Universidad Politécnica
de Madrid

**Escuela Técnica Superior de
Ingenieros Informáticos**

Master's Programme in Data Science

Master Thesis

A Genetic Atlasing Toolbox with a Standalone Web Interface and Basic Functionality Plugin in the EBRAINS Interactive Atlas Viewer

Author: Javier Gallego Gutiérrez
Supervisors: Concha Bielza and Pedro Larrañaga

Madrid, July, 2022

Master Thesis
Master in Data Science

Title: A Genetic Atlasing Toolbox with a Standalone Web Interface and Basic Functionality Plugin in the EBRAINS Interactive Atlas Viewer
July, 2022

Author:	Javier Gallego Gutiérrez
Supervisors:	Concha Bielza and Pedro Larrañaga
	Computational Intelligence Group
	Departamento de Inteligencia Artificial
	ETSI Informáticos
	Universidad Politécnica de Madrid

Acknowledgements

I would like to express my gratitude for everyone involved during the course of this project. First of all, I would like to thank my professors, Pedro and Concha, for their dedicated help and guidance. Also, thanks to all my colleagues in the Computational Intelligence Group for their company and daily help throughout these months. I also wish to thank all those who have previously participated in the development of *NeuroSuites*, especially Mario and Hugo. Finally, I would like to thank the members of the Big Data Analytics Group of the Institute of Neuroscience and Medicine (INM-1), Forschungszentrum Jülich GmbH, for their help in the plugin development on their viewer.

This project has received funding from the European Union's Horizon 2020 Framework Programme for Research and Innovation under the Specific Grant Agreement No. 945539 (Human Brain Project SGA3).

Abstract

Bayesian networks are a widely used mathematical framework with different fields of application. One of them is the computational biology problem of learning gene regulatory networks from genomic data. Many models have been proposed for this problem, but there is no perfect one. Among them, Bayesian networks show some pros that make their usage an interesting topic to study.

This project presents a new Python package for learning and manipulating massive Bayesian networks. This software is based on the existing framework *BayeSuites*. It is focused on learning gene regulatory networks from data modeled as Bayesian networks. Additionally, we present a plugin for `siibra-explorer`, the EBRAINS 3D Atlas Viewer. EBRAINS is the digital brain research infrastructure created by the Human Brain Project. This plugin uses the capability of `siibra-python` for extracting gene expression data from the Allen Brain Atlas and the aforementioned Python package to learn those networks interactively.

Keywords. Bayesian networks, gene regulatory networks, `siibra`, EBRAINS

Resumen

Las redes bayesianas representan un modelo matemático con múltiples campos de aplicación. Uno de ellos es el problema presentado en el mundo de la biología computacional de aprender redes de regulación genética a partir de datos. Diferentes modelos han sido propuestos para este problema, pero no hay ninguno que sea el mejor en todos los aspectos. Entre ellos, el uso de redes bayesianas presenta algunas ventajas que hacen de este un tema de estudio en el que puede ser interesante profundizar.

A lo largo de este trabajo, presentamos un nuevo paquete de Python que permite aprender, visualizar y manipular redes bayesianas grandes. Este software está basado en el marco *BayeSuites*. Se centra en el aprendizaje de redes de regulación genética modeladas como redes bayesianas a partir de datos. Además, presentamos una extensión para `siibra-explorer`, la herramienta de visualización 3D del atlas del cerebro de EBRAINS. EBRAINS es la infraestructura digital para la investigación sobre el cerebro creada por el Human Brain Project. Esta extensión hace uso de la posibilidad de extraer datos de expresión genética del Allen Brain Atlas mediante `siibra-python`, así como del previamente mencionado paquete de Python para aprender interactivamente esas redes.

Palabras clave. redes bayesianas, redes de regulación genética, `siibra`, EBRAINS

Contents

Chapter 1

Introduction

First of all, we shortly introduce the reasons for carrying out this project as well as its main goals.

1.1 Motivation

The project explained in this document is integrated into the Human Brain Project[1]. The Human Brain Project is a 10-year scientific research project funded by the European Commission. It aims to develop a useful infrastructure that helps neuroscience, computing, and brain-related medicine researchers with their work. The Human Brain Project has developed a neuroscience research digital infrastructure called EBRAINS. Its goal is to make it accessible for all the brain research community in the EU and accelerate human brain understanding. As part of the SGA3 phase WP4 running from 2019 to 2023, and within the EBRAINS Brain Atlas Services (SC2) group, the development of some brain atlas analysis tools was proposed. One of them was a toolbox that analyzes gene expression data and learns gene regulatory networks based on the already existing framework *BayeSuites* [1]. *BayeSuites* is another project that makes part of the Human Brain Project. It is a framework for learning and interpreting Bayesian networks applied to neuroscience included in the web application *NeuroSuites*. The inclusion of *BayeSuites* functionality to the EBRAINS Atlas Viewer for learning such networks with gene expression data that can be retrieved within the EBRAINS framework and `siibra` toolsuite would help neuroscience researchers to represent gene regulatory networks easily and in the same web application they can use for other purposes. That is the main reason for carrying out this project.

1.2 Objectives

This project is divided into two different, but related objectives. The first one is to take *BayeSuites* functionality and develop a proper Python package for learning, visualizing, and handling Bayesian networks with a particular focus on neuroscience and gene regulatory networks. The development of this package includes taking decisions about the features from *BayeSuites* to include, contributing with some extensions, giving it the standard form of Python packages, writing full documentation, and publishing it as any other public library. Once this process is finished, the second goal is to add the aforementioned tool to the EBRAINS Atlas

[1]https://www.humanbrainproject.eu/en/

Viewer. This web application presents the capability of easily allowing the implementation of plugins, so the tool will follow this path to be included in the viewer. The developed package is the base of the tool, i.e., it provides the functionality for learning gene regulatory networks modeled as Bayesian networks.

1.3 Contributions

This chapter summarizes the contributions made throughout this project to both the Human Brain Project and the Computational Intelligence Group.

- **Modularizing *BayeSuites***: *BayeSuites* functionality has been moved to a Python package. This process includes the following tasks:
  1. **Package structure**: adapting the code to a package project and the PEP 8 style guide, writing docstrings that follow the PEP 257 convention, adding all necessary files (setup.py, LICENSE, requirements.txt, unitary tests, example notebooks, README, etc.), and uploading the package to *PyPI*.
  2. Generating **web documentation** from docstrings and uploading it to *Read the docs*.
  3. **Input/output extension**: adding formats such as GEXF and JSON for reading and writing networks.
  4. **Inference extension**: adding inference for discrete Bayesian networks.
  5. **Minor changes**: restructuring code, adding more layouts, and using the `logging` API.
- The **plugin implementation** may be divided into two different tasks:
  1. **User interface**: designing and implementing a new user interface for handling Bayesian networks in `siibra-explorer`.
  2. **Plugin build**: coding, and deploying all technical requirements to get the final plugin application running on the atlas viewer.

1.4 Document structure

This document has four main chapters. The first two chapters detail the state of the art both theoretically and in software terms, while the following two chapters show the software development done during this project.

Chapter 2 introduces the theoretical background needed for understanding the rest of the document. It mainly presents all the notions about Bayesian networks and gene regulatory networks that we use later.

Chapter 3 describes all the capabilities and technical information about the existing software we use, i.e., *NeuroSuites* and `siibra`.

Chapter 4 explains the process followed to obtain the desired Python package, its features and some explanations about how to use it.

Chapter 5 presents the plugin developed for the EBRAINS Atlas Viewer including explanations about how it has been built, images of its graphical user interface, and an example run on it.

Chapter 6 concludes this document discussing the outcome of our work, its pros and cons, and future steps that could be taken to improve our software.

Chapter 2

Theoretical background

This chapter briefly introduces the definition of Bayesian network, the process of learning their structure and parameters, how inference in Bayesian networks works, and some other useful properties about them (Section 2.1). Then, we present the notion of gene regulatory networks, the problem of learning them, and the most typical methods used for modeling gene regulatory networks focusing on the use of Bayesian networks (Section 2.2).

2.1 Bayesian networks

Bayesian networks [2], [3] try to represent knowledge under uncertainty in a visual way. They represent probabilistic relationships over a vector of random variables $\mathcal{X} = (X_1, X_2, \dots, X_n)$ using a *directed acyclic graph* (DAG) $\mathcal{G}$. This representation tries to build the joint probability distribution (JPD)

$$P(X_1, X_2, X_3, \dots, X_n) = P(X_1)P(X_2|X_1)P(X_3|X_1, X_2) \cdots P(X_n|X_1, X_2, \dots, X_{n-1}) \tag{2.1}$$

Bayesian networks representation is based on conditional independence. Two variables X_i and X_j are *conditionally independent* given another one X_k if $P(X_i|X_j, X_k) = P(X_i|X_k)$. The graph $\mathcal{G}$ represents conditional independence relations in a way that each variable is conditionally independent of its non-descendants given its parents in the graph. That is called the *Markov condition* and applying it to Equation (2.1) we can decompose the JPD as

$$P(\mathcal{X}) = \prod_{i=1}^{n} P(X_i|\mathbf{Pa}(X_i))$$

where $\mathbf{Pa}(X_i)$ represent the parents of the node X_i in the graph.

2.1.1 Properties

There exist two particularly useful properties related to Bayesian networks: *d-separation* and the *Markov blanket* of a node.

Let $\mathcal{X}, \mathcal{Y}$ and $\mathcal{Z}$ be disjoint sets of nodes in a DAG. $\mathcal{X}$ and $\mathcal{Y}$ are *d-separated* by $\mathcal{Z}$ if for every undirected path between $\mathcal{X}$ and $\mathcal{Y}$ there is an intermediate variable $T \notin \mathcal{X} \cup \mathcal{Y}$ such that

- When there is a structure $T_{-1} \rightarrow T \leftarrow T_{+1}$ in the path, no node among T and its descendants belong to $\mathcal{Z}$ or

- There is no such structure and T belongs to $\mathcal{Z}$.

The d-separation property is actually interesting because it implies conditional independence. If $\mathcal{X}$ and $\mathcal{Y}$ are d-separated by $\mathcal{Z}$, each pair of nodes $X \in \mathcal{X}, Y \in \mathcal{Y}$ are conditionally independent given $\mathcal{Z}$.

Given a node X_i in a Bayesian network, we call *Markov blanket* of X_i to the set of nodes $\mathbf{MB}(X_i)$ such that X_i is conditionally independent of all other nodes in the network given $\mathbf{MB}(X_i)$, i.e.,

$$P(X_i|\mathcal{X} \setminus \{X_i\}) = P(X_i|\mathbf{MB}(X_i))$$

When conditional independence also implies d-separation, the Markov blanket of a node is formed by its parents, children and the parents of its children.

2.1.2 Gaussian Bayesian networks

Regarding the continuous case, it is not possible to determine a general distribution. However, there is a useful situation where the JPD is also easy to handle: multivariate Gaussian distributions [4], [5]. Gaussian distributions are a good enough approximation for some real world cases, but we have to take into account they assume linear relationships between variables. For a random vector $\mathcal{X} \in \mathbb{R}^n$, a multivariate Gaussian distribution is given by the function

$$f(\mathcal{X}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{\pi/2}|\boldsymbol{\Sigma}|^{1/2}} \exp\left[-\frac{1}{2}(\mathcal{X} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\mathcal{X} - \boldsymbol{\mu})\right]$$

where $\boldsymbol{\mu}$ is the mean vector and $\boldsymbol{\Sigma}$ is the covariance matrix. This distribution can be written in the same way as we did in Equation (2.1), but using a density function instead of a mass function

$$f(\mathcal{X}) = f(X_1, X_2, \ldots, X_n) = f(X_1)f(X_2|X_1) \cdots f(X_n|X_1, X_2, \ldots, X_{n-1})$$

Equation (2.2) details the form of each conditional probability distribution:

$$f(X_i|X_1, \ldots, X_{i-1}) = \mathcal{N}(\beta_0 + \sum_{k=1}^{i-1} \beta_{ik} X_k; v_i) \tag{2.2}$$

where v_i represents the conditional variance of X_i given $X_1, \ldots X_{i-1}$ and β_{ik} represents the coefficient of X_k in the regression of X_i on $X_1, \ldots, X_{i-1}$.

A Gaussian Bayesian network is usually represented in one of two possible ways:

- By the unconditional mean vector $\boldsymbol{\mu}$ and the covariance matrix $\boldsymbol{\Sigma}$ of the joint probability distribution, or
- By $\boldsymbol{\mu}$, the conditional variances $\{v_i : i \in \{1, \ldots, n\}\}$ and the coefficients $\{\beta_{ij} : i, j \in \{1, \ldots, n\}\}$.

Both ways are equivalent, but the second one may be a better option in computational terms if the number of parents per variable is small because fewer elements are needed to fully represent it.

2.1.3 Learning the network

Given a data set $\mathcal{D}$ of independent instances of a vector of random variables $\mathcal{X}$, we are interested in finding the Bayesian network that best fits this data. For that purpose, this task is divided into two different subtasks: learning the graph structure of the network and learning the parameters that determine the conditional probability distributions.

2.1.3.1 Structure learning

The structure of a Bayesian network can be learned from data using different approaches. We can divide these approaches into two different groups: *constraint-based* methods and *score and search* methods.

Constraint-based methods consider triplets of variables from data and test conditional independence relations among the nodes in the triplet. Some examples of this type of algorithms are PC [6], *grow-shrink* [7], or different variants of the *incremental association Markov blanket* algorithm [8].

On the other hand, score and search methods measure each network structure in a particular space of structures using a score function and keep the network that maximizes this score. As searching for the structure that maximizes the score is an $\mathcal{NP}$-hard problem [9], [10], heuristic algorithms are used to find such a structure. *Hill climbing* or *greedy equivalence search* (GES) [11] are common search algorithms and BDeu and K2, popular scores.

2.1.3.2 Parameter learning

Learning the parameters of the network is the last step of the learning process. There are two main approaches: *maximum likelihood estimation* and *Bayesian estimation*.

Maximum likelihood estimation looks for the value $\hat{\boldsymbol{\theta}}$ that maximizes the likelihood function in Equation (2.3)

$$\mathcal{L}(\boldsymbol{\theta}|\mathcal{D},\mathcal{G}) = P(\mathcal{D}|\mathcal{G},\boldsymbol{\theta}) = \prod_{h=1}^{N} P(\mathbf{x}^h|\mathcal{G},\boldsymbol{\theta}) \tag{2.3}$$

where $\boldsymbol{\theta}$ is formed by $\theta_{ijk} = P(X_i = k|\mathbf{Pa}(X_i) = \mathbf{pa}_i^j)$, $i \in \{1,\ldots,n\}$. $\mathbf{pa}_i^j$ is the j-th instance of the possible values the parents of node X_i can take, and $\mathbf{x}^h$, the h-th instance of $\mathcal{D}$.

Maximizing $\mathcal{L}$ of Equation (2.3) [12] results in

$$\hat{\theta}_{ijk} = \frac{N_{ijk}}{N_ij}$$

where N_{ijk} represents the number of cases in $\mathcal{D}$ where $X_i = k$ and $\mathbf{Pa}(X_i) = \mathbf{pa}_i^j$, and N_{ij}, the number of cases where $\mathbf{Pa}(X_i) = \mathbf{pa}_i^j$.

In the Gaussian case, maximum likelihood estimation estimates the mean vector as the sample mean vector of the data. Conditional variances and coefficients β_{ik} are estimated performing the regression of X_i on its parents and keeping the sample conditional variance and the coefficients obtained.

Bayesian estimation represents a different approach for parametric learning. This method adds some prior knowledge about $\boldsymbol{\theta}$ using a probability distribution $f(\boldsymbol{\theta}|\mathcal{G})$. The posterior distribution given the data set $\mathcal{D}$, $f(\boldsymbol{\theta}|\mathcal{D},\mathcal{G})$, is computed using $P(\mathcal{D}|\mathcal{G},\boldsymbol{\theta})$ and the prior. Then, we can set $\hat{\boldsymbol{\theta}}$ as the posterior mean or MAP estimate, i.e.,

$$\hat{\boldsymbol{\theta}} = \int \boldsymbol{\theta}\, f(\boldsymbol{\theta}|\mathcal{D},\mathcal{G})\, d\boldsymbol{\theta} \quad \text{or} \quad \hat{\boldsymbol{\theta}} = \arg\max_{\boldsymbol{\theta}} f(\boldsymbol{\theta}|\mathcal{D},\mathcal{G})$$

The Dirichlet distribution is the common prior distribution used for discrete Bayesian networks.

2.1.4 Inference

If we know evidence about some variables in the network, $\mathbf{E} = \mathbf{e}$, the most common query we will be interested in has the form $P(X_i|\mathbf{E} = \mathbf{e})$. This probability can be computed as

$$P(X_i|\mathbf{E} = \mathbf{e}) = \frac{P(X_i, \mathbf{e})}{P(\mathbf{e})} \tag{2.4}$$

It is possible to calculate each instantiation of the numerator, $P(x_i, \mathbf{e})$, by summing out all entries that correspond to the set of unobserved variables $\mathbf{U} = \{\mathcal{X} \setminus (\mathbf{E} \cup \{X_i\})$ in the joint distribution, i.e., $P(x_i, \mathbf{e}) = \sum_{\mathbf{u}} P(x_i, \mathbf{e}, \mathbf{u})$. After that, it is easy to get the conditional probability of Equation (2.4) by dividing each instantiation by $P(\mathbf{e})$.

This method, referred to as *brute-force*, provides a way of answering any possible query of the form $P(X_i|\mathbf{E} = \mathbf{e})$, but it leads to the exponential blow-up of the joint distribution. To avoid this computational drawback, other methods are effective in most cases. We can't guarantee it for all situations because the problem of exact inference is $\mathcal{NP}$-hard [13]. The most common methods of exact inference are *variable elimination* [14] and *message passing* [2], [15], [16]. We will not explain these algorithms, but the main idea they handle is that computing and saving some expressions which only depend on some variables allow us not to compute them as many times as in the brute-force case.

2.1.4.1 Inference in Gaussian Bayesian networks

In the Gaussian case, exact inference becomes easier. Given some evidence $\mathbf{E} = \mathbf{e}$, we can get the distribution $f(X_i|\mathbf{E} = \mathbf{e})$ for each unknown variable X_i. It will be a Gaussian distribution determined by its mean μ'_{X_i} and variance $\sigma'^2_{X_i}$ and they both can be obtained from the previous distribution according to Equation (2.5) where μ_{X_i} and $\sigma^2_{X_i}$ are the unconditional mean and variance, $\mathbf{e}$ is the evidence and $\boldsymbol{\mu}_{\mathbf{E}}$ is the unconditional mean of the variables for which there is evidence.

$$\begin{aligned}
\mu'_{X_i} &= \mu_{X_i} + \mathbf{A}_{X_i\mathbf{E}}\mathbf{e} - \mathbf{A}_{X_i\mathbf{E}}\boldsymbol{\mu}_{\mathbf{E}} \\
\sigma'^2_{X_i} &= \sigma^2_{X_i} - \mathbf{A}_{X_i\mathbf{E}}\boldsymbol{\Sigma}_{\mathbf{E}X_i}
\end{aligned} \tag{2.5}$$

Equation (2.6) represents the matrices used in Equation (2.5) and the reordering of the elements in the covariance matrix given the observed ($\mathbf{E}$) and unobserved ($\mathbf{U}$) variables:

$$\mathbf{A}_{X_i\mathbf{E}} = \boldsymbol{\Sigma}_{X_i\mathbf{E}}\boldsymbol{\Sigma}^{-1}_{\mathbf{EE}}, \quad \boldsymbol{\Sigma} = \begin{pmatrix} \boldsymbol{\Sigma}_{\mathbf{UU}} & \boldsymbol{\Sigma}_{\mathbf{UE}} \\ \boldsymbol{\Sigma}_{\mathbf{EU}} & \boldsymbol{\Sigma}_{\mathbf{EE}} \end{pmatrix} \tag{2.6}$$

2.2 Gene regulatory networks

Gene regulatory networks (GRNs) are basic for understanding many different processes of life. Representing them is a central problem in molecular biology, cellular biology and biomedical sciences. Recent advances in technology, such as DNA microarrays and in-situ hybridization allow us to measure the expression levels of genes. After these advances, the problem of learning GRNs from data became computationally feasible and researchers started looking for models to learn them.

Given a set of n genes, a GRN is represented by a directed graph $\mathcal{G} = (V, E)$ with $|V| = n$. A directed edge from gene i to gene j represents a relationship between genes i and j. This relationship depends on the model, but it usually represents that i activates or inhibits the expression of gene j.

Different computational methods for learning GRNs from data have been studied [17] such as logical models, Boolean networks, differential equations, or linear models. Some of them behave better than others, but no model is a perfect fit in terms of faithfulness, amount of data needed, speed, or ability to perform inference. Among all these models, Bayesian networks have not shown particularly good performance in past studies such as the DREAM5 Challenge [18], but it is an easy-to-interpret model and network learning algorithms can still be improved. For these reasons, it remains as an interesting model for approximating GRNs.

2.2.1 GRNs modeled as Bayesian networks

Expression data is continuous data, so we present two different approaches to deal with it. The first one is considering the underlying distribution of each gene as a Gaussian distribution. As we said before, this assumption implies linear relationships between all the variables (genes) in the network. However, the normality assumption is common when dealing with expression data. The other option is to discretize the data into different categories. This process implies some information loss, but could be better than making a mistake when choosing the underlying distribution of the data.

2.2.1.1 Discrete case

Discretization techniques for gene expression data have been deeply studied and discussed [19]. Multiple approaches are available. However, we will follow the same simple discretization technique as in [20]. Other options may perform better, but we took this decision for simplicity.

For discretizing gene expression data, we take into account three categories: *downregulated* or *inhibition* levels, represented as -1; *upregulated* or *activation* levels, by +1; and *no-change* levels, by 0. For each gene, we call *control* the average expression level $\overline{x_i}$ of the gene across the instances of the data set. Given a data set $\mathcal{D} = \{\mathbf{x}^1, \ldots, \mathbf{x}^N\}$ of gene expression data, with $\mathbf{x}^k = (x_1^k, \ldots, x_n^k)$, for each $i \in \{1, \ldots, n\}$, we have

$$\overline{x_i} = \frac{1}{N} \sum_{k=1}^{N} x_i^k$$

Additionally, for the expression level of gene X_i in instance j, x_i^j, we consider the base 2 logarithm of the ratio between it and the *control* as shown in Equation (2.7).

$$r(x_i^j) = \log_2 \left(\frac{x_i^j}{\overline{x_i}} \right) \tag{2.7}$$

If this value is over a threshold $\delta > 0$, we assign x_i^j to the *activation* category. If it is lower than $-\delta$, to *inhibition*. Otherwise, it is assigned to *no-change*. In our case, the chosen value for δ is 0.2. After the discretization process we get a data set $\mathcal{D}' = \{\mathbf{x}'^1, \ldots, \mathbf{x}'^N\}$ such that for every x_i^j we have

$$x_i'^j = \begin{cases} +1 & \text{if } r(x_i^j) > \quad \delta \\ -1 & \text{if } r(x_i^j) < -\delta \\ 0 & \text{otherwise} \end{cases} = \begin{cases} +1 & \text{if } x_i^j > 2^{\delta}\overline{x_i} \\ -1 & \text{if } x_i^j < 2^{-\delta}\overline{x_i} \\ 0 & \text{otherwise} \end{cases} \tag{2.8}$$

Chapter 3

Previous software

This project is mainly based on and related to two neuroscience web applications: *NeuroSuites* and the `siibra` tool suite, which includes the EBRAINS Interactive Atlas Viewer, also known as `siibra-explorer`. This chapter details their essential characteristics, emphasizing those related to our objectives.

3.1 NeuroSuites and BayeSuites

NeuroSuites[1] is a platform that integrates multiple statistical and machine learning tools focused on neuroscience applications. It was developed by the Computational Intelligence Group of the Universidad Politécnica de Madrid starting in 2015 and funded by the Human Brain Project. One of the purposes of *NeuroSuites* is to provide easy-to-use tools that do not require coding. Among them, *BayeSuites* [1] collects all the functionality for learning, visualizing, and interpreting Bayesian networks. An example of the visualization of a Bayesian network with *NeuroSuites* is shown in Figure 3.1.

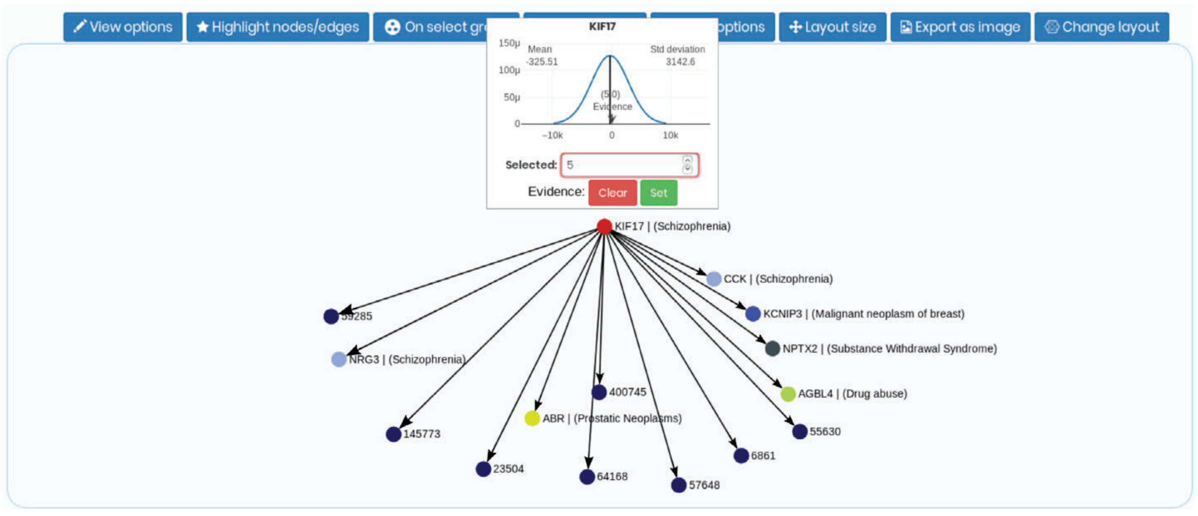


Figure 3.1. Example of Gaussian Bayesian network graph structure in *NeuroSuites*.

[1]https://neurosuites.com/

3.1.1 Implementation

BayeSuites is developed in Python and makes use of some powerful Python packages, such as `NetworkX` [21] for handling graphs, `pgmpy` [22] for parametric learning in the discrete case, `igraph` for computing graph layouts, or `NumPy` [23], `scikit-learn` [24], and `scipy` [25] for multiple scientific computations. It also uses `bnlearn` [26] and `sparsebn` [27], two Bayesian network libraries written in R, for most of the structure learning algorithms provided. Bindings to R code were done via `rpy2` [28].

The frontend of *NeuroSuites* is built using JavaScript. Particularly, for representing Bayesian networks, it uses *Sigma.js*, which is a Javascript library for graph visualization that makes use of *graphology*, another Javascript library for graph representation. *Sigma.js* renders graphs using *WebGL*, so it is a good option for large graphs visualization.

On the other hand, *NeuroSuites* backend is built following the Django framework with *Celery*, an open source asynchronous task queue written in Python, as its tasks queue and *RabbitMQ* as the message broker.

The full implementation of *NeuroSuites* is available in its GitLab repository[2].

3.1.2 Bayesian network learning

BayeSuites includes multiple structure learning algorithms. Table 3.1 summarizes all these algorithms, their types, and the kind of data they currently support. The reader can see that not only constraint-based and score and search methods are included, but also some Bayesian network classifiers (that require a class variable) and some statistical-based methods that are not Bayesian network specific, like the GENIE3 algorithm [29]. It is an algorithm that showed great performance in the DREAM5 challenge. Most of the algorithms are executed via `bnlearn`, but some others depend on `scikit-learn`, `sparsebn`, or `pgmpy`.

The implementation of the FGES-Merge algorithm [30] is one of the main features of *BayeSuites*. It is a score and search algorithm thought for learning large Bayesian networks. For that reason, it uses the Message Passing Interface (MPI) standard via `mpi4py`[31] for achieving efficient parallel computing.

Regarding the parametric learning process, *BayeSuites* includes maximum likelihood estimation and Bayesian estimation for the discrete case. `pgmpy` is the package used for this task. In the Gaussian case, maximum likelihood estimation is the only available method. *BayeSuites* provides its own implementation of this method because it best suited its needs.

3.1.3 Other features

BayeSuites also provides an input/output module, that let users read or write a Bayesian network in Bayesian Interchange Format (BIF). This is a format proposed by `pmgpy` and *BayeSuites* uses that implementation. Nevertheless, it is only available in the discrete case. There is also the option of exporting the graph structure of the network in CSV or Apache parquet format.

Additionally, *BayeSuites* includes different layout algorithms for the network structure graph. Most of them are executed with `igraph` [33]: circular, grid, Fruchterman-Reingold and Sugiyama layouts. Dot layout is included via `NetworkX`, ForceAtlas2 via `fa2` [34]. Finally, *BayeSuites* provides the implementation of their own layout, called image layout, that detects the outline of an image and displays the nodes over it.

[2]https://gitlab.com/mmichiels/neurosuite/

Learning type	Algorithm	Data type
Constraint-based	PC [6]	Gaussian and discrete
	Grow shrink [7]	Gaussian and discrete
	iamb [8]	Gaussian and discrete
	Fast.iamb [32]	Gaussian and discrete
	Inter.iamb [8]	Gaussian and discrete
Score and search	Hill climbing	Gaussian and discrete
	Hill climbing with tabu search	Gaussian and discrete
	Chow-Liu tree	Gaussian and discrete
	Hiton parents and children	Gaussian and discrete
	sparsebn [27]	Gaussian and discrete
	FGES-Merge [30]	Gaussian
Hybrid	MMHC	Gaussian and discrete
	MMPC [32]	Gaussian and discrete
Statistical-based	Pearson correlation	Gaussian
	Mutual information	Gaussian
	Linear regression	Gaussian
	Graphical lasso	Gaussian
	GENIE3 [29]	Gaussian
Bayesian classifiers	Naive Bayes	Discrete
	Tree augmented naive Bayes	Discrete
	Multidimensional BN classifier	Gaussian

Table 3.1. Structure learning algorithms available in *BayeSuites*.

BayeSuites only supports inference for the Gaussian case. To adapt well to the parameter learning output, *BayeSuites* own implementation is also used in this case. `NumPy` is the package used for developing it. It allows the user to condition on some evidence and retrieve new distributions of the form $f(X_i|\mathbf{E} = \mathbf{e})$.

Finally, some performance measures are provided too. Users can compare the learned network with another network introduced by them and get the confusion matrix, F1 score or accuracy measure, among others.

3.2 `siibra` and the EBRAINS Interactive Atlas Viewer

The software interface for interacting with brain atlases, shortened as `siibra`, is a neuroscience toolsuite developed by the Big Data Analytics Group of the Institute of Neuroscience and Medicine (INM-1), Forschungszentrum Jülich GmbH, in Jülich, Germany, as part of the Human Brain Project. It uses the EBRAINS brain atlas for providing different services. One of them is the EBRAINS Interactive Atlas Viewer or `siibra-explorer`, which is an interactive 3D Atlas Viewer. Apart from this, `siibra` also includes a Python client named `siibra-python`, which is the base of the viewer, and a tool for statistical analysis of differential gene expression named `siibra-jugex`.

3.2.1 siibra-explorer

`siibra-explorer`[3] is a web application for visualizing volumetric brain volumes at high resolutions. It stores its contents in the EBRAINS Knowledge Graph[4]. Additionally, it includes support for the openMINDS[5] metadata standards. Figure 3.2 shows the initial view of the human brain atlas in the explorer. All available atlases are depicted in Figure 3.3. Some of them are EBRAINS atlases, like in the human atlas case. Other atlases are borrowed, like the mouse brain atlas, which is openly shared by the Allen Brain Institute [35].

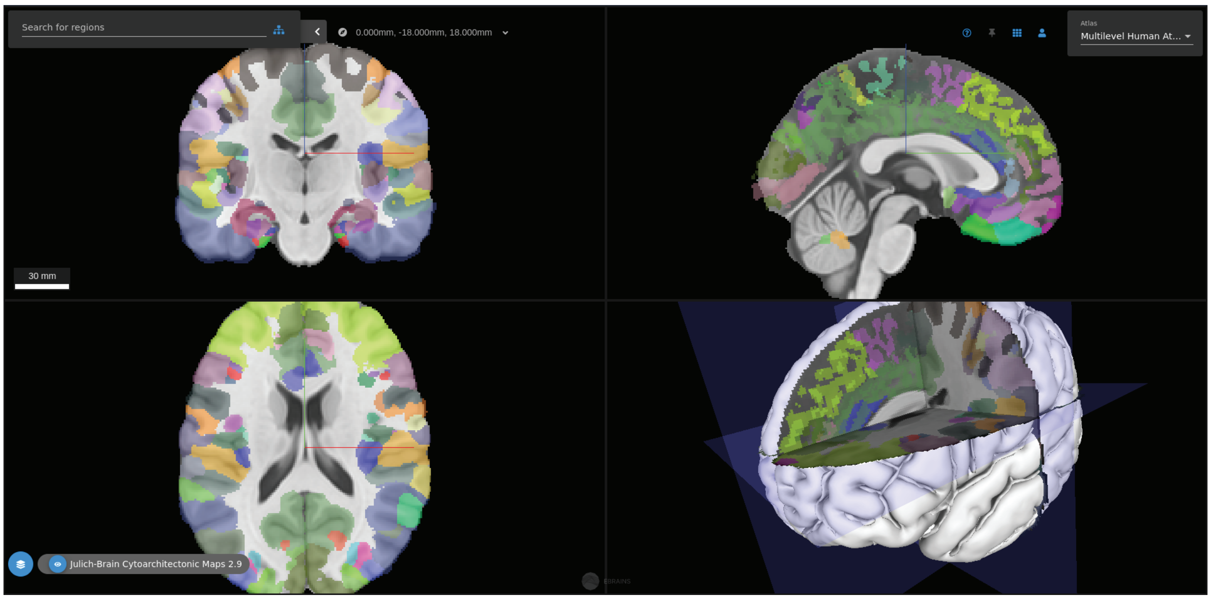

Search for regions
0.000mm, -18.000mm, 18.000mm
Atlas
Multilevel Human At...
30 mm
Julich-Brain Cytoarchitectonic Maps 2.9
EBRAINS

Figure 3.2. View of the human brain atlas in `siibra-explorer`.

The explorer also includes an easy way of adding plugins. Available plugins are listed in the plugins banner of Figure 3.4. These plugins need to comply to some specifications and have some restrictions, like a maximum height and width. There are currently two available plugins in the viewer, being `siibra-jugex` the one we are most interested in. Figure 3.5 shows its view in the explorer.

The main structure of a plugin for the latest version of `siibra-explorer` includes a "manifest.json" file with the following three fields:

- "name": the plugin name.
- "iframeUrl": it includes a reference to the HTML where the iframe is located.
- "siibra-explorer": the `siibra-explorer` version the plugin expects. It should be greater than 2.7.0 in that case.

3.2.2 siibra-python

`siibra-python`[6] is the Python client that manages all the interaction with brain atlas frameworks provided in the viewer. It tries to facilitate that interaction to researchers and provides some basic analysis tools.

[3]https://interactive-viewer.apps.hbp.eu/
[4]https://kg.ebrains.eu/
[5]https://github.com/HumanBrainProject/openMINDS
[6]https://siibra-python.readthedocs.io/en/latest/

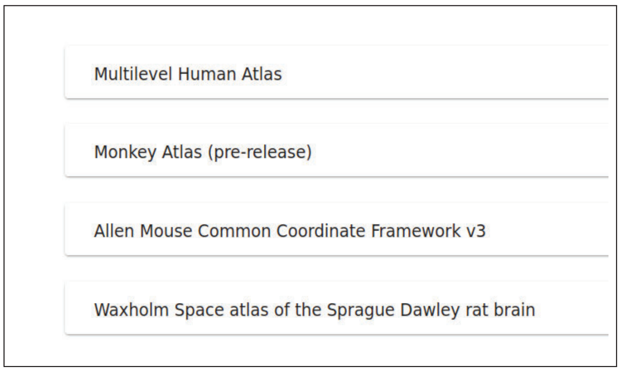

Multilevel Human Atlas
Monkey Atlas (pre-release)
Allen Mouse Common Coordinate Framework v3
Waxholm Space atlas of the Sprague Dawley rat brain

Figure 3.3. Available atlases in `siibra`'s front page.

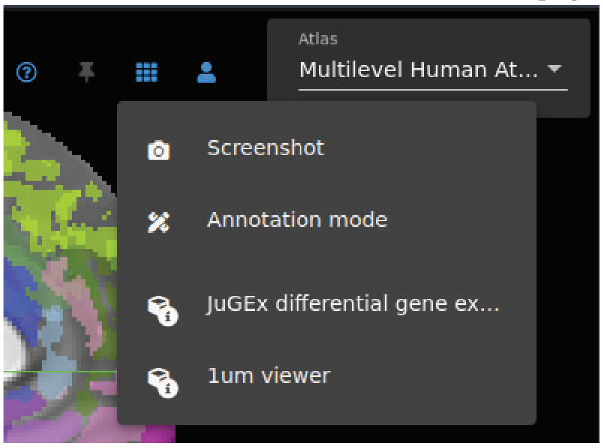


Figure 3.4. Plugins banner.

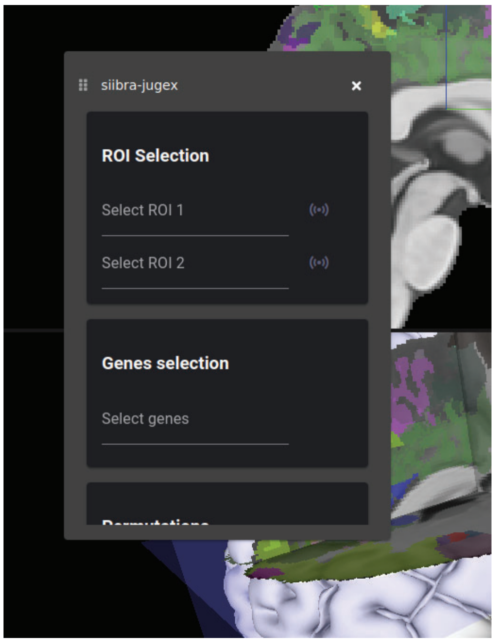


Figure 3.5. `siibra-jugex` plugin view.

One of the most important `siibra-python` features for us is its capability of querying gene expression data from the Allen brain atlas [35], [36]. This data is linked to regions and any query about a gene has to be performed specifying one region. There are six different donors and, for each donor, there are multiple instances that correspond to different locations in the same region. For each location, there are usually four different probes.

3.2.3 `siibra-jugex`

`siibra-jugex` [37] (as `siibra` Jülich-Brain Gene Expression) is a toolbox for statistical analysis of differential gene expression in the adult human brain. It is written in Python and uses `siibra-python` to retrieve gene expression data from the Allen brain atlas in the same way that we are interested in querying these data.

As we mentioned before, `siibra-jugex` includes a plugin implementation for the interactive viewer. The last version of this plugin uses the following software:

- *Svelte*, which is a modern open-source frontend compiler.
- *Celery* and *Redis* are used in the backend. *Celery* is its task queue and *Redis*, the message-broker for it.
- *Docker* is used to build the full application. In particular, it uses the `docker-compose` tool to build a multi-container Docker application with a server and a worker.

Attending to its design, the plugin uses *Material icons*, a set of material design icons provided by Google, and *Svelte Material UI*, a provider of Svelte components based on Material Design Components for Web.

Chapter 4

NeurogenPy

Once the theoretical background and related software to use was already clear, it was time to start working on the first objective of the project: the modularization of all the Bayesian network functionality explained in Section 3.1 in an independent Python package. This package is called `NeurogenPy` and from now on we will refer to it using this name. This chapter summarizes all the work done for achieving this first objective, the software artifacts obtained as the result of this process and a detailed user guide of the package.

4.1 Adapting and extending BayeSuites

During the development of `NeurogenPy`, we kept most of the implementation decisions made in the development of *BayeSuites*. However, in some cases, we had to modify *BayeSuites* in order to follow package production standards, make it easier to use, or get better performance. Furthermore, some new functionality has been added focusing on the needs of the later plugin deployment. Nevertheless, there are still some improvements and extensions that can be added in the near future as we will discuss in Chapter 6. We would also like to remark that the code has been written with this extension idea always in mind.

The first steps were about getting to know the inner characteristics of *BayeSuites* implementation and how to adapt them to fit Python package development standards. The subpackages structure was partially changed and the code was modified to make it more readable, gain simplicity or stick to the Python Enhancement Proposal 8 (PEP 8)[1], which establishes a style guide for Python code. In other cases, the code was extended to add new functionality as we explain in the following sections. Once each module was properly written, we started the documentation process. It followed the Python Enhancement Proposal 257 (PEP 257)[2], which sets docstring conventions. Docstrings are string literals that document modules, functions, classes, or method definitions. Then, automatic documentation of the project in formats such as HTML or PDF can be easily obtained from them. Different docstring styles are available, but we chose the `NumPy` documentation format (`numpydoc`) because it is the most widely used.

4.1.1 `BayesianNetwork` class

The core of the package is represented by the `BayesianNetwork` class. We tried to simplify it as much as it was possible, but it keeps similar attributes as before. One of the main goals

[1]https://peps.python.org/pep-0008/
[2]https://peps.python.org/pep-0257/

was to develop an easy to use framework, so we decided to allow the user to perform all the functionality provided by `NeurogenPy` using just one class and a small set of methods. For that reason, we created the methods `fit`, `save`, `load` and `compare` that allow the user not to directly handle the structure learning, parametric learning, input/output or score subpackages. The distributions subpackage is masked by some `BayesianNetwork` class methods as it was in *BayeSuites*. The only difference may be in the name or implementation of these methods, as they were changed along with the structure of the class.

The class now also includes a generalization of the data type in some cases, because discrete inference was included and the joint probability distribution attribute can be now Gaussian or discrete.

4.1.2 Structure learning

We showed in Table 3.1 all the structure learning algorithms supported by *BayeSuites* and the type of data they can be used on. `NeurogenPy` keeps all these algorithms and, as most of them were executed via third-party packages, we did not modify much. Nevertheless, it was not the case for the FGES-Merge algorithm. We split the code into three different files: one for the FGES [38] algorithm (with BIC scores rather than pairwise MI as explained in [30]), another one for the FGES-Merge algorithm, and the last one for the core implementation both algorithms use. Message Passing Interface via `mpi4py` was replaced by the `multiprocessing` package. The algorithm now uses a pool and the number of processes to use can be set by the user with the `n_jobs` attribute. The reason for the change is because it also provides parallel computation and we believe it represents a better option for parallelism in a package. Nevertheless, it can be easily modified as `mpi4py` provides a similar pool class. On the other hand, we kept `numba` usage for speeding up some functions. Temporary graphs in FGES-Merge are now saved with the `tempfile` package using `TemporaryDirectory` and `NamedTemporaryFile` classes. Some other minor changes were also included, such as the use of `tqdm` for logging information in potentially long-time tasks.

4.1.3 Joint probability distributions

BayeSuites provided an inference subpackage for Gaussian Bayesian networks. We kept this functionality but introduced a generalization to include the discrete case. A new abstract class `JPD` represents a joint probability distribution and provides some abstract methods needed by any joint distribution. They are mainly `from_parameters`, for building the joint distribution given the conditional distributions; `condition`, for conditioning on some evidence; `marginal`, for retrieving marginal distributions; and `get_cpd`), for getting conditional distributions. The previous Gaussian case is now included in the class `GaussianJPD` with some changes: the distribution in the case of a large graph is now stored in a temporary file to ease the memory usage thanks to the `tempfile` Python package.

Finally, a new `DiscreteJPD` class has been created. It provides the four aforementioned functions via `pgmpy`. *BayeSuites* parametric learning subpackage used `pgmpy` in the discrete case and the conditional probability tables were represented using `pgmpy` class `TabularCPD`. Then, we decided to stick to this package and use the discrete case inference functionality they provide. In this joint distribution class, we do not build the full JPD, but keep all the `TabularCPD` objects and use variable elimination or message passing algorithms, depending on the user's desire, to get it in the `condition` function.

4.1.4 Input/output formats

BayeSuites provided three file formats for the input/output subpackage: CSV, Apache Parquet, and BIF for the discrete case. In `NeurogenPy`, we kept these formats but added two more that were useful for us, especially for visualization. Additionally, we related the visualization oriented format of these two to the already existing layouts module.

The first format is the Graph Exchange XML Format (GEXF), designed to represent complex network structures and supported by some of the most popular graph tools, such as `NetworkX`, *Gephi*, *Sigma.js*, or *graphology*. Taking into account that we represent the graph structure with a `NetworkX` graph and they provide GEXF read and write functions, it was easy to properly tweak and export or import it using them. GEXF format addition represents a simple way to visualize the network using other tools or load structures created out of the package. In this case, all export functions allow the user to set the desired layout for setting the nodes positions.

The second format is the JavaScript Object Notation (JSON), which is a text format for data interchange. As `NetworkX` has JSON read/write functionality, it was easy to convert the network parameters into JSON format and join them with the graph in a unique JSON object. This way, we provide an option for Bayesian network objects serialization that can be used in the plugin implementation.

All direct visualization tools included in *BayeSuites* Python code were removed because we believe they are not useful in our case. As we detail in Section 4.3, it is easy to visualize the structure in small networks using `NetworkX draw` function. In large networks, exporting the structure as a GEXF file and visualizing it using a specific tool for large graphs visualization like Gephi is the approach we recommend.

4.1.5 Minor changes

Apart from the already explained modifications, some other small and technical changes in the code were needed to get everything working. For example, the util subpackage was organized and all the data structures transformations needed were put together in the same module. In the layout subpackage, all layout classes that used `igraph` were joined in a `IgraphLayout` class and all layouts provided by this package are now available. The full list of 2D layout provided by `igraph` is formed by automatic, bipartite, circular, Davidson-Harel, DrL, Fruchterman-Reingold, grid, graphopt, Kamada-Kawai, large graph, multidimensional scaling, random, Reingold-Tilford tree, circular Reingold-Tilford, star and Sugiyama layouts. Finally, we added the use of `logging` to handle tracking events.

4.2 Results

The final Python package code is available in its GitHub repository[3]. It includes all the code, documentation, configuration, examples, and necessary files that are usually included in Python packages. The full documentation[4] of the project is hosted on *Read the docs*. This documentation has been produced using `sphinx`, an automatic tool for generating documentation from docstrings, with the `sphinx_rtd_theme` and `numpydoc` validation. The package has been licensed using a GNU General Public License Version 3 license (GPLv3) that complies with the

[3]https://github.com/javiegal/neurogenpy

[4]https://neurogenpy.readthedocs.io/en/latest/

licenses of all the used packages. Finally, `NeurogenPy` is also included in *PyPI*[5] and it can be installed running `pip install neurogenpy`.

4.2.1 Summary

Table 4.2 summarizes all our contributions during `NeurogenPy` development process.

Contribution	Description
***BayeSuites* adaptation**	Code cleaning and modifications of *BayeSuites* functionality to get a usable package.
Input/output extension	JSON and GEXF formats added. String read/write added.
FGES adaptation	Modularization and code cleaning of FGES algorithm implementation.
Discrete inference	Modularization of joint distributions and discrete inference added using `pgmpy`.
Layout subpackage	Code restructured and more `igraph` layouts added.
`logging` **use**	Tracking events added with information, warnings and errors.
Docstrings and web documentation	Docstrings added to all methods according to PEP 257 guide and documentation uploaded to *Read the docs*.
Notebooks examples	Jupyter notebooks (available in the documentation too) developed as examples of package use.
Package structure	README, license, setup, requirements, tests, __init__.py, and some other typical Python package needed files added.

Table 4.2. Contributions to `NeurogenPy` development.

4.3 User guide

This user guide shows easy examples to illustrate how to use the package. They include code and console output. We believe it is easy to follow for anybody with basic Python knowledge.

4.3.1 Bayesian network creation

The use of the package is focused on the `BayesianNetwork` class. The two main ways of creating new networks are using the constructor, in case the user already has a graph structure or parameters, and learning it from data using `fit` function.

4.3.1.1 Using the constructor

If the user already has a graph structure and the network parameters in the right formats, it is posible to use the constructor for creating the network object. The graph structure is represented using a `DiGraph` object from the `NetworkX` package.

[5]https://pypi.org/project/neurogenpy/

```
[ ]: from networkx import DiGraph

     graph = DiGraph()
     graph.add_nodes_from([1, 2])
     graph.add_edges_from([(1, 2)])
```

Gaussian case

The network parameters are represented with a dictionary where the keys are the identifiers of the nodes (they must be the same as in the DiGraph object) and the values are dictionaries with four keys: uncond_mean, cond_var, parents and parents_coeffs. For each node, these elements represent the unconditional mean, conditional variance, parents and coefficients in the regression of the node on its parents.

```
[ ]: parameters = {1: {'uncond_mean': 0, 'cond_var': 1, 'parents_coeffs': [],
      ↪'parents': []},
                   2: {'uncond_mean': 0, 'cond_var': 1, 'parents_coeffs': [1],
      ↪'parents': [1]}}
```

Discrete case

In the discrete case, we use pgmpy as the core package, and the parameters of the network are TabularCPD objects from pgmpy.factors.discrete.CPD. They represent conditional probability tables. Suppose node 1 has three possible categories (0, 1 and 2). See its conditional probability distribution in Table 4.3. For node 2, suppose it also has three possible categories and its conditional probability distribution conditioned on node 1 is the one in Table 4.4.

	Prob
1(0)	0.3
1(1)	0.3
1(2)	0.4

Table 4.3. CPD table for node 1.

1	**1(0)**	**1(1)**	**1(2)**
2(0)	0.2	0.05	0.1
2(1)	0.2	0.5	0.1
2(2)	0.6	0.45	0.8

Table 4.4. CPD table for node 2.

```
[ ]: from pgmpy.factors.discrete.CPD import TabularCPD

     cpd1 = TabularCPD(1, 3, [[0.3], [0.3], [0.4]])
     cpd2 = TabularCPD(2, 3, [[0.2,0.05,0.1], [0.2,0.5,0.1],[0.6,0.45,0.8]],␣
      ↪evidence=[1], evidence_card=[3])

     parameters = {1: cpd1, 2: cpd2}
```

Once you have both the graph and parameters, the network can be instantiated in the usual way. In the discrete case, the user needs to pass data_type='discrete' as an argument, and in the continuous case, data_type='continuous'.

```
[ ]: from neurogenpy import BayesianNetwork

     bn = BayesianNetwork(graph=graph, parameters=parameters, data_type='continuous')

     print('Nodes:', bn.graph.nodes())
     print('Edges:', bn.graph.edges())
```

4.3.1.2 Learning the full network from data

As said before, it is possible to learn the structure and parameters of a Bayesian network from data. First of all, you should create a pandas `DataFrame` from your data with the following structure:

Instances	**Feature 1**	**Feature 2**	...	**Feature n**
Instance 1	$Value_{11}$	$Value_{12}$	...	$Value_{1n}$
Instance 2	$Value_{21}$	$Value_{22}$	...	$Value_{2n}$
...	...	...	...	...
Instance N	$Value_{N1}$	$Value_{N2}$	...	$Value_{Nn}$

In our example, we create the `DataFrame` by reading a CSV file.

```
[ ]: import pandas as pd

     df = pd.read_csv('data.csv')
```

Once the data is in the correct format, there are two ways for learning the network: using the arguments of the `fit` function or using the particular `LearnStructure` and `LearnParameters` subclasses. They are analogous and we particularly recommend the first one as it is simpler.

Using arguments of `fit` function

Once users have read the file, they can fit the data using the `fit` method and setting the structure learning algorithm and estimation method.

```
[ ]: bn = BayesianNetwork().fit(df, data_type='continuous', estimation='mle',␣
     ↪algorithm='pc')
```

Additional parameters for these learning methods can be provided too.

```
[ ]: bn = BayesianNetwork().fit(df, data_type='continuous', estimation='mle',␣
     ↪algorithm='fges_merge', penalty=0.01)
```

Using `LearnStructure` or `LearnParameters` subclasses

Another option is to use the desired subclass of `LearnStructure` or `LearnParameters`.

```
[ ]: from neurogenpy import PC, GaussianMLE

     pc = PC(df, data_type='continuous')

     mle = GaussianMLE(df)

     bn = BayesianNetwork().fit(algorithm=pc, estimation=mle)
```

4.3.1.3 Combinations

Users can handle combinations of the above methods to build a network. If they are only interested in the graph structure, it is possible to just learn the structure and not the parameters by not providing any value for the attribute `estimation`.

```
[ ]: bn = BayesianNetwork().fit(df, data_type='continuous', algorithm='pc')
```

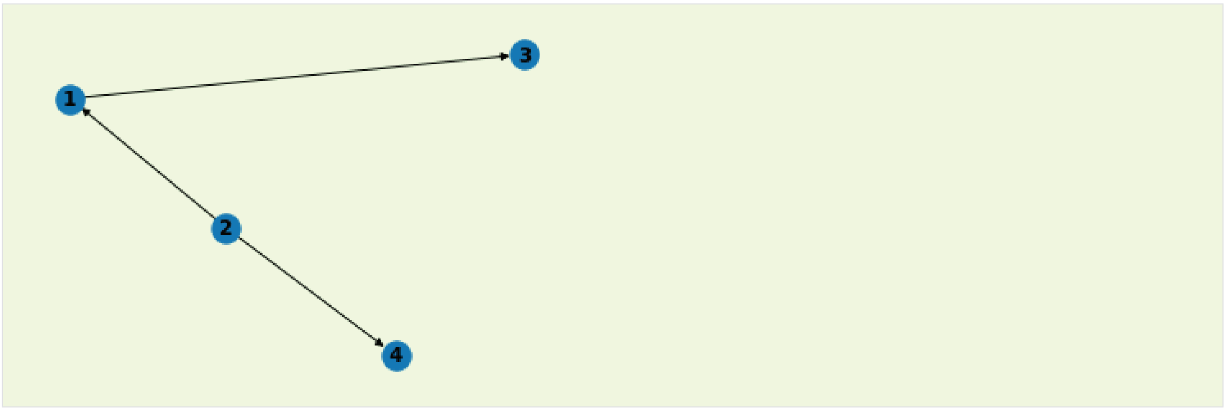
On the other hand, if the user already has a graph structure and want to learn the parameters, it is also possible to provide it in the constructor or load it before calling `fit` with `skip_structure` set to `True`.

```
[ ]: bn = BayesianNetwork(graph=graph)
     bn.fit(df, data_type='continuous', estimation='mle', skip_structure=True)

     bn2 = BayesianNetwork().load('adjacency_matrix.csv')
     bn2.fit(df, data_type='continuous', estimation='mle', skip_structure=True)
```

4.3.2 Visualization

The graph structure of the network is represented using a `DiGraph` from `NetworkX`. This package provides functionality for visualizing basic graphs. To plot the graph structure, the user will also need to use `matplotlib` [39].

```
[4]: import matplotlib.pyplot as plt
     import networkx as nx
     from neurogenpy import BayesianNetwork

     digraph = nx.DiGraph()

     digraph.add_edge(2, 1)
     digraph.add_edge(1, 3)
     digraph.add_edge(2, 4)

     bn = BayesianNetwork(graph=digraph)

     nx.draw(bn.graph, with_labels=True, font_weight='bold')
     plt.show()
```



This way of visualizing the network is only recommended for small networks. As graph visualization is a difficult task, we recommend exporting the structure in GEXF format and use a dedicated tool to show it, such as *Gephi* or *Sigma.js*. Exporting the graph structure is carefully explained in the next section.

4.3.3 Load and save networks

Bayesian networks can be stored and loaded from the `BayesianNetwork` class or using the classes of the io subpackage (`BIF`, `AdjacencyMatrix`, `GEXF`, or `JSON`).

4.3.3.1 Using `BayesianNetwork` class

The network can be read or written using `load` and `save` methods, respectively. Some formats save or load the full Bayesian network and some others can only handle the structure.

Full network

There are two formats that manage the full network: JSON and `pgmpy` BIF. The last one is only available in the discrete case.

[1]:
```
from neurogenpy import BayesianNetwork
from networkx import DiGraph

graph = DiGraph()
graph.add_edge('A', 'C')
graph.add_edge('B', 'C')
graph.add_edge('D', 'E')
parameters = {'A': {'uncond_mean': 4, 'cond_var': 3, 'parents_coeffs': [],
↪'parents': []},
              'B': {'uncond_mean': 5, 'cond_var': 1, 'parents_coeffs': [],
↪'parents': []},
              'C': {'uncond_mean': 3, 'cond_var': 2, 'parents_coeffs': [-0.2, 0.
↪5], 'parents': ['A', 'B']},
              'D': {'uncond_mean': 2, 'cond_var': 1, 'parents_coeffs': [],
↪'parents': []},
              'E': {'uncond_mean': 1, 'cond_var': 0.5, 'parents_coeffs': [0.7],
↪'parents': ['D']}}
```

(continues on next page)

(continued from previous page)

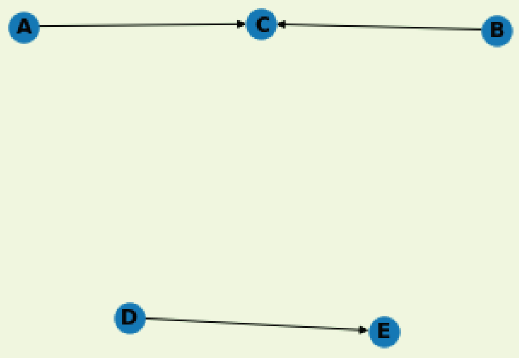
```
bn = BayesianNetwork(graph=graph, parameters=parameters, data_type='continuous')

bn.save('bn.json')

bn2 = BayesianNetwork().load('bn.json')

print('Some checking:')
print(bn2.get_cpds(['C']))
```

```
Some checking:
{'C': {'uncond_mean': 3.0, 'cond_var': 2.0, 'parents_coeffs': [-0.
↪20000000000000004, 0.5], 'parents': ['A', 'B']}}
```

Network structure

In the case of a GEXF, CSV, or Apache Parquet file, it only loads the graph structure of the network. In the GEXF case, a `layout_name` argument in `save` allows the user to determine how the positions of the nodes are stored in the file. Loading a GEXF file also keep the positions written in the file, but the user should be careful because `draw` function from `NetworkX` do not take them into account. In the following example, we save a graph using circular layout and show the visualization obtained with `draw` after loading it.

```
[4]: bn.save('bn.gexf', layout_name='circular')

bn2 = BayesianNetwork().load('bn.gexf')

import matplotlib.pyplot as plt
import networkx as nx

nx.draw(bn.graph, with_labels=True, font_weight='bold')
plt.show()
```



The display obtained is not the desired one. We would expect something like the graph in Figure 4.1. It was drawn with *Sigma.js* reading the stored GEXF file.

An option to make it work with `NetworkX` could be using a layout object, running the layout and passing the result to the `draw` function.

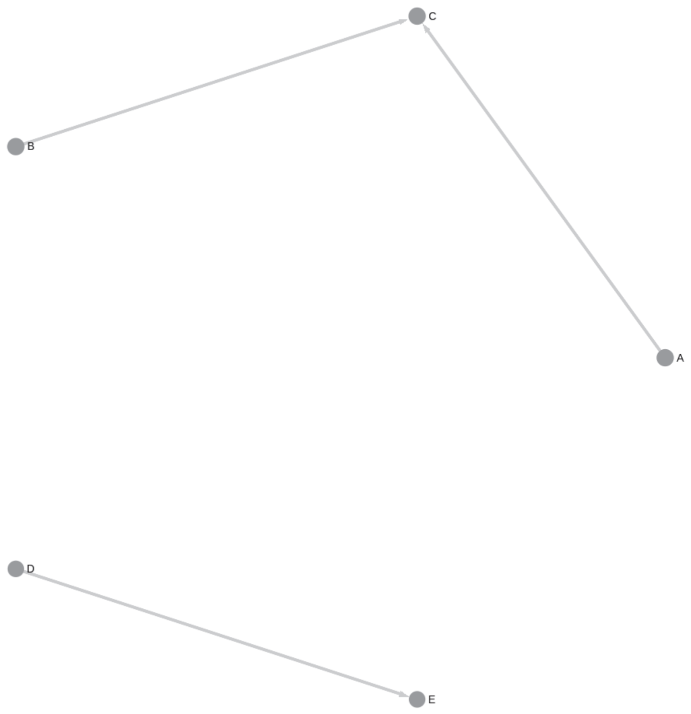
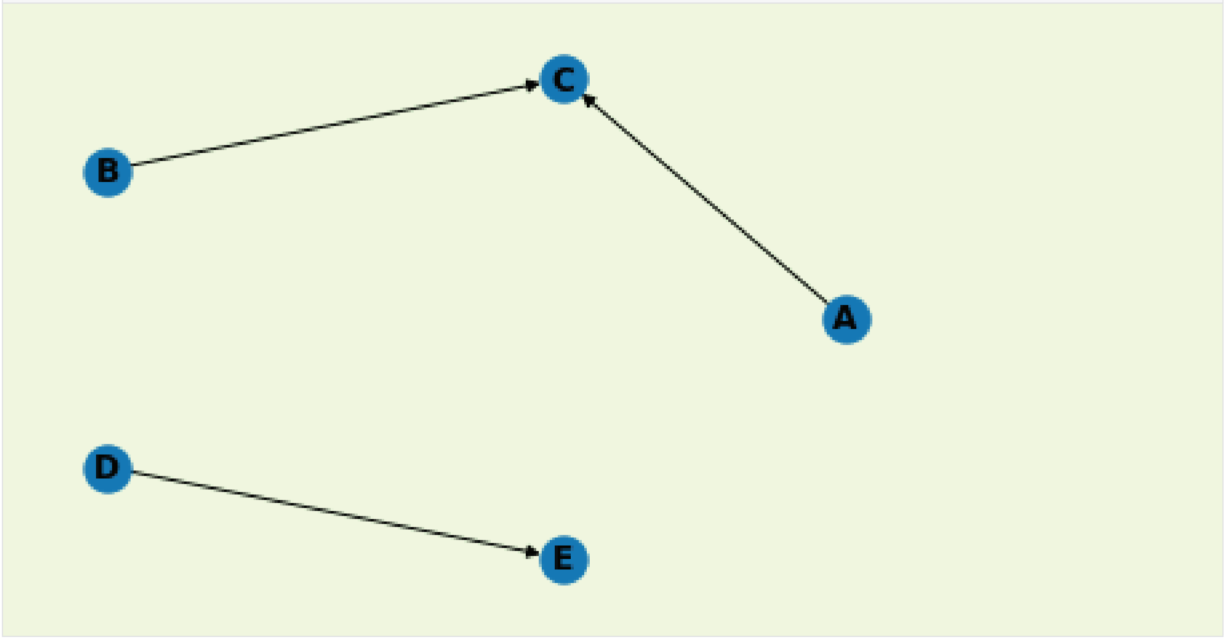


Figure 4.1. Example of circular layout visualization with *Sigma.js*.

```
[15]: from neurogenpy.io import IgraphLayout

      layout = IgraphLayout(bn.graph, layout_name='circular')
      positions = layout.run()

      nx.draw(bn.graph, pos=positions, with_labels=True, font_weight='bold')
      plt.show()
```



4.3.3.2 Using io subpackage

It is also possible to instantiate a particular `io` class (`JSON`, `GEXF`, `AdjacencyMatrix` or `BIF`) and use `write_file` or `read_file` to load or save Bayesian networks.

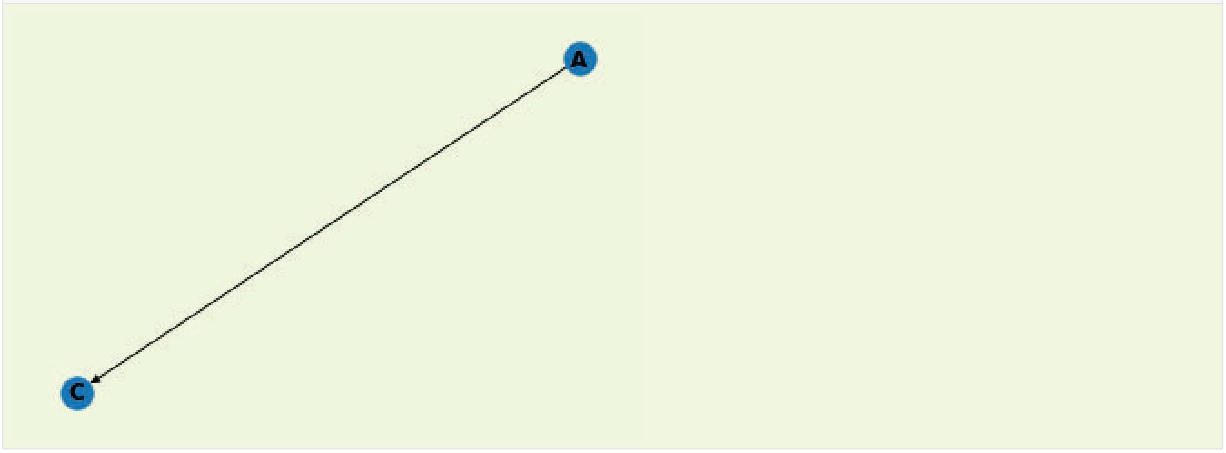
```
[ ]: from neurogenpy import GEXF, AdjacencyMatrix

     writer = GEXF(bn)
     writer.write_file(layout_name='circular', communities=True, sizes_method=
      ↪'neighbors')

     reader = AdjacencyMatrix()
     bn_structure = reader.read_file('bn_structure.csv')
```

Additionally, the `io` subpackage provides two other methods: `generate` and `convert`. The first one allows the user to get an input/output object from the Bayesian network and the second one to get a Bayesian network (or at least its structure) from an input/output object. In the GEXF and JSON case the input/output object is a string representation of the structure. In the AdjacencyMatrix case, it is a numpy array that represents it. We now show one example of each method.

```
[3]: from neurogenpy import JSON, GEXF

     writer = GEXF(bn)

     print('GEXF representation:')
     print(writer.generate())
```

```
GEXF representation:
<gexf xmlns="http://www.gexf.net/1.2draft" xmlns:xsi="http://www.w3.org/2001/
↪XMLSchema-instance" xsi:schemaLocation="http://www.gexf.net/1.2draft http://
↪www.gexf.net/1.2draft/gexf.xsd" version="1.2">
  <meta lastmodifieddate="2022-07-12">
    <creator>NetworkX 2.8.4</creator>
  </meta>
  <graph defaultedgetype="directed" mode="static" name="">
    <nodes>
      <node id="A" label="A" />
      <node id="C" label="C" />
      <node id="B" label="B" />
      <node id="D" label="D" />
      <node id="E" label="E" />
    </nodes>
    <edges>
      <edge source="A" target="C" id="0" />
      <edge source="B" target="C" id="1" />
      <edge source="D" target="E" id="2" />
    </edges>
  </graph>
</gexf>
```

[4]:
```
json_str = r"""{"graph": {"directed": true, "multigraph": false, "graph": {},
↪"nodes": [{"id": "A"}, {"id": "C"}],
"links": [{"source": "A", "target": "C"}]},
"parameters": {
    "A": {"uncond_mean": 4, "cond_var": 3, "parents_coeffs": [], "parents": []},
    "C": {"uncond_mean": 3, "cond_var": 2, "parents_coeffs": [-0.2], "parents":␣
↪["A"]}},
"data_type": "continuous"}"""
reader = JSON()
bn2 = reader.convert(json_str)


nx.draw(bn2.graph, with_labels=True, font_weight='bold')
plt.show()
```



4.3.4 Bayesian network information

Much information can be retrieved once you have created a network with available methods like `markov_blanket`, `important_nodes`, `communities`, `marginal`, `is_dseparated`, `get_cpds`, etc. For example, the following queries output as a dictionary the communities (calculated with the Louvain algorithm [40]), the Markov blanket of node A and checks if A and C are d-separated by B.

[1]:
```
from neurogenpy import BayesianNetwork
from networkx import DiGraph

graph = DiGraph()
graph.add_edge('A', 'B')
graph.add_edge('B', 'C')
graph.add_edge('D', 'E')
parameters = {'A': {'uncond_mean': 4, 'cond_var': 3, 'parents_coeffs': [],
↪'parents': []},
              'B': {'uncond_mean': 5, 'cond_var': 1, 'parents_coeffs': [0.5],
↪'parents': ['A']},
              'C': {'uncond_mean': 3, 'cond_var': 2, 'parents_coeffs': [-0.2],
↪'parents': ['B']},
```

(continues on next page)

(continued from previous page)

```
              'D': {'uncond_mean': 2, 'cond_var': 1, 'parents_coeffs': [],
↪'parents': []},
              'E': {'uncond_mean': 1, 'cond_var': 0.5, 'parents_coeffs': [0.7],
↪'parents': ['D']}}

bn = BayesianNetwork(graph=graph, parameters=parameters, data_type='continuous')
print('- Communities Louvain:', bn.communities(method='louvain'))
print('- Markov blanket of node A', bn.markov_blanket('A'))
print('- A and C d-separated by B?', bn.is_dseparated(['A'], ['C'], ['B']))
```

```
- Communities Louvain: {'A': 1, 'B': 1, 'C': 1, 'D': 0, 'E': 0}
- Markov blanket of node A ['A', 'B']
- A and C d-separated by B? True
```

4.3.5 Distributions and inference

The joint distribution is stored using a JPD subclass. Depending on the data type, the used class will be GaussianJPD or DiscreteJPD. However, there is no need to directly use them.

These classes provide some methods for getting the joint distribution object from parameters (from_parameters), conditioning the distribution on some evidence (condition), or retrieving conditional or marginal distributions (get_cpds or marginal). These methods are called from the corresponding functions provided in the BayesianNetwork class. We will show examples of condition and marginal in Gaussian and discrete cases.

4.3.5.1 Gaussian joint probability distribution

```
[1]: from neurogenpy import BayesianNetwork
from networkx import DiGraph

graph = DiGraph()
graph.add_edge('A', 'B')
graph.add_edge('B', 'C')
graph.add_edge('D', 'E')
parameters = {'A': {'uncond_mean': 4, 'cond_var': 3, 'parents_coeffs': [],
↪'parents': []},
              'B': {'uncond_mean': 5, 'cond_var': 1, 'parents_coeffs': [0.5],
↪'parents': ['A']},
              'C': {'uncond_mean': 3, 'cond_var': 2, 'parents_coeffs': [-0.2],
↪'parents': ['B']},
              'D': {'uncond_mean': 2, 'cond_var': 1, 'parents_coeffs': [],
↪'parents': []},
              'E': {'uncond_mean': 1, 'cond_var': 0.5, 'parents_coeffs': [0.7],
↪'parents': ['D']}}

bn = BayesianNetwork(graph=graph, parameters=parameters, data_type='continuous')

print('Marginal distribution f(B):')
```

(continues on next page)

(continued from previous page)

```
print(bn.marginal(['B'])['B'])

bn.set_evidence({'A': 1})
print('\nNew distribution f(B|A=1):')
print(bn.condition()['B'])
```

```
Marginal distribution f(B):
{'mu': 5.0, 'sigma': 1.75}

New distribution f(B|A=1):
{'mu': 3.5, 'sigma': 1.0}
```

4.3.5.2 Discrete joint probability distribution

[2]:
```
from pgmpy.factors.discrete.CPD import TabularCPD

graph = DiGraph()
graph.add_edge('A', 'B')

cpd1 = TabularCPD('A', 3, [[0.3], [0.3], [0.4]])
cpd2 = TabularCPD('B', 3, [[0.1,0.4,0.1], [0.1,0.3,0.1],[0.8,0.3,0.8]],␣
↪evidence=['A'], evidence_card=[3])

parameters = {'A': cpd1, 'B': cpd2}

bn = BayesianNetwork(graph=graph, parameters=parameters, data_type='discrete')

print('Marginal distribution f(B):')
print(bn.marginal(['B'])['B'])

bn.set_evidence({'A': 1})
print('\nNew distribution f(B|A=1):')
print(bn.condition()['B'])
```

```
Marginal distribution f(B):
{0: 0.19999999999999998, 1: 0.16666666666666666, 2: 0.6333333333333334}

New distribution f(B|A=1):
{0: 0.4, 1: 0.3, 2: 0.3}
```

4.3.6 Performance

The user can compare the graph structure of the network with its actual graph structure using `compare` function from `BayesianNetwork` class. Different performance measures are available. Let's see an example where the graph structure of the network is 1→2→3→4 and the actual graph structure to be approximated is 1→2←3→4. Considering the edges directions (using `undirected=False`) shows a different result than not doing it (`undirected=True`), where both structures would be seen as the same.

```
[1]: from numpy import array
     from networkx import DiGraph
     from neurogenpy import BayesianNetwork

     matrix = array([[0,1,0,0], [0,0,0,0], [0,1,0,1], [0,0,0,0]])
     graph = DiGraph()
     graph.add_edge(1, 2)
     graph.add_edge(2, 3)
     graph.add_edge(3, 4)

     # No JPD is needed for this, just the graph structure.

     bn = BayesianNetwork(graph=graph)
     res = bn.compare(matrix, nodes_order=[1, 2, 3, 4], metric='all',␣
      ↪undirected=False)

     print('Directed case:')
     print('Accuracy:', res['accuracy'])
     print('Confusion matrix:')
     print(res['confusion'])

     res = bn.compare(matrix, nodes_order=[1, 2, 3, 4], metric='all',␣
      ↪undirected=True)
     print('\nUndirected case:')
     print('Accuracy :', res['accuracy'])
     print('Confusion matrix:')
     print(res['confusion'])
```

```
Directed case:
Accuracy: 0.8333333333333334
Confusion matrix:
[[2 1]
 [1 8]]

Undirected case:
Accuracy : 1.0
Confusion matrix:
[[3 0]
 [0 3]]
```

4.3.7 Using siibra-python to learn gene regulatory networks

siibra-python allows users to query gene expression data from the Allen brain atlas. Their documentation[6] provides a full explanation on how it works.

First of all, you should determine the atlas, region and genes of interest. After that, you can use siibra's get_features function to retrieve the desired data. The data provided comes from six different donors. Given a region, multiple samples from different locations are obtained for

[6]https://siibra-python.readthedocs.io/en/latest/examples/03_data_features/004_gene_expressions.html

each donor. Each sample provides data from different probes (usually four probes). We take the average for all the probes as the value for a particular donor and location.

[1]:
```
import siibra
import statistics
import pandas as pd
from neurogenpy import BayesianNetwork

atlas = siibra.atlases.MULTILEVEL_HUMAN_ATLAS
region = atlas.get_region("V1")

genes = ["CREM", "ATP5G1", "RAB33B"]


samples = {gene_name: [statistics.mean(f.expression_levels) for
                      f in siibra.get_features(region, 'gene',
                                                   gene=gene_name)] for
           gene_name in genes}
```

```
[siibra:INFO] Version: 0.3a14
[siibra:WARNING] This is a development release. Use at your own risk.
[siibra:INFO] Please file bugs and issues at https://github.com/FZJ-INM1-BDA/
↪siibra-python.
[siibra:INFO] No parcellation specified, using default 'Julich-Brain␣
↪Cytoarchitectonic Maps 2.9'.
[siibra:INFO] Retrieving probe ids for gene CREM
```

```
    For retrieving microarray data, siibra connects to the web API of
    the Allen Brain Atlas (© 2015 Allen Institute for Brain Science),
    available from https://brain-map.org/api/index.html. Any use of the
    microarray data needs to be in accordance with their terms of use,
    as specified at https://alleninstitute.org/legal/terms-use/.
```

```
[siibra:INFO] Retrieving probe ids for gene ATP5G1
[siibra:INFO] Retrieving probe ids for gene RAB33B
```

Then, you can learn the network with the `BayesianNetwork` class after creating a pandas `DataFrame`.

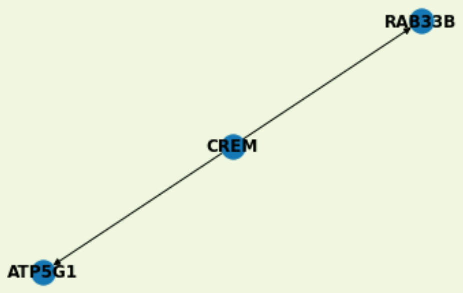
[2]:
```
df = pd.DataFrame(samples)
print('First five instances of the obtained DataFrame:')
print(df.head())

bn = BayesianNetwork().fit(df, data_type="continuous", algorithm="cl",␣
↪estimation="mle")


import networkx as nx
import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```
nx.draw(bn.graph, with_labels=True, font_weight='bold')
plt.show()

for gene, cpd in bn.get_cpds(genes).items():
    print(gene, cpd)
```

```
First five instances of the obtained DataFrame:
       CREM    ATP5G1   RAB33B
0  5.389167  9.709525  3.81760
1  5.435017  9.666150  3.57550
2  5.475183  9.812725  3.67285
3  5.418250  9.954475  3.72965
4  5.440733  9.008950  4.49230
```



```
CREM {'uncond_mean': 5.433495238095238, 'cond_var': 0.04456554986049554,
↪'parents_coeffs': [], 'parents': []}
ATP5G1 {'uncond_mean': 9.024532142857145, 'cond_var': 0.39210540972760805,
↪'parents_coeffs': [-0.8244179397109443], 'parents': ['CREM']}

RAB33B {'uncond_mean': 4.281339999999999, 'cond_var': 0.6899283651657337,
↪'parents_coeffs': [-0.29881861893417766], 'parents': ['CREM']}
```

4.3.7.1 Discretization of gene expression data

There are multiple ways of discretizing gene expression data. Here, we use a simple one. We use three levels (*inhibition*, *activation* and *no-change*). See Section 2.2.1.1 for a complete explanation of this process.

[3]:
```
df = df.apply(lambda col: pd.cut(col,
                                 bins=[-float('inf'), 2 ** (-0.2) * col.mean(),
↪2 ** 0.2 * col.mean(), float('inf')],
                                 labels=['inh', 'no-c', 'act']))

print(df.head())
```

```
   CREM ATP5G1 RAB33B
0  no-c   no-c   no-c
1  no-c   no-c    inh
2  no-c   no-c    inh
```

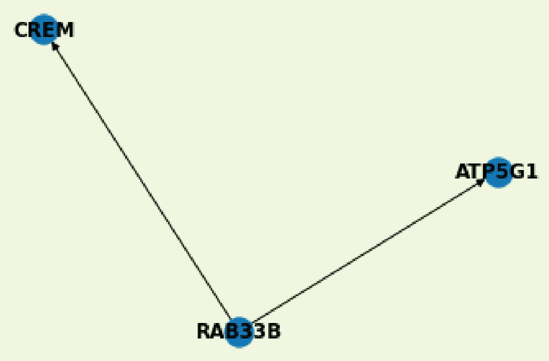
(continues on next page)

(continued from previous page)

```
3  no-c   no-c   no-c
4  no-c   no-c   no-c
```

Once the data is discretized, the network can be learned the usual way:

```
[4]: bn = BayesianNetwork().fit(df, data_type="discrete", algorithm="cl", estimation=
     ↪"bayesian")

     nx.draw(bn.graph, with_labels=True, font_weight='bold')
     plt.show()

     for gene, cpd in bn.get_cpds(genes).items():
         print(gene)
         print(cpd)
```



```
CREM
+------------+-------------+-------------+--------------+
| RAB33B     | RAB33B(act) | RAB33B(inh) | RAB33B(no-c) |
+------------+-------------+-------------+--------------+
| CREM(no-c) | 1.0         | 1.0         | 1.0          |
+------------+-------------+-------------+--------------+
ATP5G1
+--------------+----------------+--------------------+--------------------+
| RAB33B       | RAB33B(act)    | RAB33B(inh)        | RAB33B(no-c)       |
+--------------+----------------+--------------------+--------------------+
| ATP5G1(inh)  | 0.0961538461538 | 0.26562500000000006 | 0.08870967741935486 |
+--------------+----------------+--------------------+--------------------+
| ATP5G1(no-c) | 0.9038461538461 | 0.734375           | 0.9112903225806452  |
+--------------+----------------+--------------------+--------------------+
RAB33B
+--------------+----------+
| RAB33B(act)  | 0.216667 |
+--------------+----------+
| RAB33B(inh)  | 0.266667 |
+--------------+----------+
| RAB33B(no-c) | 0.516667 |
+--------------+----------+
```

Chapter 5

`NeurogenPy` as a `siibra-explorer` plugin

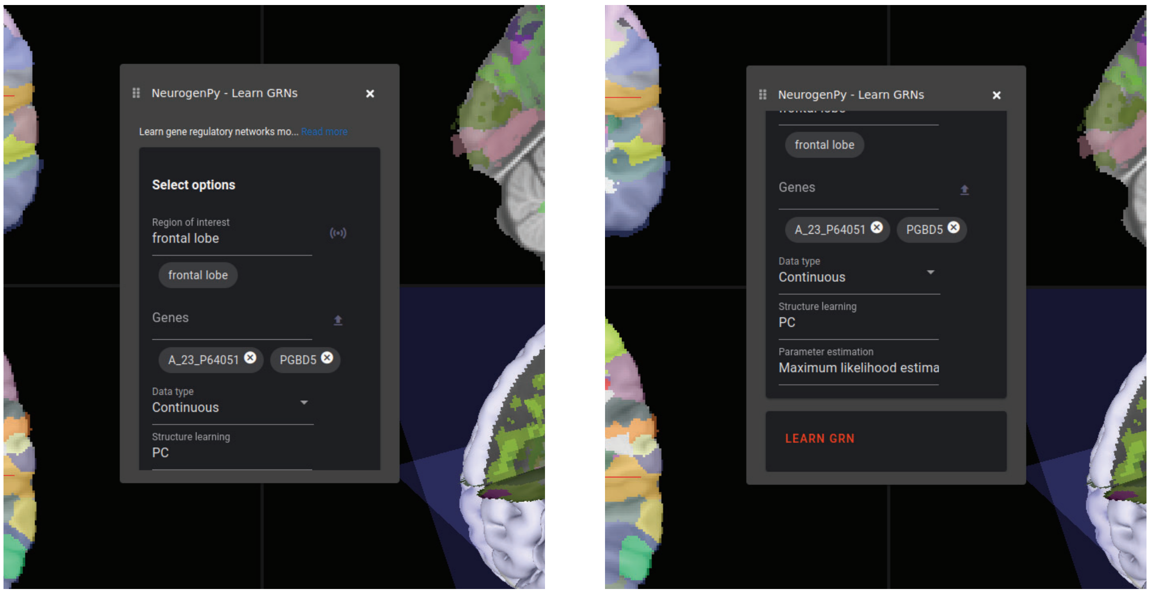
The `NeurogenPy` plugin development started once the package got to a stable version. We followed the same approach as `siibra-jugex`. Some *Svelte* files were written for the plugin user interface components. They were then compiled into JavaScript. The backend was implemented in Python as different *Celery* tasks and *Redis* was the message broker. This is where the package has been used. When both parts were ready, a *Docker* container was built using `docker-compose`. This container included the manifest file and the iframe `siibra-explorer` needs to load it.

5.1 Plugin design

The design of the plugin view was based on both `siibra-jugex` and *BayeSuites*. They are different tools, so we sometimes had to decide which one to follow. In general, we tried not to differ much from the `siibra-jugex` implementation so we focused more on it. The style of the view is also based on `siibra-jugex`. We used the same dark theme, *Material icons*, and *Svelte Material* components they used. This is because the plugin had to keep the appearance of all the other components in `siibra-explorer`. These requirements and all the differences between our package and the original *BayeSuites* Python code made reusing the *BayeSuites* interface extremely difficult. Even though we developed a new interface, its development was based on the functions and characteristics of the previous one.

5.1.1 Initial view

The initial view of the plugin is very similar to the latest `siibra-jugex` view. `siibra-explorer` sets a width constraint for any plugin, so this initial view is only focused on introducing the set of options used to learn the network. Figure 5.1 shows this initial view. Its use is quite intuitive. First of all, the user has to select the region of interest in the brain. There are two ways to do it: searching it by its name or selecting the sensor button and clicking on a particular region in the 3D visualization. After that, the user can select the set of genes used to learn the network. They can be selected by searching them one by one or by clicking on the upload file button and introducing a JSON file with the format presented in Listing 5.1. Then, the data type (continuous or discrete), the structure learning algorithm and the parameter learning method can be selected using their corresponding boxes. The available algorithms for each case are the same as those written in Table 3.1 and the discretization process followed the same scheme as the one discussed in Section 2.2.1.1. Finally, the "Learn GRN" button triggers the expression data retrieval and the GRN learning process. Once it is completed, the GRN view is opened.



Select options

Region of interest: frontal lobe

frontal lobe

Genes: A_23_P64051, PGBD5

Data type: Continuous

Structure learning: PC

Parameter estimation: Maximum likelihood estima

Figure 5.1. Plugin first view in `siibra-explorer`.

```
{
    "genes": [
        "Gene1",
        "Gene2",
        "Gene3",
    ]
}
```

Listing 5.1. Genes JSON input format example.

5.1.2 GRN view

The GRN view focuses on displaying the learned gene regulatory network and providing useful ways to manipulate it, perform inference and download the results. It is divided into three different views. The main view is focused on the visualization of the network, the second one offers general manipulations of the network, and the third shows information about particular genes and allows performing inference.

5.1.2.1 Graph display

The graph display view presents the graph structure obtained after the learning process. It has been developed using *Sigma.js*, the same tool used in *NeuroSuites*. `NeurogenPy` allows us to easily export a network structure in GEXF format as we explained in Section 4.1.4 and *Sigma.js* can easily read this format via *graphology*.

The view includes some simple options to manipulate the graph, such as dragging nodes, selecting different layouts, or hiding the labels. It also offers some visualization tools, like zooming in and out, going full screen, or selecting a node, highlighting it, and showing only its neighbors. Node selection is available in two different ways: clicking on the node or using the autocomplete search engine in the upper left corner. Most of these functions are executed in the frontend by *Sigma.js*, except for Dot, Sugiyama, Fruchterman Reingold, and Grid layouts that are executed with NeurogenPy in the backend. ForceAtlas2 and circular layouts are also included in NeurogenPy, but we kept the *Sigma.js* implementations because of performance reasons. Figure 5.2 shows an example of this view.

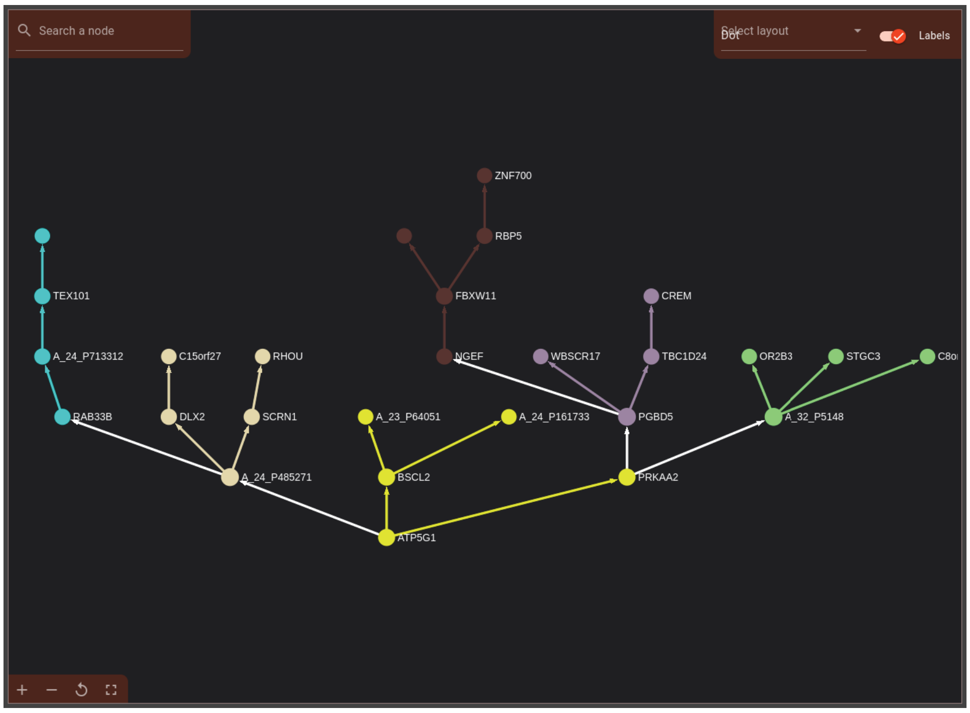
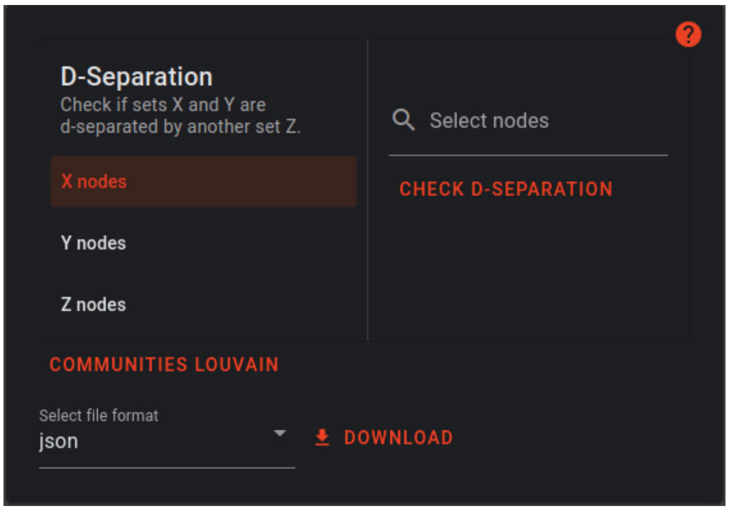


Figure 5.2. Example of graph display.

5.1.2.2 General network manipulation

The general network manipulation view (Figure 5.3) is focused on some actions that can be performed over the whole graph. It mixes different features, but they all have in common their general scope.

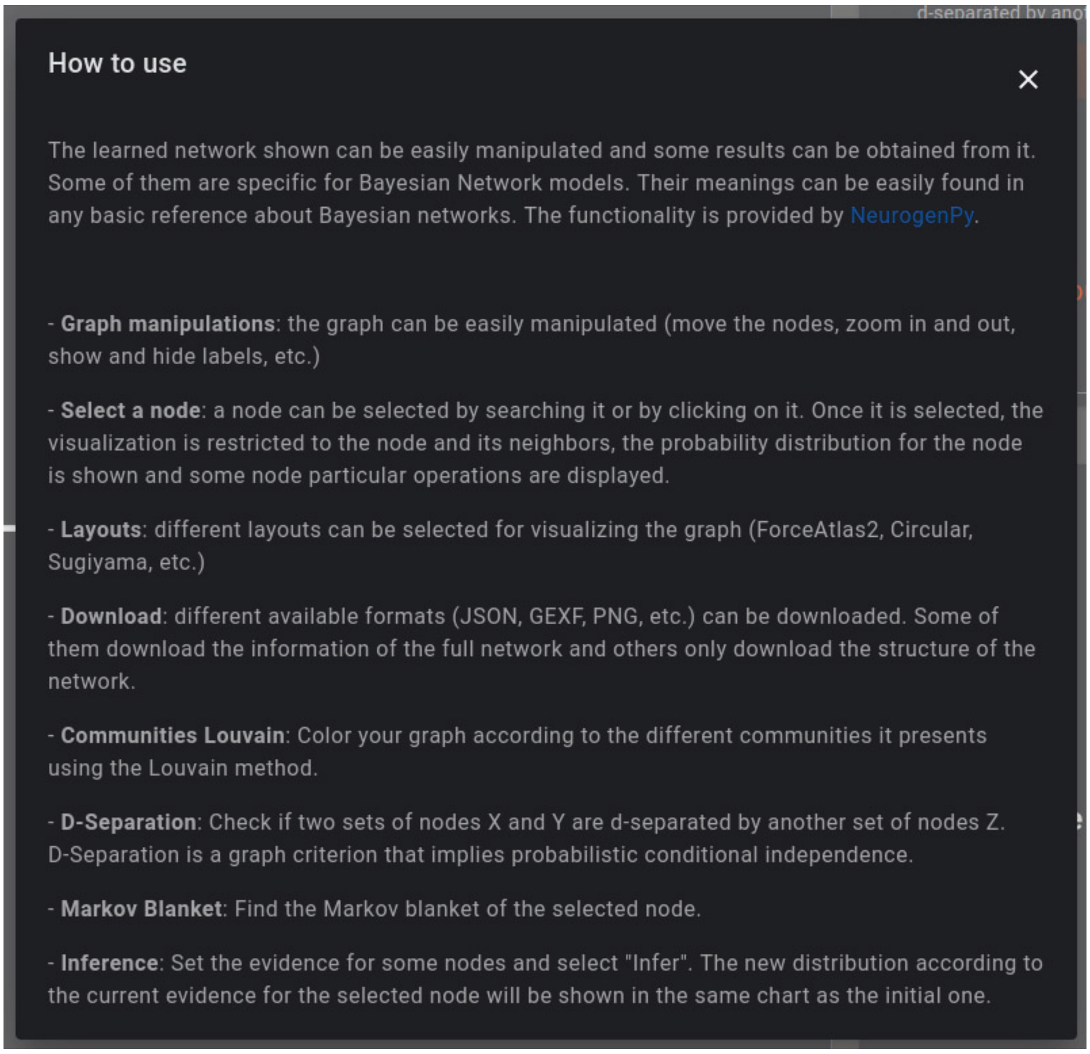

D-Separation

Check if sets X and Y are d-separated by another set Z.

Select file format: json

Figure 5.3. General network manipulation view.

The first one has to do with checking if two sets of nodes $\mathcal{X}$ and $\mathcal{Y}$ are d-separated by another set of nodes $\mathcal{Z}$. It is an important property because, as we explained in Section 2.1.1, d-separation implies conditional independence. The second one allows the execution of the Louvain method for community detection and colors the graph according to the found communities. Nodes of the same community receive the same color, edges between nodes of the same community receive the color of that community, and edges between nodes of different communities remain white, which is the default color. Figure 5.2 showed an example of how it works. As in the case of the layout, the Louvain algorithm is run on *Sigma.js* although `NeurogenPy` also provides it. The third function of this view lets the user download the resulting network in the desired format. Available formats are JSON, GEXF, PNG, BIF, and CSV. Finally, a help button is included in this view. It opens the dialog in Figure 5.4 where a simple user guide is shown.


How to use

The learned network shown can be easily manipulated and some results can be obtained from it. Some of them are specific for Bayesian Network models. Their meanings can be easily found in any basic reference about Bayesian networks. The functionality is provided by NeurogenPy.

- **Graph manipulations**: the graph can be easily manipulated (move the nodes, zoom in and out, show and hide labels, etc.)

- **Select a node**: a node can be selected by searching it or by clicking on it. Once it is selected, the visualization is restricted to the node and its neighbors, the probability distribution for the node is shown and some node particular operations are displayed.

- **Layouts**: different layouts can be selected for visualizing the graph (ForceAtlas2, Circular, Sugiyama, etc.)

- **Download**: different available formats (JSON, GEXF, PNG, etc.) can be downloaded. Some of them download the information of the full network and others only download the structure of the network.

- **Communities Louvain**: Color your graph according to the different communities it presents using the Louvain method.

- **D-Separation**: Check if two sets of nodes X and Y are d-separated by another set of nodes Z. D-Separation is a graph criterion that implies probabilistic conditional independence.

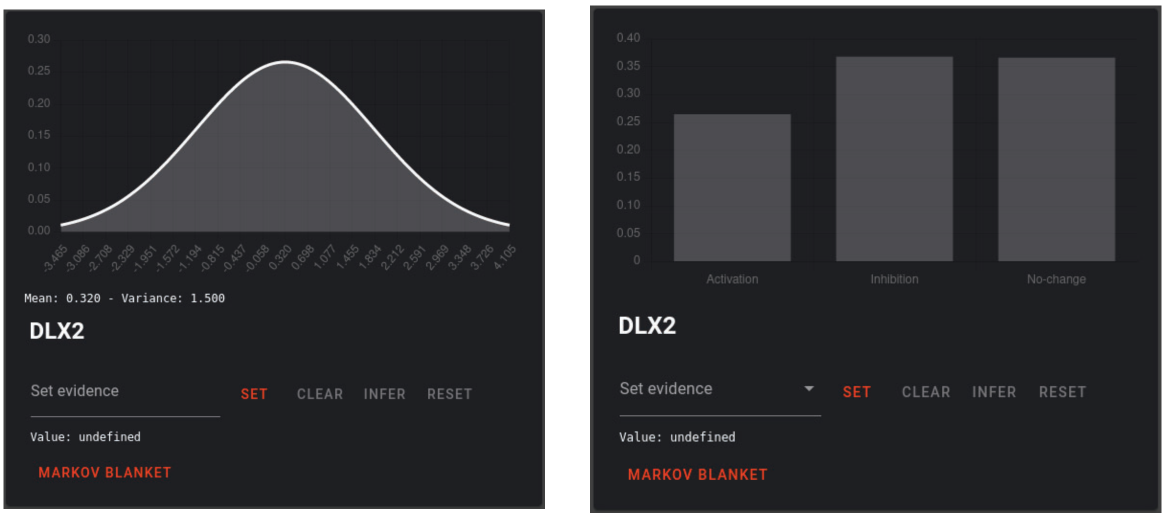
- **Markov Blanket**: Find the Markov blanket of the selected node.

- **Inference**: Set the evidence for some nodes and select "Infer". The new distribution according to the current evidence for the selected node will be shown in the same chart as the initial one.

Figure 5.4. Help dialog.

5.1.2.3 Node information

Once you select a node in one of the possible ways explained in Section 5.1.2.1, the view for this node appears. It provides some information about the node and some possible actions the user can do over it. First of all, the marginal distribution of the selected node, $P(X_i)$ or $f(X_i)$, is displayed. Figure 5.5 shows the different views depending on the type of the data. *Chart.js* is the JavaScript library used to show them. In addition to the distribution chart, the known evidence about that node can be easily set by selecting its value. The users can repeat this process for all the nodes they know evidence about. After setting all these values, pressing the "Infer" button will trigger the inference process. Once it is finished, the distribution chart will include the updated distribution, $P(X_i|\mathbf{E} = \mathbf{e})$ or $f(X_i|\mathbf{E} = \mathbf{e})$, too. Examples of the updated chart are shown in Figure 5.6. They are the same inference examples as in Section 4.3.5.1 and Section 4.3.5.2. Finally, the user can also retrieve the Markov blanket of the node. After doing so, the graph display will hide all nodes except those forming the Markov blanket.


```
Mean: 0.320 - Variance: 1.500
```
DLX2

Set evidence

Value: undefined

DLX2

Set evidence

Value: undefined

Figure 5.5. Node information view in Gaussian (left) and discrete cases (right).

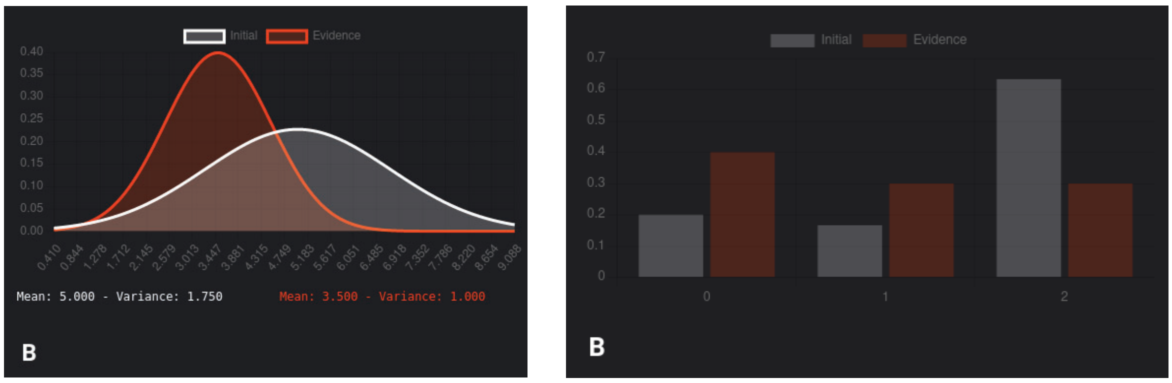

```
Mean: 5.000 - Variance: 1.750        Mean: 3.500 - Variance: 1.000
```

Figure 5.6. Chart views after introducing known evidence.

5.2 GRN example

Just to test the plugin, we decided to learn a bigger network than those shown in previous examples. The first 200 genes of the full list of genes included in siibra-python were taken. This set is presented in Listing 5.2. There was no available data for one of these genes, so the final size was 199. We learned a GRN with the expression data available for the frontal lobe with the Chow-Liu algorithm. The resulting network with ForceAtlas2 layout and colored by communities was exported in PNG format. See Figure 5.7.

```
{"genes": ["C15orf27", "PGBD5", "A_23_P64051", "ATP5G1", "RAB33B", "A_24_P485271", "CREM
    ", "DLX2", "OR2B3", "A_24_P713312", "A_32_P5148", "BSCL2", "STGC3", "SCRN1", "PRKAA2
    ", "RPL27", "TEX101", "NGEF", "RBP5", "RHOU", "FBXW11", "WBSCR17", "A_23_P96262", "
    TBC1D24", "ZNF700", "A_24_P161733", "C8orf22", "A_24_P118946", "KANSL1L", "ACOT6", "
    A_32_P12494", "RDH12", "MEF2C", "A_24_P942374", "A_24_P136911", "TOLLIP", "
    CUST_2664_PI416261804", "SMIM19", "OR11H4", "ANKRD36C", "MED12L", "CYP3A43", "ARL5A"
    , "A_32_P219635", "A_24_P706236", "TNFRSF10C", "ZNF484", "AADAT", "NADK2", "
    A_24_P221724", "CLEC6A", "ASPG", "IKBKAP", "RALB", "DGCR6", "A_24_P738859", "GOLPH3"
    , "A_23_P21804", "SNHG8", "NAA20", "KIAA0922", "A_24_P487877", "A_23_P26367", "KCNC3
    ", "A_24_P340886", "KIF18A", "SLC12A8", "OPTC", "GLUL", "NDUFB5", "ARMCX6", "DPYSL5"
    , "LIMS3-LOC440895", "A_24_P612020", "A_24_P213073", "MAPRE1", "A_24_P934971", "
    A_24_P392622", "LARP4", "GNMT", "MLH1", "CNTFR", "EEF2K", "ATP2A3", "ZNF730", "
    TP53I13", "A_24_P384379", "LOC728327", "SUSD6", "A_23_P93109", "C7orf13", "
    A_24_P127063", "A_24_P178643", "NINL", "PAMR1", "A_24_P232763", "GGT7", "ACOT12", "
    TEAD2", "AP4S1", "HNRNPCL1", "BTBD16", "LNPEP", "CDY1", "A_24_P485105", "SPDYE2B", "
    LOC728804", "GEMIN2", "PSG2", "OR7E91P", "SKAP2", "A_24_P654368", "DENND2A", "GPS1",
     "A_24_P222054", "ITGB8", "FAM21C", "A_24_P299137", "ERG", "RBM15", "SPATA7", "MXRA8
    ", "A_24_P592487", "ERICH5", "A_32_P193792", "A_24_P489399", "C6orf62", "SLC6A7", "
    HYMAI", "HSPA1B", "ACACA", "ZSCAN1", "MSANTD4", "TGIF2", "CYP3A4", "CALM3", "DSCAML1
    ", "A_23_P32821", "MGARP", "LINC01420", "A_23_P210451", "IGFBP5", "PPA2", "
    A_24_P791814", "PRSS50", "MMP11", "PZP", "A_32_P463538", "A_24_P936419", "
    A_32_P48054", "A_24_P791862", "PRKCZ", "COX15", "PNMT", "COPA", "C5orf58", "SEPT4",
    "ADGRG6", "A_24_P75856", "NOSIP", "A_24_P911051", "A_32_P51005", "A_24_P945069", "
    A_24_P608931", "CUST_2438_PI416261804", "A_24_P938281", "TM7SF3", "FLJ20021", "
    C19orf67", "KLHL34", "A_24_P938284", "L3MBTL1", "TRIP13", "PIK3CA", "MOSPD3", "
    A_32_P114372", "TXLNG", "A_32_P97968", "A_32_P170564", "A_24_P878561", "ARHGAP28", "
    A_32_P140153", "UBXN4", "CCDC64B", "LINC00165", "SNRNP40", "NEIL3", "A_32_P40327", "
    LMAN1L", "WTIP", "A_24_P934744", "FCHSD2", "LOC728836", "A_32_P208978", "GAGE10", "
    MAPK10", "ADCY10P1", "TRAF3IP3", "LOC100132014", "ARHGAP1"]}
```

Listing 5.2. Used genes.

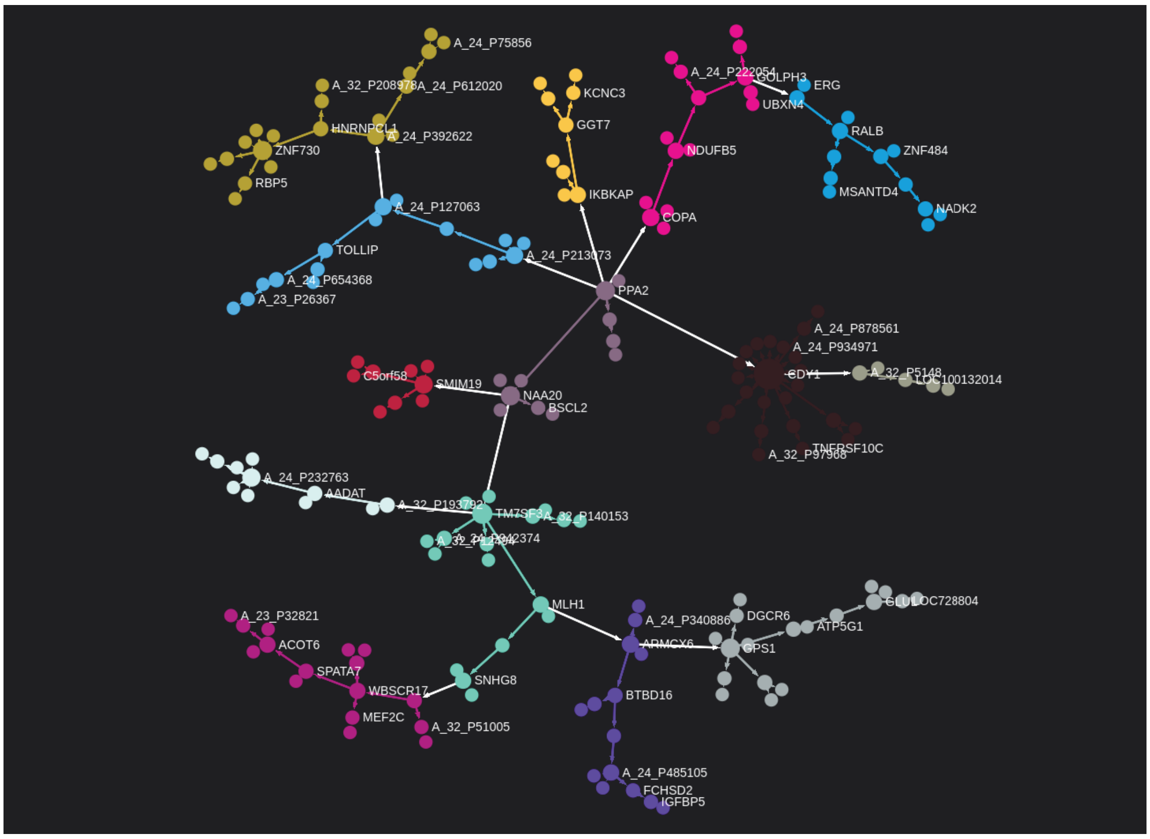


Figure 5.7. Learned GRN.

Chapter 6

Conclusions and future work

6.1 Conclusions

In this Master Thesis, based on *BayeSuites*, we have developed `NeurogenPy`, a Python package for learning and handling Bayesian networks focused on the problem of learning gene regulatory networks modeled as Bayesian networks. This package has been developed according to the current standards and is properly documented to facilitate reuse and extension by others. We have been able to successfully adapt and extend the functionality offered by *BayeSuites*. The software uses some of the current state-of-the-art Bayesian network and graph management packages. The resulting package has been used to develop a plugin for `siibra-explorer`, the EBRAINS Interactive Atlas Viewer, and to learn gene regulatory networks from gene expression data provided by the Allen Brain Institute through `siibra-python`. Finally, we were able to build some easy examples of the use of the package and plugin.

6.2 Future work

Although the development of the library and the plugin has been successful, we believe there are many different ways they both could be improved and extended. Along this section we will try to detail the most important ones.

`NeurogenPy` package provides many different structure learning algorithms. However, some of them are run in R and embedded in Python via `rpy2`. That is not necessarily a bad practice, but it may not show the best performance in computational time terms if the network is large. Additionally, it extends the dependencies needed to use the package, because R, `bnlearn`, and `sparsebn` have to be installed on their machines. Finding alternative solutions to this situation could be useful.

One of the most interesting features of the package is the inclusion of an implementation of the FGES-Merge algorithm. This algorithm uses hypothesis tests that are currently implemented only in the Gaussian case. However, its adaptation to the discrete case should not be very difficult and it would expand the package capabilities.

Apart from the previous two improvements, we believe there are many other possible advances for `NeurogenPy`. If we take a look to the supported parametric estimation methods, we see Bayesian estimation is not supported in the Gaussian case. Its implementation would enhance this software. Moreover, *BayeSuites* included the skeleton code for probabilistic clustering. We did not include it in `NeurogenPy` because it was not actually implemented or directly related to our topic, but it would be nice to develop it. Finally, adding other learning algorithms or supporting all type of data for those methods only available in the discrete or continuous case would be interesting too.

On the other hand, if we focus on learning gene regulatory networks, there are different paths that can be followed to make this package a more powerful tool. Right now, discretization of gene expression data should be done out of the package. An interesting improvement would be including some of the most typical discretization techniques for treating gene expression data [19]. Furthermore, adding other models for learning GRNs rather than Bayesian networks could be helpful too.

Focusing on the plugin itself, other than including the `NeurogenPy` improvements on it, some actions could be done. First of all, the download of the data could be openMINDS conformant. Additionally, the Apache Parquet case could be added too. On the visualization part, some other tools would be helpful. Selecting groups of nodes, adding other methods for detecting communities, or some other typical graph manipulation options, such as modifying the sizes of the nodes, represent some ideas in that way. However, if that is the case, it should be thoroughly designed to obtain a user-friendly interface.

Finally, we have to admit that gene expression data extraction with `siibra` is currently slow. If the set of genes for which the data has to be downloaded is large, it takes a very long time and it supposes an issue for the plugin performance. Nevertheless, at the time these lines are being written, `siibra-python` development team has been informed of that issue and they are working on providing a more suitable way of downloading expression data. By the time it is done, we will modify our plugin.

Bibliography

[1] M. Michiels, P. Larrañaga, and C. Bielza, "BayeSuites: An open web framework for massive Bayesian networks focused on neuroscience," *Neurocomputing*, vol. 428, pp. 166–181, 2021.

[2] J. Pearl, *Probabilistic Reasoning in Intelligent Systems*. Morgan Kaufmann, 1988.

[3] D. Koller and N. Friedman, *Probabilistic graphical models: principles and techniques*. The MIT press, 2009.

[4] R. D. Shachter and C. R. Kenley, "Gaussian influence diagrams," *Management Science*, vol. 35, no. 5, pp. 527–550, 1989.

[5] D. Geiger and D. Heckerman, "Learning Gaussian networks," in *Proceedings of the 10th International Conference on Uncertainty in Artificial Intelligence*, Morgan Kaufmann, 1994, pp. 235–243.

[6] P. Spirtes, C. N. Glymour, R. Scheines, and D. Heckerman, *Causation, Prediction, and Search*. The MIT Press, 2000.

[7] D. Margaritis, "Learning Bayesian network model structure from data," Carnegie-Mellon University, Tech. Rep., 2003.

[8] I. Tsamardinos, C. F. Aliferis, A. R. Statnikov, and E. Statnikov, "Algorithms for large scale Markov blanket discovery," in *FLAIRS Conference*, vol. 2, 2003, pp. 376–380.

[9] D. M. Chickering, "Learning Bayesian networks is NP-complete," in *Learning from Data: Artificial Intelligence and Statistics V*, Springer, 1996, pp. 121–130.

[10] D. M. Chickering, D. Heckerman, and C. Meek, "Large-sample learning of Bayesian networks is NP-hard," *Journal of Machine Learning Research*, vol. 5, pp. 1287–1330, 2004.

[11] D. M. Chickering, "Optimal structure identification with greedy search," *Journal of Machine Learning Research*, vol. 3, pp. 507–554, 2002.

[12] C. Bielza and P. Larrañaga, *Data-Driven Computational Neuroscience: Machine Learning and Statistical Models*. Cambridge University Press, 2020, pp. 521–522.

[13] G. F. Cooper, "The computational complexity of probabilistic inference using Bayesian belief networks," *Artificial Intelligence*, vol. 42, no. 2-3, pp. 393–405, 1990.

[14] N. L. Zhang and D. Poole, "A simple approach to Bayesian network computations," in *Proc. of the Tenth Canadian Conference on Artificial Intelligence*, 1994, pp. 171–178.

[15] J. Kim and J. Pearl, "A computational model for causal and diagnostic reasoning in inference systems," in *International Joint Conference on Artificial Intelligence*, 1983, pp. 190–193.

[16] S. L. Lauritzen and D. J. Spiegelhalter, "Local computations with probabilities on graphical structures and their application to expert systems," *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 50, no. 2, pp. 157–194, 1988.

[17] G. Karlebach and R. Shamir, "Modelling and analysis of gene regulatory networks," *Nature Reviews Molecular Cell Biology*, vol. 9, no. 10, pp. 770–780, 2008.

[18] D. Marbach, J. C. Costello, R. Küffner, *et al.*, "Wisdom of crowds for robust gene network inference," *Nature Methods*, vol. 9, no. 8, pp. 796–804, 2012.

[19] C. A. Gallo, R. L. Cecchini, J. A. Carballido, S. Micheletto, and I. Ponzoni, "Discretization of gene expression data revised," *Briefings in Bioinformatics*, vol. 17, no. 5, pp. 758–770, 2016.

[20] N. Friedman, M. Linial, I. Nachman, and D. Pe'er, "Using Bayesian networks to analyze expression data," *Journal of Computational Biology*, vol. 7, no. 3-4, pp. 601–620, 2000.

[21] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using NetworkX," in *Proceedings of the 7th Python in Science Conference*, Pasadena, CA USA, 2008, pp. 11–15.

[22] A. Ankan and A. Panda, "Pgmpy: Probabilistic graphical models using Python," in *Proceedings of the 14th Python in Science Conference*, 2015, pp. 6–11.

[23] C. R. Harris, K. J. Millman, S. J. van der Walt, *et al.*, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, 2020.

[24] F. Pedregosa, G. Varoquaux, A. Gramfort, *et al.*, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[25] P. Virtanen, R. Gommers, T. E. Oliphant, *et al.*, "SciPy 1.0: Fundamental algorithms for scientific computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.

[26] M. Scutari, "Learning Bayesian networks with the bnlearn R package," *Journal of Statistical Software*, vol. 35, no. 3, pp. 1–22, 2010.

[27] B. Aragam, J. Gu, and Q. Zhou, "Learning large-scale Bayesian networks with the sparsebn package," *Journal of Statistical Software*, vol. 91, no. 11, pp. 1–38, 2019.

[28] L. Gautier, *Rpy2 - R in Python*, 2013.

[29] V. A. Huynh-Thu, A. Irrthum, L. Wehenkel, and P. Geurts, "Inferring regulatory networks from expression data using tree-based methods," *PloS One*, vol. 5, no. 9, e12776, 2010.

[30] N. Bernaola, M. Michiels, P. Larrañaga, and C. Bielza, "Learning massive interpretable gene regulatory networks of the human brain by merging Bayesian networks," *bioRxiv*, 2020.

[31] L. Dalcin, *Mpi for Python*, 2021.

[32] S. Yaramakala and D. Margaritis, "Speculative Markov blanket discovery for optimal feature selection," in *Fifth IEEE International Conference on Data Mining*, 2005, pp. 809–812.

[33] G. Csardi, T. Nepusz, *et al.*, "The igraph software package for complex network research," *InterJournal, complex systems*, vol. 1695, no. 5, pp. 1–9, 2006.

[34] B. Chippada, *Forceatlas2 for Python*, 2017.

[35] E. S. Lein, M. J. Hawrylycz, N. Ao, *et al.*, "Genome-wide atlas of gene expression in the adult mouse brain," *Nature*, vol. 445, no. 7124, pp. 168–176, 2007.

[36] M. J. Hawrylycz, E. S. Lein, A. L. Guillozet-Bongaarts, *et al.*, "An anatomically comprehensive atlas of the adult human brain transcriptome," *Nature*, vol. 489, no. 7416, pp. 391–399, 2012.

[37] S. Bludau, T. W. Mühleisen, S. B. Eickhoff, M. J. Hawrylycz, S. Cichon, and K. Amunts, "Integration of transcriptomic and cytoarchitectonic data implicates a role for MAOA and TAC1 in the limbic-cortical network," *Brain Structure and Function*, vol. 223, no. 5, pp. 2335–2342, 2018.

[38] J. Ramsey, M. Glymour, R. Sanchez-Romero, and C. Glymour, "A million variables and more: The fast greedy equivalence search algorithm for learning high-dimensional graphical causal models, with an application to functional magnetic resonance images," *International Journal of Data Science and Analytics*, vol. 3, no. 2, pp. 121–129, 2017.

[39] J. D. Hunter, "Matplotlib: A 2D graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[40] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 10008, no. 10, pp. 1–12, 2008.