A workflow for continuous and collaborative benchmarking

This manuscript (permalink) was automatically generated from komparo/manuscript-workflow@285bb1b on November 19, 2018.

Authors

- **Wouter Saelens** *
  0000-0002-7114-6248 · zouter · zouters
  Data Mining and Modelling for Biomedicine, VIB Center for Inflammation Research, Ghent, Belgium; Department of Applied Mathematics, Computer Science and Statistics, Ghent University, Ghent, Belgium · Funded by Fonds Wetenschappelijk Onderzoek

- **Robrecht Cannoodt** *
  · rcannood · rcannood
  Data Mining and Modelling for Biomedicine, VIB Center for Inflammation Research, Ghent, Belgium; Department of Applied Mathematics, Computer Science and Statistics, Ghent University, Ghent, Belgium · Funded by Fonds Wetenschappelijk Onderzoek

- **Lukas Weber**
  · lmwebr
  Institute of Molecular Life Sciences, University of Zurich, Zurich, Switzerland; SIB Swiss Institute of Bioinformatics, University of Zurich, Zurich, Switzerland

- **Charlotte Soneson**
  · CSoneson
  Institute of Molecular Life Sciences, University of Zurich, Zurich, Switzerland; SIB Swiss Institute of Bioinformatics, University of Zurich, Zurich, Switzerland; Friedrich Miescher Institute for Biomedical Research, Basel, Switzerland

- **Yvan Saeys** *

  Data Mining and Modelling for Biomedicine, VIB Center for Inflammation Research, Ghent, Belgium; Department of Applied Mathematics, Computer Science and Statistics, Ghent University, Ghent, Belgium

- **Mark D. Robinson** *
  · markrobinsonca

Institute of Molecular Life Sciences, University of Zurich, Zurich, Switzerland; SIB Swiss Institute of Bioinformatics, University of Zurich, Zurich, Switzerland

Abstract

Benchmarking is a critical step in the development of bioinformatics tools, but the way benchmarking is done at the moment has some limitations. Because each benchmark is developed in isolation, they tend to be hard to compare, extend and are rapidly outdated. Moreover, benchmarks are usually rapidily outdated as new methods are developed. To address these challenges, we combined modern software development tools to create a workflow for continuous and collaborative benchmarking. The structure of the benchmark is highly modular, so that anyone can contribute a set of datasets, metrics, methods or interprete the results, and get credit for their contributions. We also discuss some organisational issues, and propose ways on how to solve them. As a test case, we applied this worklow on an emerging type of analysis in the single-cell field: trajectory differential expression, available at https://github.com/komparo/tde. A skeleton version of the workflow, which can be used to create a similar benchmarking workflow for a different type of methods, can be found at https://github.com/komparo/skeleton.

Introduction

Evaluating the performance of a new method, and comparing it to the state-of-the-art, is a critical step in the development of bioinformatics methods. Benchmarks are essential to showcase the advantages and weaknesses of a method, and assure that new tools improve upon related methods. Despite this, well-designed and balanced benchmarking strategy can be difficult to create, especially when a ground truth on real data is not available.

The breadth of a benchmark is influenced by its purpose. In some studies, the goal is to review the methods available in the field, and highlight current challenges. Such independent benchmarks are usually very comprehensive, involving many datasets and different metrics ranging assessing the accuracy, scalability and robustness of a method. A special case of such a benchmark are competitions, where the focus lies on promoting the development of new methods within the field, while using existing methods as baseline. Other benchmarks are used as a companion to a study proposing a new method, demonstrating its improvements and usefulness.

While benchmarks are unmistakingly important, the way benchmarking is usually done has some limitations:

- Benchmarks are quickly outdated when new methods come along.
- Benchmarks are difficult to extend, as this is usually only added as an afterthought.
- While benchmarks often reach different conclusions, they are difficult to compare, because of (unclear) differences in datasets, method parameters, metric implementation and aggregation.
- Independent benchmarks and competitions tend to be authoritative, with only a small group of people deciding on how methods should be compared.

- Independent benchmarks are usually published quite late, only after a lot of methods are already available.
- Companion benchmarks represent in some way a lot of wasted effort, because datasets are often reanalysed, metrics reimplemented, and methods rewrapped.

To resolve these issues, we created a workflow for benchmarking which centers around the following three core concepts:

- **Modular**: It should be possible to extend the benchmark simply by adding a self-contained "module". Such a module could be: a dataset generator, a method, a set of metrics, or a report generator that interpretes the results and produces a report. Several tools exist already for making benchmarks modular: SummarizedBenchmark [1], Dynamic Statistical Comparisons and iCOBRA [2].
- **Collaborative**: Anyone with a computer and internet connection should be able to run and contribute to the benchmark. This can range from contributing a module, to changing the structure of the benchmark itself. Discussions on the benchmark or any of the reports should also be open. The collaborative aspect of benchmarking has usually focused on the level of methods, with countless competitions and challenges, such as those organised by DREAM or kaggle.
- **Continuous**: A benchmark should be continously updated when new modules are added. This has quite a long history in bioinformatics, particularly in structure prediction [3], but also in other fields [4].

To construct a workflow which fulfills combines these three concepts, we used several ideas and tools coming from modern software development, such as continuous integration, containerisation and workflow management.

In brief, our workflow is structured as follows. We define several different **types of modules** (Figure 1a): dataset generators can generate datasets and optionally use another dataset as input, methods use a dataset to generate some model, metrics will calculate some scores using the model and optionally also parts of the dataset, and finally a report generator which summarise the datasets, models and scores into a report. Each type of module can generate a set of files which are constrained to a particular set of **formats** (Figure 1b). Each format has an unambiguous description, a set of good and bad examples, and includes a validator which validates the output files generated by each module. While each format is defined beforehand, new formats can be added over time as the field progresses. A **module** (Figure 1c) is a set of scripts and packages, which are run inside a portable environment. This module is put under version control, shared on a code sharing platform, and tested automatically using continuous integration. When all tests of a module are succesful, these modules can be integrated into the actual **benchmarking workflow** (Figure 1d). Within this workflow, modules are connected through a particular design, which is executed using a workflow manager. The output of the benchmark are a set of reports and apps, which are made available through a publishing platforms. To add a new module, a pull request is

created to integrate the module within the benchmarking workflow, after which the contribution is reviewed openly. When accepted, the module is automatically integrated within the workflow, and the necessary parts of the workflow are re-executed. Finally, in regular time intervals (e.g. monthly), the full set of reports and apps are gathered and versioned.

As a test case, we developed a proof-of-concept benchmark for single-cell trajectory differential expression (TDE) methods. TDE methods try to find genes which are differentially expressed along a trajectory, the latter of which is an positioning of cells along a graph structure. Given that only a few of such methods have been developed yet [5,6,7], this is the ideal scenario for developing the idea of a continuous and collaborative benchmark, and try to find solutions to the inevitable challenges which will come up as the field develops.

We will further discuss each element of the workflow in detail, along with how we currently implemented it in practice. It is important to acknowledge here that this is only one possible implementation, and that other tools, some of which still have to be developed, could better fit the benchmarking use-case. What is the most important is not the way the benchmark is implemented, but the ideas behind its implementation.

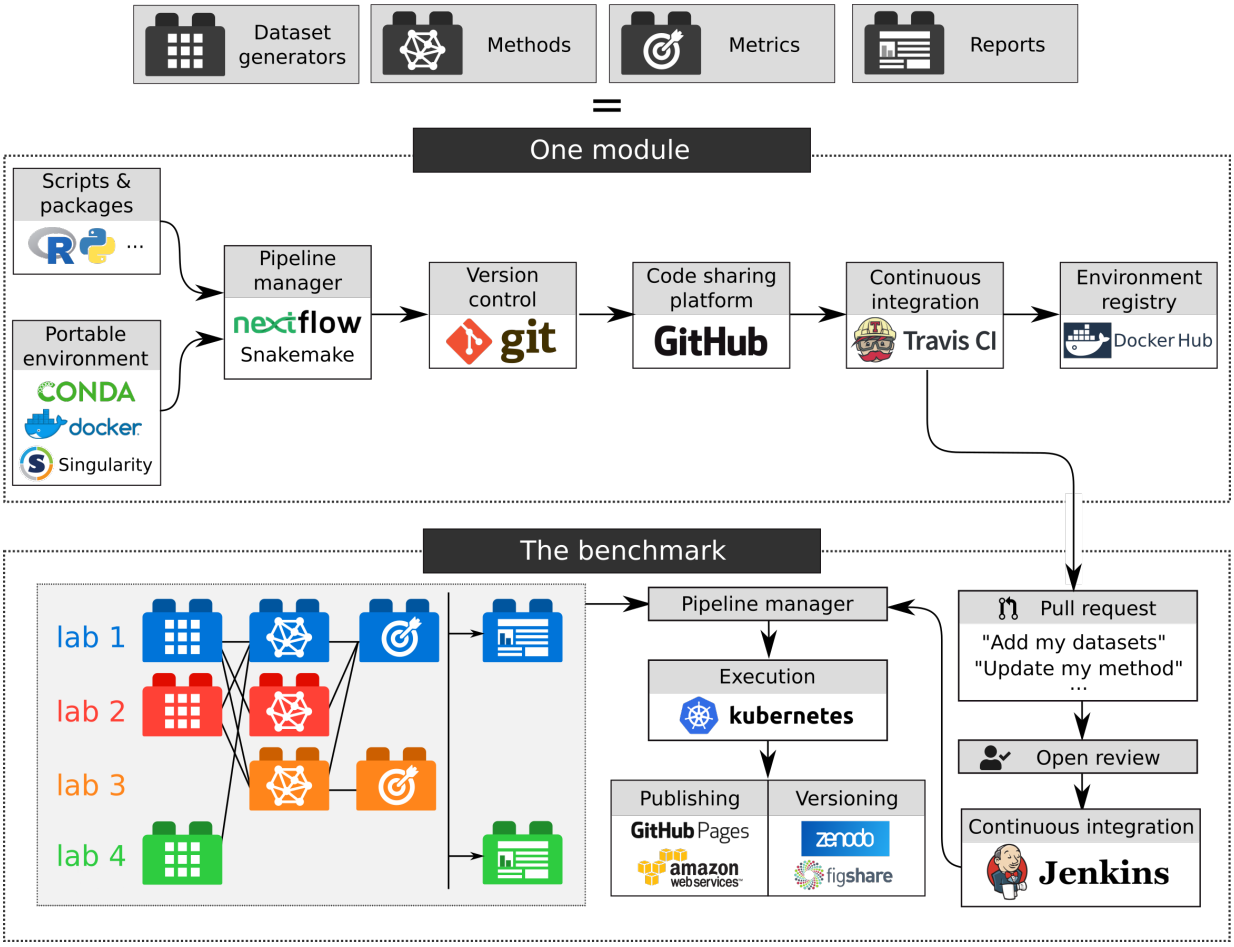

Figure 1: The pipeline.

Data formats

The basis of any collaborative effort in computation biology is agreeing on how data will be interchanged, and benchmarking is no exception. Sometimes, the differences between data formats can be minor, for example whether the samples within a gene expression matrix are put in the rows or in the column. In other cases, different data formats can have a significant impact on storage and/or the speed by which the data can be processed.

In the benchmarking workflow presented here, a format represents how a particular part of a dataset, model, score or report should be represented on disk.

What a format entails

For a format to be useful, it should have an unambiguous description. In this way, someone developing a module can be sure how the inputs look like, even if these inputs do not exist yet, and can also be sure that the outputs will be useful as input for other modules.

While a description is meant to be readable by humans, this description should also be translated into a lanuage computers can understand, so that each data file produced by a module can be validated. In this way, developers of a module can get immediate feedback on whether their output matches the format description. To make a format validatable, it is one possibility to use one of the many “schemas” available, such as json-schema (http://json-schema.org/), XML schemas or Apache Arrow Schemas. Often, there are already validators available for these schemas (json-schemas for example: https://json-schema.org/implementations.html#validators). On the other hand, for more custom file formats or complex behaviour, we will have to implement the validator ourselves.

Finally, connecting the human-readable description with the computation validation can be done with providing good and bad examples of the data format. These examples serve a double purpose, because they provide the module contributors several examples as a help to understand the description, but can also be used as test cases for the format validators.

Formats change as the field progresses

Usually, the most optimal representation of a dataset or the output of a method only becomes apparent when several methods have been developed already. This means that any effort to make a benchmark collaborative and continuous should strive to make its data formats flexible. Flexibility can take several forms. New features could be added to the format, without invalidating the old data and modules. When this is not an option, new formats could be added alongside the old. When applicable, converters should then be written which convert the old formats into the new, so that old modules keep on functioning. Finally, in some extreme cases, old formats could be

invalidated and replaced with new formats, which would require some versioning system to make sure modules are run on the version of the formats they were developed.

It is inevitable that disagreements about data representation will pop up in a collaborative effort. But in any case, having common formats, even if they are suboptimal for certain use cases, is usually better then having none at all.

Test case

For the TDE use case, to keep the formats simple and accesible, we decided to mainly use text-based formats such as comma-separated values (CSV) and JSON files, as these can be rapidly parsed in almost any programming language. For each format, we wrote custom validators in R, although JSON files were also partly validated using a json schema validator (ajv, https://ajv.js.org/). These validators are available as an R package (https://github.com/komparo/tde_formats). For each format, we also wrote a description and several examples, which are shown together in the contributor's guide (https://komparo.github.io/tde/formats.html).

To represent a dataset and a model, we needed to define several formats grouped within three categories:

- Single-cell expression datasets: This is relatively straightforward, given that the sheer number of single-cell RNA-seq tools available. As is commonly done, we represent the dataset as a matrix, together with two tables containing the metadata of the cells and features respectively.
- Trajectory: This is more difficult, given the vast diversity in outputs of trajectory inference methods. In our recent benchmarking of these methods, we already created a common format for trajectories, and included this format in this benchmark as well. However, as we also acknowledge in that study, alternative formats may be possible.
- Trajectory differential expression: This is the the most difficult, because we had to make some educated guesses about how we expect the field to progress. While it impossible to know beforehand what formats will be ultimately needed, we can at least try to already predict for them. Based on what is already done by some existing methods, we defined several types of trajectory differential expression such as oscillatory (within a cyclical trajectory), pseudotime (expression that changes along pseudotime), and branch point (expression that happens at a particular branch point). We extended this with some other types of differential expression, such as local (changes in expression at a particular point of the trajectory) and overall (changes in expression anywhere in the trajectory). Each kind of differential expression is represented in a table format, where each differential expression event is represented in a row which can contain information on the associated p-value, a ranking and/or effect size.

Scores and reports have less requirements regarding the formats. The most simplest format to represent a score is as an atomic value, in which each model can be summarized by one quantitave or qualitative value. Of course, more complex score formats are also possible. For

reports, we included two formats. A static report with at least an index HTML or markdown file, together with any extra assets such as figures. A dynamic report on the other hand produces a software container which will expose a port serving a web app when ran.

Module types

In the benchmarking workflow, we define four types of modules. In our experience, these four modules are in most cases enough to construct a fully comprehensive benchmark of certain group of methods.

Dataset generators

This module will generate a dataset, which can from very simple “toy” data, synthetic data which try to mimic the characteristics of real data as best as possible, or real datasets. Optionally, a dataset generator can also use another dataset as input, for example when generating some synthetic data based on some real data. Only rarely will a dataset generator contain some primary data itself. Rather, data should be gathered directly from primary sources, for example using APIs by various databases or by downloading the data directly from data management systems such as Zenodo or Figshare.

Within the TDE test case, we included

Methods

A method module reads in (parts) of a dataset, and uses this to generate a model. Some special types of methods can be helpful to include at the start of a benchmark. Positive controls, for example a method that simply return the reference model of the dataset, and negative controls, for example a method which generate a random model, could be useful to make sure the metrics work correctly. *Off-the-shelf* methods, methods that can be easily implemented with just a few lines of code, could be helpful as a baseline to other methods and to assess the difficulty of particular datasets.

A common issue when benchmarking is the selection method parameters. It is not uncommon that the authors of a method disagree on what parameter settings were used for benchmarking [8;]. In our workflow, method authors are required to define for each parameter a default value, but also a distribution of values which can be used for parameter tuning in any form.

Metrics

Metric modules score the output of a model. Some metrics assess the accuracy of a model by comparing it with some reference model present in the dataset. Others will look at the resources consumed by the method, such as CPU time and memory, to assess its scalability. Models can

also be compared to other models, for example to examine the stability of a method. Finally, some qualitative metrics can also be defined here, for example those that look at the usability of a method.

Report generators

In the end, the scores are aggregated and interpreted using a report generator. This modules generates a report which can be static, such as a markdown document with some figures, or dynamic in the form of a web application. By crowdsourcing the interpretation of the benchmark in this way, it would become much less authorative and instead promote open discussion in the field [9]. It might certainly happen that different reports would contain contradicting results, but because each reports starts from the same set of data, it would be immediatly clear why the conclusions differ. For example, there might be subtle differences in how the scores have been averaged. Or, a report may only have focused on only a subset of the dataset which the authors found the most relevant for their method.

Modules

Scripts, a portable environment and metadata

A module needs to contain at least command, which will run some code that reads in the input data, process it in some way, and ultimately write the output data in the correct format.

Given the large diversity of programming languages used in computation biology, a collaborative benchmarking effort should try to avoid imposing limits on the kind of programming languages that can be used. As an example, the single-cell analysis field is split between tools written for R and Python [10], and choosing one of these two would therefore alienate a signficant part of the field. Moreover, a collaborative effort should also be open for new languages such as Julia [11], which could be more powerful and developer friendly for certain use cases.

Apart from being language agnostic, the execution of a module should also happen on any computer in exactly the same way. To make the execution reproducible, we therefore require that a module defines a portable environment, which contains the necessary operating system, language interpreters and other packages to execute the code within the module. An environment can be portable on many levels: within one programming language such as virtualenv for python or packrat for R or across languages using package managers such as Conda. The most complete level of reproducibility can be obtained by working at the level of the operating system, through container systems such as docker or singularity. Finally, to be able execute stochastic code in a reproducible manner, it is also necessary to fix the pseudo-random number generator in some way, we do this by always setting an a priori defined seed through R or numpy.

A module also contains metadata, which lists the requirements to run the method such as the inputs, outputs and the name of the portable environment. Within our workflow, we also require some data for organisational purposes, such as a list of authors with their contributions, and the license of the code within the module.

Version control and code sharing

We require that the complete module, including the portable environment and metadata, is placed under version control so that any changes are tracked. The module is then shared on a code sharing platform, which makes it possible for other module authors and maintainers of the benchmark to file issues on the module, request some changes to the code through pull requests, and create a modifications if the license allows it. In our workflow, we use git for version control and GitHub as the platform to share modules, although it should be noted that powerful variants of the latter exist, including self-hosted ones.

Continuous testing and validation

To keep the development of a module and benchmark maintainable, it is important that each element of the module is automatically tested and validated. In this way, many errors are catched early, before they can impact other modules in the benchmark. Including automated testing also reduces the burden for those reviewing the modules. This crosstalk between automated testing and manual reviewing is already commonplace in many package repositories, such as CRAN and Bioconductor.

In our proposed workflow, we automatically trigger a new test on travis-ci.com, which is free for open-source projects. We test each module on several levels. We first check whether it contains all required content, and whether the metadata is complete. Next, we activate the portable environment, run the module on some small input data, and validate the produced output. If any of these steps fail, the author is notified. Only when tests are sucessful can the new module be integrated into the whole benchmark procedure.

Combining modules within a benchmark

To make the benchmark as inclusive as possible, it should be possible for anyone to extend or adapt the benchmark for their own purposes. At the same, it would also be useful to have a central place which lists all the modules and provides the most up-to-date set of reports for interested readers. To reconcile these two criteria, our benchmarking workflow has one “main” repository, which lists the location of the different modules and how they are combined in the benchmark. Anyone can however create a fork of this repository, adapt the modules or benchmarking design in any way, and run it using their own infrastructure.

Executing the benchmark

For the execution of the modules, a pipeline manager such as snakemake [12] or nextflow [13] is almost indispensable. These tools make sure the modules are executed in the correct order and within a reproducible environment. Moreover, to make the benchmark scalable, a pipeline manager will only rerun those exeuctions for which any inputs have changed, which includes changes to any scripts or packages inside the portable environment. Within our benchmarking workflow, we created a custom pipeline manager for this, which provided us with some features that are lacking in most current pipeline managers, such as incrementality at the level of the portable environment, output validation and fixation of the pseudo-random number generator.

Adding or updating a module

While anyone is able to fork and modify the benchmark repository, modifications to the main repository, such as additions or updates of a module, still requires some form of control by a group of maintainers. This group of maintainers, which would primarily consist of authors of other modules, are responsible for checking whether the module has passed all automated checks. and give feedback regarding data formats and testing results. Important here is that this reviewing happens in a completely open fashion, similar as to what is done in some open-source communities such as Bioconductor and ropensci (https://github.com/ropensci/onboarding).

In our workflow, adding or updating a module can be done by cloning the repository, making the necessary changes, and then creating a pull request on Github.

Continuous benchmarking and versioning

Every time the main repository is updated, for example with a new version of a module, an update of the whole benchmark workflow is triggered. Only those modules with outdated input, because the code, environment or some input files changed, are rerun.

The end results of the benchmark are one or more reports, which are freely accessible online. On regulator time intervals, such as on a monthly basis, all the reports are gathered and released as a new “version”, which includes a changelog of updates to any of the modules made before the last release. This release is given a digital object identifier and registered at an open-access repository such as zenodo or figshare.

Outlook

Continuous and collaborative benchmarking provides an alternative and powerful way to evaluate methods in computational biology. It relies heavily on modularisation and tools from software development, which make it possible to design a benchmarking strategy that can be easily

extended by anyone, while still allowing for open discussion to exist in a field. Because less effort is spend developing new benchmarking workflows for every new method, this workflow would speed up method development in bioinformatics. Because the results of the benchmark can be easily interpreted by anyone, it would also avoid other issues such as the self-assessment trap [14].

We created both a case study of the workflow https://github.com/komparo/tde, and a skeleton version which provides instructions on how to set up the workflow https://github.com/komparo/tde. Of course, as technologies and the communities building them come and go, the tools we used for this benchmark will almost certainly be replaced by more powerful tools soon. The crucial point is not which tools that are used, but what advantages they provide for the community: a portable environment, a reproducible workflow and a set of tools to collaboratively design a benchmark.

In the ideal case, a continuous benchmarking project should be supported by a larger consortium, such as the Human Cell Atlas, which would not only assure its continuity, but would also provide infrastructural support. In particular, services which have strong requirements on the side of storage and/or computing power would benefit from this, such as continuous integration, the code sharing platform and execution environment.

References

1. **Reproducible and replicable comparisons using SummarizedBenchmark**
Patrick K Kimes, Alejandro Reyes
Bioinformatics (2018-07-17) https://doi.org/gdvt5p
DOI: 10.1093/bioinformatics/bty627 · PMID: 30016409

2. **iCOBRA: open, reproducible, standardized and live method benchmarking**
Charlotte Soneson, Mark D Robinson
Nature Methods (2016-04) https://doi.org/gfj2zx
DOI: 10.1038/nmeth.3805 · PMID: 27027585

3. **Critical assessment of methods of protein structure prediction (CASP)-Round XII**
John Moult, Krzysztof Fidelis, Andriy Kryshtafovych, Torsten Schwede, Anna Tramontano
Proteins: Structure, Function, and Bioinformatics (2017-12-15) https://doi.org/gfj2zw
DOI: 10.1002/prot.25415 · PMID: 29082672 · PMCID: PMC5897042

4. **A benchmark for RNA-seq quantification pipelines**
Mingxiang Teng, Michael I. Love, Carrie A. Davis, Sarah Djebali, Alexander Dobin, Brenton R. Graveley, Sheng Li, Christopher E. Mason, Sara Olson, Dmitri Pervouchine, … Rafael A. Irizarry
Genome Biology (2016-04-23) https://doi.org/gfj2zz
DOI: 10.1186/s13059-016-0940-1 · PMID: 27107712 · PMCID: PMC4842274

5. **A descriptive marker gene approach to single-cell pseudotime inference**
Kieran R. Campbell, Christopher Yau
Cold Spring Harbor Laboratory (2016-06-23) https://doi.org/gfj23j
DOI: 10.1101/060442

6. **SCORPIUS improves trajectory inference and identifies novel modules in dendritic cell development**
Robrecht Cannoodt, Wouter Saelens, Dorine Sichien, Simon Tavernier, Sophie Janssens, Martin Guilliams, Bart N Lambrecht, Katleen De Preter, Yvan Saeys
Cold Spring Harbor Laboratory (2016-10-06) https://doi.org/gfj23n
DOI: 10.1101/079509

7. **Reversed graph embedding resolves complex single-cell trajectories**
Xiaojie Qiu, Qi Mao, Ying Tang, Li Wang, Raghav Chawla, Hannah A Pliner, Cole Trapnell
Nature Methods (2017-08-21) https://doi.org/gc5v2g
DOI: 10.1038/nmeth.4402 · PMID: 28825705 · PMCID: PMC5764547

8. **Parameter tuning is a key part of dimensionality reduction via deep variational autoencoders for single cell RNA transcriptomics**

Qiwen Hu, Casey S Greene
Cold Spring Harbor Laboratory (2018-08-05) https://doi.org/gdxxjf
DOI: 10.1101/385534

9. **Crowdsourced research: Many hands make tight work**
Raphael Silberzahn, Eric L. Uhlmann
Nature (2015-10-07) https://doi.org/gc4ntn
DOI: 10.1038/526189a · PMID: 26450041

10. **Exploring the single-cell RNA-seq analysis landscape with the scRNA-tools database**
Luke Zappia, Belinda Phipson, Alicia Oshlack
PLOS Computational Biology (2018-06-25) https://doi.org/gdqcjz
DOI: 10.1371/journal.pcbi.1006245 · PMID: 29939984 · PMCID: PMC6034903

11. **Julia: A Fresh Approach to Numerical Computing**
Jeff Bezanson, Alan Edelman, Stefan Karpinski, Viral B. Shah
arXiv (2014-11-06) https://arxiv.org/abs/1411.1607v4

12. **Snakemake–a scalable bioinformatics workflow engine**
J. Koster, S. Rahmann
Bioinformatics (2012-08-20) https://doi.org/gd2xzq
DOI: 10.1093/bioinformatics/bts480 · PMID: 22908215

13. **Nextflow enables reproducible computational workflows**
Paolo Di Tommaso, Maria Chatzou, Evan W Floden, Pablo Prieto Barja, Emilio Palumbo, Cedric Notredame
Nature Biotechnology (2017-04) https://doi.org/gfj52z
DOI: 10.1038/nbt.3820 · PMID: 28398311

14. **The self-assessment trap: can we all be better than average?**
R. Norel, J. J. Rice, G. Stolovitzky
Molecular Systems Biology (2014-04-16) https://doi.org/bxxmvz
DOI: 10.1038/msb.2011.70 · PMID: 21988833 · PMCID: PMC3261704