nature reviews genetics

https://doi.org/10.1038/s41576-023-00679-6

Review article

Check for updates

Challenges and best practices in omics benchmarking

In the format provided by the authors and unedited

Challenges and Best Practices in Omics Benchmarking

Thomas G. Brooks[1], Nicholas F. Lahens[1], Antonijo Mrčela[1], Gregory R. Grant[1,2]

1. Institute for Translational Medicine and Therapeutics
2. Department of Genetics, University of Pennsylvania

Supplementary Methods

Here we provide details on three topics. First, the evaluation of *p*-values and false discovery rates. Second, the inclusion of dependency within benchmark data (e.g., gene-gene expression level correlations). Thirdly, the production of maintainable benchmarks.

Note 1. Evaluating *p*-values

We address the issue of evaluating *p*-values in depth since we are not aware of other treatments of this targeted at benchmarkers and have observed it adversely affecting benchmark study recommendations to end users.

A tool to produce *p*-values is said to be *conservative* if it produces $p < \alpha$ with probability at most α, when run on a true null hypothesis. For example, we would expect the *p*-values generated by a test for DE between two conditions to be less than 0.05 at most 5% of the time on genes that satisfy the null hypothesis, typically meaning that their mean expression is the same in both groups. (Often this is instead called *super-uniform*[1,2] and conservative is reserved for cases where the *p*-values are super-uniform but not uniform.)

Testing for conservativeness is the most important test for tools that produce *p*-values as non-conservative *p*-values will lead to inflated Type I error rates.

Given data generated from a true null hypothesis (which is often generated from simulation, random permutation of real data, or negative controls), the *p*-value of the tool can be computed. Performing this many times, typically N=1000 to 10000 times, on different draws of the null datasets, the *p*-value distribution under the null can be computed. To test that the *p*-values are conservative at a particular α, compute the number of times it had $p < \alpha$ for a true null hypothesis, divide by the number of *p*-values computed, and ensure that this value is at most α (an explicit algorithm is given below).

This procedure depends upon a choice of α, which is most commonly set at $\alpha=0.05$. However, that a *p*-value procedure is conservative at $\alpha=0.05$ does not mean it will also be conservative at other values of α and often methods lose control at lower values. In omics we often test tens of thousands of hypotheses simultaneously (e.g., one for each gene), so $p < 0.05$ is generally inadequate, so control at lower values of α should also be tested, chosen based off a typical use for the tool. For example, a genome-wide association study (GWAS) tool should be assessed for control at the typical threshold of $\alpha=5\times10^{-8}$.[3] Furthermore, the choice of N limits the choice of α since we must always have N to be at least $1/\alpha$. This means that large numbers of simulations are necessary to properly assess *p*-values in some omics contexts.[4]

A useful tool for assessing *p*-values is the quantile-quantile plot, comparing reported *p*-values with the expected uniform distribution. We recommend plotting this on a log-log scale. Compared to a histogram

of *p*-values, this emphasizes deviation from uniformity at low *p*-values, which is most problematic for conservativeness. The plot should be assessed for being (on average) at or above the expected quantiles. For benchmarking purposes, this plot should only be done on *p*-values from true null hypotheses.

Uniformity of the p-value distribution

We recommend against including other measures of uniformity of the *p*-value distribution in a benchmark evaluation. While assessing uniformity (outside of just conservativeness and power assessments) has value in other contexts, we find it is prone to misinterpretation in benchmarks. For example, a tool developer may know that their method is intended to produce uniform *p*-values and so finding a non-uniform distribution would be concerning even if conservativeness or power were not compromised. However, many tools are not expected to produce a uniform distribution of *p*-values, and so a benchmarker making the same assessment will misjudge these other tools as broken, which we have observed in practice[5].

In fact, tools that correctly produce non-uniform *p*-values are quite common. One example is any method that performs a Bonferroni correction for testing multiple hypotheses. Bonferroni correction makes no assumptions (other than that each input *p*-value is conservative), but it is only uniform in the unlikely special case that the corrected *p*-values are completed anti-correlated with each other (so that no two are ever simultaneously significant). Another example is a one-sided *t*-test, which is non-uniform in the case where the effect goes in the opposite direction. For example, consider evaluating a one-sided test for peak calling algorithms. Peaks are only expected to be enrichments, so a one-sided test is justified, but some genomic regions will be depleted due to biases in the data. These will lead to a skew in *p*-values towards 1 even for true nulls. However, this creates no increase in either false positives or false negatives. Another example are tests on discrete data, particularly when there are few possible outcomes. Lastly, more robust tests often have more non-uniform *p*-values, with a typical example being the Welch's *t*-test will produce non-uniform *p*-values yet benefits from not needing to assume homoscedasticity. Therefore, we recommend that benchmarkers do not evaluate uniformity of *p*-values as it does not distinguish between validly non-uniform and erroneously non-uniform distributions.

Conservative but non-uniform distributions suggests that a test might have low power. However, this can be assessed directly by computing power (as above), which we recommend. Likewise, uniformity of *p*-values indicates conservativeness, but is not as good a measure of that the direct tests for conservativeness that we recommend. In practice, one recent benchmark that computed both control of false discovery rate and *p*-value distributions[6] showed uniform-looking distributions in some methods that had very poor control of the false discovery rate, indicating the need to go beyond histograms of *p*-value distributions.

Algorithms for performing assessments

Below, we write these common *p*-value assessment procedures as explicit algorithms.

Checking *p*-values for conservativeness:

1. Choose α.
2. Generate N (1000-10000) null data sets.
3. Run the tool on each data set, obtaining a *p*-value for each run.
4. Compute the number of times $p < \alpha$ occurred and divide by N.
5. If this is at most α, then the tool was conservative.

Computing the power of a tool:

1. Choose α and an effect size T (determined by the type of hypothesis being tested).
2. Simulate N data sets with the effect size T. N does not need to be too large (100-1000 typically).
3. Compute the *p*-values of each data set.
4. The power at effect size T is then the number of times $p < \alpha$ divided by N.
    a. If any tools were not conservative (see above), then instead one must first correct the reported *p*-values according to the non-conservativeness. Specifically, if the tool required $p < C$ in order to control false positive rate at α, then use $p < C$ instead of $p < \alpha$.

Comparing the power of tools:

1. Choose an α.
2. Run all tools on real data sets.
3. Compute the number of data sets with $p < \alpha$ for each tool.
    a. If any tools were not conservative (see above), then instead one must first correct the reported *p*-values according to the non-conservativeness. Specifically, if the tool required $p < C$ in order to control false positive rate at α, then use $p < C$ instead of $p < \alpha$.
4. The tool with the highest number has the highest power (on the datasets evaluated), but exact power cannot be calculated. This is only useful if the dataset has non-null behavior in some subset of the variables.

Evaluating False Discovery Rates (*q*-values):

1. Choose an FDR cutoff α.
2. Run all tools on data with known true/false status of the null hypotheses and compute FDR *q*-values.
3. Mark each hypothesis with $q \leq \alpha$ as discoveries.
4. Compute the fraction F of discoveries that are true nulls (i.e., false positives). If no discoveries at all are made, set F to 0.
5. Repeat on multiple such datasets, recording each F.
6. Check if the mean of F is at most α.

Note 2. Generating dependency in data

Most real omics datasets contain dependencies and correlations between individual measurements done on the same biological sample. For example, gene expression values are not independent between genes in the same sample. This is often not captured in benchmarking data, particularly when simulated. However, there are several approaches to including it in a benchmark study.

Multivariate normal distribution

Perhaps the simplest is when generating spreadsheet level data parametric distributions (particularly the normal distribution), the data can be generated with dependence by using the appropriate multivariate distributions, in particular the multivariate normal. Levels of dependence may thereby be parametrized as well by choosing the covariance matrix of the data to generate. The covariance matrix is an $n \times n$ matrix for the n dimensional multivariate distribution. See Covariance Matrices below for strategies for choosing a covariance matrix.

Copulas and non-parametric rank dependence

Copulas are the statistical object that encodes the dependencies of random variables. One particular use for them is to define dependencies for simulation. These can be employed to generate dependencies between data regardless of their marginal distributions. For example, in RNA-seq, gene expression values are often assumed to follow a negative binomial distribution, which would be the marginal distribution of each gene's expression. A copula can be used to impart dependence between two negative binomial distributed gene expression values. Guidance and software for working with copulas can be found at the R package *copula*[7].

Covariance matrices

Alternatively, there are means of generating data with a specified covariance matrix while still controlling the marginal distributions. These work by means of a multivariate normal distribution. There are existing general-purpose data-simulation packages, including the R packages *bindata*[8] (for correlated binary data), *GenOrd*[9] (for correlated discrete data), and SimMultiCorrData[10] (for both discrete or continuous correlated data).

Non-parametric

The Iman-Conover transformation[11] provides a non-parametric method for inducing a desired rank correlation to data without affecting the marginal distributions.

Real data

Using real data to prime a simulation often inherently captures dependence without needing it to be fully characterized. For example, estimating values from one dataset and then generating simulated data from independent distributions whose means are equal to the estimated real values. This would include dependence in the biological variation but exclude dependence in technical variation (such as Poisson shot noise from sequencing). Depending on the assay type and the nature of the ground truth being used for benchmarking, this might be appropriate. For another example, when benchmarking polygenic risk scores, using actual genotype data naturally captures linkage disequilibrium while still allowing a simulation with known causal variants of specified effect sizes[12].

Permutation

When generating benchmarking datasets via permutation of real datasets, permute labels on the samples rather than permuting measured values within the sample. For example, given a spreadsheet of

gene expression data with rows for genes and columns for samples, one should treat entire columns as units, do not permute the values for each gene independently of the other genes. This way, dependent measurements should stay together in the permutation, and that should be the goal in general. This is particularly effective at creating null datasets with no true positive effects. Sometimes, these can then be used as steppingstones to generate datasets that also include true positive effects.

Covariance Matrices

It is often unclear what to pick when simulating large omics datasets. Two general strategies are to specify top principal components or to do block dependence.

When performing principal component analysis (PCA), a dataset is reduced to a small number (often just 2) of components. Likewise, we can specify the top principal components of our correlation matrix by picking some number of vectors of length *n* (which, recall, is equal to the number of features measured per sample). These vectors give weights and should be thought of as 'how much' each feature contributes to a component. In particular, zero values indicate that a feature does not contribute at all and so can be helpful to give the desired structure. If we pick *k* such vectors, we then can put them as the columns of a matrix *U*. Then we pick how important each component should be (its variance) by choosing values $\lambda_1, \lambda_2, \ldots \lambda_k$. These must all be positive. Then we can generate a covariance matrix that has these components by: $\Sigma = UDU^T$ where D is the diagonal matrix that has $\lambda_1, \lambda_2, \ldots \lambda_k$ along its diagonal and U^T means the matrix transpose of *U*. One way to choose U is to perform PCA on a real dataset and set U to be the PCA's weight matrix and $\lambda_1, \lambda_2, \ldots \lambda_k$ as the variance of each principal component.

However, this has the downside that the data will then only fall on the *k*-dimensional subspace spanned by the chosen vectors, i.e., it will only ever be in the *k*-dimensional PCA projection of the original data. This is unlikely to reflect the true variability of real dataset, even if *k* is set as high as possible (i.e., the number of samples, which is generally much smaller than the number of variables measured in omics experiments). Nonetheless, it does capture large dependencies and we can improve the situation by adding on to our matrix Σ any diagonal matrix (with non-negative values on the diagonal, usually with mostly positive values). Building off the PCA construction above, one good choice for this step would be to just set the diagonal elements of Σ equal to the observed variance of the corresponding variable. This is equivalent to adding on all the 'missing' variance that wasn't captured by the *k*-dimensional PCA but assuming that all of this extra variance has each variable independent from the others. So it assumes that all the dependence was captured by the *k*-dimensional PCA, but this is a much less severe assumption than the naïve simulation where all variables are completely independent. We provide code to compute this covariance matrix below using R:

```
covariance_from_pc <- function(data, k) {
  # Finds a covariance matrix from the data that is
  # appropriate for simulation by assuming that all
  # the independence lies in the top k principal components
  # but still makes the variance of each individual
  # variable equal to the observed variance.
  # The output is a covariance matrix, suitable for
  # use with mvrnorm as the Sigma parameter.

  # Compute the (sample) variances of each measurement in the data
```

```
  variances <- apply(data, c(1), var)

  # Perform principal component analysis
  pc <- prcomp(t(data), rank. = k)
  pcvars <- diag(head(pc$sdev, k), nrow=k)^2
  # Extract the covariance matrix of the k principal components
  sim_cov <- pc$rotation %*% pcvars %*% t(pc$rotation)
  # Ensure all variables have the appropriate variances
  diag(sim_cov) <- variances
  return(sim_cov)
}
```

The second common structure to use for covariance matrices is a block diagonal format. This setup assumes that variables can be grouped into blocks where there is no correlation outside of the block. For example, this can approximate linkage-disequilibrium. Data with block covariance can be simulated by using a block diagonal covariance matrix. The most straight-forward approach is to divide the features into different groups, where we assume there is no dependence between the groups. Then, simulate each group individually (such as by using the previous principal component strategy) and concatenate the results. This allows for control over more aspects of the dependence structure and makes for more efficient simulation.

Lastly, we note that with omics-wide approaches, directly using full covariance matrices can be computationally prohibitive. If block structures like above are not realistic, it may also be possible to directly work with the top principal components strategy to efficiently generate data in two steps. We describe this for the multivariate normal distribution. First, generate data using just the top *k* components of the covariance matrix. This can be done efficiently by generating k univariate standard normal variables (i.e., assuming independence and with mean zero and standard deviation 1), then multiplying by the *k* loadings vectors of the principal component. This generates data for every feature but with unnaturally low variance and relatively high dependence. To correct this, then generate the remaining variance as univariate normal data for each feature (i.e., assuming independence). This is equivalent to the previous method mathematically but computationally very fast if *k* is small. We provide code in R for this strategy below.

```
mvrnorm_from_pc <- function(data, k, n_samples) {
  # Samples from a multivariate normal distribution that has:
  # - each variable has the same variance as in the observed data
  # - the same variance in the top k principal components observed in data
  # The mean is assumed to be zero

  # Compute the (sample) variances of each measurement in the data
  variances <- apply(data, c(1), var)

  # Perform principal component analysis
  pc <- prcomp(t(data), rank. = k)

  # Draw from the multivariate normal distribution with
  # the dependence structure of the principal components
  # but done efficiently by transforming a standard normal
  indep_draws <- matrix(rnorm(k*n_samples), c(k, n_samples))
```

```
  sdev <- diag(head(pc$sdev, k), nrow=k)
  pc_draws <- pc$rotation %*% sdev %*% indep_draws

  # Add in the missing variance to match the actual data
  # by drawing independent data with the appropriate variance
  missing_var <- variances - (pc$rotation %*% sdev)^2
  n_features = dim(data)[1]
  indep_draws <- matrix(rnorm(n_features*n_samples,
                              sd=rep(sqrt(missing_var), n_samples)),
                        c(n_features, n_samples))

  return(pc_draws + indep_draws)
}
```

Note 3. Reproducibility and maintainability of benchmarking pipelines

In order to enhance the reproducibility of the benchmarking study and facilitate the incorporation of future revisions to its findings and recommendations following the publication of new tools or updates to existing tools, it is strongly advisable to define the full benchmarking pipeline in code. There are many bioinformatics workflow managers that enable the definition of pipelines in a programmatic manner and use containerization[13] to ensure portability. Considering the criteria important for specifying and subsequently executing benchmarking pipelines[14], particularly the expressiveness of the workflow specification language (including support for modularity) and portability, as well as the ease of use and widespread adoption, Snakemake[15] and Nextflow[16] emerge as the most favorable options [17,18].

In the context of benchmarking pipelines, it is evident that adopting a modular approach, wherein each benchmarked tool is placed in a dedicated container image, is highly desirable, as it not only improves portability, but also allows incorporation of any future update to a benchmarked tool in a confined manner. However, several obstacles can arise when attempting to accomplish this level of modularity:

1. The workflow manager needs to support executing tasks in containers
2. The tools have to be containerizable
3. The execution of tools in containers has to be feasible in terms of time and cost

We give below our perspective on each of these issues.

Execution in containers

While both Snakemake and Nextflow (as well as many other bioinformatics workflow managers) support executing tasks in containers, the extent of that support varies. According to our assessment of their respective documentation, Nextflow offers a wider range of functionalities and options pertaining to the specification and execution of containerized tasks in comparison to Snakemake. In particular, Nextflow's Wave containers allow for definition of task images in the form of Dockerfiles which can be updated and version controlled as benchmarked tools evolve. (Dockerfile describes the contents of a container image using a domain specific language (DSL). If a Dockerfile specifies installation of a software inside the image, without pinning the software's version, then two images built from the same Dockerfile may differ. If needed, special package managers, such as Conda[19] or GNU Guix[20], may be used to minimize these risks.) Based on this analysis, we find that Nextflow presents several benefits over Snakemake in the context of pipeline specification for benchmarking studies. However, we acknowledge the dynamic nature of software development and the possibility that the comparative advantage may shift in the future.

Containerizing tools

If a benchmarked tool is a package or module in a language such as Julia, Python or R, or a custom script written in Bash, Perl or Python, its containerization is typically straightforward[21]. Compiled binaries provided by the developers of the tool will generally function properly, but on occasion, issues may arise as a result of absent or incompatible dependencies, necessitating the compilation of the tool from its source code during the image creation process. Regardless of whether the tool is implemented in a scripting or a compiled language, difficulties may emerge, and cracking the appropriate combination of dependencies and configurations can be a laborious task.

A benchmarked tool might itself constitute a pipeline, integrating several publicly accessible or custom designed tools. Since such a benchmarked tool often needs significant computational resources for its

execution, the authors may have equipped it with the ability to run its workload in a distributed way on high-performance computing (HPC) or cloud platforms using custom orchestration scripts or a bioinformatics workflow manager. This situation may increase the challenge of containerization, especially if the tool does not directly allow local execution or has more specific hardware requirements. However, most workflow managers provide local execution option which can be used to run the tool's workflow inside of a container, while custom orchestration scripts may sometimes be handled by intercepting their job submission or monitor requests with a program which, inside the container, mimics the HPC or cloud job manager interface.

Additional problems can arise when attempting to containerize a benchmarked tool which needs to run containers. In the case of Docker, this can be handled by mounting the host's Docker socket into the tool's container. However, this approach may not easily generalize across different container runtimes or execution platforms (e.g., Apptainer[22]).

Computational resources and feasibility

The execution of a benchmarked tool within a container may give rise to cost and time implications if the tool requires distribution of workload across many CPU cores. While the container can utilize as many CPU cores as there are available on the host (possibly hundreds), the benchmarked tool's workload may only be partially parallelized (e.g., only the initial steps in the pipeline are parallelized), resulting in underutilization of the host and higher execution cost. This may be possible to mitigate by optimizing the number of CPU cores assigned to the container or dividing the benchmarked tool's pipeline into highly and minimally parallelized sections and allocating the CPU resources accordingly.

Depending on the type of the benchmarked tool, the ease of its containerization and execution may play a role in the evaluation of the tool's ease of installation and use.

We note that these issues are not exhaustive, and benchmarkers should anticipate additional challenges when defining and executing their benchmarking pipelines. If the benchmarkers encounter uncommon difficulties and recognize that due to these difficulties their pipeline relies heavily on a particular hardware or operating system environment, they may consider employing infrastructure-as-code methodology to specify these environment requirements. This can be achieved by utilizing domain-specific languages, for example declarative Terraform HCL, or a more well known imperative programming languages such as Python or Java using Cloud Development Kit for Terraform (CDKTF) or AWS Cloud Development Kit (AWS CDK). By explicitly defining the environment requirements in such languages, researchers gain the ability to provision the required resources on a cloud platform in an automated and reproducible way.

References

1 Chen, X. & Sarkar, S. K. On Benjamini–Hochberg procedure applied to mid p-values. *Journal of Statistical Planning and Inference* **205**, 34-45 (2020). https://doi.org:https://doi.org/10.1016/j.jspi.2019.06.001

2 Lyu, P., Li, Y., Wen, X. & Cao, H. JUMP: replicability analysis of high-throughput experiments with applications to spatial transcriptomic studies. *Bioinformatics* **39** (2023). https://doi.org:10.1093/bioinformatics/btad366

3 Xu, C. *et al.* Estimating genome-wide significance for whole-genome sequencing studies. *Genet Epidemiol* **38**, 281-290 (2014). https://doi.org:10.1002/gepi.21797
4 Mehta, T., Tanik, M. & Allison, D. B. Towards sound epistemological foundations of statistical methods for high-dimensional biology. *Nature Genetics* **36**, 943-947 (2004). https://doi.org:10.1038/ng1422
5 Laloum, D. & Robinson-Rechavi, M. Methods detecting rhythmic gene expression are biologically relevant only for strong signal. *PLoS Comput Biol* **16**, e1007666 (2020). https://doi.org:10.1371/journal.pcbi.1007666
6 Buratin, A., Bortoluzzi, S. & Gaffo, E. Systematic benchmarking of statistical methods to assess differential expression of circular RNAs. *Briefings in Bioinformatics* **24** (2023). https://doi.org:10.1093/bib/bbac612
7 copula: Multivariate Dependence with Copulas v. 1.1-2 (2023).
8 bindata: Generation of Artificial Binary Data (R package version 0.9-20, 2021).
9 GenOrd: Simulation of Discrete Random Variables with Given Correlation Matrix and Marginal Distributions (R package version 1.4.0, 2015).
10 SimMultiCorrData: Simulation of Correlated Data with Multiple Variable Types (R package version 0.2.2, 2018).
11 Iman, R. L. & Conover, W. J. A distribution-free approach to inducing rank correlation among input variables. *Communications in Statistics - Simulation and Computation* **11**, 311-334 (1982). https://doi.org:10.1080/03610918208812265
12 Pham, D. *et al.* Assessing polygenic risk score models for applications in populations with under-represented genomics data: an example of Vietnam. *Briefings in Bioinformatics* **23** (2022). https://doi.org:10.1093/bib/bbac459
13 Kadri, S., Sboner, A., Sigaras, A. & Roy, S. Containers in Bioinformatics: Applications, Practical Considerations, and Best Practices in Molecular Pathology. *J Mol Diagn* **24**, 442-454 (2022). https://doi.org:10.1016/j.jmoldx.2022.01.006
14 Ahmed, A. E. *et al.* Design considerations for workflow management systems use in production genomics research and the clinic. *Sci Rep* **11**, 21680 (2021). https://doi.org:10.1038/s41598-021-99288-8
15 Molder, F. *et al.* Sustainable data analysis with Snakemake. *F1000Res* **10**, 33 (2021). https://doi.org:10.12688/f1000research.29032.2
16 Di Tommaso, P. *et al.* Nextflow enables reproducible computational workflows. *Nat Biotechnol* **35**, 316-319 (2017). https://doi.org:10.1038/nbt.3820
17 Wratten, L., Wilm, A. & Goke, J. Reproducible, scalable, and shareable analysis pipelines with bioinformatics workflow managers. *Nat Methods* **18**, 1161-1168 (2021). https://doi.org:10.1038/s41592-021-01254-9
18 Jackson, M., Kavoussanakis, K. & Wallace, E. W. J. Using prototyping to choose a bioinformatics workflow management system. *PLoS Comput Biol* **17**, e1008622 (2021). https://doi.org:10.1371/journal.pcbi.1008622
19 Gruning, B. *et al.* Bioconda: sustainable and comprehensive software distribution for the life sciences. *Nat Methods* **15**, 475-476 (2018). https://doi.org:10.1038/s41592-018-0046-7
20 Courtès, L. Functional package management with guix. *arXiv preprint* (2013). https://doi.org:https://doi.org/10.48550/arXiv.1305.4584
21 Strozzi, F. *et al.* Scalable Workflows and Reproducible Data Analysis for Genomics. *Methods Mol Biol* **1910**, 723-745 (2019). https://doi.org:10.1007/978-1-4939-9074-0_24
22 Kurtzer, G. M., Sochat, V. & Bauer, M. W. Singularity: Scientific containers for mobility of compute. *PLoS One* **12**, e0177459 (2017). https://doi.org:10.1371/journal.pone.0177459