Feature-Based Time-Series Analysis in R using the Theft Ecosystem

by Trent Henderson and Ben D. Fulcher

Abstract Time series are measured and analyzed across the sciences. One method of quantifying the structure of time series is by calculating a set of summary statistics or 'features', and then representing a time series in terms of its properties as a feature vector. The resulting feature space is interpretable and informative, and enables conventional statistical learning approaches, including clustering, regression, and classification, to be applied to time-series datasets. Many open-source software packages for computing sets of time-series features exist across multiple programming languages, including 'catch22' (22 features: Matlab, R, Python, Julia), 'feasts' (43 features: R), 'tsfeatures' (62 features: R), 'Kats' (40 features: Python), 'tsfresh' (783 features: Python), and 'TSFEL' (156 features: Python). However, there are several issues: (i) a singular access point to these packages is not currently available; (ii) to access all feature sets, users must be fluent in multiple languages; and (iii) these feature-extraction packages lack extensive accompanying methodological pipelines for performing feature-based time-series analysis, such as applications to time-series classification. Here we introduce a solution to these issues in the form of two complementary statistical software packages for R called 'theft': Tools for Handling Extraction of Features from Time series and 'theftdlc': theft 'downloadable content'. 'theft' is a unified and extendable framework for computing features from the six open-source time-series feature sets listed above as well as custom user-specified features. 'theftdlc' is an extension package to 'theft' which includes a suite of functions for processing and interpreting the performance of extracted features, including extensive data-visualization templates, low-dimensional projections, and time-series classification. With an increasing volume and complexity of large time-series datasets in the sciences and industry, 'theft' and 'theftdlc' provide a standardized framework for comprehensively quantifying and interpreting informative structure in time series.

1 Introduction

Taking repeated measurements of some quantity through time, forming a time series, is common across the sciences and industry. The types of time series commonly analyzed are diverse, ranging from time-varying signals of an electroencephalogram (West et al., 1999), CO_2 concentration in the atmosphere (Kodra et al., 2011), light-curves from distant stars (Barbara et al., 2022), and the daily number of clicks on a webpage (Kao et al., 2021). We can ask many different questions about such data, for example: (i) "can we distinguish the dynamics of brain disorders from neurotypical brain function?"; (ii) "can we classify different geospatial regions based on their temporal CO_2 concentration?"; or (iii) "can we classify new stars based on their light curves?". One approach to answering such questions is to capture properties of each time series and use that information to train a classification algorithm. This can be achieved by extracting from each time series a set of interpretable summary statistics or 'features'. Using this procedure, a collection of univariate time series can be represented as a time series $\times$ feature matrix which can be used as the basis for a range of conventional statistical learning procedures (Fulcher and Jones, 2017; Fulcher, 2018).

The range of time-series analysis methods that can be used to define time-series features is vast, including properties of the distribution, autocorrelation function, stationarity, entropy, methods from the physics nonlinear time-series analysis literature (Fulcher et al., 2013). Because features are real-valued scalar outputs of a mathematical operation, and are often tightly linked to underlying theory (e.g., Fourier analysis or information theory), they can yield interpretable understanding of patterns in time series and the processes that produce them—information that can guide further investigation. The first work to organize these methods from across the interdisciplinary literature encoded thousands of diverse time-series analysis methods as features and compared their behavior on a wide range of time series (Fulcher et al., 2013). The resulting interdisciplinary library of thousands of time-series features has enabled new ways of doing time-series analysis, including the ability to discover high-performing methods for a given problem in a systematic, data-driven way through large-scale comparison (overcoming the subjective and time-consuming task of selecting methods manually) (Fulcher and Jones, 2014). This approach has been termed 'highly comparative time-series analysis' and has been implemented in the Matlab software `hctsa`, which computes > 7700 time-series features (Fulcher and Jones, 2017). The approach of automated discovery provided by `hctsa` has been applied successfully to many scientific problems, such as classifying zebra finch motifs across different social contexts (Paul et al., 2021), classifying cord pH from fetal heart-rate dynamics (Fulcher et al., 2012), and classifying changes in cortical dynamics from manipulating the firing of excitatory and inhibitory neurons (Markicevic et al., 2020). While `hctsa` is comprehensive in its coverage of time-series analysis methods, calculating all of its features on a given dataset is computationally expensive and it requires access to the proprietary Matlab software, limiting its broader use.

The past decade has seen the development of multiple software libraries that implement different sets of time-series features across a range of open-source programming languages. Here, we focus on the following six libraries:

- `catch22` (C, Matlab, R, Python, Julia) computes a representative subset of 22 features from `hctsa` (Lubba et al., 2019). The > 7700 features in `hctsa` were applied to 93 time-series classification tasks to retain the smallest number of features that maintained high performance on these tasks while also being minimally redundant with each other, yielding the `catch22` set. `catch22` was coded in C for computational efficiency, with wrappers for Matlab, and packages for: R, as **Rcatch22** (Henderson, 2021); Julia, as `Catch22.jl` (Harris, 2021); and Python, as `pycatch22`. `catch22` is also commonly extended to include mean and variance to form "`catch24`" in order to achieve competitive performance on tasks where the dataset has not been standardized (Henderson et al., 2023).

- **tsfeatures** (R) is the most prominent package for computing time-series features in R (Hyndman et al., 2020). The 62 features in **tsfeatures** include techniques commonly used by econometricians and forecasters, such as crossing points, seasonal and trend decomposition using Loess (Cleveland et al., 1990), autoregressive conditional heteroscedasticity (ARCH) models, unit-root tests, and sliding windows. **tsfeatures** also includes sixteen features from `hctsa` that were previously used to organize tens of thousands of time series in the *CompEngine* time-series database (Fulcher et al., 2020).

- **feasts** (R) shares a subset of the same features as **tsfeatures**, computing a total of 43 features (O'Hara-Wild et al., 2021). However, the scope of **feasts** as a software package is larger: it is a vehicle to incorporate time-series features into the software ecosystem known as the **tidyverts**[1]—a collection of packages for time series that follow tidy data principles (Wickham, 2014). This ensures alignment with the broader and popular **tidyverse** collection of packages for data wrangling, summarization, and statistical graphics (Wickham et al., 2019). **feasts** also includes functions for producing graphics, but these are largely focused on exploring quantities of interest in econometrics, such as autocorrelation, seasonality, and Seasonal and Trend decomposition using Loess (STL).

- `tsfresh` (Python) includes 783 features that measure properties of the autocorrelation function, entropy, quantiles, fast Fourier transforms, and distributional characteristics (Christ et al., 2018). `tsfresh` also includes a built-in feature filtering procedure, FeatuRe Extraction based on Scalable Hypothesis tests (FRESH), that uses a hypothesis-testing process to control the percentage of irrelevant extracted features (Christ et al., 2017). `tsfresh` has been used widely to solve time-series problems, such as anomaly

[1] `https://tidyverts.org`

detection in Internet-of-Things streaming data (Yang et al., 2021) and sensor-fault classification (Liu et al., 2020). `tsfresh` is also commonly accessed through the `FreshPRINCE` functionality within the popular `sktime` Python library (Löning et al., 2019).

- `TSFEL` (Python) contains 156 features that measure properties associated with distributional characteristics, the autocorrelation function, spectral quantities, and wavelets (Barandas et al., 2020). `TSFEL` was initially designed to support feature extraction of inertial data—such as data produced by human wearables—for the purpose of activity detection and rehabilitation.
- `Kats` (Python), developed by Facebook Research, contains a broad range of time-series functionality, including operations for forecasting, outlier and property detection, and feature calculation (Jiang et al., 2022). The feature-calculation module of `Kats` is called **tsfeatures** and includes 40 features (30 of which mirror R's **tsfeatures** package). `Kats` includes features associated with crossing points, STL decomposition, sliding windows, autocorrelation and partial autocorrelation, and Holt–Winters methods for detecting linear trends.

The six feature sets vary over several orders of magnitude in their computation time, and exhibit large differences in both within-set feature redundancy—how correlated features are within a given set—and between-set feature redundancy—how correlated, on average, features are between different pairwise comparisons of sets (Henderson and Fulcher, 2021). While each set contains a range of features that could be used to tackle time-series analysis problems, there are currently no guidelines for selecting an appropriate feature set for a given problem, nor methods for combining the different strengths of all sets. Performance on a given time-series analysis task depends on the choice of the features that are used to represent the time series, highlighting the importance of being able to easily compute many different features from across different feature sets. Furthermore, following feature extraction, there exists no set of visualization and analysis templates for common feature-based problem classes, such as feature-based time-series classification—like the tools provided in `hctsa` (Fulcher and Jones, 2017). Here we present a solution for these challenges in the form of two connected open-source packages for R called **theft**: Tools for Handling Extraction of Features from Time series (which unifies the six disparate feature sets and provides a consistent interface for general feature extraction) (Henderson, 2025b); and **theftdlc**: theft downloadable content (which handles the subsequent processing, analysis, and visualization of time-series features) (Henderson, 2025c). Together, we refer to these packages as the **theft** 'ecosystem'.

2 The theft ecosystem for R

theft unifies the six free and open-source feature sets described in Section 1, thus overcoming barriers in using diverse feature sets developed in different software environments and the differences in their syntax and input-output structures. The package also enables users to manually specify the functions for any number of features they wish to extract on top of the six pre-existing sets. **theftdlc** builds upon the foundation of **theft** by providing an extensive analytical pipeline as well as statistical data visualization templates for understanding feature behavior and performance. To our knowledge, such pipelines and templates do not currently exist in the free and open-source setting, making **theftdlc** a useful tool for both computing and understanding features. While there is some software support for computing features in a consistent setting (such as in `tsflex` (Van Der Donckt et al., 2022), which also provides sliding window extraction capability), such software is limited to only specifying the functional form of individual time-series features rather than automatically accessing features contained in existing feature sets. We partition the analytical capabilities of **theft** and **theftdlc** into two separate packages for two reasons: (i) it reduces dependencies if users wish only to extract features and conduct analysis themselves; and (ii) having a separate analysis package means additional functionality can be continuously added without exhausting R package dependency limits.

An overview of the broad functionality of the **theft** ecosystem is presented in Figure 1. The workflow begins in **theft** with automated installation of Python libraries (if the three Python-based feature sets are required) through the function `install_python_pkgs` which takes the name of the virtual environment to create (`venv`) and the path to the Python interpreter to use (`python`) as arguments (Fig.1A). The Python environment containing the installed software is then instantiated within the R session using `init_theft` (Fig.1B). Time-series data (Fig.1C) is loaded into the environment and converted to a `tsibble::tbl_ts` data structure if required (Fig.1D) (Wang et al., 2020). Time-series features are then extracted in **theft** using the desired pre-existing sets or user-supplied features (Fig.1E). The workflow transitions to **theftdlc** where the user can pass the extracted features into a range of statistical and visualization functions to derive interpretable understanding of the patterns in their dataset (Fig.1F–M). A variety of plot types are readily available, including heatmaps of the time-series $\times$ feature matrix (Fig.1F) and feature $\times$ feature matrix (Fig.1G), and violin plots of feature distributions (Fig.1H). Basic feature selection functionality is available through the `shrink` function (Fig.1I) which implements penalized maximum likelihood generalized linear models using a backend to the R package **glmnet**. Low-dimensional projection functionality is provided by the `project` function (Fig.1J). Time-series classification operations are accessible via the `classify` function (Fig.1K). Distributional summaries of time-series feature and time-series classification values are available through the `interval` function (Fig.1L). Finally, basic cluster analysis is possible through the `cluster` function (Fig.1M). Importantly, **theft** and **theftdlc** use R's `S3` object-oriented programming system, meaning classes and their methods are defined to ensure easy usage with R generic functions, such as `plot`. Classes are defined for feature-calculation objects (Fig.1E; `feature_calculations`), low dimensional projection objects (Fig.1I; `feature_projection`), interval calculation objects (Fig.1K; `interval_calculations`), and cluster objects (Fig.1L; `feature_clusters`). The individual functions of both **theft** and **theftdlc** are discussed in detail in the following sections.

In this paper, we demonstrate how the **theft** ecosystem can be used to tackle a time-series classification problem, using the Bonn University electroencephalogram (EEG) dataset as a case study (Andrzejak et al., 2001). The dataset contains 500 time series, each of length $T = 4097$, with 100 time series each from five labeled classes: (i) awake with eyes open (labeled `eyesOpen`); (ii) awake with eyes closed (`eyesClosed`); (iii) epileptogenic zone (`epileptogenic`); (iv) hippocampal formation of the opposite hemisphere of the brain (`hippocampus`); and (v) seizure activity (`seizure`). Note that classes (i) and (ii) are from healthy volunteers, while classes (iii), (iv), and (v) are from a presurgical diagnosis archive. The time series are comprised of EEG segments 23.6 seconds in duration that were cut out of continuous multichannel recordings, converted from analog to digital using 12-bit conversion, and then written onto a computer at a sampling rate of 173.614Hz. Further trimming of start and end discontinuities from the original 4396 samples was then performed, resulting in a final time series of length T = 4097 samples. This dataset was chosen as a demonstrative example because it has been widely studied as a time-series classification problem, and prior studies have focused on properties of the dynamics that accurately distinguish the classes—which is well-suited to the feature-based approach. For example, prior analysis (using `hctsa`) revealed that seizure recordings are characterized most notably by higher variance, as well as lower entropy, lower long-range scaling exponents, and many other differences (Fulcher et al., 2013). We can easily read the data file in from its zipper format online and convert to a `tsibble` ready for use:

2.1 System requirements

In order to access the features from `tsfresh`, `TSFEL`, and `Kats` in **theft**, Python >=3.10 is required (Python 3.10 is recommended). To use all other functionality across both **theft** and **theftdlc**, only R >=3.5.0 is required.

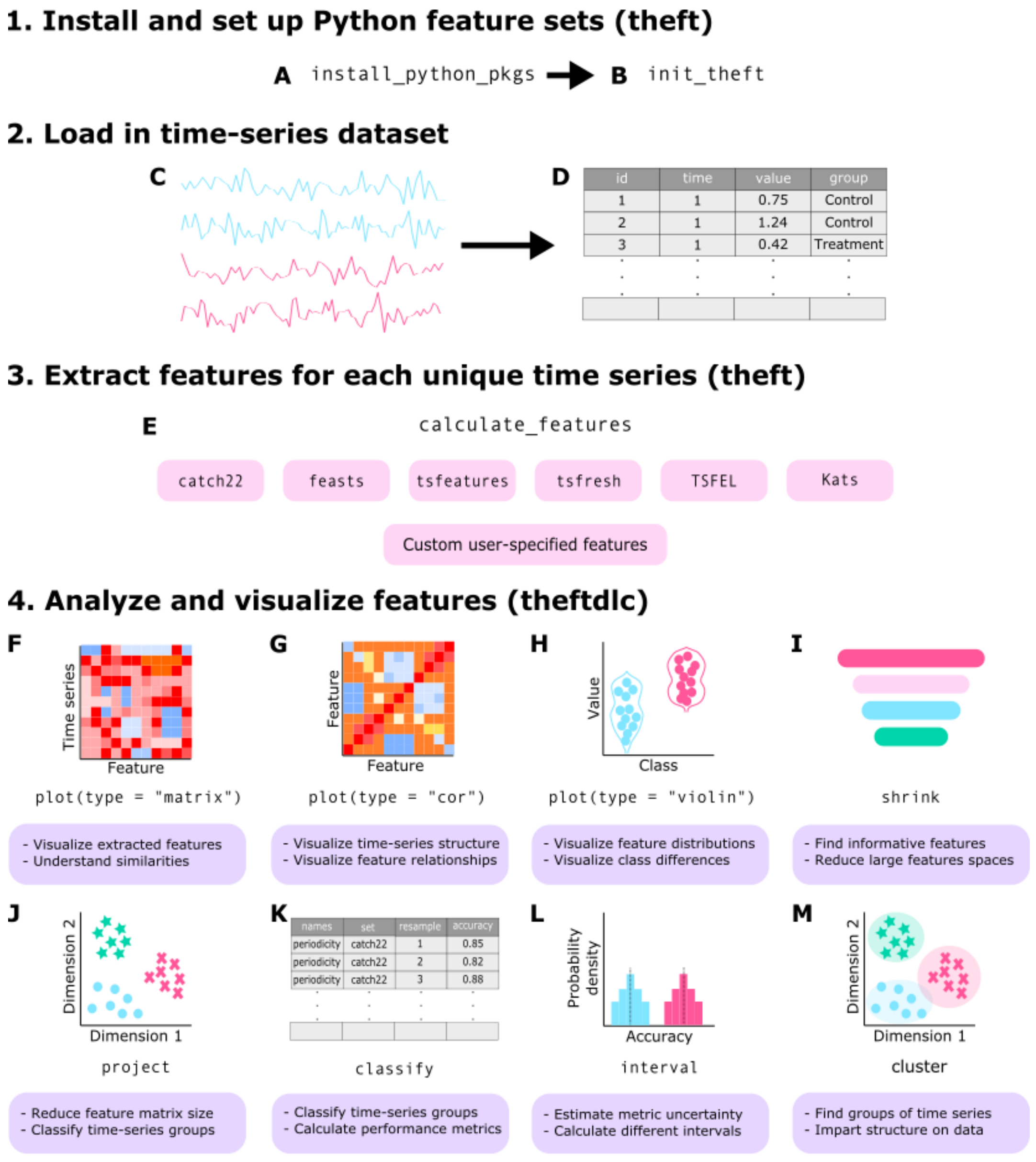
1. Install and set up Python feature sets (theft)
A install_python_pkgs
B init_theft
2. Load in time-series dataset
C
D
id time value group
1 1 0.75 Control
2 1 1.24 Control
3 1 0.42 Treatment
3. Extract features for each unique time series (theft)
E calculate_features
catch22
feasts
tsfeatures
tsfresh
TSFEL
Kats
Custom user-specified features
4. Analyze and visualize features (theftdlc)
F
Time series
Feature
plot(type = "matrix")
- Visualize extracted features
- Understand similarities
G
Feature
Feature
plot(type = "cor")
- Visualize time-series structure
- Visualize feature relationships
H
Value
Class
plot(type = "violin")
- Visualize feature distributions
- Visualize class differences
I
shrink
- Find informative features
- Reduce large features spaces
J
Dimension 2
Dimension 1
project
- Reduce feature matrix size
- Classify time-series groups
K
names set resample accuracy
periodicity catch22 1 0.85
periodicity catch22 2 0.82
periodicity catch22 3 0.88
classify
- Classify time-series groups
- Calculate performance metrics
L
Probability density
Accuracy
interval
- Estimate metric uncertainty
- Calculate different intervals
M
Dimension 2
Dimension 1
cluster
- Find groups of time series
- Impart structure on data

Figure 1: theft and theftdlc together implement a workflow for extracting features from univariate time series and processing and analyzing the results. First, the user can install the relevant Python libraries (A) within R and point it to the correctly installed versions (B). Next, a time-series dataset (C) is converted into a tsibble (D) with key variables, a time index variable, and a measured variable. One or more feature sets or custom user-supplied features are then computed on the dataset (E). A range of statistical analysis and data visualization functionality is available in theftdlc on the resulting feature data, including: (F) normalized time series x feature matrix visualization; (G) normalized feature x feature correlation matrix visualization; (H) violin plots of feature distributions (including by group/class where applicable); (I) basic feature selection using penalized maximum likelihood generalized linear models; (J) low-dimensional projections of the feature space; (K) time-series classification procedures (a common application of feature-based time-series analysis); (L) evaluating the uncertainty intervals of the resulting performance metrics through a range of distributional summary methods; and uncovering hidden structure in time-series data through cluster analysis in the feature space.

2.2 Installing Python libraries

Prior to calculating features, the requisite Python feature sets need to be installed. The `install_python_pkgs` function in **theft** handles this operation entirely within R—all that needs to be supplied is `venv` (a string specifying the name of the new virtual environment to create) and `python` (a string specifying the filepath to the Python interpreter to use). Note that the filepath to the Python interpreter will differ depending on the user's operating system and will require correct specification. For example, on Windows it might be a path similar to `"C:/Users/YourName/AppData/Local/Programs/Python/Python310/python.exe"`, whereas on Linux or MacOS (which was used for the present work) it could be a path similar to `"/usr/local/bin/python3.10"` or `"/usr/bin/python3"`.

An example call which installs the Python libraries into a new virtual environment and initiates the virtual environment within R is shown in the code below. Note that `init_theft` only needs to be run once per session.

```
install_python_pkgs(venv = "theft-eco-py", python = "/usr/local/bin/python3.10")
init_theft("theft-eco-py")
```

2.3 Extracting features

In feature-based time-series analysis, each univariate time series in a dataset is represented as a feature vector, such that the dataset can be represented as a time series × feature data matrix. Any single feature set, or combination of multiple feature sets, can be computed for a given time-series dataset with the **theft** function `calculate_features`. `calculate_features` takes a `tbl_ts` as input, using the data structure defined by the **tsibble** package for R (Wang et al., 2020). This ensures consistency with the broader `tidyverts` collection of R packages for conducting time-series analysis. A `tbl_ts` is a temporal data structure which is defined by a `key` which identifies each unique time series and an `index` which identifies the time indices. Other columns are treated as measured variables. Since `theft` is a univariate tool, `calculate_features` only accepts inputs that have one measured variable. Since much of the functionality in `theftdlc` is associated with time-series classification problems, if multiple keys are defined, the first is treated as an "ID" variable, while the second is treated as the "grouping" variable for classification.

Users can control various aspects of the feature extraction process through modification of optional arguments. The `catch22` feature set can be expanded to form 'catch24' with included mean and standard deviation by setting `catch24 = TRUE`. Features from the `"compengine"` subset of **tsfeatures** can be calculated by setting `use_compengine = TRUE` which substantially increases computation time for the addition of 16 features. The in-built algorithm in `tsfresh` for selecting relevant features can be used for that set by specifying `tsfresh_cleanup = TRUE` (Christ et al., 2017). In addition, it is a common practice in feature-based time-series analysis to quantify the relative performance of features. For situations where differences in mean and variance are not of interest, z-scoring can be used to standardize each time series prior to the calculation of time-series features. The argument `z_score` enables automatic normalization of each time series within `calculate_features` prior to computing features. Further, users may always wish to disable package warnings when computing features. This can be achieved by setting `warn = FALSE`. Last, parallel processing can be engaged by setting the `n_jobs` argument to a value ≥ 2 (`calculate_features` defaults to serial processing). Currently, parallelization has only been implemented for **tsfeatures**, `tsfresh`, and `tsfel`.

The output of `calculate_features` is an S3 object of class `feature_calculations`, which in this example is stored in the R environment as `all_features`. Within this object is a data frame which contains five columns if the dataset is labeled (as in time-series classification), and four otherwise: `id` (unique identifier for each time series), `names` (feature name), `values` (feature value), `feature_set` (feature set name), and `group` (class label, if applicable). This output structure ensures that, regardless of the feature set selected, the resulting object is always of the same format and can be used with the rest of the **theftdlc** functions without manual data reshaping.

For the Bonn EEG dataset of 500 time series, each of length $T = 4097$ samples, calculating features for all sets in **theft** took $\approx$ 5.7 hours on a 2019 MacBook Pro with an Intel Core i7 2.6 GHz 6-Core CPU. The extensive computation time was largely driven by **tsfeatures**, noting that the other five sets ranged from just seconds (`catch22`) to several minutes. With the sixteen `"compengine"` features enabled, computation time increased to $\approx$ 11.8 hours. Previous work provides a comprehensive discussion of computation speed between the sets and the scalability with time-series length (Henderson and Fulcher, 2021). An example call which extracts features from all six sets for the Bonn EEG dataset is shown in the code below.

```
all_features <- calculate_features(
  data = bonn_eeg,
  feature_set = c("catch22", "feasts", "tsfeatures",
                  "tsfresh", "tsfel", "kats"),
  use_compengine = FALSE, catch24 = TRUE)
```

theft is also set up to enable users to calculate their own custom features. For example, we could specify two new features such as the mean and standard deviation by adding the requisite functions and the names for those features in a list to the additional `features` argument, as demonstrated in the code below. User-supplied functions must take a vector input and return a numeric scalar value to be a valid time-series feature.

```
all_features_msd <- calculate_features(
                        data = bonn_eeg,
                        feature_set = c("catch22", "feasts", "tsfeatures",
                                        "tsfresh", "tsfel", "kats"),
                        features = list("mean" = mean, "sd" = sd),
                        use_compengine = FALSE)
```

In addition, previous work highlighted that it can be helpful to provide a simple benchmark of performance for the more comprehensive feature sets (Henderson et al., 2023). As such, two simple feature sets—"`quantiles`" and "`moments`"—are also available in `calculate_features` to enable the quick computation of a set of quantiles and the first four moments of the distribution to serve as a baseline for more sophisticated feature sets. "`quantiles`" and "`moments`" are both able to be specified directly as values to the `feature_set` argument of `calculate_features`. By default, the `"quantiles"` feature set includes a collection of 100 quantiles from the range 0.01 to 1.

Users who wish to explore the computed results efficiently can also download the pre-computed features and classifiers used in this paper:

```
files <- c("all_features", "mf_results", "feature_classifiers")

for(f in files){
  temp <- tempfile()
 download.file(paste0("https://github.com/hendersontrent/bonn-eeg-data/raw/refs/heads/main/",
                      f, ".Rda"), temp)
  load(temp)
  unlink(temp)
}
```

2.4 Normalizing features

Different features vary over very different ranges; e.g., features that estimate p-values from a hypothesis test vary over the unit interval, whereas a feature that computes the length of a time series takes (often large) positive integer values. These differences in scale can complicate the visualization of feature behavior and the construction of statistical learning algorithms involving diverse features. To overcome these limitations, a common pre-processing step involves scaling all features. Several of **theftdlc**'s internal functions utilize rescaling functionality—specifically, providing the user the choice of five methods for converting a set of raw feature values, $\mathbf{x}$, to a normalized version, $\mathbf{z}$:

1. z-score (`"zScore"`): $z_i = \frac{x_i - \mu}{\sigma}$,
2. linear scaling to unit interval (`"MinMax"`): $z_i = \frac{x_i - \min(\mathbf{x})}{\max(\mathbf{x}) - \min(\mathbf{x})}$,
3. maximum absolute scaling (`"MaxAbs"`): $z_i = \frac{x_i}{|\max(\mathbf{x})|}$
4. sigmoid (`"Sigmoid"`): $z_i = \left[1 + \exp(-\frac{x_i - \mu}{\sigma})\right]^{-1}$,
5. and outlier-robust sigmoid (`"RobustSigmoid"`): $z_i = \left[1 + \exp\left(-\frac{x_i - \mathrm{median}(\mathbf{x})}{\mathrm{IQR}(\mathbf{x})/1.35}\right)\right]^{-1}$,

where μ is the mean, σ is the standard deviation, and $\mathrm{IQR}(\mathbf{x})$ is the interquartile range of $\mathbf{x}$. All four transformations end with a linear rescaling to the unit interval. The outlier-robust sigmoid transformation, introduced in (Fulcher et al., 2013), can be helpful in normalizing feature-value distributions with large outliers. Feature normalization is implemented in the R package **normaliseR** which is a key dependency for **theftdlc** (Henderson, 2024).

2.5 Visualizing the feature matrix

A hallmark of large-scale feature extraction is the ability to visualize the intricate patterns of how different time-series analysis algorithms behave across a time-series dataset. This can be achieved in **theftdlc** by specifying `type = "matrix"` when calling `plot` on a `feature_calculations` object to produce a heatmap of the time series (rows) $\times$ feature matrix (columns) which organizes the rows and columns to help reveal interesting patterns in the data. The plot of the combination of all six open feature sets for the Bonn EEG dataset is shown in Figure 2. We can see some informative structure in this graphic, including many groups of features with similar behavior on this dataset (i.e., columns with similar patterns), indicating substantial redundancy across the joint set of features (Henderson and Fulcher, 2021). The bottom block of 100 rows, which visually have the most distinctive properties, was found to correspond to time series from the `seizure` class, indicating the ability of this large combination of time-series features to meaningfully structure the dataset.

```
plot(all_features,
     type = "matrix",
     norm_method = "RobustSigmoid",
     clust_method = "average")
```

In matrix plots in **theftdlc**, hierarchical clustering is used to reorder rows and columns so that time series (rows) with similar properties are placed close to each other and features (columns) with similar behavior across the dataset are placed close to each other—where similarity in behavior is quantified using Euclidean distance in both cases (Day and Edelsbrunner, 1984). In Figure 2, we specify the usage of average (i.e., unweighted pair group method with arithmetic mean) agglomeration. Default settings within `plot` enable users to easily generate outputs in a single line of code, but more advanced users may seek to tweak the optional arguments. For example, different linkage algorithms for hierarchical clustering can be controlled by supplying the argument to `clust_method`, which uses average agglomeration as a default, and the different rescaling methods defined earlier can be supplied to the `norm_method` argument, which defaults to `"zScore"`.

2.6 Projecting low-dimensional feature-spaces

Low-dimensional projections are a useful tool for visualizing the structure of high-dimensional datasets in low-dimensional spaces. Here we are interested in representing a

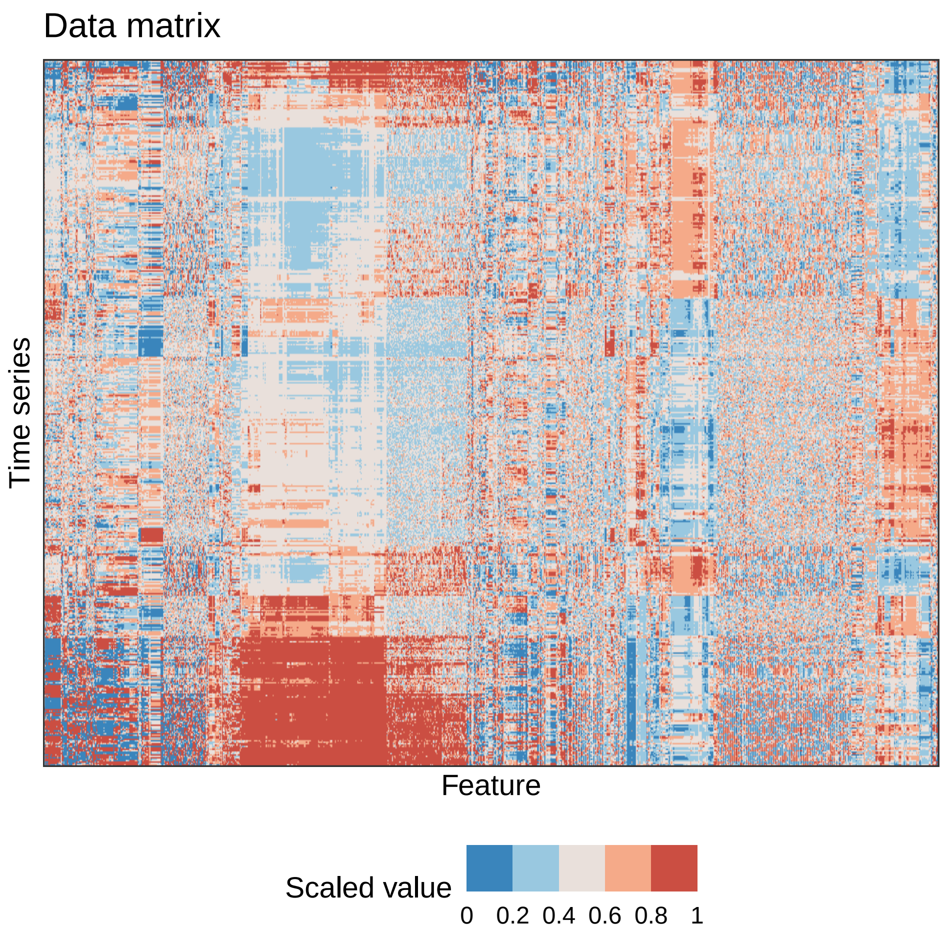

Figure 2: A time series by feature matrix heatmap produced by generating a matrix plot on the feature calculations object. Extracted feature vectors for each time series (500) in the Bonn EEG dataset using all six feature sets in theft (1005 features in total, after filtering out 85 features with NaN values) are represented as a heatmap. Similar features (columns) and time series (rows) are positioned close to each other using (average) hierarchical clustering. Each tile is a normalized value for a given time series and feature.

time-series dataset in a two-dimensional projection of the feature space, which can reveal structure in the dataset, including how different labeled classes are organized. For linear dimensionality reduction techniques, such as principal components analysis (PCA) (Jolliffe, 2002), the results can be visualized in two dimensions as a scatterplot, where the principal component (PC) that explains the most variance in the data is positioned on the horizontal axis and the second PC on the vertical axis, and each time series is represented as a point (colored by its group label in the case of a labeled dataset). When the structure of a dataset in the low-dimensional feature space matches known aspects of the dataset (such as class labels), it suggests that the combination of diverse time-series features can capture relevant dynamical properties that differ between the classes. It can also reveal new types of structure in the dataset, like clear sub-clusters within a labeled class, that can guide new understanding of the dataset. Low-dimensional projections of time-series features have been shown to meaningfully structure time-series datasets: revealing sex and day/night differences in *Drosophila* (Fulcher and Jones, 2017), distinguishing types of stars based on their light curves (Barbara et al., 2022), and categorizing sleep epochs (Decat et al., 2022).

Several algorithms for projecting feature spaces are available in **theftdlc**:

- Principal components analysis (PCA)—`"PCA"`
- *t*-Stochastic Neighbor Embedding (*t*-SNE)—`"tSNE"`
- Classical multidimensional scaling (MDS)—`"ClassicalMDS"`
- Kruskal's non-metric multidimensional scaling—`"KruskalMDS"`
- Sammon's non-linear mapping non-metric multidimensional scaling—`"SammonMDS"`
- Uniform Manifold Approximation and Projection for Dimension Reduction (UMAP)—`"UMAP"`

Time-series datasets can be projected in low-dimensional feature spaces in **theftdlc** using the `project` function. Users can control algorithm hyperparameters by supplying additional arguments to the `...` argument of `project`—the function then passes these arguments to the requisite dimension reduction algorithm internally. The `project` function returns an `S3` object of class `feature_projection`, which can then be passed to `plot` which will automatically draw the appropriate two-dimensional scatterplot. The `feature_projection` class is a list object which contains elements representing the user-supplied feature data frame, the model data, the two-dimensional projected data frame, and the model fit object itself. The plot method for this object contains only one other argument (`show_covariance`) which specifies whether to draw covariance ellipses for each group in the scatterplot (if a grouping variable is detected).

The low-dimensional projection plot for the Bonn EEG dataset (using *t*-SNE and all $>$ 1000 non-`NaN` features across the six feature sets included in **theft**) is shown in Figure 3 with perplexity 15, as produced by the code below. The low-dimensional projection meaningfully structures the labeled classes of the dataset. Specifically, two of the presurgical diagnosis classes—`epileptogenic` (epileptogenic zone) and `hippocampus` (hippocampal formation of the opposite hemisphere of the brain)—appear to exhibit considerable overlap in the projected space, while the two healthy volunteer classes `eyesOpen` (awake state with eyes open) and `eyesClosed` (awake state with eyes closed) occupy space further away from the other classes but closer to each other. The `seizure` class occupies a space largely separate from the other four classes in the projection, consistent with its distinctive periodic dynamics (Fulcher et al., 2013).

```
low_dim_calc <- project(all_features,
                        method = "MinMax",
                        low_dim_method = "tSNE",
                        perplexity = 15)

plot(low_dim_calc)
```

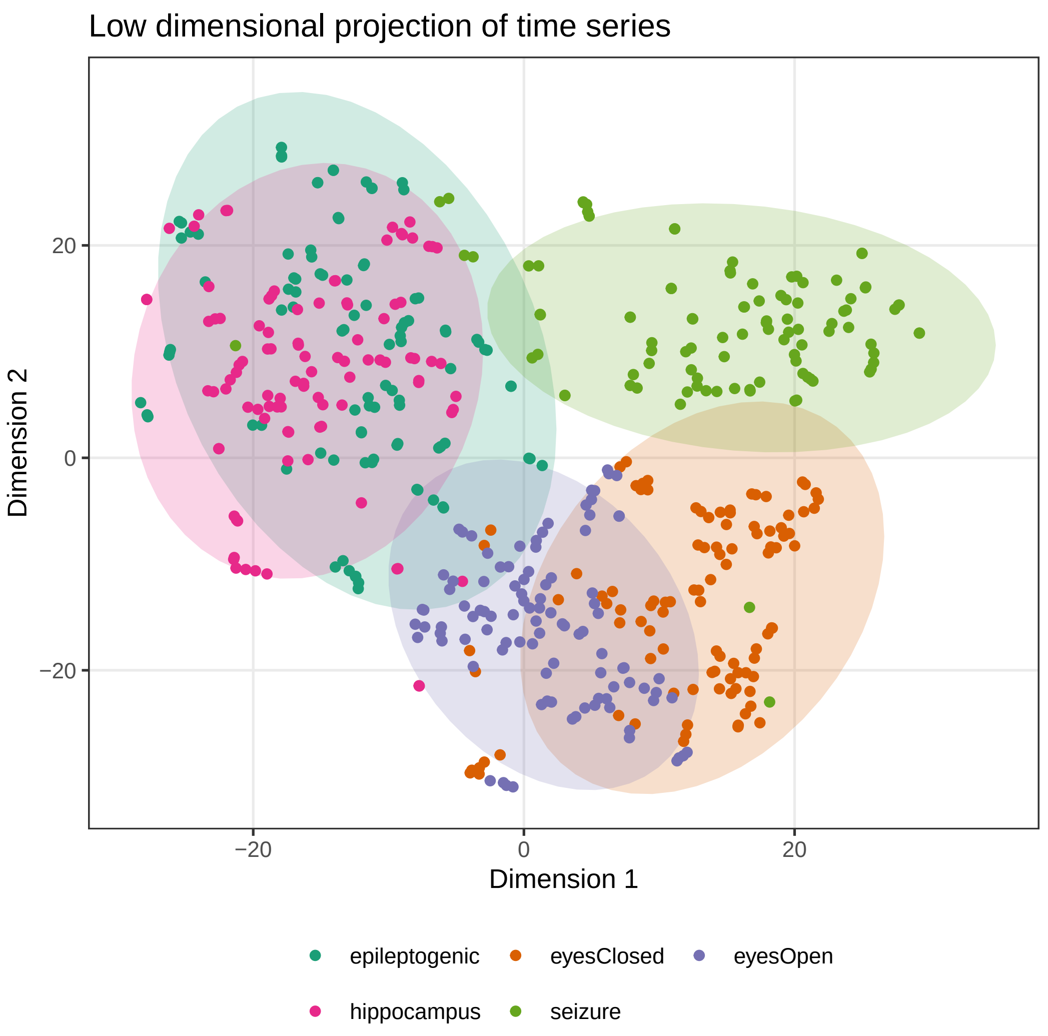

Figure 3: Low-dimensional projection of the Bonn EEG dataset using theft. Using t-SNE with perplexity 15, the high-dimensional feature space of >1000 features is projected into two dimensions. Each point represents a time series which is colored according to its class label. Time series that are located close in this space have similar properties, as measured by the six feature sets in theft.

2.7 Constructing classifiers with multiple features

Combinations of complementary, discriminative features can often be used to construct accurate time-series classifiers (Fulcher and Jones, 2014). Drawing on computed time-series features (that may derive from one or more existing feature sets), **theftdlc** can fit and evaluate classifiers using the `classify` function. This allows users to evaluate the relative performance of each feature set, of the combination of all sets, or any other combination of features. Providing easy access to a range of classification algorithms and accompanying inferential tools (such as null permutation testing to obtain p-values and performance distributions through the resampling-based algorithm) through `classify` allows users to compare sets of features to better understand the most accurate feature sets for a given time-series classification problem. The code presented below provides an example usage for the Bonn EEG dataset with a linear support vector machine (SVM) classifier (which is the default and so does not require explicit specification).

```
mf_results <- classify(
                data = all_features,
                by_set = TRUE,
                train_size = 0.8,
                n_resamples = 100,
                use_null = TRUE)
```

The `classify` function is flexible in that users can supply any function that can be used with R's native `stats::predict` generic. For example, a user could easily use a radial basis function SVM instead:

```
rbfClassifier <- function(formula, data){
  mod <- e1071::svm(formula, data = data, kernel = "radial", scale = FALSE,
                    probability = TRUE)
}

mf_results_rbf <- classify(
                    data = all_features,
                    classifier = rbfClassifier,
                    by_set = TRUE,
                    train_size = 0.8,
                    n_resamples = 100,
                    use_null = TRUE)
```

In the above code, we specified that we want `classify` to fit separate classifiers for each feature set, using a training set size that is 80% of the input data size, with 100 resamples for each feature set as a way to incorporate uncertainty. We also enabled null model fitting for permutation testing. In applications involving small datasets, or when small effects are expected, it is useful to quantify how different the observed classification performance is from a null setting in which data are classified randomly (i.e., could the same results have been obtained by chance?). One method for inferring test statistics is to use permutation testing—a procedure that samples a null process many times to form a distribution against which a value of importance (i.e., the classification accuracy result from a model) can be compared to estimate a p-value (Ojala and Garriga, 2009). In **theft**, permutation testing is implemented for evaluating classification performance in `classify` through the `use_null` argument. When set to `TRUE`, `classify` will compute results for `n_resamples` models where the class labels match the data, but also `n_resamples` models where the class labels are shuffled, thus severing the input-output relationship. In the absence of any data errors, we would expect the mean classification accuracy for the empirical null distribution to approximate chance for the problem. This provides a useful comparison point for the main (non-shuffled) models—if the main models statistically outperform the empirical null models, then the result, quantified by the p-value, is likely not due to chance, thus indicating that the set of time-series features represent quantities that can meaningfully capture differences between classes.

The resampling procedure begins by partitioning the input data into `n_samples` number of seeded train-test splits for reproducibility, using `train_size` to govern the size of the training set for each split. Importantly, the procedure tracks the class representation in the first resample and preserves these proportions for all subsequent resamples. The feature data for each resample is then normalized as a z-score by computing the mean and standard deviation of each feature in the training set, and using these values to rescale both the training and test sets. This ensures that test data are completely unseen. The procedure then iterates over each resample and fits the classification algorithm for each. By default, `classify` calculates the following accuracy metrics, where C is the number of classes, TP is the number of true positives, FP is the number of false positives, and FN is the number of false negatives:

- Accuracy (`"accuracy"`): $\frac{\sum_{i=1}^{C} TP_i}{\sum_{i=1}^{C}(TP_i+FP_i+FN_i)}$
- Mean precision (`"precision"`): $\frac{1}{C}\sum_{i=1}^{C}\frac{\mathrm{TP}_i}{\mathrm{TP}_i+\mathrm{FP}_i}$
- Mean recall (`"recall"`): $\frac{1}{C}\sum_{i=1}^{C}\frac{\mathrm{TP}_i}{\mathrm{TP}_i+\mathrm{FN}_i}$
- F1 score (`"f1"`): $2\frac{\mathrm{Precision}\cdot\mathrm{Recall}}{\mathrm{Precision}+\mathrm{Recall}}$

`classify` returns a list with two elements: (i) the train and test sizes; and (ii) a data frame of classification results. While the raw results are useful, **theftdlc** enables automated analysis of them. The `theftdlc::interval` and `compare_features` functions were designed to enable fast and intuitive comparisons between individual features and entire sets. `theftdlc::interval` produces summaries of classification results with uncertainty using three distinct methods: (i) standard deviation—`"sd"`; (ii) confidence interval based off the t-distribution—`"se"`; and (iii) quantile summary—`"quantile"`.

Users can compute different interval summaries by modifying the additional arguments:

- `metric`—the classification performance metric to calculate intervals for. Can be one of `"accuracy"`, `"precision"`, `"recall"`, or `"f1"`
- `by_set`—whether to compute intervals for each feature set. If `FALSE`, the function will instead calculate intervals for each individual feature
- `type`—whether to calculate a $\pm SD$ interval with `"sd"`, confidence interval based off the t-distribution with `"se"`, or a quantile with `"quantile"`
- `interval`—the width of the interval to calculate. Defaults to 1 if `type = "sd"` to produce a $\pm 1SD$ interval. Defaults to `0.95` if `type = "se"` or `type = "quantile"` for a 95% interval
- `model_type`—whether to calculate intervals for main models with `"main"` or null models with `"null"` if the `use_null` argument of `classify` was `use_null = TRUE`

Below we calculate the mean $\pm 1SD$ for each feature set. For the Bonn EEG dataset, we find that the set of all features (`All features`) produced the highest mean classification accuracy (91.2%) and `catch22` (the smallest feature set) produced the lowest mean accuracy (80.1%). The best performing individual feature set was the largest—`tsfresh`—with a mean classification accuracy of 90.8% which was marginally below the set of all features.

```
set_intervals <- theftdlc::interval(
  mf_results,
  metric = "accuracy",
  by_set = TRUE,
  type = "sd",
  model_type = "main"
  )

set_intervals
```

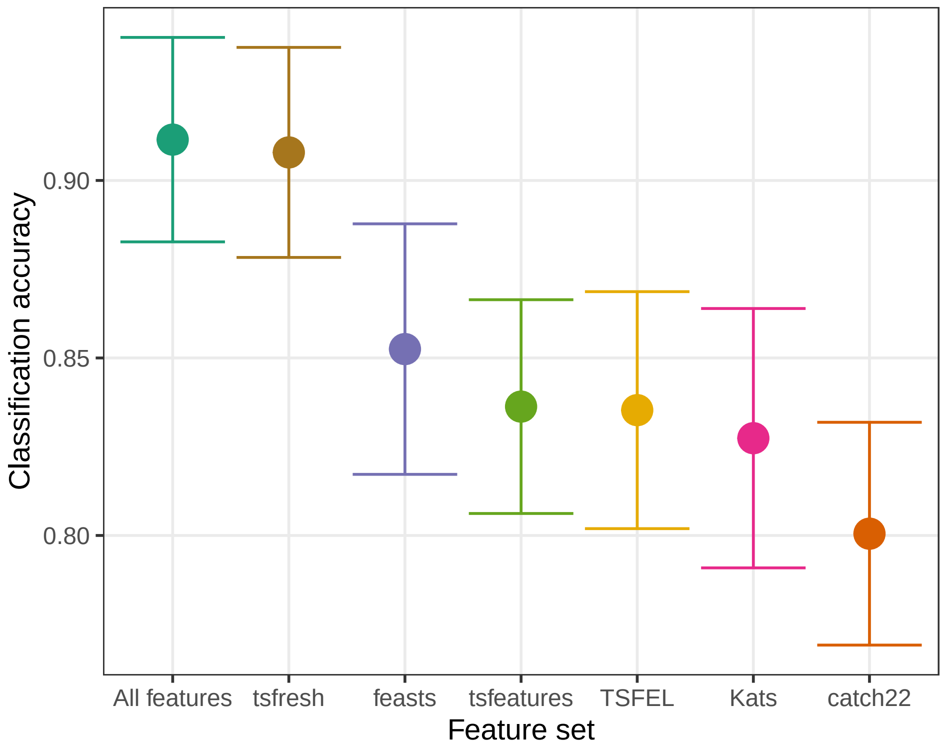

Figure 4: Comparison of mean classification accuracy between feature sets in theft for the five-class Bonn EEG classification task. Classification accuracy using a linear SVM is presented for each of the six feature sets in theft as well as the combination of all their features. The number of features retained for analysis after filtering is displayed in parentheses after the feature set name on the horizontal axis which has been sorted from highest to lowest mean accuracy. Mean classification accuracy across the same 100 resamples is displayed as colored points for each set with one standard deviation error bars.

```
#>    feature_set  .mean    .lower    .upper
#> 1 All features 0.9115 0.8827069 0.9402931
#> 2         Kats 0.8274 0.7908680 0.8639320
#> 3        TSFEL 0.8353 0.8019226 0.8686774
#> 4      catch22 0.8005 0.7691217 0.8318783
#> 5       feasts 0.8525 0.8172126 0.8877874
#> 6   tsfeatures 0.8363 0.8061940 0.8664060
#> 7      tsfresh 0.9079 0.8783138 0.9374862
```

`theftdlc::interval` returns an `S3` object of class `interval_calculations` which is a data frame that can be used with the `plot` generic through **theftdlc**. The resulting plot is presented in Figure 4 where we see the considerable overlap in mean $\pm 1 SD$ classification accuracy between the feature sets.

```
plot(set_intervals)
```

While the visual aid is useful, we often want to understand if a given feature set has performed better than we might expect due to chance. Or can we determine if a given feature set has statistically outperformed another (such as the set of all features compared to `tsfresh`), therefore providing guidance on which features to retain for subsequent analysis? How can we estimate if the performance of the smallest and fastest-to-compute set – `catch22` – is meaningfully different from the larger feature sets with close average classification accuracy values (i.e., `Kats`, `TSFEL`, and `tsfeatures`)? The `compare_features` function in **theftdlc** enables pairwise comparisons between either individual features or entire feature sets, or to their own respective empirical null distributions (i.e., the 'chance' distribution). It does so through usage of the resampled t-test which accounts for the correlation between samples in the calculation of the test statistic (Nadeau and Bengio, 2003). The resampled t-test is implemented through the **correctR** R package (Henderson, 2025a). `compare_features` returns a data frame with columns for the hypothesis that was tested, the test statistic, p-value, adjusted p-value (if specified), and the means of each group for each hypothesis test. `compare_features` can operate on any of the metrics computed in `classify`. The function only takes a small number of arguments in addition to the output of `classify`:

- `metric`—the classification performance metric to calculate intervals for. Can be one of `"accuracy"`, `"precision"`, `"recall"`, or `"f1"`
- `by_set`—whether to compare entire feature sets (`TRUE`) or individual features (`FALSE`)
- `hypothesis`—whether to compare each entire feature set or individual feature to its own empirical null distribution (`"null"`, if permutation testing was used in `classify`) or to each other (`"pairwise"`)
- `p_adj`—method for adjusting p-values for multiple comparisons. Defaults to `"none"` for no adjustments, but can take any valid option received by `stats::p.adjust`

For example, we can statistically compare the performance of each feature set against their empirical null distributions (formed by the distribution of performance on shuffled class label data) using the code presented below. When specifying `hypothesis = "null"`, **theftdlc** automatically applies a one-tailed test, since the hypothesis is that the main models should outperform their null counterparts. We find that, for the full Bonn EEG dataset with 100 time series per class and strong differences between signals, mean classification accuracy for each feature set is far higher than chance level (20%) over 100 resamples. Further, we obtain extremely small p-values close to zero (displayed in the table below as zero due to rounding for visual clarity), providing support for the low probability of obtaining classification accuracy results at least as extreme as what we observed under the null hypothesis. This confirms that time-series features can effectively distinguish between classes in the dataset. Note that we drop some of the columns reported by **theftdlc** here for spatial reasons.

```
compare_features(mf_results,
                 metric = "accuracy",
                 by_set = TRUE,
                 hypothesis = "null",
                 p_adj = "none")
```

hypothesis	set_mean	null_mean	t_statistic	p.value
All features != own null	0.911	0.192	12.496	0
Kats != own null	0.827	0.188	9.701	0
TSFEL != own null	0.835	0.197	10.638	0
catch22 != own null	0.800	0.203	8.699	0
feasts != own null	0.853	0.192	10.003	0
tsfeatures != own null	0.836	0.197	11.157	0
tsfresh != own null	0.908	0.192	13.787	0

We can then compare the feature sets to one another to provide statistical evidence for any differences in the performance ranges visualized in Figure 4 using the code below. For `hypothesis = "pairwise"`, **theftdlc** applies a two-tailed test. We find that the set of all features outperforms all individual sets at $\alpha < 0.05$ except for `tsfresh` ($p = 0.820$), `feasts` ($p = 0.094$), and `tsfeatures` ($p = 0.077$).

```
compare_features(mf_results,
                 metric = "accuracy",
                 by_set = TRUE,
                 hypothesis = "pairwise",
                 p_adj = "none")
```

hypothesis	set_a_mean	set_b_mean	t_statistic	p.value
All features != catch22	0.911	0.800	3.026	0.003
All features != feasts	0.911	0.853	1.713	0.090
All features != Kats	0.911	0.827	2.306	0.023
All features != tsfeatures	0.911	0.836	2.066	0.041
All features != TSFEL	0.911	0.835	2.058	0.042
All features != tsfresh	0.911	0.908	0.198	0.843
catch22 != feasts	0.800	0.853	-1.249	0.215
catch22 != Kats	0.800	0.827	-0.634	0.527
catch22 != tsfeatures	0.800	0.836	-0.871	0.386
catch22 != TSFEL	0.800	0.835	-0.873	0.385
catch22 != tsfresh	0.800	0.908	-2.776	0.007
feasts != Kats	0.853	0.827	0.579	0.564
feasts != tsfeatures	0.853	0.836	0.414	0.680
feasts != TSFEL	0.853	0.835	0.419	0.676
feasts != tsfresh	0.853	0.908	-1.436	0.154
Kats != tsfeatures	0.827	0.836	-0.198	0.843
Kats != TSFEL	0.827	0.835	-0.183	0.855
Kats != tsfresh	0.827	0.908	-2.006	0.048
tsfeatures != TSFEL	0.836	0.835	0.025	0.980
tsfeatures != tsfresh	0.836	0.908	-1.796	0.076
TSFEL != tsfresh	0.835	0.908	-1.913	0.059

2.8 Finding and understanding informative individual features

Fitting models which use multiple features as inputs is often useful for predicting class labels. However, users are also typically interested in understanding patterns in their dataset, such as interpreting the types of time-series analysis methods that best separate different classes, and the relationships between these top-performing features. This can be achieved using mass univariate statistical testing of individual features, quantifying their performance either relative to an empirical null distribution or each other. **theftdlc** implements the ability to identify top-performing features in the `compare_features` function by setting `by_set = FALSE`, with an example usage for the Bonn EEG dataset (using features from all six packages) shown in the code below. We implement parallel processing to speed up computation time.

```
feature_classifiers <- classify(data = all_features,
                               by_set = FALSE,
                               train_size = 0.8,
                               n_resamples = 100,
                               use_null = TRUE)

feature_vs_null <- compare_features(feature_classifiers,
                                    by_set = FALSE,
                                    hypothesis = "null",
                                    n_workers = 6)
```

Straightforward **dplyr** syntax can then be used to identify the top n features (Wickham et al., 2023). We have the choice of either mean classification accuracy or p-values relative to the empirical null to determine informative features. For illustrative purposes, here we have used mean classification accuracy to find the top $n = 40$ features. We show the top 20 features below for spatial reasons. We see that the 17 best-performing individual features achieve $> 50\%$ accuracy (which far exceeds the chance probability for a five-class problem of 20%).

```
top_40 <- feature_vs_null |>
  dplyr::slice_max(feature_mean, n = 40)

top_40 |>
  top_n(feature_mean, n = 20)
```

hypothesis	feature_mean	null_mean	t_statistic	p.value
tsfresh_values__ autocorrelation__lag_6 != own null	0.537	0.138	4.054	0.000
tsfresh_values__ autocorrelation__lag_7 != own null	0.537	0.136	4.244	0.000
tsfresh_values__ autocorrelation__lag_8 != own null	0.527	0.138	4.513	0.000
tsfresh_values__ change_quantiles__f_agg_"mean"__ isabs_True__qh_1.0__ql_0.4 != own null	0.524	0.142	7.373	0.000
tsfresh_values__ change_quantiles__f_agg_"mean"__ isabs_True__qh_1.0__ql_0.2 != own null	0.521	0.146	6.819	0.000
tsfresh_values__ change_quantiles__f_agg_"mean"__ isabs_True__qh_0.8__ql_0.4 != own null	0.520	0.148	7.016	0.000
TSFEL_0_Wavelet energy_25.0Hz != own null	0.515	0.145	7.903	0.000
TSFEL_0_Wavelet standard deviation_25.0Hz != own null	0.515	0.145	7.903	0.000
tsfresh_values__ change_quantiles__f_agg_"mean"__ isabs_True__qh_0.8__ql_0.2 != own null	0.512	0.148	6.362	0.000
tsfresh_values__ change_quantiles__f_agg_"mean"__ isabs_True__qh_0.6__ql_0.2 != own null	0.511	0.141	7.350	0.000
TSFEL_0_Median absolute diff != own null	0.504	0.147	6.645	0.000
TSFEL_0_Mean absolute diff != own null	0.504	0.146	6.298	0.000
TSFEL_0_Sum absolute diff != own null	0.504	0.146	6.298	0.000
tsfresh_values__ absolute_sum_of_changes != own null	0.504	0.146	6.298	0.000
tsfresh_values__ change_quantiles__f_agg_"mean"__ isabs_True__qh_1.0__ql_0.0 != own null	0.504	0.146	6.298	0.000
tsfresh_values__ mean_abs_change != own null	0.504	0.146	6.298	0.000
TSFEL_0_Signal distance != own null	0.501	0.146	6.136	0.000
tsfresh_values__ change_quantiles__f_agg_"mean"__ isabs_True__qh_0.8__ql_0.0 != own null	0.500	0.144	7.167	0.000
tsfresh_values__ autocorrelation__lag_5 != own null	0.500	0.132	3.280	0.001
tsfresh_values__ augmented_dickey_fuller__attr_ "teststat"__autolag_"AIC" != own null	0.497	0.140	5.100	0.000

Understanding what each feature within the table measures can provide insight into the types of features relevant for the classification problem. For example, we see features that measure properties associated with autocorrelation (e.g., tsfresh_values__autocorrelation_lag_6 which measures the value of the autocorrelation function at lag 6) and signal peaks (e.g., TSFEL_0_Median absolute diff which measures the median value of all absolute differences along the signal), among others. However, interpreting this table is challenging as the relationships between the features are unknown—are all the 40 features behaving differently, or are they all highly correlated to each other and essentially proxy metrics for the same underlying time-series property? We can better understand these relationships by visualizing the pairwise feature $\times$ feature correlation matrix.

To achieve this, we can filter the original feature data to only include the top features, assign it to a feature_calculations object type, and make use of the plot function in **theftdlc** to visualize pairwise absolute correlations between the top performing features. This is presented visually in Figure 5. The plot reveals two main groups of highly correlated ($|\rho| \gtrapprox 0.8$) features: in the bottom left and upper right of the plot. The cluster in the bottom left contains features that capture different types of autocorrelation structure in the time series, including linear autocorrelation coefficients (e.g., tsfresh_values__autocorrelation_lag_6) and the variance of means over sliding windows (e.g., tsfeatures_stability). The large cluster in the top right (containing features from tsfresh and TSFEL exclusively) contains features sensitive to variance—including change quantiles (e.g., tsfresh_values__change_quantiles__f_agg_"mean"__isabs_True__qh_1.0__ql_0.4 which measures the mean of the absolute change of the time series values inside quantiles $0.4 - 1.0$), wavelet variance (e.g., TSFEL_0_Wavelet standard deviation_25.0Hz which measures the variance of coefficients from Ricker wavelets with widths $1 - 10$ at the lower quarter of the assumed sampling frequency of 100Hz), and the 'distance traveled' by the signal (e.g., TSFEL_0_Signal distance which measures the sum of square root squared differences). While the differences between classes—as identified through the list of top features—in this case were simple (i.e., autocorrelation and variance), other, more complex features may perform the strongest on other problems, or even different pairs of classes within the five-class dataset investigated here. Identifying when simple features perform well is important as it can provide interpretable benchmarks for assessing relative performance gains achieved by more complex and/or less interpretable alternative classifiers (Henderson et al., 2023).

```
feature_matrix_filt <- all_features |>
  dplyr::filter(feature_set %in% top_40$feature_set &
                names %in% top_40$original_names) |>
  structure(class = c("feature_calculations", "data.frame"))

plot(feature_matrix_filt, type = "cor")
```

Having identified the discriminative features, it can be important to understand how they differ amongst the labeled classes of a dataset. This can be achieved by visualizing the distribution of values for each class for each of the features. In **theftdlc**, a violin plot can be produced in plot by setting type = "violin", where each time series is represented as a point organized and colored by its class label. Note that a boxplot alternative (which highlights univariate outliers as points) is also possible through specifying type = "box". Here, for visual clarity, we show violin plots for a selected feature from the variance cluster of features from Figure 5: 0_Signal distance from TSFEL (mean classification accuracy 50.1% over 100 resamples); and a selected feature from the autocorrelation-sensitive cluster of features: values_autocorrelation_lag_6 from tsfresh (mean classification accuracy 53.7% over 100 resamples). The outputs are shown in Figure 6. Consistent with their high classification scores relative to chance (20%), both features are individually informative of class differences. The plot shows that with regard to autocorrelation structure, we see that eyesClosed exhibits typically weak to moderate negative coefficient values at lag

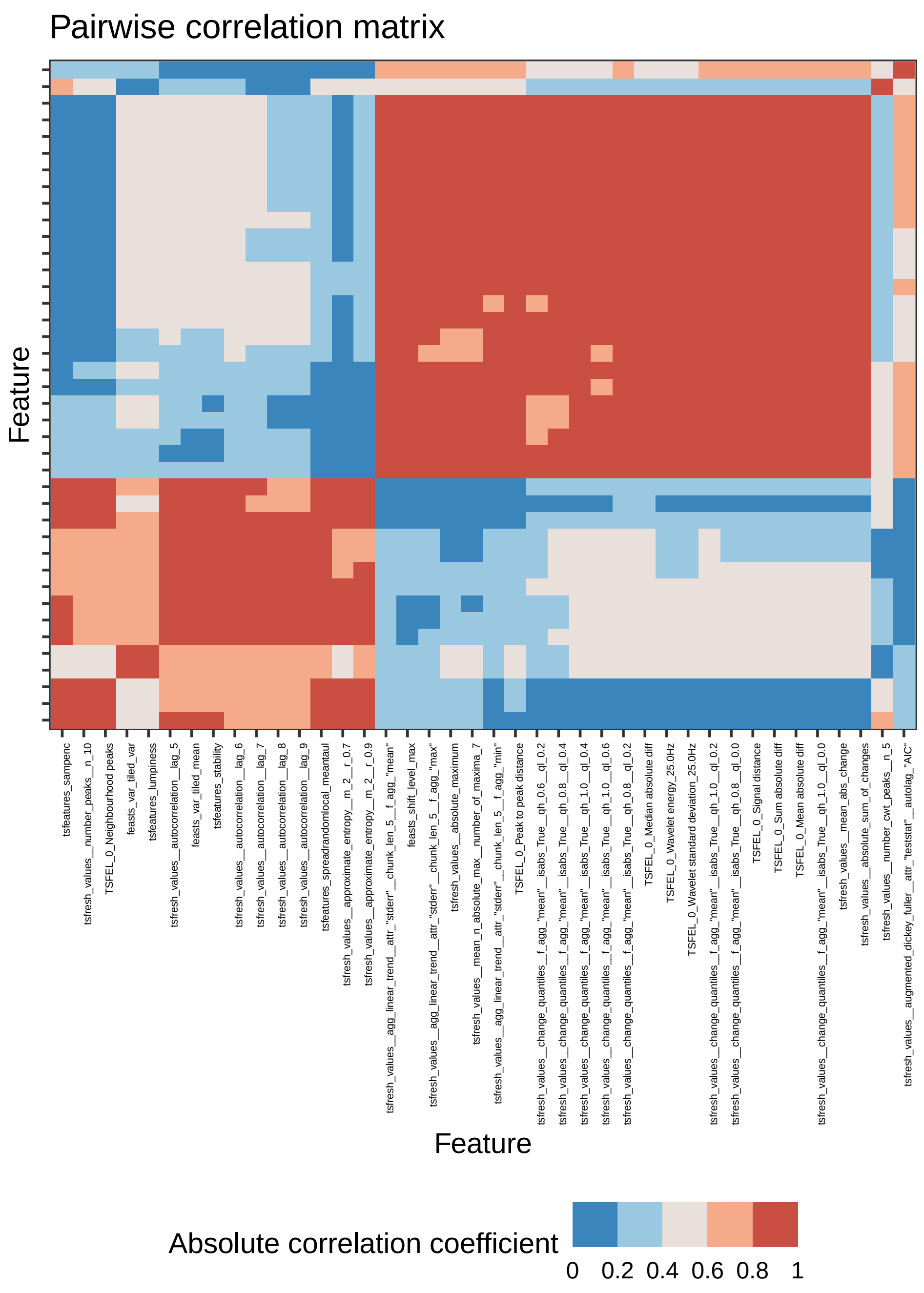

Figure 5: A group of change quantile and difference-associated features and a group of autocorrelation-sensitive features perform the best at distinguishing between the five classes in the Bonn EEG dataset using the absolute Spearman correlation coefficient to capture feature-feature similarity. To aid the identification of similarly performing features, the matrix of correlation coefficients between features were then organized using hierarchical clustering (on Euclidean distances with average linkage) along rows and columns to order the heatmap graphic.

6, while hippocampus and epileptogenic exhibit typically moderate positive coefficients. eyesClosed is characterized by weak to moderate coefficient values largely within the $0 - 0.5$ range, while seizure exhibits a substantially wider distribution of values than the other classes—a distribution which almost spans the entire range of coefficients exhibited by the others. This defining lack of temporal predictability in short-range autocorrelation coefficients for seizure time series is consistent with prior work and is characteristic of the more erratic nature of seizure state brain activity (Fulcher et al., 2013).

The plot also shows that with regard to signal distance (i.e., $\sum_{t=1}^{T-1} \sqrt{1 + (x_{t+1} - x_t)}$ where T is the length of the time series and x is the vector of values), all classes except seizure exhibit similar values with subtle differences in the mean and variance of their feature value distributions. This is consistent with the lower classification performance of this feature compared to values_autocorrelation_lag_6. For seizure, we again see a wide distribution of feature values definitive of this class. Practically, since signal distance measures the total distance traveled by the signal between time points, it can be inferred that seizure state brain activity (as measured by an EEG) fluctuates far more than healthy brain activity and measurements from the epileptogenic zone (i.e., the difference between values at any two consecutive time points is, on average, larger), consistent with known dynamics of seizure states (Andrzejak et al., 2001).

Together, these two feature case studies reinforce the interpretative benefits to a feature-based approach to time-series analysis. Features can not only organize time-series data, reveal structure, and predict class membership, but they can also provide insight into the underlying generative properties that distinguish different time series—such as the difference in brain activity between seizure state and regular brain function.

```
plot(feature_matrix_filt,
     type = "violin",
     feature_names = c("values__autocorrelation__lag_6",
                       "0_Signal distance")) +
  theme(strip.text = element_text(size = 6))
```

2.9 Additional functionality

In addition to the functionality demonstrated here, **theft** and **theftdlc** include a collection of other functions, not demonstrated in this article for brevity, including cluster analysis (through the cluster function in **theftdlc**), simple feature selection using penalized maximum likelihood generalized linear models (through the shrink function in **theftdlc**), and the processing of hctsa-formatted Matlab files in **theft**. Readers are encouraged to explore this additional functionality in the detailed vignettes included with the packages.

3 Discussion

Feature-based time-series analysis is a powerful computational tool for tackling statistical learning problems using sequential (typically time-ordered) data. We have introduced the **theft** and **theftdlc** packages for R which implement the extraction, processing, visualization, and statistical analysis of time-series features. The value of time-series features stems from their interpretability and strong connection to theory that can be used to understand the empirical properties of their dynamics. **theft** provides a unified interface to extracting features from six open-source packages—catch22, **feasts**, **tsfeatures**, Kats, tsfresh, and TSFEL—while **theftdlc** provides a comprehensive range of analyses to leverage the combined contributions from all of these packages. For the first time in the free and open-source software setting, the **theft** ecosystem provides a full workflow for conducting feature-based time-series analysis, taking the analyst from feature extraction through to generating interpretable insights about their data. **theftdlc** introduces a set of simply named functions that make analysis of time-series features calculated in **theft** intuitive and streamlined:

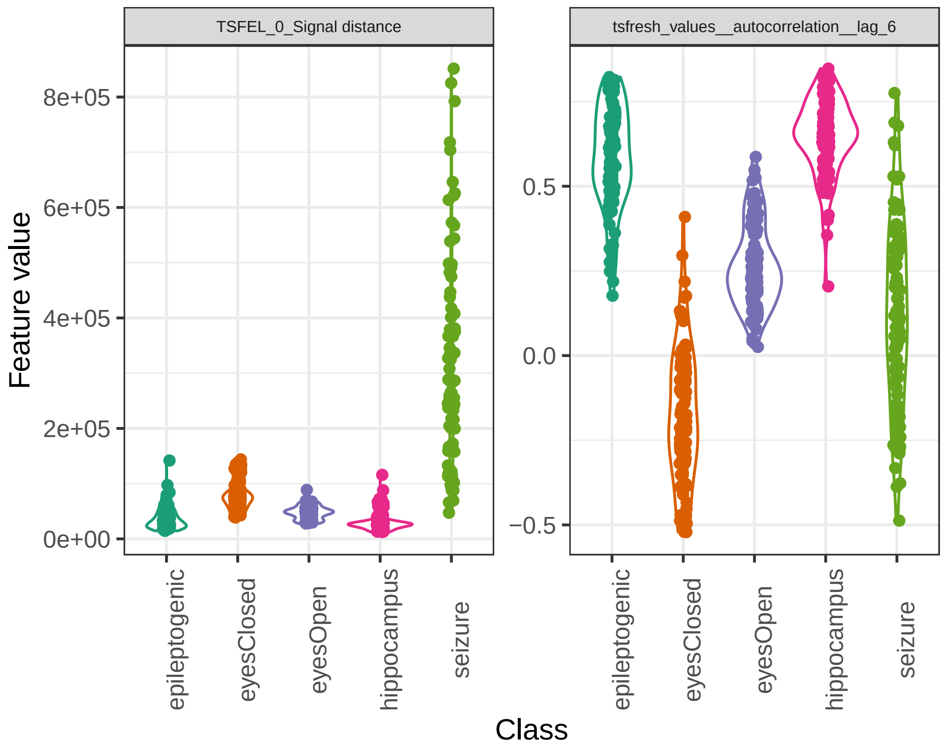

Figure 6: Violin plots (on original feature value scale) of a sample of two of the top 40 features of all six feature sets in theft for classifying Bonn EEG groups. Classes differ in their variance and autocorrelation properties.

`classify`, `project`, `cluster`, `interval`, `plot`, and `compare_features`. **theft** and **theftdlc** reduce the need to construct complex, bespoke workflows with multiple software libraries that were not designed to work together—the **theft** ecosystem provides an extensive suite of functions, but also presents a set of templates for advanced users to alter and adapt as their research requires.

We demonstrated the **theft** ecosystem on the five-class Bonn EEG time-series classification problem (Andrzejak et al., 2001), in which the full feature-based classification analysis pipeline—from feature extraction to normalization, classification, and interpretation of individual features—was achieved using a small number of key functions in **theft** and **theftdlc**. We showed that this intuitive pipeline could be used to derive insights about the temporal patterns which distinguish different classes of EEG time series and produce high-performing results in a simple statistical learning classification context. In other settings, emphasis may be placed on the classification procedure, where more complex classifiers—such as Gaussian processes or generalized additive models—may yield strong results. In others, users may not have a labeled dataset and instead seek to uncover structure in their data. For such cases, the `cluster` functionality of **theftdlc** may prove valuable in deriving scientific understanding. Regardless of the feature-based time-series analysis context, **theft** and **theftdlc** enable consistent, end-to-end analytical pipelines.

As new and more powerful features (and feature sets) are developed in the future, they can be incorporated into **theft** to enable ongoing assessments of the types of problems they are best placed to solve. In addition to the analysis templates provided through functions in **theftdlc**, there is much flexibility for users to adapt them or build new functionality for their own use-cases, such as applying different types of statistical learning algorithms on extracted feature matrices (e.g., feature selection), or to adapt the results to different applications such as extrinsic regression (Tan et al., 2021) or forecasting (Montero-Manso et al., 2020). Future work could also aim to reduce redundancy from across the combined features towards a new reduced feature set that combines the most generically informative and unique features from across the available feature-extraction packages (following the aims of the `catch22` feature set, selected from a library of >7700 candidate features in `hctsa` (Lubba et al., 2019)).

References

R. G. Andrzejak, K. Lehnertz, F. Mormann, C. Rieke, P. David, and C. E. Elger. Indications of nonlinear deterministic and finite-dimensional structures in time series of brain electrical activity: Dependence on recording region and brain state. *Physical Review. E, Statistical, Nonlinear, and Soft Matter Physics*, 64(6 Pt 1):061907, Dec. 2001. ISSN 1539-3755. doi: 10.1103/PhysRevE.64.061907. [p46, 63, 64]

M. Barandas, D. Folgado, L. Fernandes, S. Santos, M. Abreu, P. Bota, H. Liu, T. Schultz, and H. Gamboa. TSFEL: Time Series Feature Extraction Library. *SoftwareX*, 11:100456, Jan. 2020. ISSN 2352-7110. doi: 10.1016/j.softx.2020.100456. [p45]

N. H. Barbara, T. R. Bedding, B. D. Fulcher, S. J. Murphy, and T. Van Reeth. Classifying Kepler light curves for 12,000 A and F stars using supervised feature-based machine learning. *Monthly Notices of the Royal Astronomical Society*, page stac1515, June 2022. ISSN 0035-8711. doi: 10.1093/mnras/stac1515. [p43, 52]

M. Christ, A. W. Kempa-Liehr, and M. Feindt. Distributed and parallel time series feature extraction for industrial big data applications, May 2017. [p44, 48]

M. Christ, N. Braun, J. Neuffer, and A. W. Kempa-Liehr. Time Series FeatuRe Extraction on basis of Scalable Hypothesis tests (tsfresh – a Python package). *Neurocomputing*, 307: 72–77, Sept. 2018. ISSN 0925-2312. doi: 10.1016/j.neucom.2018.03.067. [p44]

R. B. Cleveland, W. S. Cleveland, J. E. McRae, and I. Terpenning. STL: A seasonal-trend decomposition procedure based on loess (with discussion). *Journal of Official Statistics*, 6: 3–73, 1990. [p44]

W. H. E. Day and H. Edelsbrunner. Efficient algorithms for agglomerative hierarchical clustering methods. *Journal of Classification*, 1(1):7–24, Dec. 1984. ISSN 1432-1343. doi: 10.1007/BF01890115. [p50]

N. Decat, J. Walter, Z. H. Koh, P. Sribanditmongkol, B. D. Fulcher, J. M. Windt, T. Andrillon, and N. Tsuchiya. Beyond traditional sleep scoring: Massive feature extraction and data-driven clustering of sleep time series. *Sleep Medicine*, 98:39–52, Oct. 2022. ISSN 1389-9457. doi: 10.1016/j.sleep.2022.06.013. [p52]

B. D. Fulcher. Feature-based time-series analysis. In *Feature Engineering for Machine Learning and Data Analytics*. CRC Press, 2018. ISBN 978-1-315-18108-0. [p43]

B. D. Fulcher and N. S. Jones. Highly comparative feature-based time-series classification. *IEEE Transactions on Knowledge and Data Engineering*, 26(12):3026–3037, Dec. 2014. ISSN 1041-4347, 1558-2191, 2326-3865. doi: 10.1109/TKDE.2014.2316504. [p44, 54]

B. D. Fulcher and N. S. Jones. hctsa: A computational framework for automated time-series phenotyping using massive feature extraction. *Cell Systems*, 5(5):527–531.e3, Nov. 2017. ISSN 2405-4712. doi: 10.1016/j.cels.2017.10.001. [p43, 44, 45, 52]

B. D. Fulcher, A. E. Georgieva, C. W. G. Redman, and N. S. Jones. Highly comparative fetal heart rate analysis. In *2012 Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, pages 3135–3138, Aug. 2012. doi: 10.1109/EMBC.2012.6346629. [p44]

B. D. Fulcher, M. A. Little, and N. S. Jones. Highly comparative time-series analysis: The empirical structure of time series and their methods. *Journal of The Royal Society Interface*, 10(83):20130048, June 2013. doi: 10.1098/rsif.2013.0048. [p43, 44, 46, 50, 52, 63]

B. D. Fulcher, C. H. Lubba, S. S. Sethi, and N. S. Jones. A self-organizing, living library of time-series data. *Scientific Data*, 7(1):213, July 2020. ISSN 2052-4463. doi: 10.1038/s41597-020-0553-0. [p44]

B. J. Harris. *Catch22.jl*, 2021. v0.2.1. [p44]

T. Henderson. *Rcatch22: Calculation of 22 CAnonical Time-Series CHaracteristics*, 2021. URL `https://CRAN.R-project.org/package=Rcatch22`. R package version 0.2.3. [p44]

T. Henderson. *normaliseR: Re-Scale Vectors and Time-Series Features*, 2024. URL `https://CRAN.R-project.org/package=normaliseR`. R package version 0.1.2. [p50]

T. Henderson. *correctR: Corrected Test Statistics for Comparing Machine Learning Models on Correlated Samples*, 2025a. URL `https://CRAN.R-project.org/package=correctR`. R package version 0.3.1. [p57]

T. Henderson. *theft: Tools for Handling Extraction of Features from Time Series*, 2025b. URL `https://CRAN.R-project.org/package=theft`. R package version 0.8.2. [p45]

T. Henderson. *theftdlc: Analyse and Interpret Time Series Features*, 2025c. URL `https://CRAN.R-project.org/package=theftdlc`. R package version 0.2.1. [p45]

T. Henderson and B. D. Fulcher. An Empirical Evaluation of Time-Series Feature Sets. In *2021 International Conference on Data Mining Workshops (ICDMW)*, pages 1032–1038, Dec. 2021. doi: 10.1109/ICDMW53433.2021.00134. [p45, 49, 50]

T. Henderson, A. G. Bryant, and B. D. Fulcher. Never a dull moment: Distributional properties as a baseline for time-series classification, 2023. URL `https://arxiv.org/abs/2303.17809`. [p44, 49, 61]

R. Hyndman, Y. Kang, P. Montero-Manso, T. Talagala, E. Wang, Y. Yang, and M. O'Hara-Wild. *tsfeatures: Time Series Feature Extraction*, 2020. URL `https://CRAN.R-project.org/package=tsfeatures`. R package version 1.1.1. [p44]

X. Jiang, S. Srivastava, S. Chatterjee, Y. Yu, J. Handler, P. Zhang, R. Bopardikar, D. Li, Y. Lin, U. Thakore, M. Brundage, G. Holt, C. Komurlu, R. Nagalla, Z. Wang, H. Sun, P. Gao, W. Cheung, J. Gao, Q. Wang, M. Guerard, M. Kazemi, Y. Chen, C. Zhou, S. Lee, N. Laptev, T. Levendovszky, J. Taylor, H. Qian, J. Zhang, A. Shoydokova, T. Singh, C. Zhu, Z. Baz, C. Bergmeir, D. Yu, A. Koylan, K. Jiang, P. Temiyasathit, and E. Yurtbay. Kats, 3 2022. URL `https://github.com/facebookresearch/Kats`. [p45]

I. T. Jolliffe. *Principal Component Analysis*. Springer Series in Statistics. Springer-Verlag, New York, 2002. ISBN 978-0-387-95442-4. doi: 10.1007/b98835. [p52]

L.-J. Kao, C.-C. Chiu, H.-J. Wang, and C. Y. Ko. Prediction of remaining time on site for e-commerce users: A SOM and long short-term memory study. *Journal of Forecasting*, 40(7): 1274–1290, 2021. doi: https://doi.org/10.1002/for.2771. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/for.2771`. [p43]

E. Kodra, S. Chatterjee, and A. R. Ganguly. Exploring Granger causality between global average observed time series of carbon dioxide and temperature. *Theoretical and Applied Climatology*, 104(3):325–335, June 2011. ISSN 1434-4483. doi: 10.1007/s00704-010-0342-3. [p43]

G. Liu, L. Li, L. Zhang, Q. Li, and S. S. Law. Sensor faults classification for SHM systems using deep learning-based method with Tsfresh features. *Smart Materials and Structures*, 29(7):075005, May 2020. ISSN 0964-1726. doi: 10.1088/1361-665X/ab85a6. [p45]

M. Löning, A. Bagnall, S. Ganesh, V. Kazakov, J. Lines, and F. J. Király. sktime: A unified interface for machine learning with time series. *arXiv preprint arXiv:1909.07872*, 2019. [p45]

C. H. Lubba, S. S. Sethi, P. Knaute, S. R. Schultz, B. D. Fulcher, and N. S. Jones. Catch22: CAnonical Time-series CHaracteristics. *Data Mining and Knowledge Discovery*, 33(6):1821–1852, Nov. 2019. ISSN 1573-756X. doi: 10.1007/s10618-019-00647-x. [p44, 65]

M. Markicevic, B. D. Fulcher, C. Lewis, F. Helmchen, M. Rudin, V. Zerbi, and N. Wenderoth. Cortical Excitation:Inhibition Imbalance Causes Abnormal Brain Network Dynamics as Observed in Neurodevelopmental Disorders. *Cerebral Cortex*, 30(9):4922–4937, 2020. ISSN 1047-3211. doi: 10.1093/cercor/bhaa084. [p44]

P. Montero-Manso, G. Athanasopoulos, R. J. Hyndman, and T. S. Talagala. FFORMA: Feature-based forecast model averaging. *International Journal of Forecasting*, 36(1):86–92, Jan. 2020. ISSN 0169-2070. doi: 10.1016/j.ijforecast.2019.02.011. [p64]

C. Nadeau and Y. Bengio. Inference for the generalization error. *Machine Learning*, 52:239, 2003. [p57]

M. O'Hara-Wild, R. Hyndman, and E. Wang. *feasts: Feature Extraction and Statistics for Time Series*, 2021. URL https://CRAN.R-project.org/package=feasts. R package version 0.4.1. [p44]

M. Ojala and G. C. Garriga. Permutation Tests for Studying Classifier Performance. In *2009 Ninth IEEE International Conference on Data Mining*, pages 908–913, Miami Beach, FL, USA, Dec. 2009. IEEE. ISBN 978-1-4244-5242-2. doi: 10.1109/ICDM.2009.108. [p54]

A. Paul, H. McLendon, V. Rally, J. T. Sakata, and S. C. Woolley. Behavioral discrimination and time-series phenotyping of birdsong performance. *PLOS Computational Biology*, 17(4): e1008820, Apr. 2021. ISSN 1553-7358. doi: 10.1371/journal.pcbi.1008820. [p44]

C. W. Tan, C. Bergmeir, F. Petitjean, and G. I. Webb. Time series extrinsic regression. *Data Mining and Knowledge Discovery*, 35(3):1032–1060, May 2021. ISSN 1573-756X. doi: 10.1007/s10618-021-00745-9. [p64]

J. Van Der Donckt, J. Van Der Donckt, E. Deprost, and S. Van Hoecke. tsflex: Flexible time series processing & feature extraction. *SoftwareX*, 17:100971, Jan. 2022. ISSN 2352-7110. doi: 10.1016/j.softx.2021.100971. [p45]

E. Wang, D. Cook, and R. J. Hyndman. A new tidy data structure to support exploration and modeling of temporal data. *Journal of Computational and Graphical Statistics*, 29(3):466–478, 2020. doi: 10.1080/10618600.2019.1695624. URL https://doi.org/10.1080/10618600.2019.1695624. [p46, 48]

M. West, R. Prado, and A. D. Krystal. Evaluation and Comparison of EEG Traces: Latent Structure in Nonstationary Time Series. *Journal of the American Statistical Association*, 94 (446):375–387, June 1999. ISSN 0162-1459. doi: 10.1080/01621459.1999.10474128. [p43]

H. Wickham. Tidy data. *Journal of Statistical Software*, 59(1):1–23, Sept. 2014. ISSN 1548-7660. doi: 10.18637/jss.v059.i10. [p44]

H. Wickham, M. Averick, J. Bryan, W. Chang, L. D. McGowan, R. François, G. Grolemund, A. Hayes, L. Henry, J. Hester, M. Kuhn, T. L. Pedersen, E. Miller, S. M. Bache, K. Müller, J. Ooms, D. Robinson, D. P. Seidel, V. Spinu, K. Takahashi, D. Vaughan, C. Wilke, K. Woo, and H. Yutani. Welcome to the Tidyverse. *Journal of Open Source Software*, 4(43):1686, Nov. 2019. ISSN 2475-9066. doi: 10.21105/joss.01686. [p44]

H. Wickham, R. François, L. Henry, K. Müller, and D. Vaughan. *dplyr: A Grammar of Data Manipulation*, 2023. URL https://CRAN.R-project.org/package=dplyr. R package version 1.1.4. [p58]

Z. Yang, I. A. Abbasi, E. E. Mustafa, S. Ali, and M. Zhang. An anomaly detection algorithm selection service for IoT stream data based on tsfresh tool and genetic algorithm. *Security and Communication Networks*, 2021:6677027, Feb. 2021. ISSN 1939-0114. doi: 10.1155/2021/6677027. [p45]

Trent Henderson
The University of Sydney
School of Physics
Sydney, Australia
ORCiD: 0009-0005-5467-9914
then6675@uni.sydney.edu.au

Ben D. Fulcher
The University of Sydney
School of Physics
Sydney, Australia
ORCiD: 0000-0002-3003-4055
ben.fulcher@sydney.edu.au