Exploring Alternatives to WGCNA in Neuroscience

by Benjamin Koppe

Abstract Weighted Gene Co-expression Network Analysis (WGCNA) is a well-established cornerstone for understanding gene expression patterns and uncovering the systems-level functionality of genes. However, WGCNA has a number of limitations, making application often tedious and difficult. This drives the need for more efficient alternatives. This paper details an investigation into the efficacy of some of these alternatives, focusing particularly CEMiTool: a novel approach in gene co-expression analysis. Files for this project can be found in its GitHub Repository.

1 Introduction

1.1 WGCNA

The process of clustering genes into modules is an important tool to understand complex biological systems. Gene clustering allows researchers to unravel complex interactions between genes and understand their impact. In neuroscience, for example, it is commonly used to analyze brain development, disorders, and behavior. WGCNA is an R package developed to perform this complex process by constructing networks (graphs) of genes based on the similarity of their expression patterns. [10] WGCNA is by far the most popular and widely used package in its class.

However, WGCNA poses a multitude of problems. [7] WGCNA has a long and tedious workflow that is time-consuming to configure and generalize, often requiring large amounts of code. This requires users to spend time constantly surveying for problems and errors. WGCNA also forces users to manually select a number of different parameters and filter input genes by hand. This process impacts reproducibility, as parameter selection is highly influential in output, and places the burden on researchers to justify the selection of each parameter. It also makes the already difficult workflow much harder to automate. Finally, WGCNA is limited in options for functional analyses.

In summary, WGCNA is weighed down by its high barriers to entry, which can be problematic for bioinformatics researchers both with and without technical and computing knowledge. To alleviate these problems, this investigation was launched to evaluate the number of alternative packages that have been written since the introduction of WGCNA.

1.2 CEMiTool

The Bioconductor package CEMiTool [7] was selected as the primary alternative in this investigation. CEMiTool was created primarily to address problems with WGCNA. The problems listed in 1.1 are all referenced in CEMiTool's introductory paper and used as a motivation for the package's development.

> "[CEMiTool] allows users to easily identify and analyze co-expression modules in a fully automated manner [and] [...] provides users with a novel unsupervised gene filtering method, automated parameter selection for identifying modules, enrichment and module functional analyses, as well as integration with interactome data. [It] then reports everything in HTML web pages with high-quality plots and interactive tables." [7]

CEMiTool's many features and highly simplified and automated workflow make it a very compelling alternative for WGCNA. As such, all implementations and testing shown in this paper will use CEMiTool.

2 Methods

2.1 General Investigation

Before focusing specifically on CEMiTool, a few packages were assessed in order to create a general overview. Besides CEMiTool, other alternative candidates were BioNERO [1], petal [9], and minet [6]. Later, GWENA [5] was also briefly examined. Packages were evaluated primarily from their descriptions in their corresponding papers. Desired properties were ease of use, explicit pathways for comparison to WGCNA results, and explicit implementation details designed to address the primary WGCNA problems listed in 1.1.

Once CEMiTool was selected, implementation was focused around the development of an optimal simple workflow for dataset analysis. To do this, a 2012 microarray dataset [4] was used for all testing. A slightly reformatted copy of the dataset is available in the project repository.

2.2 R

Due to its strength in statistical analysis, large community, and close association to the Bioconductor project, among other factors, R is the near-universally used language in this area. All co-expression analysis packages examined in this report are written in R. As a result, R is a critical component of this report's methodology, as all implementations must use or interface with R.

An especially notable advantage of R is its high degree of flexibility. R has excellent capabilities for integration with other languages and tools, such as Python, C++, and Java. This is valuable to this study, as it allows for the testing of other languages for a further optimized workflow. Though R has many advantages, its age and design can lead to numerous problems: it is riddled with dangerous inconsistency where many parts disagree; it lacks clear convention and documentation in many cases; its package management completely lacks proper support for dependencies, version control, and environment management; and it suffers from very poor scalability through largely unaddressed problems like name collisions. Problems like these have been well documented and continue to persist over time. [2][3] R's ease of interface, however, allows us to experiment with solutions that avoid these problems.

2.3 Python

Python was used as a potential alternative to R code in the workflows investigated in this report. As a result, most code in this report is written in Python. Implementing between R and Python is simple and seamless, and Python includes a number of advantages over R, such as simple and reliable package management, a much higher degree of consistency, and a broader range of libraries such as `cupy` (for GPU-accelerated computation) and `PySpark` (for large-scale data processing) that benefit scalability significantly. Though statistical analysis is not as closely tied with the language's design philosophy as with R, this project's code was implemented in Python with the hope that it can benefit greatly from the language's generality and highly-enforced standards, leading to a better overall workflow.

2.4 Visual Studio Code (VSCode)

VSCode was used as the Integrated Development Environment (IDE) throughout this process. This means that all code was written and run in the VSCode environment. VSCode's extremely large community and lightweight implementation makes it the only IDE with comprehensive support of nearly any language or file type that can be encountered. This was especially useful for this project, as it allowed for R, Python, and Jupyter Notebook files to be simultaneously edited and run in a single place. VSCode's high modularity and large array of extensions also put it ahead of other IDEs. Some especially useful extensions are Remote-ssh, GitHub Copilot, Prettier, autoDocstring, and Todo Tree.

2.5 Jupyter Notebooks

Another benefit of Python is support for Jupyter Notebooks, which allow for inline display of markdown text, runnable code, and dynamic displayed data/analysis. Jupyter Notebooks have built-in caching, where CPU-bound tasks can be isolated to a single block of code, run a single time, and

kept cached for use in all other blocks without repeating the calculation. Jupyter Notebooks are also compatible with VSCode. In this project, these notebooks allow for easy and simplistic code demoing and display, and are hosted on Google Colab. All Jupyter Notebooks in this project are also kept in the main repository.

3 Results

3.1 Running CEMiTool

CEMiTool automates the entire module discovery process with a single function call – the `cemitool` function. With this function, a `cem` object is created, and all future processing is centered around this object. In R, this functionality looks like this:

```
df <- read.delim2("data/probe_data_copy.txt")
cem <- cemitool(df)

generate_report(cem)
modules(cem)
...
```

The only required argument is a gene expression file that contains genes as rows and samples as columns (`df` in this case). From there, genes are selected from an unsupervised filtering method based on the inverse gamma distribution, a soft-thresholding power β is chosen and used to determine a similarity criterion, and the genes are separated into modules using the Dynamic Tree Cut package. [7]

The `cemitool` function accepts a number of additional parameters: filtering and verbose output can be individually enabled or disabled; gene interactions can be provided in an additional file to return network graphs of interacting genes; a sample annotation file can be provided to perform gene set enrichment analysis (GSEA), allowing users to visualize induced or repressed modules in different phenotypes; and a file containing gene sets can be provided to allow for over-representation analysis (ORA) to determine more significant module functions. A number of other parameters are also provided for additional control, but all are optional to provide minimal user responsibility. For reference and ease of comparison, CEMiTool's original paper kept all parameters at their default value. In this analysis, CEMiTool was run with default parameters and no optional files.

3.2 Visualizing CEMiTool

From there, CEMiTool contains multiple built-in plotting and analysis functions. Most notable are its `diagnostic_report` and `generate_report` functions. Rather than generate simple plots for viewing, these functions generate interactive HTML files that display multiple useful diagnostic plots at once, module data, and full `cemitool` object parameterization. These files can be viewed in a browser (or in VSCode with the *live preview* extension), and because multiple runs don't modify file path, the paths for these HTML files can be left open in the user's browser and conveniently reloaded on each new run of a dataset. An example of these reports can be seen in Figure 1.

Plotting functions can always save their results to PDFs, and CEMiTool can automatically generate all plots and store them to a folder. CEMiTool can generate ten different types of plots, which include:

- `plot_sample_tree`: Displays similarities between samples in the expression data with a dendrogram. This can be used to quickly analyze connectivity.
- `plot_profile`: Compares module gene expression profiles along samples and highlights eigengenes. These plots can simplify module characterization.
- `plot_beta_r2`: Compares soft-thresholding values β and its corresponding R^2 value. This plot helps in interpreting the choices of the automatically selected β value.
- `plot_mean_k`: Graphs mean connectivity of genes in the network as a function of β.
- `plot_hist`: Plots a histogram of the distribution of gene expression. This can be used to assess data normality.
- `plot_mean_var`: Scatter plots the mean by the variance of gene expression. With RNA sequencing data this plot can be evaluated for a linear relationship. [8]

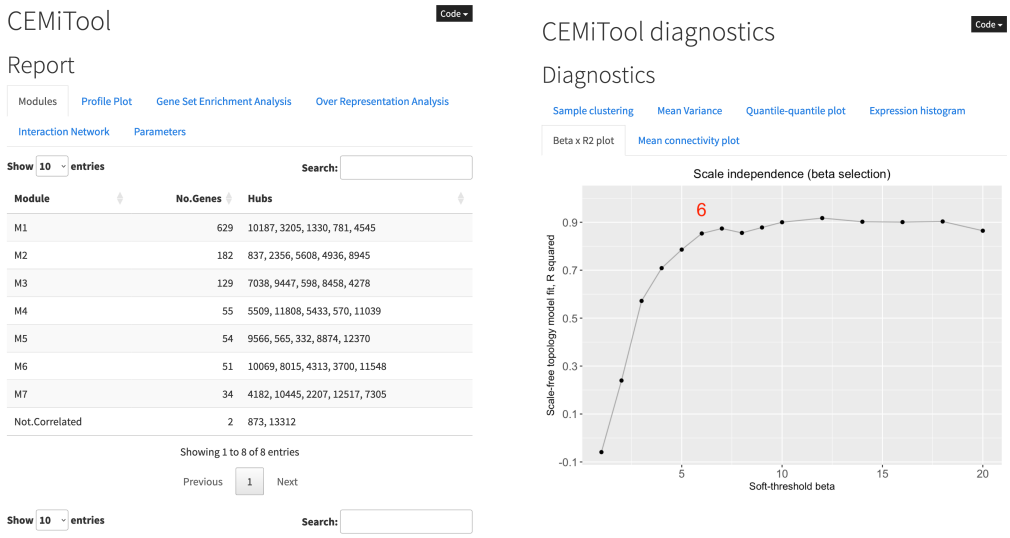

Figure 1: CEMiTool report and diagnostics HTML files.

All of these plots are already displayed in the report and diagnostics files. Some plots can only be generated if additional files are included in the initial `cemitool` function call.

One important goal of this investigation was to generate a dendrogram that displays similarities between genes, rather than samples. No function for this exists in CEMiTool and the package lacks documentation for proper creation of custom plots. As a result, there hasn't yet been successful at properly generating gene dendrograms from `cemitool` objects. This would take more time to develop and would significantly simplify output comparisons between WGCNA and CEMiTool outputs within the scope of this paper.

3.3 Necessary Preprocessing

Though CEMiTool is designed to automate most of the clustering process, it still adheres to the general concept of "garbage in, garbage out." The input dataset must have high-quality and relatively unbiased data, otherwise the output result could be very inaccurate. Ensuring data quality adds steps of necessary preprocessing, which often comes in the form of additional checks and filters – as is the case in this report.

With our example dataset, which contained gene data for bird samples, three filters were performed before the data was passed into the `cemitool` function:

1. *Sample Information:* Bird types were filtered such that only bird samples labeled with "areaX" were included in the dataset. This ensured that all samples were of the same type. Labels were stored in a separate file. With the dataset in use for this paper, this reduced the number of samples from 54 to 27.
2. *NA & Sum Filtering:* Empty cells (cells with NA/NaN values) are replaced with zeroes, all genes were summed for each sample, and samples with sums outside a certain tolerance were dropped from the dataset. This ensures that samples overwhelmed with empty cells or irregular data are removed. With the dataset in use for this paper. this reduced the number of samples from 27 to 26.
3. *Standard Deviation Filtering:* All genes with values greater than two standard deviations outside the other data in the sample causes the entire gene to be removed. This controls for abnormal and poor data. With the dataset in use for this paper, this reduced the number of genes from about 20,000 to about 17,000.

Time didn't permit extensive testing of this method, but a potentially more effective and more autonomous strategy for dataset cleanup and filtering might have been through various functions available in the GWENA co-expression analysis tool. [5] These functions, such as `filter_low_var`, are intended to perform this task. When first implemented in this project, the function failed to remove any data from the dataset, however.

3.4 R-Python Interface

The package `rpy2` was used to interface with R in Python. This package can convert seamlessly between R and Python data structures, import R packages, and fully access R objects and functions. In Python, importing CEMiTool takes only a few lines:

```
from rpy2.robjects.packages import importr

cemitool = importr('CEMiTool')
doParallel = importr('doParallel')

doParallel.registerDoParallel(cores = ...)
```

From there, data can be loaded with `pandas` and fed directly into the `ccemitool` object's methods – all of which share the same name as within the original package. This can be seen with CEMiTool's methods:

```
import pandas as pd

preprocessed_dataframe = pd.read_csv(...)
cem = cemitool.cemitool(preprocessed_dataframe, plot=True)

cemitool.generate_report(cem) # command mirrors format in R
cemitool.modules(cem)
...
```

As a result, all features and packages available in R are now put in reach of Python code with relatively minimal effort. Scripts in this project use this functionality to load and preprocess the input dataset with Python packages and operations before being run in CEMiTool, which is implemented in R. This largely neglects the flexibility brought by a Python implementation, but nevertheless displays the relatively seamless interaction between languages that `rpy2` enables.

3.5 Creating a Workflow

With all of the aforementioned pieces in place, a simple workflow could be developed in Python. This workflow would take a dataset, preprocess it, run it through CEMiTool, and generate plots, reports, and diagnostics. This workflow was implemented both as a Python script and as a Jupyter Notebook. The script can be configured from within and run from the command line, and the notebook is also configured from within and can be run both locally and in Google Colab. Both implementations are available in the project repository, in the `workflow/` folder, along with a link to Google Colab. Examples of both can be seen in Figure 2

3.6 Figures & Comparisons

All tests done in this investigation were done with microarray data from singing zebra finches obtained from a *Neuron* paper. [4] The dataset and sample info used is provided in the `workflow/data/` folder. This data was run with the provided workflow, with all four types of filtering enabled (including experimental GWENA). A sample run of this dataset is provided in the `workflow/example_output` folder. Some plots from this run are included in this report, seen in Figure 3, Figure 4, and Figure 5.

No plot types can yet be generated to allow for simple comparison with WGCNA data, and as a result the efficacy of the generated data cannot be compared to WGCNA. This misses the mark on a major goal of this project. To attempt to alleviate this issue, implementation work was begun on a gene dendrogram, as discussed in 3.2. Right now, this implementation doesn't work and produces results that do not make sense. This implementation can be seen in the `extras/dendrogram-testing.ipynb` file, and an example of its plot generated from the sample data can be seen in Figure 6.

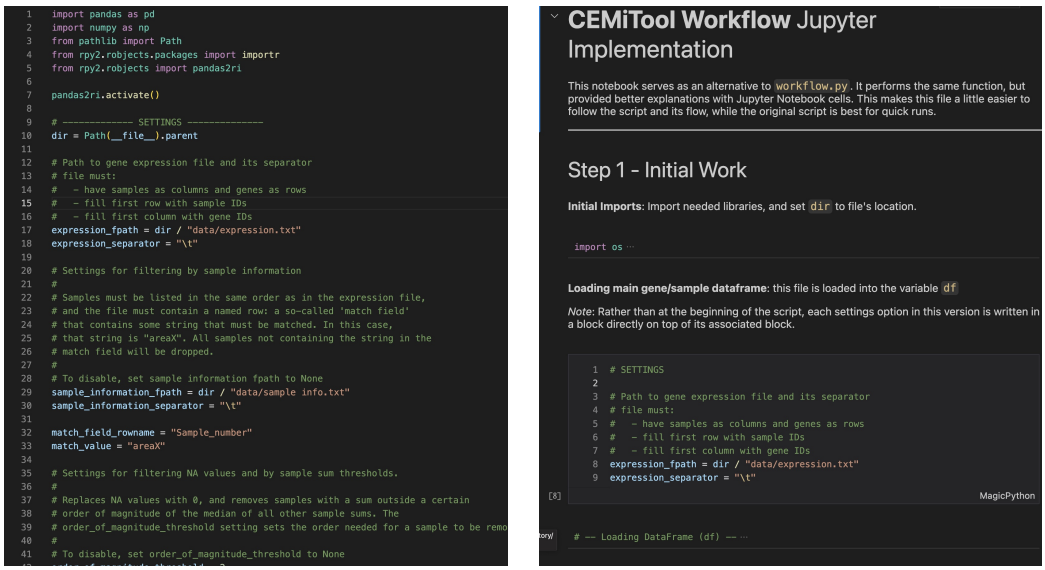

Figure 2: Python script and Jupyter Notebook implementations of the workflow.

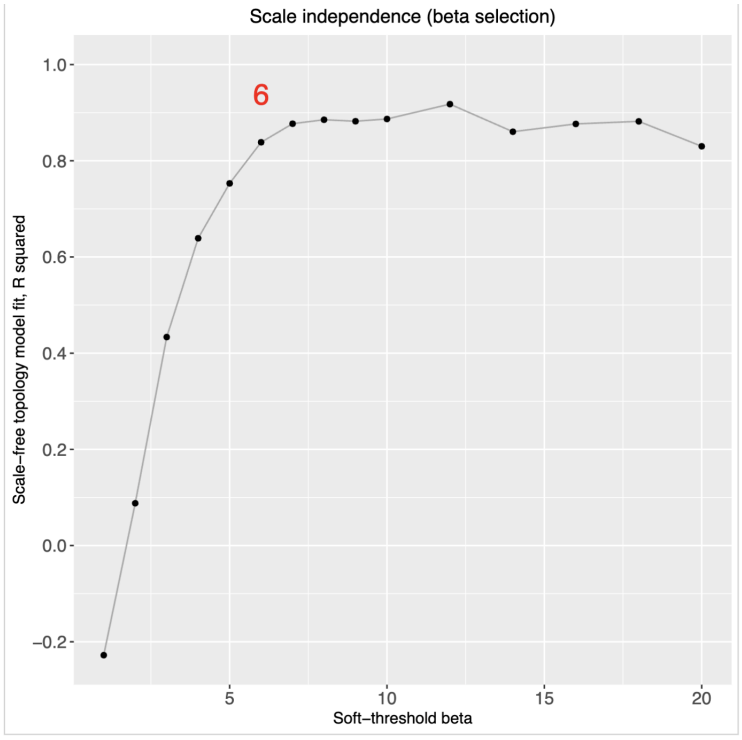

Figure 3: The `plot_beta_r2` output for the example dataset.

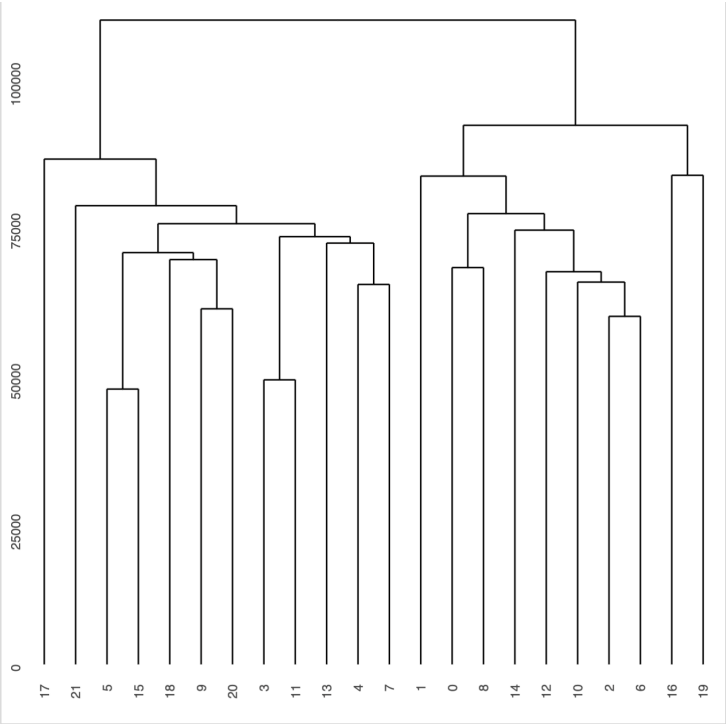

Figure 4: The `plot_sample_tree` output for the example dataset.

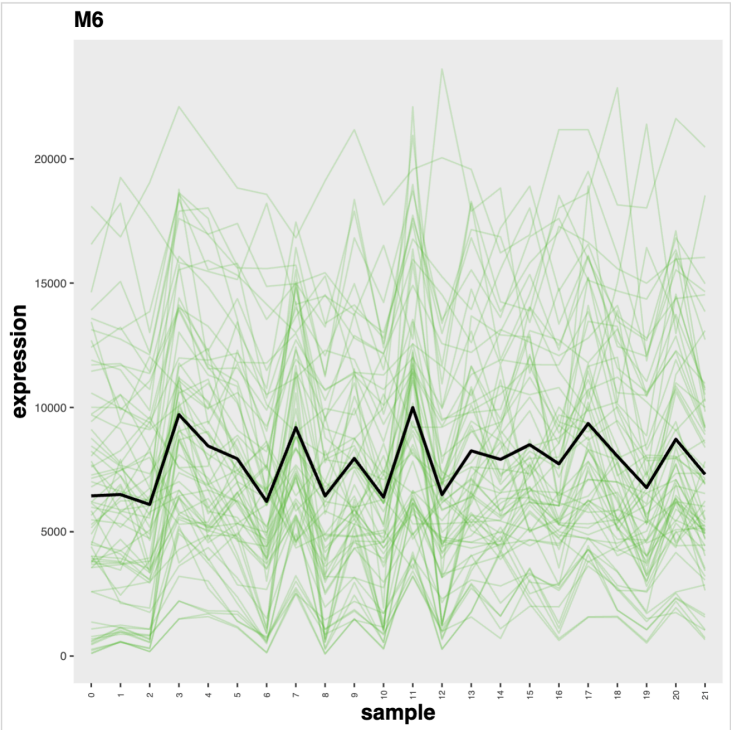

Figure 5: One of the 9 `plot_beta_r2` outputs for the example dataset. One plot is generated for each module made, plus one more for uncorrelated genes.

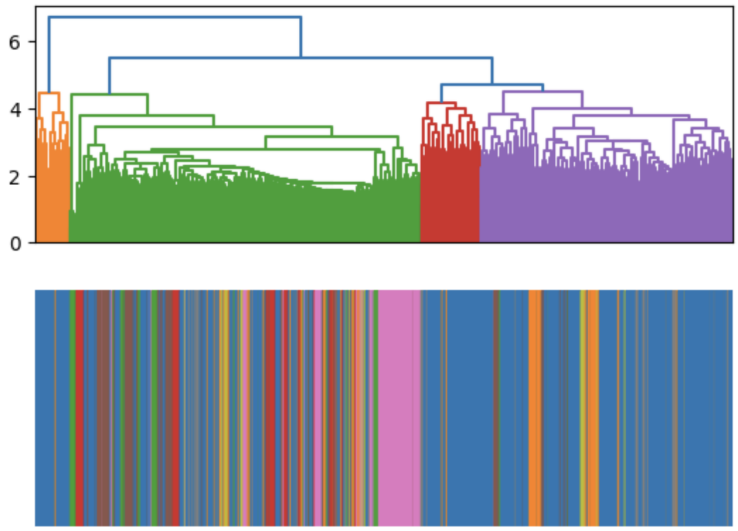

Figure 6: The output of the custom gene dendrogram implementation with the sample dataset. The dendrograph is shown on the top, and the module associated with each leaf of the dendrogram is color-coded on the bottom.

4 Discussion

This project made significant progress with CEMiTool, establishing a general workflow and beginning to produce results. However, with more time, there is significant room for advancement. The area that most needs work is in comparison with WGCNA results. This is completely lacking from this project, as nearly all time was spent interfacing with CEMiTool and working to obtain proper outputs. With more time, CEMiTool can be extended in ways that meet our needs for comparison, such as with gene dendrograms as seen in 3.2, 3.6, and in Figure 6. Other methods of comparison between CEMiTool and WGCNA were already seen in CEMiTool's original paper [7], and with more time these methods could be explored and implemented as well.

In addition to increased capacity for comparison, more time could be spent exploring other package options and solutions. For example, the newer package GWENA [5] was found near the end of the investigation and, as such, could only contribute to the preprocessing methodology in 3.3. Other packages could ease the need for custom implementations such as the gene dendrogram, as these needs may be already implemented in these packages.

Finally, with more time, the `rpy2` R-Python interface shown in 3.4 could be further taken advantage of. This could be done in a number of ways, such as those mentioned in 2.3. The benefits of implementing this workflow in Python will become most apparent as the project is scaled up and begins to require increased generality, more robust package support, and a wider array of external modules.

Nevertheless, the advantages of more modern packages like CEMiTool are clear: this workflow is very short and readible. With WGCNA, the same task would take much more code, parameterization, and overall effort. If it is made clear that CEMiTool's results (or some other package) are comparable in quality to WGCNA, this simplified workflow will substantially increase the ease of future clustering work in research both inside and outside neuroscience applications.

Bibliography

[1] F. Almeida-Silva and T. M. Venancio. Bionero: an all-in-one r/bioconductor package for comprehensive and easy biological network reconstruction. *bioRxiv*, 2021. [p2]

[2] P. Burns. *The R inferno*. Lulu Com, 2011. [p2]

[3] R. Goding. Reecegoding/frustration-one-year-with-r: An extremely long review of r., Dec 2021. [p2]

[4] A. T. Hilliard, J. E. Miller, E. R. Fraley, S. Horvath, and S. A. White. Molecular microcircuitry underlies functional specification in a basal ganglia circuit dedicated to vocal learning. *Neuron*, 73(3):537–552, Feb 2012. [p2, 5]

[5] G. G. Lemoine, M.-P. Scott-Boyer, B. Ambroise, O. Périn, and A. Droit. Gwena: gene co-expression networks analysis and extended modules characterization in a single bioconductor package. *BMC Bioinformatics*, 22(1):267, 2021. [p2, 4, 9]

[6] P. E. Meyer, F. Lafitte, and G. Bontempi. minet: A r/bioconductor package for inferring large transcriptional networks using mutual information. *BMC Bioinformatics*, 9(1):461, 2008. [p2]

[7] P. S. T. Russo, G. R. Ferreira, L. E. Cardozo, M. C. Bürger, R. Arias-Carrasco, S. R. Maruyama, T. D. C. Hirata, D. S. Lima, F. M. Passos, K. F. Fukutani, M. Lever, J. S. Silva, V. Maracaja-Coutinho, and H. I. Nakaya. Cemitool: a bioconductor package for performing comprehensive modular co-expression analyses. *BMC Bioinformatics*, 19(1):56, 2018. [p1, 3, 9]

[8] P. S. T. Russo, G. R. Ferreira, L. E. Cardozo, M. C. Burger, R. Arias-Carrasco, S. R. Maruyama, T. D. C. Hirata, D. S. Lima, F. M. Passos, K. F. Fukutani, M. Lever, J. S. Silva, V. Maracaja-Coutinho, and H. T. I. Nakaya. Cemitool: a bioconductor package for performing comprehensive modular co-expression analyses. *BMC Bioinformatics*, 19(56):1–13, 2018. [p3]

[9] G. Sgroi, G. Russo, and F. Pappalardo. PETAL: a Python tool for deep analysis of biological pathways. *Bioinformatics*, 36(22-23):5553–5555, 12 2020. [p2]

[10] B. Zhang and S. Horvath. *Statistical Applications in Genetics and Molecular Biology*, 4(1), 2005. [p1]

Benjamin Koppe
University of Arizona
Department of Computer Science
United States
koppe@arizona.edu

Dr. Charles Higgins
University of Arizona
Department of Neuroscience
Department of Electrical Engineering
United States
cmh@arizona.edu