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

Chapter 1

Feature Selection - DNetPRO algorithm

After the end of the Human Genome Project (HGP, 2003) [45] 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) [52] 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 [40]. 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 [30]. 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 [43]. 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 [54, 11, 38, 5].

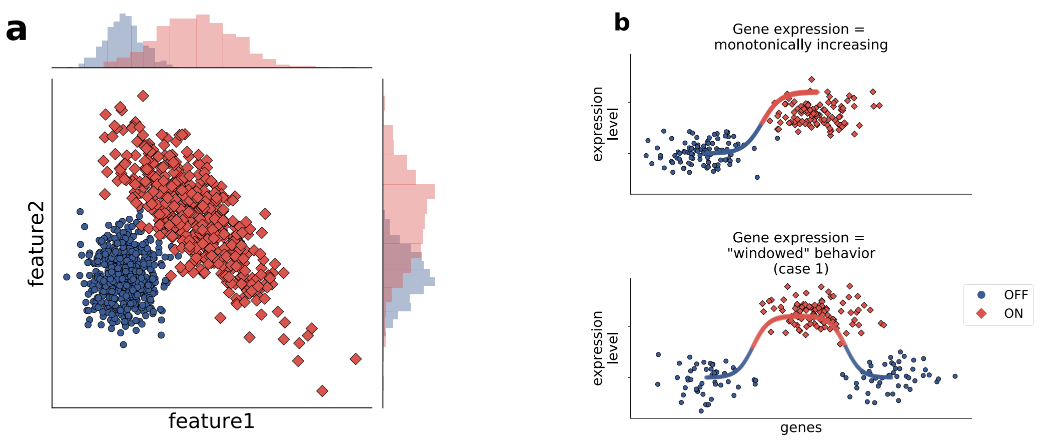

Figure 1.1: (**a**) An example in which single-parameter classification fails in predicting higher-dimension classification performance. Both parameters (*feature1* and *feature2*) badly classify in 1-D, but have a very good performance in 2D. Moreover, classification can be easily interpreted in terms 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 [31], such as Elastic Net [35], Support Vector Machine, K-nearest Neighbor, Neural networks and Random Forest [49]. Some methods select signature variables by means of single-variable scoring methods [28, 33] (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 ← (feature_1, feature_2) ∈ 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 ∈ 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 [4, 54, 9, 59].

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* [51] 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 [14]. 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 pre-calculated 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 [25][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

```python
import pandas as pd
import itertools
import multiprocessing
from functools import partial

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

def couple_evaluation (couple, data, labels):
  f1, f2 = couple

  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'

  data, labels = read_data(filename)

  Nfeature, Nsample = data.shape

  couples = itertools.combinations(range(0, Nfeature), 2)
  couples_eval = partial(couple_evaluation, data=data, labels=labels)

  nth = multiprocessing.cpu_count()

  with multiprocessing.Pool(nth) as pool:
    score = zip(*pool.map(couple_eval, 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 `argsort`s. 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] We are talking about couples performances meaning the classification accuracy of the feature pair up to now. 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 [58] (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

```cpp
#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.

```cpp
        if ( boost :: in_degree (v, Signature) < 2 ){
          removed_set.insert (v);
          ++ leave;
        }

      if ( leave == 0 )
        break;
    }

    if ( num_vertices (G) - removed_set.size() ){

      components.resize (num_vertices (G));

      Ncomp = boost :: connected_components (Signature, &components[0]);

      if ( merging ){

        BGL_FORALL_VERTICES (v, Signature, Filtered)
          core.push_back ( static_cast < int >(v) );
      }
      else {

        std :: map < int, int > size;
        for ( auto && comp : components ) ++ size[comp];

        auto max_key = std :: max_element (std :: begin(size), std :: end(size), [] (auto && p1, auto && p2) { return p1.second < p2.second; })->first;

        BGL_FORALL_VERTICES (v, Signature, Filtered)
          if ( components[v] == max_key )
            core.push_back( static_cast < int >(v) );
      }

      components.resize (0);
      L = static_cast < int >(core.size());
    }

    removed_set.clear();

    if ( L >= min_size ) break;

    ++ i;

    core.resize (0);
  }

  return core;
}
```

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 [46, 24].

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 [7] 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 [39] 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.

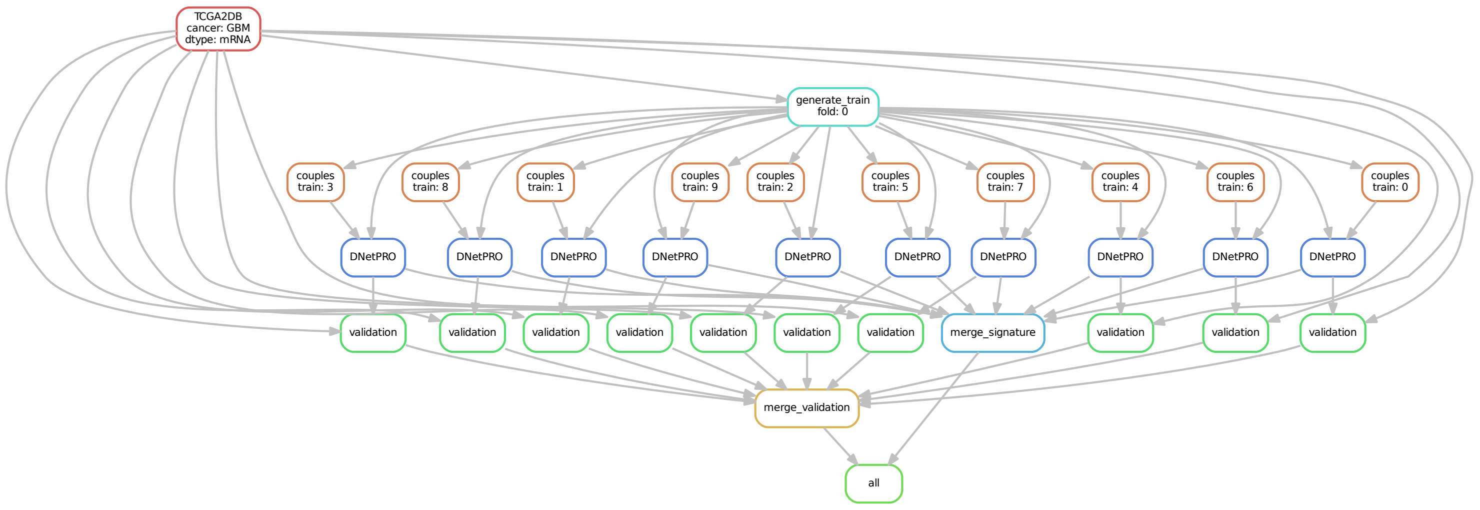

Figure 1.2: Example of DNetPRO pipeline on a single cross validation. It is highlighted the independence of each fold from each other. This scheme shows a possible distribution of the jobs on a multi-threading architecture or for a distributed computing architecture. The second case allows further parallelization 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 [4, 54, 9, 59] 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 [23].

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 [60] 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.

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.

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 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*.

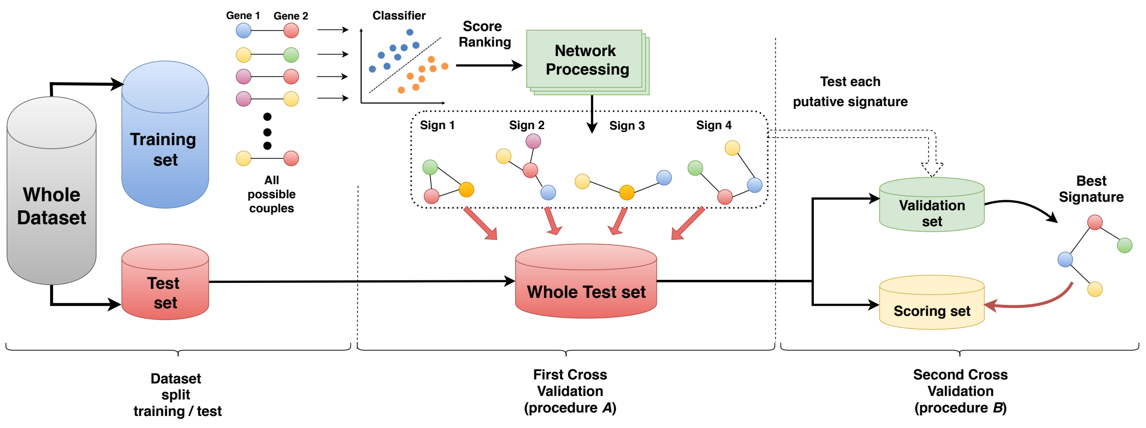

Figure 1.3: Scheme of DNetPRO algorithm. On the "training set", all possible couples of variables are used for Discriminant Analysis, generating the fully connected network weighted by classification performance. Thresholding ranked couples, several signatures can result (as connected components) and 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).

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.

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

Increasing evidence suggests that inflammation is involved in Alzheimer's disease (AD) pathogenesis. Elevated peripheral levels of different cytokines and chemokines in subjects affected by AD compared with healthy control (CTL) have emphasized the role of peripheral inflammation in the disease. Thus, these proteins can represent specific factors of disease development and progression. Considering the cross-talking between the central nervous system and the periphery, the inflammatory analytes may provide utility as biomarkers to identify AD at earlier stages, in particular for the diagnosis of Mild Cognitive Impairment (MCI), a condition at risk of development of dementia. AD is a major neurocognitive disorder and the most common cause of dementia in the old age, accounting for 60% to 80% of all causes. During the past decade, a conceptual shift occurred in the field of AD considering the disease as a continuum. In this context, there is an urgent need for biomarkers identification able to accurately detect AD in an early stage, before the appearance of neurologic signs. An early diagnosis can hopefully lead to a better and more effective treatment, which could potentially limit neuronal damage and prevent the development of overt AD. An emerging field in the study of neuroinflammation is the sex-related differences: in the last years, gender studies have been increasingly developed with the aim to adopt gender differences as a key to interpretation many diseases, including neurodegenerative diseases.

Experimental data showed that many mechanisms are involved in AD pathogenesis including neuroinflammation. The dysregulation of cytokines and chemokines is a central feature in the development of neuroinflammation, neurodegeneration, and demyelination both in the central and peripheral nervous systems. Among many chemokines and cytokines, pro-inflammatory IFNα2, TNFα, and IL-1α are described as heterogeneously implicated in AD pathogenesis.

The interactive network of cytokines/chemokines, defined as "cytokinome", is extremely complex. Using the DNetPRO algorithm as statistical feature selection method, we might discriminate the groups and propose a useful tool to follow the progression and evolution of AD from its early stages, also in light of gender differences. With this study, we aimed first at the identification of a potential proteins profile able to discriminate AD, MCI and CTL and, therefore identify a potential early and easy to get a diagnostic marker of subjects at risk.

1.5.1 Dataset

In this case-control observational study, we evaluated 289 old-age subjects referred to our Geriatric Memory Clinic. The dataset comprises 189 female and 100 male individuals with a mean age of 78.6 ($\pm$7.5) years. The date were provided by the co-authors of this project at the Institute of Gerontology and Geriatrics at the University of Perugia (Department of Medicine). For each patient a set of 26 cytokine expression level were computed with the additional informations about subject set, age and diagnosis label (AD, MCI or CTL). Of the 289 enrolled subjects, the whole set of cytokines was available for 284 subjects (98%), specifically 87/88 CTL (99%), 70/73 MCI (96%), 127/129 AD (98%).

Firstly we applied the DNetPRO algorithm looking for a signature capable of discriminating between CTL and AD: to this purpose, we performed a Hold-Out cross-validation procedure to identify the cytokine signature, considering 2/3 of samples to train the model and then we tested the signature performance on the remaining 1/3 of the total samples. In this analysis we did not separate male from female samples, to avoid the bias given by the uneven number of samples in these two groups, and since previous analysis at a single-cytokine level did not find significant differences due to sex. Then, we classified MCI samples with the CTL-AD signature obtained in the previous step, that allowed labeling MCI samples as CTL or non-CTL.

1.6 Bovine Paratuberculosis

Paratuberculosis or Johne's disease (JD) in cattle is a chronic granulomatous gastroenteritis caused by infection with *Mycobacterium avium subspecies paratuberculosis* (MAP). JD is not treatable; therefore the early identification and isolation of infected animals is a key point to reduce its incidence worldwide. In this work DNetPRO algorithm was applied to RNAseq experimental data of 5 cattle positive to MAP infection compared to 5 negative uninfected controls. The purpose was to find a small set of differentially expressed genes able to discriminate between infected animals in a pre-clinical phase. Results of the DNetPRO algorithm identified a small set of 10 transcripts that differentiate between potentially infected, but clinically healthy, animals belonging to paratuberculosis positive herds and negative unexposed animals. Furthermore, the same set of 10 transcripts differentiate negative unexposed animals from positive animals based on the results of the ELISA test[15] for bovine paratuberculosis and fecal culture. Within the 10 transcripts that together had good

[15] The enzyme-linked immunosorbent assay. It is a common diagnostic tool as well as a quality control check in various bio-medical industries and in medicine.

discriminative potential, 5 (TRPV4, RIC8B, IL5RA, ERF and CDC40) show significant differential expression between the three groups while the remaining 5 transcripts (RDM1, EPHX1, STAU1, TLE1, ASB8) did not show a significant differences in at least one of the pairwise comparisons. In conclusion, the discriminant analysis described here identified a set of 10 genes that discriminate between the exposed and sero-converted animals. When tested in a larger cohort, these finding lead the possible use of RNA expression analysis as new diagnostic test for paratuberculosis. Such a signature could allow early interventions to reduce the sanitary and economic burden, and to reduce the risk of infection spreading.

In the next sections a description of the dataset and of main DNetPRO results will be discussed. Further informations can found in the original paper [42].

1.6.1 Dataset

Paratuberculosis or Johne's disease (JD) in cattle is a chronic granulomatous gastroenteritis caused by infection with *Mycobacterium avium subspecies paratuberculosis* (MAP). JD is present worldwide, is a welfare issue and causes significant economic losses. Cattle are usually infected as young calves but typically do not show clinical signs before 24 months of age, however not all infected animals progress to clinical disease. JD is not treatable, therefore the early identification and isolation of infected animals, before they start shedding the bacteria, is a key point to reduce its incidence in cattle herds worldwide. In addition, an association between MAP and Crohn's disease (CD) in humans has been suggested and intensively explored. Given the economic losses and welfare concerns for livestock, and possible human health risk, the research interest in JD has been driven by the substantial difficulty in early diagnosis of infected animals and the exploration of potentially new diagnostic techniques.

The dataset used in this work was previously discussed and generated by some of the authors of the original paper. In detail, the dataset used comprised 15036 transcripts from 15 samples, classified as "serologically negative non exposed cows/healthy" (5 samples, labeled as NN), "serologically negative exposed cows/ infected" (5 samples, NP) and "serologically positive cows/clinical" (5 samples, PP). Only transcripts with non-zero measures for all samples were considered, reducing the dataset to 13529 transcripts.

All data generated or analyzed during this study is available upon request, furthermore all transcript counts per sample are given as supplementary information files of the original paper.

1.6.2 Bovine Signature

In the context of high-throughput data analysis, a challenge is the search for an optimal choice of variables (a "signature") to classify groups of samples or regress trends with optimal performance and minimum dimensionality. Usually high-throughput omics data (e.g. transcriptomics, ge-nomics, methylomics) provide datasets with few tens to hundreds of samples, and often 1000 times larger numbers of variables. The objective of dimensionality reduction through the choice of an optimal signature is twofold: 1) the identification of relevant variables, that should separate the signal from the noise (i.e. variables not significantly associated to, or descriptive of the studied process); 2) in a practical context, it is important to establish future diagnostic criteria that can be implemented in cheap and simple toolkits, such as PCR cards or dedicated microarray chips, that usually test a small number of transcripts (ranging from tens to hundreds, at most). The quantity of samples compared to the available features of this work, join with the final purposes of this kind of analysis, set the well-known ill-posed problem conditions for which the DNetPRO algorithm was thought.

Since the number of sample is drastically small no robust cross-validation procedure can be applied. So we focused on the identification of a putative gene-signature able to discriminate between NN and NP samples, leaving the PP data as validation set. In this case we hypothesize that PP samples will be classified more closely with NP sample rather than NN as exposed, possibly infected samples, should be more similar to positive samples, than to negative controls.

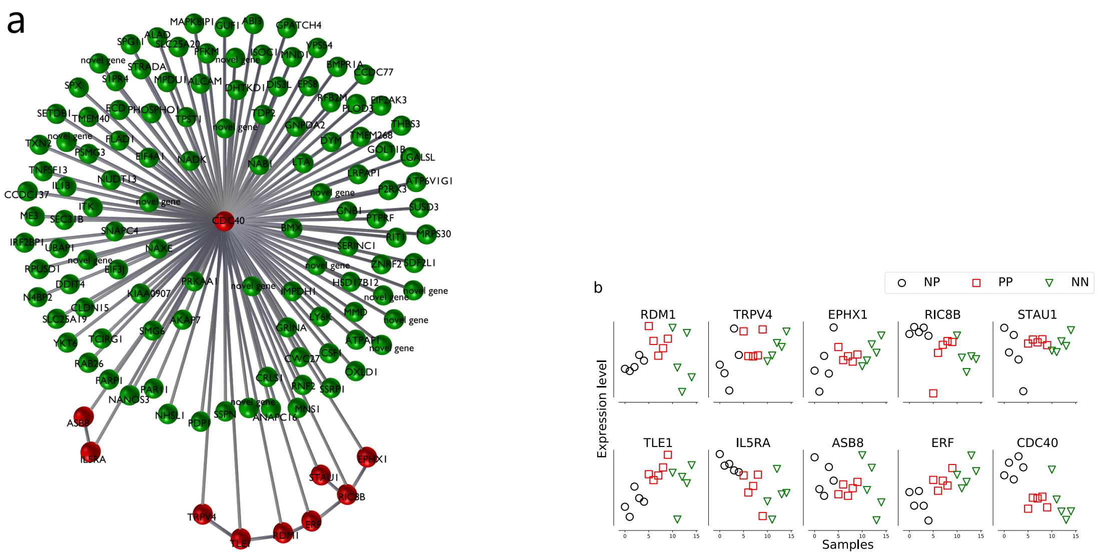

Figure 1.4: (**a**) Plot of the 123-transcript network, with a details of the 10-probe signature (red nodes)[17]. (**b**) Transcript levels for the 10 genes belonging to the classification signature identified by the combinatorial discriminant analysis (CDA). Some transcripts (EPHX1, RIC8B, IL5RA, ERF, CDC40) show a clear trend between 5 animals serologically positive to the ELISA test for MAP (PP), 5 exposed serologically negative (NP) and 5 serologically negative unexposed control animals (NN).

Starting from the top-performing couples of transcripts, we obtained an initial signature of 123 different transcripts (Fig 1.4(a), all the nodes), capable to correctly classify 4 out of the 5 NN samples (80%) and all 5 NP samples (100% performance). The average performance was therefore 90% with Matthews correlation coefficient MCC = 0.82. Processing the 123-transcript network by removing all pendant nodes (i.e. removing all single transcripts belonging to only one best-performing couple) we obtained a final signature with 10 transcripts with a 100% performance classifying all NN and NP samples (Fig 1.4(a), only red nodes). As it can be seen, many nodes are directly connected to the central node (belonging to the 10-transcript signature), while only the 10 transcripts of the signature are also connected between each other.

Principal Component Analysis of the 10-transcript signature showed that in many cases there was a progressive increase or decrease in the transcript levels when passing from a healthy (NN) sample to a positive (PP) sample, passing through the infected (NP) sample class. Fig 1.4(b) shows the expression levels of the transcripts belonging to the signature for all samples.

To further validate the goodness of the signature, we generated 10000 different signatures with 10 randomly chosen transcripts, and then applied a Leave-One-Out cross validation procedure to classify all 15 samples with these signatures. Comparing the performance of the random signatures with the true 10-transcript signature, only 50 of these signatures (corresponding to 0.5% of the random signature distribution) produced better

[17] The figure was generated using a custom network visualizer described in Appendix C - BlendNet.

performance than our signature in terms of classification performance, confirming its high significance.

We even characterized the possible biological role of the signature genes, among the significantly differentially expressed genes, the cell division cycle 40 gene (CDC40) showed the smallest fold change between classes. However in the identified signature the CDC40 gene is the most central node associated with the health status of the animals related to JD. CDC40 was also under expressed in the NP and PP groups, compared with the NN group and it has been shown to be involved in clathrin medated endcytosis from a biological point-of-view. Clathrin is the best characterized coat protein involved in the endocytosis process, specifically in receptor-mediated-internalization. *Mycobacterium paratuberculosis* enters the host macrophages, its primary target cell, and manages to survive within their phagosome. It is possible that the under-expression of CDC40 in infected and sick animals compared to unexposed animals may be associate with down regulation of macrophage genes post mycobacterial invasion, facilitating the survival of the pathogen with the host target cell.

Interestingly within the set of 10 discriminating transcripts, in addition to CDC40, others show links with immune response mechanisms, these include IL5RA, ERF and TRPV4. These genes potentially have functions related to the biology of progression of JD. Also for the other genes of the final 10-transcript signature a possible biological interpretation related to JD was given (see the original paper for further descriptions).

In conclusion, the DNetPRO algorithm identified a set of 10 genes, the expression levels of which could discriminate between the exposed and sero-converted animals. These finding lead the possible use of RNA expression analysis as new diagnostic test for JD. In particular the approach may be able to identify infected animals prior to sero-conversion, prior to a positive ELISA test result. However, further tests for specificity and validation in a larger cohort are required.

Chapter 2

Deep Learning - Neural Network algorithms

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

2.1 Neural Network models

Neural Networks are mathematical models commonly used in data analysis. They are becoming a standard tool in Machine Learning and Deep Learning research and many complex problems can be easily solved by these models. From a theoretical point-of-view we can define a Neural Network as a series of non-linear multi-parametric functions. The model parameters are tuned during a so called *training section* in which we feed our model with a set of data with human supervision, i.e we have prior knowledge about the right and desired output of the model. After the training section we can verify the efficiency of our training using a new set of data, called *test set*, which is never seen by the model. If we have prior knowledge about the output of our test set we can compute the accuracy (or more generally the score) of our model; in the other case we will simply have an extrapolation of our data.

A wide range of documentations and implementations have been written on this topic and it is more and more hard to move around the different sources. Leader on this topic are became the multiple open-source Python libraries available on-line as *Tensorflow* [1], *Pytorch* [50] and *Caffè* [37]. Their portability and efficiency are closely related on the simplicity of the Python language and on the simplicity in writing complex models in a minimum number of code lines. Only a small part of the research community uses more deeper implementation in C++ or other low-level programming languages. About them it should be mentioned the *darknet project* of Redmon J. et al. which created a sort of standard in object detection applications using a pure Ansi-C library.

In this section we firstly retrace the mathematical background of these models. To each theoretical explanation we discuss the numerical problems associated and we provide an efficient custom implementation of each algorithm. The numerical aspects will be traced following two developed custom libraries: NumPyNet library [17] and Byron library [18].

NumPyNet was born as educational framework for the study of Neural Network models. It is written trying to balance code readability and computational performances and it is enriched with a large documentation to better understand the functionality of each script. The library is written in pure Python and the only external library used is Numpy [48] (a based package for the scientific research).

Despite all common libraries are correlated by a wide documentation is often difficult for novel users to move around the many hyper-links and papers cite in it. NumPyNet tries to overcome this problem with a minimal mathematical documentation associated to each script and a wide range of comments inside the code.

An other "problem" to take in count is related to performances. Libraries like *Tensorflow* as certainly efficient by a computational point-of-view and the numerous wrappers (like *Keras* library) guarantees an extremely simple user interface. On the other hand the deeper functionality of the code and the implementation strategies used are unavoidably hidden behind tons of code lines. In this way the user can performs complex computational tasks using the library as black-box package. NumPyNet wants to overcome this problem using simple Python codes with extremely readability also for novel users to better understand the symmetry between mathematical formulas and code.

The simplicity of this library we will allow to give a first numerical analysis of the model functions and, moreover, to show the results of each function to a simple image to better understand the effects of their applications on real data[1]. Each NumPyNet function was tested against the *Tensorflow* implementation of the same methods with an automatic testing routine through *PyTest* [47]. The full code is open-source on the Github page of the project. Its installation is guaranteed by a continuous integration framework of the code through *Travis CI* for Unix environments and *Appevyor CI* for Windows users. The library supports Python version ≥ 2.6[2].

As term of comparison we will discuss the more sophisticated implementations into the Byron library. Byron (Build YouR Own Neural network) library is written in pure C++ with the support of the modern standard 17. We deeply use the c++17 functionality to reach the better performances and flexibility of our code. What makes Byron an efficient alternative to the competition is the complete multi-threading environment in which it works. Despite the most common Neural Network libraries are optimized for GPU environments, there are only few code implementations which exploit the fully functionality of a multiple CPUs architecture. This gap discourage multiple research groups on the use of such computational intensive models in their applications. Byron works in a fully parallel section in which is single computational function is performed using the full set of available cores. To further reduce the time of thread spawn and so optimize as much as possible the code performances, the library works using a single parallel section which is opened at the beginning of the computation and closed at the end[3].

The Byron library is release under MIT license and public available on the Github page of the project. The project also includes a list of common examples like object detection, super resolution, segmentation, ecc. (see the next sections for further details about this models). The library is also completely wrapped using *Cython* to enlarge the range of users also to the Python ones. The complete guide to its installation is provided; it can be done using *CMake*, *Make* or *Docker* and the Python version is available with a simple *setup*. The testing of each function is performed using *Pytest* automatic framework against the NumPyNet implementation (faster and lighter to import than *Tensorflow*).

We will use Byron library as term of comparison with the other common library used in Neural Network models and for each function we test its computational efficiency and scalability on multiple cores. Two machines will be used in the computational testing: a common laptop (8 GB RAM memory and 1 CPU i7-6500U, with 2 cores) and a classical bioinformatics server (128 GB RAM memory and 2 CPU E5-2620, with 8 cores each).

Starting from the next section we introduce the fundamental Neural Network model,

[1] Aware of the author no other example implementations have been done. This makes the NumPyNet library a useful tool for neural network study and a virtual laboratory for new neural network functions.

[2] The library provides also an Image object to load and process images. The object is based on OpenCV API [8]. OpenCV does not yet support Python version 2.7 and 3.3 so the whole NumPyNet package does not work on these two version of Python. You can just exclude the Image scripts from the package or use a novel wrap based on different library (e.g Pillow).

[3] For real-time applications also the time required for the thread spawn must be taken into account.

the so-called *Simple Perceptron*. From the simplest model we will add complexity layers to overcome the relative problems (mathematical and numerical), introducing the main functionality of the modern Neural Network architectures.

2.2 Simple Perceptron

The fundamental unit of each Neural Network model is the *simple Perceptron* (or single neuron). The *Perceptron* it the simpler mathematical model of biological neuron and it is based on the Rosenblatt [53] model which identifies a neuron as a computational unit with input, synaptic weights and an activation threshold (or function). Following the biological model of Hodgkin and Huxley [34] (H-H model), we have an action potential, i.e the output of the neuron, given by

$$y = \sigma\left(\sum_{i=1}^{N} w_i x_i + w_0\right)$$

where σ is the activation function, w_i are the synaptic weights and x_i the inputs. The w_0 coefficient identifies the bias of the linear combination and it is left as parameter to be tune by the optimization algorithm.

The connection weights w_i are tuned during the training section by the chosen updating rule. The standard updating rule is simply given

$$w_i(\tau + 1) = w_i(\tau) + \gamma(t - y)x$$

where γ is the gain or step size ($\gamma \in [0, 1]$) and t is the desired output. In other words we have to firstly compute the difference between the current output and the desired one, i.e the error or cost function or loss function[4], and weight this error by the gain factor and the corresponding input. Repeating the error computation and the updating rule we can bring the weights to convergence. From a geometrical point-of-view this process is equivalent to an hyper-plane placement defined by $w_0 + < w, x >$ which splits an $n-$dimensional space into two half-spaces, i.e two desired classes.

The mathematical formulation already highlights the numerous limits of this model. The output function is a simple linear combination of the input with a vector of weights and so only linearly separable problems can be learned[5] by the *Perceptron*[6]. Moreover we can manage only two classes since an hyper-plane divide the space in only two half-spaces.

A key role is assumed by the activation function. The classical activation function used in the discrete Perceptron model is the *unit step function* (or *Heaviside step function*). If we chose a continuous and so differentiable activation function we can treat the problem using a continuous cost function. In this case we can define it as

$$E(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{N} (t_i - y_i)^2$$

where in this case both t_i and y_i are continuous variables. Now the updating rule can be given by the gradient of the cost function applied to the original weights as

[4] There are multiple loss functions in the Neural Network world. We will further discuss their use and their effective on a learning model in the next section.

[5] A simple mathematical proof of it can be found here.

[6] A classical example of learning problems is given by the XOR logic function. Since the XOR output is not linearly separable the Perceptron could not converge.

$$\mathbf{w} = \mathbf{w} + \Delta\mathbf{w}$$

where $\Delta\mathbf{w}$ is given by

$$\Delta\mathbf{w_i} = -\gamma\frac{\partial E}{\partial w_i} = -\gamma\sum_{i=1}^{N}(t_i - y_i)(-x_i)$$

which looks identical to previous updating rule but in this case are managing real number and not simple class label. Moreover in this way we compute the weight updates according to the full set of training sample and not for each sample (this approach leads to the so-called *batch*-update, i.e small subsets of data).

To implement this kind of model into a pure Python application we do not need extra libraries but we can just use the native keyword of the language. A possible implementation of this model was developed and release in a on-line Gist. In this simple snippet we examine the functionality of the Simple Perceptron model across different logical function and we proof its fast convergence on linear separable datasets[7].

An equivalent C++ implementation of the model is also provided and can be found in this other Gist.

The model is too naive for computational efficiency discussions. Thus we can just observe how a learning algorithm could be easily implemented using basic programming language keywords either in Python either in C++.

2.3 Fully Connected Neural Network

To overcome problems arising from the Simple Perceptron model we can join together multiple Perceptron units into a more complex network of interaction in which the output of a neuron feed-forward the input of the next one. This is the Multi-layers Perceptron (MLP) configuration and if the graph is fully connected, i.e each neuron is connected to all the others, we talk about *fully connected neural networks* (or *dense* neural network, DNN).

Given the Perceptron formulas, the extrapolation to the MLP architecture is straightforward and given by

$$y = \sigma(X \cdot W + W_0)$$

where we simply pass from the vector formulation to the matrix one. The updating rule consequentially becomes

$$W(\tau + 1) = W(\tau) + \gamma X^T(T - Y)$$

where also in this case we simply pass to the matrix formalism. From the re-iteration of such structures we can join together multiple fully connected layers and so obtain multiple neuron layers jointly together with different levels of complexity and units (an input layer followed by multiple *hidden* layers).

The fully connected Neural Networks overcome the told above *Perceptron* problems using a combination of linear functions (single *Perceptron* units) and they gain more useful properties:

[7] The proof the non-linear separable convergence introducing an extra stop criteria during the weights tuning given by a maximum number of step.

- If the activation functions of *all* the hidden units in the Neural Network are linear, then the network architecture is equivalent to a network without hidden units.
- If the number of hidden units is smaller than either the number of input units either the number of output ones, then the network can generate transformations from inputs to outputs as much general as possible since the information is lost in the dimensionality reduction performed by the hidden units.
- We can find multiple weight configurations, i.e W matrices, which give us the same mapping function from inputs to outputs.

2.3.1 Matrix Product

Despite the mathematical formulation of the model we have to take in count also an efficient implementation. From a numerical point-of-view we can notice that all the computation requires by this kind of Networks (or layer if we consider it into an hybrid Neural Network architecture as we will see in the next sections) can be summarized into the matrix product evaluation. The matrix product is a well-known numerical problems and the complexity of the algorithm can be hardly reduced under $O(N^3)$[8]. A crucial role on this kind of algorithms is played by the cache accesses. The CPU cache is the hardware cache used by the CPU to store small portion of data in order to reduce the average cost (in time or energy consumption) to data access from the main memory. Cache optimization in writing an algorithm is one of the most difficult parts to perform but can lead to higher performance gains.

In the matrix product we have to multiply each row of the previous matrix by each column of the second one. We work in the assumption that each matrix is stored into an array of 1D or 2D without nested structures. In this case we can access to a contiguous memory portion of the first matrix since each row will be given by a series of sequential index locations (the row elements will be given by $x[0], x[1], \ldots, x[N]$). This configuration allows the cache optimization in the access to the first matrix since we can store in the small portion of cache memory a series of row elements and use them in a vectorization environment.

From the second matrix we have to extract the elements from each column. This means that the elements will be given by a discontinuous portion of memories (the column elements will be given by $x[0], x[M], x[2M], \ldots, x[N(M-1)]$). In this case we can not insert a full column into the cache memory and in consequence we will have a *cache-miss* at each iteration[9].

The simple matrix product as given by row-column multiplication is already affected by an intrinsic numerical problem which can drastically affect its performances. The simplest workaround of this problem is to perform a transposition of the second matrix to obtain a row-row matrix product[10]. In this way both matrices can be accessed in a sequential order. The total complexity of the computation increase to $O(N^2)$ (for the matrix transposition, in the better case) $+O(N^3)$ (for matrix product) but the numerical performances increase

[8] The complexity is often given in the assumption of only square matrices ($N \times N$) involved in the computation. For no-square matrix the algorithm complexity is given by the product of the three possible different matrix dimensions involved ($(N \times K) = (N \times M)(M \times K)$ brings to $O(NMK)$ complexity). More sophisticated implementation of the algorithm are able to reduce the algorithm complexity (e.g Strassen algorithm) but neither implementation is able to overcome the $O(N^{2.7})$ complexity up-to-now.

[9] The *cache-miss* happens when a required data can not be found into the cache and so its search has to be done in the main memory (RAM).

[10] In the discussion we have silently ignored the problems of matrix storage and the cache optimization for the resulting matrix accesses but in the above discussion we want to focus only on the main problems raising from the matrix product.

due to the cache-miss minimization[11].

Fallowing back to our Neural Network implementation we can obtain the output values using the above technique. Moreover we can assumes from the beginning that the weight matrix is transpose and so remove the transposition step from the matrix product. This simple (but carefully studied) optimization allows us to obtain numerically better results in the feed-forward evaluation but it paybacks a revision of the standard mathematical formulation a carefully implementation of the code.

In the proposed numerical implementations of this model we implements both the matrix product cases to compare the performance results.

2.4 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[12] can be computed as dot product of the input and neuron weights (see previous section); 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 (see the fully connected Neural Network properties in the previous section). 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. 2.1 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. 2.1 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[13].

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.

[11] The cache memory is a very tight portion of memory and it is impossible to completely remove cache-misses.

[12] We assume for simplicity a fully connected Neural Network neuron.

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

Name	**Equation**	**Derivative**
Linear	$f(x) = x$	$f'(x) = 1$
Logistic	$f(x) = \frac{1}{1+\exp(-x)}$	$f'(x) = (1 - f(x)) * f(x)$
Loggy	$f(x) = \frac{2}{1+\exp(-x)} - 1$	$f'(x) = 2 * (1 - \frac{f(x)+1}{2}) * \frac{f(x)+1}{2}$
Relu	$f(x) = \max(0, x)$	$f'(x) = \begin{cases} 1 & \text{if} x > 0 \\ 0 & \text{if} x \leq 0 \end{cases}$
Elu	$f(x) = \max(\exp(x) - 1, x)$	$f'(x) = \begin{cases} 1 & \text{if} x \geq 0 \\ x + 1 & \text{if} x < 0 \end{cases}$
Relie	$f(x) =$	$f'(x) = \begin{cases} 1 & \text{if} x > 0 \\ 1e - 2 & \text{if} x \leq 0 \end{cases}$
Ramp	$f(x) =$	$f'(x) = \begin{cases} x + 1 & \text{if} x > 0 \\ x & \text{if} x \leq 0 \end{cases}$
Tanh	$f(x) = \tanh(x)$	$f'(x) = 1 - x^2$
Plse	$f(x) =$	$f'(x) =$
Leaky	$f(x) =$	$f'(x) = \begin{cases} 1 & \text{if} x > 0 \\ C & \text{if} x \leq 0 \end{cases}$
HardTan	$f(x) =$	$f'(x) =$
LhTan	$f(x) =$	$f'(x) =$
Selu	$f(x) =$	$f'(x) =$
Swish	$f(x) =$	$f'(x) =$
SoftMax	$f(x) = \frac{\exp(x)}{\sum_{i=1}^{N} x}$	$f'(x) =$

Table 2.1: List of common activation functions with correspondig mathematical equation and derivative. The derivative is expressed as function of $f(x)$ to optimize their numerical evaluation.

2.4.1 Rectified Linear Unit

The ReLU (Rectified Linear Unit) activation functions are the most used into the modern Neural Networks models. Their diffusion is imputed to their numerical efficiency and to the benefits they bring [29]:

- Information disentangling: the main purpose of a Neural Network model is to tune a discriminant function able to associate a set of input to a prior-known output classes. A dense information representation is considered *entangled* because small differences in input highly modifies the data representation inside the network. On the other hand, a sparse representation tends to guarantee a conservation of the learning features.

- Information representation: different inputs can lead different quantities of useful informations. The possibility to have null values in output (ref Tab. 2.1) allows a better representation of the representation dimension inside the network.

- Sparsity: sparsity representation of data are exponentially efficient in comparison to dense ones, where the exponential power is given by the number of no-null features [29].

- Vanish gradient reduction: if the activation output is positive we have a no-bound gradient value.

2.5 Dropout function

Many times along this work we have been talked about the *over-fitting* problem. The over-fitting problems arise when the complexity of our model becomes too high regard the amount of available data, i.e when the number of parameters of our model is comparable to the number of available data. A classical example is given by the polynomial fitting problem. Given an initial set of N data points we can always find a polynomial curve of degree equal to $N - 1$ which can perfectly fit our data. In this case the model flexibility is minimum and new additional data points difficulty lies on the same curve. In other words we tuned each model parameter according to the given data set but we completely lose the possibility of generalization.

In Neural Network models we have to manage a large quantities of parameters and it is quite easy to stumble on this problem. A possible workaround is given by a Dropout function. This function simply dropping out some neuron units into a neural network during the training phase. Ignoring some neurons means that they will not be considered during a particular (single) forward/backward step. So, given a set of neuron we have a probability p to keep the neuron and $1 - p$ to remove it. In this way we can reduce the co-dependency of nearest neurons inside the network and so reduce the possibility of over-fitting. An other common way is to regularize the neuron outputs with penalty loss function[14].

2.5.1 Numerical Implementation

The above description bring us to a straight forward implementation of the algorithm into the NumPyNet library.

[14] Classical examples are given by the L1 (Laplacian) and L2 (Gaussian) penalties. Both these functions are implemented either in NumPyNet and Byron but for sake of brevity we will not discuss about them.

Listing 2.1: NumPyNet version of Dropout function

```python
import numpy as np

class Dropout_layer(object):

  def __init__(self, prob, **kwargs):

    self.probability = prob
    self.scale = 1. / (1. - prob)

    self.batch, self.w, self.h, self.c = (0, 0, 0, 0)

    self.output, self.delta = (None, None)

  def __str__(self):
    batch, out_width, out_height, out_channels = self.out_shape()
    return 'Dropout       p = {:.2f}        {:4d}, {:4d}, {:4d}, {:4d}  ->
     {:4d}, {:4d}, {:4d}, {:4d}'.format(
           self.probability,
           batch, out_width , out_height , out_channels,
           batch, out_width , out_height , out_channels)

  def out_shape(self):
    return (self.batch, self.w, self.h, self.c)

  def forward(self, inpt):

    self.batch, self.w, self.h, self.c = inpt.shape

    self.output = inpt.copy()

    self.rnd = np.random.uniform(low=0., high=1., size=self.output.shape) <
     self.probability
    self.output[self.rnd] = 0.
    self.rnd = ~self.rnd
    self.output[self.rnd] *= self.scale

  def backward(self, delta=None):

    if delta is not None:
      delta[self.rnd] *= self.scale
      self.rnd = ~self.rnd
      delta[self.rnd] = 0.
```

The above code numerically reproduce the theoretical formulation given. After the initialization of the private object variables the forward function generate a set of random positions and apply them (if they are less than the given probability) to the output: these positions will be turned off and the others will be multiply by a scale probability factor to increase their importances. The backward function simply invert the transformation on the back-propagated gradient `delta`.

Despite this straight forward implementation we have to carefully manage some crucial points into the C++ equivalent. We are working on a complete parallel region so after the (sequential) initialization initialization of the layer object the forward/backward phases are evaluated by all the available threads in parallel. This bring us to a standard problem in multi-threading programming: the generation of independent random numbers among threads. Inside a parallel region all the variables declared are (by definition) shared among threads and so if we simply create a random number generator in it we have to face on the thread-concurrency. As consequence the random number generated will not be independent but (most probably) repeated by each thread[15]. The simple workaround implemented into

[15] The deterministic generation of random number is hard to reproduce into a parallel environments

Byron library is given by assigning a random number generator to each thread (with its own seed and indexed by the thread Id). In this way we can ensure a totally independence of the random numbers generated during the forward phase (ref. on-line).

As visualization example we can use a simple test image and apply our transformation (see Fig.**??**). As expected our input image shows many pixel turned off according to the given probability.

2.6 Convolutional function

Convolutional Neural Network (CNN) are particularly designed for image analysis. Convolution is the mathematical integration of two functions in which the second one is translated by a given value.

In signal processing this operation is also called *crossing correlation* ad it is equivalent to the *autocorrelation* function computed in a given point. In image processing the first function is represented by the image I and the second one is a kernel k (or filter) which shift along the image. In this case we will have a 2D discrete version of the formula given by:

$$C = k * I$$

$$C[i,j] = \sum_{u=-N}^{N} \sum_{v=-M}^{M} k[u,v] \cdot I[i-u, j-v]$$

where $C[i,j]$ is the pixel value of the resulting image and N, M are kernel dimensions.

The use of CNN in modern image analysis applications can be traced back to multiple causes. First of all the image dimensions are increasingly bigger and thus the number of variables/features, i.e pixels, is often too big to manage with standard DNN[16]. Moreover if we consider detection problems, i.e the problem of detecting an set of features (or an object) inside a larger pattern, we want a system able to recognize the object regardless of where it appears into the input. In other words, we want that our model would be independent by simple translations.

Both the above problems can overcome by CNN models using a small kernel, i.e weight mask, which maps the full input. A CNN is able to successfully capture the spatial and temporal dependencies in an signal through the application of relevant filters.

2.7 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 it improves the classification performances.

despite the seed initialization. The "probability" of repeat the same sequence is related to the affinity of each thread to the given process.

[16] If we consider a simple image 224×224 with 3 color channels we obtain a set of $150'528$ features. A classical DNN layer with this input size should have 1024 nodes for a total of more than 150 million weights to tune.

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 mapping procedure (see section 2.6 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.

2.8 BatchNorm function

A common practice before the training of a Neural Network model is to apply some pre-processing to the input patterns. A classical example is the normalization of training set, i.e it resembles a normal distribution with zero mean and unitary variance. The initial preprocessing is useful to prevent the early saturation of non-linear activation functions (see section 2.4). Moreover in this case we can ensure that all inputs are in the same range of values.

In a deep neural network architecture we can find the same problem also into the intermediate layers because the distribution of the activations is constantly changing during training. This behavior produces a slowdown in the training convergence because each layer have to adapt itself to a new distribution of data in every training step (or *epoch*). This problem is also called *internal covariate shift*.

A second problem arises from the heterogeneity of available input data. If we tune the model parameters according to a given set of data which inevitably be limited we can meet problems during the generalization, i.e the validation of our model using new data, to new samples if they belongs to an equivalent but deformed distribution. A classical example is given by the image detection: if we train a Neural Network model using gray-scale images we can find generalization problem using colored images.

BatchNorm function (Batch Normalization) allows to overcome these problems with a continuous rescaling of the Neural Network intermediate values during the training[17] [36]. In particular, the method processes the input of a given layer in order to fight the internal covariate shift problem removing the batch mean and normalizing by the batch variance:

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i \qquad {\sigma_B}^2 = \frac{1}{m-1}\sum_{i=1}^{m}(x_i - \mu_B)^2$$

and so the input data becomes:

$$\hat{x_i} = \frac{x_i - \mu_B}{\sqrt{{\sigma_B}^2 + \epsilon}}$$

where we add an extra ϵ in the denominator for numerical stability[18]. After this common rescaling we also a apply a scaling-shift to previous results:

[17] The input data to feed the Neural Network model are commonly packed into a series of *batches*, i.e small subsets of data. The BatchNorm function takes its name from this nomenclature and it processes each batch independently.

[18] The floating point numbers into a computer have finite precision and the variance can underflow bringing to infinite values in the BatchNorm equation.

$$y_i = \gamma \hat{x_i} + \beta$$

where the γ and β coefficients are left as variables to be tuned during the training (they are learned during training). The updating rule of the function parameters (γ and β) is given by the simple derivative of the previous function. In this way we can ensure more stability of the extracted features [41] during the training a faster convergence.

Since the BatchNorm function is became a sort of standard into a deep learning models an efficient implementation of this algorithm is essential to achieve the best computational performances. We have also to take in count that the batch-normalization procedure is commonly performed after a fully-connected layer or a convolutional one. Thus the best performances could be obtained merging the two functionality as much as possible as suggested in [2]. In the next sections we will show our implementation of the algorithm and we discuss about the code optimization performed[19].

2.9 Shortcut function

2.10 Route function

2.11 Pixel Shuffle

Pixel Shuffle layer is one of the most recent layer type introduced in modern deep learning Neural Network. Its application is closely related to the single-image super-resolution (SISR) research, i.e the techniques ensemble which aim at restoring a high-resolution image from a single low-resolution one (see section 2.14 for further details).

The first SISR neural networks start with a pre-processing of low-resolution image in input with a bicubic up-sample. Then the image, with the same dimension of the desired output, feeds the model which aim to increase the resolution and fix its details. In this way the amount of parameters and moreover the computational required by the training section increase (by a factor equal to the square of the desired up-sample scale), despite the image processing required is smaller. To overcome this problem a Pixel Shuffle transformation, also known as *sub-pixel convolution*, was introduced [56]: in this work the authors proofed the equivalence between a regular transpose convolution, i.e the previous standard transformation to enlarge the input dimensions, and the sub-pixel convolution transformation without losing any information. The Pixel Shuffle transformation reorganize the low-resolution image channels to obtain a bigger image with few channels. An example of this transformation is shown in Fig. 2.1.

Pixel Shuffle rearranges the elements of the input tensor expressed as $H \times W \times C^2$ to form a $scale \cdot H \times scale \cdot W \times C$ tensor. This can be very useful after a convolutional process, in which the number of filters chosen drastically increase the number of channels, to "invert" the transformation like a sort of *deconvolution* function.

The main gain in using this transformation is the increment of computational efficiency of Neural Network model. The introduction of Pixel Shuffle transformation in the Neural

[19] The Byron library was inspired by the *darknet* library provided by Redmon J. et al. and by its many branches. Despite in each implementation we can find the BatchNorm function, aware of the author, in any version we can find a right implementation of this function as standalone method. We have already highlighted that this normalization function can be efficiently joined to other function to increase the computational performances but in these case we have to different manage the dimensions of the involved arrays. A standalone implementation of the BatchNorm function required a rearrangement of its functions and it was provided into the Byron library. This was one of the various improvements provided by Byron against the other *darknet*-like libraries.

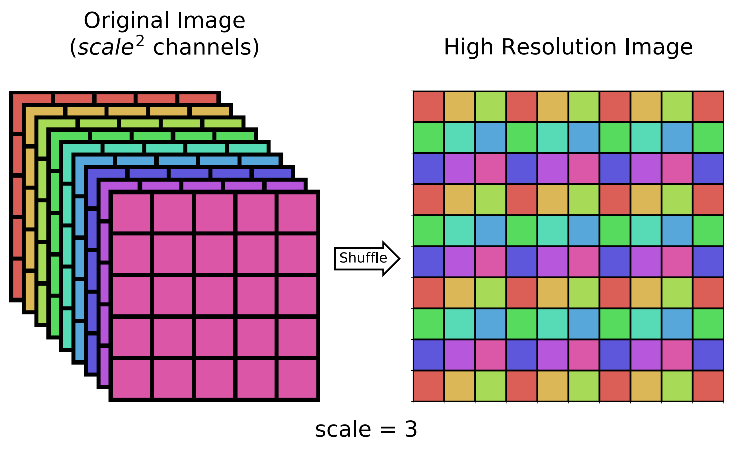

Figure 2.1: Pixel Shuffle transformation. On the left the input image with *scale*^{2} ($:= 9$) channels. On the right the result of Pixel Shuffle transformation. Since the number of channels is perfect square the output is a single channel image with the rearrangement of the original ones.

Network tail, i.e after a sequence of small processing steps which increase the number of features, we can reorganize the set of features into a single bigger image, i.e the desired output in a SISR application. The feature processing steps, which generally are faced on with convolutional layers, can be used with smaller images in input and so can be performed faster since the upscaling task will be performed by a single Pixel Shuffle transformation.

Despite this transformation is became a standard in super-resolution applications and thus it can be found into the most common deep learning libraries (e.g *Pytorch* and *Tensorflow*) a C++ implementation is hard to find. Moreover each library implements the transformation following its own data organization[20]. For this reason we will propose in our libraries a dynamic version of the algorithm in C++ able to perform both version of the algorithm.

2.11.1 Numerical Implementation

The algorithmic implementation of the pixel-shuffle transformation is essentially a re-indexing of the input values. While in a C++ implementation of the algorithm we can obtain the desired result inside a sequence of nested for loops playing with the loop indexes, for an efficient Python version we will use a sequence of transposition and reshaping to rearrange the input values. The following snippet shows the NumPyNet version of this algorithm.

Listing 2.2: NumPyNet version of Pixel-Shuffle function

```python
import numpy as np

class Shuffler_layer(object):

  def __init__(self, scale, **kwargs):

    self.scale = scale
    self.scale_step = scale * scale

    self.batch, self.w, self.h, self.c = (0, 0, 0, 0)
```

[20] The main difference between *Pytorch* and *Tensorflow* is related to the storage organization of the image. *Tensorflow* has a "standard" input assessment as $H \times W \times C$. *Pytorch* has a so-called channel-first implementation and so the input tensor is organized as $C \times H \times W$.

```python

    self.output, self.delta = (None, None)

  def __str__(self):
    batch, out_width, out_height, out_channels = self.out_shape
    return 'Shuffler x {:3d}            {:>4d} x{:>4d} x{:>4d} x{:>4d}   ->  {:>4d} x{:>4d} x{:>4d} x{:>4d}'.format(
           self.scale,
           batch, self.w, self.h, self.c,
           batch, out_width, out_height, out_channels)

  @property
  def out_shape(self):
    return (self.batch, self.w * self.scale, self.h * self.scale, self.c // (self.scale_step))

  def _phase_shift(self, input, scale):
    b, w, h, c = input.shape
    X = input.transpose(1, 2, 3, 0).reshape(w, h, scale, scale, b)
    X = np.concatenate(X, axis=1)
    X = np.concatenate(X, axis=1)
    X = X.transpose(2, 0, 1)
    return np.reshape(X, (b, w * scale, h * scale, 1))

  def _reverse(self, delta, scale):
    # This function apply numpy.split as a reverse function to numpy.concatenate
    # along the same axis also

    delta = delta.transpose(1, 2, 0)

    delta = np.asarray(np.split(delta, self.h, axis=1))
    delta = np.asarray(np.split(delta, self.w, axis=1))
    delta = delta.reshape(self.w, self.h, scale * scale, self.batch)

    # It returns an output of the correct shape (batch, in_w, in_h, scale**2)
    # for the concatenate in the backward function
    return delta.transpose(3, 0, 1, 2)

  def forward(self, input):

    self.batch, self.w, self.h, self.c = input.shape

    channel_output = self.c // self.scale_step # out_c

    # The function phase shift receives only in_c // out_c channels at a time
    # the concatenate stitches together every output of the function.

    self.output = np.concatenate([self._phase_shift(input[:, :, :, range(i, self.c, channel_output)], self.scale)
                                  for i in range(channel_output)], axis=3)

    self.delta = np.zeros(shape=self.out_shape, dtype=float)

  def backward(self, delta):

    channel_out = self.c // self.scale_step # out_c

    # I apply the reverse function only for a single channel
    X = np.concatenate([self._reverse(self.delta[:, :, :, i], self.scale)
                                      for i in range(channel_out)], axis=3)


    # The 'reverse' concatenate actually put the correct channels together
   but in a
    # weird order, so this part sorts the 'layers' correctly
    idx = sum([list(range(i, self.c, channel_out)) for i in range(
   channel_out)], [])
    idx = np.argsort(idx)

    delta[:] = X[:, :, :, idx]
```

The two function _phase_shift and _reverse[21] produce the rearrangement of the indexes according to the pixel-shuffle transformation and its inversion[22] In the forward function we apply the _phase_shift to the sequence of channels (in the right order) and then we concatenate the results into a single tensor (output). The backward function instead needs a reordering of channels sequence after the concatenation.

As told above, in the C++ implementation provided into Byron library we can compute the desired re-indexing using a series of nested for loops. An equivalent solution can be obtained also by the contraction of the loops into a single one using divisions to obtain the right indexes. This solution was taken in count in a first version of the library but the amount of required divisions weight the computational performances. The division operations are the most computationally cost operation in terms of CPU clock time. The old versions of OpenMP multi-threading library forced the users to spend time in the evaluation of the "loop-contraction" to obtain the better performances by a single parallel for loop. The new features of OpenMP library provided the very powerful *collapse* keyword which provides an automatic loop-contraction. The keyword can be applied only with a series of independent and perfectly nested[23] for loops which is exactly our case. Moreover we have not to take in care any thread concurrency trouble since the iteration as the output indexes are totally independent. We largely used the collapse keyword in the Byron library to simplify the codes and the function evaluation but the Pixel-Shuffle case is one of the most efficient one since we have to collapse six nested loops[24] (ref. on-line.

2.12 Cost function

A machine learning algorithm is used to minimize or maximize a function. In other words when we implement a machine learning algorithm we want to know how good is our result according to prior knowledge about the desired results. So we have to establish a function ables to represent the error of our model. This kind of function are commonly called *error functions* or *loss functions* or just simply *cost function*. In the previous sections we have shown many algorithms used into a neural network model and we always talked about how to update the functional parameters according to the evaluated error. This error is provided by the cost function.

The cost function represents the final output of our neural network model so it is reasonable to talk about it at the end of this chapter. There are many kinds of loss functions and there is not a particular one able to works with all kinds of data. So we have to pay attention to chose the right one in our problems. In particular we have to take in count the possible presence of outliers, the structure of our model, the computational efficiency of our algorithm and most of all the number of classes that we want to predict.

[21] These function are "private" function of the object class.

[22] During the back-propagation, in fact, we have to apply the reverse transformation to the gradient.

[23] Two for loops are perfectly nested if there are not other code lines between them.

[24] In the Pixel-Shuffle we have to loop over batch, width, height, channels plus a couple of loops over the scale factor that we want to apply. In total we have to manage six dimensions that can be easily collapsed into a single one given by their product.

Broadly, we can classify the loss functions into two major categories: the classification losses and the regression losses. In the first case we want to predict a finite number of categorical values (classes). In the second case the prediction is performed on a series of continuous values. Since in this work we are focusing only on classification problems we will only talk about the first case.

The most common cost function is given by the *Mean Square Error* (MSE) or *L2 loss.* Its mathematical formulation is quite simple and it is given by

$$MSE = \frac{\sum_{i=1}^{N} (y-t)^2}{N}$$

where we follow the nomenclature given in 2.2 and N is the number of output which is equivalent to the number of classes. It is one of the most used cost function due to its simplicity either from a mathematical either from a numerical point of view. The possible range of values ranged from 0 to ∞. With MSE function predictions which are far away from actual values are heavily penalized, due to the squaring.

A slight different function is given by the *Mean Absolute Error* (MAE) or *L1 loss* in which we replace the squaring with a module of the error.

$$MAE = \frac{\sum_{i=1}^{N} |(y-t)|}{N}$$

With MAE we loose the information about the error direction (preserved by the squaring in MSE) and just simply evaluate the absolute value of the error.

The main differences between these two function can be summarized as follow: using the MSE function we can easily solve the problem but the MAE function is more robust against possible outliers. Despite both function reaches the minimum in a perfect classification configuration (error equal to zero), in presence of outliers we have to manage with large differences in the numerator of the function. With large differences, the square values are greater than the absolute values but while the MSE tries to adjust its performance to minimize those cases, the other common examples pays the higher cost.

A problem related to the MAE function arises during the gradient evaluation. Its gradient, in fact, is the same throughout, which means that we will have large gradient values also with small differences which is a worse configuration during the training. A simple possible workaround is to introduce a shrinking parameters, given by a dynamic learning rate, when we move closer to the minimum.

When we have to manage multi-class problems there are other common cost functions based on likelihood scores. The simpler one is the *Cross Entropy loss* or *Log loss*:

$$CrossEntropyLoss = -(y \cdot \log(t) + (1-y) \cdot \log(1-t))$$

This function just multiply the log of the actual predicted probability by the ground truth class. In this way when we have two classes (e.g $t \in [0,1]$) we can alternatively nullify the two parts of the function[25]. In this way the loss function heavily penalizes the predictions that are confident but wrong. This functions works with binary classification problems where the output classes are binned in $[0,1]$. For this reason the output of the model must be constrained into the $[0,1]$ domain and thus a proper activation function should be provided. Classically this loss function is used jointly to the sigmoid activation (ref. 2.4) which constrains the output of the model in the desired interval. For this reason

[25] When the actual label is equal to 1, i.e $y = 1$, the second half of the Log Loss function disappears whereas in case of actual label is equal to 0 the first half is null.

in our implementation of the algorithm we chose to merge the sigmoid function and the Log Loss function into a single object[26].

A last duty to mention loss function is the extension of the Log loss function to multiple classes, the so-called *Categorical Cross Entropy Loss*.

$$CategoricalCrossEntropyLoss = -\sum i = 1^N \left(y \cdot \log(t)\right)$$

This function generalized the previous one for multiple-classes, i.e for problems where the correct output can be only one. The loss compare the distribution of the predictions, i.e output of the model, with the prior known distribution. In this way only the probability of the true class will be 1 and all the other classes will be set to 0. Also in this case we have to pay attention to the output of our model which is intended as a probability value ranging in $[0, 1]$. In particular this function commonly works jointly to a softmax activation function. As in the previous case we chose to implements this loss function in a separated object associated to the softmax transformation.

Many other loss function can be mentioned and are formulated to overcome different kind of problems. The list of presented loss function was related to the implementation of the *darknet*-like library which are ported also into the NumPyNet and Byron libraries, i.e either in Python either in C++. NumPyNet and Byron libraries also provided a wider list of loss functions to improve the usability of them and improve their computation (and fixed some *darknet* issues). A full list of available loss functions can be found in the on-line version of the libraries with a list of easy visual examples.

A further improvements was given by the Byron library from a numerical point-of-view: many mathematical formulas needs expensive math operations as logarithms and trigonometric functions. An efficient (but with less precision) math formulas was implemented in the C++ version to reach faster computational performances. These numerical math operations are widely used into the Byron library to increase the performances despite their used can be turned off by user at compile time. The full set of functions, in fact, is enclosed into a macro definition (`__fmath__`) that can be enabled during the compilation of the library.

2.13 Object Detection - Yolo architecture

2.14 Super Resolution - WDSR architecture

2.15 Image Segmentation - UNet architecture

2.16 Replicated Focusing Belief Propagation

Until now we are talking about neural networks based on the standard update rule of backpropagation. Other learning rule for weight updates were proposed and the choice of the best one it is a still open-problem. The final purpose is to obtain a feasible learning rule able to model the biological learning of the human brain.

The learning problem could be faced on through statistical mechanic models joined with the so called Large Deviation Theory. In general the learning problem can be split in two sub-parts: the classification problem and the generalization one. The first aims to completely store a pattern sample, i.e a prior known ensemble of input-output associations (*perfect learning*). The second one corresponds to compute a discriminant function based on a set of features of the input which guarantees a unique association of a pattern.

[26] We also try to prevent wrong uses of this loss function for laypersons. This implementation was already suggested by the *darknet* library so we simply propagate it in our implementations.

From a statistical point-of-view many Neural Network models have been proposed and the most promising seem to be models based on spin-glasses. Starting from a balanced distribution of the system, generally based on Boltzmann distribution, and under proper conditions, we can proof that the classification problem became a NP-complete computational problem. A wide range of heuristic solution to that type of problem were proposed.

In this section we show one these algorithms developed by Zecchina et al. [3] and called *Replicated Focusing Belief Propagation* (rFBP). The theoretical background of the algorithm is beyond the scope of this thesis so we focus on its numerical implementation and optimization.

Moreover, despite their proofed theoretical efficiency, the applications on real data are still few. Thus we show the application of the optimized version of the rFBP algorithm on the Genome Wide Association (GWA) data provided by the European COMPARE project. This work was also presented on the CCS-Italy (Conference of Complex System) of the 2019 [26].

2.17 Algorithm Optimization

The rFBP algorithm is a learning algorithm model developed to justify the learning process of a binary neural network framework. The model is based on a spin-glass distribution of neurons put on a fully connected neural network architecture. In this way each neuron is identified by a spin and so only binary weights (-1 and 1) can be assumed by each entry. The learning rule which controls the weight updates is given by the Belief Propagation method.

A first implementation of the algorithm was proposed in the original paper [3] jointly with an open-source Github repository. The original version was written in Julia language and despite it is a quite efficient implementation the Julia programming language stays on difficult and far from many users. To broaden the scope and use of the method a C++ implementation was developed with a jointly *Cython* wrap for Python users. The C++ language guarantees also better computational performances against the Julia implementation. This implementation is optimized for parallel computing and is endowed with a newly written C++ library called *Scorer* (see Appendix D for further details), which is able to compute a large number of statistical measurements based on a hierarchical graph scheme. With this optimized implementation we believe we can encourage researchers to approach these alternative algorithms and to use them more frequently in real context.

Like the Julia implementation also the C++ one provides the entire rFBP framework in a single library callable via a command line interface. The library widely uses template method to perform dynamic specialization of the methods between two magnetization version of the algorithm. The main categories of objects needed by the algorithm are wrapped in handy C++ objects easy to use also from the Python interface. A further optimization is given by the reduction of the number of available functions: in the original implementation a large amount of small functions are used to perform a single complex computation step enlarging the amount of call stack; in the C++ implementation the main functions are re-written with the minimum quantity of functions to ease the vectorization of the code.

The full rFBP library is released under MIT license and it is open-source on Github [19]. The on-line repository provides also a full list of installation instructions which could be performed via *CMake* or *Makefile*. The continuous integration of the project is guaranteed in every operative system using *Travis CI* and *Appveyor CI* which test more than 15 different C++ compilers and environments.

To encourage the Machine Learning community in the use of this kind of methods we provide a Python version based on a *Cython* wrap of the C++ objects. This wrap guarantees also a good integration with the other common Machine Learning tools provided in the *scikit-learn* Python package; in this way we can use the rFBP algorithm as equivalent in other pipelines. Like other Machine Learning algorithm also the rFBP one depends on many parameters, i.e its hyper-parameters, which has to be tuned according to the given problem. The Python wrap of the library was written also according the *scikit-optimize* Python package to allow an easy hyper-parameters optimization using the already implemented classical methods.

2.18 SNP classification

The few available applications of the rFBP algorithm to real data are amenable to two aspects: I) learning technique; II) algorithm implementation. The first one is related to the intrinsic definition of the algorithm which is designed to reach a complete memorization of the training dataset; in the other Machine Learning processes we normally want to avoid this kind of results since it could bring to *over-fitting* problems. The second one is given by the binary values involved in each step of the algorithm which intrinsically limit the possible applications[27].

Classification problems which involved only binary quantities are quite small but the GWA is one of them. In the GWA we have a series of genome data belonging to different classes as input. A genome is the ensemble of genes of an organism and each gene is identified by a series of nucleotides with 4 possible values (G, guanine; C, cytosine; A, adenine; T, thymine). The comparison between a reference (healthy) genome and an infected one highlights the biological mutation related to the understudy disease. This mutation are the so-called SNPs (Single Nucleotide Polymorphisms). So we can identify a genome as a sequence of its mutation in relation to a reference genome, i.e a sequence of two possible values given by the on/off of the mutation in each nucleotide.

The COMPARE project aims to develop new methods to avoid the genetic disease transmission. In this project plays a crucial role the *Source Attribution*, i.e the classification of a given disease based on the list of its mutation.

We tested the rfBP on 210 Salmonella enterica genome sequences, 4857450 bp (base pairs) long, living inside animals. Our early goal was to discriminate those bacteria living in pigs (159 samples) with respect to all the others animals (51 samples).

First of all we filter our data removing from each genome a base if it is not mutated in each sample. In this way we reduce the number of bases to 8189 bp. A graphical representation of these samples is given in Fig. **??**. The dataset was divided in training and test sets using a stratified cross-validation procedure to guarantee a proportional subdivision of the samples into the two classes. The algorithm hyper-parameters was tuned on the training set based on the performances obtained using a internal stratified 10-fold cross-validation: in each fold the training was performed by a given sequence of hyper-parameters and the performances evaluated on the corresponding test set; the hyper-parameters configuration which obtains the best performances on the full training set was chosen as best configuration. The performances evaluation was performed using the custom *Scorer* library. Considering the unbalanced sample quantities the Matthews Correlation Coefficient (MCC) is chosen as good scorer indicator for the evaluation.

With the tuned hyper-parameters we performed the training of rFBP algorithm on different percentage of the training set: 25%, 45%, 65% and 85%. In the same way we train also a list of the most common Machine Learning classifiers: single perceptron with floating-point weights (Perc); standard Neural Network with gradient descent as updating rule

[27] The Neural Network weights can assume only binary values since they model up/down spins. Moreover also the input is required to be a spin configuration and thus binary. The common Machine Learning problems involve floating-point values as input pattern.

(MLP); support vector machine with linear kernel (lSVM); support vector machine with radial kernel (rSVM); linear discriminant analysis (LDA); decision tree (DT); random forest (RF); k-nearest neighbors with 2-clusters (kNN); Guassian process (GP); diag-quadratic discriminant analysis (GNB); Bernoulli naive bayes (BNB); AdaBoost (AdaB). For each training percentage we perform the optimization of the hyper-parameters of each classifier with the same number of optimization steps. In Fig. **??** the accuracies and MCC results are shown, respectively.

From this analysis we can conclude that the rFBP algorithm shows comparable performances with the other classifiers. These performances globally grow with the training set size but only the rFBP is able to reach a "perfect learning" configuration, i.e accuracy of 100% and MCC=1. We have also noticed that the rFBP classifier and the GNB are the only two algorithms which qualitatively does not show performances saturation on their training.

A second analysis was performed on the data distribution using a multiple χ^2-test. Starting from the whole set of genomes we can compute the contingency-matrix of the two classes[28]. The χ^2-test was performed on the full set of 8189 *bp* and so the extracted *p-values* were corrected according multiple-tests. Using the Šidák [57] correction method and by the definition of significant threshold of 0.05 we found 1103 significant bases. An analogous χ^2-test was performed on the rFBP weights to identify a putative

2.19 Results

[28] The contingency-matrix displays the (multivariate) frequency distribution of the variables. Each row will count the number of hosts with/without the SNPs. Each column will identify a class.

Chapter 3

Biological Big Data - CHIMeRA project

3.1 The CHIMeRA project

3.2 CHIMeRA query

3.3 Data extraction - Web scraping

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 background

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

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

  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

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

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

Tourist flows in historical cities are continuously growing in a globalized world and adequate governance processes, politics and tools are necessary in order to reduce impacts on the urban livability and to guarantee the preservation of cultural heritage. The ICTs offer the possibility of collecting large amount of data that can point out and quantify some statistical and dynamic properties of human mobility emerging from the individual behavior and referring to a whole road network. In this work we analyze a new dataset that has been collected by the Italian mobile phone company TIM, which contains the GPS positions of a relevant sample of mobile devices when they actively connected to the cell phone network. Our aim is to propose innovative tools allowing to study properties of pedestrian mobility on the whole road network. Venice is a paradigmatic example for the impact of tourist flows on the resident life quality and on the preservation of cultural heritage. The GPS data provide anonymized geo-referenced information on the displacements of the devices. After a filtering procedure, we develop specific algorithms able to reconstruct the daily mobility paths on the whole Venice road network. The statistical analysis of the mobility paths suggests the existence of a travel time budget for the mobility and points out the role of the rest times in the empirical relation between the mobility time and the corresponding path length. We succeed to highlight two connected mobility subnetworks extracted from the whole road network, that are able to explain the majority of the observed mobility. Our approach shows the existence of characteristic mobility paths in Venice for the tourists and for the residents. Moreover the data analysis highlights the different mobility features of the considered case studies and it allows to detect the mobility paths associated to different points of interest. Finally we have disaggregated the Italian and foreigner categories to study their different mobility behaviors.

The datasets

The dataset used in this study has been provided by the Italian mobile phone company TIM and contains geo-referenced positions of tens of thousands anonymous devices (e.g. mobile phones, tablets, etc. ...), whenever they performed an activity (e.g. a phone call or an Internet access) during eight days from 23/2/2017 up to 02/03/2017 (Carnival of Venice dataset), and from 14/7/2017 up to 16/7/2017 (*Festa del Redentore* dataset). According to statistical data, 66% of the whole Italian population has a smart-phone and TIM is one the greatest mobile phone company in Italy whose users are $\sim$ 30% of the whole smart-phone population. The datasets refer to a geographical region that includes an area of the Venice province, so that it is possible to distinguish commuters from sedentary people and the different transportation means used to reach Venice. Each valid record gives information about the GPS localization of the device, the recording time, the signal quality and also the roaming status, which in turns allow to distinguish between Italian and foreigners. The devices are fully anonymized and not reversible identification numbers (ID) are automatically provided by the system for mobile phones and calls within the scope of the trial; the ID is kept for a period of 24 hours. During each activity a sequence of GPS data is recorded with a 2 sec. sampling rate and the collection stops when the activity ends. As matter of fact during an activity most of people reduce their mobility except if they are on a transportation mean, so that the dataset contains a lot of small trajectories that have to be joined to reconstruct the daily mobility. After a filtering procedure these data provide information on the mobility of a sample containing 3000 – 4000 devices per day. Since the presences during the considered events were of the order of 105 individuals per day, as reported by the local newspapers, we estimate an overall penetration of our sample of 3 – 4%. The filtering procedure and the other statistical informations about the sample penetration are discussed in the original paper [46].

Mobility paths reconstruction on the road network

The procedure of mobility path reconstruction considers separately the land mobility and the water mobility since the two mobility networks have different features, so that it is necessary to check carefully the transitions from one network to the other. To create a mobility path, we connect two successive points left by the same device using a best path algorithm on the road network with a check on the estimated travel speed to avoid unphysical situations and discarding the paths whose velocity is clearly not consistent with the typical pedestrian velocity (or ferryboat velocity). To end a land path and to start a water path, we require that at least two successive points of the same device are attributed to a ferryboat line by the localization algorithm. In the case of a single point on a ferryboat line, we force the localization of this point on the nearest road on the land.

The reconstruction of the mobility paths also allows to study how people perform their mobility on the road network. We consider the problem of determining the most used subnetwork of the Venice road network. The existence of mobility subnetworks could be the consequence of the peculiarity of Venice road network, where it is quite easy to get lost if you do not have a map. Therefore people with a limited knowledge of the road network move according to paths suggested by Internet sites or following the signs on the roads. To point out a mobility subnetwork we rank the roads of Venice according to a weight proportional to the number of mobility paths passing through each road. Thus We define a relevant subnetwork as a connected subnetwork that explains a considerable fraction of the observed mobility. In this case each road (identified by two nodes in the poly-line format) represents the link of our weighted graph and we can apply the DNetPRO technique shown in 1.3.3 to identify the network core with only closed paths[3].

Starting from the previously evaluated daily flows for each road, we order in a decreasing way the roads according to the observed flows. The DNetPRO algorithm scrolls down the list adding the road to a temporary list. At every step the "pruning process" starts on the selected roads cutting the isolated roads in order to get a connected subnetwork[4]. Therefore the number of nodes of the subnetwork increases in a discontinuous way, when the adding of a new road in the list allows to connect several previously selected roads. After several parametric scans, we found that the best result for our purposes is achieved by choosing about the 10% of the nodes in the whole Venice road network. In the Fig **??** we show four consecutive selected subnetworks in the case of Carnival dataset to illustrate how the algorithm operates.

Using the DNetPRO algorithm we are able to extract a subnetwork which explains the 64% of the observed mobility using 13% of the total road network length for the case of the Carnival dataset and 15% of the total length in the case of the *Festa del Redentore* dataset.

[3] Pendant nodes are unphysical solutions in our model since we are interested on the pedestrian mobility paths that bring people from one location to an other.

[4] Since we are interested on the largest connected component the *merging* parameter is off.

The selected road subnetworks are plotted in Fig **??** for both the datasets. As a matter of fact, many of the highlighted paths are also suggested by Internet sites. However, we remark some differences that can be related by the different nature of the considered events. During the Carnival of Venice the mobility seems to highlight three main directions connecting the railway station and the *Piazzale Roma* (top-left in the map), which are the main access points to the Venice historical centre, with the area around San Marco square, where many activities where planned during 26/02/2017. In the case of the *Festa del Redentore* the structure is more complex due to the appearance of several paths connecting the station and *Piazzale Roma* with the *Dorsoduro* district in front of the *Giudecca* island.

This geometrical structure could have a double explanation: on one hand the *Festa del Redentore* introduces an attractive area near the *Giudecca* island, where the fireworks take place in the evening; on the other hand the *Festa del Redentore* is a festivity very much felt by the local population, that knows the Venice road network and performs alternative paths.

On these subnetwork we also map the mobility of Italians and foreigns separately. The results of this application are deeply discussed in the paper.

Appendix C - BlendNet

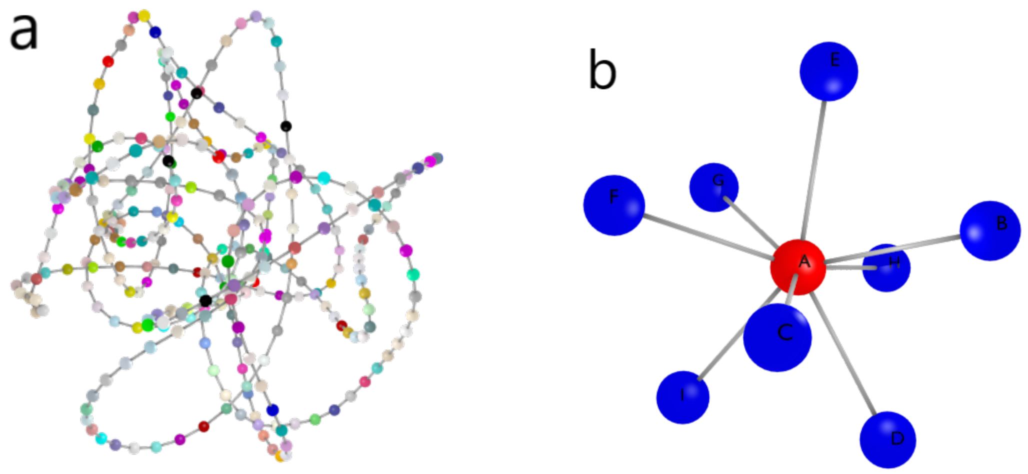

Figure 3.1: (**a**) Chain graph example rendered by BlendNet software. Node colors are random generated by the tool. (**b**) Star graph example rendered by BlendNet software. Node colors and labels are given as extra columns in node-list file.

Graph visualization is still an open problem in many applications. Commonly the problem is related to large graph visualization in which problems arise from the rendering of a large number of nodes and a greater number of links between them. An other open problem concern the multi-dimensional visualization of the graphs. Despite the most common graph tools compute the node coordinates in a any space dimensions (and clearly the maximum number of possible dimension for a visualization is still three) the real visualization is often allowed only a 2D space. The counterpart of these problems concern the pretty visualization of the graphs that it is often ignored in many tools but it can be guarantee a good result, the so called wow-effect, in a presentation.

In this section we introduce a new custom graph viewer developed for pretty small network visualization in 2D and 3D called *BlendNet* [13] (*Blender Network viewer*). BlendNet is open-source and it is released under GPL license. All the small-graphs showed in this work are made using this tool and in particular the feature-signature generated by the DNetPRO algorithm.

BlendNet is a custom tool written in Python with the help of Blender API. Blender is now a standard in the 3D rendering and it is commonly used in a wide range of graphical applications, starting from the simpler 3D dynamics to the video-games applications. Blender is certainly more than a simple graphical viewer but the easy Python interface and the wide on-line documentation and blogs make it a useful tool for graphical representation of 3D structures.

To use the Blender API we are forced to use the Python version installed inside software and any extra-package required by our application have to be installed with the appropriate `pip`. In our case we base our viewer on the `networkx` library for the computation of the possible node coordinates so we have to update our Python-Blender. Moreover since the code can be difficult to manage for non-expert users we create an easy user command-line interface to set the whole set of parameters required by the graph visualization that can be piloted by *Makefile* rules. The list of nodes and edges can be passed via command-line filename in the same format of the concurrent graph viewer (e.g *Gephi* software, the other graph viewer used in this work to generate the largest network structure of the CHIMeRA project).

The software project is a single script file and it includes a full list of possible examples and usages of the software. Some of this examples are shown in Fig. 3.1. A full list of installation instructions is also given for any operative system (Unix, MacOS and Windows). These instructions cover a full installation of Blender, Python and BlendNet package either for admin users either for no-root users [16]. With slight modifications of the code we can obtain different nodes coordinates and a node shapes. Nodes color, size and position can also be given in the node-list file as independent columns.

Appendix D - Multi-Class Performances

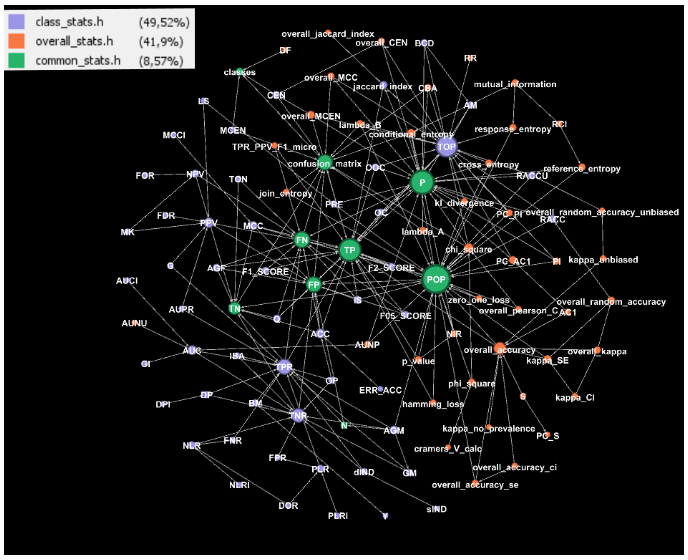

Figure 3.2: Multi class score interaction graph. Each node identify a different performance evaluator and link are given by the interaction between the mathematical formulation of each quantity. The graph has more than 100 nodes and more than 200 links. The node colors are given by the classes identified in the work of Sepand et al. [32].

The evaluation of performances is a crucial task in any Machine Learning application. Given a set of pattern and its corresponding (true) labels we can evaluate the efficiency of the understudy model with a comparison between them and the output of the model, i.e the predicted labels. There are a lot of different scores that can be computed and any of them evaluate some aspects of the model efficiency. Any paper author chose the score that better highlight the advantages of its model and it is difficult to move around this large zoo of indicators. Moreover (it is quite a constant in scientific research) when a paper is send to a peer-review in many cases the reviewer suggests to the author to check if other performance indicators are good enough for the showed results. This means that a lot of large simulations should be performed again and the appropriated variables re-computed to obtained the requested score.

At this point the main question is: are these scores totally independent one from each other? The brief answer is simply no. In a very interesting work of Sepand et al. [32] they show how we can compute the wide part of these scores starting from the evaluation of the simple confusion matrix[5]. Sepand et al. provide a full mathematical documentation and

[5] The confusion matrix is a square matrix of shapes (N, N), with N the total number of classes in the

references about the computation of this wide range of scores starting from the evaluation of the confusion matrix.

Despite the Python code provided by Sepand et al. explain this links between the mathematical quantities they stop their analysis on the scores evaluation without any interest on the optimization of these computations. Starting from their work we can analyze the inter-connections between these mathematical formulas and extract the dependencies between the involved variables. In particular, a score quantity can be interpreted as a node and its connections are given by the variables needed to evaluate it. Graphs of this type are commonly called *factor graphs*. In a mathematical formulation of *factor graphs* there are different kinds of nodes (variables and factors, or equations). The focus of our analysis is not on mathematical formalism of these kind of graphs but on the visualization of the functions interaction and on the results that we can obtain from it.

In the work of Sepand et al. the authors identify three classes of functions: common statistics, class statistics and overall stats, respectively. In Fig. 3.2 the interaction graph of these three classes is shown. The figure shows deeper interactions between the three classes of functions and highlights the dependencies of the different quantities. We can also use this kind of visualization to formulate computational considerations about the order in which compute these quantities. Since the graph is a direct graph by definition, we can start from the root node (the node without links which bring to it) and cross the network until the leaf nodes (nodes without link which go out from the node) like in a tree-graph. At each step of the percolation the incoming nodes identify totally independent quantities. This independences means that the node-quantities can be potentially computed in parallel. To clarify this considerations we can re-organize the graph visualization minimizing the link lengths and obtain a stratified graph in which each level identifies a potentially parallel section. A graph with these properties can be obtained using the *dot* visualization and it is shown in Fig. 3.3. As can be seen in the figure we can identify 7 levels in the graph and so 7 potentially parallel regions for the computation of the full set of functions.

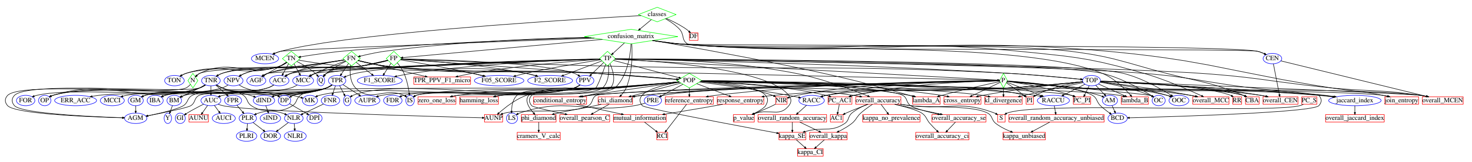

Figure 3.3: Re-organization of the graph in Fig. 3.2. The rendering was obtained using the *dot* visualization, i.e the minimization of the link lengths. The direct graph identifies the tree of dependencies and each level of the tree represents a set of independent functions that can be potentially computed in parallel. This graph is used as parallel scheme for the *Scorer* library.

These considerations allow the creation of the optimized version of the code of Sepand et al., the *Scorer* library [20]. The *Scorer* library is the C++ porting of the *PyCM* library of Sepand et al. with a *Cython* wrap for the Python compatibility. Following the pre-told graph the computation of the score quantities are performed in parallel according to the levels of the tree-graph in Fig. 3.3. The parallelization strategy chosen uses the `section` keyword of OpenMP library to perform no-wait task that are computed by each thread of the parallel region.

current problem, whose entries are the number of rights and false classification. In particular, each entry of the matrix represents the instances predicted in a given class. If the class is the right one we call it a true positive item. As counterpart we will have a false positive item.

The graph covers more than 100 different quantities so writing the full set of parallel sections becomes an hard work in C++. Moreover the updating of the graph with new quantities requires the updating of the full code and also of the parallelization strategy. Each function was written as an anonymous-struct, i.e a functor, with an appropriate operator overloading. Moreover each functor has a name given by a pre-determined regex (`get_{function}`) and the list of argument follows the same nomenclature[6]. With these expedients we created a fully automated creation of the C++ script which parses the list of above functors, it computes the dependency graph and the parallelization levels and give back a compilable C++ script with the desired characteristics. In this way we can guarantee an easy way to update the library and moreover we overcome the boring writing of a long code. The automatic pipeline creation script is provided in the *Scorer* library and can be used at each pull request or version update.

For a pretty/useful visualization of the computed quantities we render the interaction graph in an HTML framework. In this way in each node we can insert with a CSS table the computed values that can be discovered passing the mouse over the figure. An example of this rendering is given in the on-line version of the library [20].

In conclusion the developed *Scorer* library is a very powerful tool for Machine Learning performances evaluation which can be used either in C++ either in Python codes through the *Cython* wrap. The code is automatically generated at each update and automatically tested using Continuous Integration for any platform[7]. The code can be compiled using CMakefile or Makefile and a setup is provided for the Python version. So when you write a new paper on Machine Learning and you do not know what could be the most appropriate indicator to show in your research or you are afraid that a referee could ask you to compute an other one there is only one solution: compute them all using *Scorer*.

[6] If the functor receives in input the variable A and B we have to ensures that two functors named get_A and get_B will be provided. The only exception is given by the root functor.

[7] We perform tests for Unix and Windows environments. We check more than 15 combinations of environments and compilers.

Appendix E - Neural Network as Service

One of the final goals of Machine Learning is certainly the automation of the processes. We develop complex models to perform tasks that can be automatically executed by a computer without human supervision. Neural Networks are classically mathematical tools used for these purposes and was wide discussed in Chapter 2 of this work. Beyond the Neural Network structures and purposes for which they are made there is a still uncovered topic to discuss: the automation of these kind of algorithms inside a computer device. In this section we discuss an example of implementation of these algorithms as service in computer server. In particular we will talk about the implementation of the *FiloBlu* service which is a project developed in collaboration with the University of Sapienza (Roma) and the INFN-CNAF of Bologna. Since this work is still in progress and its purpose goes beyond the current topic, we will focused only on the implementation of the service without any reference on the Machine Learning algorithm used. This is a further proof that the developed techniques are totally independent by the final application purpose.

A service is a software that is executed in background in a machine. In Unix machines it is often call *daemon* while in Windows machine is called *Windows service*. A service can be started only by admin users and it goes on without any user presence. An other important requirement is the ability to re-start when some troubles occurs in the machine functionality and/or at the boot of the machine.

A Machine Learning service could be used in applications in which we have to manage an asynchronous stream of data for long time intervals. An example could be the case which the data provider is identified by an App or a video-camera. These data should stored inside a central database that can be located in a different device or in the same computer in which the service run. Since the service process runs in background the only communication channel with the user is given by log files. A log file is a simple readable file in which are saved the base informations about the current status of the service. Thus, it is crucial to set appropriate check-points inside the service script and chose the minimum quantity of informations that the service should write to make user-understandable its status.

FiloBlu Service

In the *FiloBlu* project we have a stream of data provided by an external App that are stored in a central database server. The Machine Learning service has to read the information in the database, to process them and finally write the results in the same database. All these operations have to be performed with high frequency since the result of the algorithm are shown in a real-time application. This frequency will be the clock-time of the process function, i.e at each time interval (as small as we like) the process task will be called and we have the desired results in output. At the same time we have to be care of the time required by our Machine Learning algorithm: not all the algorithms can process data in real time and the process function frequency has to be less than the time required by the algorithm or we can lose some frequency clock.

The best efficiency by a service can be obtained splitting as much as possible the required functionality in small-and-easy tasks. Small task can evaluated as independent functions with an associated frequency that in this case can be reduced as much as possible. The *FiloBlu* required functionality can be reviewed as a sequence of 3 fundamental steps and other 2 optional ones: read the data from the database, process the data with the Machine Learning algorithm and write the obtained results on the database are certainly the fundamental ones; update the Machine Learning model and clear old log files are optional steps. To further improve the efficiency of the service we can give each independent step to a different thread. The whole set of tasks will be piloted by a master thread given by the service itself. In this way the service will be computational efficient and moreover it does not weight on the computer performances. We have always take in mind that the computer which host the service have to be effected by the daemon process as less as possible either in memory either on computational point-of-view. Now we only have to synchronize our steps with appropriate clock frequencies.

Let's start from the reading data function. Since our data are assumed to be stored in a database this function have to perform a simple query and extract the latest data inserted. Obviously the efficiency of the step is based on the efficiency of the chosen query. The data extracted will be saved in a common container shared between the list of thread and thus belonging to the master. The choice of an appropriate shared container is a second point to carefully take in mind. This container should be light an thread-safe to avoid thread concurrency. While the second request is implementation dependent the first one can be faced on using a FIFO container[8]. In this way we can ensure that the application will saved a fixed maximum of data and it will not occupy large portion of memory (RAM).

The second task is identified by the Machine Learning function which process the data. The algorithm will take from the FIFO container of the previous step (if there is) and it will save the result in a second FIFO container for the next step. The time frequency of the step is given by the time required by the Machine Learning algorithm.

The third step will keep the data from the FIFO container of results (if there is) and it performs a second query (a writing one in this case) to the database. Also in this case the frequency is given by the efficiency of the chosen query.

The last two steps can be executed without press time requirements and are useful only on a large time scale.

Each step perform its independent logging on a single shared file. If an error occurs the service logs the message and save the current log-file in a different location to prevent possible log-cleaning (optional step). Then the service will be re-started.

We implemented this type of service in pure Python [15]. The developed service was customize according to the server requirements of the project[9]. We chose the Python language either for its simplicity in the code writing either for its thread native module which ensures a total thread-safety of each variable. Using simple decorator we are able to run each function in a separate-detached thread as required by the previous instructions. The project includes a documentation about its use (also in general applications) and it can be easily installed via `setup`. In the *FiloBlu* project we use a Neural Network algorithm written in *Tensorflow*. *Tensorflow* does not allow to run background process directly so the problem was overcame using a direct call to a Python script which perform full list of steps into an infinite loop. In this way the service can be re-started also if the process-service is

[8] FIFO container, i.e *First-In-First-Out*, is a special data structure in which the first element added will be processed as first and then automatically removed from it.

[9] The FiloBlu service is a Windows service and it can not run on Unix machines. Moreover the database used in the project is a MySQL one so the queries and the libraries used are compatible only with this kind of databases.

killed. The service can be driven using a simple *Powershell* script provided in the project.

Data Transmission

In the above configuration we have focused on the pipeline which process the stream of data ignoring the problems about the communication between the external device and the machine which host the service. The *FiloBlu* project uses an external App to send data to the main server. So we have two systems which have to communicate between them automatically via Internet connection. In general we could also manage sensitive data and the Internet communication could became a vulnerability in our application. To face on this problem we developed a simple TCP/IP client-server package which also supports a RSA cryptography, the *CryptoSocket* package [21].

The communication security could be an important point in many research applications and a valid cryptography is essential. The RSA cryptography is considered one of the most secure cryptography for data transmission and it is quite easy to implement. In this package we implemented a simple wrap around the *socket* Python library to perform a serialization of our data which will be (optionally) processed by a custom RSA algorithm. In this way different kind of data could be sent by the client at the same time. The client script could be adapted with slight modification for any user need and also complex Python structure could be transmitted between two machines. The cryptography module was written in pure C++ for computational efficiency and a *Cython* wrap was provided for a pure-Python application. *CryptoSocket* has only demonstrative purpose and so it works only for a 1-by-1 data transmission (1 server and 1 client).

Since this second implementation could be used also for other applications it was treated as a separated project and it has its own open-source code. The *CryptoSocket* package can be installed via *CMake* in any platform and a full list of installation instructions is provided in the project repository. The continuous integration of the project is guaranteed by testing the package installation across multiple C++ compilers and Unix and Windows platforms.

Appendix F - Bioinformatics Pipeline Profiling

In this work many times we have talked about the performances evaluation of a scripts in terms of time performances and other system statistics. The importance in the understanding the state of our infrastructure is essential not only for ensuring the reliability and stability of a software but also for a more efficiency use of the available resources. In particular about what concern the memory, CPUs and diskIO management is useful to know the required amount of each step of our software to perform the better parallelization strategy. Metrics represent the raw measurements of resource usage that are used by a software or a collection of them. These might be low-level usage summaries provided by the operating system, or they can be higher-level types of data tied to the specific functionality or work of a component. These kind of data could be collected and aggregated by a monitoring system like *Telegraf*[10]. In general, the difference between metrics and monitoring mirrors the difference between data and information. Monitoring takes metrics data, aggregates it, and presents it in various ways that allow humans to extract insights from the collection of individual pieces.

In this section we focused on the importance of software monitoring. In particular we will talk about a work conducted in collaboration with INFN-CNAF of Bologna about the monitoring and the performance evaluation of a bioinformatics pipeline across various computational environments [22].

In this work a previously published bioinformatics pipeline was reimplemented across various computational platforms, and the performances of its steps evaluated. The tested environments were: I) dedicated bioinformatics-specific server II) low-power single node III) HPC single node IV) virtual machine. The pipeline was tested on a use case of the analysis of a single patient to assess single-use performances, using the same configuration of the pipeline to be able to perform meaningful comparison and search the optimal environment/hybrid system configuration for biomedical analysis. Performances were evaluated in terms of execution wall time, memory usage and energy consumption per patient.

GATK-LODn pipeline

The pipeline used in this work, GATK-LODn, has been developed in 2016 by Do Valle et al. [27], and codifies a new approach aimed to Single Nucleotype Polimorphism (SNP) identification in tumors from Whole Exome Sequencing data (WES). WES is a type of "next generation sequencing" data [62, 6, 55], focused on the part of the genome that actually codifies proteins (the exome). Albeit known that non-transcriptional parts of the genome can affect the dynamic of gene expression, the majority of cancers inducing mutations are known to be on the exome, thus WES data allow to focus the computational effort on the most interesting part of the genome. Being the exome in human approximately 1% of the

[10] An automatic installation guide for Telegraf is provided in the Shut [16] project for any OS and also for no-root users.

	Coverage	No. of Reads	Read Length	BAM file size	NGS size
Whole genome	37.7x	975,000,000	115	82 GB	104 GB
Whole genome	38.4x	3,200,000,000	36	138 GB	193 GB
Exome	40x	110,000,000	75	5.7 GB	7.1 GB

Table 3.1: Typical dataset size for a single patient of different types of next generation sequencing. BAM file size refers to the size of the binary file containing the reads from the machine.

total genome, this approach helps significantly in reducing the number of false positives detected by the pipeline. The different sizes of next generation sequencing dataset are shown in Tab 3.1.

The GATK-LODn pipeline is designed to combine results of two different SNP-calling softwares, GATK [44] and MuTect [12]. These two softwares employ different statistical approaches for the SNP calling: GATK examines the healthy tissue and the cancerous tissue independently, and identifies the suspect SNPs by comparing them; Mutect compares healthy and cancerous tissues at the same time and has a more strict threshold of selection. In identifying more SNPs, GATK has a higher true positive calling than Mutect, but also an higher number of false positives. On the other end Mutect has few false positives, but often does not recognize known SNPs. The two programs also call different set of SNPs, even when the set size is similar. The pipeline therefore uses a combination of the two sets of chosen SNPs to select a single one, averaging the strictness of Mutect with the recognition of known variants of GATK.

The pipeline work-flow includes a series of common steps in bioinformatics analysis and in the common bioinformatics pipelines. It includes also a sufficient representative sample of tools for the performances statistical analysis. In this way the results extracted from the single steps analysis could be easily generalized to other standard bioinformatics pipelines.

With the increasing demand of resources from ever-growing datasets, it is not favorable to focus on single server execution, and is better to distribute the computation over cluster of less powerful nodes. The computational pipeline also has to manage a high number of subjects, and several steps of the analyses are not trivial to be done in a highly parallel way. Thus, the importance of system statistics management as the efficiency usage of available resources are crucial to reach a compromise between computational execution time and energy cost. For these reasons our main focus is on the performance evaluation of a single subject without using all the available resources, as these could be more efficiently allocated to concurrently execute several subjects at the same time. Due to the nature of the employed algorithms, not all steps can exploit the available cores in a highly efficient way: some scales sub-linearly with the number of cores, some have resource access bottleneck. Other tools are simply not implemented with parallelism in mind, often because they are the result of the effort of small teams that prefer to focus their attention on the scientific development side rather than the computational one.

Moreover in order to obtain an optimal execution of bioinformatics pipelines, each analysis step might need very different resources. This means that any suboptimal component of a server could act as a bottleneck, requiring bleeding edge technology if all the steps are to be performed on a single machine. Hybrid systems could be a possible solution to these issues, but designing them requires detailed information about how to partition the different steps of the pipeline.

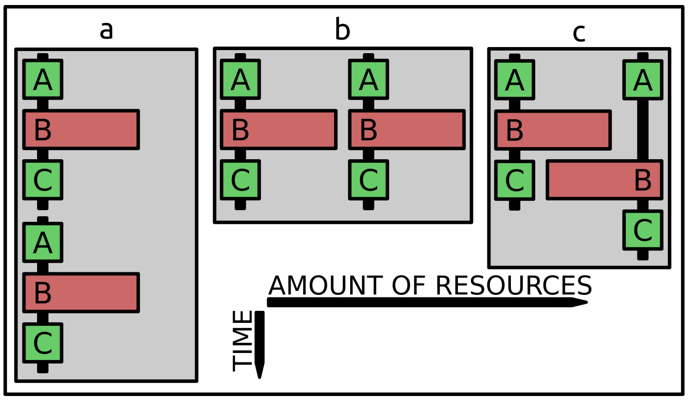

Figure 3.4: Examples of concurrency work-flow of two processes. The first case (*a*) represents a simple (naive) sequential work-flow; the second (*b*) highlights a brute force parallelization; the third (*c*) is the case of a perfect match between the available resources and the requested resources. Often brute force parallelization of pipelines done as in the image *b* ends up overlapping the most computationally intensive steps. Measuring the minimum viable requirements for the execution allow to better allocate resources as seen in the image *c*.

Computational Environments

There are two main optimization strategies: the first is to improve the efficiency of a single run on a single patient and the second is to employ massive parallelization on various samples. In both cases we have to know the necessary resources of the pipeline (and in a fine grain the resources of each step) and the optimal concurrency strategy to be applied to our work-flow (see Fig. 3.4). In the analyses we want to highlight limits and efficiencies of the most common computational environments used in big data analytics, without any optimization strategy of the codes or systems.

We also focused on a single patient analysis, the base case study to design a possible parallelization strategy. This is especially relevant for the multi-sample parallelization, that is the most promising of the two optimization strategies, as it does not rely on specific implementations of the softwares employed in the pipeline.

The pipeline was implemented on 5 computational environments: 1 server grade machine (Xeon E52640), 1 HPC node (Xeon E52683), 2 low power machines (Xeon D and Pentium J) and one virtual machine built on an AMD Opteron hypervisor. The characteristics of each node are presented in Tab. 3.2.

The server - grade node is a typical node used for bioinformatics computation, and as such features hundreds of GB of memory with multiple cores per motherboard: for these reasons we chose it as reference machine and the following results are expressed in relation to it.

The two low - power machines are designed to have a good cost - to - performance ratio, especially for the running cost[11]. These machines have been proven to be a viable solution for high performance computations [10]. Their low starting and running cost mean that a cluster of these machines would be more accessible for research groups looking forward to increase their computational power.

The last node is a virtual machine, designed to be operated in a cloud environment.

The monitoring tool used is *Telegraf*, which is an agent written in Go for collecting, processing, aggregating, and writing metrics. Each section of the pipeline sends messages

[11] Running cost is evaluated as the energy consumption that the node requires per subject, assuming that the consumption scales linearly with the number of cores used in the individual step.

CLASS	**server grade machines**		**low power machines**		**virtual machine**
CPU	Intel Xeon	Intel Xeon	Intel Pentium	Intel Xeon	AMD Opteron
version	E5-2683v3	E5-2640v2	J4205	D-1540	6386 SE
Microarchitecture	Haswell	Ivy Bridge EP	Apollo Lake	Broadwell	Piledriver
Launch Date	Q3'14	Q3'13	Q4'16	Q1'15	Q3'12
Lithography	22 nm	22 nm	14 nm	14 nm	32 nm
Cores/threads	14/28	8/16	4/4	8/16	16
Base/Max Freq	2.00/3.00	2.00/2.50	1.50/2.60	2.00/2.60	2.80/3.50
L2 Cache	35 MB	20 MB	2 MB	12 MB	16 MB
TDP	120 W	95 W	10 W	45 W	115 W
Total CPUs	2	2	1	1	1
total cores/threads	28/56	16/32	4/4	8/16	16
Total Memory	256 GB	252 GB	8 GB	32 GB	60 GB
System power	240 + 60 W	190 + 60 W	10 + 2 W	45 + 10 W	115 + 10 W
Electrical costs	650 €/year	550 €/year	26 €/year	120 €/year	273€ /year
System price	4000-6000 €	3000-5000 €	100-130 €	900-1200 €	2000-3000€

Table 3.2: Characteristics of the tested computational environments. Electrical costs are estimated as 0.25 €/kWh; CPU frequencies are reported in GHz; TDP: Thermal Design Power, an estimation indicator of maximum amount of heat generated by a computer chip when a "real application" runs.

to the *Telegraf* daemon independently.

Regardless of the number of cores of each machine we restrict the number of cores used to only two to compare the statistics: this restriction certainly penalize the environment with multiple cores but with a view of maximizing the parallelizations and minimize the energy cost it is the playground to compare all the available environments. Another restriction is applied to the chosen architectures: since available low - power machines provides only x86 - architectures also the other environments are forced to work in x86 to allow the statistics comparison.

Pipeline steps

The pipeline steps that have been examined are a subset of all the possible steps: we only focus on those whose computational requirements are higher and thus require the most computational power. These steps are:

1. **mapping:** takes all the reads of the subjects and maps them on the reference genome;
2. **sort:** sorts the sequences based on the alignment, to improve the reconstruction steps;
3. **markduplicates:** checks for read duplicates (that could be imperfections in the experimental procedures and would skew the results);
4. **buildbamindex:** indexes the dataset for faster sorting;
5. **indexrealigner:** realigns the created data index to the reference genome;
6. **BQSR:** base quality score recalibration of the reads, to improve SNPs detection;
7. **haplotypecaller:** determines the SNPs of the subject;

8. **hardfilter:** removes the least significant SNPs.

The following statistics were evaluated:

1. **memory per function:** estimate percentage of the total memory available to the node used for each individual step of the pipeline;

2. **energy consumption:** estimated as the time taken by the step, multiplied by the number of cores used in the step and the power consumption per core (TDP divided by the available cores). As mentioned before this normalization unavoidably penalize the multi-core machines but give us a term of comparison between the different environment;

3. **elapsed time:** wall time of each step.

The pipeline was tested on the patient data from the 1000 genome project with access code NA12878, sample SRR1611178. It is referred as a Gold Standard reference dataset [61]. It is generated with an Illumina HiSeq2000 platform, SeqCap EZ Human Exome Lib v3.0 library and have a 80x coverage. As Gold Standard reference it is commonly used as benchmark of new algorithm and for our purpose can be used as valid prototype of genome.

Results

Memory occupation is one of the major drawbacks of the bioinformatics pipelines, and one of the greater limits to the possibility of parallel computation of multiple subjects at the same time. As it can be seen in Fig. 3.5, the memory occupation is comprised between 10% and 30% on all the nodes. This is due to the default behavior of the GATK libraries to reserve a fixed percentage of the total memory of the node. The authors could not find any solution to prevent this behavior from happening. As it can be noticed, in the node with the greatest amount of total memory (both Xeon E5 and the virtual machine) the requested memory is approximately stable, as is always sufficient for the required task. The memory allocation is less stable in the nodes with a limited memory (Xeon D and Pentium J), as GATK might requires more memory than what initially allocated to perform the calculation. The exception to this behavior is the “mapping” step, that uses a fixed amount of memory independently from the available one (between 5 and 7 GB). This is due to the necessity of loading the whole human reference genome (version hg19GRCh37) to align each individual read to it. All the other steps do not require the human reference genome but can work on the individual reads, allowing greater flexibility in memory allocation.

As can be seen in Fig. 3.6 and Fig. 3.7, this increase of memory consumption does not correspond to a proportional improvement of the time elapsed in the computation.

The elapsed time for each step and for the whole pipeline can be seen in Fig. 3.6. It can be seen that there is a non consistent trend in the behavior of the different environments. Aside from the most extreme low power machine, the pentium J, the elapsed times are on average higher for the low power and slightly higher for the cloud node, but the time for the individual rule can vary. In the sorting step, Pentium J is 20 times slower than the reference. This is probably due to the limited cache and memory size of the pentium J, that are both important factors determining the execution time of a sorting algorithm and are both at least four to six times smaller than the other machines. The HPC machine, the Xeon E52683, is consistently faster than the reference node.

The energy consumption per step can be seen in Fig. 3.7. The low power machines are consistently less than half the baseline consumption. Even considering the peak of

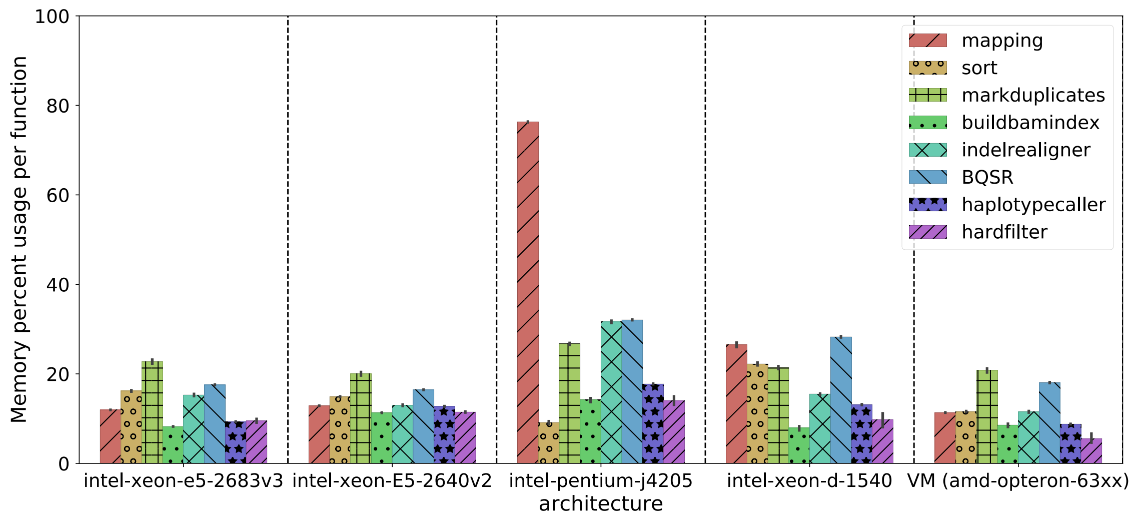

Figure 3.5: Memory used for each step of the pipeline. Due to the GATK memory allocation strategy, all steps use a baseline amount of memory proportional to the available memory. Smaller nodes, like the low power ones, require more memory as the baseline allocated memory is not sufficient to perform the calculation.

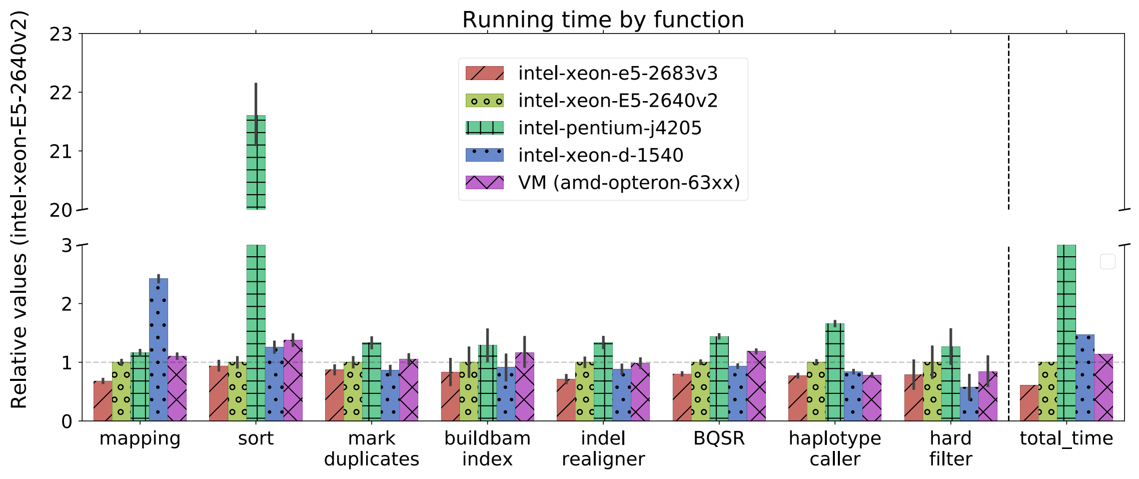

Figure 3.6: Time elapsed per step of the pipeline, and total elapsed time. In the sorting step, Pentium J is 20 times slower than the reference, probably due to the limited cache size.

consumption due to the long time required to perform the sorting, the most efficient low power machine, the pentium J, consumes 40% of the reference, and the Xeon D consumes 60% of the reference. The HPC machine, the Xeon E52683, have consumption close to the low power nodes, balancing out the higher energy consumption with a faster execution speed. The virtual machine has the highest consumption despite the fact that the execution time of the whole pipeline is comparable to the reference due to the high TDP compared to its execution time.

Conclusions

Bioinformatics pipelines are one of the most important uses of biomedical big data and, at the same time, one of the hardest to optimize, both for their extreme requisites and the constant change of the specification, both in input-output data format and program API.

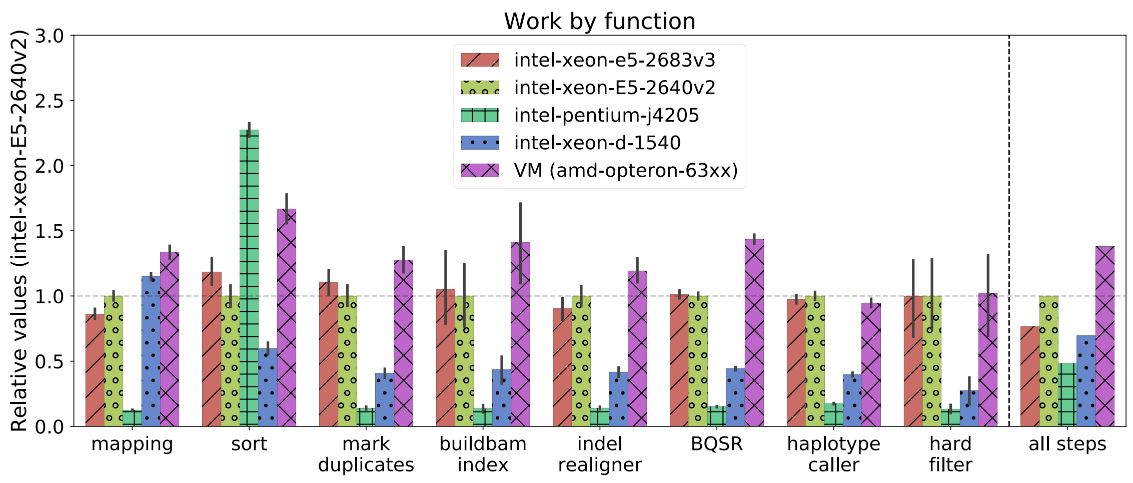

Figure 3.7: Energy consumption per pipeline step and on the whole pipeline. Energy consumption is estimated as the time taken by the step, multiplied by the number of cores used in the step and the power consumption per core (TDP divided by the available cores).

This makes the task of pipeline optimization a daunting one, especially for the final target of the results; physicians and biologists could lack the technical expertise (and time) required to optimize each new version of the various softwares of the pipelines. Moreover, in a verified pipeline updating the software included without a long and detailed cross-validation with the previous one is often considered a bad practice: this means that often these pipelines are running with under-performing versions of each software.

Clinical use of these pipelines is growing, in particular with the rise of the concept of “personalized medicine”, where the therapy plan is designed on the specific genotype and phenotype of the individual patient rather than on the characteristic of the overall population. This would increase the precision of the therapy and thus increase its efficacy, while cutting considerably the trial and error process required to identify promising target of therapy. This requires the pipelines to be evaluated in real time, for multiple subjects at the same time (and potentially with multiple samples per subject). To perform this task no single node is powerful enough, and thus it is necessary to use clusters. This brings the need to evaluate which is the most cost and time efficient node that can be employed.

In the cost assessment there are several factors that need to be considered aside of the initial setup cost, namely cost for running the server and opportunity cost for obsolescence. Scaled on medium sized facilities, such the one that could be required for a hospital, this cost could quickly overcome the setup cost. This cost does also include not only the direct power consumption of the nodes, but also the required power for air conditioning to maintain them in the working temperature range. Opportunity costs are more complex, but do represent the loss of possibility of using the most advanced technologies due to the cost of the individual node of the cluster. Higher end nodes require a significant investment, and thus can not be replaced often.

With this perspective in mind, we surmise that energy efficient nodes present an interesting opportunity for the implementation of these pipelines. As shown in this work, these nodes have a low cost per subject, paired with a low setup cost. This makes them an interesting alternative to traditional nodes as a workhorse node for a cluster, as a greater number of cores can be bought and maintained for the same cost.

Given the high variability of the performances in the various steps, in particular with the sorting and mapping steps, it might be more efficient to employ a hybrid environment, where few high power nodes are used for specific tasks, while the bulk of the computation is done by the energy efficient nodes. This is true even for those steps that can be massively parallelized, such as the mapping, as they benefit mainly from a high number of processors rather than few powerful ones. In this work we focused only on CPUs computation, but another possibility could be an hybrid-parallelization approach in which the use of a single GPU accelerator can improve the parallelization of the slower steps. Each pipeline work-flow requires its own analyses and tuning to reach the best performances and the right parallelization strategy based on the use which it is intended but a low energy node approach is emerging as a good alternative to the more expensive and common solutions.

Bibliography

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] AlexeyAB. darknet. https://github.com/AlexeyAB/darknet, 2019.

[3] C. Baldassi, C. Borgs, J. T. Chayes, A. Ingrosso, C. Lucibello, L. Saglietti, and R. Zecchina. Unreasonable effectiveness of learning neural networks: From accessible states and robust ensembles to basic algorithmic schemes. *Proceedings of the National Academy of Sciences*, 113(48):E7655–E7662, 2016.

[4] A. Battle, S. Mostafavi, X. Zhu, J. B. Potash, M. M. Weissman, C. McCormick, C. D. Haudenschild, K. B. Beckman, J. Shi, R. Mei, A. E. Urban, S. B. Montgomery, D. F. Levinson, and D. Koller. Characterizing the genetic basis of transcriptome diversity through rna-sequencing of 922 individuals. *Genome Research*, 2014.

[5] 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.

[6] S. Behjati and P. S. Tarpey. What is next generation sequencing? *Archives of disease in childhood - Education & practice edition*, 98(6):236–238, 2013.

[7] 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.

[8] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[9] C. Cenik, E. S. Cenik, G. W. Byeon, F. Grubert, S. I. Candille, D. Spacek, B. Alsallakh, H. Tilgner, C. L. Araya, H. Tang, E. Ricci, and M. P. Snyder. Integrative analysis of rna, translation, and protein levels reveals distinct regulatory variation across humans. *Genome Research*, 2015.

[10] D. Cesini, E. Corni, A. Falabella, A. Ferraro, L. Morganti, E. Calore, S. F. Schifano, M. Michelotto, R. Alfieri, R. De Pietri, T. Boccali, A. Biagioni, F. Lo Cicero, A. Lonardo, M. Martinelli, P. S. Paolucci, E. Pastorelli, and P. Vicini. Power-efficient computing: Experiences from the cosa project. *Scientific Programming*, 2017, 2017.

[11] 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.

[12] K. Cibulskis, M. S. Lawrence, S. L. Carter, A. Sivachenko, D. Jaffe, C. Sougnez, S. Gabriel, M. Meyerson, E. S. Lander, and G. Getz. Sensitive detection of somatic point mutations in impure and heterogeneous cancer samples. *Nature Biotechnology*, 31(3):213–219, 2013.

[13] N. Curti. BlendNet: Blender network viewer. https://github.com/Nico-Curti/BlendNet, 2019.

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

[15] N. Curti. FiloBlu: Machine learning as service. https://github.com/Nico-Curti/FiloBlue, 2019.

[16] N. Curti. Shut: Shell utilities for no-root users. https://github.com/Nico-Curti/Shut, 2019.

[17] N. Curti and M. Ceccarelli. Numpynet: Neural network in pure numpy. https://github.com/Nico-Curti/NumPyNet, 2019.

[18] N. Curti, M. Ceccarelli, A. Baroncini, S. Sinigardi, and F. Alessandro. Byron: Build your own neural network library. https://github.com/Nico-Curti/Byron, 2019.

[19] N. Curti and D. Dall'Olio. Replicated focusing belief propagation. https://github.com/Nico-Curti/rFBP, 2019.

[20] N. Curti and D. Dall'Olio. Scorer: Machine learning scorer library. https://github.com/Nico-Curti/Scorer, 2019.

[21] N. Curti and A. Fabbri. Cryptosocket - tcp/ip client server with rsa cryptography. https://github.com/Nico-Curti/CryptoSocket, 2019.

[22] N. Curti, E. Giampieri, A. Ferraro, C. Vistoli, E. Ronchieri, D. Cesini, B. Martelli, C. Duma Doina, and G. Castellani. Cross-environment comparison of a bioinformatics pipeline: Perspectives for hybrid computations. *Springer, Cham, Euro-Par 2018: Parallel Processing Workshops*, 11339, 2019.

[23] N. Curti, E. Giampieri, G. Levi, G. Castellani, and D. Remondini. Dnetpro: A network approach for low-dimensional signatures from high-throughput data. 2019.

[24] 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.

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

[26] D. Dall'Olio, N. Curti, G. Castellani, A. Bazzani, and D. Remondini. C++ implementation, optimization and application of the focusing belief propagation algorithm, 2019.

[27] Í. F. do Valle, E. Giampieri, G. Simonetti, A. Padella, M. Manfrini, A. Ferrari, C. Papayannidis, I. Zironi, M. Garonzi, S. Bernardi, M. Delledonne, G. Martinelli, D. Remondini, and G. Castellani. Optimized pipeline of MuTect and GATK tools to improve the detection of somatic single nucleotide polymorphisms in whole-exome sequencing data. *BMC Bioinformatics*, 17(S12):341, 2016.

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

[29] X. Glorot, A. Bordes, and Y. Bengio. Deep sparse rectifier neural networks. In G. Gordon, D. Dunson, and M. Dudík, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pages 315–323, Fort Lauderdale, FL, USA, 11–13 Apr 2011. PMLR.

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

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

[32] S. Haghighi, M. Jasemi, S. Hessabi, and A. Zolanvari. PyCM: Multiclass confusion matrix library in python. *Journal of Open Source Software*, 3(25):729, may 2018.

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

[34] A. Hodgkin and A. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *Bltn Mathcal Biology*, 1990.

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

[36] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv e-prints*, page arXiv:1502.03167, Feb 2015.

[37] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22Nd ACM International Conference on Multimedia*, MM '14, pages 675–678, New York, NY, USA, 2014. ACM.

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

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

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

[41] Y. Lecun, L. Bottou, G. Orr, and K.-R. Müller. Efficient backprop. 08 2000.

[42] M. Malvisi, N. Curti, D. Remondini, G. Gandini, F. Palazzo, G. Pagnacco, J. L. Williams, and G. Minozzi. Combinatorial discriminant analysis applied to rnaseq data reveals a set of 10 transcripts as signatures of infection of cattle with mycobacterium avium subsp. paratuberculosis. 2019.

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

[44] A. McKenna, M. Hanna, E. Banks, A. Sivachenko, K. Cibulskis, A. Kernytsky, K. Garimella, D. Altshuler, S. Gabriel, M. Daly, and M. A. DePristo. The genome analysis toolkit: A MapReduce framework for analyzing next-generation DNA sequencing data. *Genome Research*, 20(9):1297–1303, 2010.

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

[46] 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.

[47] B. Okken. *Python Testing with Pytest: Simple, Rapid, Effective, and Scalable*. Pragmatic Bookshelf, 1st edition, 2017.

[48] T. Oliphant. NumPy: A guide to NumPy. USA: Trelgol Publishing, 2006–. [Online; accessed <today>].

[49] 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.

[50] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in PyTorch. In *NIPS Autodiff Workshop*, 2017.

[51] 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.

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

[53] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, pages 65–386, 1958.

[54] 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.

[55] J. Shendure and H. Ji. Next-generation DNA sequencing. *Nature Biotechnology*, 26(10):1135–1145, 2008.

[56] W. Shi, J. Caballero, F. Huszár, J. Totz, A. P. Aitken, R. Bishop, D. Rueckert, and Z. Wang. Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network. *arXiv e-prints*, page arXiv:1609.05158, Sep 2016.

[57] Z. Sidak. Rectangular confidence regions for the means of multivariate normal distributions. *Journal of the American Statistical Association*, 62(318):626–633, 1967.

[58] J. Siek, L.-Q. Lee, and A. Lumsdaine. Boost graph library. `http://www.boost.org/libs/graph/`, June 2000.

[59] C. Terragna, D. Remondini, M. Martello, E. Zamagni, L. Pantani, F. Patriarca, A. Pezzi, G. Levi, M. Offidani, I. Proserpio, G. De Sabbata, 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.

[60] Y. Yuan, E. M. Van Allen, L. Omberg, N. Wagle, 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, L. A. Garraway, A. A. Margolin, G. Getz, and H. Liang. Assessing the clinical utility of cancer genomic and proteomic data across tumor types. *Nature Biotechnology*, 32(7):644–652, 2014.

[61] J. M. Zook, B. Chapman, J. Wang, D. Mittelman, O. Hofmann, W. Hide, and M. Salit. Integrating human sequence data sets provides a resource of benchmark snp and indel genotype calls. *Nature Biotechnology*, 32, 2014.

[62] M. Zwolak and M. Di Ventra. Colloquium: Physical approaches to DNA sequencing and detection. *Reviews of Modern Physics*, 80(1):141–165, 2008.

Acknowledgment