Alma Mater Studiorum · Università di Bologna

Physics and Astronomy Department
PhD Thesis in Applied Physics

Implementation and optimization of algorithms in Biological Big Data Analytics

Supervisor:

Prof. Daniel Remondini

Correlator:

Prof. Gastone Castellani
Prof. Armando Bazzani

Presented by:

Nico Curti

Session 2019/2020

""*No one know nothing,*
everyone know something,
but something is nothing to someone,
while
something is important to everybody"

Daudi, Manyara

Abstract

Contents

Introduction

in questo lavoro si affronteranno diverse tematiche relative alla Big Data Analytics e si propongono soluzioni inerenti ad ognuna di esse con esempi sviluppati ed applicati a dati reali. Partendo dalla curse of dimensionality e la feature extraction (dnet), passando per la visualizzazione dei dati con le NN fino alla eterogeneità dei dati (chimera)

definire feature come variable e dire che nel resto del testo verranno usati in maniera indistinta i due termini

Chapter 1

Feature Selection - DNetPRO algorithm

After the end of the Human Genome Project (HGP, 2003) [18] there has been growing interest on biological data and their analysis. At the same time, the availability of this type of data increased exponentially with the technological improvement of data extractors (High-Throughput technologies) [22] and with lower production costs. Lower costs and efficiency in time extraction are the main factors that allow us to go into the new scientific era of Big Data. Biological Big Data works with very large and complex datasets which are typically impossible to store, handle and analyze using standard computer and techniques [16]. Just think that we need around 140 Gb for the storage of the DNA of a single person and an Array Express, a compendium of public gene expression data, contains more than 1.3 million of genomes which have been collected in more than 45000 experiments [10]. Since the number of available data is getting greater, we need to design several storage databases to organize, classify and moreover to extract informations from them. The Bioinformatics European Institute (EBI) at Hinxton (UK), which is part of the European Laboratory of Biological Molecular and one of the biggest repositories of biological data, stores 20 petabytes of data and genomics and proteomics back-ups. The amount of the genomics data is only 2 petabytes, and it doubles every year: it is not worth to remark that these quantities represent about a tenth of data stored by CERN of Ginevra [17]. On the other hand, the ability of processing data and the computational techniques of analysis do not grow the same way. Therefore the gap between the great growth of the number of available data and our ability to work with them is getting bigger.

From a computational point of view, the Bioinformatics new-science is looking for new methods to analyze these large amount of data. The common Machine Learning methods, i.e computational algorithms able to identify significant patterns into large quantities of data, needs to be optimized and modified to increase their computational and statistical performances. To optimize the computational times we need to extend existing methods and algorithms and to develop new dimensionality reduction techniques. In Machine Learning, in fact, as the dimensionality of the data increases, the amount of data required to perform a reliable analysis grows exponentially[1]. The dimensionality reduction techniques are methods able to identify the more significant variables of a given problem or a combination of them, where "significant" means that this smaller number of variables (or features) preserves the information about the problem as much as possible. So this huge amount of high-dimensional omics data (e.g. transcriptomics through microarray or NGS, epigenomics, SNP profiling, proteomics and metabolomics, but also metagenomics of gut microbiota) poses enormous challenges as how to extract useful information from them. One of the prominent problems is to extract low-dimensional sets of variables – sig-

[1] Often this phenomenon is called "curse of dimensionality".

natures – for classification and diagnostic purposes, for example to better stratify patients for personalized intervention strategies based on their molecular profile [23, 5, 14, 2].

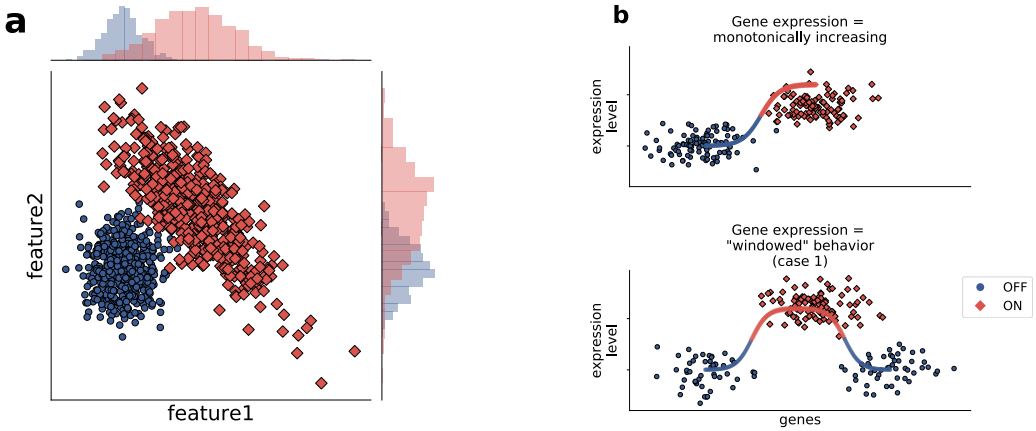

Figure 1.1: (**a**) An example in which single-parameter classification fails in predicting higher-dimension classification performance. Both parameters (*feature1* and *feature2*) badly classify in 1-D, but have a very good performance in 2D. Moreover, classification can be easily interpreted in terms of relative higher/lower expression of both probes. (**b**) Activity of a biological feature (e.g. a gene) as a function of its expression level: top) monotonically increasing, often also discretized to an on/off state; center, bottom) "windowed" behavior, in which there are two or more activity states that do not depend monotonically on expression level. X axis: expression level, Y axis, biological state (arbitrary scales).

Many approaches are used for these classification purposes [11], such as Elastic Net [13], Support Vector Machine, K-nearest Neighbor, Neural networks and Random Forest [20]. Some methods select signature variables by means of single-variable scoring methods [9, 12] (e.g. Student's t test for a two-class comparison), while others search for projections in variable space, and then perform a dimensionality reduction by thresholding the projection weights, but these approaches could fail even in simple two-dimensional situations (Fig. 1.1).

Methods that select variables for multi-dimensional signatures based on single-variable performance can have limits in predicting higher-dimensional signature performance. As shown in Fig. 1.1(a), in which both variables taken singularly perform poorly, but their performance becomes optimal in a 2-dimensional combination, in terms of linear separation of the two classes.

It is known that complex separation surfaces characterize classification tasks associated to image and speech recognition, for which Deep Networks are used successfully in recent times, but in many cases biological data, such as gene or protein expression, are more likely characterized by a up/down-regulation behavior (as shown in Fig. 1.1(b) top), while more complex behaviors (e.g. a "windowed" optimal range of activity, Fig. 1.1(b) bottom) are much less likely. Thus, discriminant-based methods (and logistic regression methods alike) can very likely provide good classification performances in these cases (as demonstrated by our results with DNetPRO) if applied in at least two-dimensional spaces. Moreover, the "linearity" of these methods (that generate very simple class separation surfaces, i.e. linear or quadratic) guarantee that a "buildup" of a signature based on lower-dimensional signatures is feasible.

This consideration are relevant in particular for microarray data where we face on a small number of samples compared to a huge amount of variables (gene probes). This kind of problem, often called "large N, small S" problem (where N is the number of features,

i.e variables, and S is the number of samples), tend to be prone to overfitting[2] and they are classified to ill-posed. The difficulty on the feature extraction can also increase due to noisy variables that can drastically affect the machine learning algorithms. Often is difficult to discriminate between noise and significant variables and even more as the number of variables rises.

In this thesis I propose a new method of features selection - DNetPRO, *Discriminant Analysis with Network PROcessing* - developed to outperform the mentioned above problems. The method is particularly designed to gene-expression data analysis and it was tested against the most common feature selection techniques. The method was already applied on gene-expression datasets but my work focused on the benchmark of it and on its optimization for Big Data applications. The pipeline algorithm is made by many different steps and only a part of it was designed to biological application: this allow me to apply (part of) the same techniques also in different kind of problems with good results (see next sections).

1.1 DNetPRO algorithm

The DNetPRO algorithm generates multivariate signatures starting from all couples of variables tested with Discriminant Analysis. For this reason it can be classified as a combinatorial method and the computational time for variable space exploration is proportional to the square of the number of available variables (ranging from 10^3 to 10^5 in a typical high-throughput omics study). This behavior allows it to overcome some of the limits of single-feature selection methods, and provides a hard-thresholding approach at difference with projection-based variable selection methods. Certainly the combination evaluation is the most time expensive step of the algorithm and it needs accurate algorithmic implementation for Big Data applications (see the next section for further informations about the algorithm implementation strategy). The algorithm can be summarize as shown in 1.

Data: Data matrix (N, S)
Result: List of putative signatures
Divide the data into training and test by an Hold-Out method;
for *couple* $\leftarrow$ *(feature_1, feature_2)* $\in$ *Couples* **do**
    Leave-One-Out cross validation;
    Score estimation using a Classifier;
end
Sorting of the couples in ascending order according to their score;
Threshold over the couples score (Kbest couples);
for *component* $\in$ *connected_components* **do**
    if *reduction* **then**
        Iteratively pendant node remotion;
    else
        S
    end
    ignature evaluation using a Classifier;
end

Algorithm 1: DNetPRO algorithm for Feature Selection.

So, given an initial dataset, consisting in S *samples* (e.g. cells or patients) with N observations each (our *variables*, e.g. gene or protein expression profiles), the signature identification procedure can be summarized with the following pipeline:

[2] A solution to a problem is classified as "overfitted" if small fluctuations on the data variance produce classification errors.

- separation of available data into a training and a test set (e.g. 33/66, or 20/80);
- estimation of the classification performance on the training set of all $S(S-1)/2$ variable couples through a computationally fast and reproducible cross-validation procedure (leave-one-out cross validation was chosen);
- selection of top-performing couples through a hard-thresholding procedure. The performance of each couple constitutes a *weighted link* of a network in which nodes are the variables connected at least through one link;
- every *connected component* in which the network is divided into constitutes an identified classification signature.
- (optional) in order to reduce the size of an identified signature, the pendant nodes of the network (i.e. nodes with degree equal to one) can be removed, in a single step or recursively until the core network (i.e. a network with all nodes with at least two links) is reached.
- all signatures are applied onto the test set to estimate their performance.
- a further cross validation step is performed (with a further dataset splitting into test and validation sets) to identify the best performing signature.

I would stress that this method is completely independent to the chose of the classification algorithm but from a biological point of view a simple one is preferred to preserve an easy interpretability of the results. The geometrical simplicity of the resulting class-separation surfaces, in fact, allows an easier interpretation of the results, as compared with very powerful but black-box methods like nonlinear-kernel SVM or Neural Networks. Moreover the network interaction of variables can keep an internal ranking score of features importance or possible features cooperation. These are the reasons that move us to use very simple classifier methods in our biological application as diag-quadratic Discriminant Analysis or Quadratic Discriminant Analysis (Appendix A for more informations about the mathematical background and implementation in the different languages). Both these methods allow fast computation and easy interpretation of the results. This linear separation might not be common in some classification problems (e.g. image classification) but it is very plausible in biological systems, in which many responses to perturbation consist in increase or decrease of variable values (e.g. expression of genes or proteins, see Fig. 1.1(b)).

In a general classification problem (e.g. image analysis) this could not be the case, since complex non linear separating surfaces may exist among the classes, but we hypothesize (and our results seem to confirm so) that in classification problems based on biological data such as gene expression these situations are not so common. This assumption is very plausible for biological data, since genes are in general up- or down-regulated in order to modify their activity, and protein and metabolites most of the times respond consequently.

A second direct gain by the couples evaluation is related to the network structure: the DNetPRO network signatures allow a hierarchical ranking of the features according to their centrality compared to possible Kbest signatures. This underlying network structure of the signature could suggest further methods for signature dimensionality reduction based on network topological properties to fit real application needs and it could help to evaluate the cooperation of the variables for the class identification.

In the end we remark that the discriminating signatures have a purely statistical relevance, being generated with a purpose of maximal classification performance, but sometimes the selected features (e.g. genes, DNA loci, metabolites) can be of clinical and biological interest, helping to improve knowledge on the mechanism associated to the studied phenomenon [1, 23, 4, 24].

1.2 Synthetic dataset benchmark

We firstly tested the DNetPRO method with synthetic data, consisting in a small set of discriminating variables together with a large number of "noisy" variables. Fixing the number of informative features and classes we test the DNetPRO efficiency on the features extraction, compared to the results obtained by individually single features ranking (*Kbest* feature selection).

To simulate a synthetic "gene expression dataset", with a large number of variables and a much smaller number of samples, we use the toy model generator provided by the *scikit-learn* [21] python package. This model generator allows to set a precise number of classes and it distinguishes between *informative features*, i.e. features which easily separate the class populations, and *redundant features*, i.e. features which represent noise in our problem. The number of informative features should be realistically small compared to the noise, so in our simulations we chose to introduce at least a 10% of informative features in the whole dataset.

1.3 Algorithm implementation

The DNetPRO algorithm is made by a sequence of different steps which have to be performed sequentially for a signature extraction. For this purpose each step can be optimized independently using the full set of available computational resources[3]. In this section will be analyzed each part of the pipeline focusing on the optimization strategies used for the algorithm implementation.

The full code is open source and available at [6]. The code installation is automatically tested using *travis* (for Linux and MacOS environments) and *appveyor* (for Windows environments) at every commit. The installation can be performed using *CMake* or *Makefile* and a full set of installation instructions can be found in the on-line project documentation.

The Python version of the algorithm (see next sections) can be installed via *setup.py* and the compilable parts of it checked via *CMake* or *Makefile*. The Python installer provides also the full list of dependencies of the project which will be automatically installed by the main.

In the Github repository can be found also a full list of example scripts and utilities to obtain the results shown in the next sections. The complete benchmark pipeline can be found which can be run on cluster environment using the SnakeMake version of it (see next section).

1.3.1 Combinatorial algorithm

The most computational time expensive step of the algorithm is certainly the couples evaluation. From a computation point-of-view this step requires $(O(N^2))$ operations for the full set of combination. Since we want to perform also an internal Leave-One-Out cross validation for the couple performances estimation we have to add a $(O(S-1))$ to the algorithmic complexity. Let's focused on some preliminary considerations before the implementation discussion:

- **Performances:** we aim to apply our method on large datasets since we have to focused on time performances of the code and particularly on this step (identified as bottleneck). To reduce as much as possible the call stack inside our code we should perform the entire code with the small number of functions as possible and possibly inside a unique main. Moreover we can simplify the for loop and take care of the

[3] Further optimization can be performed in a cross validation environment and they will be discussed later in this section.

automatic code vectorization performed by the optimizer at compile time (SIMD, *Single Instruction Multiple Data*). A further optimization step to take in count is related to the cache accesses: the use of custom objects inside the code should benefit from cache accesses (AoS vs SoA, *Array of Structure* vs *Structure of Arrays*).

- **Interdependence:** the variable couple performances evaluation is a completely independent computational process and can be faced on as N^2 separately tasks. Thus it can be easily parallelizable to increase speed performance.
- **Simplify:** the use of simple classifier for performance evaluation simplify the computation and the storage of the relevant statistical quantities. In the discussed implementation we focused on a Diag-Quadratic classifier (see Appendix A for further informations) and only means and variances of the data plays a role in its evaluation.
- **Cross Validation:** the use of Leave-One-Out cross validation allows to perform substantially optimizations in the statistical quantities evaluations across the folds (see discussion in Appendix A - Numerical Implementation).
- **Numerical stability:** we have also to take in care the numerical stability of the statistics since we are working in the assumption of a reasonable small number of samples compared to the amount of variables. This behavior particularly affects the variance estimation: the chose of a numerical stable formula for this quantity play a crucial role for the computation because the classifier score has to be normalized by it.

With these idea in mind we can write a C++ code able to optimize this step of computation in a multi-threading environment with the purpose of testing its scalability over multi-core machines.

Starting from the first discussed point we chose to implement the full code inside a unique main function with the help of only a single SoA custom object and one external function (*sorting algorithm* discussed in the next section). This allows us to implement the code inside a single parallel section reducing the time of thread spawns. We chose to import the data from file in sequential mode since the I/O is not affected by parallel optimizations.

Following the instructions suggested in Appendix A - Numerical Implementation we compute the statistic quantities on the full set of data before starting the couples evaluation. Taking a look to the variance equation

$$\sigma^2 = \frac{\sum_{i=1}^{S}(x_i - \mu)^2}{S-1} = \frac{\sum_{i=1}^{S}({x_i}^2)}{S-1} - \mu^2$$

we can see that the first equation involve the computation of the mean as a simple sum of the elements but a large number of subtractions from it that are numerical unstable for data outliers (moreover because they are elevated to square). The better choice in this case is given by the second formulation that allows us to compute the both quantities in the formula inside a single parallel loop[4]. At each cross validation we will use the two precalculated sums of variables removing the only data point excluded by the Leave-One-Out. Another precaution to take in care is to add a small epsilon to the variance before its use at denominator inside the classifier function to prevent numerical underflow.

The main role is still given by the couples loop. The set of pair variables can be obtained only by two nested for loops in C++ and naive optimization can be obtained by

[4] To facilitate the SIMD optimization the code is written using only float (single precision) and integer variables. This precaution takes in care the register alignment inside the loops and facilitate the compile time optimizer.

simply reduce the number of iterations following the triangular indexes of the full matrix (by definition the score of the couple (i, j) is equal to the score of (j, i)). This precaution easily allows the parallelization of the external loop and drastically reduce the number of iteration but it also creates a link between the two iteration variables. The new release of OpenMP libraries [8][5] (from OpenMP 4.5) introduce a new *keyword* of the language that allows the collapsing of nested for loops in a single one (whose number of iterations is given by the product of the single dimensions) in the only exception of completely independences of iteration variables. So the best strategy to use in this case is to perform the full set of N^2 iterations with a single collapse clause in the external loop[6].

Listing 1.1: Python parallel couples evaluation algorithm

```
import pandas as pd
import itertools
import multiprocessing

from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import LeaveOneOut, cross_val_score

def couple_evaluation (pair):
  f1, f2 = pair
  samples = data.iloc[[f1, f2]]
  score = cross_val_score(GaussianNB(), samples.T, labels,
                          cv=LeaveOneOut(), n_jobs=1).mean() # nested
    parallel loops are not allowed

  return (f1, f2, score)

def read_data (filename):
  data = pd.read_csv(filename, sep='\t', header=0)
  labels = data.columns.astype('float').astype('int')
  data.columns = labels

  return (data, labels)

if __name__ == '__main__':

  filename = 'data.txt'

  global data, labels
  data, labels = read_data(filename)

  Nfeature, Nsample = data.shape

  couples = itertools.combinations(range(0, Nfeature), 2)

  nth = multiprocessing.cpu_count()

  with multiprocessing.Pool(nth) as pool:
    score = zip(*pool.map(couple_evaluation, couples))
```

In this section we also provide an "equivalent" Python implementation with the use of common machine learning libraries and parallel settings (ref. 1.1). In the next sections we will discuss the computational performances of this naive implementation with C++ one discussed above.

[5] The OpenMP library is the most common non-standard library for C++ multi-threading applications.
[6] Obviously the iteration where the inner loop variable is lower than the outer one will be skipped by an if condition.

1.3.2 Pair sort

The sorting algorithm starts at the end of variable couple evaluation and re-order the pairs in ascending order to ease the next steps of signature identification[7]. This step is performed in the same code (and same parallel section) of the before section but it deserves an own topic for a better focus on the parallelization strategy chosen. Moreover there are many common parallel implementation of sorting algorithm and to reach the best performances we have to chose the appropriated one.

The sorting algorithm are already implemented in serial version in the major part of the languages (Python and C++ included). The naive version of the algorithms are also quite optimized and they perform the computation with complexity $(O(N\dot{\log}(N)))$[8]. In this case we have not to re-invent any sorting technique but only insert as well as possible these algorithms inside a parallel sections and use the variable format chosen for couple performances storage. Since we are working with SoA objects we need to re-order all the structure arrays in the same way. So we can not use the a simple sort function but can compute the set of indexes that allow the re-order of the arrays, the so called argsort method. To rearrange the indexes according to a given array of values we can use the templates in C++.

As parallelization strategy we can yet invoke the new *keywords* of OpenMP libraries and apply a *divide-and-conquer* architecture using a tree of independent *tasks*[9]. Using the maximum power of two of the available threads we split the computation in equal size sub-arrays and perform independent argsorts. Then, going backwards to the subdivisions at each step we merge the sub-arrays two-by-two until the root.

1.3.3 Network signature

After the rearrangement of feature pairs in ascending order we can start to create the variable network and looking for its connected components as putative signatures. Each feature will be represent a node in the network and a given pair will be a connection between them. Since the full storage of the network would require a matrix $(N \times N)$ we have to chose a better strategy for the processing[10].

The ordered set of couples computes in the previous section represents a so-called *COO sparse matrix* (Coordinate Format sparse matrix) and we can reasonable assume that the desired signature will be composed by the top ranking of them. So, the first step will be to cut a reasonable percentage of the full set of pairs and process only them.

Moreover we are interested in a small set of variables unknown a prior. The load of all the node pairs into the same graph can slow down the computation. An iterative method (with stop criteria) can perform better in the large case of samples and only in worst cases the full set of pair will be loaded.

Since the described algorithm step does not require particular performance efficiency now, the main code used in our simulation was written in pure Python. A C++ implemen-

[7] Up to now we are talking about couples performances meaning the classification accuracy of the feature pair. In some cases the simple accuracy is useless, especially when we are working with unbalanced population classes. In this case we can use a statistical score which takes in count the balanced between the right classifications of our samples (e.g Matthews Correlation coefficient, MCC). The developed code evaluate either the global accuracy of classification either the MCC and, with slight changes allows to perform the re-ordering of feature pair according to the desired score. Since in the next section we will discuss the application of the DNetPRO algorithm to real data using only the classification accuracy as score, we focused only on it in the next sections.

[8] We are considering only un-stable sort in which the preserving order of equivalent elements in the array is not guaranteed.

[9] Tasks in OpenMP are code blocks that the compiler wraps up and makes available to be executed in parallel.

[10] We are working in the hypothesis of very large N.

tation of the same algorithm was developed with the help of the Boost Graph Library [?] (BGL) but to not overweight the code installation was reserved just for a style exercise. In this section we discuss about this second version and about the strategies chosen to implement an efficiency version of it. This version of the algorithm was also used as stand alone method for other applications that will be presented later.

BGL is a very wide framework for graph analysis based on template structures. The library efficiency discourage the users to re-implement the same algorithms and for the current purposes it was resulted more than sufficient.

Starting from the top scorer feature pairs we progressively add each couple of nodes to an empty graph. At each iteration step the number of connected components is evaluated until a desired number of nodes (greater or equal) is not reached[11]. Two degree of freedom are left to the user: in order, pruning and merging. The first one perform an iteratively remotion of nodes with degree equal (or lower) than 1, i.e pendant nodes, until the graph core is not filtered. The merging clause choose between the biggest connected component or the the set of all the disjoint connect components as putative signature. The output of merging step determine the number of nodes in the graph which have been considered for the stop criteria.

A crucial role in the optimization of the algorithm is played by the BGL graph structure chosen. Since the two degree of freedom imply a continuous rearrangement of the graph nodes the strategy chosen is to apply a filter mask over the main graph structure that highlights the only part of interest. This can be done using the boost :: filtered_graph object of BGL. In 1.2 the C++ snippet is shown.

Listing 1.2: DNetPRO signature extraction

```
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/connected_components.hpp>
#include <boost/graph/filtered_graph.hpp>
#include <boost/function.hpp>
#include <boost/graph/iteration_macros.hpp>

typedef typename boost :: adjacency_list< boost :: vecS, boost :: vecS,
    boost :: undirectedS, boost :: property< boost :: vertex_color_t, int
    >, boost :: property < boost :: edge_index_t, int > > Graph;
using V = Graph :: vertex_descriptor;
using Filtered = boost :: filtered_graph < Graph, boost :: keep_all, boost
    :: function < bool(V) > >;


std :: vector < int > FeatureSelection (int ** couples, const int &
    min_size, bool pruning=true,  bool merging=true)
{
  Graph G;
  std :: vector < V > removed_set;
  Filtered Signature (G, boost :: keep_all {}, [] (V v) {return removed_set
    .end() == removed_set.find(v);});

  int L = 0, leave, Ncomp, i = 0;

  while ( true ){

    boost :: add_edge (couples[i][0], couples[i][1], G);

    while ( pruning ){

      leave = 0;
      BGL_FORALL_VERTICES (v, Signature, Filtered);
```

[11] This procedure is quite similar to put a threshold value on the couple performances or just simpler highlight inside the full network the components linked by weights greater than a given value.

```
28          if ( boost :: in_degree (v, Signature) < 2 ){
29            removed_set.insert (v);
30            ++ leave;
31          }
32
33        if ( leave == 0 )
34          break;
35      }
36
37      if ( num_vertices (G) - removed_set.size() ){
38
39        components.resize (num_vertices (G));
40
41        Ncomp = boost :: connected_components (Signature, &components[0]);
42
43        if ( merging ){
44
45          BGL_FORALL_VERTICES (v, Signature, Filtered)
46            core.push_back ( static_cast < int >(v) );
47        }
48        else {
49
50          std :: map < int, int > size;
51          for ( auto && comp : components ) ++ size[comp];
52
53          auto max_key = std :: max_element (std :: begin(size), std :: end(
    size), [] (auto && p1, auto && p2) { return p1.second < p2.second; })->
    first;
54
55          BGL_FORALL_VERTICES (v, Signature, Filtered)
56            if ( components[v] == max_key )
57              core.push_back( static_cast < int >(v) );
58        }
59
60        components.resize (0);
61        L = static_cast < int >(core.size());
62      }
63
64      removed_set.clear();
65
66      if ( L >= min_size ) break;
67
68      ++ i;
69
70      core.resize (0);
71    }
72
73    return core;
74  }
```

From the above description should be clear that given any set of ordered (in ascending order) couples of variables, this algorithm allows to extract the core network independently by the procedure which generate them. So it can be used as dimensionality reduction algorithm of general purpose network structures. An example of this kind of application was reported in Appendix B - Venice Road Network in which we summarized the results of [19, 7].

1.3.4 DNetPRO in Python

Up to now we are focusing on the algorithm performances leaving out the usability of the DNetPRO algorithm for the (research) community. Despite the C++ is one of the most

efficient and old programming language[12], the Python language users are increasing in the last few years. Python is becoming leader in scientific research publications and the large part of Machine Learning analysis are performed using Python libraries (in particular *scikit-learn* library). So we have to reach a compromise between the performances and usability of new developed codes and it can be reached using the Cython [3] framework.

Cython "language"[13] allows an easy interface between C++ codes and Python language. With a relatively simple wrapping of the C++ functions they can be used inside a pure Python code preserving as much as possible the computational performances of the pure C++ version. In this way we can create a simple Python object which performs the full set of DNetPRO steps and moreover which is compatible with the functions provided by the other machine learning libraries.

With this purposes we chose to operate a double wrap of the C++ functions to separate as much as possible the C++ component from the Python one[14]. The Python object was written considering a full compatibility with the *scikit-learn* library to allow the use of the DNetPRO feature selection as an alternative component of other Machine Learning pipelines.

1.3.5 DNetPRO in Snakemake

The starting (silent) hypothesis done until now is that we are running the DNetPRO algorithm on a single dataset (or better on a single Hold-Out subdivision of our data). On this configuration it is legal to stress as much as possible the available computational resources and parallel processing each step of the algorithm.

If we want introduce our algorithm inside a larger pipeline in which we compare the resulting obtained over a Cross-Validation of our datasets we have to re-think about the parallelization done. In this case each fold of the cross validation can be interpreted as independent task and following the main programming rule *"parallelize the outer, vectorize the inner"* we should spawn a thread for each fold and perform the couple evaluation in sequential mode. Certainly, the optimal solution would be to separate our jobs across a wide range of inter-connected computers and still perform the same computation in parallel but it would required to implement our hybrid (C++ and Python) pipeline in a Message Passing Interface (MPI) environment.

An easier solution to overcome all these problems can raise by the use of SnakeMake [15] rules. SnakeMake is an intermediate language between Python and Make. Its syntax is almost like the Make language but with the help of the easier and powerful Python functions. It is wide use for bioinformatic pipeline parallelization since it can easily applied over single or multi-cluster environment (master-slave scheme) with a simple change of command line.

So to improve the scalability of our algorithm we implement the benchmark pipeline scheme using Snakemake rules and a work-flow example for a single cross-validation is

[12] Still in common use in scientific research groups.

[13] It is not a real programming language since it is based on Python. However it has its own syntax and keywords which are different either from Python either from C++. In the end it needs a compiler to run and it is certainly different from Python.

[14] Cython wrap are very powerful tools for C++ integration into Python code but, by experience, they are difficult to manage by pure-Python users. A simple workaround is to perform a first wrap of the C++ function inside a Cython object and a second wrap of it into a pure-Python one. This two-steps wrap certainly gets worse the computational performances but it allows a complete separation between the compiled part of the code (Cython) and the interpreted (Python) one. Moreover we can leave back all the checks on input parameters in the C++ version since they will be performed at run time in the Python wrap.

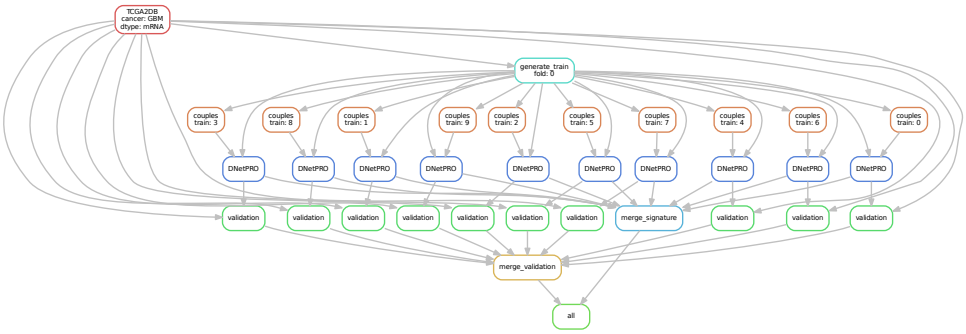

Figure 1.2: Example of DNetPRO pipeline on a single cross validation. It is highlighted the independence of each fold from each other. This scheme shows a possible distribution of the jobs on a multi-threading architecture or for a distributed computing architecture. The second case allows further parallelization scheme (hidden in the graph) for each internal step (e.g. the evaluation of each pair of genes).

shown in Fig. 1.2. In this case each step of Fig. 1.2 can be performed by a different computer unit preserving the multi-threading steps, with a maximum scalability and the possibility to enlarge the problem size and the number of variables.

1.3.6 Time performances

1.4 Benchmark of DNetPRO algorithm

Up to now we have been talked about the DNetPRO algorithm from a theoretical point-of-view. Starting from this section we discuss about the application of the algorithm on real biological datasets (see Appendix B - Venice Road Network for results obtained on a different kind of data).

Despite previous version of the DNetPRO method were already applied on biological data [1, 23, 4, 24] a wide range benchmark of it was still missing. In the following sections we describe the results obtained on the Synapse dataset and published in [**?**].

1.4.1 Synapse dataset

As benchmark dataset was chosen the core sets extracted from the The Cancer Genome Atlas (accession number syn300013, doi:10.7303/syn300013) (*Synapse dataset* in the following), used in a previous study [25] which aimed at quantifying the role of different omics data types (e.g. mRNA and miRNA microarray data, protein levels measured with Reverse Phase Protein Array - RPPA) via different state-of-the-art classification methods. This allowed us to compare our results to a large set of commonly used classification methods, by using their performance validation pipeline (accession number syn1710282, doi:10.7303/syn1710282).

The Synapse dataset is composed by four tumors datasets: kidney renal clear cell carcinoma (KIRC), glioblastoma multiforme (GBM), ovarian serous cystadenocarcinoma (OV) and lung squamous cell carcinoma (LUSC). For each cancer type we applied the DNetPRO algorithm on mRNA, miRNA and RPPA data and we compare the performances results with the Yuan et al. ones.

The summary description of the datasets used is reported in the Tab. 1.1.

Each tumor dataset was pre-processed by adding a zero-mean Gaussian random noise ($\sigma = 10^{-4}$)) to remove the possible null values in the database, which could produce numerical errors in the distances evaluation between genes. Then, we randomly split each

Cancer	mRNA	miRNA	Protein	Number of samples
GBM	AgilentG4502A	H-miRNA_8x15k	RPPA	
	17814	533	[a]	210
KIRC	HiseV2	GA+Hiseq	RPPA	
	20530	1045	166	243
OV	AgilentG4502A	H-miRNA_8x15k	RPPA	
	17814	798	165	379
LUSC	HiseqV2	GA+Hiseq	RPPA	
	20530	1045	174	121

Table 1.1: In the first row platforms are reported and the second shows the dimension of dataset as number of probes. AgilentG4502A: Agilent 244K Custom Gene Expression G4502A; HiseqV2: Illumina HiSeq 2000 RNA Sequencing V2; H-miRNA_8x15K: Agilent 8 × 15K Human miRNA-specific microarray platform; GA+Hiseq: Illumina Genome Analyzer/HiSeq 2000 miRNA sequencing platform; RPPA: MD Anderson reverse phase protein array. The last column shows the number of sample.
[a] Missing data-type for that cancer type.

dataset in training and test sets with a stratified (i.e. balanced for class sample ratio) 10-fold procedure: with the stratification we are reasonably sure that each training-set is a good representative of the whole sample set. The choice of a 10-fold splitting is aimed to reproduce the analysis pipeline presented by Yuan et al. with an analogous cross-validation procedure. Since we don't have exact details of their data splitting, the cross validation was repeated 100 times, for a total of 1000 training procedures for each tumor (OV, LUSC, KIRC, GB) and data type (mRNA, miRNA, RPPA). Each training procedure led to the extraction of multiple signatures.

We chose threshold values in order to obtain a resulting number of variables (network nodes) in the order of $10^2 - 10^3$, and identified all connected components of the network as signatures. If more than one component existed, each one was considered as a different signature.

The final multidimensional signatures were tested by a Discriminant Analysis with a diag-quadratic distance, to avoid possible problems about covariance matrix inversion (as for the Mahalanobis distance).

We remark that DNetPRO can provide more than one signature as a final outcome, given by all the connected components found in the variable network, or a unique top-performing signature can be obtained by a further cross-validation step (procedure A and procedure B in Fig. 1.3, respectively).

In the single cross validation configuration (procedure A in Fig. 1.3), the best signature was extracted as the one reaching the highest accuracy score during the training step. This best signature was then tested over the available test set.

When also the second cross validation was used (procedure B in Fig. 1.3) the best signature wasted as the most performing over a subset of the whole test set (*validation set*), and the final performance was evaluated on the remaining *scoring set*.

To compare our results with the work of Yuan et al., we used the AUC (*Area Under the Curve*) score, that they provided in the paper as the result of their analyses. The distribution of our results could be compared to the single score value given in the other work.

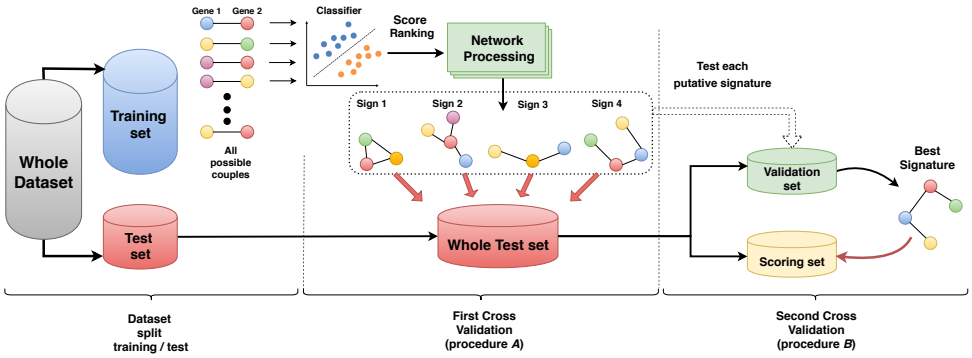

Figure 1.3: Scheme of DNetPRO algorithm. On the "training set", all possible couples of variables are used for Discriminant Analysis, generating the fully connected network weighted by classification performance. Thresholding ranked couples, several signatures can result (as connected components) and their performance is evaluated on the "whole test set" (procedure *A*). A unique best signature can be identified on a "validation set" and tested in a "scoring set", obtained by further splitting the "whole test set" (procedure *B*).

1.4.2 mRNA dataset

1.4.3 miRNA and RPPA dataset

1.4.4 Couple ranking

1.4.5 Characterization of signature overlap

1.5 Cytokinoma dataset

The second application of the DNetPRO algorithm on real biological dataset concerns the

1.6 Bovine Paratuberculosis

Chapter 2

Deep Learning - Neural Network algorithms

Description of the modern deep neural networks. Computational problems and potential applications

Talk about connected layer

2.1 Activation Functions

Activation functions (or transfer functions) are linear or non linear equations which process the output of a neural network neuron and bound it into a limit range of values (commonly $\in [0, 1]$ or $\in [-1, 1]$). The output of simple neuron[1] can be computed as dot product of the input and neuron weights; in this case the output values ranging from $-inf$ to $+inf$ and moreover it is just a simple linear function. Linear functions are very simple to trait but they are limited in their complexity and thus in their learning power. Neural Networks without activation functions are just simple linear regression model. Neural Networks are considered as *Universal Function Approximators* so the introduction of non-linearity allow them to model a wide range of functions and to learn more complex relations in the pattern data. From a biological point of view the activation functions model the on/off state of a neuron in the output decision process.

Many activation functions were proposed during the years and each one has its characteristics but not an appropriate field of application. The better activation function to use in a particular situation (to a particular problem) is still an open question. Each one has its pro and cons in some situations so each Neural Network libraries implements a wide range of them and it leaves to the user to perform his tests. In Tab. **??** we show the list of activation functions implemented in our library with mathematical formulation and its derivative. An important feature of activation function, in fact, is that is should be differentiable since the main procedure of model optimization implies the backpropagation of the error gradients.

As can be shown in Tab. **??** it is easier to compute the activation function derivative as function of it. This is an (well known) important type of optimization in computation term since it reduces the number of operations and it allows to apply the backward gradient directly.

To better understand the effects of activation functions we can perform these functions on a simple test image and comment the results. This can be easy done using the example scripts inserted inside our library[2].

[1] We assume for simplicity a fully connected neural network neuron.

[2] Aware of the author no other example implementations have been done. This makes the NumPyNet library a useful tool for neural network study.

In Fig. **??** the effects of the told above functions are reported on a test image. For each function we show the output of the activation and its gradient. For visualization purposes the image values are rescaled $\in [-1, 1]$ before the input to the functions.

2.2 Convolutional function

2.3 Pooling function

Output Neural Network feature maps often suffer of sensitivity on feature location in the input. One possible approach to overcome this problem is to down sample the feature maps making the resulting feature map more robust to changes in the position. Pooling functions perform this kind of down sample and they reduce the spatial dimension (but not depth) of the input. Their use represents an important computational performance improver tool (less feature, less operations) and a useful dimensionality reduction method. The reduction of feature quantity can also prevent over-fitting problems and improves the classification performances.

Pooling layers are intrinsically related to convolutional layers. The analogy lives in the filter mapping procedure which produces the output in both methods. While in the convolutional layer we map a filter over the input signal and we apply a multiplication of the layer weights and the signal values, in the pooling layer we simply change the filter function keeping the same filter (or *kernel*) mapping procedure (see Convolutional Layer section for more informations). The input parameters of the method are the same of the convolutional one:

- Input: (batch, width, height, channels) of the input data.
- Stride: scalar which control the amount of pixels that the filter slide.
- K: (kx, ky) window filter size.

2.4 BatchNorm function

2.5 Shortcut function

2.6 Route function

2.7 Neural Network laboratory - NumPyNet

Description of the Neural Network laboratory developed in pure numpy. Study of the neural network functionality. Testing of the code against tensorflow.

2.8 Replicated Focusing Belief Propagation

Description of the rFBP library as optimization of the Julia code. Pure c++ implementation with Python wrap (sklearn compatibility). Scorer library as performance evaluation tool with parallel evaluation of scorers.

2.9 Build YouR Own Neural network - Byron library

Limits of the most common neural network frameworks. Neural Network library for parallel computing developed in C++. Pyron as python wrap of the library. Description of the algorithms used to optimize the computation (ex. im2col vs winograd).

2.10 Object Detection - Yolo architecture

Introduction on the image classification and detection with Yolo architecture. Implementation in Byron with description of performances against darknet (original implementation). Focus on performances (time, memory, cpu).

2.11 Super Resolution - WDSR architecture

Introduction on Super Resolution problem with focus on state-of-art neural network architecture. Description of the Byron implementation and application on NMR data with the most common measurements. Super-resolution allows better detection!

2.12 Image Segmentation - UNet architecture

Introduction on Image Segmentation problem. Creation of the datasets with common image-processing methods Application of Unet (Byron implementation) on femur images.

Chapter 3

Biological Big Data - CHIMeRA project

Many public datasets available. Description of the database used in chimera. Problems about the intersections and partial informations (single db).

3.1 Data extraction - Web scraping

Description of the web scraping techinques used to obtain the "no-public" datasets. Reference to the github project.

3.2 The CHIMeRA project

What is CHIMeRA project and which is its potentiality. Description of the database created and of the query implemented to obtain the results

3.3 CHIMeRA query

Some query examples like leukemia subnetwork and PRNP subnetwork. Description of the information extracted by these subnetworks.

Conclusions

Appendix A - Discriminant Analysis

The classification problems aim to associate a set of *pattern* to one or more *classes*. With *pattern* we identify a multidimensional array of data labeled by a pre-determined tag. In this case we talk about *supervised learning*, i.e the full set of data is already annotated and we have prior knowledge about data association to the belonging classes. Since in this work only supervised learning algorithms have been analyzed we do not cite other different learning methods.

In machine learning a key rule is assumed by Bayesian methods, i.e methods which use a Bayesian statistical approach to the analysis of data distributions. It can be proof that if the distributions under analysis are known, i.e a sufficient number of moments of it is known with a sufficient precision, the Bayesian approach is the best possible method to face on the classification problem.

Mathematical formulation

Since the exact knowledge of the prior probabilities and conditional probabilities is possible only on theory a parametric approach is often needed. A parametric approach aim to create reasonable hypothesis about the distribution under analysis and its fundamental parameters (e.g mean and variance). In the next of this discussion we focused only on normal distributions for convenience.

Given the multi-dimensional form of Gauss distribution:

$$G(\mathbf{x}|\mu,\Sigma) = \frac{1}{(2\pi)^{d/2}\cdot|\Sigma|^{1/2}}\cdot exp\left[-\frac{1}{2}(\mathbf{x}-\mu)^T\Sigma^{-1}(\mathbf{x}-\mu)\right]$$

where $\mathbf{x}$ is a column d-dimensional vector, μ the mean vector of the distribution, Σ the covariance matrix $(d\times d)$, $|\Sigma|$ and Σ^{-1} the determinant and the inverse of Σ, respectively, we can notice the G depends quadratically by $\mathbf{x}$,

$$\Delta^2 = (\mathbf{x}-\mu)^T\Sigma^{-1}(\mathbf{x}-\mu)$$

where the exponent (Δ^2) is called Mahalanobis distance of vector $\mathbf{x}$ from its mean. This distance can be reduced to the Euclidean distance when the covariance matrix is the identity $\mathbf{I}$.

The covariance matrix is always symmetric and positive semi-definite (useful information for next algorithmic strategies) so it has an inverse. If the covariance matrix has only diagonal terms the multidimensional distribution can be express as simple product of d mono-dimensional normal distributions. In this case the main axes are parallel to the Cartesian axes.

Starting from the multi-variate Gaussian distribution expression[1], the Bayesian rule for classification problems can be rewrite as:

[1] In Machine Learning it will correspond to the conditional probability density.

$$g_i(\mathbf{x}) = P(w_i|\mathbf{x}) = \frac{p(\mathbf{x}|w_i)P(w_i)}{p(\mathbf{x})} = \frac{1}{(2\pi)^{d/2} \cdot |\Sigma_i|^{1/2}} \cdot exp\left[-\frac{1}{2}(\mathbf{x} - \mu_\mathbf{i})^T {\Sigma_i}^{-1}(\mathbf{x} - \mu_\mathbf{i})\right] \frac{P(w_i)}{p(\mathbf{x})}$$

where, removing constant terms (π factors and absolute probability density $p(\mathbf{x}) = \sum_{i=1}^{s} p(\mathbf{x}|w_i) \cdot P(w_i)$) and using the monotonicity of the function, we can extract the logarithmic relation:

$$g_i(\mathbf{x}) = -\frac{1}{2}(\mathbf{x} - \mu_i)^T {\Sigma_i}^{-1}(\mathbf{x} - \mu_i) - \frac{1}{2}\log|\Sigma_i| + \log P(w_i)$$

which is called Quadratic Discriminant function.

The function dependency by the covariance matrix allows 5 different cases:

- $\Sigma_i = \sigma^2 I$ **- DiagLinear Classifier**

  This is the case of completely independence of features, where they have equal variance for each class. This hypothesis allow us to simplify the discriminant function as:

  $$g_i(\mathbf{x}) = -\frac{1}{2\sigma^2}(\mathbf{x^T x} - 2{\mu_i}^T \mathbf{x} + {\mu_i}^T \mu_i) + \log P(w_i)$$

  and removing all the $\mathbf{x^T x}$ constant terms for each class

  $$g_i(\mathbf{x}) = -\frac{1}{2\sigma^2}(-2{\mu_i}^T \mathbf{x} + {\mu_i}^T \mu_i) + \log P(w_i) = \mathbf{w_i}^\mathbf{T}\mathbf{x} + \mathbf{w_0}$$

  This simplifications create a linear discriminant function where the separation surfaces between classes are hyper-planes ($g_i(\mathbf{x}) = g_j(\mathbf{x})$).

  With equal prior probability the function can be rewritten as

  $$g_i(\mathbf{x}) = -\frac{1}{2\sigma^2}(\mathbf{x} - \mu_i)^T(\mathbf{x} - \mu_i)$$

  which is called *nearest mean classifier* where the equal-probability surfaces are hyper-spheres.

- $\Sigma_i = \Sigma$ **(diagonal matrix) - Linear Classifier**

  In this case the classes have same covariances but each feature has its own different variance. After the Σ substitution in the equation, we obtain

  $$g_i(\mathbf{x}) = -\frac{1}{2}\sum_{k=1}^{s}\frac{(\mathbf{x_k} - \mu_{i,k})^2}{{\sigma_k}^2} - \frac{1}{2}\log\prod_{k=1}^{s}{\sigma_k}^2 + \log P(w_i)$$

  where we can remove constant $\mathbf{x_k}^2$ terms (equals for each class) and obtain another time a linear discriminant function where the discriminant surfaces are hyper-planes and equal-probability boundaries given by hyper-ellipsoids. Note that the only difference from the previous case is the normalization factor of each axes that in this case is given by the its variance.

- $\Sigma_i = \Sigma$ **(non-diagonal matrix) - Mahalanobis Classifier**

  In this case we assume that each class has the same covariance matrix but they are non-diagonal ones. The discriminant function becomes

  $$g_i(\mathbf{x}) = -\frac{1}{2}(\mathbf{x} - \mu_i)^T \Sigma^{-1}(\mathbf{x} - \mu_i) - \frac{1}{2}\log|\Sigma| + \log P(w_i)$$

  where we can remove the $\log|\Sigma|$ term because it is constant for all the classes and we can assume equal prior probability. In this case we obtain

  $$g_i(\mathbf{x}) = -\frac{1}{2}(\mathbf{x} - \mu_i)^T \Sigma^{-1}(\mathbf{x} - \mu_i)$$

  where the quadratic term is the Mahalanobis distance, i.e a normalization of the distance according to the inverse of their covariance matrix. We can proof that expanding the scalar product and removing the constant term $\mathbf{x^T \Sigma^{-1} x}$, we obtain yet a linear discriminant function with the same properties of the previous case. In this case the hyper-ellipsoids have axes aligned according to the eigenvectors of the Σ matrix.

- $\Sigma_i = {\sigma_i}^2 I$ **- DiagQuadratic Classifier**

  In this case we have different covariance matrix for each class but they are proportional to the identity matrix, i.e diagonal matrix. The discriminant function in this case becomes

  $$g_i(\mathbf{x}) = -\frac{1}{2}(\mathbf{x} - \mu_i)^T {\sigma_i}^{-2}(\mathbf{x} - \mu_i) - \frac{1}{2} s \log\left|{\sigma_i}^2\right| + \log P(w_i)$$

  where this expression can be further reduced obtaining a quadratic discriminant function. In this case the equal-probability boundaries are hyper-spheres aligned according to the feature axes.

- $\Sigma_i \neq \Sigma_j$ **(general case) - Quadratic Classifier**

  Starting from the more general discriminant function we can relabel the variables and highlight its quadratic form as

  $$g_i(\mathbf{x}) = \mathbf{x^T W_{2,i} x + w_{1,i}^T x + w_{0,i}} \quad \text{with} \quad \begin{cases} \mathbf{W_{2,i}} = -\frac{1}{2}{\Sigma_i}^{-1} \\ \mathbf{w_{1,i}} = {\Sigma_i}^{-1}\mu_i \\ \mathbf{w_{0,i}} = -\frac{1}{2}{\mu_i}^T {\Sigma_i}^{-1}\mu_i - \frac{1}{2}\log|\Sigma_i| + \log P(w_i) \end{cases}$$

  In this case each class has its own covariance matrix Σ_i and the equal-probability boundaries are hyper-ellipsoids oriented according to the eigenvectors of the covariance matrix of each class.

The Gaussianity of dataset distribution should be tested before using this classifiers. It can be performed using statistical tests as *Malkovic-Afifi* based on *Kolmogorov-Smirnov* index or just simpler with the empirical visualization of the data points.

Numerical Implementation

From a numeric point of view we can exploit each mathematical information and assumption to simplify the computation and improve the numerical stability of our computation. I would remark that this consideration were taken into account in this work only for the C++ algorithmic implementation since these methods are already implemented in the high-level programming languages as *Python* and *Matlab*[2].

In the previous section we highlight that the covariance matrix is a positive semi-definite and symmetric matrix by definition and this properties allows the matrix inversion. The computation of the inverse-matrix is a well known complex computation step from a numerical point-of-view and in a general case can be classified as an $O(N^3)$ algorithm. Moreover the use of a Machine Learning classifier commonly match the use of a cross validation method, i.e multiple subdivision of the dataset in a training and test sets. This involves the computation of multiple inverse matrix and it could represent the performance bottleneck in many cases (the other computations are quite simple and the algorithm complexity is certainly less than $O(N^3)$).

Using the information about the covariance matrix we can find the best mathematical solution for the inverse matrix computation that in this case is given by the Cholesky decomposition algorithm. The Cholesky decomposition or Cholesky factorization allows to re-write a positive-definite matrix into the product of two triangular matrix (the first is the conjugate transpose of the second)

$$\mathbf{A} = \mathbf{L}\mathbf{L}^{\mathbf{T}} = \mathbf{U}^{\mathbf{T}}\mathbf{U}$$

The complexity of the algorithm is the same but the inverse estimation is simpler using a triangular matrix and the entire inversion can be performed in-place. It can also be proof that general inverse matrix algorithms have numerical instability problems compared to the Cholesky decomposition. In this case the original inverse matrix can be computed by the multiplication of the two inverses as

$$\mathbf{A^{-1}} = (\mathbf{L^{-1}})^T(\mathbf{L^{-1}}) = (\mathbf{U^{-1}})(\mathbf{U^{-1}})^T$$

As second bonus, the cross validation methods involve the subdivision of the data in multiple non-independent chunks of the original data. The extreme case of this algorithm is given by the Leave-One-Out cross validation in which the superposition of the data between folds are $N-1$ (where N is the size of the data). The statistical influence of the swapped data is quite low and the covariance matrix will be quite similar between one fold to the other (the inverse matrix will be drastically affected from each slight modification of the original matrix instead). A second step of optimization can be performed computing the original full-covariance matrix of the whole set of data ($O(N^2)$) and at each cross-validation

[2] For completeness we have to highlight that for the Matlab case classification functions, i.e *classify*, is already included in the base packages of the software, i.e no external Toolbox are needed, while for the Python case the most common package which implements these techniques are given by the *scikit-learn* library. Matlab allows to set the classifier type as input parameter in the function using a simple string which follows the same nomenclature previously proposed. Python has a different import for each classifier type: in this case we find correspondence between our nomenclature and the Python one only in *quadratic* and *linear* cases, while the *Mahalanobis* is not considered a putative classifier. The *diagquadratic* classifier is called *GaussianNB* (*Naive Bayes Classifier*) instead. The last important discrepancy between the two language implementation is in the computation of the variance (and the corresponding covariance matrix): Matlab proposes the variance estimation only in relation to the mean so the normalization coefficient is given by the number of sample except by one ($N-1$), while Python compute the variance with a simple normalization by N.

step evaluate the right set of k indexes needed to modify the matrix entrances ($O(N * k)$) that in the Leave-One-Out case are just one. This second optimization consideration can also be performed in the Diag-Quadratic case substituting the covariance matrix with the simpler variance vector.

Both these two techniques were used in the custom C++ implementation of the Quadratic Discriminant Analysis classifier and in the Diag-Quadratic Discriminant Analysis classifier for the DNetPRO algorithm implementation (see 1.1).

Appendix B - Venice Road Network

Appendix N - Bioinformatic Pipeline Profiling

Bibliography

[1] A. Battle, S. Mostafavi, X. Zhu, J. B. Potash, M. M. Weissman, C. McCormick, and D. ... Koller. Characterizing the genetic basis of transcriptome diversity through rna-sequencing of 922 individuals. *Genome Research*, 2014.

[2] J. S. Beckmann and D. A. Lew. Reconciling evidence-based medicine and precision medicine in the era of big data: challenges and opportunities. In *Genome Medicine*, 2016.

[3] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. Seljebotn, and K. Smith. Cython: The best of both worlds. *Computing in Science Engineering*, 13(2):31 –39, 2011.

[4] C. Cenik, E. S. Cenik, G. W. Byeon, F. Grubert, S. I. Candille, D. Spacek, and M. P. ... Snyder. Integrative analysis of rna, translation, and protein levels reveals distinct regulatory variation across humans. *Genome Research*, 2015.

[5] I. S. Chan and G. S. Ginsburg. Personalized medicine: Progress and promise. *Annual Review of Genomics and Human Genetics*, 12(1):217–244, 2011. PMID: 21721939.

[6] N. Curti. DNetPRO pipeline: Implementation of the dnetpro pipeline for tcga datasets. https://github.com/Nico-Curti/DNetPRO, 2019.

[7] N. Curti, E. Giampieri, C. Mizzi, A. Fabbri, A. Bazzani, G. Castellani, and D. Remondini. A network approach for dimensionality reduction from high-throughput data. 2019.

[8] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[9] L. Eckhard. A universal selection method in linear regression models. *Open Journal of Statistics*, 2, 2012.

[10] C. Greene, J. Tan, M. Ung, J. Moore, and C. Cheng. Big data bioinformatics. *Journal of cellular physiology*, 229(12), 2014.

[11] I. Guyon, J. Weston, S. Barnhill, and V. Vapnik. Gene selection for cancer classification using support vector machines. *Mach. Learn.*, 2002.

[12] R. R. Hocking. A biometrics invited paper. the analysis and selection of variables in linear regression. *Biometrics*, 32(1):1–49, 1976.

[13] J. J. Hughey and A. J. Butte. Robust meta-analysis of gene expression using the elastic net. *Nucleic Acids Research*, 2015.

[14] T. M. Johnson. Perspective on precision medicine in oncology. *Pharmacotherapy: The Journal of Human Pharmacology and Drug Therapy*, 37(9):988–989, 2017.

[15] J. Koster and S. Rahmann. Snakemake - a scalable bioinformatics workflow engine. *Bioinformatics*, 28:2520–2522, 08 2012.

[16] D. Kumari and R. Kumar. Impact of biological big data in bioinformatics. *International Journal of Computer Applications*, 101(11):22–24, 2014.

[17] V. Marx. The big challenges of big data. *Nature Reviews*, 498(255), 2013.

[18] M. McKinney. Human genome project information. *Reference Reviews*, 26(3):38–39, 2012.

[19] C. Mizzi, A. Fabbri, S. Rambaldi, F. Bertini, N. Curti, S. Sinigardi, R. Luzi, G. Venturi, M. Davide, G. Muratore, A. Vannelli, and A. Bazzani. Unraveling pedestrian mobility on a road network using icts data during great tourist events. *EPJ Data Science*, 7(1):44, Oct 2018.

[20] H. Pang, S. L. George, K. Hui, and T. Tong. Gene selection using iterative feature elimination random forests for survival outcomes. *IEEE/ACM Transactions on Computational Biology and Bioinformatics / IEEE*, 2012.

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

[22] J. A. Reuter, D. V. Spacek, and M. P. Snyder. High-throughput sequencing technologies. *Molecular Cell*, 2015.

[23] K. Scotlandi, D. Remondini, G. Castellani, M. C. Manara, F. Nardi, L. Cantiani, M. Francesconi, M. Mercuri, A. M. Caccuri, M. Serra, S. Knuutila, and P. Picci. Overcoming resistance to conventional drugs in ewing sarcoma and identification of molecular predictors of outcome. *Journal of Clinical Oncology*, 27(13):2209–2216, 2009. PMID: 19307502.

[24] C. Terragna, D. Remondini, M. Martello, E. Zamagni, L. Pantani, F. Patriarca, A. Pezzi, G. Levi, M. Offidani, I. Proserpio, G. D. S. P. Tacchetti, C. Cangialosi, F. Ciambelli, C. V. Viganò, F. A. Dico, B. Santacroce, E. Borsi, A. Brioli, G. Marzocchi, G. Castellani, G. Martinelli, A. Palumbo, and M. Cavo. The genetic and genomic background of multiple myeloma patients achieving complete response after induction therapy with bortezomib, thalidomide and dexamethasone. *Oncotarget*, 2016.

[25] Y. Yuan, E. M. Van Allen, W. N. Omberg, Larsson, A. Amin-Mansour, A. Sokolov, L. A. Byers, Y. Xu, K. R Hess, L. Diao, L. Han, X. Huang, M. S. Lawrence, J. N. Weinstein, J. M. Stuart, G. B. Mills, and et al. Assessing the clinical utility of cancer genomic and proteomic data across tumor types. *Nature Biotechnology*, 32(7):644–652, 2014.

Acknowledgment