cesium: Open-Source Platform for Time-Series Inference

Brett Naul[‡][*], Stéfan van der Walt[‡], Arien Crellin-Quick[‡], Joshua S. Bloom[§][‡], Fernando Pérez[§][‡]

https://youtu.be/ZgHGCfwExw0

Abstract—Inference on time series data is a common requirement in many scientific disciplines and internet of things (IoT) applications, yet there are few resources available to domain scientists to easily, robustly, and repeatably build such complex inference workflows: traditional statistical models of time series are often too rigid to explain complex time domain behavior, while popular machine learning packages require already-featurized dataset inputs. Moreover, the software engineering tasks required to instantiate the computational platform are daunting. `cesium` is an end-to-end time series analysis framework, consisting of a Python library as well as a web front-end interface, that allows researchers to featurize raw data and apply modern machine learning techniques in a simple, reproducible, and extensible way. Users can apply out-of-the-box feature engineering workflows as well as save and replay their own analyses. Any steps taken in the front end can also be exported to a Jupyter notebook, so users can iterate between possible models within the front end and then fine-tune their analysis using the additional capabilities of the back-end library. The open-source packages make us of many use modern Python toolkits, including `xarray`, `dask`, Celery, Flask, and `scikit-learn`.

Index Terms—time series, machine learning, reproducible science

Introduction

From the reading of electroencephalograms (EEGs) to earthquake seismograms to light curves of astronomical variable stars, gleaning insight from time series data has been central to a broad range of scientific disciplines. When simple analytical thresholds or models suffice, technicians and experts can be easily removed from the process of inspection and discovery by employing custom algorithms. But when dynamical systems are not easily modeled (e.g., through physics-based models or standard regression techniques), classification and anomaly detection have traditionally been reserved for the domain expert: digitally recorded data are visually scanned to ascertain the nature of the time variability and find important (perhaps life-threatening) outliers. *Does this person have an irregular heartbeat? What type of supernova occurred in that galaxy?* Even in the presence of sensor noise and intrinsic diversity of the samples, well-trained domain specialists show a remarkable ability to make discerning statements about complex data.

* *Corresponding author: bnaul@berkeley.edu*
‡ *University of California, Berkeley*
§ *Lawrence Berkeley National Laboratory*

In an era when more time series data are being collected than can be visually inspected by domain experts, computational frameworks must necessarily act as human surrogates. Capturing the subtleties that domain experts intuit in time series data (let alone besting the experts) is a non-trivial task. In this respect, machine learning has already been used to great success in several disciplines, including text classification, image retrieval, segmentation of remote sensing data, internet traffic classification, video analysis, and classification of medical data. Even if the results are similar, some obvious advantages over human involvement are that machine learning algorithms are tunable, repeatable, and deterministic. A computational framework built with elasticity can scale, whereas experts (and even crowdsourcing) cannot.

Despite the importance of time series in scientific research, there are few resources available that allow domain scientists to easily build robust computational inference workflows for their own time series data, let alone data gathered more broadly in their field. The difficulties involved in constructing such a framework can often greatly outweigh those of analyzing the data itself:

> It may be surprising to the academic community to know that only a tiny fraction of the code in many machine learning systems is actually doing "machine learning"...a mature system might end up being (at most) 5% machine learning code and (at least) 95% glue code. [SHG+14]

Even if a domain scientist works closely with machine learning experts, the software engineering requirements can be daunting. It is our opinion that being a modern data-driven scientist should not require an army of software engineers, machine learning experts, statisticians and production operators. `cesium` [cT16] was created to allow domain experts to focus on the inference questions at hand rather than needing to assemble a complete engineering project.

The analysis workflow of `cesium` can be used in two forms: a web front end which allows researchers to upload their data, perform analyses, and visualize their models all within the browser; and a Python library which exposes more flexible interfaces to the same analysis tools. The web front end is designed to handle many of the more cumbersome aspects of machine learning analysis, including data uploading and management, scaling of computational resources, and tracking of results from previous experiments. The Python library is used within the web back end for the main steps of the analysis workflow: extracting features from raw time series, building models from these features, and

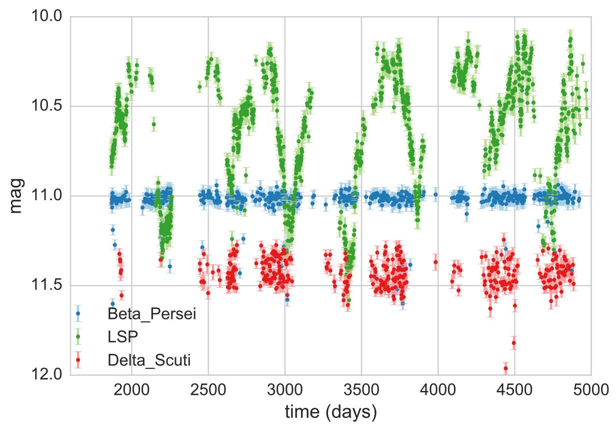


Fig. 1: *Typical data for a classification task on variable stars from the All Sky Automated Survey; shown are flux measurements for three stars irregularly sampled in time [RSM+12].*

generating predictions. The library also supplies data structures for storing time series (including support for irregularly-sampled time series and measurement errors), features, and other relevant metadata.

In the next section, we describe a few motivating examples of scientific time series analysis problems. The subsequent sections describe in detail the `cesium` library and web front end, including the different pieces of functionality provided and various design questions and decisions that arose during the development process. Finally, we present an end-to-end analysis of an EEG seizure dataset, first using the Python library and then via the web front end.

Example time series machine learning problems

`cesium` was designed with several time series inference use cases across various scientific disciplines in mind.

1) **Astronomical time series classification.** Beginning in 2020, the Large Synoptic Survey Telescope (LSST) will survey the entire night's sky every few days producing high-quality time series data on approximately 800 million transient events and sources with variable brightness (Figure 1 depicts the brightness of several types of star over the course of several years) [AAA+09]. Much of the best science in the time domain (e.g., the discovery of the accelerating universe and dark energy using Type Ia supernovae [PAG+99], [RFC+98]) consists of first identifying possible phenomena of interest using broad data mining approaches and following up by collecting more detailed data using other, more precise observational tools. For many transient events, the time scale during which observations can be collected can be on the order of days or hours. Not knowing which of the millions of variable sources to examine more closely with larger telescopes and specialized instruments is tantamount to not having discovered those sources at all. Discoveries must be identified quickly or in real time so that informed decisions can be made about how best to allocate additional observational resources.

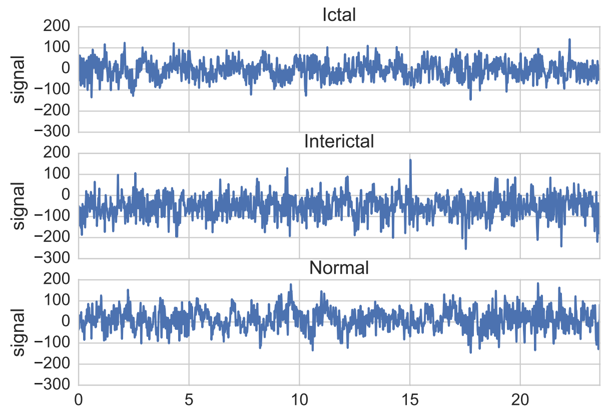


Fig. 2: *EEG signals from patients with epilepsy [ALM+01].*

2) **Neuroscience time series classification.** that might need to be classified in order to make treatment decisions. Neuroscience experiments now produce vast amounts of time series data that can have entirely different structures and spatial/temporal resolutions, depending on the recording technique. Figure 2 shows an example of different types of EEG signals The neuroscience community is turning to the use of large-scale machine learning tools to extract insight from large, complex datasets [LCL+07]. However, the community lacks tools to validate and compare data analysis approaches in a robust, efficient and reproducible manner: even recent expert reviews on the matter leave many of these critical methodological questions open for the user to explore in an ad hoc way and with little principled guidance [PG07].

3) **Earthquake detection, characterization and warning.** Earthquake early warning (EEW) systems are currently in operation in Japan, Mexico, Turkey, Taiwan and Romania [AGKB09] and are under development in the US [BAH+11]. These systems have employed sophisticated remote sensors, real-time connectivity to major broadcast outlets (such as TV and radio), and have a growing resumé of successful rapid assessment of threat levels to populations and industry. Traditionally these warning systems trigger from data obtained by high-quality seismic networks with sensors placed every ~10 km. Today, however, accelerometers are embedded in many consumer electronics including computers and smartphones. There is tremendous potential to improve earthquake detection methods using streaming classification analysis both using traditional network data and also harnessing massive data from consumer electronics.

Simple and reproducible workflows

In recent years, there has been rapid growth in the availability of open-source tools that implement a wide variety of machine learning algorithms: packages within the R [T+13] and Python programming languages [PVG+11], standalone Java-based packages such as Moa [BHKP10] and Weka [HFH+09], and online webservices such as the Google Prediction API, to name a few. To a domain scientist that does not have formal training in machine learning, however, the availability of such packages is both a blessing and a curse. On one hand, most machine learning algorithms are now widely accessible to all researchers. At the same time, these algorithms tend to be black boxes with potentially many enigmatic knobs to turn. A domain scientist may rightfully ask just which of the many algorithms to use, which parameters to tune, and what the results actually mean.

The goal of `cesium` is to simplify the analysis pipeline so that scientists can spend less time solving technical computing problems and more time answering scientific questions. `cesium` provides a library of feature extraction techniques inspired by analyses from many scientific disciplines, as well as a surrounding framework for building and analyzing models from the resulting feature information using `scikit-learn` (or potentially other machine learning tools).

By recording the inputs, parameters, and outputs of previous experiments, *cesium*‘ allows researchers to answer new questions that arise out of previous lines of inquiry. Saved `cesium` workflows can be applied to new data as it arrives and shared with collaborators or published so that others may apply the same beginning-to-end analysis for their own data.

For advanced users or users who wish to delve into the source code corresponding to a workflow produced through the `cesium` web front end, we are implementing the ability to produce a Jupyter notebook [PG07] from a saved workflow with a single click. While our goal is to have the front end to be as robust and flexible as possible, ultimately there will always be special cases where an analysis requires tools which have not been anticipated, or where the debugging process requires a more detailed look at the intermediate stages of the analysis. Exporting a workflow to a runnable notebook provides a more detailed, lower-level look at how the analysis is being performed, and can also allow the user to reuse certain steps from a given analysis within any other Python program.

`cesium` library

The first half of the `cesium` framework is the back-end Python-based library, aimed at addressing the following uses cases:

1) A domain scientist who is comfortable with programming but is **unfamiliar with time series analysis or machine learning**.
2) A scientist who is experienced with time series analysis but is looking for **new features** that can better capture patterns within their data.
3) A user of the `cesium` web front end who realizes they require additional functionality and wishes to add additional stages to their workflow.

Our framework primarily implements "feature-based methods", wherein the raw input time series data is used to compute "features" that compactly capture the complexity of the signal space within a lower-dimensional feature space. Standard machine learning approaches (such as random forests [Bre01] and support vector machines [SV99]) may then be used for supervised classification or regression.

`cesium` allows users to select from a large library of features, including both general time series features and domain-specific features drawn from various scientific disciplines. Some specific advantages of the `cesium` featurization process include:

- Support for both regularly and irregularly sampled time series.

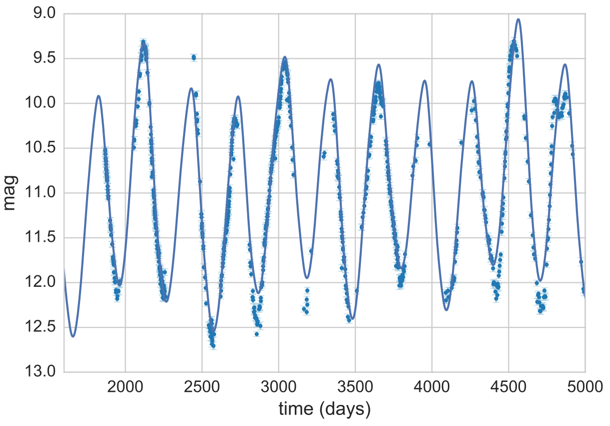


Fig. 3: *Fitted multi-harmonic Lomb-Scargle model for a light curve from a periodic Mira-class star.* `cesium` *automatically generates numerous features based on Lomb-Scargle periodogram analysis.*

- Ability to incorporate measurement errors, which can be provided for each data point of each time series (if applicable).
- Support for multi-channel data, in which case features are computed separately for each dimension of the input data.

Example features

Some `cesium` features are extremely simple and intuitive: summary statistics such as maximum/minimum values, mean/median values, and standard deviation or median absolute deviation are a few such examples. Other features involve measurement errors if they are available: for example, a mean and standard deviation that is weighted by measurement errors allows noisy data with large outliers to be modeled more precisely.

Other more involved features could be the estimated parameters for various fitted statistical models: Figure 3 shows a multi-frequency, multi-harmonic Lomb-Scargle model that describes the rich periodic behavior in an example time series [Lom76], [Sca82]. The Lomb-Scargle method is one approach for generalizing the process of Fourier analysis of frequency spectra to the case of irregularly sampled time series. In particular, a time series is modeled as a superposition of periodic functions

$$\tilde{y}(t) = \sum_{k=1}^{m} \sum_{l=1}^{n} A_{kl} \cos k\omega_l t + B_{kl} \sin k\omega_l t,$$

where the parameters A_{kl}, B_{kl}, and ω_l are selected via non-convex optimization to minimize the residual sum of squares (weighted by measurement errors if applicable). The estimated periods, amplitudes, phases, goodness-of-fits, and power spectrum can then be used as features which broadly characterize the periodicity of the input time series.

Usage overview

Here we provide a few examples of the main `cesium` API components that would be used in a typical analysis task. A workflow will typically consist of three steps: featurization, model building, and prediction on new data. The majority of `cesium` functionality is contained within the `cesium.featurize` module; the `cesium.build_model` and `cesium.predict` modules primarily provide interfaces between sets of feature data, which contain both feature data and a variety of metadata about the input time series, and machine learning models from `scikit-learn` [PVG+11], which require dense, rectangular input data. Note that, as `cesium` is under active development, some of the following details are subject to change.

The featurization step is performed using one of two main functions:

- `featurize_time_series(times, values, errors, ...)`
    - Takes in data that is already present in memory and computes the requested features (passed in as string feature names) for each time series.
    - Features can be computed in parallel across workers via Celery, a Python distributed task queue [Sol14], or locally in serial.
    - Class labels/regression targets and metadata/features with known values are passed in and stored in the output dataset.
    - Additional feature functions can be passed in as `custom_functions`.
- `featurize_data_files(uris, ...)`,
    - Takes in a list of file paths or URIs and dispatches featurization tasks to be computed in parallel via Celery.
    - Data is loaded only remotely by the workers rather than being copied, so this approach should be preferred for very large input datasets.
    - Features, metadata, and custom feature functions are passed in the same way as `featurize_data_files`.

The output of both functions is a `Dataset` object from the `xarray` library [Hoy15], which will also be referred to here as a "feature set" (more about `xarray` is given in the next section). The feature set stores the computed feature values for each function (indexed by channel, if the input data is multi-channel), as well as time series filenames or labels, class labels or regression targets, and other arbitrary metadata to be used in building a statistical model.

The `build_model` contains tools meant to to simplify the process of building `sckit-learn` models from (non-rectangular) feature set data:

- `model_from_featureset(featureset, ...)`
    - Returns a fitted `scikit-learn` model based on the input feature data.
    - A pre-initialized (but untrained) model can be passed in, or the model type can be passed in as a string.
    - Model parameters can be passed in as fixed values, or as ranges of values from which to select via cross-validation.

Analogous helper functions for prediction are available in the `predict` module:

- `model_predictions(featureset, model, ...)`
    - Generates predictions from a feature set outputted by `featurize_time_series` or `featurize_data_files`.
- `predict_data_files(file_paths, model, ...)`
    - Like `featurize_data_files`, generate predictions for time series which have not yet been featurized by dispatching featurization tasks to Celery workers and then passing the resulting featureset to `model_predictions`.

After a model is initially trained or predictions have been made, new models can be trained with more features or uninformative features can be removed until the result is satisfactory.

Implementation details

`cesium` is implemented in Python, along with some C code (integrated via Cython) for especially computationally-intensive feature calculations. Our library also relies upon many other open source Python projects, including `scikit-learn`, `pandas`, `xarray`, and `dask`. As the first two choices are somewhat obvious, here we briefly describe the roles of the latter two libraries.

As mentioned above, feature data generated by `cesium` is returned as a `Dataset` object from the `xarray` package, which according to the documentation "resembles an in-memory representation of a NetCDF file, and consists of variables, coordinates and attributes which together form a self describing dataset". A `Dataset` allows multi-channel feature data to be faithfully represented in memory as a multidimensional array so that the effects of each feature (across all channels) or channel (across all features) can be evaluated directly, while also storing metadata and features that are not channel-specific. Storing feature outputs in NetCDF format allows for faster and more space-efficient serialization and loading of results (as compared to a text-based format).

The `dask` library provides a wide range of tools for organizing computational full process of exporting tasks. `cesium` makes use of only one small component: within `dask`, tasks are organized as a directed acyclic graph (DAG), with the results of some tasks serving as the inputs to others. Tasks can then be computed in an efficient order by `dask`'s scheduler. Within `cesium`, many features rely on other features as inputs, so internally we represent our computations as `dask` graphs in order to minimize redundant computations and peak memory usage. Part of an example DAG involving the Lomb-Scargle periodogram is depicted in Figure 4: circles represent functions, and rectangles the inputs/outputs of the various steps. In addition to the built-in features, custom feature functions passed in directly by the user can similarly make use of the internal `dask` representation so that built-in features can be reused for the evaluation of user-specified functions.

Web front end

The `cesium` front end provides web-based access to time series analysis, addressing three common use cases:

1) A scientist needs to perform time series analysis, but is **unfamiliar with programming** and library usage.
2) A group of scientists want to **collaboratively explore** different methods for time-series analysis.
3) A scientist is unfamiliar with time-series analysis, and wants to **learn** how to apply various methods to their data, using **industry best practices**.

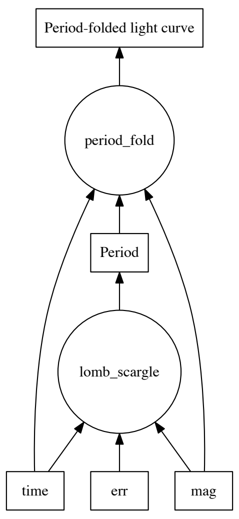


Fig. 4: *Example of a directed feature computation graph using* `dask`.

The front-end system (together with its deployed back end), offers the following features:

- Distributed, parallelized fitting of machine learning models.
- Isolated[1], cloud-based execution of user-uploaded featurization code.
- Visualization and analysis of results.
- Tracking of an entire exploratory workflow from start-to-finish for reproducibility (in progress).
- Downloads of Jupyter notebooks to replicate analyses[2].

Implementation

The `cesium` web front end consists of several components:

- A Python-based Flask [Ron15] server which provides a REST API for managing datasets and launching featurization, model-building, and prediction tasks.
- A JavaScript-based web interface implemented using React [Gac15b] and Redux [Gac15a] to display results to users.
- A custom WebSocket communication system (which we informally call *message flow*) that notifies the front end when back-end tasks complete.

While the deployment details of the web front end are beyond the scope of this paper, it should be noted that it was designed with scalability in mind. The overarching design principle is to connect several small components, each performing only one, simple task. An NGINX proxy exposes a pool of WebSocket and Web Server Gateway Interface (WSGI) servers to the user. This gives us the flexibility to choose the best implementation of each. Communications between WSGI servers and WebSocket servers happen through a ZeroMq XPub-XSub (multi-publisher publisher-subscriber) pipeline [Hin13], but could be replaced with any other broker, e.g., RabbitMQ [VW12]. The "message flow" paradigm adds WebSocket support to any Python WSGI server (Flask, Django[3], Pylons, etc.), and allows scaling up as demand increases. It also implement trivially modern data flow models such as Flux/Redux, where information always flows in one direction: from front end to back end via HTTP (Hypertext Transfer Protocol) calls, and from back end to front end via WebSocket communication.

Computational Scalability

In many fields, the volumes of available time series data can be immense. `cesium` includes features to help parallelize and scale an analysis from a single system to a large cluster.

Both the back-end library and web front end make use of Celery [Sol14] for distributing featurization tasks to multiple workers; this could be used for anything from automatically utilizing all the available cores of a single machine, to assigning jobs across a large cluster. Similarly, both parts of the `cesium` framework include support for various distributed filesystems, so that analyses can be performed without copying the entire dataset into a centralized location.

While the `cesium` library is written in pure Python, the overhead of the featurization tasks is minimal; the majority of the work is done by the feature code itself. Most of the built-in features are based on high-performance `numpy` functions; others are written in pure C with interfaces in Cython. The use of `dask` graphs to eliminate redundant computations also serves to minimize memory footprint and reduce computation times.

Automated testing and documentation

Because the back-end library and web front end are developed in separate GitHub repositories, the connections between the two somewhat complicate the continuous integration testing setup. Both repositories are integrated with Travis CI for automatic testing of all branches and pull requests; in addition, any new pushes to `cesium/master` trigger a set of tests of the front end using the new version of the back-end library, with any failures being reported but not causing the `cesium` build to fail (the reasoning being that the back-end library API should be the "ground truth", so any updates represent a required change to the front end, not a bug *per se*).

Documentation for the back-end API is automatically generated in ReStructured Text format via `numpydoc`; the result is combined with the rest of our documentation and rendered as HTML using `sphinx`. Code examples (without output) are stored in the repository in Markdown format as opposed to Jupyter notebooks since this format is better suited to version control. During the doc-build process, the Markdown is converted to Jupyter notebook format using `notedown`, then executed using `nbconvert` and converted back to Markdown (with outputs included), to be finally rendered by `sphinx`. This allows the code examples to be saved in a human-readable and version control-friendly format while still allowing the user to execute the code themselves via a downloadable notebook.

1. Isolation is currently provided by limiting the user to non-privileged access inside a Docker [Mer14] container.

2. Our current implementation of the front end includes the ability to track all of a user's actions in order to produce a notebook version, but the full process of generating the notebook is still a work in progress.

3. At PyCon2016, Andrew Godwin presented a similar solution for Django called "channels". The work described here happened before we became aware of Andrew's, and generalizes beyond Django to, e.g., Flask, the web framework we use.

Example EEG dataset analysis

In this example we compare various techniques for epilepsy detection using a classic EEG time series dataset from Andrzejak et al. [ALM+01]. The raw data are separated into five classes: Z, O, N, F, and S; we consider a three-class classification problem of distinguishing normal (Z, O), interictal (N, F), and ictal (S) signals. We show how to perform the same analysis using both the back-end Python library and the web front end.

Python library

First, we load the data and inspect a representative time series from each class: Figure 2 shows one time series from each of the three classes, after the time series are loaded from `cesium.datasets.andrzejak`.

Once the data is loaded, we can generate features for each time series using the `cesium.featurize` module. The `featurize` module includes many built-in choices of features which can be applied for any type of time series data; here we've chosen a few generic features that do not have any special biological significance.

If Celery is running, the time series will automatically be split among the available workers and featurized in parallel; setting `use_celery=False` will cause the time series to be featurized serially.

```
from cesium import featurize

features_to_use = ['amplitude', 'maximum',
                   'max_slope', 'median',
                   'median_absolute_deviation',
                   'percent_beyond_1_std',
                   'percent_close_to_median',
                   'minimum', 'skew', 'std',
                   'weighted_average']
fset_cesium = featurize.featurize_time_series(
                  times=eeg["times"],
                  values=eeg["measurements"],
                  errors=None,
                  features_to_use=features_to_use,
                  targets=eeg["classes"])
```

```
<xarray.Dataset>
Dimensions:   (channel: 1, name: 500)
Coordinates:
* channel    (channel) int64 0
* name       (name) int64 0 1 ...
  target     (name) object 'Normal' 'Normal' ...
Data variables:
  minimum    (name, channel) float64 -146.0 -254.0 ...
  amplitude  (name, channel) float64 143.5 211.5 ...
  ...
```

The resulting `Dataset` contains all the feature information needed to train a machine learning model: feature values are stored as data variables, and the time series index/class label are stored as coordinates (a `channel` coordinate will also be used later for multi-channel data).

Custom feature functions not built into `cesium` may be passed in using the `custom_functions` keyword, either as a dictionary `{feature_name: function}`, or as a `dask` graph. Functions should take three arrays `times`, `measurements`, `errors` as inputs; details can be found in the `cesium.featurize` documentation. Here we compute five standard features for EEG analysis suggested by Guo et al. [GRD+11]:

```
import numpy as np, scipy.stats

def mean_signal(t, m, e):
    return np.mean(m)

def std_signal(t, m, e):
    return np.std(m)

def mean_square_signal(t, m, e):
    return np.mean(m ** 2)

def abs_diffs_signal(t, m, e):
    return np.sum(np.abs(np.diff(m)))

def skew_signal(t, m, e):
    return scipy.stats.skew(m)
```

Now we pass the desired feature functions as a dictionary via the `custom_functions` keyword argument (functions can also be passed in as a list or a `dask` graph).

```
guo_features = {
    'mean': mean_signal,
    'std': std_signal,
    'mean2': mean_square_signal,
    'abs_diffs': abs_diffs_signal,
    'skew': skew_signal
}
fset_guo = featurize.featurize_time_series(
               times=eeg["times"],
               values=eeg["measurements"],
               errors=None, targets=eeg["classes"],
               features_to_use=guo_features.keys(),
               custom_functions=guo_features)
```

```
<xarray.Dataset>
Dimensions:     (channel: 1, name: 500)
Coordinates:
* channel    (channel) int64 0
* name       (name) int64 0 1 ...
  target     (name) object 'Normal' 'Normal' ...
Data variables:
  abs_diffs  (name, channel) float64 4695.2 6112.6 ...
  mean       (name, channel) float64 -4.132 -52.44 ...
  ...
```

The EEG time series considered here consist of univariate signal measurements along a uniform time grid. But `featurize_time_series` also accepts multi-channel data. To demonstrate this, we will decompose each signal into five frequency bands using a discrete wavelet transform as suggested by Subasi [Sub07], and then featurize each band separately using the five functions from above.

```
import pywt

eeg["dwts"] = [pywt.wavedec(m, pywt.Wavelet('db1'),
                            level=4)
               for m in eeg["measurements"]]
fset_dwt = featurize.featurize_time_series(
               times=None, values=eeg["dwts"], errors=None,
               features_to_use=guo_features.keys(),
               targets=eeg["classes"],
               custom_functions=guo_features)
```

```
<xarray.Dataset>
Dimensions:     (channel: 5, name: 500)
Coordinates:
* channel    (channel) int64 0 1 ...
* name       (name) int64 0 1 ...
  target     (name) object 'Normal' 'Normal' ...
Data variables:
  abs_diffs  (name, channel) float64 25131 18069 ...
  skew       (name, channel) float64 -0.0433 0.06578 ..
  ...
```

The output feature set has the same form as before, except now the `channel` coordinate is used to index the features by the corresponding frequency band. The functions in `cesium.build_model` and `cesium.predict` all accept feature sets from single- or multi-channel data, so no additional steps are required to train models or make predictions for multichannel feature sets using the `cesium` library.

Model building in `cesium` is handled by the `model_from_featureset` function in the `cesium.build_model` module. The feature set output by `featurize_time_series` contains both the feature and target information needed to train a model; `model_from_featureset` is simply a wrapper that calls the `fit` method of a given `scikit-learn` model with the appropriate inputs. In the case of multichannel features, it also handles reshaping the feature set into a (rectangular) form that is compatible with `scikit-learn`.

For this example, we test a random forest classifier for the built-in `cesium` features, and a 3-nearest neighbors classifier for the others, as in [GRD+11].

```
from cesium.build_model import model_from_featureset
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.cross_validation import train_test_split

train, test = train_test_split(500)

rfc_param_grid = {'n_estimators': [8, 32, 128, 512]}
model_cesium = model_from_featureset(
                    fset_cesium.isel(name=train),
                    RandomForestClassifier(),
                    params_to_optimize=rfc_param_grid)

knn_param_grid = {'n_neighbors': [1, 2, 3, 4]}
model_guo = model_from_featureset(
                fset_guo.isel(name=train),
                KNeighborsClassifier(),
                params_to_optimize=knn_param_grid)
model_dwt = model_from_featureset(
                fset_dwt.isel(name=train),
                KNeighborsClassifier(),
                params_to_optimize=knn_param_grid)
```

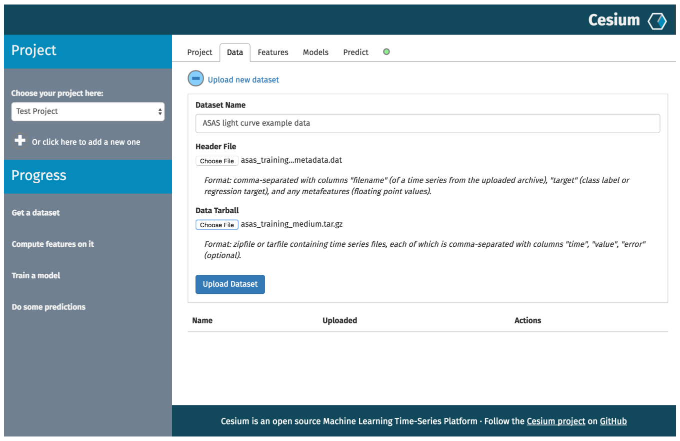
Making predictions for new time series based on these models follows the same pattern: first the time series are featurized using `featurize_timeseries` and then predictions are made based on these features using `predict.model_predictions`,

```
from cesium.predict import model_predictions
preds_cesium = model_predictions(
                    fset_cesium, model_cesium,
                    return_probs=False)
preds_guo = model_predictions(fset_guo, model_guo,
                    return_probs=False)
preds_dwt = model_predictions(fset_dwt, model_dwt,
                    return_probs=False)
```

And finally, checking the accuracy of our various models, we find:

```
Builtin: train acc=100.00%, test acc=83.20%
Guo et al.: train acc=90.93%, test acc=84.80%
Wavelets: train acc=100.00%, test acc=95.20%
```

The workflow presented here is intentionally simplistic and omits many important steps such as feature selection, model



Dataset Name: ASAS light curve example data

Header File: asas_training...metadata.dat

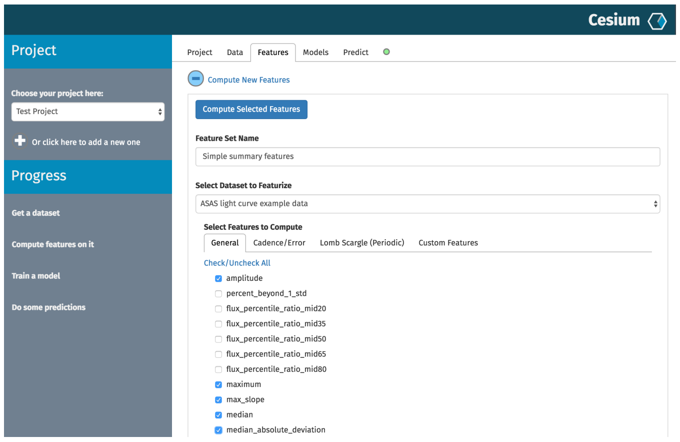
Format: comma-separated with columns "filename" (of a time series from the uploaded archive), "target" (class label or regression target), and any metafeatures (floating point values).

Data Tarball: asas_training_medium.tar.gz

Format: zipfile or tarfile containing time series files, each of which is comma-separated with columns "time", "value", "error" (optional).

Cesium is an open source Machine Learning Time-Series Platform · Follow the Cesium project on GitHub

Fig. 5: *"Data" tab*



Project: Test Project

Feature Set Name: Simple summary features

Select Dataset to Featurize: ASAS light curve example data

- ✓ amplitude
- ☐ percent_beyond_1_std
- ☐ flux_percentile_ratio_mid20
- ☐ flux_percentile_ratio_mid35
- ☐ flux_percentile_ratio_mid50
- ☐ flux_percentile_ratio_mid65
- ☐ flux_percentile_ratio_mid80
- ✓ maximum
- ✓ max_slope
- ✓ median
- ✓ median_absolute_deviation

Fig. 6: *"Featurize" tab*

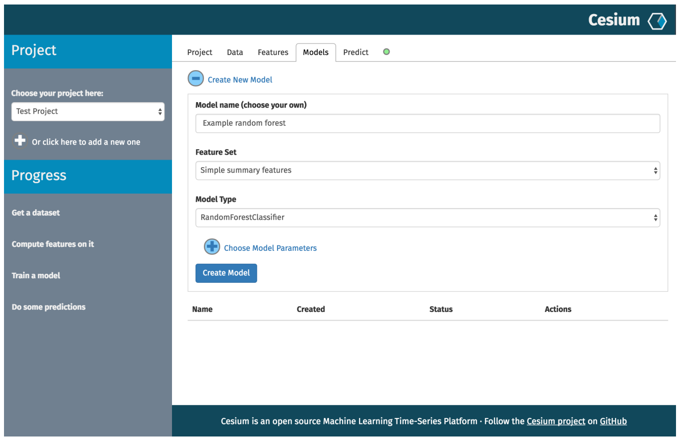
parameter selection, etc., which may all be incorporated just as they would for any other `scikit-learn` analysis. But with essentially three function calls (`featurize_time_series`, `model_from_featureset`, and `model_predictions`), we are able to build a model from a set of time series and make predictions on new, unlabeled data. In the next section we introduce the web front end for `cesium` and describe how the same analysis can be performed in a browser with no setup or coding required.

Web front end

Here we briefly demonstrate how the above analysis could be conducted using only the web front end. Note that the user interface presented here is a preliminary version and is undergoing frequent updates and additions. The basic workflow follows the same *featurize—build model—predict* pattern. First, data is uploaded as in Figure 5. Features are selected from available built-in functions as in Figure 6, or may be computed from user-uploaded Python code which is securely executed within a Docker container. Once features have been extracted, models can be created as in Figure 7, and finally predictions can be made as in Figure 8. Currently the options for exploring feature importance and model accuracy are limited, but this is again an area of active development.

Future work

The `cesium` project is under active development. Some of our upcoming goals include:



Choose your project here: Test Project
Or click here to add a new one

Model name (choose your own): Example random forest

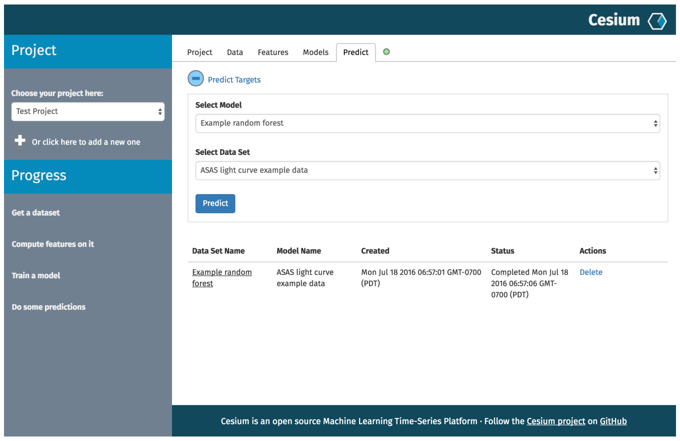
Feature Set: Simple summary features

Model Type: RandomForestClassifier

Name	Created	Status	Actions

Cesium is an open source Machine Learning Time-Series Platform · Follow the Cesium project on GitHub

Fig. 7: "Build Model" tab



Choose your project here: Test Project
Or click here to add a new one

Select Model: Example random forest

Select Data Set: ASAS light curve example data

Data Set Name	Model Name	Created	Status	Actions
Example random forest	ASAS light curve example data	Mon Jul 18 2016 06:57:01 GMT-0700 (PDT)	Completed Mon Jul 18 2016 06:57:06 GMT-0700 (PDT)	Delete

Cesium is an open source Machine Learning Time-Series Platform · Follow the Cesium project on GitHub

Fig. 8: "Predict" tab

- Full support for exporting Jupyter notebooks from any workflow created within the web front end.
- Additional features from other scientific disciplines (currently the majority of available features are taken from applications in astronomy).
- Improved web front end user interface with more tools for visualizing and exploring a user's raw data, feature values, and model outputs.
- More tools to streamline the process of iteratively exploring new models based on results of previous experiments.
- Better support for sharing data and results among teams.
- Extension to unsupervised problems.

Conclusion

The `cesium` framework provides tools that allow anyone from machine learning specialists to domain experts without any machine learning experience to rapidly prototype explanatory models for their time series data and generate predictions for new, unlabeled data. Aside from the applications to time domain informatics, our project has several aspects which are relevant to the broader scientific Python community.

First, the dual nature of the project (Python back end vs. web front end) presents both unique challenges and interesting opportunities in striking a balance between accessibility and flexibility of the two components. Second, the `cesium` project places a strong emphasis on reproducible workflows: all actions performed within the web front end are logged and can be easily exported to a Jupyter notebook that exactly reproduces the steps of the analysis. Finally, the scope of our project is simultaneously both narrow (time series analysis) and broad (numerous distinct scientific disciplines), so determining how much domain-specific functionality to include is an ongoing challenge.

References

[AAA+09] Paul A Abell, Julius Allison, Scott F Anderson, John R Andrew, J Roger P Angel, Lee Armus, David Arnett, SJ Asztalos, Tim S Axelrod, Stephen Bailey, et al. Lsst science book, version 2.0. *arXiv preprint arXiv:0912.0201*, 2009.

[AGKB09] Richard M Allen, Paolo Gasparini, Osamu Kamigaichi, and Maren Böse. The status of earthquake early warning around the world: An introductory overview. *Seismological Research Letters*, 80(5):682–693, 2009.

[ALM+01] Ralph G. Andrzejak, Klaus Lehnertz, Florian Mormann, Christoph Rieke, Peter David, and Christian 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*, 64(6):061907, 2001. `doi:10.1103/PhysRevE.64.061907`.

[BAH+11] Holly M Brown, Richard M Allen, Margaret Hellweg, Oleg Khainovski, Douglas Neuhauser, and Adeline Souf. Development of the elarms methodology for earthquake early warning: Real-time application in california and offline testing in japan. *Soil Dynamics and Earthquake Engineering*, 31(2):188–200, 2011.

[BHKP10] Albert Bifet, Geoff Holmes, Richard Kirkby, and Bernhard Pfahringer. Moa: Massive online analysis. *Journal of Machine Learning Research*, 11(May):1601–1604, 2010.

[Bre01] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[cT16] cesium Team. *cesium: Open-Source Machine Learning for Time Series Analysis*. 2016. URL: http://cesium.ml.

[Gac15a] Cory Gackenheimer. Introducing flux: An application architecture for react. In *Introduction to React*, pages 87–106. Springer, 2015.

[Gac15b] Cory Gackenheimer. What is react? In *Introduction to React*, pages 1–20. Springer, 2015.

[GRD+11] Ling Guo, Daniel Rivero, Julián Dorado, Cristian R. Munteanu, and Alejandro Pazos. Automatic feature extraction using genetic programming: An application to epileptic EEG classification. *Expert Systems with Applications*, 38(8):10425–10436, 2011. `doi:10.1016/j.eswa.2011.02.118`.

[HFH+09] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[Hin13] Pieter Hintjens. *ZeroMQ: Messaging for Many Applications*. " O'Reilly Media, Inc.", 2013.

[Hoy15] S Hoyer. xray: ND labeled arrays and datasets in Python. 2015. URL: http://github.com/xray/xray.

[LCL+07] Fabien Lotte, Marco Congedo, Anatole Lécuyer, Fabrice Lamarche, and Bruno Arnaldi. A review of classification algorithms for EEG-based brain–computer interfaces. *Journal of neural engineering*, 4(2):R1, 2007.

[Lom76] Nicholas R Lomb. Least-squares frequency analysis of unequally spaced data. *Astrophysics and space science*, 39(2):447–462, 1976.

[Mer14] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[PAG+99] Saul Perlmutter, G Aldering, G Goldhaber, RA Knop, P Nugent, PG Castro, S Deustua, S Fabbro, A Goobar, DE Groom, et al. Measurements of ω and λ from 42 high-redshift supernovae. *The Astrophysical Journal*, 517(2):565, 1999.

[PG07] Fernando Pérez and Brian E Granger. IPython: a system for interactive scientific computing. *Computing in Science & Engineering*, 9(3):21–29, 2007.

[PVG+11] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12(Oct):2825–2830, 2011.

[RFC$^+$98] Adam G Riess, Alexei V Filippenko, Peter Challis, Alejandro Clocchiatti, Alan Diercks, Peter M Garnavich, Ron L Gilliland, Craig J Hogan, Saurabh Jha, Robert P Kirshner, et al. Observational evidence from supernovae for an accelerating universe and a cosmological constant. *The Astronomical Journal*, 116(3):1009, 1998.

[Ron15] Armin Ronacher. Flask (A Python Microframework), 2015.

[RSM$^+$12] Joseph W Richards, Dan L Starr, Adam A Miller, Joshua S Bloom, Nathaniel R Butler, Henrik Brink, and Arien Crellin-Quick. Construction of a calibrated probabilistic classification catalog: Application to 50k variable sources in the all-sky automated survey. *The Astrophysical Journal Supplement Series*, 203(2):32, 2012.

[Sca82] Jeffrey D Scargle. Studies in astronomical time series analysis. ii-statistical aspects of spectral analysis of unevenly spaced data. *The Astrophysical Journal*, 263:835–853, 1982.

[SHG$^+$14] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, and Michael Young. Machine learning: The high interest credit card of technical debt. In *SE4ML: Software Engineering for Machine Learning (NIPS 2014 Workshop)*, 2014.

[Sol14] Ask Solem. Celery: Distributed task queue, 2014. URL: http://celeryproject.org.

[Sub07] Abdulhamit Subasi. EEG signal classification using wavelet feature extraction and a mixture of expert model. *Expert Systems with Applications*, 32(4):1084–1093, 2007.

[SV99] Johan AK Suykens and Joos Vandewalle. Least squares support vector machine classifiers. *Neural processing letters*, 9(3):293–300, 1999.

[T$^+$13] R Core Team et al. R: A language and environment for statistical computing. 2013.

[VW12] Alvaro Videla and Jason JW Williams. *RabbitMQ in action*. Manning, 2012.