**ABSTRACT**

Dimensionality reduction (DR) is a widely used technique for visualization. Nowadays, many of these visualizations are developed for the web, most commonly using JavaScript as the underlying programming language. So far, only few DR methods have a JavaScript implementation though, necessitating developers to write wrappers around implementations in other languages. In addition, those DR methods that exist in JavaScript libraries, such as PCA, t-SNE, and UMAP, do not offer consistent programming interfaces, hampering the quick integration of different methods. Toward a coherent and comprehensive DR programming framework, we developed an open source JavaScript library named DRUIDJS. Our library contains implementations of ten different DR algorithms, as well as the required linear algebra techniques, tools, and utilities.

**Index Terms:** Software and its engineering—Software notations and tools—Software libraries and repositories; Human-centered computing—Visualization—Visualization systems and tools—Visualization toolkits;

**1 INTRODUCTION**

Dimensionality reduction (DR) is a technique used to reduce the total amount of dimensions in a given dataset in order to visualize the dataset or reduce the effects of the curse of dimensionality in machine learning pipelines. DR methods are important tools to project high-dimensional datasets into two or three dimensions in order to visualize them. Python, R, and Matlab provide extensive libraries for DR methods, while no extensive library exists for JavaScript yet. With the increasing popularity of web applications using powerful visualization tools such as D3 [5] and Vega [34], also the use of DR techniques has become popular. At the moment, a dedicated library for DR methods is still missing though, which is particularly problematic as different types of data require different DR algorithms [28]. It is not possible to use the most appropriate algorithm in every case as many DR methods have no JavaScript implementation.

At the moment, DR methods often require either a server-client structure or a precomputed embedding which can be loaded into the browser. The latter approach is easy to implement, but no interaction with the DR method is possible. It thus impedes the user from loading their own data or to interactively change DR methods and their parameterizations. Tools based on a server-client structure have privacy issues as the data needs to be sent to a server, preventing users from using those tools on sensitive data. To address these concerns, users can create custom implementations, but this is difficult, time-consuming, and error-prone. Additionally, many DR techniques are complex and need tools for unsupervised learning, linear algebra, stochastic, etc. which makes efficient implementation a difficult endeavour.

To fill this gap, we contribute DRUIDJS, a JavaScript library for DR methods, targeted at researchers and developers. The main goal is to support them by providing access to the most frequently used DR techniques. Based on our experience working with DR for many years and the frequency of how they are used in the VIS literature at the moment [33], we equipped DRUIDJS with an implementation of ten DR algorithms — some of which have no existing JavaScript implementations so far. The programming interface is designed based on best practices used in other libraries such as Scikit-learn [30] and D3 [5]. The DRUIDJS library is dependency-free and therefore easily integrated into any project, by properly bundling all necessary tools required to use DR techniques. These necessary tools include matrix multiplications, LU-, QR-, and eigen-decompositions. In the current version, we focused on code readability and ease-of-use from a programmer’s perspective.

**2 RELATED WORK**

The main component of our framework is a set of dimensionality reduction methods which are bundled into a JavaScript library. There has been a plethora of work on DR methods and their usage. Recent surveys [22, 28, 33] characterized the most important DR algorithms and how they are used to interactively visualize high-dimensional data. Generally, the literature distinguished between two main groups of DR methods: linear and non-linear [39]. Linear techniques, such as PCA [29] allow for easy interpretation of the resulting space. In fact, this property makes PCA one of the most commonly used DR methods [35]. The results of non-linear methods, such as t-SNE [38] and UMAP [24], can uncover more complex high-dimensional structures. To do so, they most commonly try to preserve the neighborhoods as well as possible. However, projections might look very different with different starting conditions and might be harder to interpret by humans [40].

To the best of our knowledge, there is currently no JavaScript framework that implements more than two DR algorithm and works in the browser. For some implementations different libraries exist. MachineLearn.js [17] implements the PCA algorithm, which does not allow users to set the desired dimensionality and requires additional steps to create the reduced dataset. ISNEJS [16] implements the t-SNE algorithm and UMAP.js [19] implements the UMAP algorithm. Both use different programming interfaces (Ex. 7) and are therefore complicated to use together. HDSP [15] and mlda [18] are TypeScript libraries available for NodeJS only. Like MachineLearn.js, the older pca-js [14] uses the SVD [11] algorithm from numeric.js [23]. The library mdsjs [6] implements PCA and LandmarkMDS [1], but has no documentation and we thus do not include it in our comparative evaluation.

The alternative to JavaScript libraries are libraries in other programming languages like Scikit-learn [30] for Python, dimRed [20] for R, or drtoolbox [37] for Matlab. Our library does the projection directly in the browser and does not require a server to do the computations in a different language.

**3 DRUID**

We now explain the features of DRUIDJS. We start by showing the basic functions, before we depict some more advanced features.

---

*e-mail: rene.cutura@tuwien.ac.at*

†e-mail: christoph.kralj@univie.ac.at

‡e-mail: michael.sedlmair@visus.uni-stuttgart.de

1https://github.com/saehm/DruidJS
We use the Palmer Penguin [12] dataset for illustration. The dataset consists of 4 dimensions and 342 different specimens of penguins of the species Adelie, Gentoo, and Chinstrap, living on one of the islands Torgersen, Biscoe, or Dream of the Antarctica. The dataset contains features for the length and the depth of the culmen/beak, the flipper lengths, and the body mass. We normalized the values of each dimension and filtered data rows with missing values. Fig. 1 shows examples of projections with DRUIDJS.

Dimensionality Reduction We implemented the linear DR techniques PCA [29], LDA [4], and FASTMAP [9], and the non-linear DR techniques Isomap [36], LLE [32], LTSA [41], MDS [21], TriMap [2], t-SNE [38], and UMAP [24]. All algorithms have the same interface, therefore we use only Isomap, UMAP, and t-SNE to showcase our design decisions regarding the programming interface (referred to as druid.DR). The values of the Palmer Penguin dataset are stored in the 2D array penguins. One line of code (see Ex. 1) is enough to project the data with Isomap in this case (Fig. 1a).

```
let projection = new druid.ISOMAP(penguins).transform();
```

Example 1: A new DR object isomap gets created with new druid.ISOMAP. It takes as argument the data. The data can be a 2D array or an object of the druid.Matrix class. The method transform does the actual projection, and returns the result in the input type.

```
1 let isomap = new druid.ISOMAP(penguins)
2 isomap.transform() // first run with defaults
3
4 umap.transform() // second run with new parameters
```

Example 2: Changing the parameter k of isomap to 50, followed by transform projecting the already defined Palmer Penguin dataset in Ex. 1. Fig. 1b shows the result.

Transform There are different ways to create a projection. The simplest one is to just call the function transform or the async variant transform_async on the druid.DR object. We use the method name “transform” to stick with the well established naming convention used in Scikit-learn, and as it describes best what happens to the input data: it is transformed into a space of different dimensionality.

Parameterization Parameters influence the results of a DR algorithm [8]. Therefore, it is often necessary to rerun a projection, because the optimization process gets stuck in a local minimum and the result is of bad quality. To tweak the parameters, each of them can be changed by using the method parameter.

Fig. 1a (the result of Ex. 1) shows superimposed Gentoo and Chinstraps. One reason for that could be a bad parameter value for Isomap’s k: the number of neighbors per point taken into account during the projection process of Isomap. The parameters depend on the respective DR method. In our Palmer Penguin example a different value for k changes the neighborhood connections and results in better visual separability of the Gentoo and Chinstraps in the final plot (Ex. 2 & Fig. 1b).

For users it is often unclear how to properly select these parameters. We thus offer reasonable defaults, but tweaking parameters might still often be necessary.

```
1 let isomap = new druid.ISOMAP(penguins)
2 isomap.transform();
3 projection = isomap.transform();
```

Constructor The code used in Ex. 3 can be shortened by using the constructor of the respective DR class. The constructor takes as arguments the input data (see Ex. 1), then DR-dependent parameters, and then the hyper-parameters. The input data can be either a druid.Matrix object or a 2D array which gets internally converted to a druid.Matrix object.

```
1 new druid.ISOMAP(penguins, 1, 2, 2, druid.manhattan, → seed).transform();
```

Example 4: Using the constructor shortens the code (from 5 lines to 1 line), but outputs the same as Ex. 3.
Generator Some DR methods like t-SNE (see Fig. 2) are computationally complex and require more time. To integrate such methods into an interactive visualization tool, the intermediate results can be plotted. Such intermediate results also provide insight into the DR algorithm and allow for progressive approaches [27]. We use a generator function to provide the user access to the intermediate results of the projection after each optimization step (Ex. 5).

Example 5: A generator yielding the preliminary result’s until the optimization process of the DR algorithm stops.

Data Structure DRUID’s matrix class druid.Matrix stores the data values. The function from takes a 2D array and checks for proper shape of the array, and stores it efficiently in the memory as JavaScript typed arrays. The type of the return value of a DR algorithm depends on the input type. Internally, DRUID converts a 2D array to an object of the druid.Matrix class. When more than one projection of a dataset is planned, using a druid.Matrix object avoids redoing this step each time. To use a druid.Matrix object, for instance, with D3, calling the function to2dArray converts the object to a JavaScript 2D array.

Initialization Some DR algorithms comprise computational steps before the actual projection. For example, MDS and t-SNE require a distance matrix before projection. Reusing such previous computations reduces the runtime. These initialization steps are DR technique-specific and require a lookup by the user, but can provide precomputations to the DR algorithm (see Ex. 6).

Example 6: The function init provides the precomputed distance matrix Δ to the MDS- and tSNE object.

4 Evaluation
We evaluated DRUID in three steps. First, we did an individual runtime analysis of DRUID under different scales of data. Second, we compared these runtimes to common Python and some existing JavaScript DR implementations. Finally, we conducted a case study comparing code readability and runtime of DRUID to existing JavaScript DR libraries.

4.1 Runtime analysis
To analyze the runtime of DRUID, we generated random data for each pair of datasize N ∈ \{16 + 2^i | i ≥ 3\} and dimensionality D ∈ \{2 · 5^i | d ∈ \{1, ..., 8\}\}. Then, we projected each dataset five times with six DR methods, measured the runtime for each, and averaged the results. We picked the six most common ones due to page limit; the full set of ten DR methods and more detailed data can be found in the supplemental materials (also for the results in Sec. 4.2). We stopped further projections for a specific D if a projection with length N needed more than ten seconds — a common time limit for keeping a user’s attention [7, 26]. The computations were run locally on a notebook with an Intel Core i7-8705G processor with 16GB memory using the Chrome browser 83.0.4103.116.

Looking at the results (see Fig. 2), we observe that most techniques are strongly affected by the number of points. PCA is less affected by N, but much more by a high D, while t-SNE is less affected by D, but much more by N. Besides PCA and t-SNE, all DRUID methods have a reasonable runtime (under 1 second) for N < 500. For N > 500, visualizing intermediate results with the generator method can be a viable option.

We created an online demo\(^2\), so that users/readers can get a qualitative “feeling” on the duration of these runtimes. The demo allows to select an N × D dataset and then computes the respective DRUID projection on the fly.

4.2 Runtime comparison to other libraries
We now compare these runtimes to other frequently used implementations. In Python, we use Scikit-learn [30] and UMAP-learn [25] (Fig. 3 ● sklearn\(^*\)). In terms of JavaScript, we use MachineLearn.js [17], tSNEJS [16], and UMAP-js [19] (Fig. 3 ● js).

The results (Fig. 3) show that DRUID performs better than Scikit-learn for datasets with small N and small D for all methods. At some point Scikit-learn starts to be faster than DRUID. This result is not surprising, as Scikit-learn has been heavily optimized for runtime, while we have not engaged in such optimizations yet. Scikit-learn implemented an iterative version of MDS, which is slower than the direct version of DRUID.

\(^2\)https://renecutura.eu/druid_demo/
4.3 Case Study

We use a common use case — the projection of a dataset to two dimensions for visualization purposes using different DR methods — to illustrate the code readability of DRUIDJ$. We chose the well known IRIS dataset [10], consisting of 150 points with 4 dimensions for our example. We compare three existing JavaScript libraries with DRUIDJ$ based on the runtime and the number of lines of code. To compare the runtime we executed the scripts Ex. 7 and Ex. 8 1000 times with equal parameterization. We performed 100 warm-up runs before taking measurements to ensure similar JIT compiler states and therefore improve the comparability. One run with DRUIDJ$ (Ex. 8) took on average $922 \pm 109.82ms$, and with existing implementations (Ex. 7) on average $1051 \pm 70.64ms$. DRUIDJ$ is, thus, roughly 10% faster and has just 5 (Ex. 8) lines of code instead of 21 (Ex. 7). In addition, tSNEjs requires to do the iterations manually (Ex. 7, lines 8–10) which requires good knowledge about the method to choose the right iteration count. MachineLearn.js’s PCA computes only the principal components and requires the user to create the projection manually, worsening the code readability (Ex. 7, lines 15–22). The UMAP-js implementation has a state-of-the-art interface and requires only one line. With DRUIDJ$ every DR method requires only one line and is likely easier to use (Ex. 8).

5 LIMITATIONS & Future Work

During the creation of DRUIDJ$, our focus was the support of inexperienced users by offering reasonable default parameterizations. This brought us some feature limitations we plan to fix in future releases of DRUIDJ$. One of these limitations regards the addition of more points, so called “out-of-sample extensions”. Some DR algorithms allow the addition of more points to an existing projection. We do not support this so far as it is not possible for all algorithms, and we wanted to keep the interface consistent over all algorithms. We implemented a few important DR techniques, but others such as SVD [11] are still missing. Of course, there is also still plenty of room for implementations that scale JavaScript DR methods to larger datasets, e.g., with GPU acceleration like tfjs-tsne [31].

CONCLUSION

In this paper, we present the JavaScript library DRUIDJ$, which allows to consistently use different DR methods directly in the browser and integrate them into interactive tools. DRUIDJ$ is based on a comprehensive programming interface using tried-and-tested coding conventions. As most VIS-tools are online nowadays [3, 13], we hope that DRUIDJ$ will help to lower the entry barrier for developers who want to use DR, and will support applications where data privacy is of concern.

ACKNOWLEDGMENTS

This work was supported by the FFG ICT of the Future program via the ViSciPub project (no. 867378).