The Journal of Open Source Software

open_nipals: An sklearn-compatible Python package for NIPALS dimensionality reduction

Niels Schlusser [1*], **Ryan M. Wall** [2*], **and David R. Ochsenbein** [1]

1 Cilag GmbH International, Schaffhausen, Switzerland **2** Johnson & Johnson Innovative Medicine, Titusville, New Jersey, USA * These authors contributed equally.

DOI: 10.21105/joss.08598

Software

- Review
- Repository
- Archive

Editor: Yuanqing Wang

Reviewers:

- @maximtrp
- @rsenne
- @ankur-gupta

Submitted: 28 April 2025
Published: 22 July 2026

Summary

`open_nipals` is a Python package that implements the Nonlinear Iterative Partial Least Squares (NIPALS) algorithm (Geladi & Kowalski, 1986) for Partial Least Squares (PLS) regression as well as Principal Component Analysis (PCA). It employs the data transformation methods `fit()` and `transform()` from `scikit-learn` (Pedregosa et al., 2011) and leverages Nelson's Single Component Projection (SCP) method for the imputation of missing data (Nelson et al., 1996). The NIPALS algorithm represents an alternative to the common Singular Value Decomposition (SVD) procedure for both PCA and PLS implemented in `scikit-learn` (Pedregosa et al., 2011). It is an iterative procedure that processes the data and internal matrices vector-wise and iteratively. When combined with SCP, NIPALS allows natural handling of missing data and setting tailored accuracy goals.

Statement of Need

Python has emerged as a popular and comparatively simple programming environment for the development of machine learning and data science applications. Packages like `NumPy` for vector operations (Harris et al., 2020), `pandas` for the handling of tabular data (The pandas development team, 2020), and `scikit-learn` (abbreviated `sklearn` in the following) for common machine learning techniques like random forests, support vector machines (SVM), and principal component analyses (Pedregosa et al., 2011) promote Python's success in extracting patterns from big and complex data sets. However, `sklearn` relies on Singular Value Decomposition (SVD) for its PCA and PLS classes, with negative effects on performance for applications like batch manufacturing and chemometrics, where missing data is common (Nelson et al., 1996). PCA and PLS models require unit-scaled and mean-centered input data, a feature that is nicely implemented in `sklearn`'s `StandardScaler` class.
To this end, we felt the need to complement `sklearn` with an implementation of the NIPALS algorithm for PCA and PLS. As will be discussed below, the NIPALS algorithm has especially beneficial properties when it comes to peak memory consumption scaling and accuracy. Since it is an iterative algorithm, the runtime requirement can be balanced with the desired accuracy target. In general, it is a favorable option at a small number of latent variables (`n_components < 10`) and a high numerical accuracy requirement.

Why a separate package?

We decided to wrap the implementation of NIPALS PCA and PLS into a separate package instead of extending `sklearn` by another model because:

1. `open_nipals`' use case is very specific to batch manufacturing and chemometrics (cf. benchmarking). `sklearn`'s audience is a broader Python machine learning community.

2. `open_nipals` follows a slightly different philosophy in that it integrates the missing value imputation into the proper package, and therefore does not align with `sklearn` in that particular aspect. Integrating the imputation of missing data into the analysis package comes with a performance advantage.
3. Keeping `open_nipals` and `sklearn` apart facilitates maintainability of both packages.

Related Software

To our knowledge, the only other maintained open-source Python package that implements the NIPALS algorithm for PCA and PLS is Salvador García Muñoz' `pyphi` (Garcia Munoz et al., 2019). Our implementation is different in the following aspects:

1. `open_nipals` follows the template of `sklearn`, which allows:
    1. Integration with other `sklearn` modules, e.g. the `StandardScaler`
    2. Accumulation of multiple transformation steps into a `sklearn.pipeline`.
2. `open_nipals` uses Nelson's single component projection method (Nelson et al., 1996) for score calculation in the face of missing values.
3. The utility class of `open_nipals` contains `ArrangeData`, another `sklearn`-style data transformer object that ensures correct ordering and quantity of input columns.

Functionality

Wherever possible, `open_nipals` follows and inherits structures from parent classes in `sklearn`. In principle, its functionality can be split into three parts:

1. Utility functions for data preprocessing
2. Principal Component Analysis
3. Partial Least Squares regression.

We decided to combine PCA and PLS functionality into one package, such that they can share common utility functions, e.g. – but not limited to – the `ArrangeData` class and matrix multiplication with missing values.

Data Preprocessing and Utility Functions

It is *strongly* encouraged to mean-center the input data for both PCA and PLS, and scale their variance to unity, e.g. with `sklearn`'s `StandardScaler`. However, the informed user should still have the chance to also apply `open_nipals` to non-standardized data. Therefore, we did not make standardization a part of `open_nipals`, unlike `sklearn.PCA`, which automatically mean-centers input data when not already done. Moreover, the `ArrangeData` class of `open_nipals` ensures correct ordering of the input columns. A code example for preprocessing could therefore look like:

```
from open_nipals.utils import ArrangeData
import pandas as pd
from sklearn.preprocessing import StandardScaler

# Load data
df = pd.read_csv('my_data.csv')

# Create objects
arrdat = ArrangeData()
scaler = StandardScaler()

# Fit and transform data using both objects
data = scaler.fit_transform(arrdat.fit_transform(df))
```

PCA

Principal Component Analysis in `open_nipals` uses a `NipalsPCA` transformer object that can be fitted to input data and used to transform it (and both at once), e.g. with:

```
from open_nipals.nipalsPCA import NipalsPCA

model = NipalsPCA()
transformed_data = model.fit_transform(data)
```

The number of fitted components can be specified with the `n_components` argument in the constructor, which defaults to `n_components=2`. After having constructed the object, components can be added or subtracted using the `set_components()` function. Once fitted, components are stored so they do not have to be fitted again. This saves compute time should the developer decide to use a lower number of components than are fitted and later move back to a higher number of principal components.

The following functions and attributes of the `sklearn` API are implemented by `NipalsPCA`:

- `fit(X)` to fit a new `NipalsPCA` model to a data set `X`
- `transform(X)` to transform a data set `X` according to a model previously fitted
- `fit_transform(X)` to do both above steps in one
- `inverse_transform(X)` to predict original data from transformed input data `X`
- `explained_variance_ratio_` to return the variance ratios explained by each component

Please note that the NIPALS algorithm does not compute eigenvalues of the covariance matrix; therefore computing them specifically for `explained_variance_` seemed unnatural. Thus, this attribute was not implemented.

The distance of a given data point from the average of the training data within the PCA model (in-model distance, IMD) can be calculated with `calc_imd()`, where Hotelling's T^2 (Hotelling, 1931) is implemented and could be extended to other IMD metrics (e.g. Mahalanobis distance). Conversely, the out-of-model distance (OOMD, calculated by `calc_oomd()`) gives a measure of the distance to the model hyperplane. This is available as two metrics, `DModX` and `QRes` (Eriksson et al., 1999).

Finally, the `calc_limit()` function calculates theoretical limits on both IMD and OOMD such that a specified fraction `alpha` of the data lies within these limits, assuming the data follows an f-distribution (Brereton, 2016).

Following the data preparation scheme detailed in the benchmarking section, we simulated 1000 samples in a 40-dimensional data space, and trained a 4-component PCA model on it. Figure 1 shows the IMD-OOMD plot given this model and the theoretical limits calculated with it.

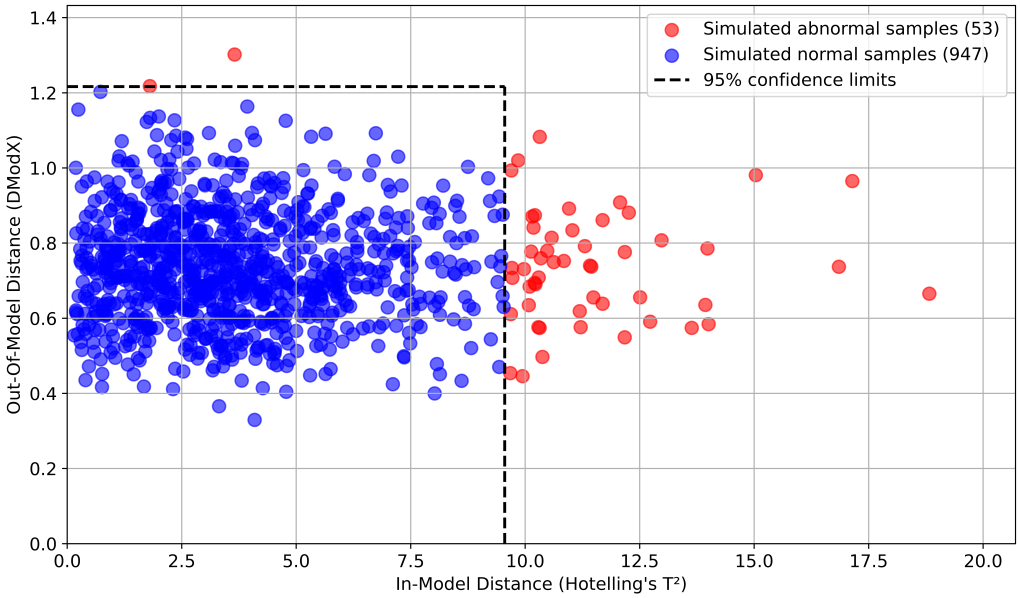

Figure 1: PCA-modelled data points on the IMD-OOMD plane with 0.95 confidence interval.

PLS

Beyond projecting input data onto a low-dimensional latent space, as PCA models do, PLS models can also predict dependent variables.

Initializing a basic PLS model with a `NipalsPLS` object and transforming the input data accordingly looks like:

```
from open_nipals.nipalsPLS import NipalsPLS

model = NipalsPLS()
transformed_x_data, transformed_y_data = model.fit_transform(data_x, data_y)
```

The following functions and attributes of the `sklearn` API are implemented by `NipalsPLS`:

- `fit(X)` to fit a new `NipalsPLS` model to data sets `X` and `y`
- `transform(X, y=None)` to transform a data set `X` according to a model previously fitted, with the option of adding a `y` data set
- `fit_transform(X,y)` to do both above steps in one
- `inverse_transform(X)` to predict original data from transformed input data `X`
- `predict(X)` to predict dependent variables `y` given the model
- `explained_variance_ratio_` to return the `X` and `y` variance ratios explained by each component

As for `NipalsPCA`, `NipalsPLS` does not implement `explained_variance_` since the eigenvalues are not accessible as a byproduct of the NIPALS PLS algorithm.

Beyond standard `sklearn` functionality, `NipalsPLS` implements `calc_oomd()` for the out-of-model distance with either `QRes` or `DModX` as implemented metrics, `calc_imd()` for the in-model distance, using the Hotelling's T^2 metric. Summary statistics of PLS models can be displayed in similar plots to Figure 1.

`NipalsPLS` primarily differs from `NipalsPCA` by the inclusion of a `predict()` method to predict a y-matrix from an x-matrix with a previously fitted model, and the calculation of the regression vector with `get_reg_vector()`. The latter serves as a measure for what input features the model considers predictive of the output, see Figure 2.

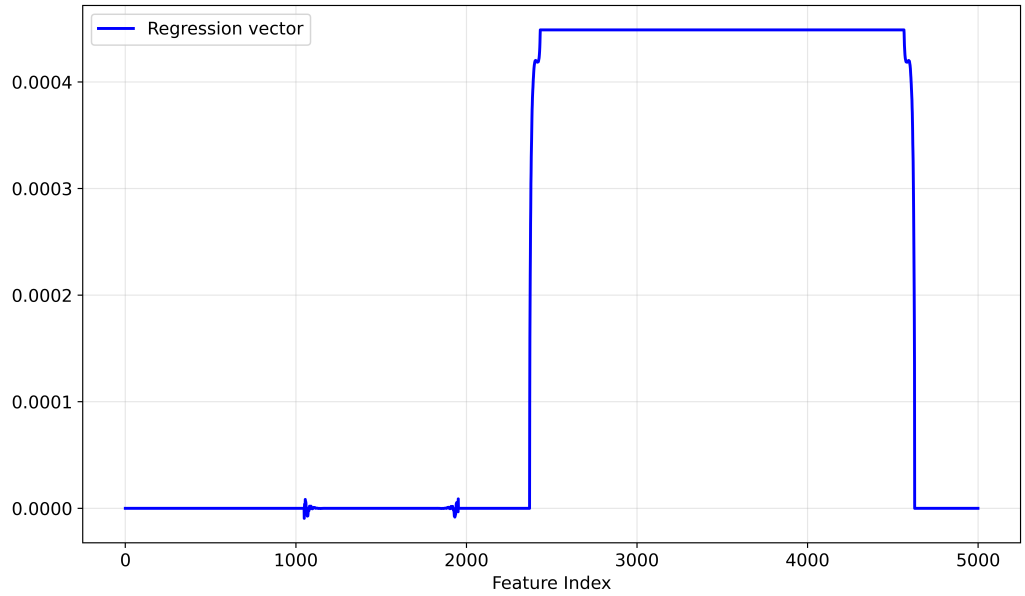

Figure 2: Visualization of a PLS regression vector, generated from simulated test data inspired by spectroscopy.

Debugging

If the maximum iteration counter is hit during the fitting procedure of new components of an `open_nipals` PCA or PLS model, a `max_iter reached on LV {ind_LV}` warning is raised. This indicates that the desired numerical tolerance for this component could not be achieved during the fit procedure. Try to increase `max_iter`, or increase `tol_criteria` if less numerical precision is still acceptable in your use case.

Default values for both `NipalsPCA` and `NipalsPLS` are `max_iter = 10000` and `tol_criteria = 1e-6`. These defaults manifest a slightly higher emphasis on numerical accuracy than performance.

If the user observes any unexpected behavior by `open_nipals` PCA or PLS models, it is recommended to enable the `verbose` flag in the `fit()` and `set_components()` methods. This will print milestone markers during the fitting procedure.

Benchmarking

In order to assess `open_nipals`' performance, we compared it to other common dimensionality reduction techniques with missing value imputation.

Since the PCA and PLS implementations of `open_nipals` are very similar, and there are more alternative implementations of PCA than of PLS, we decided to focus on PCA for most of the benchmarking discussion. The benchmark of the `open_nipals.NipalsPLS` module is briefly presented at the end of this section.

Our benchmark measures the respective runtime, peak memory consumption, and data reconstruction accuracy of the execution of the PCA model's `fit()` routine. Runtime and peak memory allocation are measured by the Python-native `time` and `tracemalloc` packages. Data reconstruction accuracy is measured as the mean squared difference between the (synthetic) input data and the data reconstructed by the model using a `model.inverse_transform(model.transform(data))` logic.

The tested dimensionality reduction and imputation techniques are:

1. `open_nipals.NipalsPCA` with its native Nelson's Single Component Projection.
2. Adding components to an existing `open_nipals.NipalsPCA` model, leveraging the `set_components()` method. This is only evaluated for adding components given a fixed dataset.
3. `sklearn.PCA` with `sklearn.SimpleImputer` for univariate imputation of sparse input matrices.
4. `sklearn.PCA` with `sklearn.MatrixImputer` for multivariate imputation of sparse input matrices.[1]
5. `sklearn.FactorAnalysis` Expectation Maximization (FA/EM) procedure, combined with `sklearn.SimpleImputer`.
6. `sklearn.FastICA` for Independent Component Analysis (ICA), combined with a `sklearn.SimpleImputer`.

Fit tolerances were set to `1e-4` for all methods, a maximum number of iterations was set to 1000 where applicable. As mentioned before, one of the upsides of `open_nipals.NipalsPCA` compared to the SVD-based `sklearn.PCA` implementation is that numerical cost and accuracy can be traded off by setting tolerance criteria. We found these settings place `open_nipals.NipalsPCA` in the middle of the pack with regard to numerical cost, justifying the setting *a posteriori*. Wherever necessary, we passed a globally initialized `NumPy` random number generator into the model constructors or `fit` methods. Bear in mind that the NIPALS algorithm is a deterministic algorithm and therefore does not require any random state initialization.

Dataset creation

A realistic synthetic dataset was created:

1. Construct an `M x M` random orthogonal matrix using the QR-decomposition, where `M` is the number of features of the dataset.
2. Construct a Gaussian `N x M` random matrix, with the number of samples `N`. Without loss of generality, decrease the variance of each of the `M` columns by a factor of 0.9 to mimic decreasing variance of the principal components.
3. Multiply the matrices built in the previous two steps.
4. Add Gaussian random noise with amplitude 0.1.
5. Standardize the resulting data matrix with an `sklearn.StandardScaler`.
6. Randomly mask 10% of the matrix entries with `np.nan`.

This procedure was repeated for any combination of number of samples and features plotted in the sections below. Five replicate data sets of each configuration of samples and features were generated. The resulting performance measures were eventually averaged over these 5 runs to make the results less susceptible to random effects.

[1] The import module is `from sklearn.impute import IterativeImputer`. This is an experimental module; therefore `from sklearn.experimental import enable_iterative_imputer` is required, too. We refer to it as `sklearn.MatrixImputer` in the following, since we find this name slightly more instructive.

Parameter scaling of numerical cost and accuracy

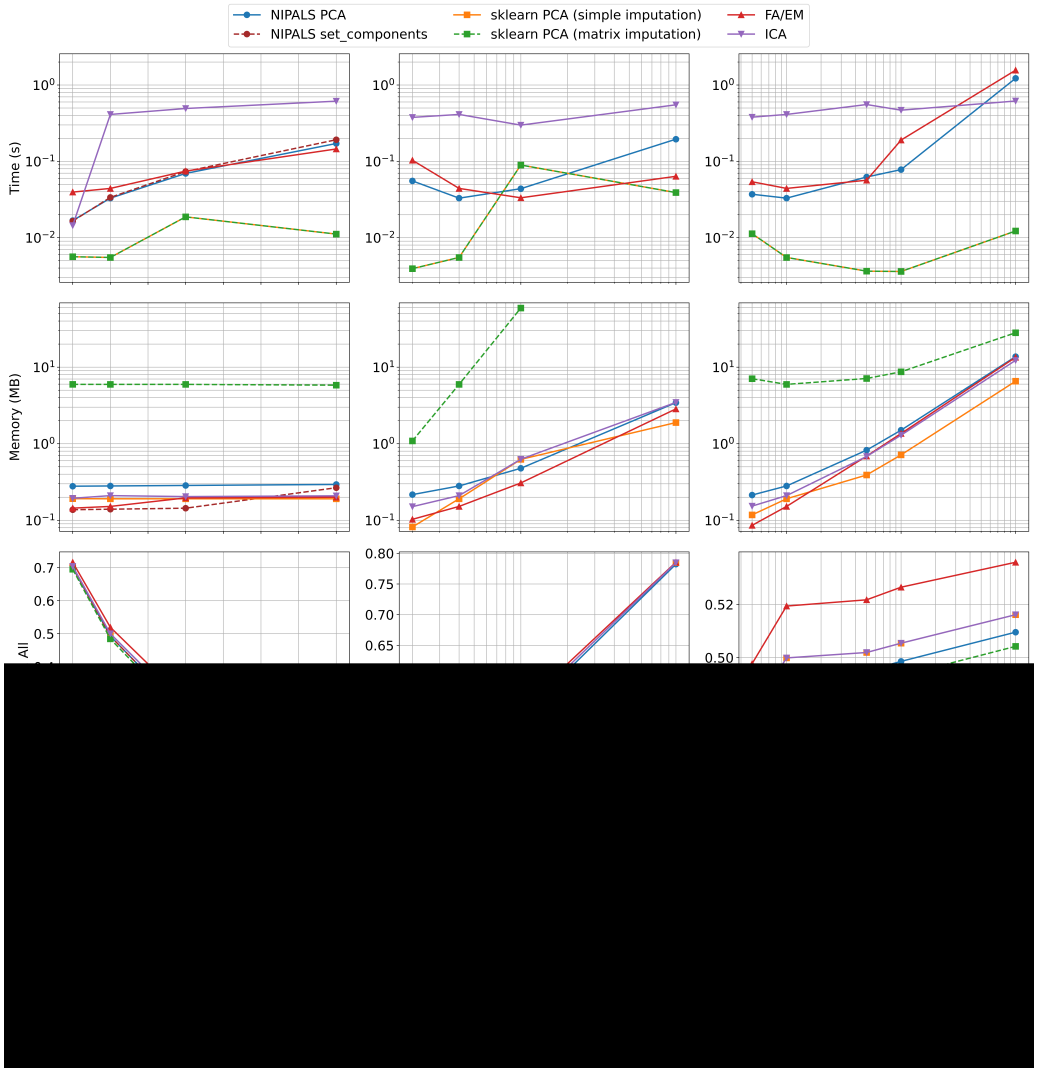

Figure 3: Computational complexity of `open_nipals.NipalsPCA` and alternatives. Figure shows runtime (upper row), peak memory consumption (second row), and accuracy in reconstructing all the input data (third row) and just the masked data (bottom row), both measured by mean squared error. Scaling of quantities evaluated with respect to `n_components` (left column), dimensionality of the dataset (number of features, middle column), and number of samples (right column).

Figure 3 compares the performance of `open_nipals.NipalsPCA`, measured by runtime, memory requirements, and accuracy, for varying `n_components`, dimensionalities, and number of samples, to alternative dimensionality reduction and imputation techniques. The data was varied around a center configuration of `n_components=4`, 100 samples, and 40 features. `FastICA` raised convergence warnings for `n_samples=100, n_features=40, n_components=4,8,16`, `n_samples=1000, n_features=40, n_components=4`, `n_samples=100, n_features=20, n_components=4`, `n_samples=100, n_features=100, n_components=4`. We found the accuracy of the fitted ICA models to be comparable to the others, and therefore decided to still use these data points. Excessive memory consumption of the experimental `MatrixImputer` made the computation of the point at `n_components=4`, 100 samples, and 1000 features numerically impossible. Therefore, this data point was omitted in the middle plot of the middle row in Figure 3.

Computational complexity and accuracy vs. number of latent variables

The left column of Figure 3 shows complexity and accuracy evaluated over a range of 2 - 16 latent variables.

The upper left plot of Figure 3 shows that the runtime cost of `open_nipals.NipalsPCA` increases with `n_components`, with ICA being by far the slowest option for `n_components>2`, and `sklearn.PCA` the fastest option at `n_components>2`. `sklearn.PCA` relies on an SVD of the centered data matrix; therefore the numerical cost (runtime and memory) of inferring any number of latent variables is equal up to small fluctuations, which can be seen from the two upper left panels of Figure 3. The two different imputer + `sklearn.PCA` combinations require precisely the same runtime at varying `n_components`, indicating that the runtime is dominated by the SVD, not the imputation. Interestingly, fitting only additional components with `set_components()` has almost no effect on runtime.

The peak memory consumption vs. `n_components` is plotted in the left plot of the middle row of Figure 3. The comparison of `sklearn.PCA` paired with either `MatrixImputer` or `SimpleImputer` immediately suggests that the memory consumption is dominated by the imputation method rather than the actual PCA model fitting. The memory consumption of `open_nipals.NipalsPCA` can be substantially reduced by adding components to an existing model with the `set_components` method.

This comes with an accuracy of `open_nipals.NipalsPCA` similar to its alternatives, only mildly surpassed by the `MatrixImputer` + `sklearn.PCA` scenario. As expected, the choice of imputation method starts to increasingly matter at higher `n_components`. Interestingly, comparing the mean squared reconstruction error for all data (third left panel of Figure 3) and only masked entries (bottom left panel of Figure 3), one immediately realizes that for `n_components>8` only `open_nipals.NipalsPCA` and matrix imputer + `sklearn.PCA` benefit from additional components fitted, while all others show similar or even worse reconstruction error at `n_components=16`.

Computational complexity and accuracy vs. number of features

The middle column of Figure 3 shows the scaling of computational complexity and accuracy with the dimensionality of the data space (`n_features` 20 - 1000). Since a new model has to be fit every time the dimensionality of the data space changes, there is no use case for `set_components()`, which was consequently not displayed in the plots.

The upper middle panel of Figure 3 shows that `open_nipals.NipalsPCA`'s model fitting has a similar runtime to FA/EM, surpassing ICA by about one order of magnitude. Its runtime in seconds vs. dimensionality curve is quite shallow (5e-2 - 3e-1), even compared to `sklearn.PCA` (5e-3 - 7e-2), which was faster than its competitors by 1-2 orders of magnitude throughout the investigated range of features.

The memory consumption in MB vs. features is plotted in the middle row's second plot of Figure 3. It shows a memory usage by `open_nipals.NipalsPCA` similar to its competitors, performing well particularly at 100 or more features. Memory usage at a very small number of features (less than 100) seems to be dominated by overhead, of which `open_nipals.NipalsPCA` requires more than traditional algorithms like `sklearn.PCA` with univariate imputation. However, as the dimensionality increases, the overhead becomes less relevant, leading to a shallow `open_nipals.NipalsPCA` memory consumption curve (2e-1 - 3.5 MB), compared to 8e-2 - 2 MB for `sklearn.PCA` with univariate imputation. The matrix imputation technique proves once more to require the most memory throughout the entire investigated range of data-space dimensions by 1 - 2 orders of magnitude, leading to the data point at 1000 features being impossible to collect.

The accuracy is evaluated against different numbers of features in the two lower middle panels of Figure 3. They prove `open_nipals.NipalsPCA` to be similarly accurate to other techniques,

with the slight exception of the matrix-imputed `sklearn.PCA` being very precise, especially for the masked values, at high computational cost. All methods lose overall reconstruction accuracy as the number of dimensions increases, while the missing data reconstruction error has a minimum at 100 features for all methods except matrix imputation + `sklearn.PCA`.

Computational complexity and accuracy vs. number of samples

The right column of Figure 3 displays computational complexity and accuracy over a broad range of dataset size (`n_samples` = 50 - 10000).

The runtime requirement scaling with `n_samples` can be found in the upper right plot of Figure 3. `open_nipals.NipalsPCA` has a similar fit time to FA/EM and is faster than ICA at all sample sizes except at the largest. Both imputation methods combined with `sklearn.PCA` are more than one order of magnitude faster.

As can be seen from the panel in the second row of the right column of Figure 3, the peak memory consumption of `open_nipals.NipalsPCA` follows a similar scaling with `n_samples` to its alternatives ICA and FA/EM. The two `sklearn.PCA` scenarios form the edge cases here. SVD-based PCA with univariate imputation consumes a bit less memory than ICA, FA/EM, and `open_nipals.NipalsPCA` at medium to large sample sizes, while SVD-based PCA with multivariate imputation consumes about one order of magnitude more memory than its competitors, although this gap closes with increasing sample size.

The numerical accuracy as a function of `n_samples` in Figure 3 (third row and bottom row) remains quite constant for both cases, with `open_nipals.NipalsPCA` being slightly more accurate than all of its competitors except matrix-imputed `sklearn.PCA`. The combination of matrix imputation + `sklearn.PCA` is again the most accurate option, followed by `open_nipals.NipalsPCA` with slightly better accuracy than the other options.

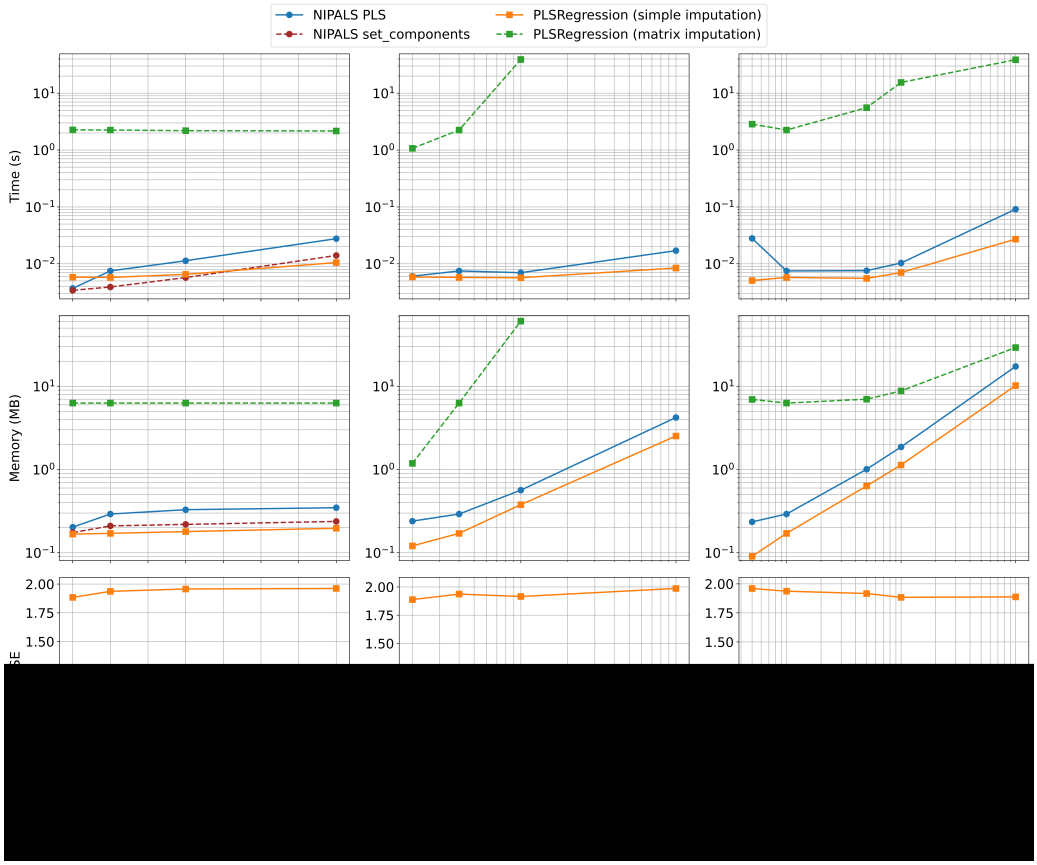

Figure 4: Computational complexity of `open_nipals.NipalsPLS` and alternatives. Figure shows runtime (upper row), peak memory consumption (second row), and mean squared prediction error (bottom row). Scaling of quantities evaluated with respect to `n_components` (left column), dimensionality of the dataset (number of features, middle column), and number of samples (right column).

We compare `open_nipals.NipalsPLS` against two alternatives: `sklearn.PLSRegression` combined with either a univariate or a matrix imputer. Synthetic data was prepared with a very similar procedure to the one elaborated above for PCA, also creating 5 replicates per data set to be averaged. A 1-dimensional y-array was created by drawing a random vector of weights and multiplying that with the (full) data vector, adding Gaussian noise with the same amplitude as for the x data. As the reconstruction error was already displayed in Figure 3, we focus on the prediction error measured by the mean squared error of the predicted y-variable, displayed in the bottom row of Figure 4.

The results in Figure 4 generally paint a similar picture to the PCA case: `open_nipals` performs similarly to its alternatives in wall-clock time and memory consumption, although being closer to the cheaper option in each case. However, the prediction accuracy of `open_nipals.NipalsPLS` clearly surpasses both its alternatives, as shown in the bottom row of Figure 4.

Benchmarking conclusion

`open_nipals` fills a practical gap for Python users in chemometrics and batch-process analytics by providing `sklearn`-style NIPALS PCA and PLS estimators with integrated missing-data handling and domain-relevant diagnostics. Benchmarks suggest that the implementation is competitive in runtime and memory use and can provide strong accuracy in low-to-moderate

latent-dimensional settings, particularly when missing data are present. The algorithmic benchmark is complemented by the possibility of trading numerical accuracy for numerical cost, and lowering memory consumption through using `set_components()` to add new latent variables to a pre-existing model.

Availability

`open_nipals` is available open-source under the BSD 3-clause license from this GitHub repository (Ochsenbein et al., 2026). We appreciate your feedback and contributions. The latest version is deployed to the Python Package Index (Schlusser, 2026b), the documentation is deployed to Read The Docs (Schlusser, 2026a). A Jupyter notebook to reproduce Figure 1, another notebook to generate Figure 2, and a notebook to run the benchmark and generate Figure 3 can be found in (Ochsenbein et al., 2026) under the `paper` branch. A permanent DOI to the version of the code referred to by this paper can be found in (Schlusser, 2026c).

Acknowledgements

We acknowledge support by Johnson & Johnson Innovative Medicine. In particular, we would like to express our gratitude to Samuel Tung and Tyler Roussos of the Open Source Working Group (OSWG) within J&J, who helped drive the publication process of `open_nipals`. Moreover, we acknowledge Calvin Ristad's contributions to v2.0 of the `open_nipals` code.

References

Brereton, R. G. (2016). Hotelling's t squared distribution, its relationship to the f distribution and its use in multivariate space. *Journal of Chemometrics*, *30*(1), 18–21. https://doi.org/10.1002/cem.2763

Eriksson, L., Johansson, E., Kettaneh-Wold, N., & Wold, S. (1999). *Introduction to multi- and megavariate data analysis using projection methods (PCA and PLS)* (p. 468). Umetrics.

Garcia Munoz, S., Codgers, J. A., & Johnson, B. J. (2019). *pyphi - A Python package for multivariate analysis* (Version 4.1). https://github.com/salvadorgarciamunoz/pyphi

Geladi, P., & Kowalski, B. R. (1986). Partial least-squares regression: A tutorial. *Analytica Chimica Acta*, *185*, 1–17. https://doi.org/10.1016/0003-2670(86)80028-9

Harris, C. R., Millman, K. J., Walt, S. J. van der, Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., Kerkwijk, M. H. van, Brett, M., Haldane, A., Río, J. F. del, Wiebe, M., Peterson, P., … Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, *585*(7825), 357–362. https://doi.org/10.1038/s41586-020-2649-2

Hotelling, H. (1931). The Generalization of Student's Ratio. *The Annals of Mathematical Statistics*, *2*(3), 360–378. https://doi.org/10.1214/aoms/1177732979

Nelson, P. R. C., Taylor, P. A., & MacGregor, J. F. (1996). Missing data methods in PCA and PLS: Score calculations with incomplete observations. *Chemometrics and Intelligent Laboratory Systems*, *35*(1), 45–65. https://doi.org/10.1016/S0169-7439(96)00007-X

Ochsenbein, D. R., Schlusser, N., & Wall, R. (2026). *open_nipals* (Version 2.0.2). https://github.com/johnsonandjohnson/open_nipals

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, *12*, 2825–2830.

Schlusser, N. (2026a). *open_nipals documentation* (Version 2.0). https://open-nipals.readthedocs.io/en/latest/index.html

Schlusser, N. (2026b). *open_nipals PyPI* (Version 2.0.2). https://pypi.org/project/open-nipals/

Schlusser, N. (2026c). *open_nipals zenodo archive* (Version 2.0.2). https://doi.org/10.5281/zenodo.20560917

The pandas development team. (2020). *Pandas-dev/pandas: pandas* (latest). Zenodo. https://doi.org/10.5281/zenodo.3509134