_{ijk} (set in a_{ij})

1: CALL rlbsas with a_{ij}, S^*_{tr} RETURNING C_{ijk}
2: **repeat**
3: **for** each pair C_{ijk^1}, C_{ijk^2} in C_{ijk} **do**
4: compute D between C_{ijk^1} and C_{ijk^2}
5: **if** $D < \theta_{ijk^1}$ **and** $D < \theta_{ijk^2}$ **then**
6: merge C_{ijk^1} and C_{ijk^2}
7: **end if**
8: **end for**
9: **until** a merge occurs
10: set resulting C_{ijk} clusters in a_{ij}

Cluster Insertion in Model

When a new cluster C is chosen to be included in the model M_C, its most voted class is computed and a copy of C is added to the corresponding sub-

model. Details about how a cluster is selected to be a candidate and how is chosen to be included in M_C are discussed in 3.4.3.

Pseudocode

Symbols

$L_{C_{ij^*}}$ - label of most voted class for C_{ij^*} cluster

E-ABC^2 Cluster Insertion

Input: M, C_{ij^*}

Output: M update

1: compute $L_{C_{ij^*}}$ by counting patterns p in C_{ij^*} grouped by ground truth L_p

2: add C_{ij^*} to the M sub-model matching $L_{C_{ij^*}}$

Evolutionary Procedure

Based on preliminary tests highlighting difficulties on keeping high quality individuals with different genetic codes in the population, the whole evolutionary procedure has been refactored, except for fitness calculation that, despite modified to take into account classification performance in agent quality evaluation, include the previous fitness computation shown in §3.3.3. An elite pool structure, whose usage is also discussed in §3.4.3, has been introduced to store high quality individuals drawn back to the active population during the evolutionary procedure.

Fitness Function Equation 3.3, used in E-ABC for addressing the quality of the discovered clusters and for the evaluation of agents which have found

them, is still valid, despite its usage has been slightly modified and has been joint with information deriving from the classification capability. First, the normalization of compactness and cardinality have been performed in an affine fashion, by considering minimum and maximum bounds encountered so far in previous generations. Hence,

$$f_{co}(C) = \frac{\left(1 - \frac{\sum_{p_x \in C} d(p_x, c)}{|C|}\right) - co_{min}}{co_{max} - co_{min}} \tag{3.4}$$

and cardinality fitness component is

$$f_{ca}(C) = \frac{|C| - ca_{min}}{ca_{max} - ca_{min}} \tag{3.5}$$

Furthermore, each agent is characterized by the best value among its clusters rather than the average value. As regards the trade-off term λ, it is in fact another parameter that tests in §4.1 revealed could be removed. Specifically, figure 4.2 shows clustering performance being pretty insensitive to λ in its mid-range. For this reason and to reduce user defined parameters we fixed $\lambda = 0.5$, so as to result in F_{cc} being the arithmetic average of f_{co} and f_{ca}. Once a cluster C has been evaluated in terms of compactness and cardinality we need a method to estimate the utility of including C in M_C. For this purpose we build a temporary decision cluster model on the single cluster under consideration and we check for its classification capability in terms of accuracy Acc_C with respect to the validation set. Testing it on the whole validation set would make no sense because most elements would fall outside C and accuracy would be very low. So we have introduced an additional

step selecting only the validation set sub-sample falling inside C, and Acc_C is computed taking into account only this sub-sample. In this way Acc_C is computed only on validation set patterns for which C would express a vote if included in M_C as it is discussed in 3.4.2. Accuracy as quality measure works until dealing with balanced data sets. More robust quality parameter would be required for unbalanced data sets, whose study has been postponed to future development. The last step is composing F_{cc} and Acc_C in a single fitness value F for the cluster to be evaluated. We sifted different possibilities for this purpose. Static fitness components trade off would be the simplest approach but it would introduce another user defined parameter to be manually defined. Moreover we expect that at the early stage of this process the agents can only look for clusters with high F_{cc}. When population knows in which subspaces patterns are better clustered and it starts collecting clusters in M_C, it can focus on searching clusters useful for classification task. Static trade off does not satisfy the requirement of adapting to modifying target during the genetic algorithm execution. To aim this target we could modify the trade off as a function of the current generation index. It would allow to move from an objective to the other, but generation index does not actually give any information about the state of searching good subspaces, so we added an additional variable computed relying on M_C. It is Acc_{gl} and it is the accuracy obtained by the global classification model collected so far, computed over the entire validation set. In this way Acc_{gl} acts as a feedback. So at the beginning the process looks for well formed clusters. When good clusters start to be found and some of them are also useful in M_C, the evolutionary process modify its target step by step toward clusters useful for

classification. The complete formula for cluster fitness computing is

$$F(C) = Acc_{gl} \cdot Acc_C(C) + (1 - Acc_{gl}) \cdot F_{cc}(C) \quad (3.6)$$

Genetic Operators The main reason for which evolutionary procedure has been slightly modified is to satisfy the population diversification requirement whose necessity has been introduced at the beginning of §3.4.3. As in E-ABC, all genetic operators structure have been reorganized to reflect agents double nature containing in their genetic code information about the search subspace and about the clustering algorithm configuration. When a genetic operator is called for an agent, in turn it calls the same operator for the subspace and the clustering algorithm configuration. In this way, using some software constructs deepened in appendix A, the software is already prepared for other kinds of space representations and other clustering algorithms without the need of modifying it. In the following we discuss details of the current implementation dealing with subspaces represented by **w** binary vectors and using RL-BSAS as the core clustering algorithm.

Overall Structure The evolutionary procedure takes as input the current population sorted by fitness and generates the next one to be tested. The first 15% individuals of the new population are chosen by elitism. The following two bunches of 35% individuals each one are generated respectively by mutation and by crossover. All individuals to be involved in mutation and crossover are selected by means of a deterministic tournament. The

remaining 15% agents are randomly drawn.

Elitism As it is in E-ABC, individuals chosen for elitism are just copied as they are for the next generation as they are in the input population.

Selection The used selection algorithm is a standard deterministic tournament. Given the current population and the number of tournament competitors, they are randomly drawn. The best one in terms of fitness value is returned as the selected individual.

Mutation Agents mutation operator calls the same operator for $\mathbf{w}$ subspace and clustering algorithm configuration. In our implementation $\mathbf{w}$ mutation chooses one element with uniform probability distribution and flips it. For clustering configuration both increment and decrement cluster energy ratio β and θ radius values are drawn with Gaussian distribution whose mean value corresponds to the previous value and the standard deviation has been set as 0.05/3. New subspace and clustering configuration obtained by means of respective mutation operators are joined to build a new agent.

Crossover As in case of mutation, the corresponding operator is individually called for subspace and clustering configuration. Each operator takes two individuals from the previous generation and returns one pair of new genetic codes. Corresponding elements in both pairs are joined together to build a pair of new agents: first new subspace with first new clustering configuration and second new subspace with second new clustering configuration. Subspace crossover applies a single point crossover (it randomly draws a $\mathbf{w}$

position and swaps a subset of binary components), whereas clustering configuration crossover, having only two components in the genetic code, applies a uniform crossover to one of them (it randomly draws a number in $[0, 1]$ with uniform probability distribution and swaps the component if the number is lower than the crossover rate).

Randomization Also randomization calls the corresponding operators for subspace and clustering configuration. Each one returns a randomly generated genetic code for its part. These two genetic codes are joined together to build a new random agent. Subspace randomization draws each $\mathbf{w}$ binary component at random with uniform probability distribution. Clustering configuration randomization draws β in $[0.001, 0.999]$ range and θ in $[0.01, 0.50]$ range both with uniform probability distribution.

Pseudocode

Symbols

GC	-	a generic genetic code (may refer to agent genetic code, metric genetic code or clustering algorithm configuration genetic code)
MR	-	mutation rate
CR	-	crossover rate

Agents Evolution

Input: P_{i-1}

Output: P_i

1: copy 15% GCs from P_{i-1} to P_i {elitism}
2: **for** 35% $|P_*|$ **do**

3: select GC_m with deterministic tournament over P_{i-1} {tournament is 10% P_{i-1} based}
4: CALL mutateAgent with GC_m RETURNING GC_m^* {mutation}
5: add mutated GC_m to P_i
6: **end for**
7: **for** 35% $|P_*|$ **do**
8: select GC_{co1} with deterministic tournament over P_{i-1}
9: select GC_{co2} with deterministic tournament over P_{i-1}
10: CALL crossoverAgent with GC_{co1}, GC_{co2} RETURNING GC_{co1}^*, GC_{co2}^* {crossover}
11: add crossovered GC_{co1}, GC_{co2} to P_i
12: **end for**
13: **for** 15% $|P_*|$ **do**
14: CALL randomAgent RETURNING GC_r^*
15: add randomly drawn GC_r^* to P_i
16: **end for**
17: **return** P_i

Agent Mutation

Input: GC_m

Output: GC_m^*

1: CALL mutateMetric with GC_m metric RETURNING GC_m^* metric
2: CALL mutateClusteringConfig with GC_m clustering configuration RETURNING GC_m^* clustering configuration
3: create GC_m^* merging GC_m^* metric and clustering configuration
4: **return** GC_m^*

Agents Crossover

Input: GC_{co1}, GC_{co2}

Output: GC_{co1}^*, GC_{co2}^*

1: CALL crossoverMetric with GC_{co1}, GC_{co2} metrics RETURNING GC_{co1}^*, GC_{co2}^* metrics

2: CALL crossoverClusteringConfig with GC_{co1}, GC_{co2} clustering configurations RETURNING GC^*_{co1}, GC^*_{co2} clustering configurations
3: create GC^*_{co1} merging GC^*_{co1} metric and clustering configuration
4: create GC^*_{co2} merging GC^*_{co2} metric and clustering configuration
5: **return** GC^*_{co1}, GC^*_{co2}

Random Agent Draw

Input:

Output: GC^*_r

1: CALL randomMetric RETURNING GC^*_r metric
2: CALL randomClusteringConfig RETURNING GC^*_r clustering configuration
3: create GC^*_r merging GC^*_r metric and clustering configuration
4: **return** GC^*_r

Subspace Mutation

Input: GC_m

Output: GC^*_m

1: copy GC_m to GC^*_m
2: flip one random binary feature weight in GC^*_m
3: **return** GC^*_m

Subspace Crossover

Input: GC_{co1}, GC_{co2}

Output: GC^*_{co1}, GC^*_{co2}

1: copy GC_{co1}, GC_{co2} to GC^*_{co1}, GC^*_{co2}
2: draw a random index position idx in the binary feature weights array
3: **for** $i = 0$ to idx **do**
4: swap binary feature weights in GC^*_{co1}, GC^*_{co2} at position i
5: **end for**
6: **return** GC^*_{co1}, GC^*_{co2}

Random Subspace Draw

Input:

Output: GC_r^*

1: create GC_r^*
2: **for** each binary feature weight in GC_r^* **do**
3: draw a random binary 0/1 value
4: **end for**
5: **return** GC_r^*

Clustering Setup Mutation

Input: GC_m

Output: GC_m^*

1: draw a random value x with uniform distribution in $[0, 1]$
2: **if** $x < MR$ **then**
3: draw new random β value with Gaussian distribution centred in old value and standard deviation $\frac{0.05}{3}$
4: bound β in $[0.001, 0.999]$
5: set new β value in GC_m^*
6: **else**
7: set old β value in GC_m^*
8: **end if**
9: **for** each cluster radius in GC_m **do**
10: draw a random value x with uniform distribution in $[0, 1]$
11: **if** $x < MR$ **then**
12: draw new random θ value with Gaussian distribution centred in old value and standard deviation $\frac{0.05}{3}$
13: bound θ in $[0.01, 0.10]$
14: set new θ value in GC_m^*
15: **else**
16: set old θ value in GC_m^*
17: **end if**
18: **end for**

19: **return** GC_m^*

Clustering Setup Crossover

Input: GC_{co1}, GC_{co2}

Output: GC_{co1}^*, GC_{co2}^*

1: copy GC_{co1}, GC_{co2} to GC_{co1}^*, GC_{co2}^*
2: draw a random value x with uniform distribution in $[0, 1]$
3: **if** $x < CR$ **then**
4: swap θ values in GC_{co1}^*, GC_{co2}^*
5: **end if**
6: **return** GC_{co1}^*, GC_{co2}^*

Random Clustering Setup Draw

Input:

Output: GC_r^*

1: draw new random β value with uniform distribution in $[0.001, 0.999]$
2: draw one new random θ value with uniform distribution in $[0.01, 0.50]$
3: create GC_r^* with β and θ
4: **return** GC_r^*

E-ABC^2 Main Loop

The main loop starts with empty model, empty elite pool and a randomly generated population. The initial population is evaluated and the main loop is entered. We do not dwell on the main loop structure and stop criteria because they are almost the same already discussed for E-ABC algorithm. The only difference is here we observe the M_C accuracy trend instead of arithmetic average fitness trend to check if it is going to converge to a solution. When the main loop is over the synthesized model is returned. The main

loop consists in adding some individuals randomly drawn from the elite pool on top of the current population to ensure they will be part of the selection procedure, hence to ensure they will be used for creating a new population by means of the evolutionary procedure. Agents evaluation consists in turn in computing current model accuracy, running all agents clustering on data set sub-samples and computing their fitness value applying formula (3.6). During this process, if a cluster whose $Acc_C(C)$ value is at least 60% and $Acc_C(C) > Acc_{gl}$, the cluster itself is added to M_C and the agent who has found it is added to the elite pool.

Pseudocode

Symbols

δ_{Acc} - minimum Acc_{gl} increment for generation

σ_{Acc} - δ_{Acc} violations counter

$\overline{\sigma_{Acc}}$ - maximum δ_{Acc} violations count

E-ABC^2

Input: S_{tr}, S_v, S_{ts}

Output: M

1: $M = \emptyset$
2: $E = \emptyset$
3: $\sigma_{Acc} = 0$
4: **for** $j = 1$ to $|P_*|$ **do**
5: set a_{0j} in P_0 to random configuration
6: **end for**
7: CALL evaluatePopulation with P_0
8: **for** $i = 1$ to i_{max} **do**
9: add to P_{i-1} agents randomly drawn from E

10: CALL evolvePopulation with P_{i-1} RETURNING P_i
11: CALL evaluatePopulation with P_i
12: update Acc_{gl} with S_v
13: **if** Acc_{gl} increment $< \delta_{Acc}$ **then**
14: INCREMENT σ_{Acc}
15: **else**
16: $\sigma_{Acc} = 0$
17: **if** $\sigma_{Acc} > \overline{\sigma_{Acc}}$ **then**
18: BREAK
19: **end if**
20: **end if**
21: **end for**
22: update Acc_{gl} with S_{ts}
23: **return** M

Agents Fitness Evaluation

Input: P_i

Output: $F_{C_{ijk}}$ (set in a_{ij})

1: compute Acc_{gl}
2: **for** each a_{ij} in P_i **do**
3: set S^*_{tr} to S_{tr} random subsample
4: CALL rlbsas with a_{ij}, S^*_{tr}
5: **end for**
6: **for** each a_{ij} in P_i **do**
7: **for** each C_{ijk} in a_{ij} **do**
8: compute $Fcc_{C_{ijk}} = \frac{\langle C_{ijk} \rangle + |C_{ijk}|}{2}$
9: **end for**
10: set C_{ij^*} as C_{ijk} with the highest $Fcc_{C_{ijk}}$
11: compute $Acc_{C_{ij^*}}$ {computed over S_v subsample falling inside C_{ij^*}}
12: compute $F_{C_{ij^*}} = Acc_{gl} \cdot Acc_{C_{ij^*}} + (1 - Acc_{gl}) \cdot Fcc_{C_{ij^*}}$
13: set a_{ij} fitness value to $F_{C_{ij^*}}$
14: **if** $Acc_{C_{ij^*}} > 0.60$ **and** $Acc_{C_{ij^*}} > Acc_{gl}$ **then**
15: CALL addToModel with M, C_{ij^*}

16: add a_{ij} to E
17: **end if**
18: **end for**

3.4.4 Parallel Implementation

After satisfactory results have been obtained we optimized the algorithm by parallelizing time consuming loops whose steps are independent one each other [79]. All of the parallelizations have been implemented by means of the OpenMP API. OpenMP provides a thread pool architecture whose size (number of available threads) can be set at the application startup. The use of a thread pool reduces the multi-threading management overhead because it avoids to create and destroy a thread each time it is used. In brief, each agent independently receives a random data set chunk and outputs a set of clusters. These tasks can be easily parallelized with a naive parfor-like loop. The master task spawns as many tasks as there are agents and forwards each one, with the agent genetic code and the data set chunk, to an available thread in thread pool: threads perform in parallel agents' search for clusters. When a thread completes its task receives a new agent to perform its cluster search, until the task queue is over. The control now returns to the master thread that updates the minimum and maximum bounds for compactness and cardinality ($co_{\min}$, $co_{\max}$, $ca_{\min}$ and $ca_{\max}$). Another parallel for loop evaluates the clusters returned by each agent (compactness and cardinality – see Eqs. (3.4), (3.5), (3.3)), elects the best cluster and evaluates its classification capabilities Acc_C over the validation set. Finally it computes the fitness value to associate with each agent as in equation (3.6). The analysis

of the validation set is also included in a parfor-like loop so that the classification of each pattern in validation set is performed in parallel. However, since the validation set might be huge as well (and analyzing it all for each agent can be a serious bottleneck) it undergoes a subsampling (like the training set). The same subsample is used to compute Acc_{gl} and Acc_C for each agent. In this way each agent is evaluated over the same classification task. At each genetic algorithm generation a new subsample is randomly drawn with a stratified sampling to keep classes proportions and to avoid loosing validation set information.

Chapter 4

Preliminary Results

This chapter is organized in two sections: the first one describes setup, tests and results obtained with E-ABC, and the second one has the same layout but is focused on testing E-ABC^2. Both of them are preliminary tests gained with a Python version of the software. They where planned to quickly realize a first implementation of the algorithms and check what it was necessary to modify before to write the C++ version.

4.1 E-ABC

We propose a set of experiments in order to check its sensitivity to the user defined parameters. To highlight the metric learning ability we designed a data set with multiple clusters grouped by different subspaces.

Most of the parameters (β, θ range, θ_{fus}) can be deduced from the data set properties and what the user wishes to discover (minimum cluster size, maximum cluster extension, ...). Our choice of getting rid of λ in E-ABC^2

Table 4.1: Schematic representation of the clusters' structures

C_1	G_1	G_2	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$
C_2	G_3	G_4	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$
C_3	$\sim$	$\sim$	G_5	G_6	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$
C_4	$\sim$	$\sim$	G_7	G_8	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$
C_5	$\sim$	$\sim$	$\sim$	$\sim$	G_9	G_{10}	G_{11}	$\sim$	$\sim$	$\sim$

stems from the following tests in E-ABC. These tests are not supposed to be extensive stress tests investigating in details the algorithm capabilities, they are instead preliminary tests we used to drive the development in direction of E-ABC^2 implementation as already mentioned in 3.4.3. The outcome of these tests has been published in [77].

4.1.1 Data Set Description

Each test is run on the same data set. The data set contains 5 clusters, each of which is composed by 1000 patterns. Each cluster is described by 10 features, but only a few of them are useful to identify it. In particular two clusters belong to S_{12} (figure 4.1), two others to S_{34} and the last one to S_{567}, where S_X is the projection on the X subspace, and X is a subset of the features indexes. The outline of the dataset is summarized in table 4.1 where C_i is the ith cluster, G_j is a column vector of 1000 samples drawn from a Gaussian distribution with standard deviation $\sigma = 0.02$. Mean values differ for each cluster C_i and they are summarized in table 4.2. The $\sim$ symbol stands for 1000 samples drawn from a uniform distribution, so the corresponding feature does not give any useful information to characterize the cluster at hand. The last three columns do not contain any useful information, but only aim to increase the metric learning task complexity.

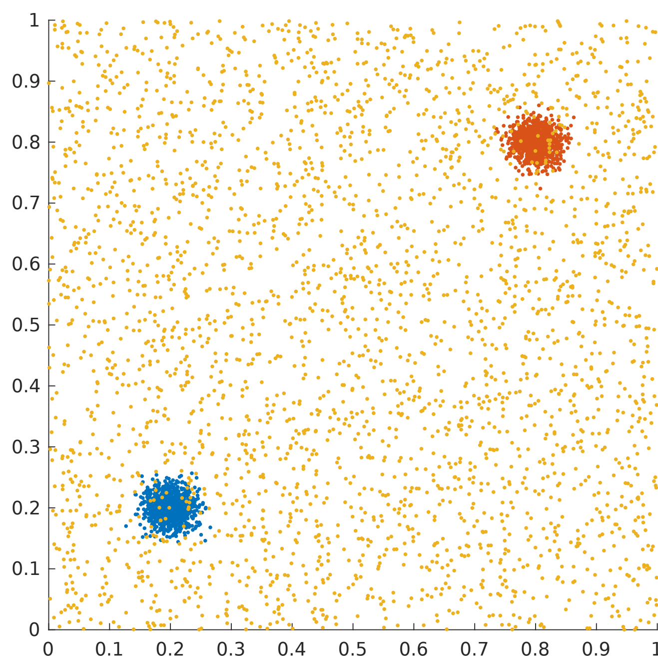

Figure 4.1: The plot shows the first two components of the entire data set, where patterns of C_1 and C_2 are plotted in blue and red, respectively, while yellow points are due to noise from clusters C_3, C_4 and C_5. We want to stress that only two groups of 1000 patterns each are concentrated in two different space regions, whereas the remaining 3000 patterns are uniformly distributed over the whole bidimensional subspace.

Table 4.2: Clusters' centroids

cluster	centroid
C_1	$(0.2; 0.2)$
C_2	$(0.8; 0.8)$
C_3	$(0.8; 0.2)$
C_4	$(0.2; 0.8)$
C_5	$(0.7; 0.7; 0.7)$

4.1.2 Evaluation Metrics

In order to address E-ABC ability in discovering clusters and, more importantly, in estimating their parameters, we propose three evaluation measures:

- Estimation Quality (EQ): whose task is to judge radius and centroid estimation accuracy as

$$EQ = 1 - [(1-\epsilon)\cdot\delta_{\mathbf{c}} + \epsilon\cdot\delta_{\theta}] \tag{4.1}$$

  where $\delta_{\mathbf{c}}$ is the Euclidean distance between true and estimated centroids, δ_{θ} is the absolute difference between true and estimated radii and $\epsilon \in [0;1]$ is a trade-off parameter (in our tests $\epsilon = 0.5$). Obviously $EQ \in [0;1]$ and as $EQ \to 1$, the more accurate the results.

- Identified Patterns (IP): which basically counts how many patterns within the estimated cluster are indeed part of the ground-truth cluster. Such value (N_C) is then normalized by the size of the cluster itself ($|C|$), in order to have $IP \in [0;1]$ and as $IP \to 1$, the more accurate the results:

$$IP = \frac{N_C}{|C|} \tag{4.2}$$

- Cluster Filling (CF): It is the number N_C normalized with respect to the cardinality of the corresponding ground-truth cluster (C_{GT}):

$$CF = \frac{N_C}{|C_{GT}|} \tag{4.3}$$

  This measure is meant to check which percentage of C_{GT} patterns has

been correctly discovered. As $CF \to 1$, the more accurate the results.

Each cluster returned by the algorithm is taken into account only if its metric matches with one of those in the ground-truth. Since multiple ground-truth clusters share the same metric, the one for which the estimated cluster has the highest EQ is selected, and IP and CF are only computed in relation to it.

4.1.3 Tests Results

To test the algorithm sensitivity to λ we have run it for λ assuming different values spanning from 0.1 to 1.0 with increment step equal to 0.1. The higher is λ, the higher is the compactness relevance in (3.3) with respect to the cardinality. Because it makes no sense to look for a cluster with high cardinality without considering its compactness, we decided to exclude the case of $\lambda = 0.0$. For the remaining parameters we have set $N_A = 100$, $|R|_{min} = 35$, $N_G^{STOP} = 10$, $M = 3$, $\theta \in [0.01; 0.14]$, $\beta \in [0.1; 0.9]$, $\theta_{fus} = 0.015$.

In figure 4.2 we show the average performances over the 5 clusters in terms of EQ, IP and CF. Due to the probabilistic nature of the algorithm the evaluation metrics are in turn averaged over multiple runs. The test stopped when the mean EQ value got steady (it does not change significantly for 10 consecutive iterations). In table 4.3 instead we present the details of a single run chosen from $\lambda = 0.5$ for which the algorithm has good average performances.

As λ changes, the EQ is pretty constant. This means that the position and the radius of the ground-truth clusters have been estimated with a small

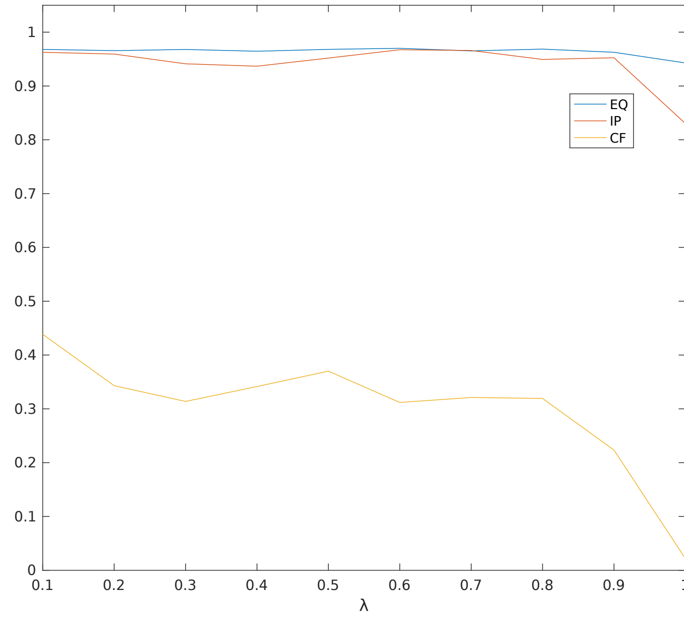

Figure 4.2: EQ, IP and CF average plots for different λ values

Table 4.3: Comparison between ground-truth (C_{xGT}) and estimated (C_{xEst}) clusters properties

cluster	centroid	radius
C_{1GT}	$(0.200; 0.201)$	0.040
C_{2GT}	$(0.800; 0.800)$	0.038
C_{3GT}	$(0.800; 0.200)$	0.037
C_{4GT}	$(0.200; 0.800)$	0.043
C_{5GT}	$(0.699; 0.699; 0.699)$	0.024
C_{1Est}	$(0.201; 0.203)$	0.025
C_{2Est}	$(0.801; 0.801)$	0.028
C_{3Est}	$(0.800; 0.199)$	0.019
C_{4Est}	$(0.199; 0.802)$	0.021
C_{5Est}	$(0.699; 0.699; 0.699)$	0.023

displacement for each considered weighting between compactness and cardinality. For high values of λ ($\lambda \in [0.8; 1.0]$) we have a quick deterioration of the performances in terms of both IP and CF. For low values of λ ($\lambda \in [0.1; 0.4]$) instead the IP decreases and oscillates, whereas the CF increases. This is imputable to the creation of huge clusters (small values of λ promote clusters with high cardinality, by almost ignoring their compactness) which may contain many patterns that do not indeed belong to the corresponding ground-truth cluster. Both of the extremes show undesirable behaviours. The results are indeed stably good for $\lambda \in [0.4; 0.7]$. It is worth to finally note that EQ and IP are very high in absolute terms: the former is never below 0.95 and the latter is below 0.90 only when the clusters are searched by taking into account just their compactness. The only drawback is the lack of clusters filling with respect to the ground-truth, despite it is imputable to the nature of the algorithm which tries to discover the clusters without exploring the entire dataset for the sake of time consumption.

4.2 E-ABC^2

As we said in above §4.1, tests shown here should not be taken as definitive results of final E-ABC^2 implementation. They are rather experiments we performed and published in [78] to check if the architecture could work and which modifications it required to improve performance.

To estimate the algorithm performances we performed two different sets of tests. In the former group we ran the algorithm on variations of the benchmark data set Iris with gradually increasing complexity, in the latter we

tested the performances on a real data set containing faults data from the Acea power grid in Rome.

4.2.1 Evaluation Metrics

Due to the balanced distribution of the patterns in the classes of the data set we used in our tests, the *accuracy* would be sufficient to validate the classifier performances. Anyway, to give a more complete picture of the results, we also report *recall*, *precision* and F_1-*score*. All of these measurements have been computed with respect to the test set S_{ts}. Let cm_{ij} be the number of misclassifications in assigning the jth class label to a pattern associated with the ith ground-true label. The accuracy Acc, the recall Rec_i, the precision $Prec_i$, the specificity $Spec_i$ and the F_1-score F_{1i}, related to the ith class as the positive one, are respectively defined as:

$$Acc = \frac{\sum_{i=1}^{L} cm_{ii}}{|S_{ts}|} \tag{4.4}$$

$$Rec_i = \frac{cm_{ii}}{\sum_{j=1}^{L} cm_{ij}} \tag{4.5}$$

$$Prec_i = \frac{cm_{ii}}{\sum_{j=1}^{L} cm_{ji}} \tag{4.6}$$

$$Spec_i = \frac{\sum_{j=1,j\neq i}^{L} cm_{jj}}{\sum_{j=1,j\neq i}^{L} cm_{jj} + \sum_{j=1,j\neq i}^{L} cm_{ji}} \tag{4.7}$$

$$F_{1i} = 2 \cdot \frac{Rec_i \cdot Prec_i}{Rec_i + Prec_i} \tag{4.8}$$

4.2.2 Test On Iris Data Set

Data Set Description

As above mentioned, the first data set used in our test is based on the well-known Iris data set. It has been slightly manipulated to be adapted to the algorithm properties and to highlight its abilities to work on suitable random samples drawn from the data set. The original Iris data set is composed of 3 classes each one containing 50 patterns and, consequently, we artificially increased the data set cardinality up to 3000 patterns, 1000 for each class:

1. the entire data set has preliminarily been normalized in $[0, 1]$ feature by feature

2. for each original pattern p_{xo} 19 new patterns are created. The ith component p^i_{xn} of each new pattern p_{xn} is computed as

$$p^i_{xn} = p^i_{xo} + d \tag{4.9}$$

   where $d \in [-0.05, 0.05]$ is a displacement randomly extracted with uniform distribution

3. the obtained patterns are normalized again to ensure they lie in the unitary hypercube.

The initial normalization is needed to avoid a data set deformation introduced by patterns extraction. The resulting T_0 data set is used to generate 10 additional data sets $T_1, ..., T_{10}$, where $1, ..., 10$ is the number of additional features appended to each pattern. The role of the additional features is to

increase the complexity of the feature extraction task. They are randomly drawn with a uniform distribution in $[0; 1]$ so that the additional features do not hold any informative content and they just act as noisy components for the model synthesis procedure. Each data set is split, by means of a stratified sampling, in training set S_{tr}, validation set S_v and test set S_{ts}. S_{tr} is used by the algorithm to look for the clusters, S_v is to check the ability of the discovered clusters to act as a classifier and S_{ts} is used to check the performances of the final classifier when the model synthesis is over.

Tests Results

For each data set T_i the algorithm has been executed on 10 different splits $\{S_{tr}|S_v|S_{ts}\}$. The results presented in the following are computed as averages of the performances gained in the 10 different runs. Figures 4.3, 4.4 and

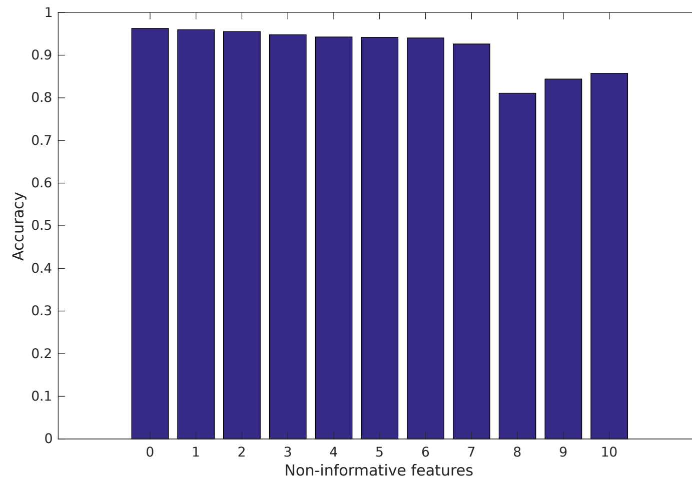

Figure 4.3: The accuracy obtained on the test set by the final synthesized models for Iris data set.

4.5 respectively show the accuracy and the class by class F_1-score obtained

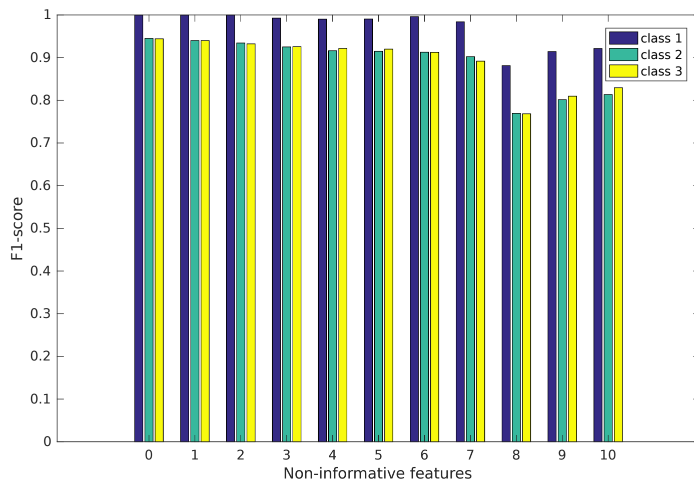

Figure 4.4: The class by class F_1-score obtained on the test set by the final synthesized models for Iris data set.

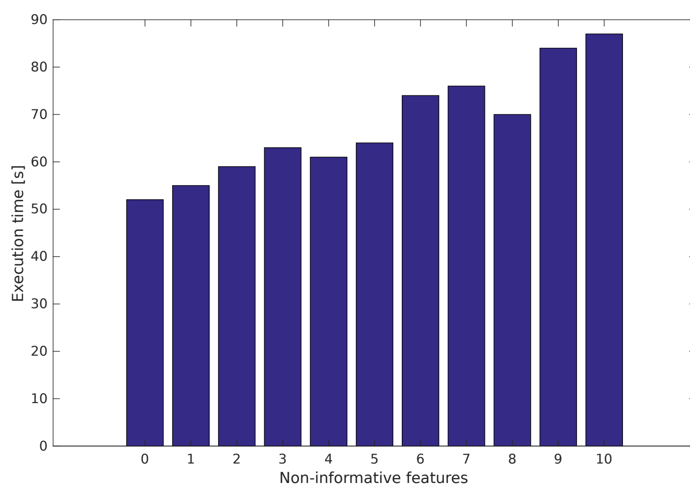

Figure 4.5: Execution time: the mean time required by the algorithm to converge to a model with stable performances (the lower is the better) for Iris data set. Tests have been carried out on a notebook equipped with Intel Core i7-840QM and 8GB RAM, running Ubuntu 16.04 LTS with Python 2.7.

Table 4.4: Detailed results for Iris data sets. Performances are measured on test sets.

	Recall			Precision			F_1-score		
	L_1	L_2	L_3	L_1	L_2	L_3	L_1	L_2	L_3
T_0	1.00	0.95	0.94	1.00	0.94	0.95	1.00	0.95	0.94
T_1	1.00	0.94	0.94	1.00	0.94	0.94	1.00	0.94	0.94
T_2	1.00	0.94	0.92	1.00	0.93	0.94	1.00	0.93	0.93
T_3	1.00	0.93	0.91	0.99	0.92	0.94	0.99	0.93	0.93
T_4	1.00	0.91	0.92	0.98	0.92	0.93	0.99	0.92	0.92
T_5	1.00	0.91	0.91	0.98	0.92	0.93	0.99	0.91	0.92
T_6	1.00	0.92	0.90	0.99	0.91	0.92	1.00	0.91	0.91
T_7	1.00	0.92	0.86	0.97	0.89	0.93	0.98	0.90	0.89
T_8	0.93	0.76	0.75	0.84	0.79	0.80	0.88	0.77	0.77
T_9	0.93	0.79	0.81	0.90	0.82	0.82	0.91	0.80	0.81
T_{10}	0.93	0.81	0.83	0.91	0.83	0.84	0.92	0.81	0.83

with respect to S_{ts}, and the time required by the algorithm for the model synthesis. The abscissa for all of these charts reports the number of non-informative features added to the Iris data set. Figures 4.3 and 4.4 show in terms of both accuracy and F_1-score a slightly performance degradation starting from 7 non-informative features. With 8 to 10 non-informative features the performances drop down, whereas the accuracy and the average F_1-score over the three classes never goes below 80%. In table 4.4 we report the detailed results in terms of class by class recall, precision and F_1-score for each test performed with the Iris data sets.

4.2.3 Test On Acea Data Set

Data Set Description

In this work, we study a real-world data set of smart grid functional states collected by Acea personnel, containing information about faults in the power

grid of Rome (Italy). Data have been gathered by smart sensors spread within the power network and stored by various Information Systems. We refer to a Localized Fault (LF) as the failure of the electrical insulation (e.g., cables insulation) that compromises the correct functioning of the grid. Therefore, a LF is actually a fault in which a physical element of the grid is permanently damaged causing long outages. The considered real-world data set consists in 2080 patterns describing the power grid states that are split into standard functioning states (SFSs) and LFs, that is, each pattern x is associated with a label $L(x) : L \in \{\texttt{LF}, \texttt{SFS}\}$. The pattern distribution between the two classes is quite balanced, so the rationale behind the choice of accuracy as performance metric still holds. The specific pattern structure has been designed together with Acea personnel and consists in 17 features. The features pertaining the Acea data set are heavily structured and they belong to different data types: categorical (nominal), quantitative (i.e., data belonging to a normed space) and times series (TSs). The "material' constituting the electric equipment (Cu or Al for cables) and its "location' (aerial or underground for cables) are categorical variables. The main real-valued features are i) the "PS-LF distance" that represents the distance between the Primary Station and the location of the LF that gives, in turn, a rough estimate of the load at that location, ii) the number of Secondary Station ("#SS") between the fault location and the Primary Station, iii) the minimum, maximum and variation of environmental temperature, the average millimeters of rain in the 2 hours time interval preceding the fault, iv) a feature estimating the increasing/decreasing change of the Backbone Electric Current ("BEC") extracted from the electric current measured on the faulty MV feeder. An-

other important measure defined as an integer feature is the Current Out of Bound ("Current OFB"), defined as the maximum operating current of the backbone that is less than or equal to 60% of the threshold "out of bounds", typically established at 90% of capacity. Other features are related to temporal information, as the number of days from the beginning of the year and the number of minutes from the beginning of the day related to when the LF fault occurred, while other concern spatial information, as the geographic coordinates of the fault. Unevenly spaced TSs describe the sequence of short outages that are automatically captured by the protection systems (Petersen alarm system) as soon as they occur. Specifically, we have three type of micro-interruptions: "Int Breaker" related to the intervention of the primary breaker, "Saving Intervention" related to the decisive interventions of Petersen's coil which have prevented the LF and, finally, "Petersen Alarms" that concerns alarms detected by the device called Petersen's coil due to loss of electrical insulation on the power line. The last three events series has been found correlated with a LF. The data set was validated by cleaning it from human errors and by completing in an appropriate way missing data, as explained in [80, 81], where it can be found even a detailed description of the features and the related data set construction. The complexity of the Acea data set measured by a suitable methodology is provided in [82].

The Custom-Based Dissimilarity Measure

In Pattern Recognition, objects pertaining many real-world problems may have a complex representation. This representation is based on a set of measurements arranged as a list (collection) of heterogeneous data types, i.e.

a structured pattern consisting of features having different semantic and, thus, syntactic nature. In other words, instead of dealing with a vectorial feature space endowed with standard algebraic structures such as the Euclidean distance, a non-metric starting space can be still treated defining a custom-based dissimilarity measure that generalizes the Minkowsky family's distances. This class of dissimilarities can be constructed component-wise by defining a suitable dissimilarity measure depending on the nature of the considered feature within the feature set describing the structured object at hand. It is the case of the Acea data set in which the starting feature space is structured by definition.
Therefore, given two structured patterns x, y representing two different power grid states, the dissimilarity measure is in general computed as

$$d(x_1, x_2; \vec{W}) = \sqrt{(x_1 \ominus x_2)^T \vec{W} (x_1 \ominus x_2)}, \tag{4.10}$$

where T denotes the transpose operator, the weight vector $\vec{w} = diag(\vec{W}) \in \{0, 1\}$, in this work, is binary, hence his functionality in E-ABC^2 algorithm consists in selecting relevant features, carrying out a (local) feature selection procedure, while the $\ominus$ operator represents a generic dissimilarity measure defined depending on the nature of the feature at hand. Specifically, the proposed weighted custom dissimilarity measure adopts the following sub-dissimilarities for each feature type. For quantitative data it is necessary to take care of the semantic difference between integer values describing temporal information and real-valued data related to other information like the physical power grid status or weather conditions. For temporal information

(integers) the circular difference is used, while real-valued data, correctly normalized, can be treated with the standard arithmetic difference. Categorical data in our SGS data set are of nominal type, thus they do not have an intrinsic metric structure and the well-known simple matching dissimilarity is adopted. The dissimilarity measure among the unevenly spaced TSs (sequences) data is performed by means of the Dynamic Time Warping (DTW) algorithm.

Tests Results

In the following section we present the main results obtained over the data set Acea evaluating the capability of E-ABC of recognizing fault states from standard functioning states. Ten runs of the algorithm have been performed and results are collected in Table 4.5. The model is learned on the Training Set S_{tr}, validated on the Validation Set S_v and, finally, tested on the Test Set S_{ts}. From Table 4.5 it is clear that E-ABC is able to learn a suitable classification model, recognizing LFs from standard functioning states. Looking at the average F_1-score and the Accuracy, classification performances are satisfying, although some runs show a lower recall.

The max and average values of the fitness as a function of the generation number are depicted in Figure 4.6 for the run T_7. For the average fitness, despite some oscillations, it realizes its process of slow growth. In Figure 4.7 the trend of the Accuracy as a function of the generation number is reported for the same run. After a first stage, following a behavior similar to that of the fitness, it rises slowly, falling down to several local optima and then resuming the trend reaching the final accuracy value. In addition to

Table 4.5: Detailed results for Acea data set. Performances are measured on test sets.

	Recall	Precision	Accuracy	Specificity	F_1-score
T_1	0.79	0.991	0.906	0.995	0.879
T_2	0.707	0.907	0.842	0.945	0.795
T_3	0.81	1.00	0.918	1.00	0.895
T_4	0.745	1.00	0.89	1.00	0.854
T_5	0.962	1.00	0.984	1.00	0.981
T_6	0.79	1.00	0.909	1.00	0.882
T_7	0.893	0.996	0.952	0.997	0.942
T_8	0.855	0.996	0.936	0.997	0.92
T_9	0.945	1.00	0.97	0.99	0.965
T_{10}	0.945	0.81	0.976	1.00	0.972
Mean	0.844	0.988	0.928	0.992	0.909
Variance	0.008	0.001	0.002	0.00	0.003

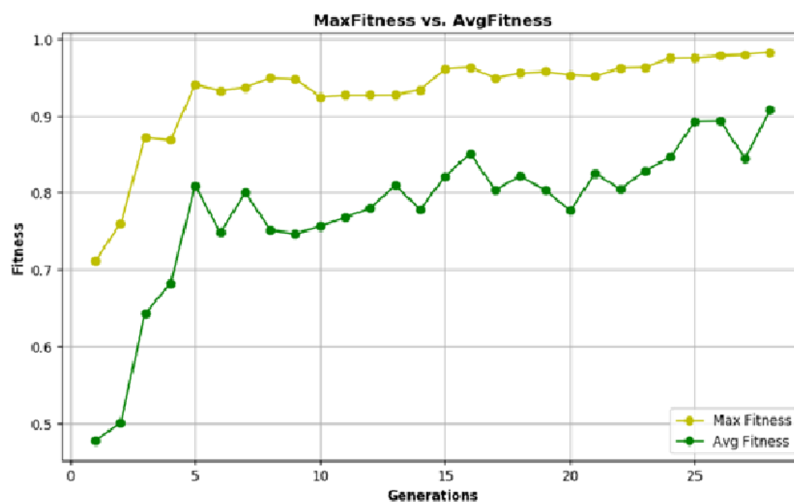

Figure 4.6: Maximum and avarage fitness related to run T_7.

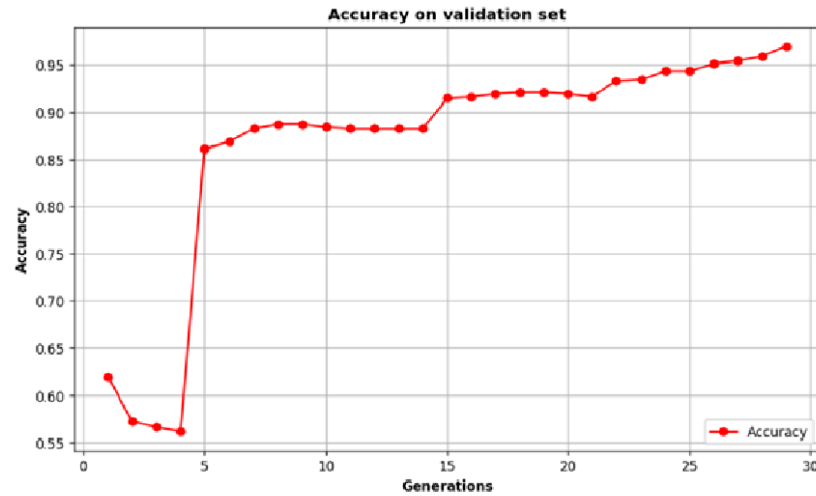

Figure 4.7: Accuracy as a function of the generation number on the validation set, related to run T_7.

the classification results, it is useful to provide the user with the most representative subspaces, namely the sets of features, which allow the separation of the two classes. Considering the faults class, we want to individuate the most frequent features, that are selected at least 20 times in the final model, over multiple simulations with a value of accuracy higher than 91%, in recognizing grid malfunction states. Figure 4.8 shows the occurrences of the chosen features by E-ABC. In other words, the bar plot represents the main subspaces in which E-ABC found meaningful clusters, whose overall number is represented by bars height. For example, in the first case, given the selected metrics we deduce that faults in this subspace are due mainly to meteorological phenomenons (presence of rain and abnormal temperatures, for instance). This fact may be related to equipment ageing that are more subject to sudden changes in atmospheric temperature. Moreover, in the second case, the related clusters characterize a sub-class of failures mainly due

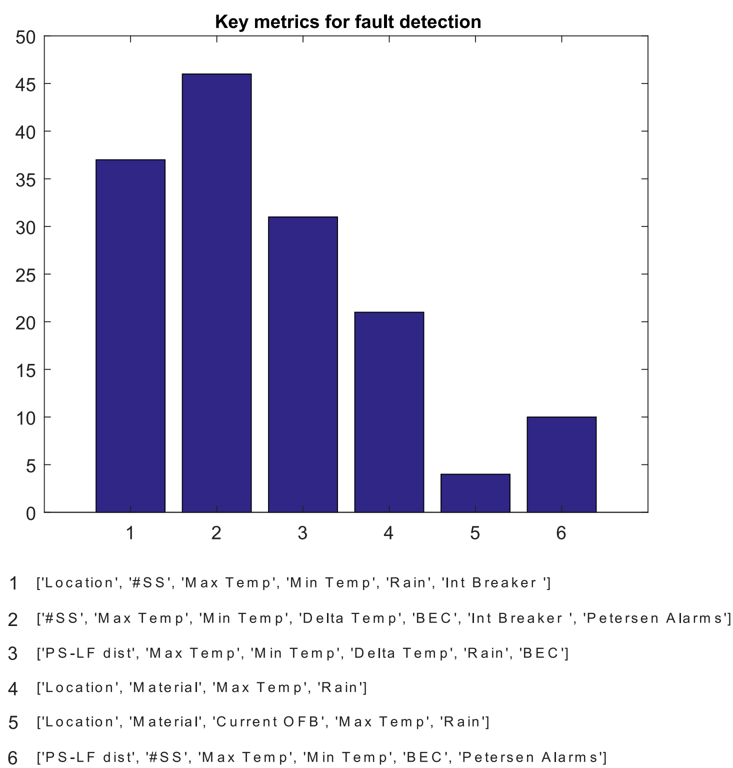

Figure 4.8: The occurrences of key metrics mostly chosen by E-ABC.

to to the sudden changes of electric current ("BEC") on aged components, as suggested by the presence of micro-interruptions related features. This is also confirmed by the hands-on experience gained over time by Acea experts.

All in all, preliminary tests with E-ABC^2 showed satisfactory classification capabilities both on the Iris data set and the Acea data set. However, a major drawback of E-ABC^2, at least in its current form, is the very poor exploration capability: the last generation used to return few subspaces with respect to our expectations. The elite pool somehow contributed towards better exploration, yet further improvements have been done and will be addressed in chapter 5.

Chapter 5

Experimental Results

In this chapter we are going to analyze extensive tests performed with C++ version of E-ABC^2 described in appendix A to have statistically valid results. These tests are grouped in three subsets: the first one was performed with the vanilla E-ABC^2 highlighting some weaknesses as it was originally designed and requiring an additional software development cycle design–implement–check; second and third group verify the effectiveness of changes proposed to face these weaknesses. We considered it was appropriate to show intermediate results because it is interesting the observed behavior related to the flaw causing it.

5.1 Data Set Description

For this tests we used two data sets: the modified Iris described in 4.2.2 and an additional synthetic data set conceived to stress E-ABC^2 local metric learning capability. Synthetic data set scheme is summarized in table 5.1. It

Table 5.1: Schematic representation of the clusters' structures for synthetic data set

	1	2	3	4	5	6	7	8	9	10	11	12	13	14-20
C_1	X	X	X	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$
C_2	$\sim$	$\sim$	$\sim$	X	X	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$
C_3	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$	O	O	O	$\sim$	$\sim$	O	O	O	$\sim$
C_4	O	O	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$	O	O	$\sim$	$\sim$	$\sim$	$\sim$
C_5	$\sim$	$\sim$	O	O	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$
C_6	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$	X	X	X	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$
C_7	$\sim$	$\sim$	$\sim$	$\sim$	O	O	O	O	O	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$
C_8	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$	X	X	X	X	X	$\sim$

is composed of 8 clusters relying in different subspaces, each one assigned to one out of two classes as it is shown in table 5.1 by "X" and "O" symbols. All patterns are in unitary hypercube and they are uniformly distribute in an hypersphere with fixed radius. The hypersphere is visible in features where is "X" or "O". Features marked with $\sim$ are not informative for that cluster: patterns in that cluster have $[0, 1]$ uniformly drawn values for that feature. Features 14-20, collapsed in a single column, are not informative for any cluster. Each cluster contains 3000 patterns. Exactly 4 clusters are assigned to "X" (in the following referred as class 1) and 4 assigned to "O" (referred as class 2) so as to have perfectly balanced classes. The whole data set is composed of 24000 patterns.

For both data sets, Iris and Synthetic, we have built in advance 100 splits in training, validation and test set (respectively S_{tr}, S_{vl} and S_{ts}) so as to be able to run one test for different data setups and different tests with the same input data to compare their performance.

5.2 Evaluation Metrics

To evaluate E-ABC^2 we built the multi-class confusion matrix over S_{ts}. Because in some situations our algorithm may return *unclassified* for some patterns we have slightly modified the confusion matrix structure to include this information. Consider the example in figure 5.1. Suppose our data set

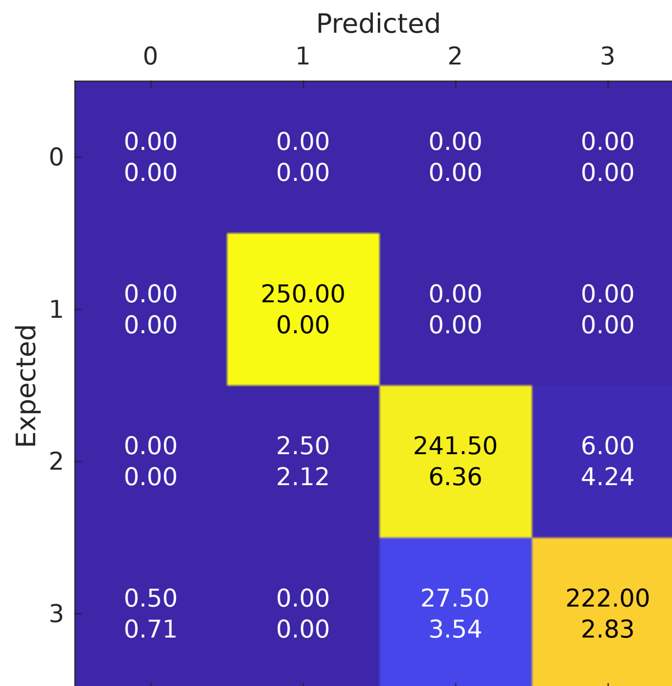

Figure 5.1: An example of confusion matrix summarizing multiple tests. Each cell contains mean value (above) and standard deviation (below) for the same cell in different tests.

contains three classes: row index is the expected class and column index is the predicted class. The first column (labeled 0) tells us how many patterns the system was not able to classify (returning *unclassified*). The first row will be always empty because each pattern in data set is supposed to have

a ground truth class assigned. There are several parameters which can be derived from the confusion matrix, each one highlighting a specific aspect of the classifier performance, but we chose to show the confusion matrix itself because we found it better readable at glance. Because the aim of these experiments was to be an extensive test, we have many confusion matrices. Most of experiments include 10 runs over 100 data set splits in training, validation and test set. We could not show 1000 confusion matrices, so we had to summarize them with a single plot. We considered it appropriate to compute cell by cell mean value and standard deviation and show them with two lines in the same cell as shown in figure 5.1. Where it seemed useful we also showed the accuracy. In these specific tests it is informative because the date sets are perfectly balanced and it has the advantage (with respect to *precision*, *recall*, *specificity* and F_1-*score*) of being a global value instead of requesting to be computed class by class.

5.3 Vanilla E-ABC^2

The first test we performed was the comparison of E-ABC^2 with its evolutionary procedure enabled against E-ABC^2 running a random walk: at each generation all agents are randomly drawn. All of the tests in this section 5.3 have been performed with 10 runs for each of 100 splits. This test was conceived to check the effectiveness of E-ABC^2 evolutionary procedure. Despite the difference is not so clear in figure 5.3 that compares the performance on Iris in terms of accuracy distribution, it becomes evident in figure 5.6 comparing the same kind of plot obtained for Synthetic. Despite these results

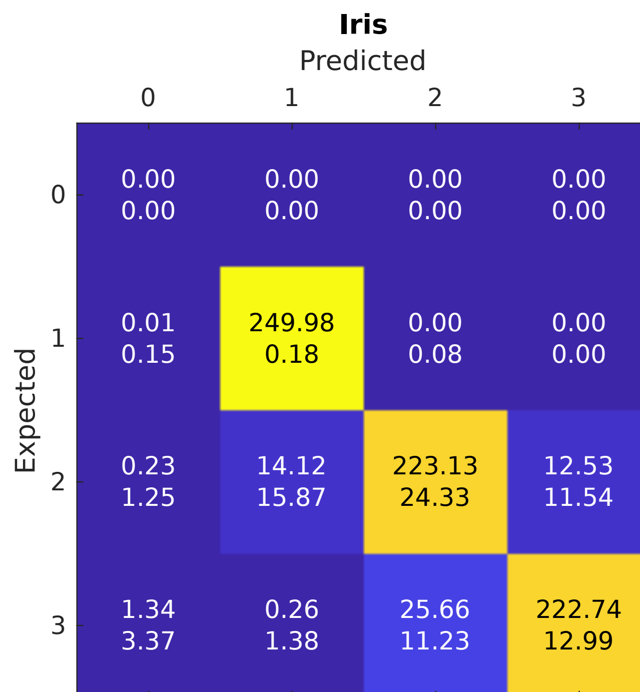

Figure 5.2: Average confusion matrix for vanilla E-ABC^2 on Iris data set

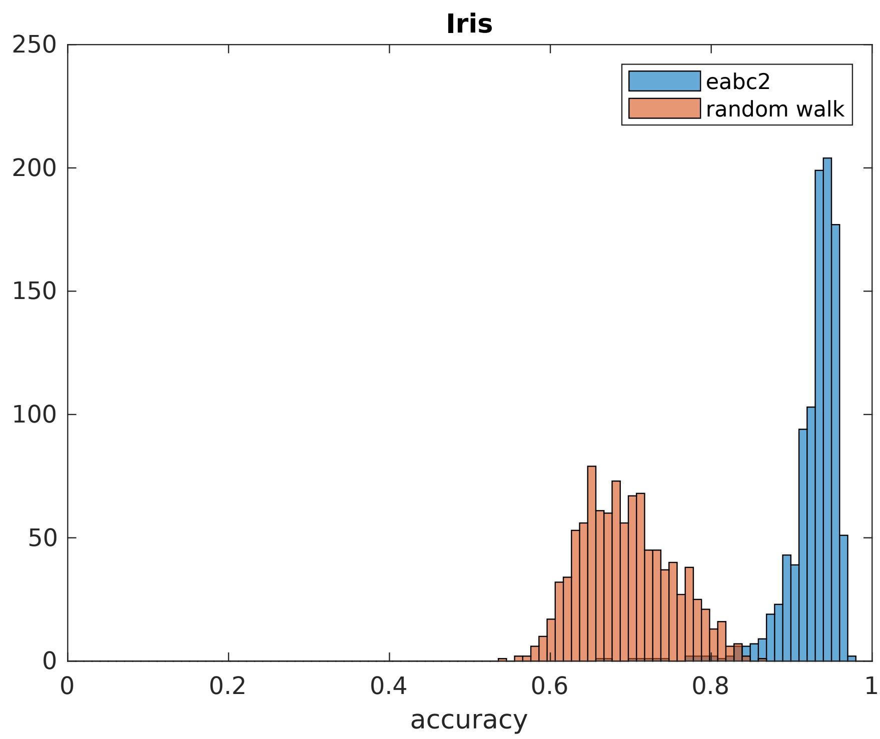

Figure 5.3: Accuracy distribution comparison on Iris data set

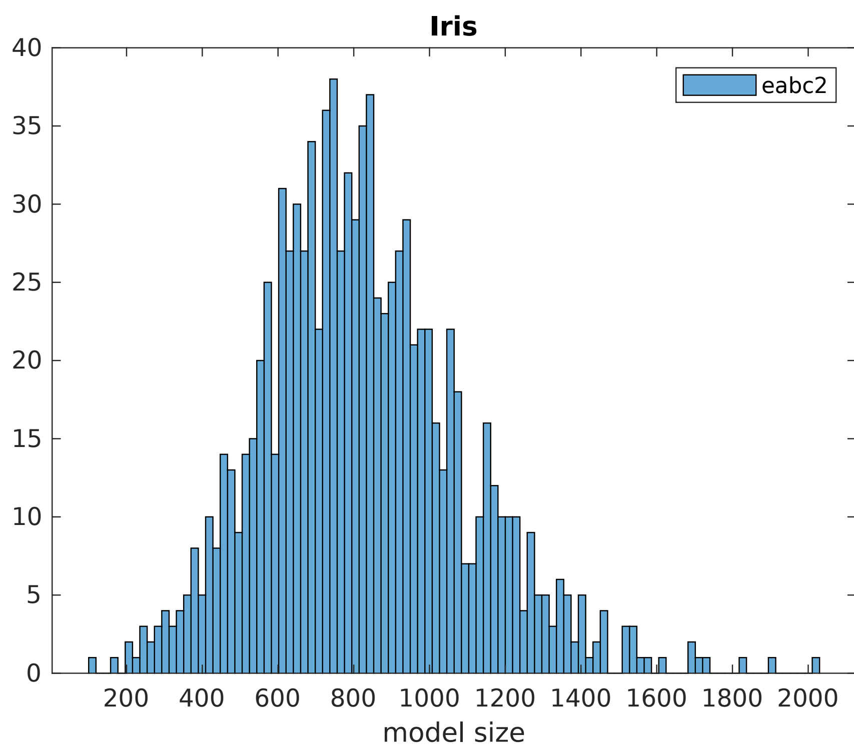

Figure 5.4: Model size distribution for vanilla E-ABC^2 on Iris data set

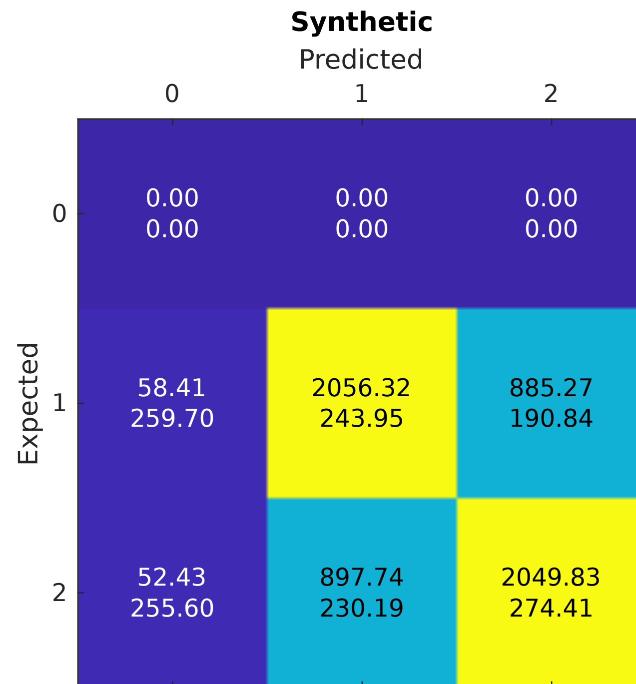

Figure 5.5: Average confusion matrix for vanilla E-ABC^2 on Synthetic data set

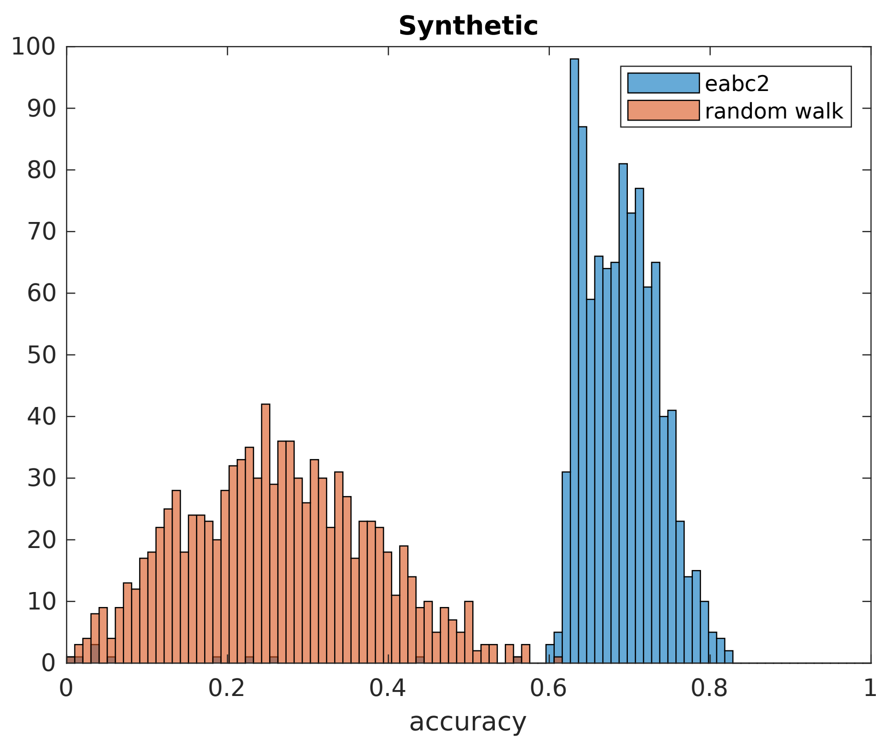

Figure 5.6: Accuracy distribution comparison on Synthetic data set

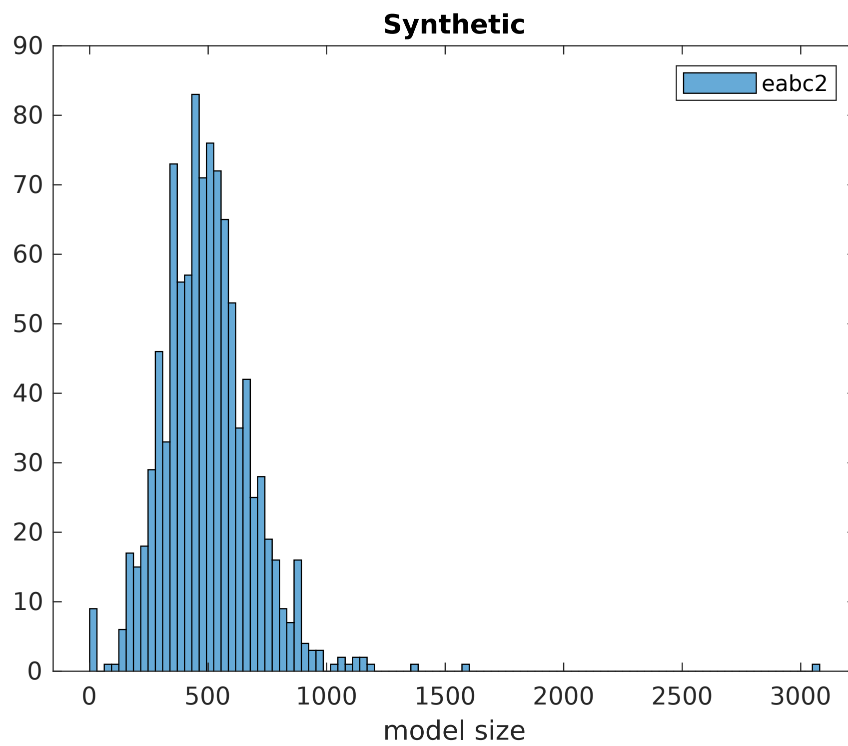

Figure 5.7: Model size distribution for vanilla E-ABC^2 on Synthetic data set

Table 5.2: Tests highlight the impossibility for this evolutionary scheme of finding different subspaces at the same time. At most it manages to find a compromise fitting multiple clusters but not all of them.

	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8
0	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
1	1,00	0,04	0,05	0,19	0,04	0,05	0,07	0,05
2	1,00	0,05	0,57	1,00	0,58	0,06	0,56	0,06
3	1,00	0,43	0,54	1,00	0,57	0,42	0,54	0,42
4	1,00	0,43	0,55	1,00	0,57	0,42	0,54	0,44
5	1,00	0,42	0,56	1,00	0,59	0,40	0,55	0,40
6	1,00	0,41	0,56	1,00	0,61	0,40	0,56	0,41
7	1,00	0,41	0,56	1,00	0,62	0,40	0,57	0,41
8	1,00	0,41	0,57	1,00	0,63	0,39	0,57	0,40
9	1,00	0,40	0,57	1,00	0,66	0,39	0,58	0,39
10	1,00	0,41	0,57	1,00	0,66	0,39	0,58	0,39
11	1,00	0,40	0,58	1,00	0,66	0,38	0,58	0,39
12	1,00	0,41	0,58	1,00	0,67	0,39	0,58	0,39
13	1,00	0,41	0,58	1,00	0,67	0,38	0,58	0,39

are acceptable for a classification algorithm in development, they immediately highlight two aspects of E-ABC^2 to be improved: model sizes, whose distributions are shown in figure 5.4 and 5.7, are too large and the accuracy degrades when real local metric learning is required. For this second defect, additional investigation revealed that the elite pool mechanism was not working as it was expected: when a good subspace is found it starts spreading and saturates the elite pool, preventing exploration of new potentially useful subspaces. It is also confirmed in table 5.2 showing generation by generation global model accuracy evaluated for each of Synthetic 8 clusters. We have associated the problem of model size to some behaviors of the evolutionary procedure and model building. First of all when a cluster enters the model there is no mechanism to remove it, neither if it could be substituted by a better one including its classification capabilities. Secondly there is no check

to verify if a given cluster has already been added to the model. In this way, when a good agent is found and its cluster is added to the model, it is likely a new agent derived from this in the same generation will find a similar cluster, and multiple copies of the same clusters are added to the classification model.

5.4 Fitness Sharing

We have found this kind of problem where multiple optima have to be found at the same time by an evolutionary algorithm is known in literature as Evolutionary Multimodal Optimization [83] [84]. We removed the elite pool and introduced a Fitness Sharing based approach [85] [86]: after each agent fitness has been evaluated an additional step occurs to modify it. As if they where sharing a limited resource, close agents (in terms of search subspace) are penalized. The more agents share the same subspace, the more their fitness value is reduced. In this way agents are encouraged to explore new subspaces that no other agents are investigating. As it was expected Iris is not affected by this change as it is shown by its confusion matrix 5.8 and plot 5.9 (all useful Iris features are in the same subspace), whereas the improvement is evident for Synthetic in its confusion matrix 5.11 and accuracy distribution plot compared to the previous implementation in figure 5.12. An additional unexpected but welcome side effect is the model size reduction observable in figures 5.10 and 5.13 if compared to figures 5.4 and 5.7. This is likely attributable to an higher efficiency in the evolutionary process. Also detailed final accuracies in table 5.3 confirm the better subspace

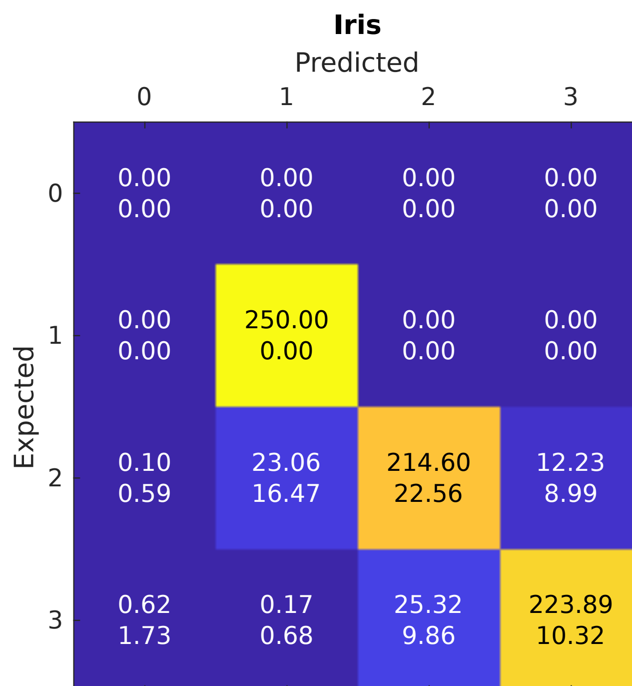

Figure 5.8: Average confusion matrix for multimodal E-ABC^2 on Iris data set

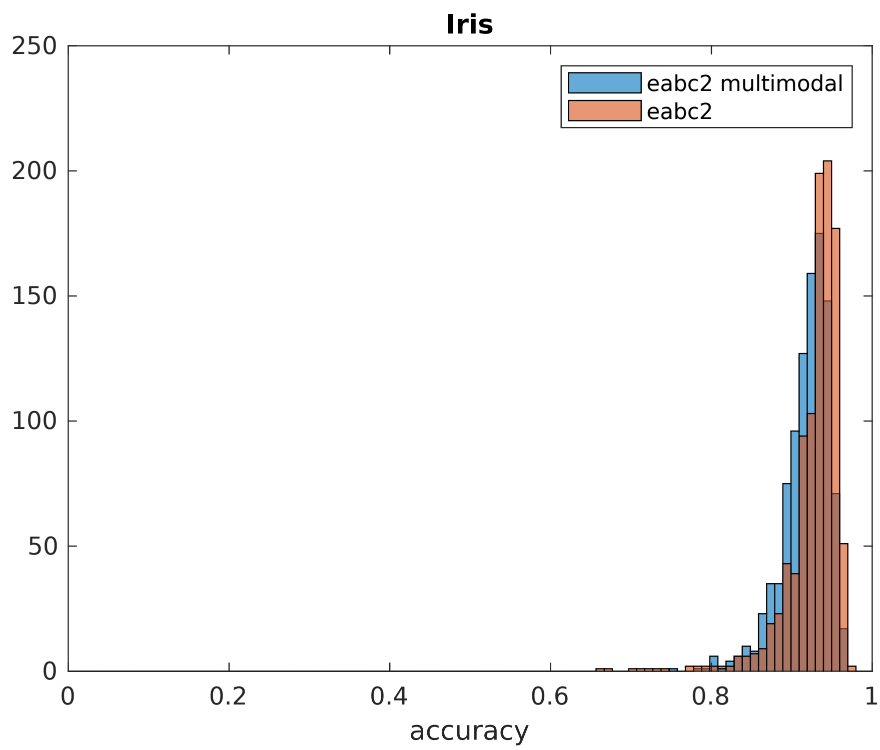

Figure 5.9: Accuracy distribution comparison on Iris data set

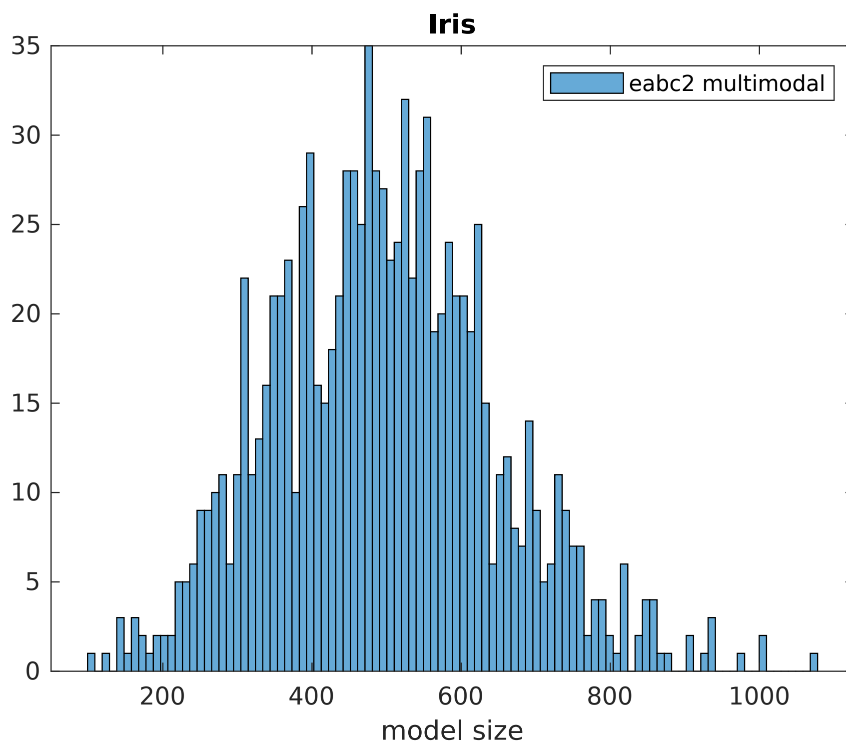

Figure 5.10: Model size distribution for multimodal E-ABC^2 on Iris data set

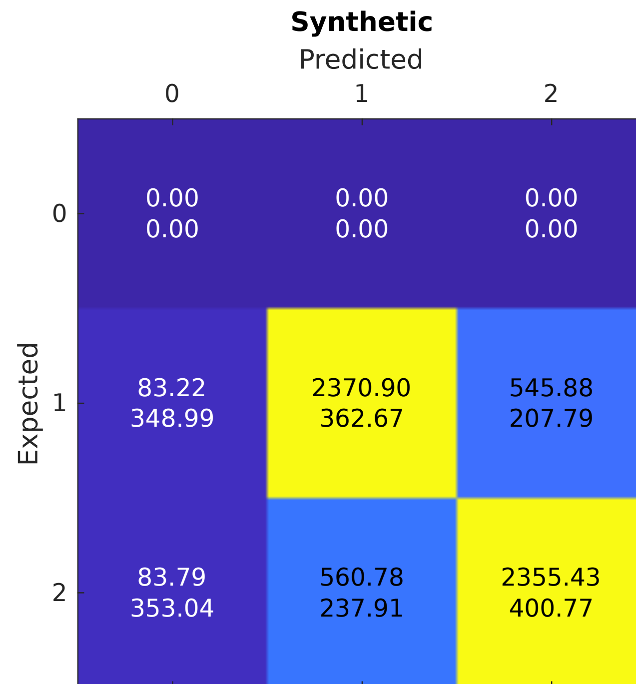

Figure 5.11: Average confusion matrix for multimodal E-ABC^2 on Synthetic data set

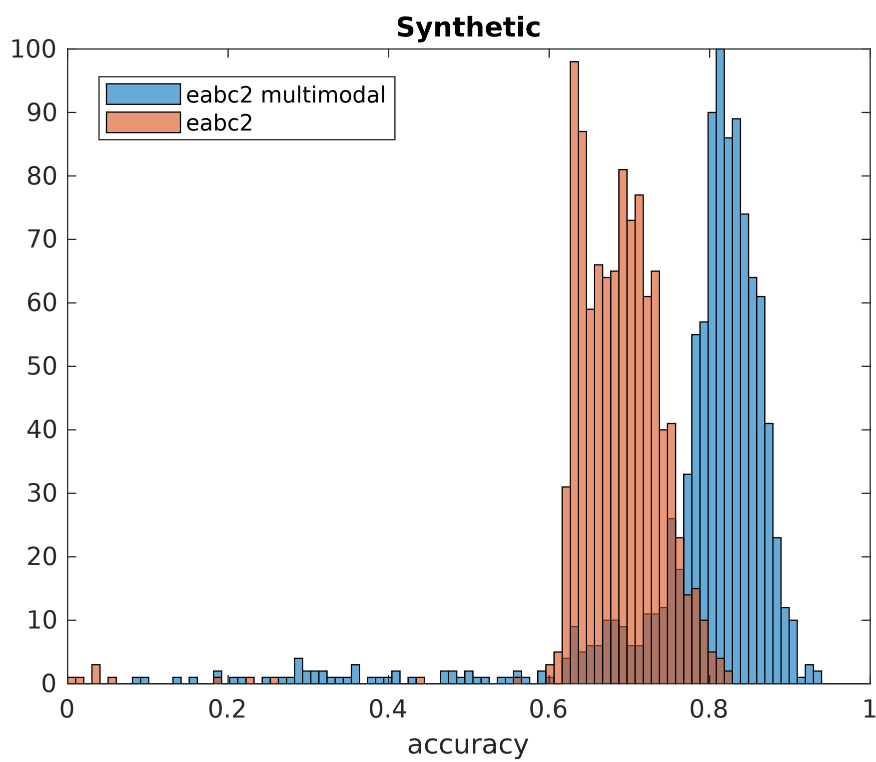

Figure 5.12: Accuracy distribution comparison on Synthetic data set

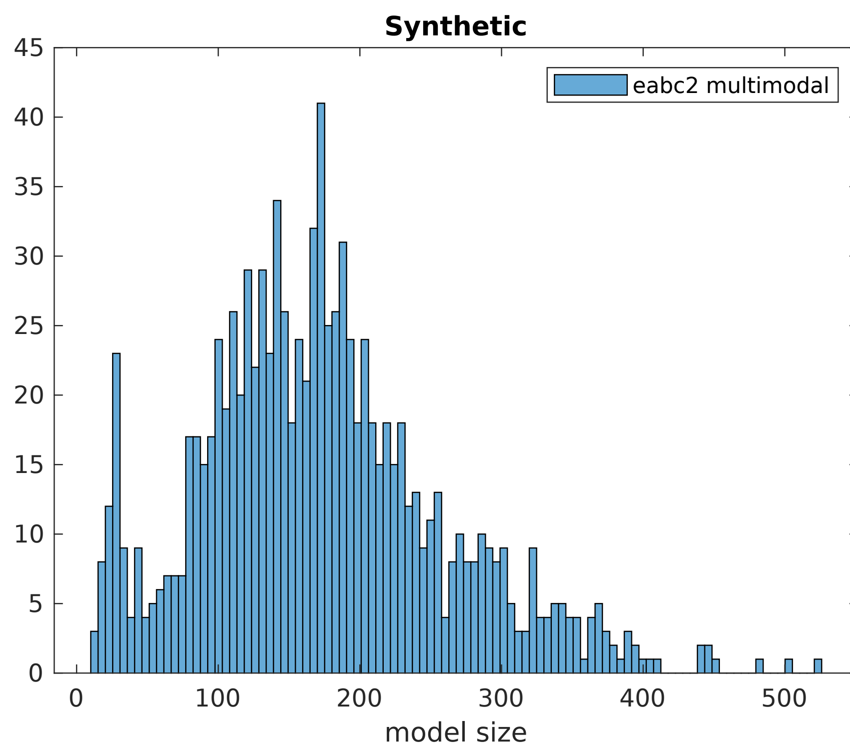

Figure 5.13: Model size distribution for multimodal E-ABC^2 on Synthetic data set

Table 5.3: Multimodal implementation shows an highly better capability for multiple subspaces detection. Despite not all of clusters are perfectly classified, for all of them the accuracy is more than acceptable.

	C_1	C_1	C_1	C_1	C_1	C_1	C_1	C_1
0	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
1	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
2	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
3	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
4	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
5	0,01	0,00	0,00	0,00	0,00	0,00	0,00	0,00
6	0,03	0,00	0,00	0,00	0,00	0,00	0,00	0,00
7	0,06	0,02	0,00	0,00	0,00	0,12	0,00	0,02
8	0,06	0,02	0,00	0,00	0,00	0,12	0,00	0,02
9	0,06	0,02	0,00	0,00	0,00	0,12	0,00	0,02
10	0,06	0,02	0,00	0,00	0,00	0,12	0,00	0,02
...								
74	0,89	0,83	1,00	0,71	0,68	0,66	0,95	1,00
75	0,89	0,86	1,00	0,71	0,68	0,67	0,95	1,00
76	0,89	0,86	1,00	0,71	0,68	0,67	0,95	1,00
77	0,89	0,86	1,00	0,71	0,68	0,67	0,95	1,00

detection if compared to table 5.2.

5.5 Model Pruning

Despite its simple formulation model size reduction is a task not to be underestimated. Given a classification model containing n clusters the computational cost to find the optimal subset to be used in the final classification model requires 2^n non-trivial tests: each of n clusters can be used or unused and for each configuration the whole validation set must be classified. The exhaustive search is then unfeasible. We have started performing some experiments with some heuristics but the development is still far away from being satisfactory: all of them are still too time consuming. In particular we tested a greedy approach and a genetic algorithm based one but we could complete an extensive test only with the first one and only with the smaller Iris data set. The greedy approach starts with an empty model and iteratively tries to add all the remaining clusters, one by one, to the current model. At each iteration it chooses the cluster causing the highest accuracy increment. It stops when no accuracy increment is possible. We underline once again how this still require much development and this is not to be supposed being an extensive test, but we want to show an example because the results are extremely promising. Classification model sizes shown in figure 5.16 are extremely lower, without performance degradation that instead is slightly better (figures 5.14 and 5.15). This confirms the model can be pruned and at the same time improved by discarding the worst clusters collected in the classification model.

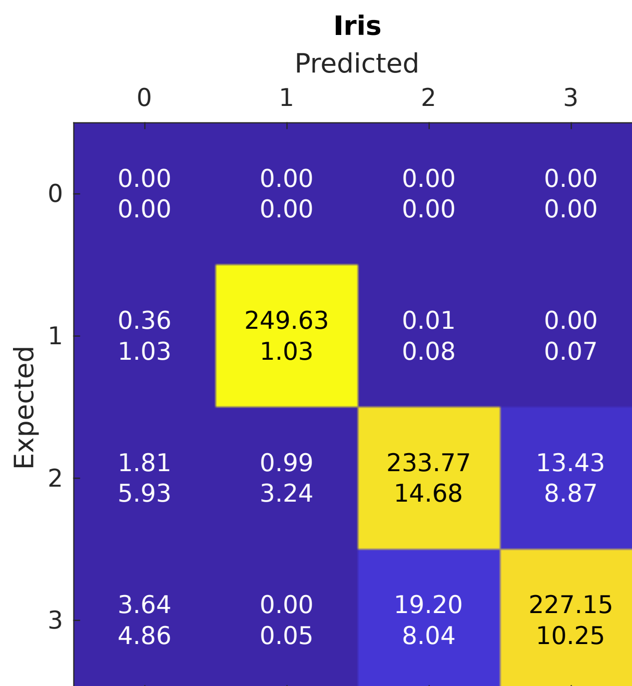

Figure 5.14: Average confusion matrix for E-ABC^2 with greedy model pruning on Iris data set

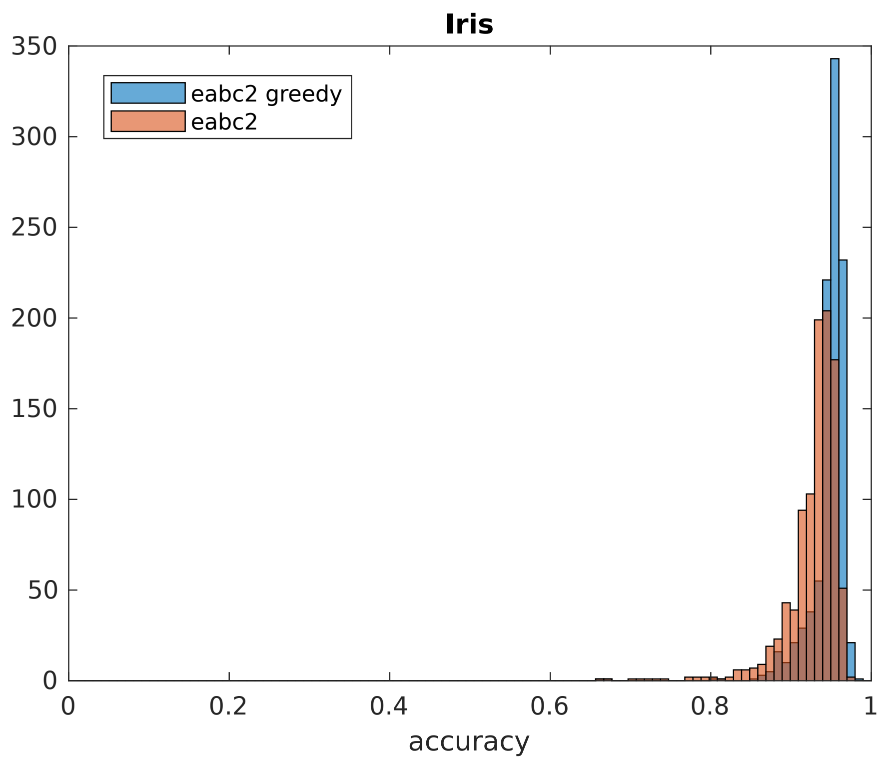

Figure 5.15: Accuracy distribution comparison on Iris data set

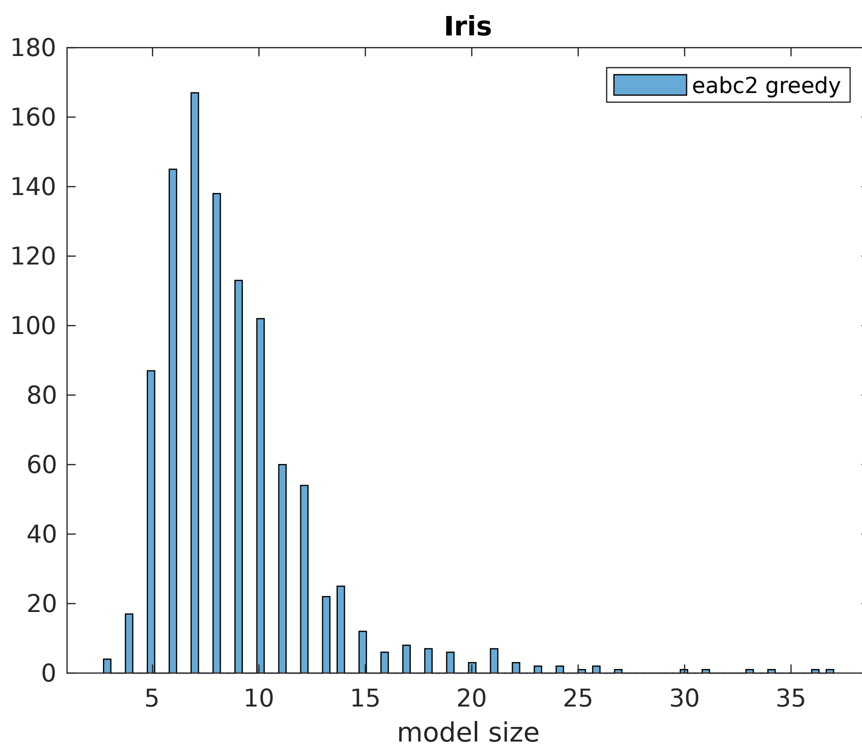

Figure 5.16: Model size distribution for E-ABC^2 with greedy model pruning on Iris data set

We are actively working to develop a distributed software to face this task taking advantage of the gossip protocol for computation of aggregate information as it is proposed in [87].

5.6 Multimodal E-ABC^2 Scalability

Due to results shown in §5.4, multimodal version has been taken as the reference for E-ABC^2 algorithm. It has been optimized as described in §3.4.4 and the performance in terms of time consumption has been analyzed. In order to address the scalability of the parallel E-ABC^2 implementation, the well-known speedup index has been considered [11, 54, 71, 72]. Speedup measures the ability of the parallel and distributed algorithm to reduce the running time as more workers are considered. The data set size is kept constant and the number of workers increases from 1 to m. Hence, the speedup with m computational units reads as

$$speedup(m) = \frac{\text{running time on 1 worker}}{\text{running time on } m \text{ workers}} \tag{5.1}$$

Tests have been performed with two different hardware configurations: a workstation equipped with two 6-cores Intel® Xeon® E5-2620v3 CPUs @2.40GHz, 64GB of RAM, running Linux Ubuntu 18.04 and a consumer level desktop computer equipped with a 6-cores Intel® Core™ i7-8700K CPU @3.70GHz, 32GB of RAM, running Linux Mint 19 (Ubuntu 18.04 derivative). Because E-ABC^2 is a probabilistic algorithm, we fixed the seed of pseudo-random function to compare the results of different hardware setups

and pool thread sizes with exactly the same task. Figures 5.17 and 5.18 respectively show the speedup with Xeon® E5-2620v3 and Core™ i7-8700K CPUs for different sizes of data chunk provided to each agent for clustering. In both cases the maximum thread pool size is bounded to the number of available physical cores for the hardware setup (12 for dual Xeon® E5-2620v3 and 6 for Core™ i7-8700K). The desired speedup behavior, expected

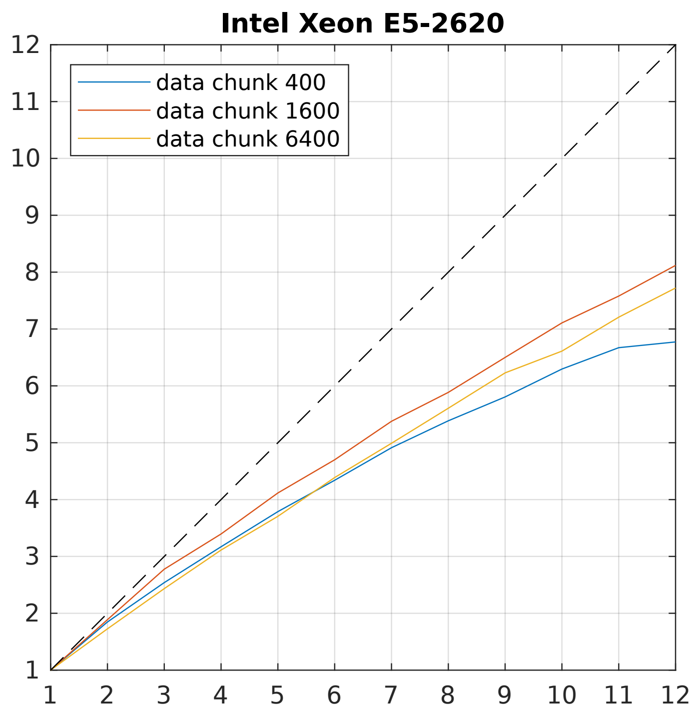

Figure 5.17: Speedup with dual Intel Xeon E5-2620 processor

if the task is fully parallelizable, is perfectly linear with derivative equal to 1 (dashed line). Indeed, with this premise, an m-times larger computing power will take m times less time to process a given data set. Because a real implementation always introduce inefficiencies with respect to the ideal behavior and E-ABC^2 is only partially parallelized, we expect real speedup plots below

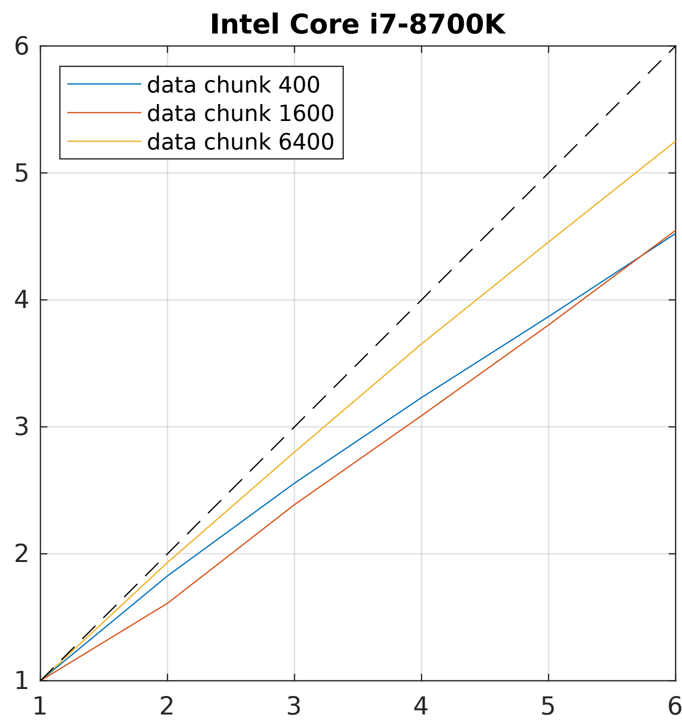

Figure 5.18: Speedup with Intel Core i7-8700K processor

the ideal line. For this test we generated an 800000-patterns data set with the same structure of Synthetic data set described in §5.1 (100000 patterns each cluster). As it was expected plots curvature reduces as data chunk size increases. It is because data chunk size increment reduces the relative weight of constant (not affected by parallelization) time consumption components whose contribution becomes negligible. In Xeon® E5-2620v3 chart the plot for data chunk containing 6400 patterns is unexpectedly below the plot with data chunk size equal to 1600. It is unexpected because, with the growing of data chunk size, the contribution in time consumption of parallelizable steps should increase, and with it also the benefit of taking advantage of a larger thread pool. It could be addressable to different situations in evolutionary procedure for different data chunk sizes, but this uneven behavior is not shown by Core™ i7-8700K chart (we remind we fixed the pseudo-random function seed so that, given a data chunk size, steps needed by the algorithm to converge to a solution are deterministic). This difference, jointly with the smaller Core™ i7-8700K plots curvature, highlights that the speedup not only depends on the algorithm design and implementation, but is also deeply affected by the hardware configuration running it.

Chapter 6

Conclusions

In this thesis we proposed an agent-based clustering algorithm orchestrated by a genetic algorithm in order to perform feature selection during clustering (E-ABC) and classification (E-ABC^2). Both E-ABC and E-ABC^2 rely on a swarm of many independent agents, each of which exploits a random data set subsample, with the final goal of extracting well-formed clusters lying in suitable subspaces. Agents therefore behave as the main actors in the local metric learning task.

Specifically, we start our development with E-ABC, an algorithm for unsupervised learning in which the final output can be briefly summarized as the set of clusters, along with the subspaces in which they lie. A second algorithm, E-ABC^2, aims at building a decision clusters-based classifier on the top of the discovered clusters. Hence, it can safely be used for supervised learning. The agents in both E-ABC and E-ABC^2 are driven by a genetic algorithm with the goal of guiding the agents themselves, following an evolutionary metaheuristic fashion, towards suitable solutions. For E-ABC, the

objective is to find well-formed clusters; whereas for E-ABC^2, the objective mainly takes into account the classifier performances.

Preliminary results both on real world data provided by Acea and on synthetic data showed encouraging results and paved the way for further improvements, including a multimodal version of the genetic optimization in order to foster exploration towards different subspaces. Further computational results addressed these improvements.

This first implementations of E-ABC and E-ABC^2 allow for further improvements and future research. First, E-ABC and E-ABC^2 can be considered as "stubs", in the sense that the philosophy behind the algorithm stays the same, but the building blocks as such can easily be customized according to the data analyst needs. Indeed, any (possibly multimodal) evolutionary mehauristic can be placed instead of a genetic algorithm and any clustering algorithm can be placed instead of the RL-BSAS.

Further, a straightforward improvement is based on a parallel and distributed implementation of E-ABC and E-ABC^2 by exploiting multi-core and many-core architectures or, by considering the small computational burden, by leveraging on low-cost architectures such as Raspberry Pi, Arduino and Parallella boards. Tests on E-ABC^2 with parallel clustering execution and agents performance evaluation show it may benefit of multi-core hardware. Additional tests are needed to quantify the extent of this benefit, to determine how to reduce the bottleneck due to genetic algorithm orchestration (possibly by making it distributed and asynchronous) and other single-thread steps and to better investigate how the speedup of parallel implementations is affected by specific multi-core and multi-machine hardware configuration.

We are also planning additional development of parallel and asynchronous paradigms to make model pruning feasible and to reduce model size and make it always up to date with respect to the analyzed data set during the evolutionary procedure. This approach would also make E-ABC^2 usable with infinite data streams where a continuous model adaptation is required to fit time dependent data changes.

Another interesting aspect is the extension of E-ABC towards structured data such as graphs and sequences: thanks to the source code modularity, level of abstraction and utilization of polymorphism, this extension can also be considered straightforward.

All in all, the encouraging performances of E-ABC and E-ABC^2 allows them to be considered as a prospective building block for complex decision support systems suitable to be used in a plethora of industrial applications.

Appendix A

Software Design

For the software implementation of our algorithm we looked mainly at two alternatives: Python and C++. Python main strength is growing interest for this language in Machine Learning community due to popular handy dedicated libraries such as SciPy, Keras and TensorFlow. On the other hand C++ is dramatically faster, being a compiled language, whereas the same is not true for Python, which is interpreted. It is a common approach in optimized Python libraries to make use of C++ source for time consuming procedures [88, 89, 90]. Nevertheless, C++ allows more effective memory management, which is a useful feature for software supposed to deal with lots of data. A first Python implementation has to be be imputed to faster prototyping and easier syntax, with a subsequent porting in C++ for efficiency.

The C++ version of E-ABC^2 makes an extensive use of *adapter* inspired design pattern [91] and polymorphism. In the following we show some short code snippets to support the description of these principles usage in our soft-

ware. For complete declarations of concerned classes we refer to appendix B. E-ABC^2 has been conceived to be used for different kinds of patterns without the need of modifying or rebuilding it. It is decided at runtime which is the data type E-ABC^2 is going to deal with. Because different data types require different procedures to compute pattern-to-pattern dissimilarity, E-ABC^2 must be taught about how to compute it. For this purpose we use a kind of adapter: E-ABC^2 constructor receives a pointer to a *Metric* object encapsulated in other adapters. In our design we decided to leave open the road to other implementations using a different core clustering algorithm instead of RL-BSAS, so we also included a pointer to a *Clustering Algorithm* functor [91] telling E-ABC^2 how to cluster patterns. An analogous discussion can be done for *Evolver* in charge of defining genetic operators to build a new population given the current one sorted by fitness.

```
1  HyperpointD d_14D;
2  d_14D.load("TestData/iris_14D.json.bz2");
3  vector<float> weights14(14, 1);
4  HyperpointM* metricModel = new HyperpointM(weights14);
5  HyperpointC* clusterModel = new HyperpointC(metricModel)
      ;
6  RlBsas_config* clustalgConf = new RlBsas_config(0.3f, 3,
       metricModel, clusterModel, 0.125f);
7  RlBsas* clustalg = new RlBsas(clustalgConf);
8  EabcAgent* agentModel = new EabcAgent(clustalg, d_14D.
      create());
9  HyperpointME* metricEvolver = new HyperpointME(
      static_cast<uint32_t>(weights14.size()));
10 RlBsasE* algconfEvolver = new RlBsasE();
11 EabcAgentE* agentEvolver = new EabcAgentE(metricEvolver,
       algconfEvolver, 100);
12 Eabc eabc(agentModel, agentEvolver, &d_14D);
13 eabc.train();
```

```
14
15 DecisionClusters* eabcOutModel = dynamic_cast<
      DecisionClusters*>(eabc.getClassifier());
16 eabcOutModel->store("TestData/iris_14D_model.json.bz2");
```

```
 1 HyperpointD d_20D;
 2 d_20D.load("TestData/synthetic_20D.json.bz2");
 3 vector<float> weights20(20, 1);
 4 HyperpointM* metricModel = new HyperpointM(weights20);
 5 HyperpointC* clusterModel = new HyperpointC(metricModel)
      ;
 6 RlBsas_config* clustalgConf = new RlBsas_config(0.3f, 3,
       metricModel, clusterModel, 0.125f);
 7 RlBsas* clustalg = new RlBsas(clustalgConf);
 8 EabcAgent* agentModel = new EabcAgent(clustalg, d_20D.
      create());
 9 HyperpointME* metricEvolver = new HyperpointME(
      static_cast<uint32_t>(weights20.size()));
10 RlBsasE* algconfEvolver = new RlBsasE();
11 EabcAgentE* agentEvolver = new EabcAgentE(metricEvolver,
       algconfEvolver, 100);
12 Eabc eabc(agentModel, agentEvolver, &d_20D);
13 eabc.train();
14
15 DecisionClusters* eabcOutModel = dynamic_cast<
      DecisionClusters*>(eabc.getClassifier());
16 eabcOutModel->store("TestData/synthetic_20D_model.json.
      bz2");
```

Two examples above show how our software can be used to load a data set and how to use it to synthesize a classification model. The first step is to create an *HyperpointD* object, that is a *Dataset* class specialization, and use it to load the data set. Then we create instances of *HyperpointM*, *HyperpointC* and *RlBsas* which are respectively extensions of *Metric*, *Cluster* and *Clusterizer*. *HyperpointM* defines dissimilarity measure specialized for *Hyperpoint* patterns and here is used to tell *HyperpointC* cluster template how

to compute pattern-by-pattern distances. *HyperpointC* in turn defines the cluster structure and, in particular, how to build its representative. *RlBsas* is a functor for RL-BSAS algorithm execution. Once we have these objects we can instantiate *EabcAgent* to build the agent template which will be used in *Eabc* to build all the population individuals. The last step, before creating the *Eabc* instance, is the creation of *Evolvers*. *HyperpointME* contains the genetic operators for the *Metric* evolution affecting the search subspace, *RlBsasE* has the analogous role for RL-BSAS configuration and *EabcAgentE* joins them together to be able to evolve an *EabcAgents* population. At this point *Eabc* is created and is ready to run the training procedure for the given data set. When training is over, these examples also retrieve the synthesized model and store it to a convenient human readable json format serialization, compressed with bzip2 algorithm. As these examples show this software can quickly be adapted to different data sets. In this specific use case the difference is in patterns length, despite they have the same nature. Notice *weights14* and *weights20* lengths are respectively 14 and 20 and they are used to initialize *HyperpointM*. Anyway, with similarly simple changes, this source code could also be adapted for different data types manipulation.

Appendix B

Classes Declarations

B.1 E-ABC^2

Main Class

```
1  #ifndef EABC_H
2  #define EABC_H
3
4  #include "liboci_global.h"
5
6  #include "classifiertrainer.h"
7  #include "classification/models/decisionclusters.h"
8  #include "eabc/eabcagent.h"
9  #include "eabc/eabcagente.h"
10
11
12 class LIBOCISHARED_EXPORT Eabc : public
      ClassifierTrainer{
13 public:
14   Eabc(EabcAgent* agentModel, EabcAgentE* evolver,
        Dataset* dataset, uint32_t verbosity = 0);
15   Eabc(EabcAgent* agentModel, EabcAgentE* evolver,
        Dataset* trainingSet, Dataset* validationSet,
        Dataset* testSet, uint32_t verbosity = 0);
```

```
16    ~Eabc();
17    int32_t train();
18    json getHistory() const;
19    int32_t saveHistoryJson(std::string filename, bool
         compress = true) const;
20
21  private:
22    std::default_random_engine generator;
23    std::uniform_int_distribution<uint32_t>
         uniformIntDistribution;
24    std::vector<GeneticCode*> evaluatePopulation();
25    std::vector<GeneticCode*> evolvePopulation(std::vector
         <GeneticCode*> agentsGCs);
26    void findCoCaBounds();
27    float normalizeCompactness(float compactness);
28    float normalizeCardinality(uint32_t cardinality);
29    EabcAgent* agentModel;
30    EabcAgentE* evolver;
31    std::vector<EabcAgent*> agents;
32    DecisionClusters* model;
33    float minCompactness;
34    float maxCompactness;
35    uint32_t minCardinality;
36    uint32_t maxCardinality;
37    std::vector<bool> rightClassifications;
38    uint32_t _verbosity;
39    std::vector<GeneticCode*> elitePool;
40    json _history;
41    static const uint32_t maxNumGens;
42    static const float minGlobalAccuracy;
43    static const float minFitnessIncr;
44    static const uint32_t maxNumSteadyGens;
45    static const uint32_t subsampleSize;
46    static const float tradeoffCoCa;
47    static const uint32_t numEliteInjection;
48    static const uint32_t maxNumElitePool;
49    static const std::string EXTENSION_CSV;
50  };
51
52  #endif // EABC_H
```

Agent

```
1  #ifndef EABCAGENT_H
2  #define EABCAGENT_H
3
4  #include "evolvable.h"
5
6  #include "eabcagentgc.h"
7  #include "clusterizer.h"
8
9
10 class EabcAgent : public Evolvable{
11 public:
12   EabcAgent(Clusterizer* clustalg, Dataset* datasetModel
       );
13   EabcAgent(const EabcAgent& src);
14   EabcAgent* clone() const;
15   ~EabcAgent();
16   GeneticCode* getGeneticCode();
17   uint32_t setGeneticCode(const GeneticCode* gc);
18   std::vector<Cluster*> getClusters() const;
19   int32_t run(Dataset* datasample);
20
21 private:
22   Clusterizer* clustalg;
23   Dataset* datasetModel;
24   static const uint32_t initSampleSize;
25 };
26
27 #endif // EABCAGENT_H
```

Agent Evolver

```
1  #ifndef EABCAGENTE_H
2  #define EABCAGENTE_H
3
4  #include "evolver.h"
5
6  #include <random>
7
8
```

```
9  class EabcAgentE : public Evolver{
10 public:
11   EabcAgentE(Evolver* metricEvolver, Evolver*
        algconfEvolver, uint32_t populationSize);
12   EabcAgentE(const EabcAgentE& src);
13   EabcAgentE* clone() const;
14   std::vector<GeneticCode*> evolve(std::vector<
        GeneticCode*> currGCs);
15   GeneticCode* select(std::vector<GeneticCode*> currGCs,
         uint32_t numCompetitors);
16   GeneticCode* mutate(GeneticCode* currGC);
17   std::pair<GeneticCode*, GeneticCode*> crossover(
        GeneticCode* mother, GeneticCode* father);
18   std::vector<GeneticCode*> randomize(uint32_t numRanded
        );
19   uint32_t getPopulationSize() const;
20
21 private:
22   std::default_random_engine generator;
23   std::uniform_real_distribution<float>
        uniformRealDistribution;
24   std::uniform_int_distribution<uint32_t>
        uniformIntDistribution;
25   Evolver* metricEvolver;
26   Evolver* algconfEvolver;
27   uint32_t populationSize;
28   static const float mutationRate;
29   static const float crossoverRate;
30   static const uint32_t percentTournament;
31   static const uint32_t percentElitism;
32   static const uint32_t percentMutatation;
33   static const uint32_t percentCrossover;
34   static const uint32_t percentRandom;
35   static const uint32_t sumContributions;
36 };
37
38 #endif // EABCAGENTE_H
```

B.2 Hyperpoint

Metric

```
1 #ifndef HYPERPOINTM_H
2 #define HYPERPOINTM_H
3
4 #include "liboci_global.h"
5
6 #include "metric.h"
7
8 class LIBOCISHARED_EXPORT HyperpointM : public Metric{
9 public:
10   HyperpointM(std::vector<float> weights);
11   HyperpointM* clone() const;
12   bool operator==(const Metric& rhs);
13   virtual GeneticCode* getGeneticCode();
14   virtual uint32_t setGeneticCode(const GeneticCode* gc)
       ;
15   virtual float distance(const Pattern& p1, const
       Pattern& p2) const;
16   virtual float distance(const Pattern& p, const
       Representative& r) const;
17   virtual float distance(const Representative& r1, const
        Representative& r2) const;
18   json toJson();
19   int32_t fromJson(const json& jmetric);
20
21 private:
22   float distance(const std::vector<float>& c1, const std
       ::vector<float>& c2) const;
23 };
24
25 #endif // HYPERPOINTM_H
```

Metric Evolver

```
1 #ifndef HYPERPOINTME_H
2 #define HYPERPOINTME_H
3
```

```
4  #include "liboci_global.h"
5
6  #include "evolver.h"
7  #include <random>
8
9
10 class LIBOCISHARED_EXPORT HyperpointME : public Evolver{
11 public:
12   HyperpointME(uint32_t numFeatures, bool binaryWeights
       = true, uint32_t populationSize = 0);
13   HyperpointME(const HyperpointME& src);
14   HyperpointME* clone() const;
15   std::vector<GeneticCode*> evolve(std::vector<
       GeneticCode*> currGCs);
16   GeneticCode* select(std::vector<GeneticCode*> currGCs,
        uint32_t numCompetitors);
17   GeneticCode* mutate(GeneticCode* currGC);
18   std::pair<GeneticCode*, GeneticCode*> crossover(
       GeneticCode* mother, GeneticCode* father);
19   std::vector<GeneticCode*> randomize(uint32_t numRanded
       );
20   uint32_t getPopulationSize() const;
21
22 private:
23   void validate(std::vector<float>& weights);
24   std::default_random_engine generator;
25   std::uniform_real_distribution<float>
       uniformRealDistribution;
26   std::uniform_int_distribution<uint32_t>
       uniformIntDistribution;
27   uint32_t numFeatures;
28   bool binaryWeights;
29   uint32_t populationSize;
30   static const float mutationRate;
31   static const float crossoverRate;
32   static const uint32_t percentTournament;
33   static const uint32_t percentElitism;
34   static const uint32_t percentMutatation;
35   static const uint32_t percentCrossover;
36   static const uint32_t percentRandom;
37   static const uint32_t sumContributions;
38 };
39
```

```
40 #endif // HYPERPOINTME_H
```

B.3 RL-BSAS

Clustering

```
1  #ifndef RLBSAS_H
2  #define RLBSAS_H
3
4  #include "liboci_global.h"
5
6  #include <inttypes.h>
7
8  #include "clusterizer.h"
9  #include "rlbsas_config.h"
10
11
12 class LIBOCISHARED_EXPORT RlBsas : public Clusterizer{
13 public:
14   RlBsas(const RlBsas_config* config);
15   RlBsas(const RlBsas& src);
16   RlBsas* clone() const;
17   int32_t clusterize(const Dataset* dataset);
18   int32_t removeCluster(uint32_t idx);
19   GeneticCode* getGeneticCode();
20   uint32_t setGeneticCode(const GeneticCode* gc);
21
22 private:
23
24   float clusterRadius;
25   uint32_t maxNumClusters;
26   float reward;
27   float penalty;
28   float initEnergy;
29   float minEnergy;
30   std::vector<float> energies;
31   std::vector<float> radii;
32 };
33
34 #endif // RLBSAS_H
```

RL-BSAS Evolver

```
1  #ifndef HYPERPOINTME_H
2  #define HYPERPOINTME_H
3
4  #include "liboci_global.h"
5
6  #include "evolver.h"
7  #include <random>
8
9
10 class LIBOCISHARED_EXPORT HyperpointME : public Evolver{
11 public:
12   HyperpointME(uint32_t numFeatures, bool binaryWeights
       = true, uint32_t populationSize = 0);
13   HyperpointME(const HyperpointME& src);
14   HyperpointME* clone() const;
15   std::vector<GeneticCode*> evolve(std::vector<
       GeneticCode*> currGCs);
16   GeneticCode* select(std::vector<GeneticCode*> currGCs,
        uint32_t numCompetitors);
17   GeneticCode* mutate(GeneticCode* currGC);
18   std::pair<GeneticCode*, GeneticCode*> crossover(
       GeneticCode* mother, GeneticCode* father);
19   std::vector<GeneticCode*> randomize(uint32_t numRanded
       );
20   uint32_t getPopulationSize() const;
21
22 private:
23   void validate(std::vector<float>& weights);
24   std::default_random_engine generator;
25   std::uniform_real_distribution<float>
       uniformRealDistribution;
26   std::uniform_int_distribution<uint32_t>
       uniformIntDistribution;
27   uint32_t numFeatures;
28   bool binaryWeights;
29   uint32_t populationSize;
30   static const float mutationRate;
31   static const float crossoverRate;
32   static const uint32_t percentTournament;
33   static const uint32_t percentElitism;
```

```
34   static const uint32_t percentMutatation;
35   static const uint32_t percentCrossover;
36   static const uint32_t percentRandom;
37   static const uint32_t sumContributions;
38 };
39
40 #endif // HYPERPOINTME_H
```

Bibliography

[1] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[2] Rupendra Nath Mitra and Dharma P. Agrawal. 5g mobile technology: A survey. *ICT Express*, 1(3):132–137, December 2015.

[3] Tom M Mitchell. *Machine Learning.* McGraw-Hill, Boston, USA, 1997.

[4] Sergios Theodoridis and Konstantinos Koutroumbas. *Pattern Recognition.* Academic Press, 4 edition, 2008.

[5] Christopher M. Bishop. *Pattern recognition and machine learning.* springer, 2006.

[6] Alessio Martino, Alessandro Giuliani, and Antonello Rizzi. Granular computing techniques for bioinformatics pattern recognition problems in non-metric spaces. In Witold Pedrycz and Shyi-Ming Chen, editors, *Computational Intelligence for Pattern Recognition*, pages 53–81. Springer International Publishing, Cham, 2018.

[7] Antonio Di Noia, Paolo Montanari, and Antonello Rizzi. Occupational diseases risk prediction by cluster analysis and genetic optimization.

In *Proceedings of the International Joint Conference on Computational Intelligence-Volume 1*, pages 68–75. SCITEPRESS-Science and Technology Publications, Lda, 2014.

[8] Antonio Di Noia, Paolo Montanari, and Antonello Rizzi. Occupational diseases risk prediction by genetic optimization: Towards a non-exclusive classification approach. In *Computational Intelligence*, pages 63–77. Springer, 2016.

[9] Antonio Di Noia, Alessio Martino, Paolo Montanari, and Antonello Rizzi. Supervised machine learning techniques and genetic optimization for occupational diseases risk prediction. *Soft Computing*, 2019.

[10] James MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*, pages 281–297, Oakland, CA, USA, 1967.

[11] Xiaowei Xu, Jochen Jäger, and Hans-Peter Kriegel. A fast parallel clustering algorithm for large spatial databases. *Data Mining and Knowledge Discovery*, 3(3):263–290, 1999.

[12] Mihael Ankerst, Markus M Breunig, Hans-Peter Kriegel, and Jörg Sander. Optics: ordering points to identify the clustering structure. In *ACM Sigmod record*, volume 28, pages 49–60. ACM, 1999.

[13] Karl Tuyls and Gerhard Weiss. Multiagent learning: Basics, challenges, and prospects. *Ai Magazine*, 33(3):41–41, 2012.

[14] Liviu Panait and Sean Luke. Cooperative multi-agent learning: The state of the art. *Autonomous Agents and Multi-Agent Systems*, 11(3):387–434, November 2005.

[15] Peter Stone and Manuela Veloso. Multiagent systems: A survey from a machine learning perspective. *Autonomous Robots*, 8(3):345–383, 2000.

[16] Michael L. Littman. Markov games as a framework for multi-agent reinforcement learning. In William W. Cohen and Haym Hirsh, editors, *Machine Learning Proceedings 1994*, pages 157 – 163. Morgan Kaufmann, San Francisco (CA), 1994.

[17] L. Liang, H. Ye, and G.Y. Li. Spectrum sharing in vehicular networks based on multi-agent reinforcement learning. *IEEE Journal on Selected Areas in Communications*, 37(10):2282–2292, 2019.

[18] F. Pourpanah, R. Wang, C.P. Lim, X. Wang, M. Seera, and C.J. Tan. An improved fuzzy artmap and q-learning agent model for pattern classification. *Neurocomputing*, 359:139–152, 2019.

[19] H. Shayeghi and A. Younesi. Adaptive and online control of microgrids using multi-agent reinforcement learning. *Power Systems*, pages 577–602, 2020.

[20] Przemyslaw Spychalski and Ryszard Arendt. Machine Learning in Multi-Agent Systems using Associative Arrays. *Parallel Computing*, 75:88–99, Jul 2018.

[21] L. Chen, Z. Chen, B. Tan, S. Long, M. Gasic, and K. Yu. Agentgraph:

Toward universal dialogue management with structured deep reinforcement learning. *IEEE/ACM Transactions on Audio Speech and Language Processing*, 27(9):1378–1391, 2019.

[22] Y.S. Nasir and D. Guo. Multi-agent deep reinforcement learning for dynamic power allocation in wireless networks. *IEEE Journal on Selected Areas in Communications*, 37(10):2239–2250, 2019.

[23] V. Gorodetsky, O. Karsaeyv, and V. Samoilov. Multi-agent technology for distributed data mining and classification. In *IEEE/WIC International Conference on Intelligent Agent Technology, 2003. IAT 2003.*, pages 438–441, Oct 2003.

[24] Morteza Alamgir and Ulrike Von Luxburg. Multi-agent random walks for local clustering on graphs. In *Data Mining (ICDM), 2010 IEEE 10th International Conference on*, pages 18–27. IEEE, 2010.

[25] Luiz Fernando Carvalho, Sylvio Barbon, Leonardo de Souza Mendes, and Mario Lemes Proença. Unsupervised learning clustering and self-organized agents applied to help network management. *Expert Systems with Applications*, 54:29–47, 2016.

[26] Santhana Chaimontree, Katie Atkinson, and Frans Coenen. Clustering in a multi-agent data mining environment. *Agents and Data Mining Interaction*, pages 103–114, 2010.

[27] Santhana Chaimontree, Katie Atkinson, and Frans Coenen. A multi-agent based approach to clustering: Harnessing the power of agents. In *ADMI*, pages 16–29. Springer, 2011.

[28] Tülin İnkaya, Sinan Kayalıgil, and Nur Evin Özdemirel. Ant colony optimization based clustering methodology. *Applied Soft Computing*, 28:301–311, 2015.

[29] Elth Ogston, Benno Overeinder, Maarten Van Steen, and Frances Brazier. A method for decentralized clustering in large multi-agent systems. In *Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 789–796. ACM, 2003.

[30] Xiaoying Pan and Hao Chen. Multi-agent evolutionary clustering algorithm based on manifold distance. In *Computational Intelligence and Security (CIS), 2012 Eighth International Conference on*, pages 123–127. IEEE, 2012.

[31] J Park and K Oh. Multi-agent systems for intelligent clustering. In *Proc. of World Academy of Science, Engineering and Technology*, volume 11, pages 97–102, 2006.

[32] Filippo Maria Bianchi, Enrico Maiorino, Lorenzo Livi, Antonello Rizzi, and Alireza Sadeghian. An agent-based algorithm exploiting multiple local dissimilarities for clusters mining and knowledge discovery. *Soft Computing*, 5(21):1347–1369, 2015.

[33] Filippo Maria Bianchi, Antonello Rizzi, Alireza Sadeghian, and Corrado Moiso. Identifying user habits through data mining on call data records. *Engineering Applications of Artificial Intelligence*, 54:49–61, 2016.

[34] Jacopo Maria Valtorta, Alessio Martino, Francesca Cuomo, and Domenico Garlisi. A clustering approach for profiling lorawan iot de-

vices. In Ioannis Chatzigiannakis, Boris De Ruyter, and Irene Mavrommati, editors, *Ambient Intelligence: 15th European Conference, AmI 2019, Rome, Italy, November 13–15, 2019, Proceedings*, pages 1–17. Springer International Publishing, 2019. In Press.

[35] R.N. Anderson, A. Boulanger, W.B. Powell, and W. Scott. Adaptive stochastic control for the smart grid. *Proceedings of the IEEE*, 99(6):1098–1115, 2011.

[36] U.S. Department of Energy. What the Smart Grid means to Americans, 2012.

[37] B. Karimi, V. Namboodiri, and M. Jadliwala. Scalable meter data collection in smart grids through message concatenation. *Smart Grid, IEEE Transactions on*, 6(4):1697–1706, July 2015.

[38] Chen Cheng, Danning Wang, and Wenbo Shi. An intelligent monitoring system based on big data mining. In *Engineering Technology and Applications*, pages 119–126. CRC Press, 2014.

[39] Enrico De Santis, Lorenzo Livi, Alireza Sadeghian, and Antonello Rizzi. Modeling and recognition of smart grid faults by a combined approach of dissimilarity learning and one-class classification. *Neurocomputing*, 170:368–383, 2015.

[40] Shengwei Mei, Ming Cao, and Xuemin Zhang. *Power Grid Complexity*. Springer, 2011.

[41] D. Raheja, J. Llinas, R. Nagi, and C. Romanowski. Data fusion/data

mining-based architecture for condition-based maintenance. *International Journal of Production Research*, 44(14):2869–2887, July 2006.

[42] J.M. Wetzer. Maintaining future (electrical) power systems. In *Future Power Systems, 2005 Proceedings of the International Conference on*, pages 6 pp.–6, 2005.

[43] C. L. Sweetser. The importance of advanced diagnostic methods for higher availability of power transformers and ancillary components in the era of smart grid. In *Power and Energy Society General Meeting, 2011 IEEE*, pages 1–3, 2011.

[44] P.J. Werbos. Computational intelligence for the smart grid-history, challenges, and opportunities. *Computational Intelligence Magazine, IEEE*, 6(3):14–21, August 2011.

[45] Ganesh K. Venayagamoorthy. Potentials and promises of computational intelligence for smart grids. *Power & Energy Society General Meeting, 2009. PES '09. IEEE*, pages 1–6, 2009.

[46] P. J. Werbos. Putting more brain-like intelligence into the electric power grid: What we need and how to do it. In *Proceedings of the International Joint Conference on Neural Networks*, pages 3356–3359, 2009.

[47] G. K. Venayagamoorthy. Dynamic, stochastic, computational, and scalable technologies for smart grids. *IEEE Computational Intelligence Magazine*, 6(3):22–35, 2011.

[48] M A Abido, E-S M El-Alfy, and Muhammad Sheraz. Computational intelligence in smart grids: Case studies. In *Computational Intelligence for Decision Support in Cyber-Physical Systems*, pages 265–292. Springer, 2014.

[49] Enrico De Santis, Antonello Rizzi, Alireza Sadeghian, and Fabio Massimo Frattale Mascioli. Genetic optimization of a fuzzy control system for energy flow management in micro-grids. In *2013 Joint IFSA World Congress and NAFIPS Annual Meeting*, pages 418–423. IEEE, 2013.

[50] Jiawei Han, Micheline Kamber, and Jian Pei. Getting to know your data. In *Data Mining*, pages 39–82. Elsevier, 2012.

[51] J. Qin, Q. Ma, Y. Shi, and L. Wang. Recent advances in consensus of multi-agent systems: A brief survey. *IEEE Transactions on Industrial Electronics*, 64(6):4972–4983, June 2017.

[52] Santhana Chaimontree, Katie Atkinson, and Frans Coenen. A multi-agent based approach to clustering: harnessing the power of agents. In *International Workshop on Agents and Data Mining Interaction*, pages 16–29. Springer, 2011.

[53] Ken-ichi Fukui, Satoshi Ono, Taishi Megano, and Masayuki Numao. Evolutionary distance metric learning approach to semi-supervised clustering with neighbor relations. In *2013 IEEE 25th International Conference on Tools with Artificial Intelligence*, pages 398–403. IEEE, 2013.

[54] Alessio Martino, Antonello Rizzi, and Fabio Massimo Frattale Mascioli. Distance matrix pre-caching and distributed computation of internal

validation indices in k-medoids clustering. In *2018 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2018.

[55] Bruce D Lucas and Takeo Kanade. An iterative image registration technique with an application to stereo vision. In *Proceedings of Imaging Understanding Workshop*, pages 121–130, 1981.

[56] Thomas Brox, Andrés Bruhn, Nils Papenberg, and Joachim Weickert. High accuracy optical flow estimation based on a theory for warping. In *European conference on computer vision*, pages 25–36. Springer, 2004.

[57] D. G. Lowe. Object recognition from local scale-invariant features. In *Proceedings of the Seventh IEEE International Conference on Computer Vision*, volume 2, pages 1150–1157 vol.2, Sep. 1999.

[58] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. Surf: Speeded up robust features. In *European conference on computer vision*, pages 404–417. Springer, 2006.

[59] Enrico De Santis, Alessio Martino, and Antonello Rizzi. On custom-based dissimilarity measures and metric properties in pattern recognition. 2019. Submitted for Publication.

[60] Brian Kulis. Metric learning: A survey. *Foundations and Trends in Machine Learning*, 5(4):287–364, 2012.

[61] Liu Yang. Distance Metric Learning: A Comprehensive Survey, 2006.

[62] Jonathon Shlens. A tutorial on principal component analysis. *arXiv preprint arXiv:1404.1100*, 2014.

[63] Joshua B Tenenbaum, Vin De Silva, and John C Langford. A global geometric framework for nonlinear dimensionality reduction. *science*, 290(5500):2319–2323, 2000.

[64] Sam T Roweis and Lawrence K Saul. Nonlinear dimensionality reduction by locally linear embedding. *Science*, 290(5500):2323–2326, 2000.

[65] Mikhail Belkin and Partha Niyogi. Laplacian eigenmaps for dimensionality reduction and data representation. *Neural computation*, 15(6):1373–1396, 2003.

[66] Dor Kedem, Stephen Tyree, Fei Sha, Gert R Lanckriet, and Kilian Q Weinberger. Non-linear metric learning. In *Advances in Neural Information Processing Systems*, pages 2573–2581, 2012.

[67] René Vidal. Subspace clustering. *IEEE Signal Processing Magazine*, 28(2):52–68, 2011.

[68] Lance Parsons, Ehtesham Haque, and Huan Liu. Subspace clustering for high dimensional data: A review. *SIGKDD Explor. Newsl.*, 6(1):90–105, June 2004.

[69] Aurélien Bellet, Amaury Habrard, and Marc Sebban. A survey on metric learning for feature vectors and structured data. *CoRR*, abs/1306.6709, 2013.

[70] Guido Del Vescovo, Lorenzo Livi, Fabio Massimo Frattale Mascioli, and Antonello Rizzi. On the problem of modeling structured data with the

minsod representative. *International Journal of Computer Theory and Engineering*, 6(1):9, 2014.

[71] Alessio Martino, Antonello Rizzi, and Fabio Massimo Frattale Mascioli. Efficient approaches for solving the large-scale k-medoids problem: Towards structured data. In Christophe Sabourin, Juan Julian Merelo, Kurosh Madani, and Kevin Warwick, editors, *Computational Intelligence: 9th International Joint Conference, IJCCI 2017 Funchal-Madeira, Portugal, November 1-3, 2017 Revised Selected Papers*, pages 199–219. Springer International Publishing, Cham, 2019.

[72] Alessio Martino, Antonello Rizzi, and Fabio Massimo Frattale Mascioli. Efficient approaches for solving the large-scale k-medoids problem. In *Proceedings of the 9th International Joint Conference on Computational Intelligence - Volume 1: IJCCI,*, pages 338–347. INSTICC, SciTePress, 2017.

[73] Antonello Rizzi, Guido Del Vescovo, Lorenzo Livi, and Fabio Massimo Frattale Mascioli. A new granular computing approach for sequences representation and classification. In *The 2012 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2012.

[74] David E. Goldberg. *Genetic algorithms in search, optimization and machine learning.* Addison-Wesley, Reading, Massachusetts, USA, 1989.

[75] Salem Alelyani, Jiliang Tang, and Huan Liu. Feature selection for clustering: A review. *Data Clustering: Algorithms and Applications*, 29:110–121, 2013.

[76] Alessio Martino, Mauro Giampieri, Massimiliano Luzi, and Antonello Rizzi. Data mining by evolving agents for clusters discovery and metric learning. In *Neural Advances in Processing Nonlinear Dynamic Signals*, pages 23–35. Springer International Publishing, July 2018.

[77] Mauro Giampieri and Antonello Rizzi. An evolutionary agents based system for data mining and local metric learning. In *2018 IEEE International Conference on Industrial Technology (ICIT)*. IEEE, February 2018.

[78] Mauro Giampieri, Enrico De Santis, Antonello Rizzi, and Fabio Massimo Frattale Mascioli. A supervised classification system based on evolutive multi-agent clustering for smart grids faults prediction. In *2018 International Joint Conference on Neural Networks (IJCNN)*. IEEE, July 2018.

[79] Mauro Giampieri, Luca Baldini, Enrico De Santis, and Antonello Rizzi. Facing big data by an agent-based multimodal evolutionary approach to classification. In *2020 International Joint Conference on Neural Networks (IJCNN)*. IEEE, July 2020. Submitted.

[80] Enrico De Santis, Lorenzo Livi, Alireza Sadeghian, and Antonello Rizzi. Modeling and recognition of smart grid faults by a combined approach of dissimilarity learning and one-class classification. *Neurocomputing*, 170:368–383, December 2015.

[81] Enrico De Santis, Antonello Rizzi, and Alireza Sadeghian. A learning intelligent system for classification and characterization of localized faults

in smart grids. In *2017 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, June 2017.

[82] Enrico De Santis, Antonello Rizzi, and Alireza Sadeghian. Hierarchical genetic optimization of a fuzzy logic system for energy flows management in microgrids. *Applied Soft Computing*, 60:135–149, November 2017.

[83] Ka-Chun Wong. Evolutionary multimodal optimization: A short survey. *CoRR*, abs/1508.00457, 2015.

[84] Jian-Ping Li, Marton E. Balazs, Geoffrey T. Parks, and P. John Clarkson. A species conserving genetic algorithm for multimodal function optimization. *Evolutionary Computation*, 10(3):207–234, September 2002.

[85] B. Sareni and L. Krahenbuhl. Fitness sharing and niching methods revisited. *IEEE Transactions on Evolutionary Computation*, 2(3):97–106, 1998.

[86] Pietro S. Oliveto, Dirk Sudholt, and Christine Zarges. On the benefits and risks of using fitness sharing for multimodal optimisation. *Theoretical Computer Science*, 773:53–70, June 2019.

[87] D. Kempe, A. Dobra, and J. Gehrke. Gossip-based computation of aggregate information. In *44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings.* IEEE Computer. Soc.

[88] Oliver Laslett, Jonathon Waters, Hans Fangohr, and Ondrej Hovorka. Magpy: A c++ accelerated python package for simulating magnetic nanoparticle stochastic dynamics, 2018.

[89] Árpád Horváth. Running time comparison of two realizations of the multifractal network generator method. *Acta Polytechnica Hungarica*, 10(4), 2013.

[90] Semen O. Yesylevskyy. Pteros 2.0: Evolution of the fast parallel molecular analysis library for c++ and python. *Journal of Computational Chemistry*, 36(19):1480–1488, May 2015.

[91] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, 1 edition, 1994.