Feature Relevance Estimation by Evolving Probabilistic Dependency Networks and Weighted Kernel Machines

A thesis submitted to the
District University Francisco José de Caldas
in fulfillment of the requirements for the degree
of
Master of Science in Information and Communications

By
Nestor Andres Rodriguez Gamboa
Advisor
Sergio A. Rojas, PhD.

November 2013

Contents

List of Figures

List of Tables

Acknowledgements

"To God who made the world and everything in it is the Lord of heaven and earth and does not live in temples built by human hands. And he is not served by human hands, as if he needed anything. Rather, he himself gives everyone life and breath and everything else. From one man he made all the nations, that they should inhabit the whole earth; and he marked out their appointed times in history and the boundaries of their lands. God did this so that they would seek him and perhaps reach out for him and find him, though he is not far from any one of us. For in him we live and move and have our being..."
Acts 17:24-28.

I want to first thanks God for *in him I live and move and have my being*, and in his sovereign he has granted me the opportunity to do my master studies.

My entire gratefulness to my supervisor, PhD. Sergio Rojas, for his unconditional guide and counseling, for his valuable contributions, his encourage to excellence and his tremendous effort and commitment for leading me to a scientific research.

My thankfulness to my wife for her lovely patient and for being my help meet during all this time, to my little daughter for her understanding when daddy was busy, and to my parents for having continuously encouraged me to finalize my master studies.

Finally I would like to acknowledge Henry Diosa, Leidy Garzón and Harry Sanchez, members of the Arquisoft Research Group from the District University of Bogota, for the design and implementation of the `BlackBoxTester` component.

Abstract

This study focuses on the problem of estimating relevance of observed variables for a classification task in high dimensional spaces, which is known as feature subset selection or feature relevance determination by the machine learning community. The main goal of the thesis was to design of a novel feature relevance estimation method by combining techniques for estimation of dependency networks with weighted kernel machines. The method, called Kernel Iterative Estimation of Dependency and Relevance Algorithm (`Kiedra`), works within a population-based stochastic search framework where relevant (but unknown) variables from a given dataset are found by iteratively evolving a set of relevance estimation candidates. These candidates will consist of parameters of bivariate conditional probability distributions. The distributions could be seen as dependency networks of how variables influence each other. With this information a subset of the most relevant features from the original sample can be selected to perform classification, the suitability of the subset being assessed as its predictive classification accuracy. Weighted kernel classifiers are used for this purpose. The method provides additional information about dependency among such variables as a byproduct of the selection process.

This thesis also contributed to the development of `TILDA`, a novel estimation of distribution algorithm, and `Goldenberry`, a supplementary machine learning and evolutionary computation suite for `Orange`.

Chapter 1

Introduction

1.1 Aim

The main aim of this study is to design an approach for feature relevance estimation that discovers variables dependencies by combining a bivariate population-based stochastic algorithm and a kernel classifier.

1.2 Problem and motivation

Feature subset selection (`FSS`) is a machine learning technique for dimensionality reduction (see Appendix A for more information). Some `FSS` methods have been proposed in connection with kernel classifier, like the Weigthed Kernel Iterative Estimation of Relevance Algorithm (`wKiera` [42]). This algorithm considers the weight vector $\mathbf{w}$ of the kernel as a relevancy estimator of input variables. The vector is tuned by a Univariate Marginal Distribution Algorithm (`UMDA` [6]) which iteratively by a population-based technique [19] refines its univariate marginal distribution. The accuracy of the kernel classifier coupled with the weight vector candidates is taken as the `UMDA`'s cost function measurement. The authors showed promising results of this algorithm in selecting relevant variables in a number of different classification tasks, including problems with linear and non-linear hypothesis targets. This `FSS` method is designed on the basis of the independent assumption(univariate `FSS` methods, see Appendix A). This assumption popular because of their low computational cost [33] but they do not provide additional insights about data relationships and missing information about those dependencies may affect negatively the prediction accuracy of the feature subset. As an alternative approach, multivariate `FSS` methods search for feature subsets and possible dependency relationships between them at the cost of an increase in computational complexity (recall that `FSS` is a combinatorial problem known to be

NP-Hard [21,41]). The challenge is then to design novel feature selection methods that take advantage of multivariate power combined with high-accuracy classifiers to obtain improved prediction and explanatory performance. Therefore the main motivation of this study is to build upon `wKiera` a refined version that incorporates a bivariate feature selection within a population-based stochastic algorithm.

1.3 Contributions

During the development of this study in chronological order the following contributions, termed `TILDA`, `Goldenberry` and `Kiedra`, were made.

- The `T`iny `I`ncremental `L`earning `D`ensity `E`stimation `A`lgorithm (`TILDA`) is an `EDA` able to work in a continuous domain with memory-efficient capabilities. `TILDA` is a mix of the *Compact Genetic Algorihtm* (`cGA`) along with the *Population-based Incremental Learning Continuous*($\texttt{PBIL}_c$). `TILDA` proved to be able to solve large-scale continuous domain problems with minimum memory requirements. `TILDA` was the result of our `EDA`s exploration where we identified `cGA` as a memory-efficient algorithm with promising results on problems of millions of discrete variables [48] and we aimed to take advantage from it but in a continuous domain context suitable for `wKiera` . `TILDA` was published and nominated to the *Best Paper Award* in the *Genetic and Evolutionary Computation Conference - GECCO 2012, Philadelphia, USA* [43].

- `Goldenberry` is a supplementary machine learning and evolutionary computation suite for `Orange`[1]. Its main purpose is to provide an user-friendly workbench for building and testing `EDA`s upon its versatile visual front-end and the powerful reuse and glue principles of component-based software development. In our exploration phase with `EDA`s we had the need to run and compare most well-known `EDA`s by a machine learning framework to identify the most relevant for `Kiedra`. `Orange` called our attention due to its visual and scripting interfaces for many machine learning algorithms. Nonetheless, `Orange` was not able to provide an optimization nor a `EDA`'s framework. Therefore we decided to extend it and this is how the `Goldenberry`'s first version was born, as an `Orange` add-on with visual components for stochastic search-based optimization tasks. `Goldenberry`'s first version was presented in the *Genetic and Evolutionary Computation Conference - GECCO 2013, Amsterdam, Netherlands* [44]. Its latest release, new developed widgets for machine learning and feature selection have been included.

[1] `Orange` is a data mining and machine learning software suite featuring visual programming. A brief explanation is given in Section 1.4.5

- `Kiedra` is a new *wrapper* `FSS` algorithm which built upon `wKiera` and uses `BMDA` . In contrast with its predecessor, `Kiedra` considers bivariate relationships among variables by estimating iteratively a *bivariate binomial distribution* with the fittest candidates. `Kiedra` uses a kernel classifier (`Kernel Perceptron` or a `SVM`) as a mean to assess the search space goodness. The former algorithm has been tested with real datasets and it has been able to find optimal feature subsets.

1.4 Literature review

In this section we provide a summarized review of the main elements involved in this study namely: Estimation of distribution Algorithms(`cGA`, `PBIL`, `BMDA`), Feature selection techniques using estimation of distribution algorithms and Kernel Classifiers (`Kernel Perceptron` and `SVM`). For a broader literature review please refer to the Appendix A.

1.4.1 Overview of estimation of distribution algorithms

An estimation of distribution algorithm (`EDA`) is a population-based stochastic search technique which uses a probability distribution model to sample candidates to explore the search space (see Figure 1.1). This distribution is estimated iteratively with the fittest candidates until it converges or an ending criterion is met.

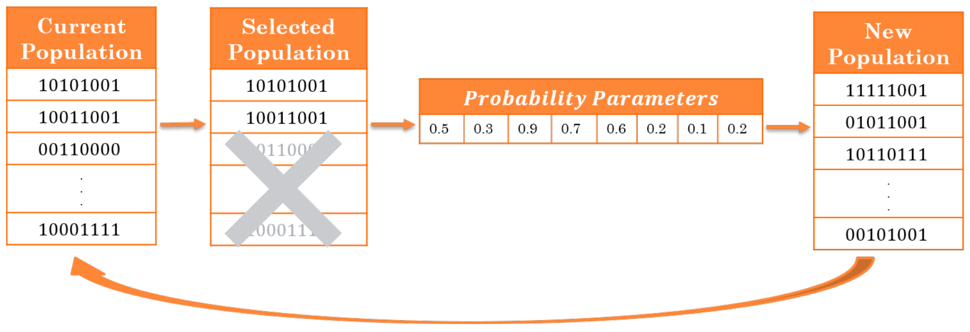


Figure 1.1: Estimation of distribution algorithms basic flow.

`EDA`s differ in how they realizes the different steps from Algorithm 1. Step (1) initializes the probability distribution parameter θ. Step (2) repeats until $P(X;\theta)$ has converged. Step (3) samples randomly a population $\mathcal{D}$ of n candidates from $P(X;\theta)$. Step (4) scores the population using a cost function $f(\cdot)$ and top-ranked candidates are selected into $\mathcal{C}$. Step (5) estimates the distribution parameter θ.

Algorithm 1 EDA pseudo-code

```
Requires: Candidate size $n$, the number variables $\ell$ and the cost function $f(\cdot)$.
1: $\theta \leftarrow \mathsf{initialize}(\ell)$
2: repeat
3:     $\mathcal{D} \leftarrow \mathsf{sample}(P(X;\theta), n)$
4:     $\mathcal{C} \leftarrow \mathsf{select}(\mathcal{D}, f(\cdot))$
5:     $\theta \leftarrow \mathsf{estimate}(\theta, \mathcal{C})$
6: until $P(X;\theta)$ has converged
Outputs: Probability distribution $P(X;\theta)$
```

The realization of Step (1) defines the different types of EDAs because it establishes the probability model (Binomial, Multinomial, Gaussian, etc), the dependency order (Univariate, Bivariate or Multivariate) and the problem domain (discrete, continuous).

Univariate EDAs

PBIL and cGA among others are two well-known EDAs. They are categorized as *discrete-univariate* because they assume binary variables that are independent. PBIL's uses a probability model factorized as the product of independent univariate marginal distributions following Eq.(1.1,1.2). The distribution parameter $\boldsymbol{\rho}$ is estimated from the pool of candidates $\mathcal{C}$, where Eq.(1.3) is used as the estimation update of Step (5). In there, α represents the learning rate and $\mathcal{C}_{ki}$ is the value of the i^{th} variable in the k^{th} candidate (see Algorithm 6).

$$P(X;\boldsymbol{\rho}) = \prod_i P(X_i;\rho_i) \tag{1.1}$$

$$P(X_i = b) = \begin{cases} \rho_i & b = 1 \\ 1-\rho_i & b = 0 \end{cases} \tag{1.2}$$

$$\rho_i = (1-\alpha)\rho_i + \alpha\frac{1}{n}\sum_{k=1}^{n}\mathcal{C}_{ki} \tag{1.3}$$

cGA has become popular for the higher efficiency to solve very large scale problems with millions to billions of variables with a lower computational demand than canonic GA [49]. This is because in cGA the population $\mathcal{D}$ in the sample Step (3) amounts to only two candidates $(\mathbf{x_1}, \mathbf{x_2})$. They both compete and the winner $\mathbf{x}^{\top}$ and loser $\mathbf{x}^{\perp}$ are used to update the probability model parameter $\boldsymbol{\rho}$ following Eq.(1.4) for the same probability model used in PBIL Eq.(1.1), where n is the population size[2] (see Algorithm 7).

$$\rho_i = \left[\!\!\left[\rho_i + \frac{1}{n}(x_i^{\top} - x_i^{\perp}) \right]\!\!\right] \tag{1.4}$$
$$\text{, where } [\![a]\!] = \begin{cases} 1 & a > 1 \\ a & 0 \le a \le 1 \\ 0 & a < 0 \end{cases}$$

The Bivariate marginal estimation of distribution algorithm

Notation: We denote a graph $\mathsf{G}(\mathbf{V}, \mathbf{E})$ where $\mathbf{V}$ is a set of vertexes and $\mathbf{E}$ is the set of pairs of elements from $\mathbf{V} \times \mathbf{V}$ representing edges whose weights are given by $w : \mathbf{E} \to \mathbb{R}$. A *disconnected* graph is a graph with k disjoint components $\mathbf{V} = \{\mathbf{V}_k \subseteq \mathbf{V} : \bigcap_k \mathbf{V}_k = \emptyset \wedge \bigcup_k \mathbf{V} = \mathbf{V}\}$ where each component $\mathbf{V}_k$ is a *connected* subgraph whose edges $\mathbf{E}_k = \{(i,j) \in \mathbf{V}_k \times \mathbf{V}_k : \bigcap_k \mathbf{E}_k = \emptyset \wedge \mathbf{E} = \bigcup_k \mathbf{E}_k\}$. Notice that $|\mathbf{V}_k| = \ell_k$ and $\ell = \sum_k \ell_k$. A *spanning-tree* $\mathbf{T}_i$ is a subset of $\mathbf{E}_k$ that spans all vertexes in $\mathbf{V}_k$ with $\ell_k - 1$ edges. The set of all spanning trees for $\mathbf{E}_k$ is denoted $\mathcal{T}_k$. A *minimum-spanning-tree* is defined as Eq.(1.5).

$$\mathsf{MST}(\mathbf{E}_k) = \underset{\mathbf{T} \in \mathcal{T}_k}{\arg\min} \left\{ \sum_{(i,j) \in \mathbf{T}} w(i,j) \right\} \tag{1.5}$$

Now, a *minimum-spanning-forest* MSF is the sub-graph of G consisting of the union of the *minimum-spanning-trees* of each connected component $\mathbf{E}_k$, that is:

$$\mathsf{MSF}(\mathbf{E}) = \bigcup_k \mathsf{MST}(\mathbf{E}_k)$$

As a result, the edges $\mathbf{E}_{\mathsf{MSF}} \subseteq \mathbf{E}$. The roots of the trees in MSF are denoted $\mathbf{R}_{\mathsf{MSF}} = \{r_k \in \mathbf{V}_k : 1 \le k \le \ell\}$.

The Bivariate marginal estimation of distribution algorithm (`BMDA`) is a *discrete-bivariate* `EDA` [36] that uses a graph G where vertexes are associated with the random variables X_i from the problem domain, and pair-wise dependencies are represented with a MSF (see Figure 1.2).

[2]The actual size of the population is achieved by generating $n/2$ tournaments of $(\mathbf{x_1}, \mathbf{x_2})$ candidates.

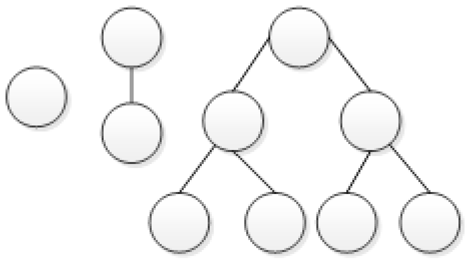


Figure 1.2: This is an example of a MSF structure used in `BMDA`. Note that in this example $k = 3$ and $\{\ell_1 = 1, \ell_2 = 2, \ell_3 = 7\}$

For this purpose, `BMDA` assumes a bivariate binomial probability model as defined in Eq.(1.6) with parameters $\boldsymbol{\theta} = \{\mathsf{MSF}, \boldsymbol{\rho}_M, \boldsymbol{\rho}_C\}$.

$$
\begin{aligned}
P(X;\boldsymbol{\theta}) &= \prod_{i \in \mathbf{R}_{\mathsf{MSF}}} P(X_i;\rho_i) \prod_{(i,j)\in \mathbf{E}_{\mathsf{MSF}}} P(X_i|X_j;\rho_j,\rho_{ij}) \qquad (1.6)\\
P(X_i) &= \frac{\rho_i}{n}\\
P(X_i\ X_j) &= \frac{\rho_{ij}}{\rho_j}
\end{aligned}
$$

In `BMDA` the population update (Step(3)) is modified as to preserve best candidates $\mathcal{C}$ from the previous iteration (see Eq.(1.7)):

$$\mathcal{D} = \mathsf{sample}(P(X;\theta), \tfrac{n}{2}) \cup \mathcal{C} \qquad (1.7)$$

Now, at every iteration t of the main loop in Algorithm 1, estimation of model parameters $\boldsymbol{\theta}$ in `BMDA` (Step(5)) comprises the following sub-steps :

1. Build a *disconnected* graph $\mathsf{G}(\mathbf{V}, \mathbf{E}_t)$ with a bivariate *Pearson* χ^2 dependency test criterion using Eq.(1.8). Notice that the statistic χ^2 is estimated from the current candidate pool $\mathcal{C}$ at iteration t.

$$\mathbf{E}_t = \{(i,j) \in \mathbf{V} \times \mathbf{V} : i \neq j \wedge \chi^2_{ij} \geq 3.84\} \qquad (1.8)$$

2. Compute $\mathsf{MSF}(\mathbf{E}_t)$ representing variable dependencies.

3. Estimate model parameters $\boldsymbol{\rho}_M = \{\rho_i : i \in \mathbf{R}_{\mathsf{MSF}}\}$ and $\boldsymbol{\rho}_C = \{\rho_{(i,j)} : (i,j) \in \mathbf{E}_{\mathsf{MSF}}\}$ using frequentist updates, again over the current candidate pool $\mathcal{C}$ at

iteration t, using Eqs.(1.9a, 1.9b).

$$\rho_i^a = \sum_{k=1}^{n} [\mathcal{C}_{ki} = a] \tag{1.9a}$$

$$\rho_{ij}^{ab} = \sum_{k=1}^{n} [\mathcal{C}_{ki} = a \wedge \mathcal{C}_{kj} = b] \tag{1.9b}$$

$$\text{where } [a] = \begin{cases} 1 & a \text{ is true} \\ 0 & \text{otherwise} \end{cases}$$

1.4.2 Feature selection with EDA

Feature Subset Selection (`FSS`) techniques are helpful in improving prediction models for supervised learning, detecting dimensions of tight data conglomeration in clustering tasks and a deeper understanding of process for data generation. Nonetheless, the design of novel `FSS` techniques aims to provide more useful information also incurs in new levels of complexity giving rise to new challenges pertaining to feasible and efficient computational techniques.

`FSS` can be seen as a optimization problem where the search space is represented as all possible combination of feature subsets. `FSS` with `EDA`s is a novel multi-objective approach able to minimize the amount of features and to maximize the classification accuracy with a learner algorithm. Resulting feature subset is therefore able to induce a learning model with good generalization capabilities. Significant `FSS-EDA`'s illustrations are the *wrapper* `FSS-EBNA` [32] and the *embedded* `wKiera` [42].

On one hand, `FSS-EBNA` uses a multivariate estimation of distribution algorithm (`EBNA`) as the search engine for exploring the search space and a Naive-Bayes classifier (NBC) as the learner algorithm, due to its simplicity and performance when dealing with high-dimensional spaces [32]. Figure 1.3 illustrates the main components for this algorithm. `FSS-EBNA` has been able to find suitable feature subsets with good generalization capabilities in problems where high-order dependencies exists [32].

On the other hand, `wKiera` uses an univariate `EDA` for exploring the search space. `wKiera` iteratively estimates a probability distribution with fittest candidates. Each candidate represents features relevance as a weighted vector $\mathbf{w}$ of a *wighted kernel learning machine*. Figure 1.4 illustrates the main components of this algorithm. `wKiera` and `FSS-EBNA` differs mainly in that `wKiera` is an *embedded* approach able to find a suitable features estimation with the robustness of

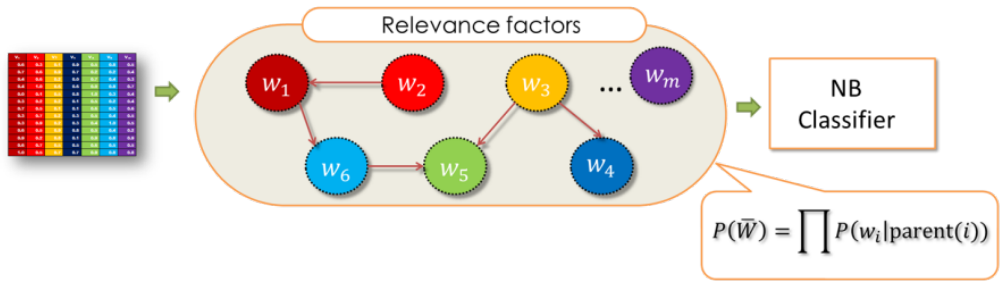


Figure 1.3: Main components of the FSS-EBNA algorithm.

its learning machine but it assumes variables are independent, and FSS-EBNA is a *wrapper* approach able to deal with high-order dependency problems efficiently but it uses a simple classifier with strong independence assumptions.

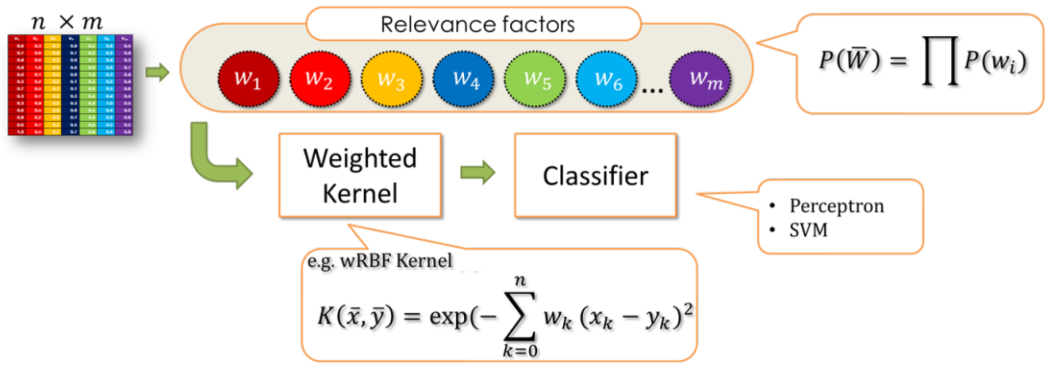


Figure 1.4: Main components of the wKiera algorithm.

1.4.3 Kernel classifiers

Supervised learning aims to find optimal learning algorithms with high generalization and prediction performance to produce classification models of the training set of examples and their associated labels. The prediction accuracy of the resulting classifier is then evaluated with a test set. Among these techniques, linear classifiers are widely used because of their theoretically simplicity and computational efficiency. Application of linear classifiers, in real world problems where nonlinearities arise, has been possible due to the advances in kernel machines [50], recently emerged as powerful high-accuracy classifiers with robust generalization abilities [50]. They use linear functions to perform classification, allowing for non-linear discriminatory borderlines in the input space by means of a kernel function. This function is a mapping of similarity measures in a transformed space where nonlinearities can be solved linearly. Two widely used kernel functions are the RBF kernel and the polynomial kernel (Eq.(1.10) and Eq.(1.11) respectively):

$$K_\sigma(\mathbf{x},\mathbf{z}) = \exp\left(-\sigma \sum_{i=1}^{\ell}(x_i - z_i)^2\right) \tag{1.10}$$

and

$$K_d(\mathbf{x},\mathbf{z}) = \langle \mathbf{x},\mathbf{z}\rangle^d, \tag{1.11}$$

where $\mathbf{x},\mathbf{z} \in X$, $X \subseteq \mathbb{R}^\ell$ represents the input space and ℓ the number of variables or dimensions. The parameter σ and d define the width of the RBF and polynomial kernel respectively.

Modified weighted versions of these kernels introduce scale factors $\mathbf{w} = \{w_1 ... w_\ell\}$ to weight the amount that each dimension contributes to the total computation [10]. The weighted RBF and weighted polynomial kernels are then defined as Eq.(1.12) and (1.13):

$$K_{\sigma,\mathbf{w}}(\mathbf{x},\mathbf{z}) = \exp\left(-\sigma \sum_{i=1}^{\ell} w_i(x_i - z_i)^2\right) \tag{1.12}$$

and

$$K_{d,\mathbf{w}}(\mathbf{x},\mathbf{z}) = \left(\sum_{i=1}^{\ell} w_i(x_i \cdot z_i)\right)^d. \tag{1.13}$$

Some FSS methods have been proposed in connection with kernel functions as `wKiera`. The idea in this case was to consider the weight vector $\mathbf{w}$ of the kernel as a relevancy estimator of input variables. The accuracy of the kernel classifier coupled with the weight vector candidates is taken as the fitness measure of the `UMDA`.

1.4.4 Component-based software development

Component-based software development has been well received in the software engineering community during the last years [1]. This approach provides substantial benefits like: Reducing software development times, re-usability and robustness because a component can be tested and reused in different contexts.

A software component

This paradigm is oriented to produce components with standard and well-defined interfaces that can be combined to create new software solutions in a faster way. The *component* term has received may definitions has been but one of the most welcome has been given by Szyperski as follows:

"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition [52]".

The extensibility and re-usability of a component allows to create a global market of components where they can be connected and combined. Therefore a component can be adapted and customized to a particular need similar to the mechanical engineering field, as depicted in Figure 1.5

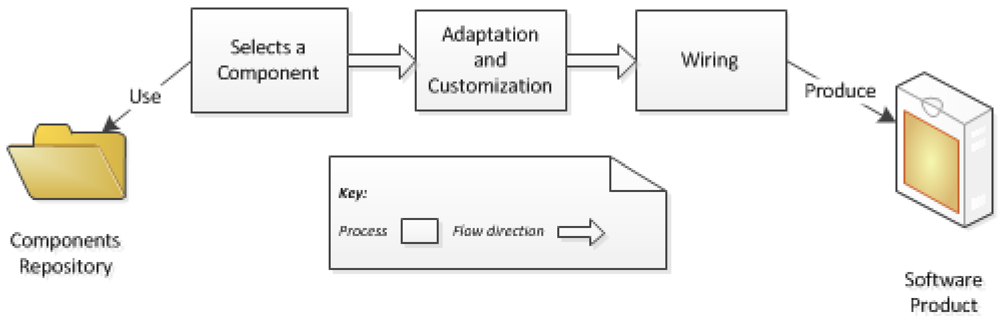


Figure 1.5: Component-based software development [53].

Modeling software components

Modeling components is one of the biggest challenges because there is not a uniform standardized way to do it. Different approaches have been proposed but the *Unified Modeling Language* (UML) is a well-known representative both in industry and academy [52] [8] [2]. This language defines the following elements: components, ports, required interface and provider interface, as depicted in Figure 1.6.

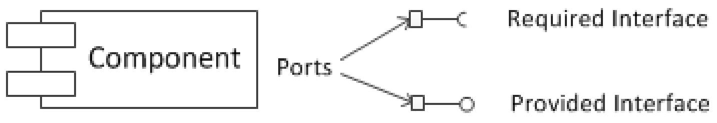


Figure 1.6: UML notation for components modeling.

A port is responsible for listening or sending the information to one or many components. In order to make the port available it is required to define either a required or provided interface. Those interfaces defines how a component can be used [8]. *UML-2* allows to model both internal and external views. External views can be seen as black-boxes, it encapsulates the component behavior and only exposes required and provided interfaces [53]. The internal view is termed white-box, and shows the internal structure and behavior of the component.

Software design patterns

Experts does not create new solutions whenever a problem occurs. They usually remember previous solutions applied to similar problems and they reuse its essence. This recurrent solutions are known as a pattern. Software design patters are therefore recurrent solutions for recurrent design problems. The following set of design patterns were relevant for the development of this thesis: Layering, template method, responsibility splitting pattern and dynamic binding (Plug-ins).

> *Layering* is a architecture pattern [9] focuses on the grouping of functional similarities distributed into distinct layers, each one with a strong separation of concern and stacked vertically on top of each other. The main aim of this patter is to supports flexibility and maintainability.
>
> *Template Method* [17] is a design pattern that defines a skeleton for an algorithm. Different behaviors can be obtained by sub-classing the abstract algorithm skeleton. The main advantage of this patter is the clear separation of the algorithm behavior and its execution flow.
>
> *Responsibility splitting* [3] is a pattern for reducing the cost of source-code modifications by dividing a general software responsibility into small cohesive pieces of software. This patter requires a deep understanding of the variation points in order to split the source-code according to system needs and to avoid over-engineering. If a new source-code modification is needed the impact will be address to only a small subset of code.
>
> *Dynamic binding or Plug-in* [3] is a pattern to introduce software modifications at runtime without recompile or redeploy the system. Usually those plugins are discovered automatically using a plug-in folder or through a configuration artifact.

1.4.5 Orange: A visual component-based platform for data mining

`Orange` is an open-source multiplatform component-based software framework, featuring visual and scripting interfaces for many machine learning algorithms.

Orange is one of such open-source visual frameworks, originally proposed for functional genomic analysis [13].

Orange has been progressively enriched with additional visual software components (*widgets*) for several machine learning tasks (see Figure 1.7). Widgets provide a set of input/output interfaces that encapsulate related services comparable to what required/provided interfaces are for a component. Orange takes advantage of three interesting features: (i) its versatile visual front-end; (ii) the powerful reuse and glue postulates of component-based software development in which it is based; and (iii) its conformity to the *open/closed* principle of object-oriented programming through an scripting interface to Python.

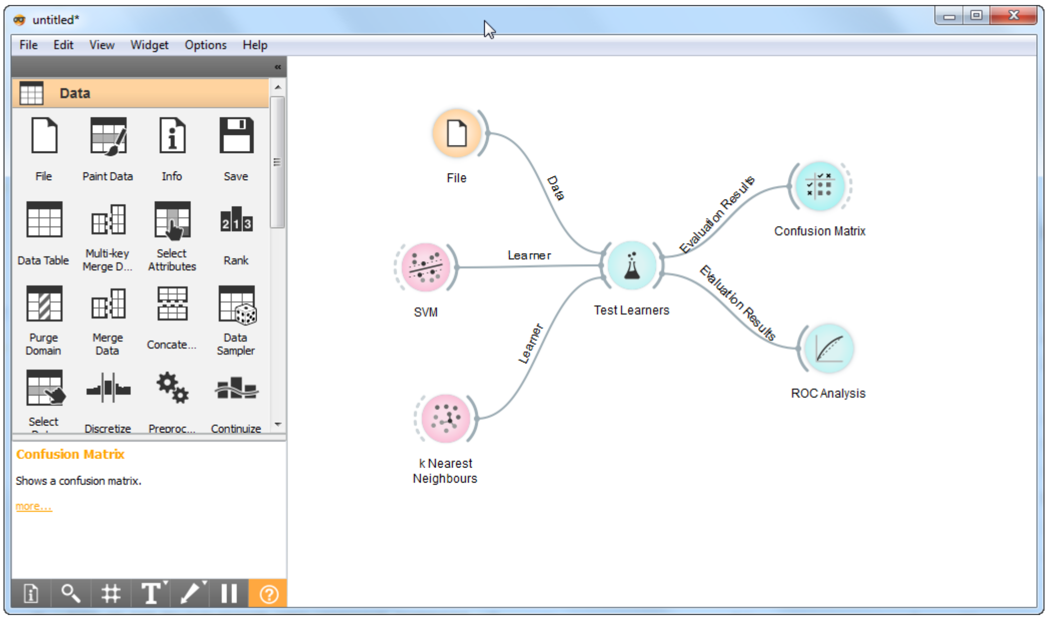


Figure 1.7: Orange's visual programming board, also known as *canvas*.

Orange relies on the following open-source initiatives: Python as the scripting programming language, PyQt as a port of the multiplatform user-interface framework Qt and numpy a scientific computing tool. Orange is able to runs on Windows and Mac OS X platforms and can be freely downloaded from http://orange.biolab.si/.

Chapter 2

A memory efficient compact EDA for continuous domains

2.1 Motivation

Estimation of Distribution Algorithms (`EDA`) [6,38] approximately optimizes a cost function by building a probabilistic model of a pool of promising sub-optimal solutions over a given search space. For very-high dimensional search spaces, storing and updating a large population of candidates may imply a computational burden in both time and memory. The *compact* approach circumvents storage limitations by incrementally updating the probability model using just two candidates at any step of the algorithm, instead of the entire population. This feature makes the compact `EDA` framework practical for large-scale optimization, a soon-to-be commonplace setting in scientific domains such as bioinformatics, particle physics, chemical crystallography, or social network analysis, to name a few.

`TILDA` is a new efficient *univariate-continuous-compact* `EDA` aimed at optimization of large-scale problems of up to 10^7 binary decision variables, nevertheless requiring low-cost computational resources. The method is tailored to optimization of additively-decomposable fitness functions in terms of the contribution of individual variables, that is, to problems where incremental fitness evaluation is feasible. Although no higher-order dependencies within the variables are considered, it might be relevant as an early-stage data analysis tool in high dimensional spaces, for example to carry out feature selection for subsequent dependencies estimation on smaller variable subsets, as other models with independence assumptions have shown [42].

2.2 TILDA: Compact updates for a continuous-valued EDA

TILDA uses a multivariate marginal joint Gaussian distribution (see Eq.(2.1)) whose parameters are defined by $(\mu_i, \sigma_i)_{i=1}^{\ell}$ for both the mean and standard deviation of each input variable following Eq.(2.1).

$$P(X; \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \prod_i \mathcal{N}(\mu_i, \sigma_i) \tag{2.1}$$

TILDA samples two candidates $(\mathbf{x_1}, \mathbf{x_2}) \sim P(X; \boldsymbol{\mu}, \boldsymbol{\Sigma})$ similar to cGA by mean of a 2-way tournaments. The winner $\mathbf{x}^{\top}$ is added into W to update the probability model parameters.

It is known that the maximum likelihood mean estimate of the new population, that is $\boldsymbol{\mu} = (\mu_1, \ldots, \mu_\ell)$, can be computed in vectorized form [43]:

$$\boldsymbol{\mu} = \frac{1}{n} \sum_{\mathbf{x}_k \in W} \mathbf{x}_k \tag{2.2}$$

hence with a fraction of each tournament's winner the probability model parameter $\mu_{\mathbf{i}}$ can be updated incrementally, as the formula in Equation (2.3) indicates.

$$\mu_i := \mu_i + \tfrac{1}{n} x_i^{k_t}; \quad t = 1, \ldots, n. \tag{2.3}$$

Now, the same rationale can be applied to the variance estimate $\boldsymbol{\Sigma} = (\sigma_1^2, \ldots, \sigma_\ell^2)$ in a vectorized form as:

$$\sigma_i^2 = \frac{1}{n} \sum_{t=1}^{n} (x_i^{k_t})^2 - \mu_i^2.$$

Then again, an incremental update can be obtained by first setting a temporal variable $\tilde{\sigma}_i^2 = 0$, then accumulating deviations of each tournament's winner in this variable using Equation (2.4), and finally subtracting the square of the previously computed mean estimate to obtain the variance estimate, as in Equation (2.5).

$$\tilde{\sigma}_i^2 := \tilde{\sigma}_i^2 + \tfrac{1}{n}(x_i^{k_t})^2; \quad t = 1, \ldots, n. \tag{2.4}$$

$$\sigma_i^2 = \tilde{\sigma}_i^2 - \mu_i^2 \tag{2.5}$$

Those compact updating rules are used as the estimation Step (5) of Algorithm1. TILDA approximately finds a solution to the unconstrained optimization problem:

$$\boldsymbol{\beta}^{\star} = \underset{\mathbf{x} \in \mathbb{R}^{\ell}}{\mathsf{argmax}}\, f(\mathbf{x}),$$

where $f(\cdot)$ is the fitness or cost function to be optimized.

Algorithm 2 `TILDA` pseudo-code

```
Requires: $\ell > 0$, fitness function $f(\cdot)$
Outputs: best candidate $\boldsymbol{\beta}$
 1: $\boldsymbol{\mu} = \mathbf{0}, \boldsymbol{\Sigma} = \mathbf{1}, \boldsymbol{\beta} = \mathbf{0}, \gamma = 0.5, \epsilon = 0.01$
 2: while ending citeria not met
 3:     $\tilde{\boldsymbol{\mu}} = \tilde{\boldsymbol{\Sigma}} = \mathbf{0}$
 4:     repeat $n$ times
 5:         $\{\mathbf{x}', \mathbf{x}''\} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$
 6:         $\hat{\mathbf{x}} = \mathsf{argmax}(f(\mathbf{x}'), f(\mathbf{x}''))$
 7:         $\boldsymbol{\beta} = \mathsf{argmax}(f(\boldsymbol{\beta}), f(\hat{\mathbf{x}}))$
 8:         $\tilde{\boldsymbol{\mu}} = \tilde{\boldsymbol{\mu}} + \frac{1}{n}(\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_\ell)$
 9:         $\tilde{\boldsymbol{\Sigma}} = \tilde{\boldsymbol{\Sigma}} + \frac{1}{n}(\hat{x}_1^2, \hat{x}_2^2, \ldots, \hat{x}_\ell^2)$
10:     end repeat
11:     $\boldsymbol{\mu} = \boldsymbol{\mu} - \gamma(\boldsymbol{\mu} - \frac{1}{2}(\boldsymbol{\beta} + \tilde{\boldsymbol{\mu}}))$
12:     $\boldsymbol{\Sigma} = \boldsymbol{\Sigma} - \gamma(\boldsymbol{\Sigma} - (\tilde{\boldsymbol{\Sigma}} - (\tilde{\mu}_0^2, \tilde{\mu}_1^2, \ldots, \tilde{\mu}_\ell^2)) + \epsilon$
13: end while
```

For illustration purposes the new resulting algorithm is depicted as Algorithm 2. The algorithm models a population of promising solutions using a multivariate marginal joint Gaussian distribution with parameters $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$. The main loop (line 2) runs for a number of generations or until the parameters converge. The inner loop (line 4) simulates the breeding of one population by sampling two candidates at a time (line 5), carrying out 2-way tournaments to determine the winner $\hat{\mathbf{x}}$ (line 6), and also keeping temporary parameter estimates for the current population (lines 8-9) using Equations (2.3-2.4). Notice that the algorithm stores the best solution found up to a certain generation in the variable $\boldsymbol{\beta}$ (line 7). The actual updates of the overall estimates are performed at the end of the main loop (lines 11-12). The overall mean estimate is computed with a memory-preserving rule with decay factor γ, incorporating an elitist mechanism that shifts the estimate towards the average of the population mean and the best candidate. Similarly, the overall variance estimate uses the population mean, as stated in Equation (2.5), plus a small constant intended to prevent premature convergence. When the ending criteria is met, the approximate optimal solution found by the algorithm can be retrieved from $\boldsymbol{\beta}$.

2.3 Further enhancements for large-scale problems

2.3.1 Sliding-pin arithmetic coding

In order to represent high dimensional search spaces for large-scale problems we revised representation in *arithmetic coding.* Given the alphabet $S = \{0, 1\}$ and a probability distribution $\theta = \{P(0) = p, P(1) = 1 - p\}$, the idea of arithmetic coding is to represent an arbitrary binary string of length ℓ using distribution θ and a single real number $q \in [0, 1]$ known as a *code value* (see details in [47]). Decoding is achieved by performing a Bernoulli trial for each bit in the sequence, that is, flipping an unfair coin with bias p, where q represents the chance of a face-up outcome. For this purpose, the decoder builds nested intervals and biases $\{[a_i, b_i], p_i\}_{i=1}^{\ell}$ that re-define the support and probability distributions throughout consecutive bits along the sequence. Thus, a decoded string $\mathbf{s}_p^q$ corresponds to $\mathbf{s}_p^q = (s_i^q)_{i=1}^{\ell} = (s_1^q, \ldots, s_\ell^q)$, where the outcome for each bit i is determined by the indicator function:

$$s_i^q = \begin{cases} 0 & q \leq p_i \\ 1 & q > p_i \end{cases}, \; i \geq 1, \tag{2.6}$$

and the bias p_i is updated iteratively according to the outcome of the preceding bit and the bounds a_i, b_i of the nested support intervals, following the recursions:

$$\begin{aligned} [a_1, b_1] &= [0, 1]\,, \\ [a_i, b_i] &= \begin{cases} [a_{i-1}, p_{i-1}] & q \leq p_{i-1} \\ [p_{i-1}, b_{i-1}] & q > p_{i-1} \end{cases}, \; i > 1, \\ p_i &= a_i + (b_i - a_i)p\,, \; i \geq 1. \end{aligned} \tag{2.7}$$

Equation (2.7) implies that the support interval for a subsequent trial is replaced by the subinterval chosen from the comparison of the code value and the bias of the current trial. As a result, the length of nested intervals shrinks away as the decoding progresses, with the values of these variables growing in precision digits. In theory, one may obtain an arbitrary number of bits with this decoding scheme. In practice, however, the number of bits is limited by machine-dependent floating point arithmetic, in other words, due to round-off errors the supporting interval (and the bias p_i too) eventually collapses and yields the code value q, resulting in a constant zero-output from that trial onwards.

In an attempt to circumvent this shrinking-interval limitation, we contemplated an alternative decoding scheme, that we called *sliding-pin* as explained next. Firstly, observe that during the shrinking procedure the nested probability distributions obtained with the successive updates of $[a_i, b_i]$ and p_i, preserve the proportions of the original distribution defined by p in θ. In fact, these are just shrunk replica distributions repeatedly shifted to different locations along the unit interval. Within these distributions, the *global* (fixed) code value q defines a *local* chance of outcome that is relative to the location of the supporting interval and bias of the respective trial. Bearing in mind this observation, we sought to switch roles and keep distribution parameter θ *global*, that is, fixed to the unit interval, whereas the otherwise constant code value is repeatedly shifted to mirror the *local* chance of outcome at each trial of the shrinking procedure. In this way, for all i we fixed $[a_i = 0, b_i = 1]$ and $p_i = p$ and rescaled the value of the now variable q_i using the following recursive rules:

$$\begin{aligned} q_1 &= q\,, \\ q_i &= \begin{cases} \dfrac{q_{i-1}}{p} & q_{i-1} \leq p \\ \dfrac{(q_{i-1} - p)}{(1-p)} & q_{i-1} > p \end{cases}, \; i > 1 \end{aligned} \tag{2.8}$$

In this new scheme, the values of q_i shift around the unit interval resembling the cursor of an old-fashion sliding rule, thus the name *sliding-pin* arithmetic decoding. Since the length and bias of the supporting interval are in a normal scale, the relocated values of q_i are not machine-precision dependent. Additionally, it is worth noting that the sliding-pin procedure only maintains one parameter (q_i) as opposed to three parameters (a_i, b_i, p_i) for the shrinking decoding. The decoded string $\mathbf{s}_p^q$ would be now obtained as:

$$\mathbf{s}_p^q = (s_i^p)_{i=1}^{\ell} = (s_1^p, \ldots, s_\ell^p), \tag{2.9}$$

where the indicator function becomes:

$$s_i^p = \begin{cases} 0 & q_i \leq p \\ 1 & q_i > p \end{cases}, \; i \geq 1. \tag{2.10}$$

The two decompression schemes are illustrated in Figure 2.1 and the sliding-pin algorithm decompression process is depicted in Table 2.3.1

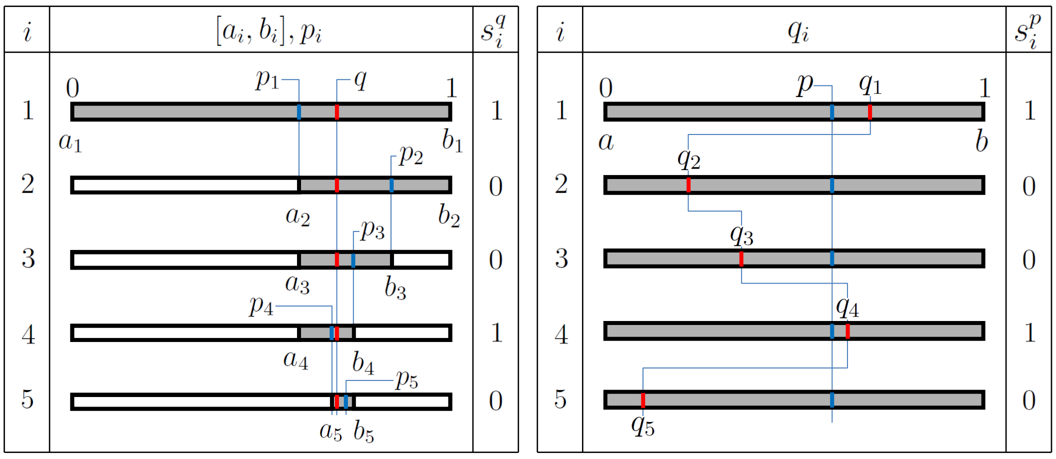


Figure 2.1: Illustration of the operation of two arithmetic-decoding techniques for a string of length $\ell = 5$. The leftmost column i of each illustration indicates the decoding step. The corresponding output bit for each step is depicted in the rightmost column. (a) *Shrinking-interval* decoding [47]: The values of $\{[a_i, b_i], p_i\}_{i=1}^5$ are updated recursively using Equations (2.7) for a given fixed q whilst the output bit s_i^q is obtained with Equation (2.6); the shaded area within the nested subintervals shrinks away as the decoding progresses. (b) *Sliding-pin* decoding: The support interval and p are now fixed, while $\{q_i\}_{i=1}^5$ is subsequently shifted throughout the unit interval using Equation (2.8), thus resembling the strip of a sliding-rule; the output bit s_i^p in this case is given by Equation (2.10).

```
Requires: $(p, q), \ell$
Outputs: $\mathbf{b} \in [\mathbf{0}, \mathbf{1}]^\ell$
  repeat $\ell$ times
    $b_i = q > p$
    if $b_i$ then
        $q = (q - p)/(1 - p)$
    else $q = q/p$
  end repeat
```

Table 2.1: Sliding-ping decoding algorithm.

2.3.2 Stream algorithms for decomposable cost functions

Now that we have a representation schema for high dimensionality search spaces we conceive cost functions as stream algorithms with *sliding-pin* decompression.

The cost function should be additively-decomposable, that is to be calculated as a composition of its component functions regardless its order. Function $f(\mathbf{s})$ is *decomposable* if it can be represented as $f(\mathbf{s}) = f(s_0) \oplus f(s_1) \oplus ... \oplus f(s_n)$ where $\oplus$ is an operator with commutative properties for a candidate solution ($\mathbf{s} \in [0,1]^\ell$). A function $f(\mathbf{s})$ is *additively-decomposable* when $\oplus$ is the *additive* $+$ operator .

Example of additively-decomposable problems `OneMax` [37] and `RoyalRoad` [27] (including noisy variations) are depicted in Table 2.2. For the case of `RoyalRoad` functions, the parameter c represents a score added to a candidate for each succeeding schema of order c that can be found within its bit string, starting at bit 1 (in contrast to the original definition given in [27], here only a single block scan of ℓ/c schemas is carried out to compute fitness; no additional hierarchical scans of higher -doubled- order schemas follow up).

Problem	**Fitness function**
`OneMax`	$f(\mathbf{s}) = \sum_{i=1}^{\ell} s_i$
`Noisy OneMax`	$\hat{f}(\mathbf{s}) = f(\mathbf{s}) + \mathcal{N}(0, \sigma_N^2)$
`RoyalRoad`	$f_c(\mathbf{s}) = \sum_{i=0}^{(\ell/c)-1} c \left(\prod_{k=ic+1}^{(i+1)c} s_k \right)$
`Noisy RoyalRoad`	$\hat{f}_c(\mathbf{s}) = f_c(\mathbf{s}) + \mathcal{N}(0, \sigma_N^2)$

Table 2.2: Fitness functions used in stream algorithms with sliding-pin decompression for a candidate $\mathbf{s} = \{s_1, \ldots, s_\ell\}$, $s_i \in \{0, 1\}$. $\mathcal{N}(0, \sigma_N^2)$ denotes a normally distributed random variable of variance σ_N^2.

Let us observe that the evaluation of these fitness functions allows additively-decomposable computations. Hence stream decompression/fitness-evaluation versions are shown in algorithms (see Table 2.3) in order to take advantage of its memory savings[1]. Instead of $\mathbf{s}$, these stream algorithms take an arithmetic code (p, q) as their input, which implicitly defines the bit string $\mathbf{s}_p^q$ of Equation (2.9), and simultaneously decompress the candidate and compute its fitness value f.

[1]We remark that the `TILDA` is suitable as well for non-decomposable fitness functions, or problems where no stream decompressing algorithm is available. In the worst of these cases its memory complexity, as mentioned earlier, would be in $O(\ell)$, still a favorable scenario for large-scale applications.

```
Requires: $(p, q), \ell, noise$
Outputs: $f$
  $f = 0$
  repeat $\ell$ times
    $bit = q > p$
    if $bit$ then
        $q = (q - p)/(1 - p)$
        $f = f + 1$
    else $q = q/p$
  end repeat
  if $noise$ then $f = f + \mathcal{N}(0, \sigma_N^2)$
```

(a) `OneMax`

```
Requires: $(p, q), \ell, c, noise$
Outputs: $f$
  $f = 0; k = 1; block =$ true
  repeat $\ell$ times
    $bit = q > p$;  $block = block \wedge bit$
    if $bit$ then $q = (q - p)/(1 - p)$
    else $q = q/p$
    if $k = c$ then
        $k = 0$
        if $block$ then $f = f + c$
        else $block =$ true
    end-if
    $k = k + 1$
  end repeat
  if $noise$ then $f = f + \mathcal{N}(0, \sigma_N^2)$
```

(b) `RoyalRoad`

Table 2.3: Stream algorithms for string decompression and fitness evaluation. (a) `OneMax`: this algorithm is a variation of the sliding-pin arithmetic decoder of Equations (2.8-2.10), where the fitness f is an additively-decomposable function calculated incrementally every time the current *bit* is set to **true**; inputs are: arithmetic code (p, q), problem size ℓ, and a *noise* flag to activate the noisy version. (b) `RoyalRoad`: based on the `OneMax` code above, and using c as an additional input defining the order of the schema, this algorithm keeps a *block* flag that is successively updated with the conjunction of the decoded *bit* variable; for every succeeding schema, whose extent is controlled with counter k, score c is added to f if the schema *block* flag is set, otherwise this flag is readjusted.

2.3.3 Truncated-normal random numbers

A final step for handling large-scale problems is to sample candidates in the arithmetic-coding search space to take advantage of stream algorithms for evaluating additively-decomposable cost functions. `TILDA` is designed to work in continuous domains where variables are assumed to be Gaussian-distributed over the real line. In an arithmetic coding scheme, however, the search of a feasible set of values for a pair (p, q) requires the evolutionary algorithm to generate random numbers bounded within the $[0, 1]$ interval[2]. For this reason we adapted the algorithm to a *truncated* Gaussian model with support on the unit interval only, following [14, page 39]: given a random variable X with cumulative probability distribution Φ, the truncated random variable X^{T} with distribution Φ^{T} and sup-

[2]Let us mention that mechanisms for truncated pseudorandom number generation in other support intervals has been also highlighted in previous `EDA` studies (e.g. [20, 26])

port $[a, b]$ is defined as

$$\Phi^{\mathsf{T}}(x) = \begin{cases} 0 & x < a \\ \frac{\Phi(x)-\Phi(a)}{\Phi(b)-\Phi(a)} & a \leq x \leq b \\ 1 & x > b \end{cases} . \tag{2.11}$$

We define Φ as the cumulative distribution of $\mathcal{N}(\mu, \sigma), a = 0, b = 1$ and normalise $x' = \frac{x-\mu}{\sigma}$, $a' = \frac{a-\mu}{\sigma} = \frac{-\mu}{\sigma}$ and $b' = \frac{b-\mu}{\sigma} = \frac{1-\mu}{\sigma}$. Let $u \sim [0, 1]$ be a uniformly distributed random number; hence we use the inversion method for random number generation (see e.g. [14]), that is, assume $u = \Phi^{\mathsf{T}}(x')$ and work out the resulting x bounded to [0,1]. Since $a' \leq x' \leq b'$ from Equation (2.11) we get:

$$\Phi(x') = u(\Phi(b') - \Phi(a')) + \Phi(a'),$$

which in turn implies that,

$$x = \sigma(\Phi^{-1}(u(\Phi(b') - \Phi(a')) + \Phi(a')) + \mu. \tag{2.12}$$

Since $\Phi(a')$ and $\Phi(b')$ are constant terms, Equation (2.12) can be used to generate a truncated random number by computing a uniform random number and one evaluation of the inverse normal distribution with given parameters (μ, σ).

With the previous three enhancements for large-scale problems we have introduced a new version of `TILDA` called `@TILDA` with arithmetic coding representation of the search space, cost functions as stream algorithms with *sliding-pin* decompression and sampling truncated-normal random numbers.

2.4 @TILDA: Arithmetic coding TILDA

To begin with, let us recall the arithmetic-coded `EDA` proposed in [51]. It is a continuous domain population-based `EDA` that takes advantage of the shrinking-interval arithmetic decoding, defined in Equations (2.6-2.7), in order to search for a solution string $\mathbf{s}_p^q$ in a high dimensional space of size ℓ. The compression scheme reduces the memory requirements needed to store and maintain the population of candidates, enabling the algorithm to solve large-scale problems in a standard desktop machine. Here we present an algorithm that further improves memory usage by means of a compact representation.

The new algorithm is an adaptation of `TILDA` (as described in Section 2). The idea is to search for the best $(p, q) \in \mathbb{R}^2$ that determines the string $\mathbf{s}_p^q$ whose fitness function is evaluated in $\{0, 1\}^\ell$ with functions `RoyalRoad` and `OneMax`. For this purpose, we let $\boldsymbol{\mu}, \tilde{\boldsymbol{\mu}}, \boldsymbol{\Sigma}, \tilde{\boldsymbol{\Sigma}}, \boldsymbol{\beta} \in \mathbb{R}^2$ in Algorithm 2 and we chose the sliding-pin scheme of Equations (2.8-2.10) to decompress $\mathbf{s}_p^q$. Likewise, we decided on using the truncated Gaussian random generator of Equation (2.12) for sampling. The resulting method (`@TILDA`: *arithmetic-coding* `TILDA`) is shown in Algorithm 3.

The time complexity of the algorithm is dependent on the convergence speed, the population size n, and the cost needed to decompress and evaluate the candidate (usually linear in ℓ). More specifically, neglecting the cost of random number generation, each inner loop iteration incurs $O(\ell)$ for the two fitness evaluations in line 6 (the fitness evaluations in line 7 can be spared by caching previous calls). So, assuming t_c iterations are needed to converge, the running time is in $O(n\ell t_c)$.

Algorithm 3 `@TILDA`

```
Requires: $\ell > 0$, fitness function $f(\cdot)$
Outputs: best candidate $\boldsymbol{\beta}$
 1: $\boldsymbol{\beta} = \boldsymbol{\mu} = (0.5, 0.5), \boldsymbol{\Sigma} = (10, 10), \gamma = 0.9, \epsilon = \ell^{-1}$
 2: while convergence
 3:     $\tilde{\boldsymbol{\mu}} = \tilde{\boldsymbol{\Sigma}} = (0, 0)$
 4:     repeat $n$ times
 5:         $\{(p', q'), (p'', q'')\} \sim \mathcal{N}^{\mathsf{T}}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ //sampling (Eq.(2.12))
 6:         $(\hat{p}, \hat{q}) = \mathsf{argmax}(f(\mathbf{s}_{q'}^{p'}), f(\mathbf{s}_{q''}^{p''}))$ //decoding and evaluating (Eq.(2.8-2.10))
 7:         $\boldsymbol{\beta} = \mathsf{argmax}(f(\mathbf{s}_{\hat{q}}^{\hat{p}}), f(\mathbf{s}_{\beta_2}^{\beta_1}))$ // update best
 8:         $\tilde{\boldsymbol{\mu}} = \tilde{\boldsymbol{\mu}} + \frac{1}{n}(\hat{p}, \hat{q})$
 9:         $\tilde{\boldsymbol{\Sigma}} = \tilde{\boldsymbol{\Sigma}} + \frac{1}{n}(\hat{p}^2, \hat{q}^2)$
10:     end repeat
11:     $\boldsymbol{\mu} = \boldsymbol{\mu} - \gamma(\boldsymbol{\mu} - \frac{1}{2}(\boldsymbol{\beta} + \tilde{\boldsymbol{\mu}}))$
12:     $\boldsymbol{\Sigma} = \boldsymbol{\Sigma} - \gamma(\boldsymbol{\Sigma} - (\tilde{\boldsymbol{\Sigma}} - (\tilde{\mu}_0^2, \tilde{\mu}_1^2)) + \epsilon$
13: end while
```

Now we discuss an interesting memory saving mechanism obtainable with this new algorithm. Observe that its memory cost is dominated by the size of the problem ℓ. This is because the working memory consists of nine bi-dimensional variables in $O(1)$ plus the temporal buffer needed to decompress $\mathbf{s}_p^q$, which is in $O(\ell)$. However, if we constrain the application domain to additively-decomposable problems exhibiting modularity of fitness evaluation to the level of single variables, that is, problems where the fitness function aggregates independent contributions in each dimension of the solution, then the memory cost can be $O(1)$. The latter is possible by incrementally evaluating and disposing bits obtained during the stream decompression of $\mathbf{s}_p^q$. And given that the best solution can be retrieved at any point of the evolutionary loop from the arithmetic code $\boldsymbol{\beta} = (\beta_1, \beta_2)$, corresponding to $\mathbf{s}_{\beta_1}^{\beta_2}$, no string buffering is actually needed. As mentioned earlier, even if bits are not discarded (i.e. buffering them into a string) the fitness evaluation requires $O(\ell)$ time to scan the string, so the memory savings using streaming implies no additional time overload. It is in this sense that the algorithm is well suited for large scale problems, ensuring minimal memory consumption, regardless of the problem and population size. The memory efficiency of `@TILDA` compares favorably with arithmetic-coded `EDA` [51], which requires $O(\ell + n)$ for the evolutionary loop (although it could be lowered to $O(n)$ by using the same streaming technique), and also with `cGA` [22], which requires $O(\ell(\log_2 n + 2))$.

It may be noted in passing that, by using the sliding-pin arithmetic-coding technique and the truncated random number generator, `@TILDA` is able to always decompress feasible solutions notwithstanding the problem size ℓ. The latter contrasts with the difficulties found for the arithmetic-coded `EDA`, where some (p, q) codes where unable to produce the desired string length due to precision issues [51].

2.5 Experiments

We run experiments in a conventional laptop Intel® Core$^{\text{TM}}$ i7 machine with 6GB memory. Algorithm 3 and test-bed scripts were implemented in both Octave version 3.2.4 with QtOctave version 0.9.1 and Matlab R2011. Fitness routines in Table 2.3 were implemented in C with the MEX interface for faster evaluation.

Noise-free experiments

The first set of experiments were aimed at testing the behavior of the algorithm on *noise-free* (regular) problems. We set the number of maximum generations to 1000 for all experiments, with an early-stopping condition whenever a candidate decoding a number of bits equals to the problem size ℓ was found. In order to set the appropriate population size needed to find a solution with respect to ℓ, we carried out a grid-search [23] over $n \in \{2^3, 2^4, \dots, 2^{12}\}$; the search was stopped when, for a given n, at least 90% of 100 runs were successful. For the `RoyalRoad` case, the schema order c was set to 100 for all experiments. The values of learning rate and premature convergence parameters ($\gamma = 0.9$ and $\epsilon = \ell^{-1}$) were chosen from empirical evidence of preliminary experiments. Results of the final experiments are listed in Table 2.4.

Problem size	Solution	`OneMax`	`RoyalRoad`
$\ell = 10^3$	n	8	16
	Success rate	100/100	96/100
	Age (avg.)	29.62	58.22
$\ell = 10^4$	n	8	16
	Success rate	99/100	92/100
	Age (avg.)	77.21	42.33
$\ell = 10^5$	n	8	16
	Success rate	98/100	99/100
	Age (avg.)	146.58	93.24
$\ell = 10^6$	n	16	16
	Success rate	94/100	94/100
	Age (avg.)	229.95	207.27
$\ell = 10^7$	n	32	32
	Success rate	95/100	91/100
	Age (avg.)	214.88	235.59

Table 2.4: Noise-free experiments. For each problem (`OneMax`, `RoyalRoad`) and dimension ℓ, results are reported on n (population size required to solve it), success rate (successful runs over 100 repetitions) and average age of the solution (generation where optimal candidate emerged).

In summary, these results show that the algorithm succeeded in solving both problems from medium- to large-scale lengths (10^3 to 10^7 variables). Interestingly enough, it can be seen that a small population size $n \leq 32$ was needed to solve the problems, even for the large dimensionalities. The average generation of solution emergence was much smaller than the maximum setting of 1000. On the other hand, a comparable trend of performance is visible in the `OneMax` and `RoyalRoad` instances, except that a bigger population size was needed in the smaller problems for the latter. This finding may drop a hint on the potential effectiveness of the algorithm to tackle basic levels of structure in the problem (recall that `RoyalRoad`fitness function assigns score only to entire schemas of length c, that is, contiguous chunks of variables).

Noisy experiments

The second set of experiments focused on testing the robustness of the algorithm when *exogenous noise* was added to the original fitness function. We followed the same settings as before but with 30 runs. Again the population size was obtained by a grid-search. The level of noise was controlled with a exogenous-noise-variance to deterministic-fitness-variance ($\frac{\sigma_N^2}{\sigma_f^2}$) parameter as defined in [48]. We tested the algorithm using a corruption $\frac{\sigma_N^2}{\sigma_f^2}$ ratio of 10^{-2}. Results of this set of experiments are listed in Table 2.5.

In these experiments, the results on both problems are dissimilar. For the `Noisy OneMax` problem the incidence of noise in medium-size instances causes no difficulty to the algorithm in solving it. In larger instances, however, ($\ell = \{10^5, 10^6\}$) the algorithm struggles and demands higher population sizes of up to $n = 2048$ but still obtains success rates inferior to 50%. In contrast, a small population size $n = 32$ seems appropriate to solve all instances of `Noisy RoyalRoad` problems. We reason that in this case, the block-structure of the fitness score diminishes the effect external noise may have in misguiding the algorithm during its search.

Problem size	Solution	`OneMax`	`RoyalRoad`
$\ell = 10^3$	n	8	32
	Success rate	28/30	30/30
	Age (avg.)	59.93	32.20
$\ell = 10^4$	n	64	32
	Success rate	30/30	30/30
	Age (avg.)	48.17	48.47
$\ell = 10^5$	n	2048	32
	Success rate	19/30	27/30
	Age (avg.)	134.37	133.90
$\ell = 10^6$	n	2048	32
	Success rate	12/30	28/30
	Age (avg.)	168.87	153.80

Table 2.5: Noisy experiments. An exogenous noise rate of $\frac{\sigma_N^2}{\sigma_f^2} = 0.01$ was used in both test problems. For each each problem (`OneMax`, `RoyalRoad`) and dimension ℓ, results are reported on n (population size required to solve it), success rate (successful runs over 30 repetitions) and average age of the solution (generation where optimal candidate emerged).

2.6 Discussion

We introduced a new compact `EDA` algorithm that is able to solve to optimality additively-decomposable bit-string problems of up to 10^7 dimensions with minimum memory requirements, both at individual or block contribution of variables to the fitness score. In the presence of noise, the algorithm is suitable to solve medium to large-size instances of `Noisy RoyalRoad` problems but struggles with large scale `Noisy OneMax` problems. On this topic we are considering exploring different ideas for noise tolerance such as the effect of using a combination of best candidates obtained from independent shorter, quicker runs, and also analyzing in depth the role of elitism in such scenario.

In addition we are planning to study the role of the learning rate and premature convergence parameters within more elaborated mechanisms as those proposed in existing `EDA` literature. Moreover, it would be interesting to study the behavior of the algorithm in different kinds of fitness landscapes, for example, uneven (not all-ones) optimal solutions and also, deceptive functions.

As a concluding remark we anticipate that a parallel implementation of the algorithm would improve running times, although at present, an average run to solve a one-million bit problem in a regular laptop with a population size of 2048 is 1 hour.

Chapter 3

Goldenberry: An EDA and feature selection add-on for Orange

3.1 Motivation

`Goldenberry` is a supplementary evolutionary computation and feature selection add-on for `Orange`. An example of a `Goldenberry` program to optimize a cost function using three different `EDA` widgets is shown in Figure 3.1. In that program, widgets are the components or processing units needed to perform the optimization task. Rather than depicting a *dataflow* between these components, connections represent provision and consumption of services encapsulated as objects, which are associated with each input/output interface. For instance, in this program the `cGA`, `PBIL` and `UMDA` widgets require a function to be optimized that is provided by the `CostFunctionBuilder` widget through three instances of the `CostFunction` object, one per `EDA`, each in charge of keeping statistics of the number of function evaluations and running times. The `EDA` components are responsible of setting up the parameters of their corresponding algorithm and of providing a ready-to-use `Optimizer` that is in turn, consumed by the `BlackBoxTester`. The latter is in fact responsible of orchestrating the execution of these `Optimizer`s, that is, of running each `EDA` algorithm (whose parameters, including cost function, must have been already defined and provided in their output interfaces before execution begins) and also of collecting and visualizing the final results. Parameter settings and data presentation are embodied within the the graphical user interface of the widgets, as we shall depict next sections. Intuitively, this programming paradigm is very much the same as building a hardware apparatus: the user first gather, connect and set-up the required units before switching-on the resulting device.

`Goldenberry` is hosted publicly under a GPL license in Codeplex (`http://goldenberry.codeplex.com`). To install the software follow these steps:

1. Download and install latest Orange 2.7 from: `http://orange.biolab.si/download/`.

2. Get the latest `Goldenberry` release from the `Downloads` tab in `http://goldenberry.codeplex.com` (the download link is located to the left side of the screen).

3. Follow the installation instructions in the release notes just below the download link.

4. Open the `Orange` application and look for the "Optimization" and "Learner" toolbar in the canvas (as shown in Figure 3.1).

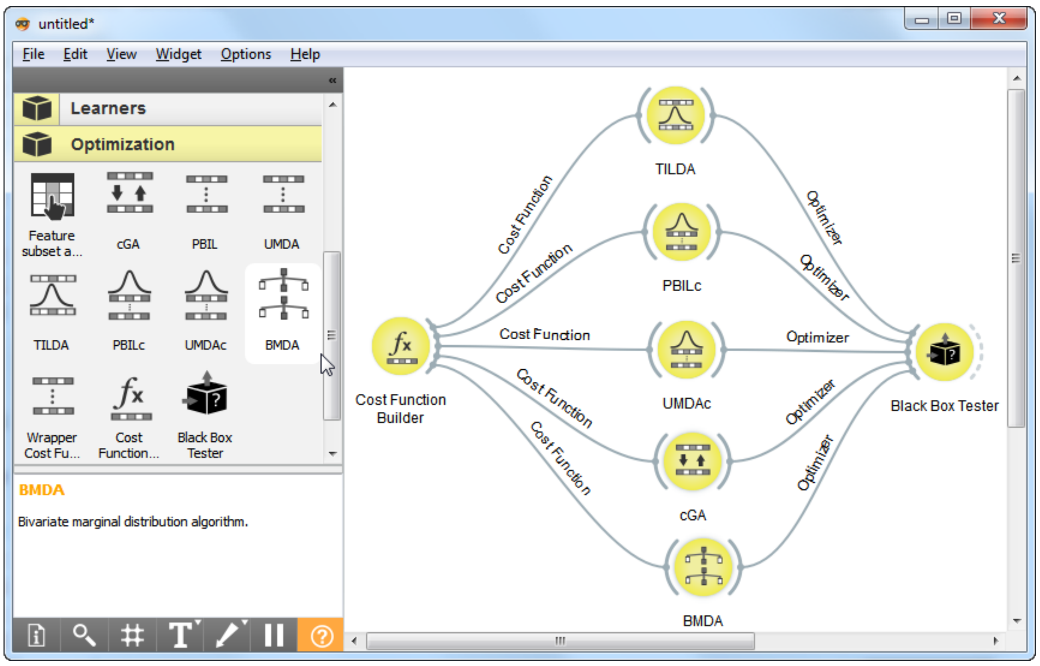


Figure 3.1: A `Goldenberry` program in the `Orange` canvas.

3.2 Architectural view

3.2.1 General design considerations

The following software patterns were taken into account during the conception of `Goldenberry`:

- A *layering* pattern [9] was applied in order to decouple the algorithmic logic of the components from their user interface (widgets); thereby all the implemented algorithms can be used and integrated in plain Python scripts or using the command-line.

- A *template method* pattern [17] was used by identifying commonalities of the different EDA approaches. In this way a generic abstraction of an EDA procedure was defined; concrete details of implementation were deferred to each particular algorithm. A unified EDA framework was achieved.

- A *responsibility splitting* pattern [3] was utilised in EDA optimisers so as to separate the search metaheuristic from the probability estimation technique. Therefore probability models were made reusable among multiple EDAs (those included in this and future releases).

- A *dynamic binding* [3] together with an *interpreter* pattern [17] were applied in order to allow the user to provide customised cost functions at runtime, in the form of Python-like scripts.

The application of such patterns guided the resulting design of the software, as described in the remainder of this section.

3.2.2 Architecture packages

Widgets are actually visual wrappers for software components written in the Phyton scripting feature provided by `Orange`. Implementation of any widget or suite of widgets require the use of the `Orange` core library, the `PyQt4` library for user-interface designing tools and optionally, the `NumPy` library for additional scientific computation support (`PyQt4` and `NumPy` are standard libraries of the Python programming language). Therefore, the top-level organization of the `Goldenberry` suite of widgets is illustrated in the context diagram of Figure 3.2.

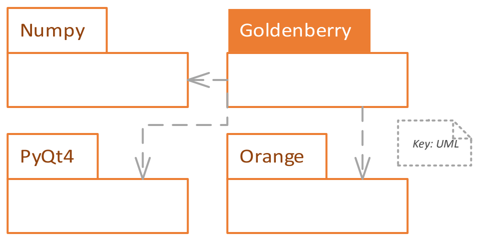


Figure 3.2: The context diagram of `Goldenberry`.

The internal architecture of the current release is shown in the module diagram of Figure 3.3. Two main modules were developed. The `widgets` module contains the visual wrappers for the suite of `Goldenberry` components. The `Algorithms` module includes objects and routines needed to implement the optimization, feature selection and classification components. Finally, the sub-module `Statistics` is a utility module to allow modeling of some probability distributions used by the `EDA` algorithms.

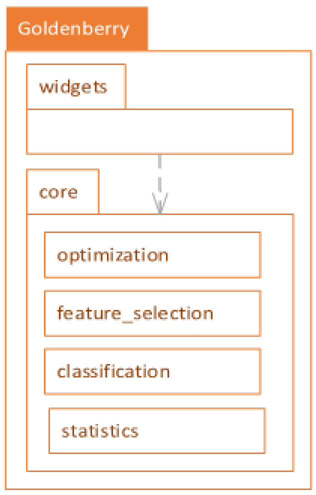


Figure 3.3: The basic modules in `Goldenberry`.

3.2.3 Structural view

A decomposition view of the `Core` module including object, classes and dependencies, is shown in the class diagram of Figure 3.4. It is expected that the modular design proposed here, would allow for new or customized `EDA` algorithms, probability distributions or other stochastic-search optimizers, to be added to the toolbox as extensions of such architecture. This would be one of the advantages of observing the software patterns earlier mentioned in Section 3.2.1.

Base classes

A core class `GbBaseOptimizer` was designed as an abstract class representing any black-box optimization algorithm (see Figure 3.4). The optimizer uses a `GbSolution` object, which holds the values of the parameters or variables in an arbitrary solution to a given problem (the vector `params[]`); it also holds its associated cost. The optimizer additionally encapsulates a `setup()` method to initialize its running parameters (e.g. problem size or number of variables, maximum number of evaluations, etc.) and the `reset()` method to set up the optimizer for a new run. The key method `search()`, defines the actual optimization algorithm

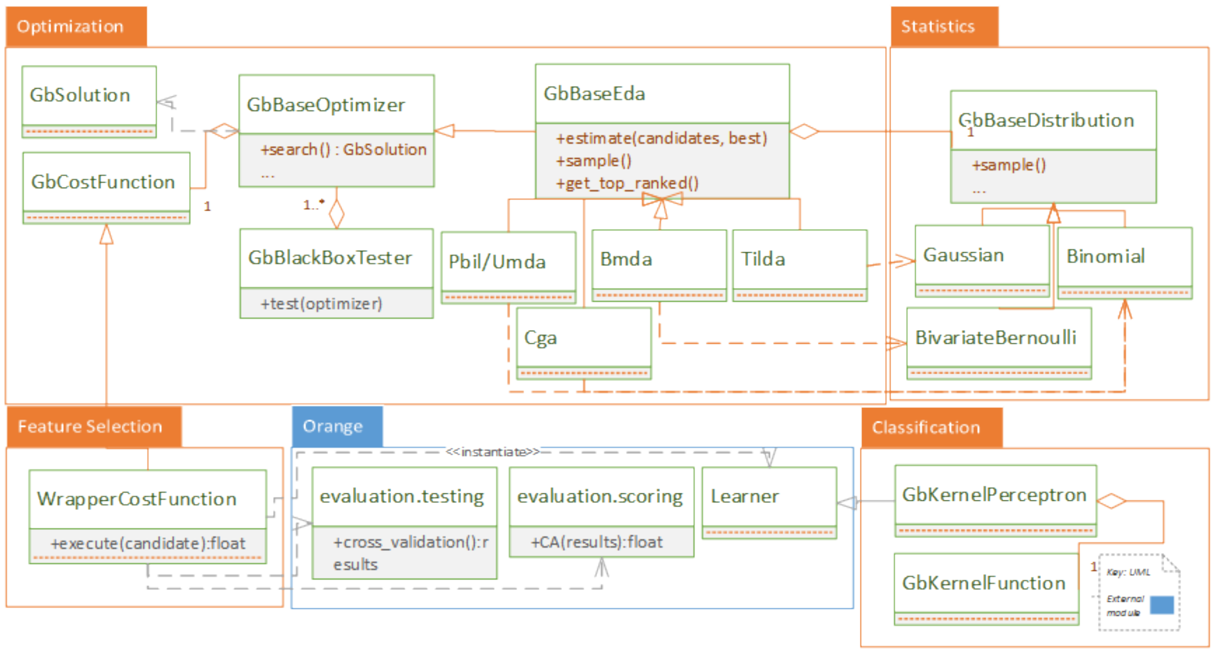


Figure 3.4: Decomposition view of `Goldenberry` modules.

which returns a best found solution; `ready()` is a checkpoint method to validate readiness of the optimizer to start the search.

A given candidate solution is evaluated with the `cost()` method from the `GbCostFunction` class (in the evolutionary computation literature this would be equivalent to computing the *fitness* of a candidate). The routine to evaluate the function is defined via the `set_func_script()` method. Furthermore, this class also includes a method to keep track of `statistics()`, such as the number of cost function evaluations or max/min/mean cost values; the `reset_statistics()` method clears up the statistics working memory.

EDA classes

In the current release `Goldenberry` provides implementation of only `EDA`-type optimizers. Other black-box optimization techniques would be added in the near future. Hence, a specialized abstract class `GbBaseEda` was derived from `GbBaseOptimizer` (see Figure 3.4). This class defines the two distinctive methods of any `EDA`: `estimate()` as the mechanism that builds up its probabilistic model, and `sample()` as the algorithm to generate new candidate solutions from that model. Two additional methods were designed: `get_top_ranked()` selects the promising candidates from where the probability model estimation is updated; and `done()`, which checks for convergence of the estimated model. These are the methods that would be iteratively executed during a run of the `search()` method from the parent class.

A particular implementation of the `GbBaseEda` class aforementioned, determines a type of `EDA` algorithm that would be used as optimizer. `Goldenberry` features a number of concrete `EDA` implementations, including univariate algorithms such as `cGA`, `TILDA`, `PBIL` and `UMDA` (discrete and continuos-domain variants for the last two), and also a bivariate algorithm, the `BMDA`. The latter extends the base class with an additional method `build_graph()` to model pairwise dependencies between problem variables. This assortment of algorithms was chosen so as to incorporate techniques using both univariate and bivariate probability distribution approaches. In the first release of the software we emphasised in univariate versions due to their algorithmic simplicity; nonetheless, future releases will build upon these algorithms and contemplate higher-order `EDA`s using tree-based, Bayes and dependency networks techniques.

For illustration purposes, we shall now outline some of the Python scripts implementing these classes. Firstly, let us recall the *Population-Based Incremental Algorithm* (`PBIL`)(see [4] and Section 1.4.1). The aim of this algorithm is to discover a real-valued probability vector from which a population of competent binary candidate solutions can be sampled. The algorithm starts-off with a random-valued vector; then, the vector is iteratively sampled in order to refine the model using the most promising solutions from the sample and an *incremental learning* re-estimation rule, until the vector converges to a fixed distribution.

In `Goldenberry`, the `PBIL` widget defines a class `Pbil` which overrides the `initialization()` and `estimate()` methods according to the previous algorithm. Thus, `PBIL` is fully implemented as the following class script (notice that the `average` vectorized auxiliary operation from the `NumPy` library (`np`) is used):

```
class Pbil(GbBaseEda):
    def initialize(self):
        self.distr = Binomial(self.var_size)

    def estimate(self, top_ranked, best):
        self.distr.p =  self.distr.p*(1-self.learning_rate) /
         + self.learning_rate * np.average(top_ranked, axis = 0)
```

The small size of the script is due to the fact that many algorithmic details have been inherited from the parent class, `GbBaseEda`, because they are common to most of other `EDA`s. For example, the parent class defines the initialization of parameters such as `sample_size`, `pop_size` (size of the population), `selection_rate` (percentage of top-ranked selected candidates), and `max_evals` (limit on the number of cost function evaluations allowed on an entire search). Other common features such as the `sample()` method (which delegates this task to the respective probability model), the `get_top_ranked()` method to select the most promising candidates from the sample, and the `search()` method itself, are defined in this abstract class. Its script is partially shown below.

```
class GbBaseEda(GbBaseOptimizer):
...
 def sample(self, sample_size, top_ranked, best):
    return self.distr.sample(sample_size)

 def get_top_ranked(self, candidates):
         fits = self.cost_func(candidates)
         index = np.argsort(fits) /
             [:(self.cand_size * self.selection_rate/100):-1]
         return candidates[index], /
            self.build_solution(candidates[index[0]], /
          fits[index[0]])
...

 def search(self):
         if not self.ready():
             raise Exception("Optimizer not ready.")
         best = GbSolution(None, float('-Inf'))
         top_ranked = None
         while not self.done():
             self.iters += 1
             candidates = self.sample(self.sample_size, top_ranked, best)
             top_ranked, winner = self.get_top_ranked(candidates)
             self.estimate(top_ranked, best)

             if best.cost < winner.cost:
                 best = winner

             if self.callback_func is not None:
                 self.callback_func(best, self.cost_func.evals /
                  float(self.max_evals))

         if self.callback_func is not None:
                 self.callback_func(best, 1.0)
         return best

 @abc.abstractmethod
 def estimate(self, candidates, best):
     #each concrete class must implement this abstract method.
     raise NotImplementedError()
```

It can be seen in the previous code that the implementation of `estimate()`, the estimation of distribution method, is deferred to the specific `EDA` class, as it was the case of the `PBIL` component.

`Goldenberry` also incorporates the `cGA`(see [22] and Section 1.4.1). The algorithm is similar in fashion to `PBIL`, the main difference being that it operates in a *compact* mode, that is, instead of estimating the distribution from a batch of many candidates (population), the algorithm works by sampling two candidates at a time and using them to incrementally build the estimation.

The `cGA`, as implemented in full in `Goldenberry`, is shown in the script below.

```
class Cga(GbBaseEda):
   def initialize(self):
       self.distr=Binomial(self.var_size)
       self.learning_rate=1.0/float(self.pop_size)
       self.sample_size=2

   def estimate(self, (winner, loser), best):
       self.distr.p =
         np.minimum(np.ones((1, self.var_size)),
           np.maximum(np.zeros((1, self.var_size)),
           self.distr.p +
  (winner.params-loser.params)*self.learning_rate))

   def get_top_ranked(self, candidates):
       costs = self.cost_func(candidates)
       maxindx = np.argmax(costs)
       winner = GbSolution(candidates[maxindx],
                           costs[maxindx])
       loser = GbSolution(candidates[not maxindx],
                          costs[not maxindx])
       return  (winner, loser), winner
```

The algorithm is initialized with the following settings: a binomial univariate probability model the size of the number of variables; learning rate of $1/n$ (where n is the population size); and sample size of two, because of its *compact* style. The `estimate()` method is overridden to account for a compact learning rule, using two competitors; here again, the auxiliary `NumPy` library is used (`np`). The `get_top_ranked()` is overridden accordingly, to return the winner and loser of the contest as the top-ranked set.

Statistics classes

The classes comprising this module were designed to account for the mechanisms needed to modeling probability distributions. A `GbBaseDistribution` is defined encapsulating a `sample()` method, a `has_converged()` validation method, and a `reset()` method for randomly initializing the parameters of the probability model (see Figure 3.4). Currently four probability distributions are implemented: `Binomial`, `Gaussian`, `TruncatedGaussian` and `BivariateBinomial`. Most of the functionality of these classes were implemented using standard `NumPy` routines such as generation of normally independent distibuted random vectors, and element-wise vector operators (details omitted).

3.3 EDA widgets

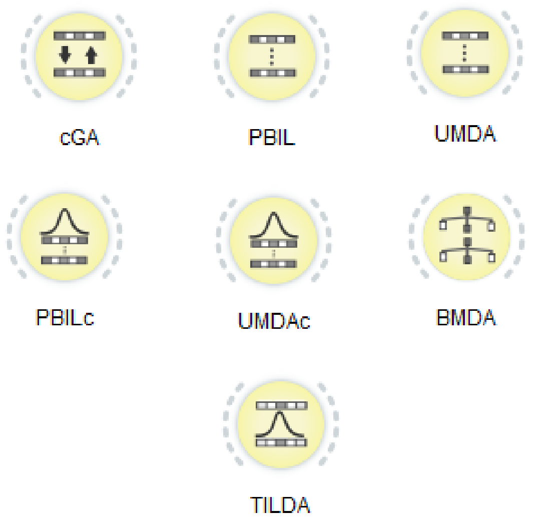
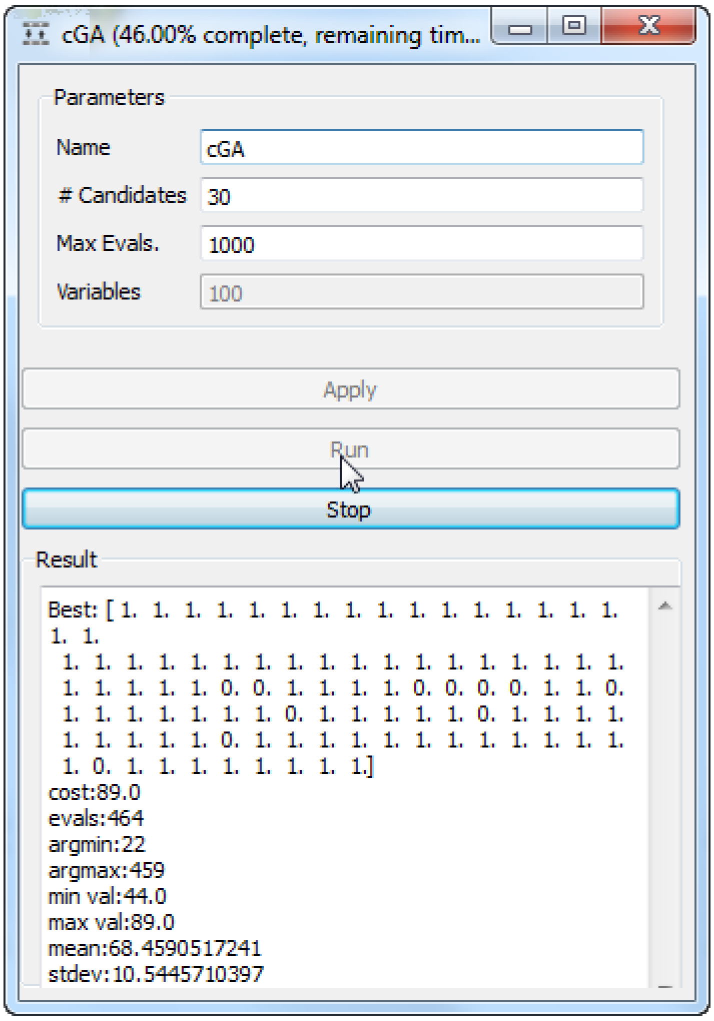


Figure 3.5: List of `Goldenberry`'s `EDA`s widgets.

`Goldenberry` provides the following `EDA` widgets: `cGA`, `UMDA`, `PBIL`, `TILDA`, $\texttt{PBIL}_c$, $\texttt{UMDA}_c$and `BMDA` as depicted in Figure 3.5. All of them are available under a new `Orange` canvas toolbar named “Optimization” (see Figure 3.1). All `EDA`s widgets uses a similar user interface. This interface consists of a setup/results window. In there, parameters settings are applied to the `EDA` widget and optimization progresses and results are displayed in an output text box during and after algorithm's execution. One example is the `cGA` widget, applied to solve the classical `OneMax` problem with 100 variables, using a population size of 30, 1000 maximum number of evaluations, and 30 candidates per iteration depicted in Figure 3.3. The other `EDA` widgets follows the same user interface concept but with some additional configuration parameter. For example `PBIL` and $\texttt{PBIL}_c$ uses a *learning rate* parameter also available by the user interface and `BMDA` allows to select the *dependency method* and variables dependencies are displayed in a hierarchical view.(see Figure 3.3).



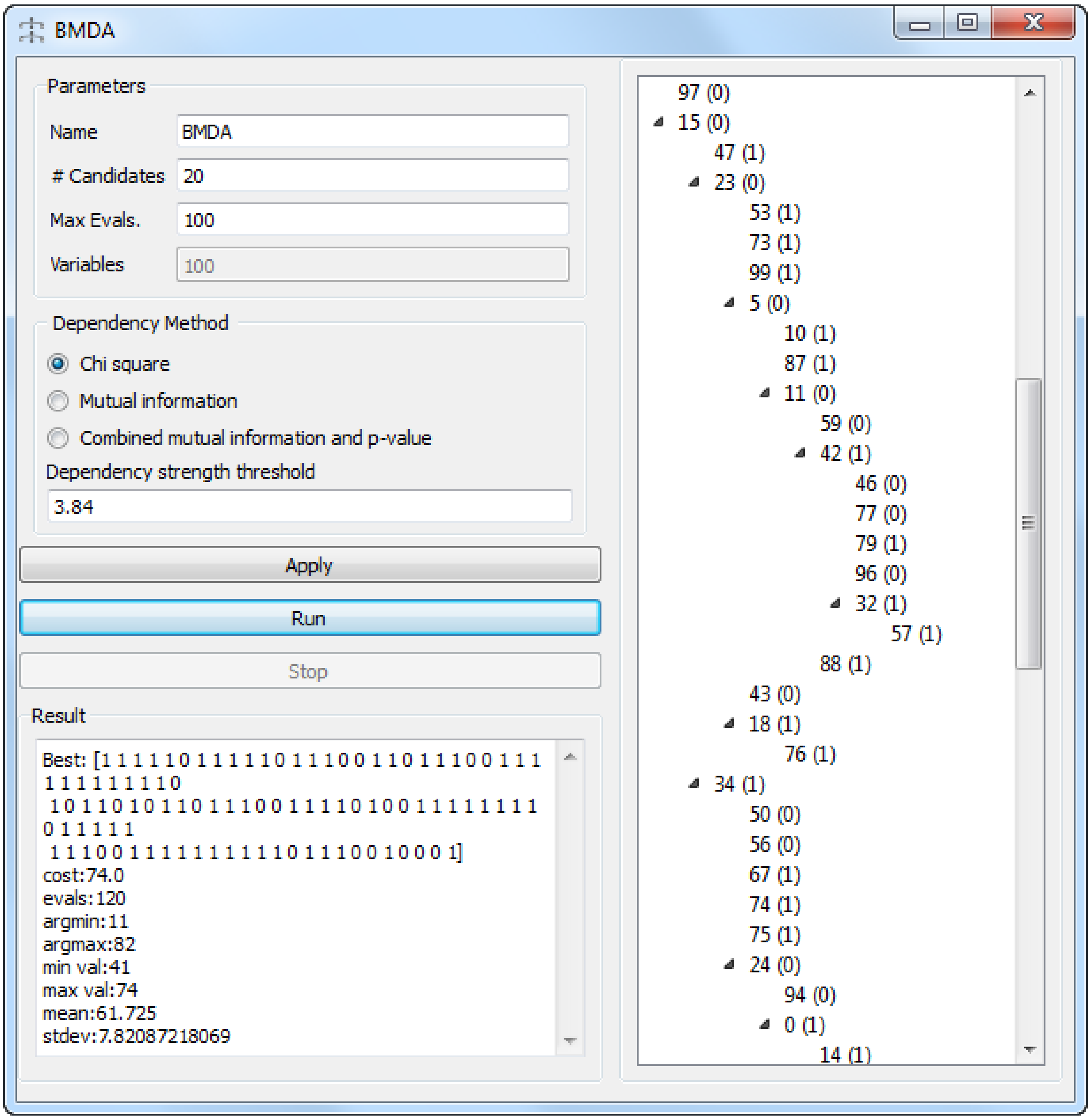
Parameters

Name: cGA
Candidates: 30
Max Evals.: 1000
Variables: 100

Result

```
Best: [ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
1.  1.
  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
  1.  1.  1.  1.  1.  0.  0.  1.  1.  1.  1.  0.  0.  0.  0.  1.  1.  0.
  1.  1.  1.  1.  1.  1.  1.  0.  1.  1.  1.  1.  1.  0.  1.  1.  1.  1.
  1.  1.  1.  1.  1.  0.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
  1.  0.  1.  1.  1.  1.  1.  1.  1.  1.]
cost:89.0
evals:464
argmin:22
argmax:459
min val:44.0
max val:89.0
mean:68.4590517241
stdev:10.5445710397
```

Figure 3.6: User interface of the `cGA` widget.



Parameters

Name	BMDA
# Candidates	20
Max Evals.	100
Variables	100

Dependency Method

[✓ Chi square]
Mutual information
Combined mutual information and p-value

Dependency strength threshold: 3.84

Result

```
Best: [1 1 1 1 1 0 1 1 1 1 1 0 1 1 1 0 0 1 1 0 1 1 1 0 0 1 1 1
1 1 1 1 1 1 1 1 0
 1 0 1 1 0 1 0 1 1 0 1 1 1 0 0 1 1 1 1 0 1 0 0 1 1 1 1 1 1 1 1
0 1 1 1 1 1
 1 1 1 0 0 1 1 1 1 1 1 1 1 1 1 0 1 1 1 0 0 1 0 0 0 1]
cost:74.0
evals:120
argmin:11
argmax:82
min val:41
max val:74
mean:61.725
stdev:7.82087218069
```

```
97 (0)
15 (0)
    47 (1)
    23 (0)
        53 (1)
        73 (1)
        99 (1)
        5 (0)
            10 (1)
            87 (1)
            11 (0)
                59 (0)
                42 (1)
                    46 (0)
                    77 (0)
                    79 (1)
                    96 (0)
                    32 (1)
                        57 (1)
                88 (1)
        43 (0)
        18 (1)
            76 (1)
    34 (1)
        50 (0)
        56 (0)
        67 (1)
        74 (1)
        75 (1)
        24 (0)
            94 (0)
            0 (1)
                14 (1)
```

Figure 3.7: User interface of the BMDA widget.

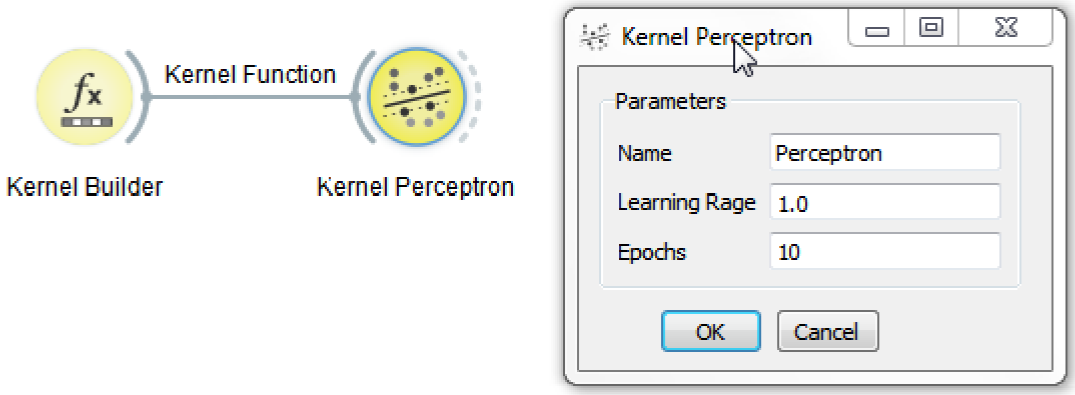
3.4 Feature selection widgets

Goldenberry provides the following feature selection widgets: FeatureSubset and WraperCostFunction. The FeatureSubset-filters the input dataset features and the

`WraperCostFunction` is the key component to build `FSS` *wrapper* methods, as `wKiera` and `Kiedra`. More details are given in the Chapter 4

3.5 Kernel classifier widgets

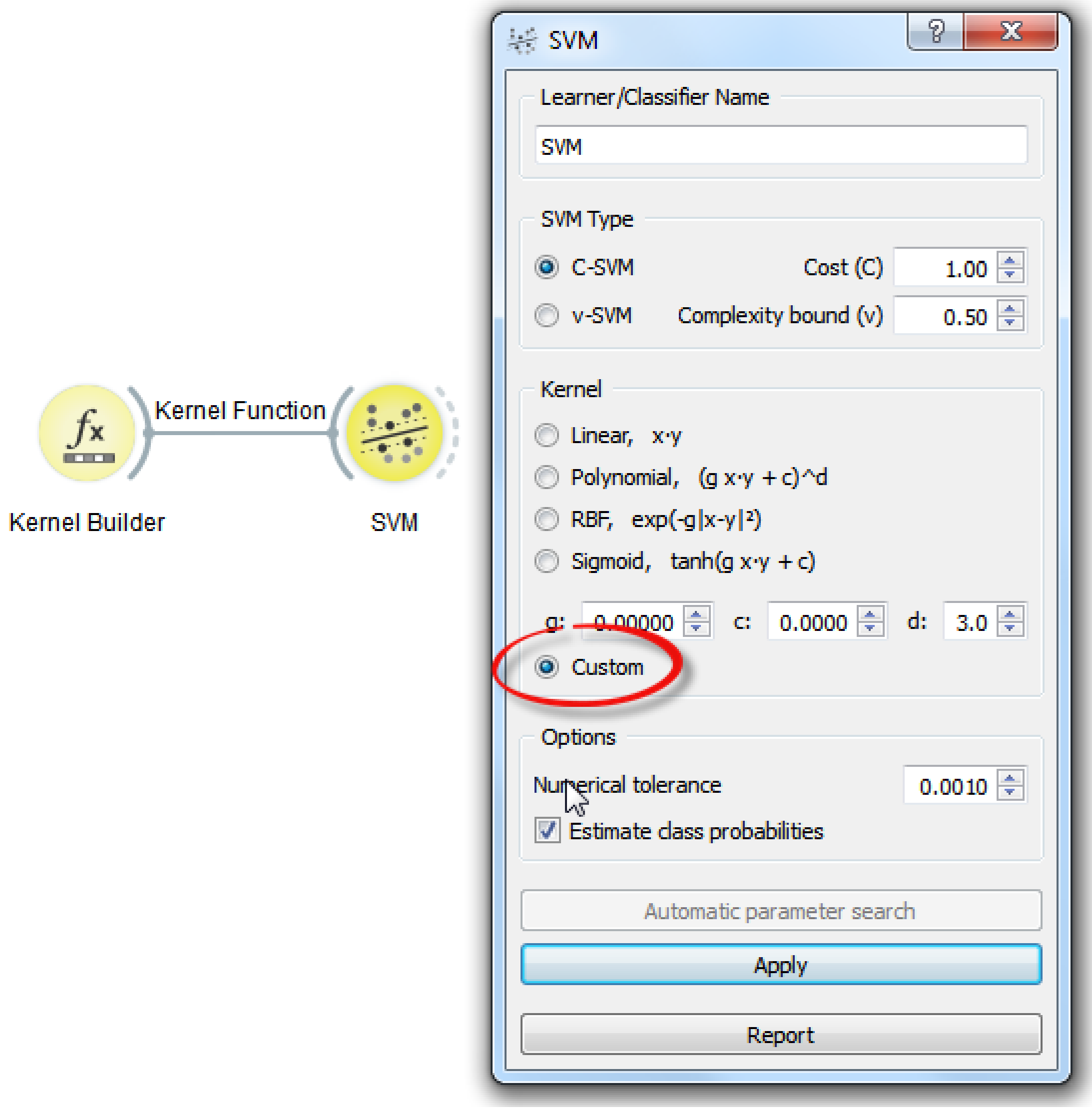
The `Kernel Perceptron` widget is responsible to provide a ready-to-execute kernel classifier (see Appendix A). The widget requires a `GbKernelFunction`, a *learning rate* and the number of learning *epochs* (see Figure 3.5). An *epoch* is a learning iteration where the training dataset is processed only once.



Name	Perceptron
Learning Rage	1.0
Epochs	10

Figure 3.8: `Kernel Perceptron` widget configuration.

The `SVM` widget is an adaptation of the original version provided by `Orange` able to use custom kernels defined in the `KernelBuilder` widget (see Figure 3.5). If there is no a special need for a custom kernel we advise to use the native `Orange` kernels because customized versions written in Python may have additional runtime overhead.



SVM

Learner/Classifier Name

SVM

SVM Type

[✓ C-SVM] Cost (C) 1.00

v-SVM Complexity bound (v) 0.50

Kernel

Linear, x·y

Polynomial, (g x·y + c)^d

RBF, exp(-g|x-y|²)

Sigmoid, tanh(g x·y + c)

g: 0.00000 c: 0.0000 d: 3.0

[✓ Custom]

Options

Numerical tolerance 0.0010

[✓] Estimate class probabilities

Automatic parameter search

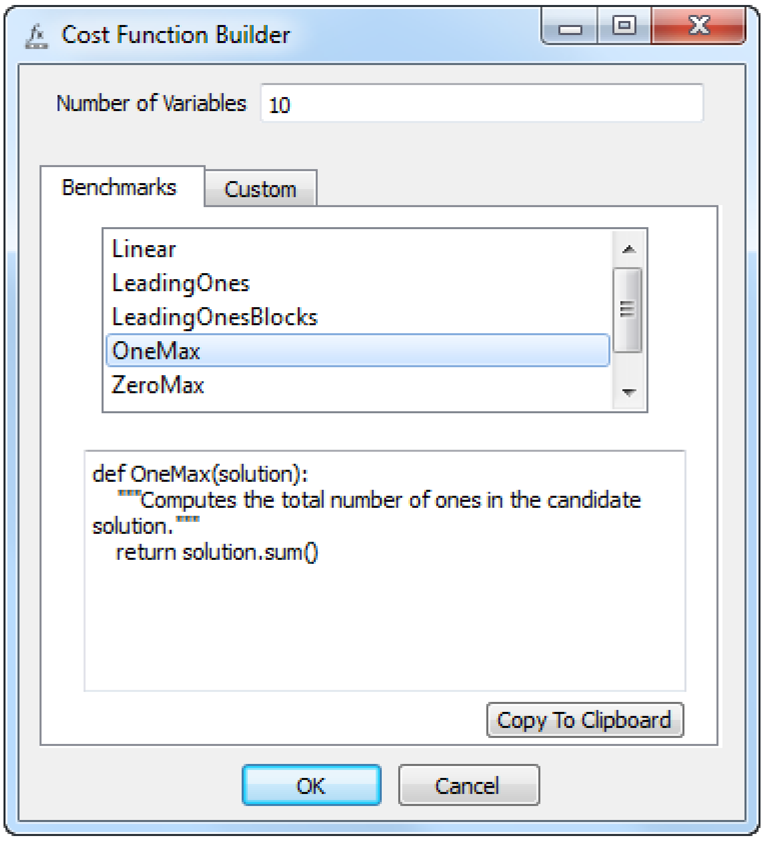
Apply

Report

Figure 3.9: `SVM` widget configuration.

3.6 Utility widgets

The `CostFunctionBuilder` widget enables the user to define the cost function for the optimization problem. It can be found under the “Optimization” tool-bar. It provides two modes to configure such a function and allows to set the number of problem’s variables. In the first mode, a set of ready-to-use built-in benchmark functions are listed to the user as it is illustrated in Figure 3.10. This mode appears on the “Benchmarks” tab of the `CostFunctionBuilder` user interface. The user chooses a function name, and then the Python script implementing the function is shown in the underlying text box; the box is read-only, so the code can not be edited but can be copied into the clipboard. The benchmark functions were taken from those suggested in [25].



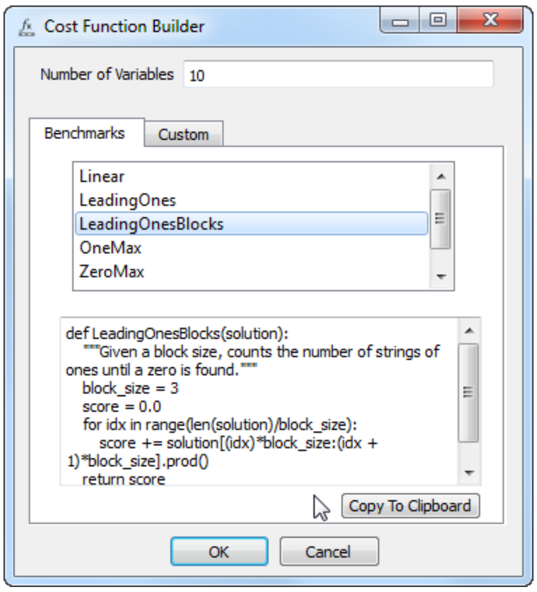
```
def OneMax(solution):
    """Computes the total number of ones in the candidate
solution."""
    return solution.sum()
```

Figure 3.10: The benchmark input mode of the `CostFunctionBuilder` component. Here the well-known `Onemax` problem is chosen.

The second mode consists of a free-text input box for writing up any customized cost function in the Python language. This mode appears on the "Custom" tab of the `CostFunctionBuilder` user interface. Custom functions must be written complying with the following Python-style signature:

```
def yourcustomfunctionname(solution):
...
return computedcostofsolution
```

An arbitrary routine to evaluate the given `solution` (a `NumPy` 1D vector array with the values of that solution to the problem variables) must be defined in order to compute its associated cost. For example, if the user wants to define a customized version of the benchmarks built-int functions, he or she may copy to the clipboard its original code, then paste it in the "Custom" tab and make the necessary adjustments (see Figure 3.11). Alternatively the user may write up his cost function code from scratch, to meet his particular problem needs. We remark that this a is non-intrusive input mode, meaning that the user-written-code is bind to the `Goldenberry` components in run time; no additional intervention has to be done in the source code of the software. We anticipate this feature would extend the usability of the `EDA` components to a wide range of discrete and continuous optimization problem domains.



Cost Function Builder

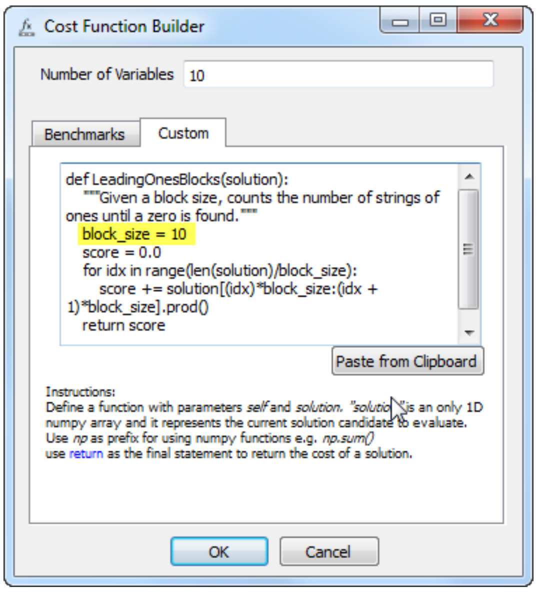
Number of Variables 10

Benchmarks | Custom

- Linear
- LeadingOnes
- LeadingOnesBlocks
- OneMax
- ZeroMax

```
def LeadingOnesBlocks(solution):
    """Given a block size, counts the number of strings of
ones until a zero is found."""
    block_size = 3
    score = 0.0
    for idx in range(len(solution)/block_size):
        score += solution[(idx)*block_size:(idx +
1)*block_size].prod()
    return score
```

(a)



Cost Function Builder

Number of Variables 10

Benchmarks | Custom

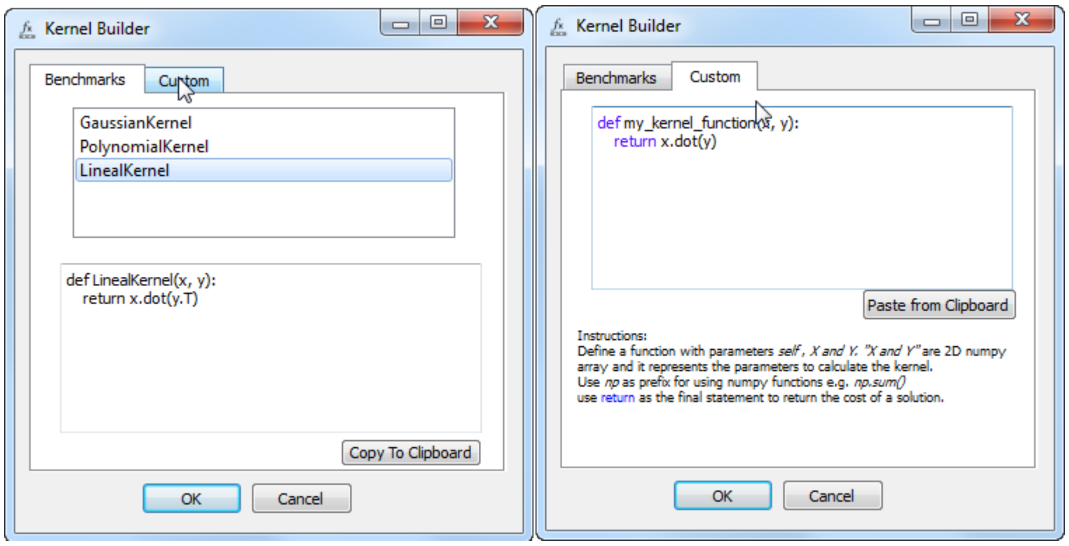
```
def LeadingOnesBlocks(solution):
    """Given a block size, counts the number of strings of
ones until a zero is found."""
    block_size = 10
    score = 0.0
    for idx in range(len(solution)/block_size):
        score += solution[(idx)*block_size:(idx +
1)*block_size].prod()
    return score
```

Instructions:
Define a function with parameters *self* and *solution*. *"solution"* is an only 1D numpy array and it represents the current solution candidate to evaluate.
Use *np* as prefix for using numpy functions e.g. *np.sum()*
use return as the final statement to return the cost of a solution.

(b)

Figure 3.11: How to customize a built-in benchmark function: (a) A benchmark function is chosen and its code is copied into the clipboard; (b) the code is pasted in the "Custom" tab and edited as required.

The `KernelBuilder` provides the same services as the `CostFunctionBuilder` but for building kernel functions. A kernel function takes two input vectors $\mathbf{x}, \mathbf{y}$ and returns a valid kernel computation (see [50]). There are a set of predefined kernel functions and it is also possible to create your own (see Figure 3.12).



```
def LinealKernel(x, y):
  return x.dot(y.T)
```

```
def my_kernel_function(x, y):
  return x.dot(y)
```

Instructions:
Define a function with parameters *self*, *X and Y*. "*X and Y*" are 2D numpy array and it represents the parameters to calculate the kernel.
Use *np* as prefix for using numpy functions e.g. *np.sum()*
use return as the final statement to return the cost of a solution.

Figure 3.12: How to define a kernel function by the `KernelBuilder` widget.

The `BlackBoxTester` widget was designed to allow the user to run and compare execution of different algorithms or algorithm configurations over the same cost function, in one experiment with several repetitions. The number of repetitions is set as an input parameter for this component. Another interesting functionality of the `BlackBoxTester` is that it is able to collect outputs of all the optimizer components provided as inputs, and display summarized statistics, and also details of the different runs and repetitions. The user interface of the widget associated to this component will be used to display results for the working experiments reported in the next section.

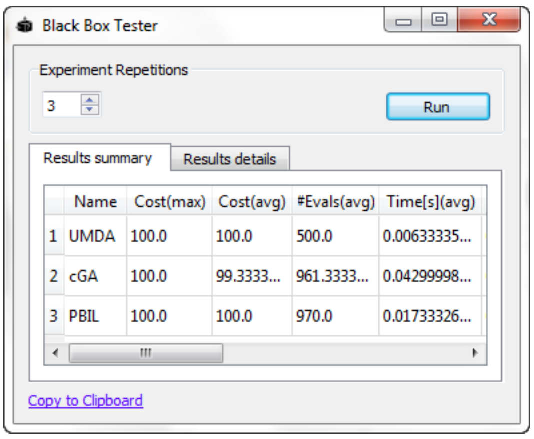
3.7 Working examples

In this section we show how to use the developed components to solve optimization problems defined as minimization of a cost function. We carried out examples using benchmark and customized cost functions. Other uses in machine learning can be also envisioned (see for example the discussion in Section 3.8).

The results of these examples are shown next. It is worth to remark that previous to deployment, additional validation for the algorithmic machinery of each of the `Goldenberry` components was carried-out using a carefully designed set of unit tests. The program shown in Figure 3.1 was used in these examples: A `CostFunctionBuilder` component is instantiated to select the optimization problem, as explained before; three `EDA` components were tested (`cGA`, `PBIL` and `UMDA`) using as input the selected `CostFunction`; finally, a `BlackBoxTester` executes and collects the outputs of the `EDA` components and shows summarised and detailed results over a number of repetitions of the experiment.

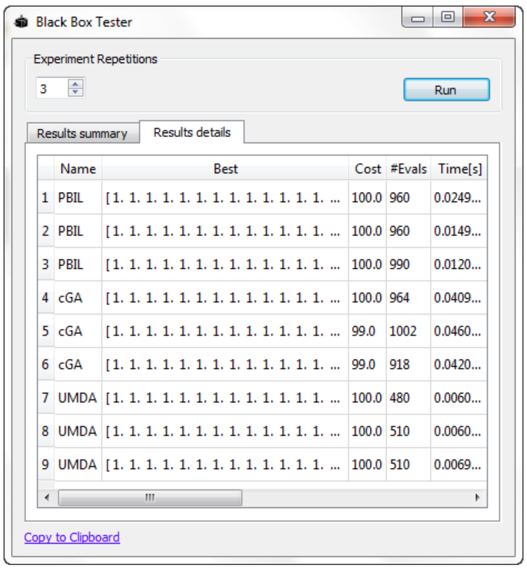
Benchmark function optimization

Two problems were chosen from the built-in benchmark library of functions: `Onemax` and `LeadingOnesBlock`. Results for the `Onemax` experiment are reported in Figure 3.13. Figure 3.13(a) shows the following aggregate results per `EDA` algorithm: maximun cost found in all experiment repetitions; average cost over all repetitions; average number of cost function evaluations per repetition; and average CPU time per repetition (in seconds). Other statistics can be displayed by scrolling the table bar to the right. Notice that the obtained tabulated results can be exported to spreadsheet software tools for further analysis by using the "Copy to Clipboard" option located underneath the output table.



	Name	Cost(max)	Cost(avg)	#Evals(avg)	Time[s](avg)
1	UMDA	100.0	100.0	500.0	0.00633335...
2	cGA	100.0	99.3333...	961.3333...	0.04299998...
3	PBIL	100.0	100.0	970.0	0.01733326...

(a)



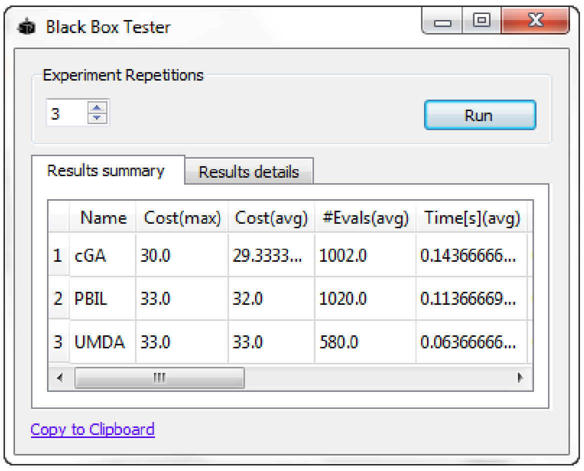
	Name	Best	Cost	#Evals	Time[s]
1	PBIL	[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. ...	100.0	960	0.0249...
2	PBIL	[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. ...	100.0	960	0.0149...
3	PBIL	[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. ...	100.0	990	0.0120...
4	cGA	[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. ...	100.0	964	0.0409...
5	cGA	[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. ...	99.0	1002	0.0460...
6	cGA	[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. ...	99.0	918	0.0420...
7	UMDA	[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. ...	100.0	480	0.0060...
8	UMDA	[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. ...	100.0	510	0.0060...
9	UMDA	[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. ...	100.0	510	0.0069...

(b)

Figure 3.13: `Onemax` experiment results: (a) Summary over all repetitions (partial view); (b) Details per repetition (partial view).

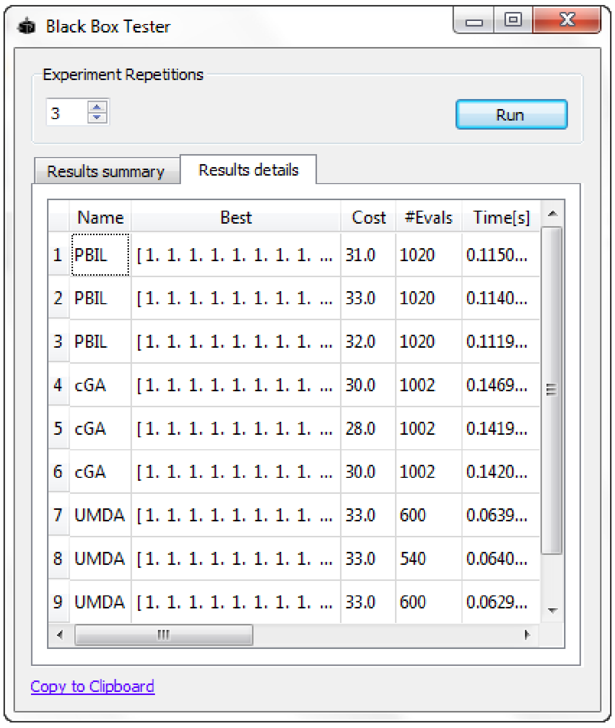
Figure 3.13(b) shows a detailed view of the results obtained in the experiments. Here each row holds the outputs per repetition for each input `EDA` component. The summarized results are statistics of these individual records. The only column not used in the summarized report is the actual vector representation of the best solution found in each run (for `Onemax` the expected solution is an all-ones vector). The cost of this solution is shown in the next column to its right.

For the `LeadingOnesBlock` experiment, results are similarly self-explained, as reported in Figure 3.14.



	Name	Cost(max)	Cost(avg)	#Evals(avg)	Time[s](avg)
1	cGA	30.0	29.3333...	1002.0	0.14366666...
2	PBIL	33.0	32.0	1020.0	0.11366669...
3	UMDA	33.0	33.0	580.0	0.06366666...

(a)



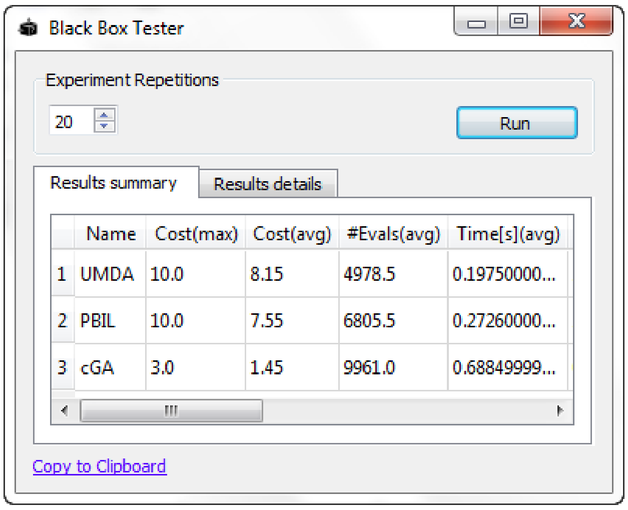
	Name	Best	Cost	#Evals	Time[s]
1	PBIL	[1. 1. 1. 1. 1. 1. 1. 1. ...	31.0	1020	0.1150...
2	PBIL	[1. 1. 1. 1. 1. 1. 1. 1. ...	33.0	1020	0.1140...
3	PBIL	[1. 1. 1. 1. 1. 1. 1. 1. ...	32.0	1020	0.1119...
4	cGA	[1. 1. 1. 1. 1. 1. 1. 1. ...	30.0	1002	0.1469...
5	cGA	[1. 1. 1. 1. 1. 1. 1. 1. ...	28.0	1002	0.1419...
6	cGA	[1. 1. 1. 1. 1. 1. 1. 1. ...	30.0	1002	0.1420...
7	UMDA	[1. 1. 1. 1. 1. 1. 1. 1. ...	33.0	600	0.0639...
8	UMDA	[1. 1. 1. 1. 1. 1. 1. 1. ...	33.0	540	0.0640...
9	UMDA	[1. 1. 1. 1. 1. 1. 1. 1. ...	33.0	600	0.0629...

(b)

Figure 3.14: `LeadingOnesBlock` experiment results: (a) Summary (partial view); (a) Details (partial view).

Customized function optimization

In this experiment we used the same Goldenberry program of Figure 3.1 to carry out optimization of the customized cost function defined in Figure 3.11, that is, the same LeadingOnesBlock problem this time with a block_size value of 10. Similarly to the other experiments, results are shown in Figure 3.15.



	Name	Cost(max)	Cost(avg)	#Evals(avg)	Time[s](avg)
1	UMDA	10.0	8.15	4978.5	0.19750000...
2	PBIL	10.0	7.55	6805.5	0.27260000...
3	cGA	3.0	1.45	9961.0	0.68849999...

Figure 3.15: Results of custom function experiment.

3.8 Discussion

User-friendly open-source visual tools may have a big potential benefit in tasks carried out daily by data mining analysts. The Orange platform is a fantastic effort complying with these premises by combining a powerful visual programming approach with the reuse and glue principles of component-based software. The Goldenberry initiative is a modest contribution intended to extend the application domain of Orange to the field of black-box and metaheuristics optimization and feature selection. In this first release we developed a number of software components featuring a non-intrusive, runtime-binding interface to allow users to define tailor-made discrete and continuous optimization problems including feature selection.

Chapter 4

Kiedra: Kernel Iterative Estimation of Dependency and Relevance Algorithm

4.1 Motivation

In this chapter a new `FSS` method is proposed termed `Kiedra`. Its overall motivation is to identify relevant feature subsets and as much as possible information about dependencies that explain significant patterns hidden in the data. This method treats the features selection and patter discovery as an optimization problem where the classification accuracy from a learning algorithm is optimized in a search space of all-possible variables subsets, by combining a `BMDA` and a *kernel machine* into a new `FSS-EDA` *wrapper* method. Other similar attempts have been proposed like `wKiera` and `FSS-EBNA`, but the first assumes variables are independent and the second uses a simple Naive Bayes classifiers [32].

4.2 Method

`Kiedra` is a *wrapper* `FSS-EDA` (see Section 1.4.2) method. In a nutshell, `Kiedra` uses `BMDA` as the optimizer algorithm to find 2-order dependencies among features and a optimal feature subset. `BMDA` explores the search space by means of a set of candidates. Each candidate provides a feature relevance estimation of the problem's feature space and is assessed by a *wrapper cost function* ; the better the candidate solution portrays the problem's feature relevances the better fitness obtained. The fitness score is the classification accuracy of a kernel classifier (such as `SVM` or `Kernel Perceptron`) using a cross-validation process of n folds with the problem's *training data*. The fittest candidates are selected per iteration to reestimate the `BMDA`'s probability model parameters until they converge or a criterion met. Finally the best candidate is again evaluated but with unseen data, that is *test data*, in order to provide a more accurate score about how good the final feature subset generalizes the problem domain. A FSS-EDA method can be seen as a *machine learning process* [32] where the *training* phase occurs during the exploration of the search space and the *test* phase occurs when the best found feature subset is again evaluated but with unseen data to provide the final score of the entire FSS process.

Kiedra building-blocks are depicted in Figure 4.1. Their interactions and responsibilities are explained here below.

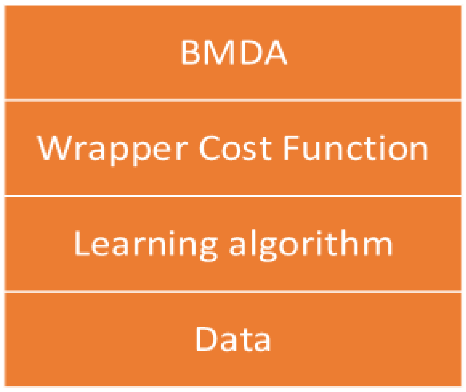


Figure 4.1: The Kiedra's building-blocks .

The Bivariate marginal estimation of distribution algorithm (BMDA)

We have introduced some variants to the canonical BMDA to accomplish our method and those are described below. Note that we based our explanation in concepts and notation defined in Section 1.4.1:

- Compared to canonical BMDA, in our method we introduced two additional dependency test criteria: *Mutual-information*(MI) [39] and the *Combined mutual-information and p-value*(SIM) [40]. This variant slightly changes edges estimation from Eq.(1.8) as shown in Eq.(4.1)

$$\mathbf{E}_t = \{(i,j) \in \mathbf{V} \times \mathbf{V} : i \neq j \wedge \mathsf{any\_of}(\{\chi^2_{ij} \geq 3.84, \mathsf{MI}_{ij} > 0, \mathsf{SIM}_{ij} > 0\})\} \tag{4.1}$$

- In contrast to the random approach for selecting root nodes $\mathbf{R}_{\mathsf{MSF}}$ from canonical BMDA, we introduced and entropy-based selection similar to MIMIC approach [7] as root variables that minimizes marginal entropy $H()$ in each connected component $\mathbf{V}_k$ from the current candidate pool $\mathcal{C}$ at iteration t, that is

$$\mathbf{R}_{\mathsf{MSF}} = \{r_k : r_k = \arg\min_i H(X_i \in \mathbf{V}_k)\}$$

- Finally we parallelize the population fitness evaluation to improve the algorithm performance, that is each candidate from $\mathcal{D}$ is evaluated in its own thread at iteration t.

BMDA is used in Kiedra to find an optimal feature subset and for that purpose BMDA requires a *wrapper cost function* responsible to determine the goodness of any feature subset from the search space.

Data

The *data* is the sample of observed variables for a given problem, defined as $\mathcal{D} = \{(\mathbf{s_1}, y_1), ..., (\mathbf{s_n}, y_n)) \in (\mathbf{S} \times Y)^n\}$ where $\mathbf{S} \in \mathbb{R}^\ell$ and $Y \in \{0, 1, ..., m\}$. This data is split in two groups: *training* $\mathbf{S_r}$ and *testing* $\mathbf{S_t}$. *Training* data is used for assessing the candidates' accuracy into the search process. *Testing* data is used exclusively with the best found solution to re-assess its prediction accuracy but with unseen data.

Wrapper cost function

The *wrapper cost function* $\mathcal{F}_w$ is a multi-objective cost function to assess the number of features and its goodness to produce an accurate predictor model. This is used as the key component to create a *wrapper* FSS-EDA method. The *wrapper cost function* $\mathcal{F}_w$ acts as an intermediary between the optimizer algorithm, in this case BMDA , and the learning machine (an SVM or Kernel Perceptron). BMDA provides candidates in the form of weighted vectors $\mathbf{w}_k \in [0, 1]^\ell$ where $k \in \{1, .., n\}$. Each candidate $\mathbf{w}_k$ is a relevance estimation of the feature space and component $w_{ki} = 0$ must be interpreted as variableX_i being irrelevant and $w_{ki} = 1$ as relevant. Other EDAs work in continuous domains like TILDA , in such cases $\mathbf{w}_k \in \mathbb{R}^\ell$ must be bounded within the real interval $[0.0, 1.0]$. BMDA evaluates the goodness of each candidate through the *wrapper cost function.* Therein, the relevance estimation from candidate $\mathbf{w}_k$ is induced to the input training data $\mathbf{S_r}$ by $\mathbf{w}_k \otimes \mathbf{S_r}$, where $\otimes$ is the component wise product. As a result, we assume the following relation:

$$K_{\mathbf{w}_k}(\mathbf{x}, \mathbf{z}) = K(\mathbf{w}_k \otimes \mathbf{x}, \mathbf{w}_k \otimes \mathbf{z}) \tag{4.2}$$

where $K_{\mathbf{w}_k}(\mathbf{x}, \mathbf{z})$ is a weighted kernel of kernel $K(\mathbf{x}, \mathbf{z})$ (see Section 1.4.3) induced by candidate $\mathbf{w}_k$. An example a weighted *RBF* kernel is obtained as follows

$$\begin{aligned} K_{\mathbf{w}_k}(\mathbf{x}, \mathbf{z}) &= \exp\left(-\sigma \sum_{i=1}^{\ell} w_{ki}(x_i - z_i)^2\right) && (4.3) \\ &= \exp\left(-\sigma \sum_{i=1}^{\ell} (\sqrt{w_{ki}}x_i - \sqrt{w_{ki}}z_i)^2\right) && (4.4) \\ &= K(\tilde{\mathbf{w}}_k \otimes \mathbf{x}, \tilde{\mathbf{w}}_k \otimes \mathbf{z}) && (4.5) \end{aligned}$$

where $\tilde{\mathbf{w}}_k = \{\tilde{w_{ki}} : \tilde{w_{ki}} = \sqrt{w_{ki}} \wedge w_{ki} \in \mathbf{w}_k\}$.

To asses the fitness of a $\mathbf{w}_k$ into the search process (*training* phase) the *wrapper cost function* $\mathcal{F}_w$ uses the weighted *training* data $\mathbf{S_{wr}}$, a learner machine $\mathcal{A}$ and a cross-validation process of c folds to provide the prediction accuracy as the fitness score. The fitness of the best found candidate $\mathbf{w}_k^\dagger$ is again estimated (*test* phase) with unseen data to avoid over-fitting and to provide a more accurate measurement of the entire `FSS` process as recommended in [32]. Again in this phase a learner machine $\mathcal{A}$ is trained and tested by a cross-validation process of c folds but using *test data*. The evaluation of the *wrapper cost function* $\mathcal{F}_w$ can be summarized in Eq.(4.6).

$$\mathcal{F}_w(\mathbf{w}_k) = \mathsf{cross\_validation}(\mathbf{w}_k, \mathcal{A}, \mathbf{S}, Y, c) \tag{4.6}$$

Learning algorithm

A feature selection method can be considered good whether it is able to produced a subset of features with higher capabilities to create a predictor model than the original input features. A predictor model is created by an algorithm which takes the problem's data as input and selects a hypothesis from the hypothesis space, it is referred to as the *learning algorithm* [12]. `Kiedra` uses a supervise learning algorithm which aims to construct a predictor model using the input data $\mathbf{S}$ and its goodness is assessed by the ability to accurately predict on unseen data. In `Kiedra` a learner comes into play when `BMDA` evaluates its population by the *wrapper cost function* $\mathcal{F}_w$; there a learner is instantiated, trained and tested per each candidate $\mathbf{w}_k$, and its prediction accuracy becomes the candidate fitness. The `Kiedra` method is able to work with multiple supervised learner algorithms but our study has only contemplated `SVMs` due to its robust generalization abilities [50].

4.3 Algorithm

A formal depiction of `Kiedra` can be seen in Algorithm 4. In Step(1) the probability parameters from a bivariate binomial probability model (see Eq.(1.9)) are initialized with an independent join distribution. Step(2) initializes the candidates pool $\mathcal{C}$ with a random sampling of $\frac{n}{2}$ candidates. Step(3) realizes the *training* phase of `Kiedra`. This loop is repeated until a solution converges. Step(8) samples preserving best candidates as explained in Section(1.4.1). Step(5) iterates from all sample candidates to obtain their fitness and it can be seen as the realization of the *wrapper cost function*. Step(6) calculates the classification accuracy from each candidate as in Eq. (4.6). Step(8) selects candidates with higher classification accuracy **ca**. In Step(9) function $\mathsf{build\_graph}$ generates a graph as explained in 1.4.1. Step(10) is the realization of the second sub-step defined in Section 1.4.1 where the *minimum-spanning-forest* is constructed. Step(11) estimates the probability parameters $\boldsymbol{\theta}$ following Eqs.(1.9a, 1.9b). Step(12) updates the best candidate $\mathcal{B}$ only if best candidate from $\mathcal{C}$ has a higher fitness score. Step(14) is the realization of the so called *testing* phase from `Kiedra`; there the accuracy from the best subset found is again assessed but with unseen data $\mathbf{S_t}$. And Step(15) returns the probability model estimated $P(X;\boldsymbol{\theta})$, the best feature subset found $\mathcal{B}$ and its prediction accuracy $f_{\mathcal{B}}$.

Algorithm 4 `Kiedra` pseudo-code.

```
Requires: Training and testing datasets $\{\mathbf{S_r}, \mathbf{S_t}\}$, learner $\mathcal{A}$, weighted function
    $\phi$, population size $n$ and the number of training and test folds $(c_r, c_t)$.
 1: $\boldsymbol{\theta} \leftarrow \mathsf{initialize}(n)$
 2: $\mathcal{C} \leftarrow \mathsf{sample}(P(X;\boldsymbol{\theta}), \frac{n}{2})$
 3: repeat
 4:     $\mathcal{D} \leftarrow \mathsf{sample}(P(X;\boldsymbol{\theta}), \frac{n}{2}) \cup \mathcal{C}$
 5:     repeat $\mathbf{w}_k \in \mathcal{D}$
 6:         $f_k \leftarrow \mathsf{cross\_validation}(\mathbf{w}_k, \mathcal{A}, \mathbf{S_r}, Y, c_r)$
 7:     end repeat
 8:     $\mathcal{C} \leftarrow \mathsf{select}(\mathcal{D}, \mathbf{f})$
 9:     $\mathsf{G}(\mathbf{V}, \mathbf{E}) \leftarrow \mathsf{build\_graph}(\mathcal{C})$
10:     $\mathsf{F_{MSF}} \leftarrow \mathsf{MSF}(\mathbf{E})$
11:     $\boldsymbol{\theta} \leftarrow \mathsf{estimate}(\mathsf{F_{MSF}}, \mathcal{C})$
12:     $\mathcal{B} \leftarrow \mathsf{update\_best}(\mathcal{B}, \mathcal{C}, \mathbf{f})$
13: until $P(X;\boldsymbol{\theta})$ converged
14: $f_{\mathcal{B}} \leftarrow \mathsf{cross\_validation}(\mathcal{B}, \mathcal{A}, \mathbf{S_t}, Y, c_t)$
15: return $P(X;\boldsymbol{\theta})$, $\mathcal{B}$, $f_{\mathcal{B}}$
```

4.4 Implementation Using Goldenberry

`Kiedra` core classes

As described in Section 3.2.3 the logic from algorithms are implemented in the *core* package. To realize the *wrapper cost function* functionality a class called `GbWrapperCostFunction` was implemented in such package. This class inherit from the `GbCostFunction` class in order to act as a multi-objective cost function. This two input datasets one for training and the other for testing. Besides it uses some `Orange` scripting tools such as the `cross_validation` and classification accuracy `CA` (as shown in Figure 3.4) to assess the fitness of a set of candidates in parallel by the method `execute(self, solutions, is_latest)` where the `solution` represents the candidates to evaluate and the flag `is_latest` helps to determine whether evaluation is in training or testing time. A *weight* parameter is required to balance a multi-objective evaluation between the total number of features and their goodness to create a predictor model, the closer to 0 the more relevant is the prediction accuracy. Its script is partially shown below:

```
class WrapperCostFunction(GbCostFunction):
...
def execute(self, solutions, is_lastest):
...
enumerate(solutions):
         thread = th.Thread(
          target = test_solution, args = /
           [self.factory, weight, self.data
          , results, idx, self.folds])
         thread.start()
...
def test_solution(factory, weight, data, results, idx, folds):
    weighted_data = data.to_numpy("ac")[0] * /
     np.concatenate((weight, [1]))
    new_data = Orange.data.Table(data.domain, weighted_data)
    learner = factory()
    results[idx] = CA(cross_validation([learner], /
     new_data, folds = folds))[0]
```

Kiedra widgets

Goldenberry has been released with two key widgets in order to realize Kiedra: the WraperCostFunctionand the FeatureSubset. The WraperCostFunction widget can be seen as the front end of the GbWrapperCostFunction. To conduct Kiedra a set of widgets must be dragged and wired into the Orange canvas following Figure 4.4.

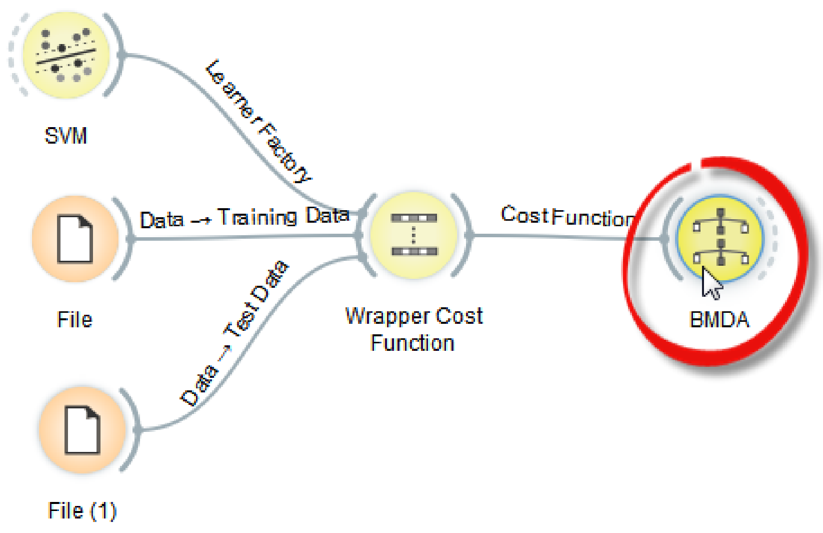


Figure 4.2: How to create Kiedra using Goldenberry and Orange.

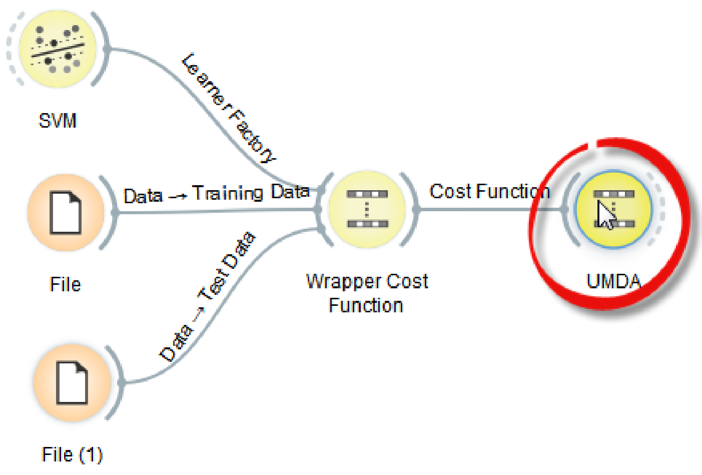
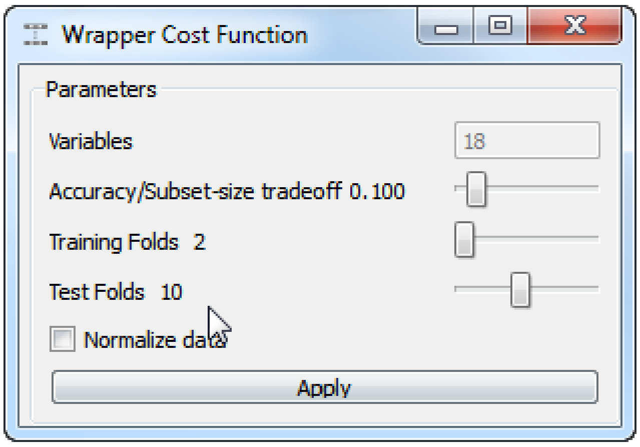


Figure 4.3: How to create `wKiera` using `Goldenberry` and `Orange`. Note that the only difference between `Kiedra` and `wKiera` is the optimizer.

There is a one-to-one correspondence between widgets and `Kiedra`'s building-blocks depicted in Figure 4.1 therefore the explanation given in Section 4.2 is fundamental to understand how the widgets interact with each other. `Kiedra` requires as input: a `Learner Factory`, as its name implies it helps to create learners, in this case the `SVM` widget provides a factory to create `SVM` learners instances, and the `Data` represents the problem's input data and it is provided in this case by two `File` widget, one for training and the other testing. As output `WraperCostFunction` provides a `GbWrapperCostFunction` instance so that it can be used by an optimizer, in this case the a `BMDA` instantiated by the `BMDA` widget. The same scheme works for conducting other `FSS-EDA` methods, for example to build `wKiera` it is only needed to change from a `BMDA` widget to a `UMDA` widget as shown in Figure 4.4.


Wrapper Cost Function

Parameters

Variables 18

Accuracy/Subset-size tradeoff 0.100

Training Folds 2

Test Folds 10

☐ Normalize data

Apply

Figure 4.4: User interface configuration for the `WraperCostFunction`widget.

The `WraperCostFunction` user interface (see Figure 4.4) is used to configure parameters like: *i training and testing folds* for the cross-validation, *ii* the *accuracy/sub-set-size trade-off* to configure the multi-objective balance between the total number of features and the classification accuracy that they produce, and *iii* a *check-box* to control whether data normalization within the real interval of $[0, 1]$ is desired prior to the classification task.

Currently there is no possible to take advantage of all the built-in `Orange` widgets learners, such as the Naive Bayes and the K-Nearest Neighbor, for conducting a `FSS-EDA` . The limitation is because the `Learner Factory`, required by the `WraperCostFunction` widget, is a concept only from `Goldenberry` and not from `Orange`. For `Goldenberry` the `Learner Factory` helps to create independent learners per each candidate when assessing in a parallel execution the fitness function of a set of candidates into the `GbWrapperCostFunction`. This was one of the motivation to create a customized `SVM` widget, extending from `Orange` original, to provide a `SVM` factory instead of a `SVM` instance. The other motivation to create such widget was to allow the `SVM` to use customized kernel functions built by the `KernelBuilder` widget, nonetheless after few experiments we noted a runtime overhead when using custom kernel functions written in python because it is costly to wrap and evaluate them into a `SVM` written in C++ which is indeed the well-known `LIBSVM` `SVM` library. To port the other `Orange`'s learners like the Naive Bayes and the K-Nearest Neighbor a similar strategy can be follow. The result

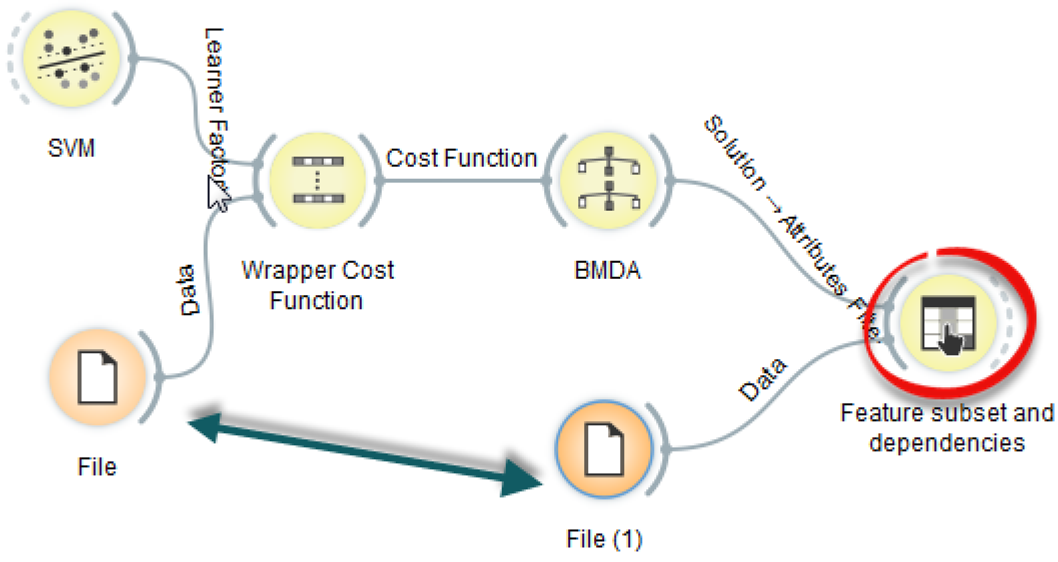
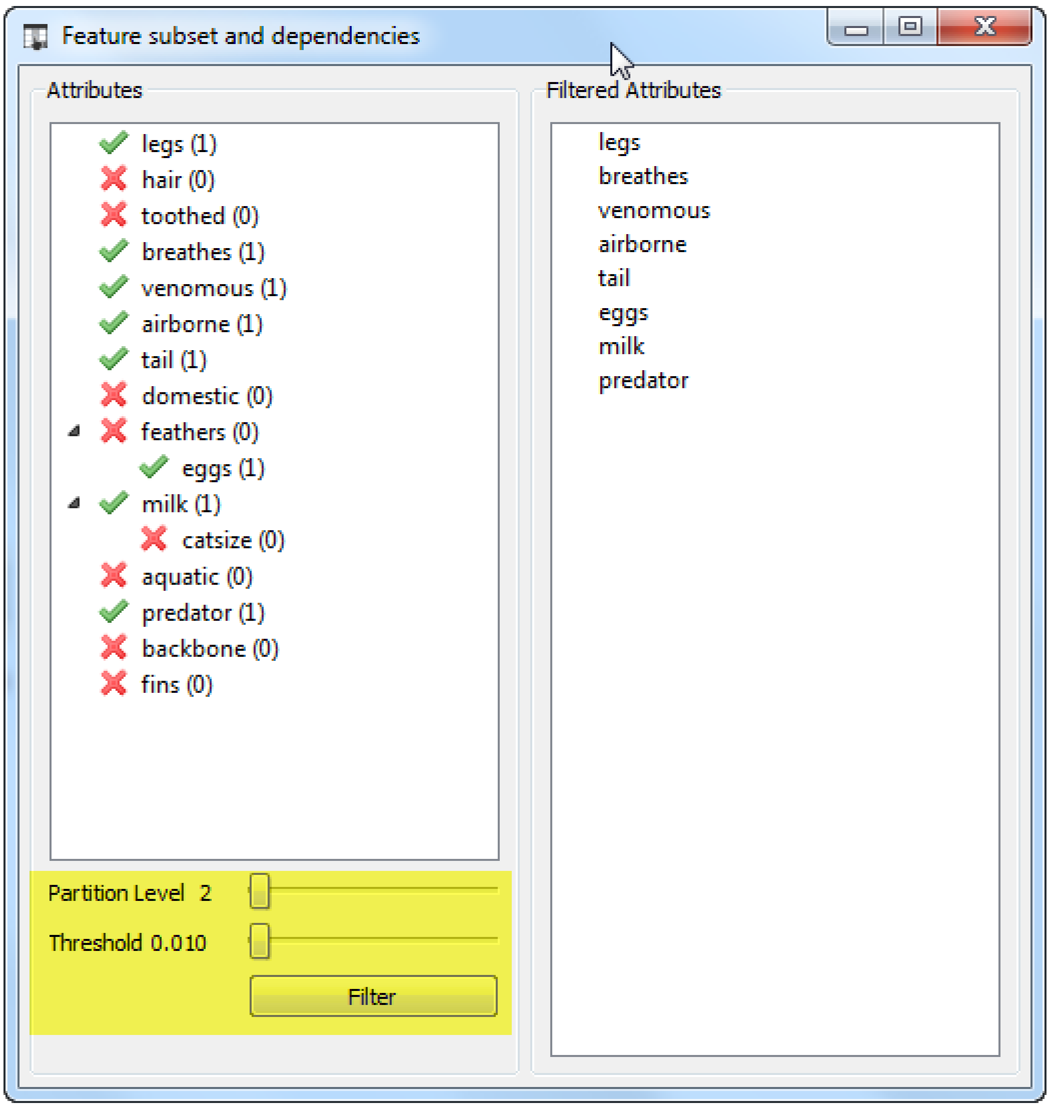


Figure 4.5: How to configure the `FeatureSubset` widget in `Orange` canvas.

of a `FSS-EDA` is an estimation of feature relevances. Those relevances can be used as an input to the `FeatureSubset` widget. This widget is responsible for displaying features estimation scores with their found dependencies and to filter features from the problem's input dataset based on the estimated scores and dependency structure. The Figure 4.4

shows how it can be dragged and wired. `FeatureSubset` takes as an input a dataset and outputs a filtered version based on the relevant scores and some adjustable parameters available by the user interface (see Figure 4.4) such as: *i)* the *partition level* which helps to filter features by their position on the dependency tree and *ii)* a threshold for filtering features by the reported estimation score. Note that widgets expect input data from the same problem's domain used in the feature selection task.



Attributes:
- ✓ legs (1)
- ✗ hair (0)
- ✗ toothed (0)
- ✓ breathes (1)
- ✓ venomous (1)
- ✓ airborne (1)
- ✓ tail (1)
- ✗ domestic (0)
- ✗ feathers (0)
  - ✓ eggs (1)
- ✓ milk (1)
  - ✗ catsize (0)
- ✗ aquatic (0)
- ✓ predator (1)
- ✗ backbone (0)
- ✗ fins (0)

Partition Level 2
Threshold 0.010

Filtered Attributes:
legs
breathes
venomous
airborne
tail
eggs
milk
predator

Figure 4.6: Graphical user interface to visualize and adjust the `FeatureSubset` resulting features.

4.5 Experiments

The experiments aim to measure the capabilities of `Kiedra` to optimize a multi-objective problem with two conflicting objectives: the feature subset size and its effectiveness to generalize the problem domain. `Kiedra` will be tested by an empirical comparison with five real datasets taken from *UCI* repository (see Table 4.5) [30] and two `FSS EDA`: `FSS-EBNA` and `wKiera`. Results for `wKiera` are taken from the tests performed in our platform. For `FSS-EBNA` we will compared based on results published in [24].

Dataset	Features	Classes	Instances
Ionosphere	34	2	351
Soybean	35	19	307
Horse colic	27	24	368
Anneal	38	6	898
Image	19	7	2310

Table 4.1: Experiment datasets properties.

Algorithm execution are reported per each dataset. Each one is preprocessed for filling missing values by a *Naive Bayes Classifier* and normalized within a $[0, 1]$ real interval. The input data is randomly split in two groups *training* and *testing* with equal size. The experiments are executed using `Orange` canvas and `Goldenberry`'s latest release. Each method is repeated 10 times using *training* and *testing* datasets. To configure the experiments, widgets in Tables 4.2 and 4.5 are dragged and wired as depicted in Figure 4.7. Finally, the `BlackBoxTester` widget is executed and results obtained are summarized in Tables (4.5,4.5).

Widget	Purpose	Quantity
`File`	Load input data from a text file.	1
`Preprocessing`	Fill missing values by a *Naive Bayes Classifier.*	1
`Continuize`	Normalize within a $[0, 1]$ real interval the input data.	1
`Data sampler`	Split the input dataset in two groups: *training* and *testing*	1

Table 4.2: `Orange` widgets inventory to perform experiment.

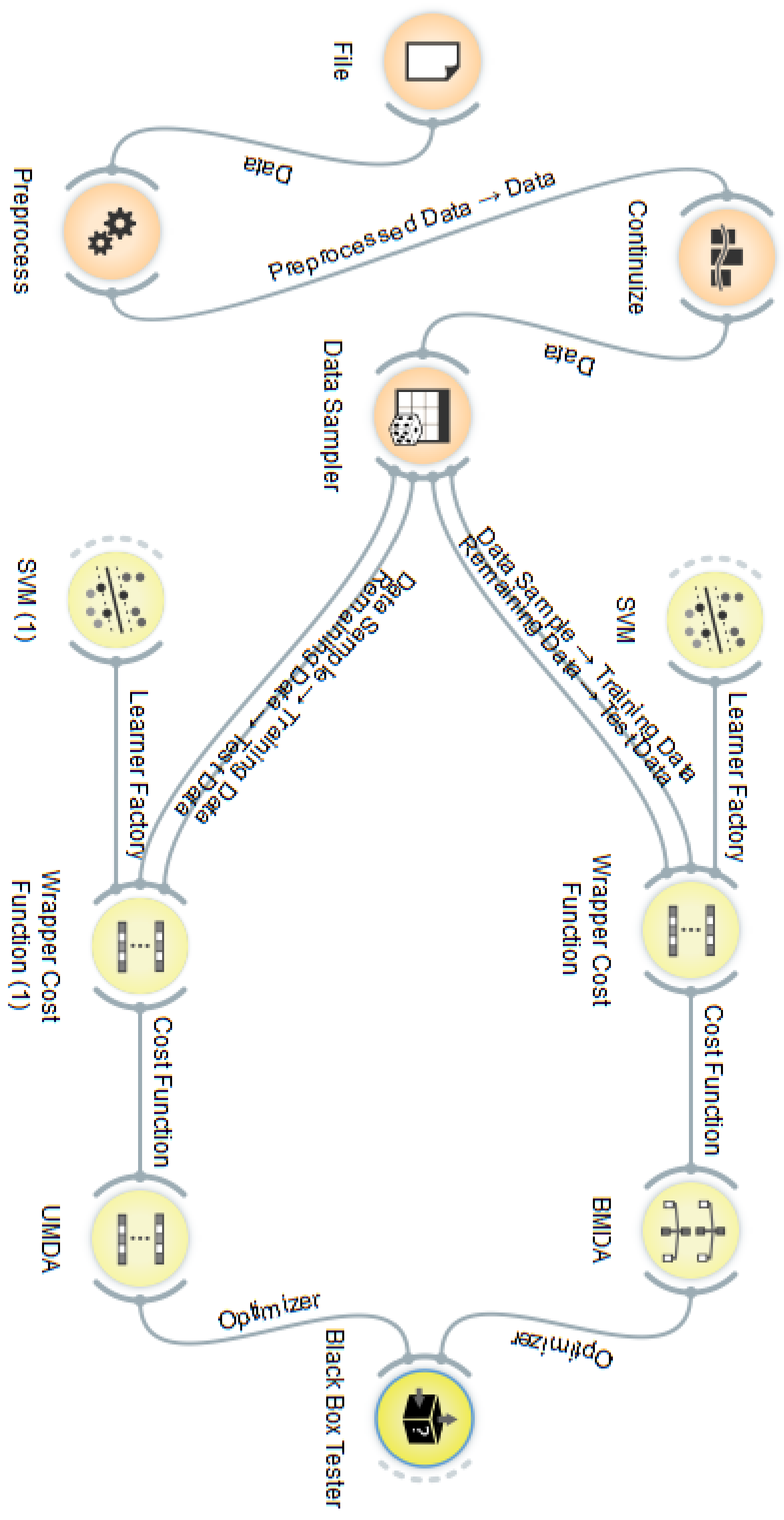


Figure 4.7: Experiments configuration in the `Orange` canvas.

Widget	Purpose	Quantity
`SVM`	Learner used to conduct both `Kiedra` and `wKiera` with a *RBF* kernel. Kernel parameters depends on the nature of each dataset and are automatically estimated using the *Automatic parameters search* functionality.	2
`WraperCostFunction`	This is the cost function for `Kiedra` and `wKiera`, configured with 2 folds for training, 10 folds for testing, a trade-off of 0.1 and with no normalization. For more information please refer to Section 4.2.	2
`BMDA`	The optimizer used for `Kiedra`. It has been configured with a population size of four times the number of variables with χ^2 dependency test method.	1
`UMDA`	The optimizer for `wKiera` . It has been configured with a population size of four times the number of variables.	1
`BlackBoxTester`	Test and collect execution statistics to compare among `FSS` methods, and it has been configured to run 10 repetitions per each experiment, that is per each method and dataset.	1

Table 4.3: `Goldenberry` widgets inventory to perform the experiments.

Datasets	`FSS-EBNA`	`wKiera`	`Kiedra`
Ionosphere	92.40 ± 2.04	98.07 ± 1.36	98.49 ± 0.9
Horse colic	83.93 ± 1.58	90.31 ± 1.94	89.89 ± 2.31
Soybean-large	88.64 ± 1.70	82.31 ± 2.78	81.42 ± 3.81
Anneal	94.10 ± 3.0	76.3 ± 3.76	76.42 ± 4.49
Image	88.98 ± 0.98	89.55 ± 1.28	90.29 ± 1.78

Table 4.4: Prediction accuracy reported by all tested methods per dataset.

Datasets	Without FSS	FSS-EBNA	wKiera	Kiedra
Ionosphere	34	13.40 ± 2.11	7.30 ± 0.82	7.20 ± 0.92
Horse colic	35	6.10 ± 1.85	5.10 ± 0.99	6.50 ± 1.58
Soybean-large	27	18.90 ± 2.76	16.40 ± 1.90	16.60 ± 2.12
Anneal	38	20.50 ± 3.13	9.60 ± 0.97	9.40 ± 1.43
Image	19	8.00 ± 0.66	7.72 ± 1.24	7.45 ± 1.36

Table 4.5: Comparison on the feature subset size.

Dataset	wKiera		Kiedra	
	evaluations	time(s)	evaluations	time(s)
Ionosphere	2893.8 ± 331.59	93.11 ± 30.21	3009.00 ± 838.81	87.62 ± 36.06
Horse colic	1826.20 ± 251.74	43.02 ± 13.19	2258.20 ± 507.71	75.07 ± 25.68
Soybean-large	3501.00 ± 437.77	151.04 ± 38.42	4747.00 ± 1647.87	141.34 ± 48.16
Anneal	1059.40 ± 159.38	87.76 ± 15.42	1160.20 ± 204.92	67.41 ± 14.11
Image	1246.60 ± 299.86	31.14 ± 9.15	1541.80 ± 860.79	44.76 ± 24.40

Table 4.6: Statistics reported only for Kiedra and Kiedra .

4.6 Discussion

In summary Kiedra showed competitive results in terms of prediction accuracy and feature subset sizes. Kiedra slightly overcomes the prediction accuracy from FSS-EBNA in three of the five datasets. Kiedra and wKiera obtained similar accuracies caused probably because they were trained and tested with the same classifier(a SVM) unlike FSS-EBNA that uses a Naive Bayes. Kiedra was also able to reduce considerably the number of features from the input dataset whilst maintaining good generalization capabilities. FSS-EBNA was also able to reduce the feature subset sizes but Kiedra and wKiera reported features size considerably smaller. wKiera showed convergences times and cost function evaluations lesser than Kiedra; it is worth noting that the difference is small considering that Kiedra also estimates variables dependencies while the cost function is evaluated. Unfortunately we could not compare cost function evaluations and execution times against FSS-EBNA because those were not reported.

Chapter 5

Conclusion and future work

In this study we developed a novel `FSS-EDA` method, termed `Kiedra`, that is able to find relevant feature subsets assuming pair-wise dependencies among the domain variables. In preliminary experiments this method found smaller subsets compared to other reported techniques, whilst still maintaining a competitive accuracy performance.

Another interesting result of this study was the development of `TILDA`, a new compact `EDA` algorithm that proved to be able to solve problems of up to 10^7 variables with minimum memory requirements through a coded representation of the search space using the arithmetic-coding technique.

From the point of view of software development, the availability of `Orange` as an open source platform for machine learning and data-mining analysis, enabled the design, implementation and deployment of `Goldenberry`, a suite of components for optimization and machine learning, applying well-known practices of component-based software development and design patterns. In our opinion `Goldenberry` constitutes a successful combination of two related disciplines such as *software engineering* and *computation intelligence.* Originally `Goldenberry` contemplated a predefined set of components to realize `Kiedra`, but along the way its component-based design yielded to the discovery of the `WraperCostFunction`: an emerging pattern to create feature selection *wrapper* methods, and a core component to represent both `Kiedra` and `wKiera` for orchestrating the interaction between the search engine and the learner machine and moreover scheduling their *training* and *testing* phases. It is expected that future projects will complement `Goldenberry` with additional `EDA`s like `MIMIC` [7], `EBNA` [15], `BOA` [35] and `hBOA` [34]. In fact there is an ongoing project for developing genetic algorithms in `Goldenberry` [18].

As future work, it would be interesting to investigate the behavior of `Kiedra` in real-world applications, and validate its effectiveness to discover hidden but existing dependencies among variables. Another interesting avenue for future research will be to determine the runtime overhead of interpreted versus compile source-code for `Goldenberry` core components.

Bibliography

[1] Jiri Adamek and Petr Hnetynka. Perspectives in component-based software engineering. In *Proceedings of the 2008 International Workshop on Software Engineering in East and South Europe*, pages 35–42, New York, NY, USA, 2008. ACM.

[2] Scott W. Ambler. *The Object Primer: Agile Model-Driven Development with UML 2.0.* Cambridge University Press, New York, NY, USA, 2004.

[3] Felix Bachmann, Len Bass, and Robert Nord. Modifiability tactics. Technical Report CMU/SEI-2007-TR-002, Software Engineering Institute, Carnegie Mellon University, 2007.

[4] Shumeet Baluja and R. Caruana. Removing the genetics from the standard genetic algorithm. In *The Proceedings of the 12th Annual Conference on Machine Learning*, pages 38 – 46. Morgan Kaufmann Publishers, 1995.

[5] Shummet Baluja. Population-based incremental learning: A method for integrating genetic search based function optimization and competitive learning. Technical report, Carnegie Mellon University Pittsburgh, PA, USA, Pittsburgh, PA, USA, 1994.

[6] Endika Bengoetxea, Pedro Larrañaga, Isabelle Bloch, and Aymeric Perchant. Estimation of distribution algorithms: A new evolutionary computation approach for graph matching problems. In *Energy Minimization Methods in Computer Vision and Pattern Recognition*, volume 2134 of *Lecture Notes in Computer Science*, pages 454–469. Springer Berlin / Heidelberg, 2001.

[7] Jeremy S. De Bonet, Charles L. Isbell, and Paul Viola. Mimic: Finding optima by estimating probability densities. In *Advances in Neural Information Processing Systems*, page 424. The MIT Press, 1997.

[8] Jean-Michel Bruel and Ileana Ober. Component modeling in uml 2. *Studia Jurnal*, 1:79–90, 206.

[9] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns.* Wiley, volume 1 edition, August 1996.

[10] Olivier Chapelle, Vladimir Vapnik, Olivier Bousquet, and Sayan Mukherjee. Choosing multiple parameters for support vector machines. *Machine Learning*, 46:131–159, 2002. 10.1023/A:1012450327387.

[11] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20:273–297, 1995. 10.1007/BF00994018.

[12] Nello Cristianini and John Shawe-Taylor. *An introduction to support vector machines: and other kernel-based learning methods.* Cambridge University Press, 1 edition, March 2000.

[13] Tomaz Curk, Janez Demsar, Qikai Xu, Gregor Leban, Uros Petrovic, Ivan Bratko, Gad Shaulsky, and Blaz Zupan. Microarray data mining with visual programming. *Bioinformatics*, 21(3):396–398, 2005.

[14] Luc Devroye. *Non-Uniform Random Variate Generation.* Springer-Verlag, 1986.

[15] R. Etxeberria and Pedro Larrañaga. Global Optimization Using Bayesian Networks. In *CIMAF 99, Second Symposium on Artificial Intelligence*, Adaptive Systems, pages 332–339, 1999.

[16] Yoav Freund and Robert E. Schapire. Large margin classification using the perceptron algorithm. *Machine Learning*, 37:277–296, December 1999.

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

[18] Leidy Garzon. Análisis, diseño y construcción de una colección de componentes de software para algoritmos genéticos. District University of Bogota FJC, 2013.

[19] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.

[20] Jörn Grahl and Franz Rothlauf. Polyeda: Combining estimation of distribution algorithms and linear inequality constraints. In *Proceedings of the 6th Annual Conference on Genetic and Evolutionary Computation*, GECCO '04, pages 1174–1185, 2004.

[21] Isabelle Guyon and André Elisseeff. An introduction to variable and feature selection. *Machine Learning*, 3:1157–1182, March 2003.

[22] Georges R. Harik and Fernando G. Lobo. The compact genetic algorithm. *IEEE Transactions on Evolutionary Computation*, 3:523–528, 1999.

[23] C. W. Hsu, C. C. Chang, and C. J. Lin. *A practical guide to support vector classification.* Department of Computer Science, National Taiwan University, 2003.

[24] Iaki Inza, Pedro Larraaga, and Basilio Sierra. Feature subset selection by bayesian networks: a comparison with genetic and sequential algorithms. *International Journal of Approximate Reasoning*, 27(2):143 – 164, 2001.

[25] Sean Luke. *Essentials of Metaheuristics.* Lulu, 2009. Available for free at http://cs.gmu.edu/∼sean/book/metaheuristics/.

[26] Ernesto Mininno, Francesco Cupertino, and David Naso. Real-valued compact genetic algorithms for embedded microcontroller optimization. *IEEE Transactions on Evolutionary Computation*, 12(3):203–219, 2008.

[27] Melanie Mitchell, Stephanie Forrest, and John H. Holland. The royal road for genetic algorithms: Fitness landscapes and GA performance. In *Proceedings of the First European Conference on Artificial Life*, pages 245–254, 1991.

[28] H. Muhlenbein and G. Paag. From recombination of genes to the estimation of distributions: I. binary parameters. In Hans-Michael Voigt, Werner Ebeling, Ingo Rechenberg, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature PPSN IV*, volume 1141 of *Lecture Notes in Computer Science*, pages 178–187. Springer Berlin / Heidelberg, 1996.

[29] Heinz Muhlenbein, Thilo Mahnig, and Alberto Ochoa Rodriguez. Schemata, distributions and graphical models in evolutionary optimization. *Journal of Heuristics*, 5:215–247, July 1999.

[30] P. Murphy and D. Aha. The UCI repository of machine learning databases. http://www.ics.uci.edu/˜mlearn/MLRepository, 1998.

[31] Pedro Larra naga, Ramon Etxeberria, Jose A. Lozano, and Jose M. Pe na. Combinatorial optimization by learning and simulation of bayesian networks. In *Proceedings of the Sixteenth Conference on Uncertainty in Artificial Intelligence*, pages 343–352. Morgan Kaufmann, 2000.

[32] Iñaki Inza, Pedro Larra naga, Ramon Etxeberria, and Basilio Sierra. Feature subset selection by bayesian network-based optimization. *Artificial Intelligence*, pages 157–184, 2000.

[33] Rubn Arma nanzas, Iñaki Inza, Roberto Santana, Yvan Saeys, Jose Luis Flores, Jose Antonio Lozano, Yves Van De Peer, Rosa Blanco, Vctor Robles, Concha Bielza, and Pedro Larra naga. A review of estimation of distribution algorithms in bioinformatics. *BioData Mining*, 2008.

[34] Martin Pelikan and David. E. Goldberg. Hierarchical bayesian optimization algorithm = bayesian optimization algorithm + niching + local structures. pages 525–532. Morgan Kaufmann, 2001.

[35] Martin Pelikan, David E. Goldberg, and Erick Cantu-Paz. Boa: The bayesian optimization algorithm. In *Genetic and Evolutionary Computation Conference (GECCO-1999)*, pages 525–532. Morgan Kaufmann, 1999.

[36] Martin Pelikan and Heinz Müehlenbein. The bivariate marginal distribution algorithm. In Rajkumar Roy, Takeshi Furuhashi, and PravirK. Chawdhry, editors, *Advances in Soft Computing*, pages 521–535. Springer London, 1999.

[37] Martin Pelikan and Kumara Sastry. Initial-population bias in the univariate estimation of distribution algorithm. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, GECCO '09, pages 429–436, New York, NY, USA, 2009. ACM.

[38] Martin Pelikan, Kumara Sastry, and Erick Cantú-Paz, editors. *Scalable Optimization via Probabilistic Modeling: From Algorithms to Applications.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[39] Hanchuan Peng, Fuhui Long, and Chris Ding. Feature selection based on mutual information: criteria of max-dependency, max-relevance, and min-redundancy. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27:1226–1238, 2005.

[40] Claudia Perlich and Saharon Rosset. Identifying bundles of product options using mutual information clustering. In *SDM*, 2007.

[41] Sergio Rojas and Delmiro Fernandez-Reyes. Adapting multiple kernel parameters for support vector machines using genetic algorithms. In *2005 IEEE Congress on Evolutionary Computation (CEC-2005)*, 2005.

[42] Sergio Rojas-Galeano, Emily Hsieh, Dan Agranoff, Sanjeev Krishna, and Delmiro Fernandez-Reyes. Estimation of relevant variables on high-dimensional biological patterns using iterated weighted kernel functions. *PLoS ONE*, 3(3), 2008.

[43] Sergio Rojas-Galeano and Nestor Rodriguez. A memory efficient and continuous-valued compact EDA for large scale problems. In *Proceedings of GECCO 2012*, pages 281–288, NY, USA, 2012. ACM.

[44] Sergio Rojas-Galeano and Nestor Rodriguez. Goldenberry: Eda visual programming in orange. In *Proceedings of GECCO 2013*, pages 281–288, NY, USA, 2013. ACM.

[45] Frank Rosenblatt. *The perceptron: a probabilistic model for information storage and organization in the brain*, pages 386–408. American Psychological Association, 1956.

[46] Yvan Saeys, Iñaki Inza, and Pedro Larrañaga. A review of feature selection techniques in bioinformatics. *Bioinformatics*, 23:2507–2517, September 2007.

[47] Amir Said. Introduction to arithmetic coding - theory and practice. Technical Report HPL-2004-76, Imaging Systems Laboratory, HP Laboratories Palo Alto, 2004.

[48] Kumara Sastry, David E. Goldberg, and Xavier Llora. Towards billion-bit optimization via a parallel estimation of distribution algorithm. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, GECCO '07, pages 577–584, 2007.

[49] Kumara Sastry, David E. Goldberg, and Xavier Llora. Towards billion bit optimization via parallel estimation of distribution algorithm. In *Genetic and Evolutionary Computation Conference (GECCO-2007)*, pages 577–584, 2007.

[50] John Shawe-Taylor and Nello Cristianini. *Kernel Methods for Pattern Analysis.* Cambridge University Press, New York, NY, USA, 2004.

[51] Worasait Suwannik and Prabhas Chongstitvatana. Solving one-billion-bit noisy onemax problem using estimation distribution algorithm with arithmetic coding. In *IEEE Congress on Evolutionary Computation*, pages 1203–1206, 2008.

[52] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[53] Xiuping Wu and Murray Woodside. Performance modeling from software components. *SIGSOFT Softw. Eng. Notes*, 29(1):290–301, January 2004.

Appendix A

Background

In this section we provide a summarized review of the main elements involved in this proposal, namely feature selection techniques, kernel classification machines, and estimation of distribution algorithms.

A.1 Feature selection techniques

A.1.1 The problem of feature selection

Nowadays the advances in technologies for data collection (the genome project, particle colliders, internet-based social networks) pose increased challenges for data analysis due to larger sizes and higher dimensionality of the observed samples. It is reasonable however to assume that the collected variables may exhibit redundancy, inconsistency, noisy and irrelevant data and therefore will be difficult to analyze by the human eye. New automated mechanisms are required to identify significant variables for pattern discovery and data mining.

Feature selection is gaining in importance and has recently become an active field of research in disciplines such as knowledge discovery, machine learning, pattern recognition, bioinformatics, geoinformatics, etc. While FSS are techniques for dimensionality reduction, they maintain the original variables compared to other entropy-based, data compression or statistical methods that modify the original data representation instead of providing a subset of significant features with useful information for the domain experts [46]. The main goal of these techniques is to obtain a better understanding of the data for visualization purposes or to speed up data analysis.

FSS techniques are helpful in improving prediction models for supervised learning, detecting dimensions of tight data conglomeration in clustering tasks and a deeper understanding of process for data generation. Nonetheless, the design of novel FSS techniques aimed to provide more useful information also incurs in new levels of complexity giving rise to new challenges pertaining to feasible and efficient computational techniques.

In the classification context, feature selection techniques are categorized in three main groups depending on how they interact with a classification model (see Figure A.1): filters, wrappers and embedded. A detailed description of this categorization is given below (based on the study in [33]).

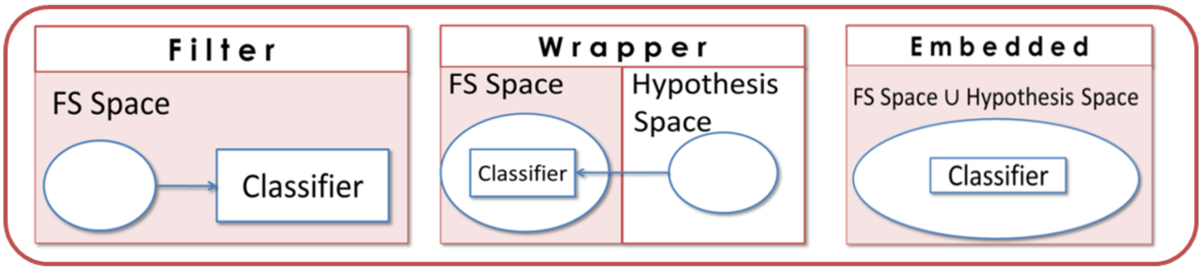


Figure A.1: A feature selection taxonomy. The feature selector category depends on how the feature and hypothesis space are combined during classifier learning (left and right box taken from [46]).

A.1.2 Filter approach

Methods in this category assess the relevance of variables based on the intrinsic properties of data. The search in feature space is separated from the search in the space of classification hypothesis. In most cases a feature relevance score is calculated (i.e. using mutual-information or data correlation) and low-scoring features are removed. These methods have the ability to scale to higher dimensions since they exhibit a low computational cost.

A.1.3 Wrapper approach

Wrappers establish a symbiotic relationship between the feature subset search and the classification algorithm. The aim of these type of methods is not only to find relevant features but also to find the best suited classification model to those features. However the interaction with the classifier may induce new problems such as a higher risk of overfitting and a higher demand of computational resources.

A.1.4 Embedded approach

Embedded methods incorporate the feature selection and the classification model construction as a single process. As a result feature and hypothesis spaces are combined providing a richer source of information during the search although incurring as well in additional computational costs.

A.2 Machine learning

Supervised learning aims to find optimal learning algorithms with high generalization and prediction performance to produce classification models of the training set of examples and their associated labels. The prediction accuracy of the resulting classifier is then evaluated with a test set. Among these techniques, linear classifiers are widely used because of their theoretically simplicity and computational efficiency. Application of linear classifiers, in real world problems where nonlinearities arise, has been possible due to the advances in kernel machines [50].

A.2.1 Kernel machines

These algorithms use kernel functions to compute similarity measures of the input samples in a feature space where nonlinearities might be easier to solve by linear classifiers. Figure A.2 shows an example of feature mapping where in the original space data can only be separated with a nonlinear function but becomes linearly separable if the data is transformed into a feature space. When using the linear classifier the feature space mapping is implicit since the machine only uses inner products of the transformed examples which are computed using kernel function.

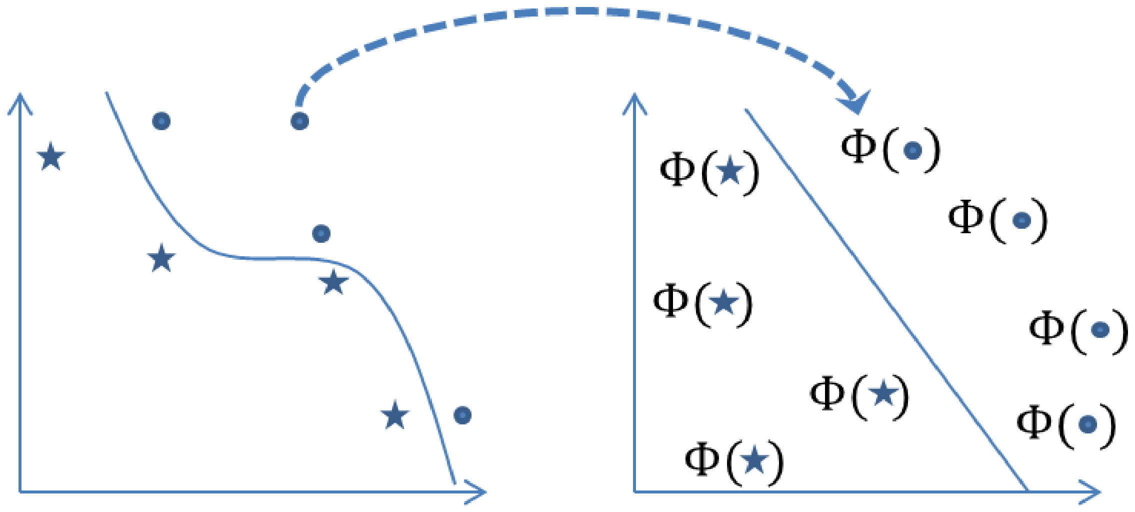


Figure A.2: Transformation from input space to a feature space simplifies the classification task (taken from [50]).

A kernel is a function K such that for all $\mathbf{x}, \mathbf{z} \in X$

$$K(\mathbf{x}, \mathbf{z}) = \langle \phi(\mathbf{x}), \phi(\mathbf{z}) \rangle,$$

where ϕ is a mapping from the input space X to an (inner product) feature space. In order to be a kernel, the function must comply with the conditions defined by Mercer's theorem [50]. Eqs.(A.1,A.2) are examples of kernel functions.

$$K_\sigma(\mathbf{x},\mathbf{z}) = \exp\left(-\sigma\sum_{i=1}^{\ell}(x_i - z_i)^2\right) \tag{A.1}$$

and

$$K_d(\mathbf{x},\mathbf{z}) = \langle \mathbf{x},\mathbf{z}\rangle^d, \tag{A.2}$$

A.2.2 Linear learning machines

A linear classifier is a linear function $f(\mathbf{x})$ where $\mathbf{x} \in X$ and $\mathbf{w} \in \mathbb{R}^\ell$, that can be written as:

$$f(\mathbf{x}) = \operatorname{sgn}(\langle \mathbf{w},\mathbf{x}\rangle) = \operatorname{sgn}\Big(\sum_i w_i x_i\Big) \tag{A.3}$$

A given sample $\mathbf{x}$ is assigned to the positive class if $f(\mathbf{x}) = 0$ or otherwise to the negative class. There are number of training algorithms to learn a classification vector $\mathbf{w}$, including the perceptron [45] and the linear support vector machine [12]. For illustration purposes, Algorithm 5 shows the learning procedure of the perceptron.

The kernelized version of the linear classifier allows for classification of nonlinearly separable datasets, as mentioned before. The classification function is written consequently as:

$$f(\mathbf{x}) = \operatorname{sgn}\Big(\sum_k \alpha_k K(\mathbf{x}_k,\mathbf{x})\Big), \tag{A.4}$$

where K represents a kernel function and α_k the classifier parameters that can be learned in this case using the kernel perceptron [16] or the support vector machine [11,12].

Algorithm 5 The Perceptron

```
Requires: Given a linearly separable training set $S$ and learning rate $\eta \in \mathbb{R}^+$
1: $w \leftarrow 0$; $k \leftarrow 0$
2: repeat
3:    repeat  $i = 1, 2, \dots \ell$
4:       if $y_i(\langle w_k, x_i\rangle) \leq 0$ then
5:          $w_{k+1} \leftarrow w_k + \eta y_i x_i$
6:          $k \leftarrow k + 1$
7:       end if
8:    end repeat
9: until no mistake made within the repeat loop in Step(3)
Outputs: $(w_k)$ where $k$ is the number of mistakes
```

A.3 Estimation of distribution algorithms EDAs

Genetic algorithms (GA) are search stochastic methods inspired in the theory of natural selection of Darwin. The idea is to evolve a population of candidates coding the parameters for the solution of an optimization problem, using genetic operations such as chromosome recombination and mutation [19]. A novel technique known as Estimation of Distribution Algorithms (EDAs) has recently emerged motivated by GAs but from a statistical viewpoint. They have proven to be better suited in many applications than canonical GAs [6].

The main distinctive aspect of EDAs is that they search for a probabilistic distribution model representing the population of candidates. Instead of using genetic operations, these algorithms are based on well-known statistical techniques to estimate the parameters of a distribution function, and the evolution is guided by sampling the evolving probabilistic model. The complexity of the algorithm lies in the robustness of this probability model and in how it is iteratively re-estimated. The probabilistic models can be as simple as a marginal distribution or as complex as a joint multivariate distribution expressing high order interactions among the observed variables. These categories are briefly described below following the review in [33].

A.3.1 Univariate EDAs

Univariate algorithms use a marginal probability model encoded in a probability vector. They are not computational intensive because they assume no interaction between variables. The probability vector is re-estimated using the fittest subset of the population of candidates. There are three widely used algorithms for this category: Univariate Marginal Distribution Algorithm (`UMDA` [28]), Popupation-based Incremental Learning (`PBIL` [5]) and the Compact Genetic Algorithm (`cGA` [4,22]).

`UMDA` and `PBIL` estimates the entire probability vector every iteration. The probability distribution is factorized as a product of independent univariate marginal distributions, which are estimated from marginal frequencies. `PBIL` unlike `UMDA` uses a learning rate α to control early convergences for the probability model parameters. `PBIL` becomes `UMDA` when $\alpha = 1.0$. `PBIL` is depicted in Algorithm 6.

Lastly, cGA has become popular for the higher efficiency to solve very large scale problems with millions to billions of variables with a lower computational demand than canonic GA [49]. cGA has a low memory consumption because only two candidates per iteration are generated. `cGA` is depicted in Algorithm 6.

A.3.2 Multivariate EDAs

Multivariate `EDA`s use joint statistics models of higher order to represent interaction among variables. These models are usually represented as probabilistic graphical models (see Figure A.3). As mentioned above, `EDA`'s complexity increases when a higher order of interaction among variables are desired. Factorized Distribution Algorithm

Algorithm 6 PBIL pseudo-code

```
Requires: Candidate size $n$, the number variables $\ell$, learning rate $\alpha$ and the cost function $f(\cdot)$.
1: $\boldsymbol{\rho} \leftarrow \mathsf{initialize}(\ell)$
2: repeat
3:     $\mathcal{D} \leftarrow \mathsf{sample}(P(X;\boldsymbol{\rho}), n)$
4:     $\mathcal{C} \leftarrow \mathsf{select}(\mathcal{D}, f(\cdot))$
5:     $\boldsymbol{\rho} \leftarrow \mathsf{estimate}(\boldsymbol{\rho}, \mathcal{C}, \alpha)$
6: until $P(X;\theta)$ has converged
Outputs: Probability distribution $P(X;\boldsymbol{\rho})$
```

Algorithm 7 cGA pseudo-code

```
Requires: Candidate size $n$, the number variables $\ell$ and the cost function $f(\cdot)$.
1: $\boldsymbol{\rho} \leftarrow \mathsf{initialize}(\ell)$
2: repeat
3:     $\{\mathbf{x}_1, \mathbf{x}_2\} \leftarrow \mathsf{sample}(P(X;\boldsymbol{\rho}), 2)$
4:     $\{\mathbf{x}_\top, \mathbf{x}_\perp\} \leftarrow \mathsf{select}(\{\mathbf{x}_1, \mathbf{x}_2\}, f(\cdot))$
5:     $\boldsymbol{\rho} \leftarrow \mathsf{estimate}(\boldsymbol{\rho}, \{\mathbf{x}_\top, \mathbf{x}_\perp\})$
6: until $P(X;\boldsymbol{\rho})$ has converged
Outputs: Probability distribution $P(X;\boldsymbol{\rho})$
```

(FDA [29]), Estimation of Bayesian Networks Algorithm (EBNA [31]) and Bayesian optimization algorithm (BOA [35]) belong to this category. BOA and EBNA both use Bayesian network structures but differs in the score metric they use to select the appropriate network structure. BOA uses Bayesian Dirichlet equivalence score (BDe) while text uses K2+Penalization and Bayesian Information Criterion (BIC). Similarly, FDA uses Boltzmann selection for Boltzman distribution.

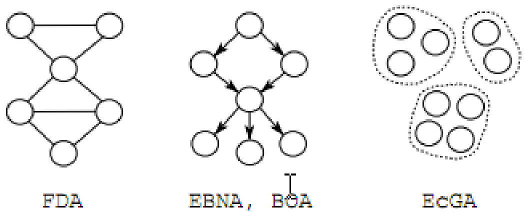


Figure A.3: Probability models used in most popular multivariate EDAs(taken from [33]).

Appendix B

Published work

B.1 Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference (GECCO), pages 281-288, 2012.

A Memory Efficient and Continuous-valued Compact EDA for Large Scale Problems

Sergio Rojas-Galeano
Engineering School
District University of Bogota
Bogota, Colombia
srojas@udistrital.edu.co

Nestor Rodriguez
Engineering School
District University of Bogota
Bogota, Colombia
nearodriguezg@correo.udistrital.edu.co

ABSTRACT

This paper considers large-scale `OneMax` and `RoyalRoad` optimization problems with up to 10^7 binary variables within a *compact* Estimation of Distribution Algorithms (`EDA`) framework. Building upon the compact Genetic Algorithm (`cGA`), the continuous domain Population-Based Incremental Learning algorithm ($\mathtt{PBIL}_c$) and the arithmetic-coding `EDA`, we define a novel method that is able to compactly solve regular and noisy versions of these problems with minimal memory requirements, regardless of problem or population size. This feature allows the algorithm to be run in a conventional desktop machine. Issues regarding probability model sampling, arbitrary precision of the arithmetic-coding decompressing scheme, incremental fitness function evaluation and updating rules for compact learning, are presented and discussed.

Categories and Subject Descriptors

I.2.8 [**Computing Methodologies**]: Artificial Intelligence—*Problem Solving, Control Methods and Search*

General Terms

Algorithms

Keywords

EDA, compact GA, large scale optimization, arithmetic coding

1. INTRODUCTION

Estimation of Distribution Algorithms (`EDA`) [11, 16] approximately optimizes a cost function by building a probabilistic model of a pool of promising sub-optimal solutions over a given search space. For very-high dimensional search spaces, storing and updating a large population of candidates may imply a computational burden in both time and memory. The *compact* approach circumvents storage limitations by incrementally updating the probability model using just two candidates at any step of the algorithm, instead of the entire population. This feature makes the compact `EDA` framework practical for large-scale optimization, a soon-to-be commonplace setting in scientific domains such as bioinformatics, particle physics, chemical crystallography, or social network analysis, to name a few.

This study reports on a new efficient, continuous-valued, compact `EDA` aimed at optimization of large-scale problems of up to 10^7 binary decision variables, nevertheless requiring low-cost computational resources. The method is tailored to optimization of decomposable cost functions in terms of the contribution of individual variables, that is, to problems where incremental fitness evaluation is feasible. Although at present no second- or higher-order dependencies within the variables are considered in our method, its value as an early-stage data analysis tool in high dimensional spaces might be relevant, for example to carry out feature selection for subsequent dependencies estimation on smaller variable subsets, as other models with independence assumptions have shown [17]. Even so, we anticipate modeling multivariate dependencies will widen the applicability of our method. In this respect, incorporating memory and time efficiencies such as fitness evaluation relaxation [19] would be an interesting avenue of future work.

Our findings are organized as follows: a short review of methods related to our research is presented in section 2; a description of the techniques involved in the design of the algorithm is given in section 3; the algorithm itself is discussed in section 4; empirical arguments are provided in section 5 and closing remarks are highlighted in section 6.

Notation. Lower-case boldface is used to denote vector arrays (e.g. $\boldsymbol{\mu} \in \mathbb{R}^2, \mathbf{s} \in \mathbb{R}^\ell$) whereas plain font denotes scalar values (e.g. $\gamma, p, q \in \mathbb{R}$). Capital letters refer to sets of elements. We use n to denote the size of a given population, and ℓ represents the size of the search problem, that is, its dimensionality.

2. LITERATURE REVIEW

The compact Genetic Algorithm (`cGA`) [8] was proposed as a memory-efficient variant of the canonical genetic algorithm [5]. The idea behind the `cGA` is to consider the population of candidates as a sample from which a probability model of Bernoulli trials can be estimated. The algorithm performs successive binary tournaments of two sampled candidates from the current distribution and also updates the model parameters with a frequentist rule based on only those two candidates; thus it avoids the need to keep the entire population in memory. As a result, the `cGA` reduces the storing capacity needed to evolve the population from $O(n\ell)$ to $O(\ell(\log_2 n + 2))$. This property enabled the algorithm to solve problems with dimensionality up to one billion binary variables, as subsequently reported in [20]. In such problems ($\ell = 10^9, \log_2 \ell \approx 30$), the memory size required for running the algorithm would roughly reach $(10^9 \times 32)$bits $\approx 3,72$Gb, yet a stringent amount for a conventional machine; in fact, some efficacies including distributed computing, vectorized and integer operations, and tailored random number generation were necessary to solve large scale problems with a `cGA` running in a 256-processor cluster machine [20].

`cGA` works in discrete domains where variables take binary values. Other non-compact `EDA`s have been proposed to address optimization in both discrete [2,14] and continuous domains [6, 21]. Particularly, continuous-domain Population-Based Incremental Learning ($\mathtt{PBIL}_c$) [21] is an algorithm that assumes variables are independently normally distributed. The mean and variance parameters are updated using the maximum likelihood marginal estimates from a sample population. Since the population is used at every iteration of the algorithm, $\mathtt{PBIL}_c$ is more demanding in terms of memory when compared to `cGA`, and would be difficult to apply to large scale problems in conventional machines.

It is also worth mentioning a recent study [22] where an encoding technique known as arithmetic-coding [18], was used within a continuous domain population-based `EDA` to solve large scale problems of up to one billion variables. The encoding technique allows compressing a binary string of arbitrary length using a couple of double precision numbers. Although the fitness evaluation still requires $\ell/8$ bytes of memory per active candidate, the compression scheme enabled the algorithm to solve the problem with a population size of 3200 candidates in a regular PC-computer in about 36 hours. Regrettably, the study did not provide details on the mechanics of the algorithm, motivation of the updating rules of the distribution parameters, or issues related to the precision of arithmetic-coding.

In this paper we build upon the above algorithms and propose a novel method to solve canonical and noisy versions of large-scale `OneMax` and `RoyalRoad` problems by using a compact and continuously-valued representation, which is further enhanced with an arithmetic-coding scheme. For this kind of additively-separated tasks and with the assumption of independently-distributed binary variables, the new method demands minimal memory consumption ($O(1)$), regardless of the problem dimension or population size. The insights are reported in the following sections.

3. TOOLS

3.1 TILDA

The basis of this study is an algorithm we devised as a mix of `cGA` and $\mathtt{PBIL}_c$. The underlying idea is to model the pool of solution candidates with a multivariate marginal joint Gaussian distribution whose parameters are estimated within a compact framework. The estimates of marginal parameters $(\mu_i, \sigma_i)_{i=1}^{\ell}$ for both the mean and standard deviation of each input variable, are periodically updated with information from the fittest candidates selected by means of 2-way tournaments, as explained next[1]. Let us assume we have a population P of $2n$ candidates were n simple 2-way tournaments are performed (two candidates from P are randomly chosen without replacement, and the the winner substitutes the loser). Let W and L be the sets of tournament's winners and losers, respectively. The new population is then $\widetilde{P} = (P \setminus L) \cup W$. Then, the maximum likelihood mean estimate of the new population, that is $\boldsymbol{\mu} = (\mu_1, \ldots, \mu_\ell)$, can be computed in vectorized form as:

$$\begin{aligned}\boldsymbol{\mu} &= \frac{1}{2n}\sum_{\mathbf{x}_k \in \widetilde{P}} \mathbf{x}_k = \frac{1}{2n}\left(\sum_{\mathbf{x}_k \in P} \mathbf{x}_k - \sum_{\mathbf{x}_k \in L} \mathbf{x}_k + \sum_{\mathbf{x}_k \in W} \mathbf{x}_k\right) \\ &= \frac{1}{n}\sum_{\mathbf{x}_k \in W} \mathbf{x}_k,\end{aligned}$$

since $P = W \cup L$. Now, assuming that variables are mutually independent, and defining the unordered set of winner indexes as $\mathcal{W} = \{k_t : t = 1, \ldots, n;\ \mathbf{x}_{k_t} \in W\}$, the estimate can be re-casted in terms of its components as:

$$\mu_i = \frac{1}{n}\sum_{t=1}^{n} x_i^{k_t},$$

where $x_i^{k_t}$ is the i-th component of the candidate $\mathbf{x}_{k_t}$. We observe that if we initialise $\mu_i = 0$, then the estimate can be updated incrementally with a fraction of each tournament's winner, as the formula in Equation (1) indicates.

$$\mu_i := \mu_i + \tfrac{1}{n} x_i^{k_t}; \quad t = 1, \ldots, n. \tag{1}$$

Now, the same rationale can be applied to the variance estimate $\boldsymbol{\Sigma} = (\sigma_1^2, \ldots, \sigma_\ell^2)$:

$$\begin{aligned}\sigma_i^2 &= \frac{1}{n}\sum_{t=1}^{n}(x_i^{k_t} - \mu_i)^2 = \frac{1}{n}\left(\sum_{t=1}^{n}(x_i^{k_t})^2 - 2\mu_i \sum_{t=1}^{n} x_i^{k_t} + n\mu_i^2\right) \\ &= \frac{1}{n}\sum_{t=1}^{n}(x_i^{k_t})^2 - \mu_i^2.\end{aligned}$$

Then again, an incremental update can be obtained by first setting a temporal variable $\tilde{\sigma}_i^2 = 0$, then accumulating deviations of each tournament's winner in this variable using Equation (2), and finally subtracting the square of the previously computed mean estimate to obtain the variance estimate, as in Equation (3).

$$\begin{aligned}\tilde{\sigma}_i^2 &:= \tilde{\sigma}_i^2 + \tfrac{1}{n}(x_i^{k_t})^2; \quad t = 1, \ldots, n. && (2) \\ \sigma_i^2 &= \tilde{\sigma}_i^2 - \mu_i^2 && (3)\end{aligned}$$

Equations (1-3) outline the compact updating rules of the new method, named the Tiny Incremental Learning Density Estimation Algorithm (`TILDA`) – see Algorithm 1.

The aim of the algorithm is to approximately find a solution to the unconstrained optimisation problem:

$$\boldsymbol{\beta}^\star = \underset{\mathbf{x} \in \mathbb{R}^\ell}{\mathsf{argmax}}\, f(\mathbf{x}),$$

where $f(\cdot)$ is the fitness or cost function to be optimised. The algorithm models a population of promising solutions using a multivariate marginal joint Gaussian distribution with parameters $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$. The main loop (line 2) runs for a number of generations or until the parameters converge. The inner loop (line 4) simulates the breeding of one population by sampling two candidates at a time (line 5), carrying out 2-way tournaments to determine the winner $\hat{\mathbf{x}}$ (line 6), and also keeping temporary parameter estimates for the current

[1]A similar reasoning was followed in [12, Appendix A].

Algorithm 1 TILDA

```
Requires: $\ell > 0$, fitness function $f(\cdot)$
Outputs: best candidate $\boldsymbol{\beta}$
 1: $\boldsymbol{\mu} = \mathbf{0}, \boldsymbol{\Sigma} = \mathbf{1}, \boldsymbol{\beta} = \mathbf{0}, \gamma = 0.5, \epsilon = 0.01$
 2: repeat until ending citeria not met
 3:     $\tilde{\boldsymbol{\mu}} = \tilde{\boldsymbol{\Sigma}} = \mathbf{0}$
 4:     repeat $n$ times
 5:         $\{\mathbf{x}', \mathbf{x}''\} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$
 6:         $\hat{\mathbf{x}} = \mathsf{argmax}(f(\mathbf{x}'), f(\mathbf{x}''))$
 7:         $\boldsymbol{\beta} = \mathsf{argmax}(f(\boldsymbol{\beta}), f(\hat{\mathbf{x}}))$
 8:         $\tilde{\boldsymbol{\mu}} = \tilde{\boldsymbol{\mu}} + \frac{1}{n}(\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_\ell)$
 9:         $\tilde{\boldsymbol{\Sigma}} = \tilde{\boldsymbol{\Sigma}} + \frac{1}{n}(\hat{x}_1^2, \hat{x}_2^2, \ldots, \hat{x}_\ell^2)$
10:     $\boldsymbol{\mu} = \boldsymbol{\mu} - \gamma(\boldsymbol{\mu} - \frac{1}{2}(\boldsymbol{\beta} + \tilde{\boldsymbol{\mu}}))$
11:     $\boldsymbol{\Sigma} = \boldsymbol{\Sigma} - \gamma(\boldsymbol{\Sigma} - (\tilde{\boldsymbol{\Sigma}} - (\tilde{\mu}_0^2, \tilde{\mu}_1^2, \ldots, \tilde{\mu}_\ell^2)) + \epsilon$
```

population (lines 8-9) using Equations (1-2). Notice that the algorithm stores the best solution found up to a certain generation in the variable $\boldsymbol{\beta}$ (line 7). The actual updates of the overall estimates are performed at the end of the main loop (lines 10-11). The overall mean estimate is computed with a memory-preserving rule with decay factor γ, incorporating an elitist mechanism that shifts the estimate towards the average of the population mean and the best candidate. Similarly, the overall variance estimate uses the population mean, as stated in Equation (3), plus a small constant intended to prevent premature convergence[2]. When the ending criteria is met, the approximate optimal solution found by the algorithm can be retrieved from $\boldsymbol{\beta}$.

3.2 Arithmetic coding of binary strings

Given the alphabet $S = \{0, 1\}$ and a probability distribution $\Phi = \{P(0)=p, P(1)=1-p\}$, the idea of arithmetic coding is to represent an arbitrary binary string of length ℓ using distribution Φ and a single real number $q \in [0, 1]$ known as a *code value* (see details in [18]). Decoding is achieved by performing a Bernoulli trial for each bit in the sequence, that is, flipping an unfair coin with bias p, where q represents the chance of a face-up outcome. For this purpose, the decoder builds nested intervals and biases $\{[a_i, b_i], p_i\}_{i=1}^{\ell}$ that re-define the support and probability distributions throughout consecutive bits along the sequence. Thus, a decoded string $\mathbf{s}_p^q$ corresponds to $\mathbf{s}_p^q = (s_i^q)_{i=1}^{\ell} = (s_1^q, \ldots, s_\ell^q)$, where the outcome for each bit i is determined by the indicator function:

$$s_i^q = \begin{cases} 0 & q \leq p_i \\ 1 & q > p_i \end{cases}, \; i \geq 1, \tag{4}$$

and the bias p_i is updated iteratively according to the outcome of the preceding bit and the bounds a_i, b_i of the nested support intervals, following the recursions:

$$\begin{aligned} [a_1, b_1] &= [0, 1]\,, \\ [a_i, b_i] &= \begin{cases} [a_{i-1}, p_{i-1}] & q \leq p_{i-1} \\ [p_{i-1}, b_{i-1}] & q > p_{i-1} \end{cases}, \; i > 1, \\ p_i &= a_i + (b_i - a_i)p\,, \; i \geq 1. \end{aligned} \tag{5}$$

Equation (5) implies that the support interval for a subsequent trial is replaced by the subinterval chosen from the comparison of the code value and the bias of the current trial. As a result, the length of nested intervals shrinks away as the decoding progresses, with the values of these variables growing in precision digits. In theory, one may obtain an arbitrary number of bits with this decoding scheme. In practice, however, the number of bits is limited by machine-dependent floating point arithmetic, in other words, due to round-off errors the supporting interval (and the bias p_i too) eventually collapses and yields the code value q, resulting in a constant zero-output from that trial onwards.

In an attempt to circumvent this shrinking-interval limitation, we contemplated an alternative decoding scheme, as explained next. Firstly, observe that during the shrinking procedure the nested probability distributions obtained with the successive updates of $[a_i, b_i]$ and p_i, preserve the proportions of the original distribution defined by p in Φ. In fact, these are just shrunk replica distributions repeatedly shifted to different locations along the unit interval. Within these distributions, the *global* (fixed) code value q defines a *local* chance of outcome that is relative to the location of the supporting interval and bias of the respective trial. Bearing in mind this observation, we sought to switch roles and keep distribution Φ *global*, that is, fixed to the unit interval, whereas the otherwise constant code value is repeatedly shifted to mirror the *local* chance of outcome at each trial of the shrinking procedure. In this way, for all i we fixed $[a_i = 0, b_i = 1]$ and $p_i = p$ and rescaled the value of the now variable q_i using the following recursive rules:

$$\begin{aligned} q_1 &= q\,, \\ q_i &= \begin{cases} \dfrac{q_{i-1}}{p} & q_{i-1} \leq p \\ \dfrac{(q_{i-1} - p)}{(1 - p)} & q_{i-1} > p \end{cases}, \; i > 1 \end{aligned} \tag{6}$$

In this new scheme, the values of q_i shift around the unit interval resembling the cursor of an old-fashion sliding rule, thus the name *sliding-pin* arithmetic decoding. Since the length and bias of the supporting interval are in a normal scale, the relocated values of q_i are not machine-precision dependent. Additionally, it is worth noting that the sliding-pin procedure only maintains one parameter (q_i) as opposed to three parameters (a_i, b_i, p_i) for the shrinking decoding. The decoded string $\mathbf{s}_p^q$ would be now obtained as:

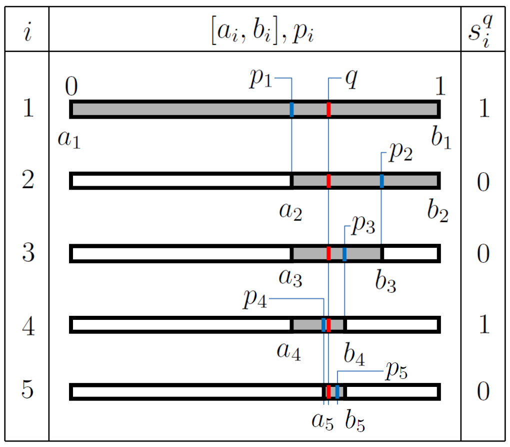
$$\mathbf{s}_p^q = (s_i^p)_{i=1}^{\ell} = (s_1^p, \ldots, s_\ell^p), \tag{7}$$

where the indicator function becomes:

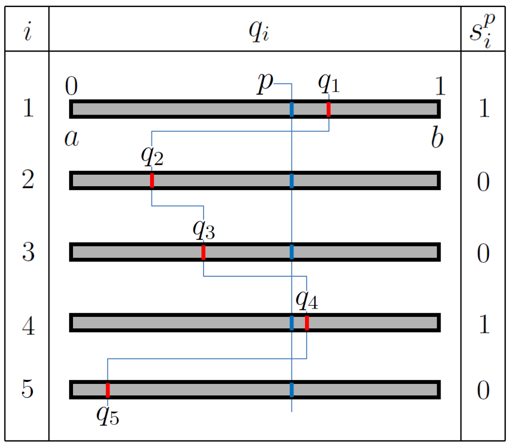
$$s_i^p = \begin{cases} 0 & q_i \leq p \\ 1 & q_i > p \end{cases}, \; i \geq 1. \tag{8}$$

The two decompression schemes are illustrated in Figure 1.

[2]It is interesting to remark that more sophisticated mechanisms to prevent premature convergence such as adaptive variance scaling from EDA and Evolutionary Strategies literature (e.g. [1, 3, 10]), are worthy of further study in future extensions.



(a)



(b)

Figure 1: Illustration of the operation of two arithmetic-decoding techniques for a string of length $\ell = 5$. The leftmost column i of each illustration indicates the decoding step. The corresponding output bit for each step is depicted in the rightmost column. (a) *Shrinking-interval* decoding [18]: The values of $\{[a_i, b_i], p_i\}_{i=1}^5$ are updated recursively using Equations (5) for a given fixed q whilst the output bit s_i^q is obtained with Equation (4); the shaded area within the nested subintervals shrinks away as the decoding progresses. (b) *Sliding-pin* decoding: The support interval and p are now fixed, while $\{q_i\}_{i=1}^5$ is subsequently shifted throughout the unit interval using Equation (6), thus resembling the strip of a sliding-rule; the output bit s_i^p in this case is given by Equation (8).

3.3 Truncated-normal random numbers

Algorithm 1 is designed to work in continuous domains where variables are assumed to be Gaussian-distributed over the real line. In an arithmetic coding scheme, however, the search of a feasible set of values for a pair (p, q) requires the evolutionary algorithm to generate random numbers bounded within the $[0, 1]$ interval[3]. For this reason we adapted the algorithm to a *truncated* Gaussian model with support on the unit interval only, following [4, page 39]: given a random variable X with cumulative probability distribution Φ, the truncated random variable X^T with distribution Φ^T and support $[a, b]$ is defined as

$$\Phi^\mathsf{T}(x) = \begin{cases} 0 & x < a \\ \frac{\Phi(x)-\Phi(a)}{\Phi(b)-\Phi(a)} & a \le x \le b \\ 1 & x > b \end{cases}. \tag{9}$$

We define $\Phi \equiv \mathcal{N}(\mu, \sigma), a = 0, b = 1$ and normalise $x' = \frac{x-\mu}{\sigma}$, $a' = \frac{a-\mu}{\sigma} = \frac{-\mu}{\sigma}$ and $b' = \frac{b-\mu}{\sigma} = \frac{1-\mu}{\sigma}$. Let $u \sim [0, 1]$ be a uniformly distributed random number; hence we use the inversion method for random number generation (see e.g. [4]), that is, assume $u = \Phi^\mathsf{T}(x')$ and work out the resulting x bounded to [0,1]. Since $0 \le x' \le 1$ from Equation (9) we get:

$$\Phi(x') = u(\Phi(b') - \Phi(a')) + \Phi(a'),$$

which in turn implies that,

$$x = \sigma(\Phi^{-1}(u(\Phi(b') - \Phi(a')) + \Phi(a')) + \mu. \tag{10}$$

Since $\Phi(a')$ and $\Phi(b')$ are constant terms, Equation (10) can be used to generate a truncated random number by computing a uniform random number and one evaluation of the inverse normal distribution with given parameters (μ, σ).

4. ALGORITHM

To begin with, let us recall the arithmetic-coded EDA proposed in [22]. It is a continuous domain population-based EDA that takes advantage of the shrinking-interval arithmetic decoding, defined in Equations (4-5), in order to search for a solution string $\mathbf{s}_p^q$ in a high dimensional space of size ℓ. The compression scheme reduces the memory requirements needed to store and maintain the population of candidates, enabling the algorithm to solve large-scale problems in a standard desktop machine. Here we present an algorithm that further improves memory usage by means of a compact representation.

The new algorithm is an adaptation of TILDA (as described in Section 3.1). The idea is to search for the best $(p, q) \in \mathbb{R}^2$ that determines the string $\mathbf{s}_p^q$ whose fitness function is evaluated in $\{0, 1\}^\ell$. For this purpose, we let $\boldsymbol{\mu}, \tilde{\boldsymbol{\mu}}, \boldsymbol{\Sigma}, \tilde{\boldsymbol{\Sigma}}, \boldsymbol{\beta} \in \mathbb{R}^2$ in Algorithm 1 and we chose the sliding-pin scheme of Equations (6-8) to decompress $\mathbf{s}_p^q$. Likewise, we decided on using the truncated Gaussian random generator of Equation (10) for sampling. The resulting method (@TILDA: *arithmetic-coding* TILDA) is shown in Algorithm 2.

The time complexity of the algorithm is dependent on the convergence speed, the population size n, and the cost needed to decompress and evaluate the candidate (usually linear in ℓ). More specifically, neglecting the cost of random number generation, each inner loop iteration incurs $O(\ell)$ for the two fitness evaluations in line 6 (the fitness evaluations in line 7 can be spared by caching previous calls). So, assuming t_c iterations are needed to converge, the running time is in $O(n\ell t_c)$.

[3]Let us mention that mechanisms for truncated pseudorandom number generation in other support intervals has been also highlighted in previous EDA studies (e.g. [7, 12])

Algorithm 2 @TILDA

```
Requires: $\ell > 0$, fitness function $f(\cdot)$
Outputs: best candidate $\boldsymbol{\beta}$
 1: $\boldsymbol{\beta} = \boldsymbol{\mu} = (0.5, 0.5), \boldsymbol{\Sigma} = (10, 10), \gamma = 0.9, \epsilon = \ell^{-1}$
 2: repeat until convergence
 3:     $\tilde{\boldsymbol{\mu}} = \tilde{\boldsymbol{\Sigma}} = (0, 0)$
 4:     repeat $n$ times
 5:         $\{(p', q'), (p'', q'')\} \sim \mathcal{N}^{\mathsf{T}}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ //sampling (Eq.(10))
 6:         $(\hat{p}, \hat{q}) = \mathsf{argmax}(f(\mathbf{s}_{q'}^{p'}), f(\mathbf{s}_{q''}^{p''}))$ //decoding (Eq.(6-8))
 7:         $\boldsymbol{\beta} = \mathsf{argmax}(f(\mathbf{s}_{\hat{q}}^{\hat{p}}), f(\mathbf{s}_{\beta_2}^{\beta_1}))$ // update best
 8:         $\tilde{\boldsymbol{\mu}} = \tilde{\boldsymbol{\mu}} + \frac{1}{n}(\hat{p}, \hat{q})$
 9:         $\tilde{\boldsymbol{\Sigma}} = \tilde{\boldsymbol{\Sigma}} + \frac{1}{n}(\hat{p}^2, \hat{q}^2)$
10:     $\boldsymbol{\mu} = \boldsymbol{\mu} - \gamma(\boldsymbol{\mu} - \frac{1}{2}(\boldsymbol{\beta} + \tilde{\boldsymbol{\mu}}))$
11:     $\boldsymbol{\Sigma} = \boldsymbol{\Sigma} - \gamma(\boldsymbol{\Sigma} - (\tilde{\boldsymbol{\Sigma}} - (\tilde{\mu}_0^2, \tilde{\mu}_1^2)) + \epsilon$
```

Now we discuss an interesting memory saving mechanism obtainable with this new algorithm. Observe that its memory cost is dominated by the size of the problem ℓ. This is because the working memory consists of nine bi-dimensional variables in $O(1)$ plus the temporal buffer needed to decompress $\mathbf{s}_p^q$, which is in $O(\ell)$. However, if we constrain the application domain to additively-separable problems exhibiting modularity of fitness evaluation to the level of single variables, that is, problems where the fitness function aggregates independent contributions in each dimension of the solution, then the memory cost can be $O(1)$. The latter is possible by incrementally evaluating and disposing bits obtained during the stream decompression of $\mathbf{s}_p^q$. And given that the best solution can be retrieved at any point of the evolutionary loop from the arithmetic code $\boldsymbol{\beta} = (\beta_1, \beta_2)$, corresponding to $\mathbf{s}_{\beta_1}^{\beta_2}$, no string buffering is actually needed. As mentioned earlier, even if bits are not discarded (i.e. buffering them into a string) the fitness evaluation requires $O(\ell)$ time to scan the string, so the memory savings using streaming implies no additional time overload. It is in this sense that the algorithm is well suited for large scale problems, ensuring minimal memory consumption, regardless of the problem and population size. The memory efficiency of @TILDA compares favorably with arithmetic-coded EDA [22], which requires $O(\ell + n)$ for the evolutionary loop (although it could be lowered to $O(n)$ by using the same streaming technique), and also with cGA [8], which requires $O(\ell(\log_2 n + 2))$.

It may be noted in passing that, by using the sliding-pin arithmetic-coding technique and the truncated random number generator, @TILDA is able to always decompress feasible solutions notwithstanding the problem size ℓ. The latter contrasts with the difficulties found for the arithmetic-coded EDA, where some (p, q) codes where unable to produce the desired string length due to precision issues [22].

5. EMPIRICAL STUDY

We conducted an empirical study of @TILDA applied to the OneMax [15] and RoyalRoad [13] problems (including noisy variations). Fitness functions for a candidate solution ($\mathbf{s} \in [0, 1]^\ell$) to these problems are defined in Table 1. For the case of RoyalRoad functions, the parameter c represents a score added to a candidate for each succeeding schema of order c that can be found within its bit string, starting at bit 1 (in contrast to the original definition given in [13], here only a single block scan of ℓ/c schemas is carried out to compute

Problem	Fitness function
OneMax	$f(\mathbf{s}) = \sum_{i=1}^{\ell} s_i$
Noisy OneMax	$\hat{f}(\mathbf{s}) = f(\mathbf{s}) + \mathcal{N}(0, \sigma_N^2)$
RoyalRoad	$f_c(\mathbf{s}) = \sum_{i=0}^{(\ell/c)-1} c \left(\prod_{k=ic+1}^{(i+1)c} s_k \right)$
Noisy RoyalRoad	$\hat{f}_c(\mathbf{s}) = f_c(\mathbf{s}) + \mathcal{N}(0, \sigma_N^2)$

Table 1: Fitness functions used in the empirical study for a candidate $\mathbf{s} = \{s_1, \ldots, s_\ell\}$, $s_i \in \{0, 1\}$. $\mathcal{N}(0, \sigma_N^2)$ denotes a normally distributed random variable of variance σ_N^2.

```
Requires: $(p, q), \ell, noise$
Outputs: $f$
  $f = 0$
  repeat $\ell$ times
    $bit = q > p$
    if $bit$ then
        $q = (q - p)/(1 - p)$
        $f = f + 1$
    else $q = q/p$
  end-repeat
  if $noise$ then $f = f + \mathcal{N}(0, \sigma_N^2)$
```

(a) OneMax

```
Requires: $(p, q), \ell, c, noise$
Outputs: $f$
  $f = 0; k = 1; block = $ true
  repeat $\ell$ times
    $bit = q > p;\ \ block = block \wedge bit$
    if $bit$ then $q = (q - p)/(1 - p)$
    else $q = q/p$
    if $k = c$ then
        $k = 0$
        if $block$ then $f = f + c$
        else $block = $ true
    end-if
    $k = k + 1$
  end-repeat
  if $noise$ then $f = f + \mathcal{N}(0, \sigma_N^2)$
```

(b) RoyalRoad

Table 2: Stream algorithms for string decompression and fitness evaluation. (a) OneMax: this algorithm is a variation of the sliding-pin arithmetic decoder of Equations (6-8), where the fitness f is increased every time the current *bit* is set to true; inputs are: arithmetic code (p, q), problem size ℓ, and a *noise* flag to activate the noisy version. (b) RoyalRoad: based on the OneMax code above, and using c as an additional input defining the order of the schema, this algorithm keeps a *block* flag that is successively updated with the conjunction of the decoded *bit* variable; for every succeeding schema, whose extent is controlled with counter k, score c is added to f if the schema *block* flag is set, otherwise this flag is readjusted.

fitness; no additional hierarchical scans of higher -doubled-order schemas follow up).

Let us observe that the evaluation of these fitness functions allows additively-separable computations. Hence we implemented stream decompression/fitness-evaluation algorithms (see Table 2) in order to take advantage of the memory savings of `@TILDA`[4]. Instead of $\mathbf{s}$, these stream algorithms take an arithmetic code (p, q) as their input, which implicitly defines the bit string $\mathbf{s}_p^q$ of Equation (7), and simultaneously decompress the candidate and compute its fitness value f. In this way we were able to run the experiments in a conventional laptop Intel® Core$^{\text{TM}}$ i7 machine with 6GB memory. Algorithm 2 and test-bed scripts were implemented in both Octave version 3.2.4 with QtOctave version 0.9.1 and Matlab R2011. Fitness routines in Table 2 were implemented in C with the MEX interface for faster evaluation.

5.1 Results on noiseless problems

The first set of experiments were aimed at testing the behaviour of the algorithm on noise-free (regular) problems. We set the number of maximum generations to 1000 for all experiments, with an early-stopping condition whenever a candidate decoding a number of bits equals to the problem size ℓ was found. In order to set the appropriate population size needed to find a solution with respect to ℓ, we carried out a grid-search [9] over $n \in \{2^3, 2^4, \ldots, 2^{12}\}$; the search was stopped when, for a given n, at least 90% of 100 runs were successful. For the `RoyalRoad` case, the schema order c was set to 100 for all experiments. The values of learning rate and premature convergence parameters ($\gamma = 0.9$ and $\epsilon = \ell^{-1}$) were chosen from empirical evidence of preliminary experiments. Results of the final experiments are listed in Table 3.

In summary, these results show that the algorithm succeeded in solving both problems from medium- to large-scale lengths (10^3 to 10^7 variables). Interestingly enough, it can be seen that a small population size $n \leq 32$ was needed to solve the problems, even for the large dimensionalities. The average generation of solution emergence was much smaller than the maximum setting of 1000. On the other hand, a comparable trend of performance is visible in the `OneMax` and `RoyalRoad` instances, except that a bigger population size was needed in the smaller problems for the latter. This finding may drop a hint on the potential effectiveness of the algorithm to tackle basic levels of structure in the problem (recall that `RoyalRoad`fitness function assigns score only to entire schemas of length c, that is, contiguous chunks of variables).

5.2 Results on noisy problems

The second set of experiments focused on testing the robustness of the algorithm when exogenous noise was added to the original fitness function. We followed the same settings as before but with 30 runs. Again the population size was obtained by a grid-search. The level of noise was controlled with a exogenous-noise-variance to deterministic-fitness-variance ($\frac{\sigma_N^2}{\sigma_f^2}$) parameter as defined in [20]. We tested

[4]We remark that the algorithm is suitable as well for non-decomposable fitness functions, or problems where no stream decompressing algorithm is available. In the worst of these cases its memory complexity, as mentioned earlier, would be in $O(\ell)$, still a favourable scenario for large-scale applications.

Problem size	Solution	`OneMax`	`RoyalRoad`
$\ell = 10^3$	n	8	16
	Success rate	100/100	96/100
	Age (avg.)	29.62	58.22
$\ell = 10^4$	n	8	16
	Success rate	99/100	92/100
	Age (avg.)	77.21	42.33
$\ell = 10^5$	n	8	16
	Success rate	98/100	99/100
	Age (avg.)	146.58	93.24
$\ell = 10^6$	n	16	16
	Success rate	94/100	94/100
	Age (avg.)	229.95	207.27
$\ell = 10^7$	n	32	32
	Success rate	95/100	91/100
	Age (avg.)	214.88	235.59

Table 3: Noise-free experiments. For each problem (`OneMax`, `RoyalRoad`) and dimension ℓ, results are reported on n (population size required to solve it), success rate (successful runs over 100 repetitions) and average age of the solution (generation where optimal candidate emerged).

the algorithm using a corruption $\frac{\sigma_N^2}{\sigma_f^2}$ ratio of 10^{-2}. Results of this set of experiments are listed in Table 4.

In these experiments, the results on both problems are dissimilar. For the `Noisy OneMax` problem the incidence of noise in medium-size instances causes no difficulty to the algorithm in solving it. In larger instances, however, ($\ell = \{10^5, 10^6\}$) the algorithm struggles and demands higher population sizes of up to $n = 2048$ but still obtains success rates inferior to 50%. In contrast, a small population size $n = 32$ seems appropriate to solve all instances of `Noisy RoyalRoad` problems. We reason that in this case, the block-structure of the fitness score diminishes the effect external noise may have in misguiding the algorithm during its search.

6. CONCLUSIONS

We introduced a new compact `EDA` algorithm that is able to solve to optimality additively-separable bit-string problems of up to 10^7 dimensions with minimum memory requirements, both at individual or block contribution of variables to the fitness score. In the presence of noise, the algorithm is suitable to solve medium to large-size instances of `Noisy RoyalRoad` problems but struggles with large scale `Noisy OneMax` problems. On this topic we are considering exploring different ideas for noise tolerance such as the effect of using a combination of best candidates obtained from independent shorter, quicker runs, and also analysing in depth the role of elitism in such scenario.

In addition we are planning to study the role of the learning rate and premature convergence parameters within more elaborated mechanisms as those proposed in existing `EDA` literature. Moreover, it would be interesting to study the behaviour of the algorithm in different kinds of fitness landscapes, for example, uneven (not all-ones) optimal solutions and also, deceptive functions. Yet another exciting avenue of

Problem size	Solution	`OneMax`	`RoyalRoad`
$\ell = 10^3$	n	8	32
	Success rate	28/30	30/30
	Age (avg.)	59.93	32.20
$\ell = 10^4$	n	64	32
	Success rate	30/30	30/30
	Age (avg.)	48.17	48.47
$\ell = 10^5$	n	2048	32
	Success rate	19/30	27/30
	Age (avg.)	134.37	133.90
$\ell = 10^6$	n	2048	32
	Success rate	12/30	28/30
	Age (avg.)	168.87	153.80

Table 4: Noisy experiments. An exogenous noise rate of $\frac{\sigma_N^2}{\sigma_f^2} = 0.01$ was used in both test problems. For each each problem (`OneMax`, `RoyalRoad`) and dimension ℓ, results are reported on n (population size required to solve it), success rate (successful runs over 30 repetitions) and average age of the solution (generation where optimal candidate emerged).

future research is extending the memory and time efficiency ideas here presented to models of higher-order variable interaction.

As a concluding remark we anticipate that a parallel implementation of the algorithm would improve running times, although at present, an average run to solve a one-million bit problem in a regular laptop with a population size of 2048 is 1 hour. Our current work aims to validate and extend these findings to very-high dimensional problems of up to one-billion variables.

7. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their insightful comments and suggestions.

8. REFERENCES

[1] T. Bäck and H.-P. Schwefel. An overview of evolutionary algorithms for parameter optimization. *Evolutionary Computation*, 1(1):1–23, 1993.

[2] S. Baluja and R. Caruana. Removing the genetics from the standard genetic algorithm. Technical Report CMU-CS-95-141, Carnegie-Mellon University, 1995.

[3] Y. Cai, X. Sun, H. Xu, and P. Jia. Cross entropy and adaptive variance scaling in continuous EDA. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, GECCO '07, pages 609–616, New York, NY, USA, 2007.

[4] L. Devroye. *Non-Uniform Random Variate Generation.* Springer-Verlag, 1986.

[5] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley Professional, 1 edition, 1989.

[6] C. Gonzáles, J. A. Lozano, and P. Larrañaga. Mathematical modelling of UMDA$_c$ algorithm with tournament selection. *International Journal of Approximate Reasoning*, 31(3):313–340, 2002.

[7] J. Grahl and F. Rothlauf. Polyeda: Combining estimation of distribution algorithms and linear inequality constraints. In *Proceedings of the 6th Annual Conference on Genetic and Evolutionary Computation*, GECCO '04, pages 1174–1185, 2004.

[8] G. R. Harik and F. G. Lobo. The compact genetic algorithm. *IEEE Transactions on Evolutionary Computation*, 3:523–528, 1999.

[9] C. W. Hsu, C. C. Chang, and C. J. Lin. *A practical guide to support vector classification.* Department of Computer Science, National Taiwan University, 2003.

[10] O. Kramer and F. Gieseke. Variance scaling for EDAs revisited. In *KI 2011: Advances in Artificial Intelligence*, volume 7006 of *Lecture Notes in Computer Science*, pages 169–178. Springer, 2011.

[11] P. Larrañaga and J. A. Lozano, editors. *Estimation of Distribution Algorithms: A New Tool for Evolutionary Computation.* Springer, 2001.

[12] E. Mininno, F. Cupertino, and D. Naso. Real-valued compact genetic algorithms for embedded microcontroller optimization. *IEEE Transactions on Evolutionary Computation*, 12(3):203–219, 2008.

[13] M. Mitchell, S. Forrest, and J. H. Holland. The royal road for genetic algorithms: Fitness landscapes and GA performance. In *Proceedings of the First European Conference on Artificial Life*, pages 245–254, 1991.

[14] H. Mühlenbein. The equation for response to selection and its use for prediction. *Evolutionary Computation*, 5(3):303–346, 1997.

[15] M. Pelikan and K. Sastry. Initial-population bias in the univariate estimation of distribution algorithm. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, GECCO '09, pages 429–436, New York, NY, USA, 2009. ACM.

[16] M. Pelikan, K. Sastry, and E. Cantú-Paz, editors. *Scalable Optimization via Probabilistic Modeling: From Algorithms to Applications.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[17] S. Rojas-Galeano, E. Hsieh, D. Agranoff, S. Krishna, and D. Fernandez-Reyes. Estimation of relevant variables on high-dimensional biological patterns using iterated weighted kernel functions. *PLoS ONE*, 3(3), 2008.

[18] A. Said. Introduction to arithmetic coding - theory and practice. Technical Report HPL-2004-76, Imaging Systems Laboratory, HP Laboratories Palo Alto, 2004.

[19] K. Sastry. Evaluation-relaxation schemes for genetic and evolutionary algorithms. Technical report, 2002.

[20] K. Sastry, D. E. Goldberg, and X. Llora. Towards billion-bit optimization via a parallel estimation of distribution algorithm. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, GECCO '07, pages 577–584, 2007.

[21] M. Sebag and A. Ducoulombier. Extending Population-Based incremental learning to continuous search spaces. *Lecture Notes in Computer Science*, 1498, 1998.

[22] W. Suwannik and P. Chongstitvatana. Solving one-billion-bit noisy onemax problem using estimation distribution algorithm with arithmetic coding. In *IEEE Congress on Evolutionary Computation*, pages 1203–1206, 2008.

B.2 Proceeding of the fifteenth annual conference companion on Genetic and evolutionary computation conference companion (GECCO), pages 1325-1332, 2013.

Goldenberry: EDA Visual Programming in Orange

Sergio Rojas-Galeano
Engineering School
District University of Bogota
Bogota, Colombia
srojas@udistrital.edu.co

Nestor Rodriguez
Engineering School
District University of Bogota
Bogota, Colombia
nearodriguezg@correo.udistrital.edu.co

ABSTRACT

`Orange` is an open-source component-based software framework, featuring visual and scripting interfaces for many machine learning algorithms. Currently it does not support Estimation of Distribution Algorithms (`EDA`) or other methods for black-box optimization. Here we introduce `Goldenberry`, an `Orange` toolbox of `EDA` visual components for stochastic search-based optimization. Its main purpose is to provide an user-friendly workbench for researchers and practitioners, building upon the versatile visual front-end of `Orange`, and the powerful reuse and glue principles of component-based software development. Architecture of the toolbox and implementation details are given, including description and working examples for the components included in its first release: `cGA`, `UMDA`, `PBIL`, `TILDA`, `UMDA`$_c$, `PBIL`$_c$, `BMDA`, `CostFunctionBuilder` and `BlackBoxTester`.

`Goldenberry` is open-source and freely available at:
`http://goldenberry.codeplex.com`.

Categories and Subject Descriptors

D.2.6 [**Software**]: Software Engineering—*Programming Environments*; I.2.8 [**Computing Methodologies**]: Artificial Intelligence—*Problem Solving, Control Methods and Search*

Keywords

Component-based evolutionary software systems; EDAs

1. INTRODUCTION

Visual environments for machine learning (e.g. Clementine or SPSS Modeler [9], Weka [7], RapidMiner [12]) provide graphical workbenches to conduct user-friendly data analysis. Instead of scripting commands in an imperative computer language, users are able to graphically sketch processing units and interactions that are needed to run said analysis. `Orange` is one of such open-source visual frameworks, originally proposed for functional genomic analysis [4]. It has been progressively enriched with additional visual software components (*widgets*) for several machine learning tasks. It is known that many of these tasks can be casted as, or make use of, optimization problems in their underlying machinery (regression analysis, regularization, clustering, feature selection); however as far as we know, to this day `Orange` does not include a set of tools available to model explicit optimization problems within the context of machine learning analysis. The latter is precisely the motivation behind the software system we introduce in this paper: `Goldenberry`, a supplementary machine learning and evolutionary computation suite for `Orange`.

The software was built by taking advantage of three interesting features in `Orange`: (i) its versatile visual front-end; (ii) the powerful reuse and glue postulates of component-based software development in which it is based; and (iii) its conformity to the *open/closed* principle of object-oriented programming through an scripting interface to Python. We reasoned that such features would be advantageous for extending its functionality to the realm of optimization problems. Since our research group has been recently working on stochastic-search-based optimization, particularly in the field of Estimation of Distribution Algorithms (`EDA` [10, 15]), a decision was made to start off the project in its first stage by focusing on this kind of algorithms. As a result, the first release of `Goldenberry` comprises a set of `EDA` components (`cGA` [8], `UMDA` [13], `PBIL` [2], `TILDA` [17], `PBIL`$_c$ [19], `UMDA`$_c$ [6] and `BMDA` [14]) and a set of utility components (`CostFunctionBuilder` and `BlackBoxTester`) that we shall describe in the following sections. The second stage of the project is currently under development; it will comprise other black-box optimization techniques and also additional meta-heuristics and machine learning components.

The paper describes a general depiction of the software architecture, and provide working examples of its operation using standard benchmark and custom optimization problems. The latter was made possible by using the built-in cost functions or the free-text Python input mode from the `CostFunctionBuilder` component, which are convenient design features allowing this suite to be applied over a wide range of discrete and continuous optimization domains. `Goldenberry` is also open-source and is freely available at:
`http://goldenberry.codeplex.com`
For download instructions please refer to the Appendix.

2. THE SOFTWARE AT A GLANCE

`Goldenberry` was built as a suite of *widgets* for `Orange`. Widgets are visual elements that can be dragged onto the `Orange` visual programming board, also known as *canvas*.

Furthermore, widgets provide a set of input/output interfaces that encapsulate related services; many widgets collaborate to perform a given task. “Programming” in the canvas amounts to manually wiring up the appropriate interfaces between widgets. An example of a `Goldenberry` program to optimize a cost function using three different `EDA` widgets is shown in Figure 1. In that program, widgets are the components or processing units needed to perform the optimization task. Rather than depicting a *dataflow* between these components, connections represent provision and consumption of services encapsulated as objects, which are associated with each input/output interface. For instance, in this program the `cGA`, `PBIL` and `UMDA` widgets require a function to be optimized that is provided by the `CostFunctionBuilder` widget through three instances of the `CostFunction` object, one per `EDA`, each in charge of keeping statistics of the number of function evaluations and running times. The `EDA` components are responsible of setting up the parameters of their corresponding algorithm and of providing a ready-to-use `Optimizer` that is in turn, consumed by the `BlackBoxTester`. The latter is in fact responsible of orchestrating the execution of these `Optimizers`, that is, of running each `EDA` algorithm (whose parameters, including cost function, must have been already defined and provided in their output interfaces before execution begins) and also of collecting and visualizing the final results. Parameter settings and data presentation are embodied within the the graphical user interface of the widgets, as we shall depict in Section 5 and 6 (also Figure 6 to Figure 11). Intuitively, this programming paradigm is very much the same as building a hardware apparatus: the user first gather, connect and set-up the required units before switching-on the resulting device.

3. ARCHITECTURAL VIEW

3.1 General design considerations

The following software patterns were taken into account during the conception of `Goldenberry`:

- A *layering* pattern [3] was applied in order to decouple the algorithmic logic of the components from their user interface (widgets); thereby all the implemented algorithms can be used and integrated in plain Python scripts or using the command-line.

- A *template method* pattern [5] was used by identifying commonalities of the different `EDA` approaches. In this way a generic abstraction of an `EDA` procedure was defined; concrete details of implementation were deferred to each particular algorithm. A unified `EDA` framework was achieved.

- A *responsibility splitting* pattern [1] was utilised in `EDA` optimisers so as to separate the search metaheuristic from the probability estimation technique. Therefore probability models were made reusable among multiple `EDAs` (those included in this and future releases).

- A *dynamic binding* [1] together with an *interpreter* pattern [5] were applied in order to allow the user to provide customised cost functions at runtime, in the form of Python-like scripts.

The application of such patterns guided the resulting design of the software, as described in the remainder of this section.

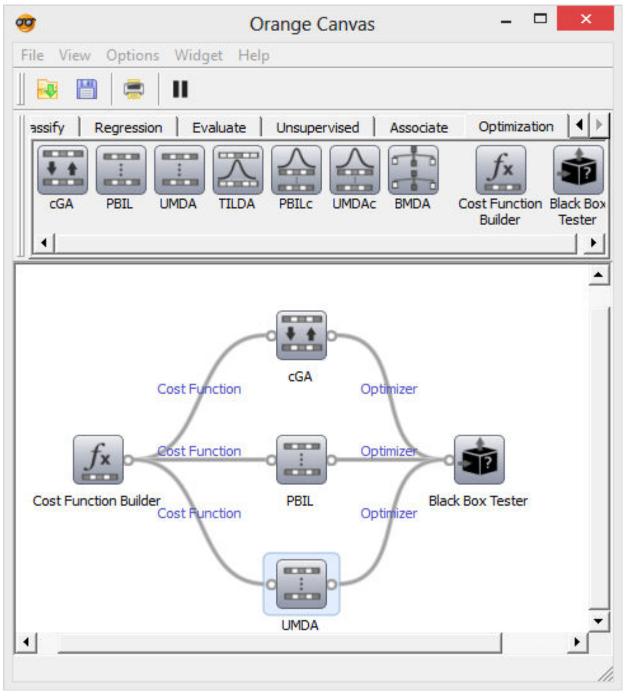

Orange Canvas
File View Options Widget Help
assify Regression Evaluate Unsupervised Associate Optimization
cGA PBIL UMDA TILDA PBILc UMDAc BMDA Cost Function Builder Black Box Tester
Cost Function
Optimizer
cGA
Cost Function
Optimizer
Cost Function Builder
PBIL
Black Box Tester
Cost Function
Optimizer
UMDA

Figure 1: A `Goldenberry` program in the `Orange` canvas.

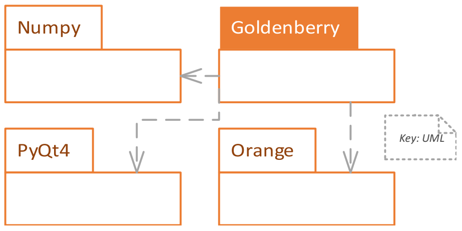


Figure 2: The context diagram of `Goldenberry`.

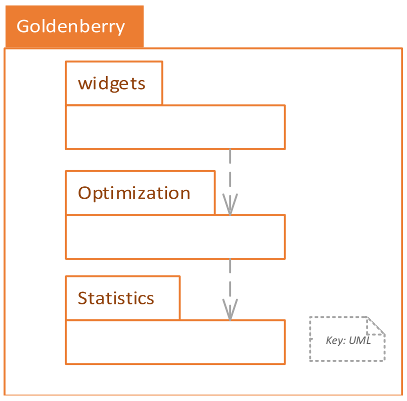


Figure 3: The three basic modules in `Goldenberry`.

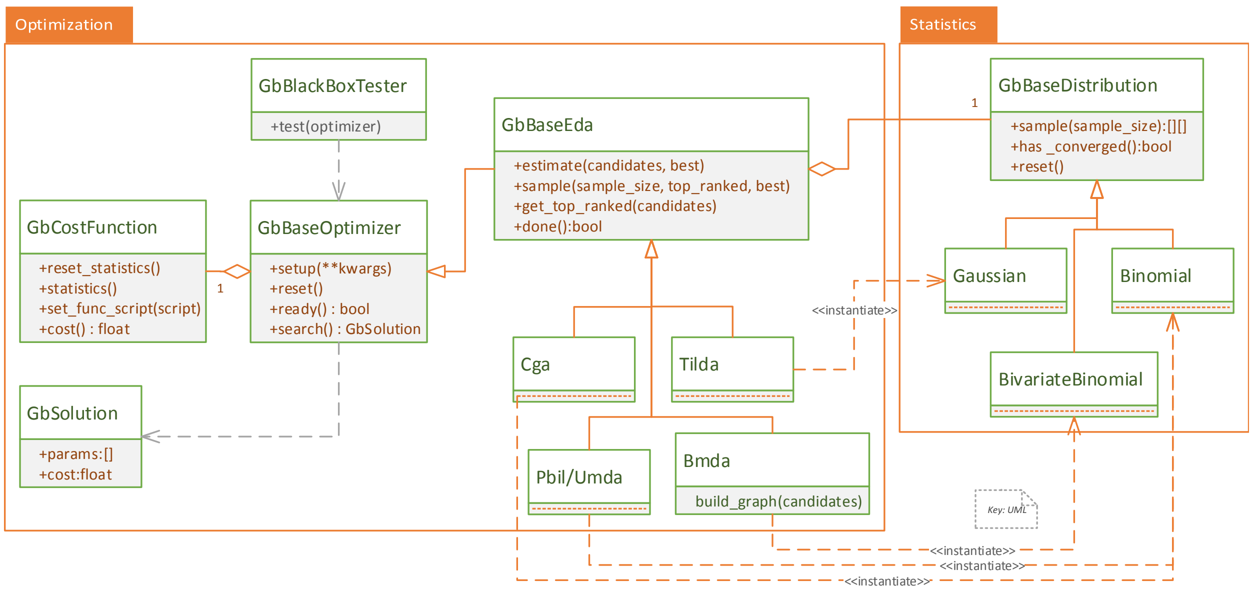


Figure 4: Decomposition view of Goldenberry modules.

3.2 Architecture packages

Widgets are actually visual wrappers for software components written in the Phyton scripting feature provided by Orange. Implementation of any widget or suite of widgets require the use of the Orange core library, the PyQt4 library for user-interface designing tools and optionally, the NumPy library for additional scientific computation support (PyQt4 and NumPy are standard libraries of the Python programming language). Therefore, the top-level organization of the Goldenberry suite of widgets is illustrated in the context diagram of Figure 2.

The internal architecture of the current release is shown in the module diagram of Figure 3. Three modules were developed. The widgets modules are the visual wrappers for the suite of Goldenberry components. The Optimization module includes objects and routines needed to implement the EDA components; this is the core module of the suite. Finally, Statistics is a utility module to allow modelling of some probability distributions used by the EDA algorithms.

4. STRUCTURAL VIEW

A decomposition view of the Optimization and Statistics modules including object classes and dependencies, is shown in the class diagram of Figure 4. It is expected that the modular design proposed here, would allow for new or customized EDA algorithms, probability distributions or other stochastic-search optimizers, to be added to the toolbox as extensions of such architecture. This would be one of the advantages of observing the software patterns earlier mentioned in Section 3.1.

4.1 Base classes

A core class GbBaseOptimizer was designed as an abstract class representing any black-box optimization algorithm (see Figure 4). The optimizer uses a GbSolution object, which holds the values of the parameters or variables in an arbitrary solution to a given problem (the vector params[]); it also holds its associated cost. The optimizer additionally encapsulates a setup() method to initialize its running parameters (e.g. problem size or number of variables, maximum number of evaluations, etc.) and the reset() method to set up the optimizer for a new run. The key method search(), defines the actual optimization algorithm which returns a best found solution; ready() is a checkpoint method to validate readiness of the optimizer to start the search.

A given candidate solution is evaluated with the cost() method from the GbCostFunction class (in the evolutionary computation literature this would be equivalent to computing the *fitness* of a candidate). The routine to evaluate the function is defined via the set_func_script() method. Furthermore, this class also includes a method to keep track of statistics(), such as the number of cost function evaluations or max/min/mean cost values; the reset_statistics() method clears up the statistics working memory.

4.2 EDA classes

In the current release Goldenberry provides implementation of only EDA-type optimizers. Other black-box optimization techniques would be added in the near future. Hence, a specialized abstract class GbBaseEda was derived from GbBaseOptimizer (see Figure 4). This class defines the two distinctive methods of any EDA: estimate() as the mechanism that builds up its probabilistic model, and sample() as the algorithm to generate new candidate solutions from that model. Two additional methods were designed: get_top_ranked() selects the promising candidates from where the probability model estimation is updated; and done(), which checks for convergence of the estimated model. These are the methods that would be iteratively executed during a run of the search() method from the parent class.

A particular implementation of the GbBaseEda class aforementioned, determines a type of EDA algorithm that would be used as optimizer. Goldenberry features a number of concrete EDA implementations, including univariate algorithms such as cGA, TILDA, PBIL and UMDA (discrete and continuos-domain variants for the last two), and also a bivariate algorithm, the BMDA. The latter extends the base class with an additional method build_graph() to model pairwise dependencies between problem variables. This assortment of algorithms was chosen so as to incorporate techniques using both univariate and bivariate probability distribution approaches. In the first release of the software we emphasised in univariate versions due to their algorithmic simplicity; nonetheless, future releases will build upon these algorithms and contemplate higher-order EDAs using tree-based, Bayes and dependency networks techniques.

For illustration purposes, we shall now outline some of the Python scripts implementing these classes. Firstly, let us recall the *Population-Based Incremental Algorithm* (PBIL) [2]. The aim of this algorithm is to discover a real-valued probability vector from which a population of competent binary candidate solutions can be sampled. The algorithm starts-off with a random-valued vector; then, the vector is iteratively sampled in order to refine the model using the most promising solutions from the sample and an *incremental learning* re-estimation rule, until the vector converges to a fixed distribution. Our rendering of this procedure is shown in Algorithm 1, were a vectorized operation mode was assumed.

Algorithm 1 PBIL

```
Requires: Cost function $\mathsf{fitness}(\cdot)$, binomial distribution model
    $\mathcal{B}(\boldsymbol{\theta})$ with parameters $\boldsymbol{\theta} \in \mathbb{R}^d$, learning rate $0 \leq \eta \leq 1$
Outputs: Solution $\mathbf{s} \in \mathbb{R}^d$
  Initialize $\boldsymbol{\theta}$ and $\mathbf{s}$ with random values in $[0,1]^d$
  repeat until convergence
    Sample $n$ candidates from model: $P \sim \mathcal{B}(\boldsymbol{\theta})$
    Assess fitness of candidate population: $\mathbf{f} = \mathsf{fitness}(P)$
    Choose $m$ candidates: $S = \{\mathbf{x}_i \in P : f_i \in$ top-$m$-ranked$\}$
    Re-estimate model: $\boldsymbol{\theta} = (1-\eta)\boldsymbol{\theta} + \eta \frac{1}{m} \sum_i \mathbf{x}_i, \quad \mathbf{x}_i \in S$
    Update solution: $\mathbf{s} = \mathsf{argmax}(\mathsf{fitness}(\mathbf{s}), \mathsf{fitness}(S^{top}))$
```

In Goldenberry, the PBIL widget defines a class Pbil which overrides the initialization() and estimate() methods according to the previous algorithm. Thus, PBIL is fully implemented as the following class script (notice that the average vectorized auxiliary operation from the NumPy library (np) is used):

```
class Pbil(GbBaseEda):
 def initialize(self):
   self.distr=Binomial(self.var_size)

 def estimate(self, top_ranked, best):
   self.distr.p=self.distr.p*(1-self.learning_rate)
      + self.learning_rate * np.average(top_ranked)
```

The small size of the script is due to the fact that many algorithmic details have been inherited from the parent class, GbBaseEda, because they are common to most of other EDAs. For example, the parent class defines the initialization of parameters such as sample_size, pop_size (size of the population), selection_rate (percentage of top-ranked selected candidates), and max_evals (limit on the number of cost function evaluations allowed on an entire search). Other common features such as the sample() method (which delegates this task to the respective probability model), the get_top_ranked() method to select the most promising candidates from the sample, and the search() method itself, are defined in this abstract class. Its script is partially shown below.

```
class GbBaseEda(GbBaseOptimizer):
...
 def sample(self, sample_size, top_ranked, best):
    return self.distr.sample(sample_size)

 def get_top_ranked(self, candidates):
    fits = self.cost_func(candidates)
    index = np.argsort(fits)
     [:(self.cand_size*self.selection_rate/100):-1]
    return candidates[index],
     bSolution(candidates[index[0]],fits[index[0]])
...
 def search(self):
     if not self.ready():
      raise Exception("Optimizer not ready.")
     best = GbSolution(None, float('-Inf'))
     top_ranked = None
     while not self.done():
         candidates=self.sample(self.sample_size,
                                top_ranked, best)
         top_ranked, winner =
                  self.get_top_ranked(candidates)
         self.estimate(top_ranked, best)
         if best.cost < winner.cost:
             best = winner
         self.iters += 1
     return best

 @abc.abstractmethod
 def estimate(self, candidates, best):
     raise NotImplementedError()
```

It can be seen in the previous code that the implementation of estimate(), the estimation of distribution method, is deferred to the specific EDA class, as it was the case of the PBIL component.

Now let us complete the illustration by mentioning another well-known EDA, the *Compact Genetic Algorithm* (cGA) [8]. The algorithm is similar in fashion to PBIL, the main difference being that it operates in a *compact* mode, that is, instead of estimating the distribution from a batch of many candidates (population), the algorithm works by sampling two candidates at a time and using them to incrementally build the estimation. The pseudo-code of this procedure is thus shown in Algorithm 2, which again is our rendition of the original, written in vectorized operation mode.

Algorithm 2 cGA

```
Requires: Cost function $\mathsf{fitness}(\cdot)$, binomial distribution model
    $\mathcal{B}(\boldsymbol{\theta})$ with parameters $\boldsymbol{\theta} \in \mathbb{R}^d$
Outputs: Solution $\mathbf{s} \in \mathbb{R}^d$
  Initialize $\boldsymbol{\theta}$ and $\mathbf{s}$ with random values in $[0,1]$
  repeat until convergence
    repeat $n$ times
      Sample 2 candidates from model: $\{\mathbf{x}_1, \mathbf{x}_2\} \sim \mathcal{B}(\boldsymbol{\theta})$
      Rank them: $\{\mathbf{w}, \mathbf{l}\} = \mathsf{compete}(\mathsf{fitness}(\mathbf{x}_1), \mathsf{fitness}(\mathbf{x}_2))$
      Re-estimate model with winner and loser: $\boldsymbol{\theta} = \boldsymbol{\theta} + \frac{1}{n}(\mathbf{w} - \mathbf{l})$
      Update solution: $\mathbf{s} = \mathsf{argmax}(\mathsf{fitness}(\mathbf{s}), \mathsf{fitness}(\mathbf{w}))$
```

The cGA, as implemented in full in Goldenberry, is shown in the script below.

```
class Cga(GbBaseEda):
   def initialize(self):
       self.distr=Binomial(self.var_size)
       self.learning_rate=1.0/float(self.pop_size)
       self.sample_size=2

   def estimate(self, (winner, loser), best):
       self.distr.p =
         np.minimum(np.ones((1, self.var_size)),
           np.maximum(np.zeros((1, self.var_size)),
           self.distr.p +
  (winner.params-loser.params)*self.learning_rate))

   def get_top_ranked(self, candidates):
       costs = self.cost_func(candidates)
       maxindx = np.argmax(costs)
       winner = GbSolution(candidates[maxindx],
                           costs[maxindx])
       loser = GbSolution(candidates[not maxindx],
                          costs[not maxindx])
       return  (winner, loser), winner
```

The algorithm is initialized with the following settings: a binomial univariate probability model the size of the number of variables; learning rate of $1/n$ (where n is the population size); and sample size of two, because of its *compact* style. The estimate() method is overridden to account for a compact learning rule, using two competitors; here again, the auxiliary NumPy library is used (np). The get_top_ranked() is overridden accordingly, to return the winner and loser of the contest as the top-ranked set.

4.3 Statistics classes

The classes comprising this module were designed to account for the mechanisms needed to modeling probability distributions. A GbBaseDistribution is defined encapsulating a sample() method, a has_converged() validation method, and a reset() method for randomly initializing the parameters of the probability model (see Figure 4). Currently four probability distributions are implemented: Binomial, Gaussian, TruncatedGaussian and BivariateBinomial. Most of the functionality of these classes were implemented using standard NumPy routines such as generation of normally independent distibuted random vectors, and element-wise vector operators (details omitted).

5. COMPONENTS VIEW

Figure 5 shows the components view of the toolbox. For illustration purposes only two EDA components are shown: cGA for discrete domains and TILDA for continuous-valued domains (we recall the other implemented components are: PBIL, UMDA, PBIL$_c$, UMDA$_c$, BMDA).

5.1 The EDA components

Any of these components internally consists of a collaboration between an EDA algorithm and a probability model (e.g. cGA uses a binomial distribution, TILDA uses a Gaussian distribution, and so on). Each EDA component exposes two interfaces. The first interface requires a GbCostFunction object that is able to evaluate the cost function on a given candidate solution, and also is able to trace statistics of the total number of evaluations made by the EDA. The second interface provides a GbBaseOptimizer object, initialized and ready to run a stochastic search to optimize the cost function.

These components are at the end of the day deployed as widgets in the Orange canvas, under a new toolbar named "Optimization" (see again Figure 1). The user interface of these widgets consists of a setup/results window. In there, parameters settings are applied to the EDA component and optimization results are displayed in an output text box after execution of the algorithm. An example of the user interface for the cGA widget, applied to solve the classical OneMax problem with 100 variables, using a population size of 30, and 1000 maximum number of evaluations, is shown in Figure 6.

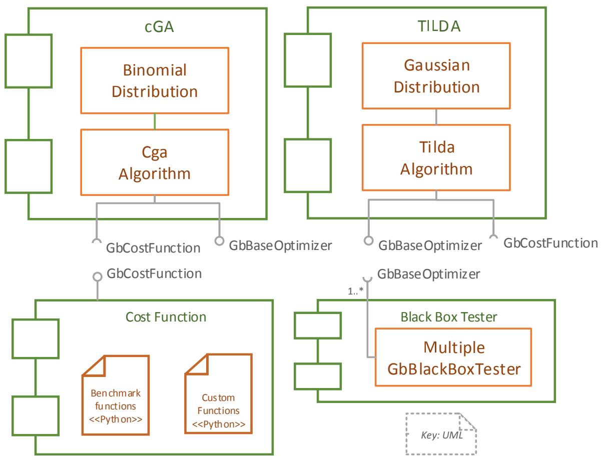
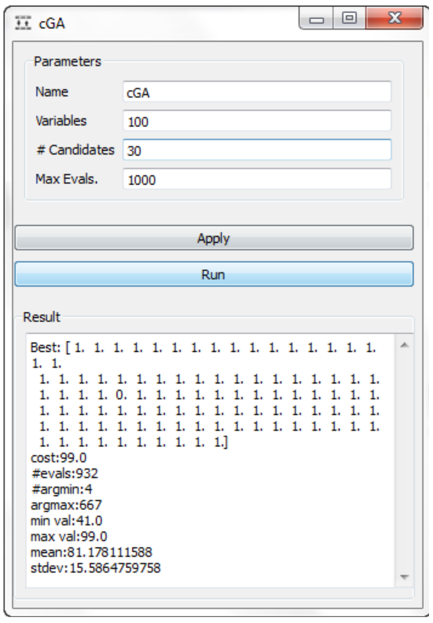


Figure 5: The components diagram of Goldenberry.



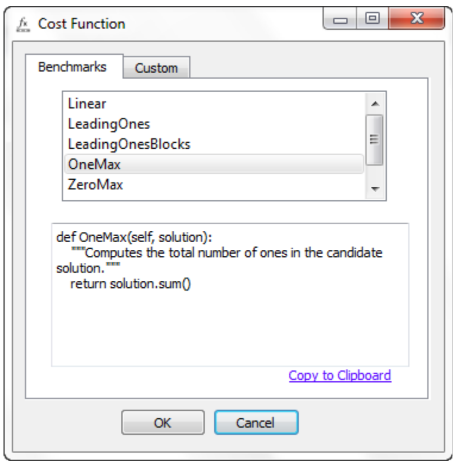
Parameters

Name	cGA
Variables	100
# Candidates	30
Max Evals.	1000

Result

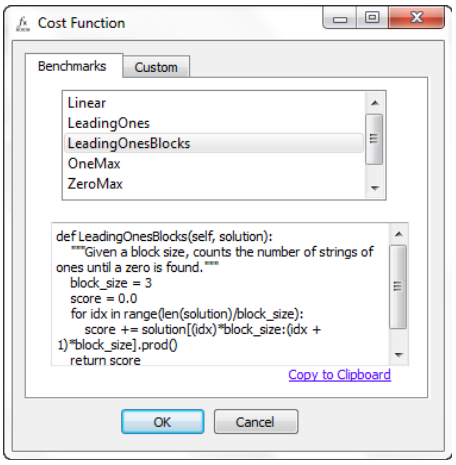
```
Best: [ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
1.  1.
  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
  1.  1.  1.  1.  0.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]
cost:99.0
#evals:932
#argmin:4
argmax:667
min val:41.0
max val:99.0
mean:81.178111588
stdev:15.5864759758
```

Figure 6: User interface of the cGA widget.



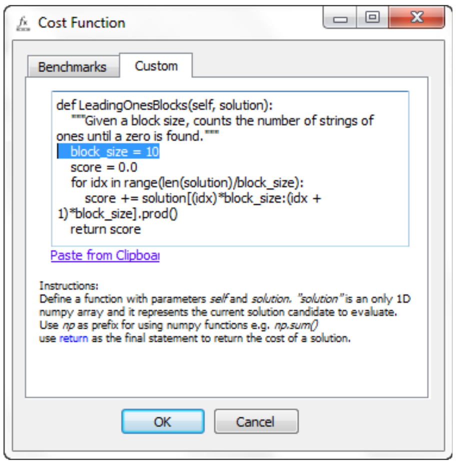
```
def OneMax(self, solution):
    """Computes the total number of ones in the candidate
solution."""
    return solution.sum()
```

Figure 7: The benchmark input mode of the `CostFunctionBuilder` component. Here the well-known `Onemax` problem is chosen.



```
def LeadingOnesBlocks(self, solution):
    """Given a block size, counts the number of strings of
ones until a zero is found."""
    block_size = 3
    score = 0.0
    for idx in range(len(solution)/block_size):
        score += solution[(idx)*block_size:(idx +
1)*block_size].prod()
    return score
```

(a)



```
def LeadingOnesBlocks(self, solution):
    """Given a block size, counts the number of strings of
ones until a zero is found."""
    block_size = 10
    score = 0.0
    for idx in range(len(solution)/block_size):
        score += solution[(idx)*block_size:(idx +
1)*block_size].prod()
    return score
```

Instructions:
Define a function with parameters *self* and *solution*. *"solution"* is an only 1D numpy array and it represents the current solution candidate to evaluate.
Use *np* as prefix for using numpy functions e.g. *np.sum()*
use return as the final statement to return the cost of a solution.

(b)

Figure 8: How to customize a built-in bechmark function: (a) A benchmark function is chosen and its code is copied into the clipboard; (b) the code is pasted in the "Custom" tab and edited as required.

5.2 The Cost Function Builder component

This component enables the user to define the cost function for the optimization problem. It provides two input modes to setup such a function. In the first mode, a set of ready-to-use built-in benchmark functions are listed to the user as it is illustrated in Figure 7. This mode appears on the "Benchmarks" tab of the `CostFunctionBuilder` user interface. The user chooses a function name, and then the Python script implementing the function is shown in the underlying text box; the box is read-only, so the code can not be edited but can be copied into the clipboard. The benchmark functions were taken from those suggested in [11].

The second mode consists of a free-text input box for writing up any customized cost function in the Python language. This mode appears on the "Custom" tab of the `CostFunctionBuilder` user interface. Custom functions must be written complying with the following Python-style signature:

```
def yourcustomfunctionname(self, solution):
...
return computedcostofsolution
```

An arbitrary routine to evaluate the given `solution` (a `NumPy` 1D vector array with the values of that solution to the problem variables) must be defined in order to compute its associated cost. For example, if the user wants to define a customized version of the benchmarks built-int functions, he or she may copy to the clipboard its original code, then paste it in the "Custom" tab and make the necessary adjustments (see Figure 8). Alternatively the user may write up his cost function code from scratch, to meet his particular problem needs. We remark that this a is non-intrusive input mode, meaning that the user-written-code is bind to the `Goldenberry` components in run time; no additional intervention has to be done in the source code of the software. We anticipate this feature would extend the usability of the `EDA` components to a wide range of discrete and continuous optimization problem domains.

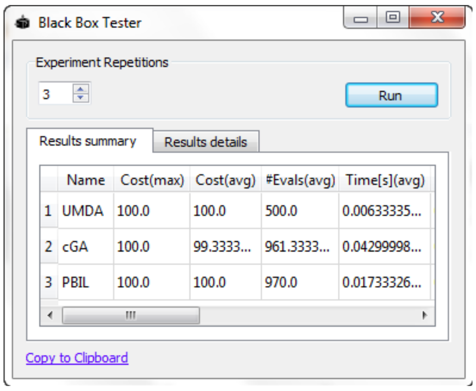
5.3 The Black-Box Tester component

This component was designed to allow the user to run and compare execution of different algorithms or algorithm configurations over the same cost function, in one experiment with several repetitions. The number of repetitions is set as an input parameter for this component. Another interesting functionality of the `BlackBoxTester` is that it is able to collect outputs of all the optimizer components provided as inputs, and display summarized statistics, and also details of the different runs and repetitions. The user interface of the widget associated to this component will be used to display results for the working experiments reported in the next section.

6. GOLDENBERRY AT WORK

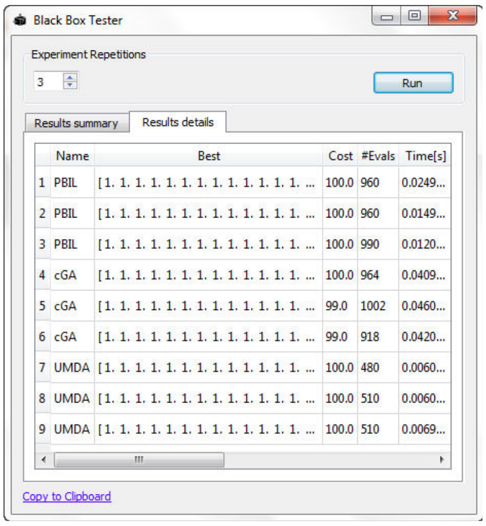
In this section we show how to use the developed components to solve optimization problems defined as minimization of a cost function. We carried out experiments using benchmark and customized cost functions. Other uses in machine learning can be also envisioned (see for example the discussion in Section 7).

The results of these experiments are shown next. It is worth to remark that previous to deployment, additional validation for the algorithmic machinery of each of the Goldenberry components was carried-out using a carefully designed set of unit tests (omitted because of space limitations). The program shown in Figure 1 was used in these experiments: A CostFunctionBuilder component is instantiated to select the optimization problem, as explained before; three EDA components were tested (cGA, PBIL and UMDA) using as input the selected CostFunction; finally, a BlackBoxTester executes and collects the outputs of the EDA components and shows summarised and detailed results over a number of repetitions of the experiment.



	Name	Cost(max)	Cost(avg)	#Evals(avg)	Time[s](avg)
1	UMDA	100.0	100.0	500.0	0.00633335...
2	cGA	100.0	99.3333...	961.3333...	0.04299998...
3	PBIL	100.0	100.0	970.0	0.01733326...

(a)



	Name	Best	Cost	#Evals	Time[s]
1	PBIL	[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. ...	100.0	960	0.0249...
2	PBIL	[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. ...	100.0	960	0.0149...
3	PBIL	[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. ...	100.0	990	0.0120...
4	cGA	[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. ...	100.0	964	0.0409...
5	cGA	[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. ...	99.0	1002	0.0460...
6	cGA	[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. ...	99.0	918	0.0420...
7	UMDA	[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. ...	100.0	480	0.0060...
8	UMDA	[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. ...	100.0	510	0.0060...
9	UMDA	[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. ...	100.0	510	0.0069...

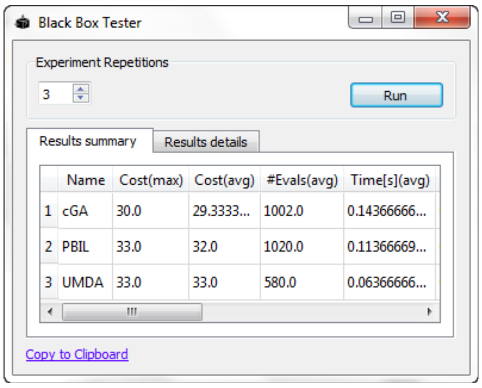
(b)

Figure 9: Onemax experiment results: (a) Summary over all repetitions (partial view); (b) Details per repetition (partial view).

6.1 Benchmark function optimization

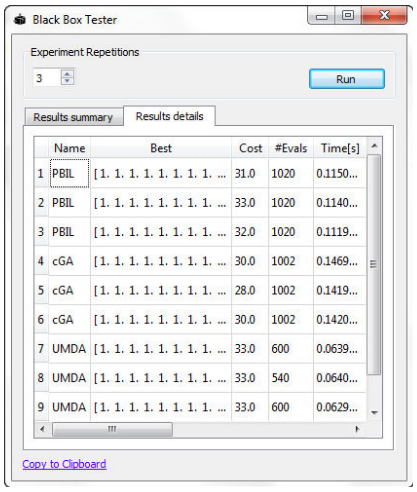
Two problems were chosen from the built-in benchmark library of functions: Onemax and LeadingOnesBlock. Results for the Onemax experiment are reported in Figure 9. Figure 9(a) shows the following aggregate results per EDA algorithm: maximun cost found in all experiment repetitions; average cost over all repetitions; average number of cost function evaluations per repetition; and average CPU time per repetition (in seconds). Other statistics can be displayed by scrolling the table bar to the right. Notice that the obtained tabulated results can be exported to spreadsheet software tools for further analysis by using the "Copy to Clipboard" option located underneath the output table. Figure 9(b) shows a detailed view of the results obtained in the experiments. Here each row holds the outputs per repetition for each input EDA component. The summarized results are statistics of these individual records. The only column not used in the summarized report is the actual vector representation of the best solution found in each run (for Onemax the expected solution is an all-ones vector). The cost of this solution is shown in the next column to its right.

For the LeadingOnesBlock experiment, results are similarly self-explained, as reported in Figure 10.



	Name	Cost(max)	Cost(avg)	#Evals(avg)	Time[s](avg)
1	cGA	30.0	29.3333...	1002.0	0.14366666...
2	PBIL	33.0	32.0	1020.0	0.11366669...
3	UMDA	33.0	33.0	580.0	0.06366666...

(a)



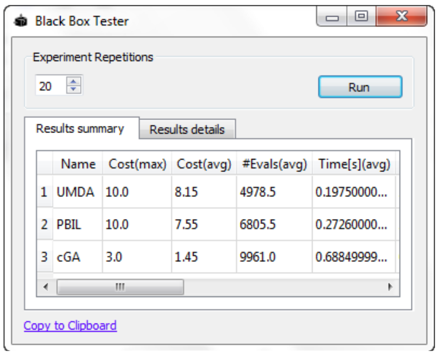
	Name	Best	Cost	#Evals	Time[s]
1	PBIL	[1. 1. 1. 1. 1. 1. 1. 1. ...	31.0	1020	0.1150...
2	PBIL	[1. 1. 1. 1. 1. 1. 1. 1. ...	33.0	1020	0.1140...
3	PBIL	[1. 1. 1. 1. 1. 1. 1. 1. ...	32.0	1020	0.1119...
4	cGA	[1. 1. 1. 1. 1. 1. 1. 1. ...	30.0	1002	0.1469...
5	cGA	[1. 1. 1. 1. 1. 1. 1. 1. ...	28.0	1002	0.1419...
6	cGA	[1. 1. 1. 1. 1. 1. 1. 1. ...	30.0	1002	0.1420...
7	UMDA	[1. 1. 1. 1. 1. 1. 1. 1. ...	33.0	600	0.0639...
8	UMDA	[1. 1. 1. 1. 1. 1. 1. 1. ...	33.0	540	0.0640...
9	UMDA	[1. 1. 1. 1. 1. 1. 1. 1. ...	33.0	600	0.0629...

(b)

Figure 10: LeadingOnesBlock experiment results: (a) Summary (partial view); (a) Details (partial view).

6.2 Customized function optimization

In this experiment we used the same Goldenberry program of Figure 1 to carry out optimization of the customized cost function defined in Figure 8, that is, the same LeadingOnesBlock problem this time with a block_size value of 10. Similarly to the other experiments, results are shown in Figure 11.



Experiment Repetitions: 20

	Name	Cost(max)	Cost(avg)	#Evals(avg)	Time[s](avg)
1	UMDA	10.0	8.15	4978.5	0.19750000...
2	PBIL	10.0	7.55	6805.5	0.27260000...
3	cGA	3.0	1.45	9961.0	0.68849999...

Figure 11: Results of custom function experiment.

7. CONCLUSIONS

It is our belief that user-friendly, open-source visual tools may have a big potential benefit in tasks carried out daily by data mining analysts. The `Orange` platform is a fantastic effort complying with these premises by combining a powerful visual programming approach with the reuse and glue principles of component-based software. The `Goldenberry` initiative is a modest contribution intended to extend the application domain of `Orange` to the field of black-box and metaheuristics optimization. In this first release we developed a number of `EDA`-based software components featuring a non-intrusive, runtime-binding interface to allow users to define tailor-made discrete and continuous optimization problems.

As it was mentioned in the introduction, it is anticipated that these components can be used independently as function optimizers, as it is reported in this paper, or as part of higher-level data mining machines. Let us illustrate the point with the task of feature selection. The aim there is to select an optimal subset of relevant variables for a classification problem; the relevant found subset can be further analysed by experts for specific purposes (in biology for example, features may represent over-expressed gene activity due to an illness condition). In the so-called *wrapper* scheme of the problem [18] the classifier is enabled to incorporate the selection mechanism during the learning stage: a weighted kernel classifier, for example, may use an `UMDA` component to define relevance coefficients for the variables and then use them as input for the kernel machine (that is the approach taken in [16]). Currently these type of mechanisms are hidden in the current implementation of classifier or clustering components of `Orange`. Thus, by providing interfaces to black-box optimizers we hope to extend applicability to this type of tasks, giving researchers a flexible workbench to design new component-based machine learning techniques. Novel components complying with the design principles described in this paper will be needed though (e.g. component-based genetic algorithms, kernel machines, etc.), and that would be matter of the next `Goldenberry` release.

8. ACKNOWLEDGEMENTS

We would like to acknowledge Henry Diosa, Leidy Garzón and Harry Sanchez, members of the Arquisoft Research Group from the District University of Bogota, for the design and implementation of the `BlackBoxTester` component. We also thank the anonymous reviewers for their valuable comments which allowed us to greatly improve readability of the paper.

9. REFERENCES

[1] F. Bachmann, L. Bass, and R. Nord. Modifiability tactics. Technical Report CMU/SEI-2007-TR-002, Software Engineering Institute, Carnegie Mellon University, 2007.
[2] S. Baluja and R. Caruana. Removing the genetics from the standard genetic algorithm. Technical Report CMU-CS-95-141, Carnegie-Mellon University, 1995.
[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. Wiley, volume 1 edition, Aug. 1996.
[4] T. Curk, J. Demsar, Q. Xu, G. Leban, U. Petrovic, I. Bratko, G. Shaulsky, and B. Zupan. Microarray data mining with visual programming. *Bioinformatics*, 21(3):396–398, 2005.
[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, Nov. 1994.
[6] C. Gonzáles, J. A. Lozano, and P. Larrañaga. Mathematical modelling of UMDA$_c$ algorithm with tournament selection. *International Journal of Approximate Reasoning*, 31(3):313–340, 2002.
[7] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. Witten. The WEKA data mining software. *SIGKDD Explor. Newsl.*, 11(1):10–18, 2009.
[8] G. R. Harik and F. G. Lobo. The compact genetic algorithm. *IEEE Transactions on Evolutionary Computation*, 3:523–528, 1999.
[9] IBM. *IBM SPSS® Algorithms Guide*. 2012.
[10] P. Larrañaga and J. A. Lozano, editors. *Estimation of Distribution Algorithms: A New Tool for Evolutionary Computation*. Springer, 2001.
[11] S. Luke. *Essentials of Metaheuristics*. Lulu, 2009. Available for free at http://cs.gmu.edu/∼sean/book/metaheuristics/.
[12] I. Mierswa, M. Wurst, R. Klinkenberg, M. Scholz, and T. Euler. Yale: Rapid prototyping for complex data mining tasks. In L. Ungar, M. Craven, D. Gunopulos, and T. Eliassi-Rad, editors, *Proceedings of the 12th ACM SIGKDD*, pages 935–940, NY, USA, August 2006. ACM.
[13] H. Mühlenbein. The equation for response to selection and its use for prediction. *Evolutionary Computation*, 5(3):303–346, 1997.
[14] M. Pelikan and H. Müehlenbein. The bivariate marginal distribution algorithm. In R. Roy, T. Furuhashi, and P. Chawdhry, editors, *Advances in Soft Computing*. Springer London, 1999.
[15] M. Pelikan, K. Sastry, and E. Cantú-Paz, editors. *Scalable Optimization via Probabilistic Modeling: From Algorithms to Applications*. Springer-Verlag, NJ, USA, 2006.
[16] S. Rojas-Galeano, E. Hsieh, D. Agranoff, S. Krishna, and D. Fernandez-Reyes. Estimation of relevant variables on high-dimensional biological patterns using iterated weighted kernel functions. *PLoS ONE*, 3(3), 2008.
[17] S. Rojas-Galeano and N. Rodriguez. A memory efficient and continuous-valued compact EDA for large scale problems. In *Proceedings of GECCO 2012*, pages 281–288, NY, USA, 2012. ACM.
[18] Y. Saeys, I. n. Inza, and P. Larrañaga. A review of feature selection techniques in bioinformatics. *Bioinformatics*, 23(19):2507–2517, Sept. 2007.
[19] M. Sebag and A. Ducoulombier. Extending Population-Based incremental learning to continuous search spaces. *Lecture Notes in Computer Science*, 1498, 1998.

Appendix. Download and installation

`Goldenberry` is hosted publicly under a GPL license in Codeplex (`http://goldenberry.codeplex.com`). To install the software follow these steps:

1. Download and install Orange 2.6.1 from: `http://orange.biolab.si/download/`.
2. Get the latest `Goldenberry` release from the `Downloads` tab in `http://goldenberry.codeplex.com` (the download link is located to the left side of the screen).
3. Follow the installation instructions in the release notes just below the download link.
4. Open the `Orange` application and look for the "Optimization" toolbar in the canvas (as shown in Figure 1).