parsnip

Release 1.3.1

Kyle Boone

Sep 16, 2022
1 About

Index
ParSNIP is a package for learning generative models of transient light curves. This code has many applications including classification of transients, cosmological distance estimation, and identifying novel transients.

1.1 Installation

ParSNIP requires Python 3.6+ and depends on the following Python packages:

- astropy
- extinction
- lcdata
- lightgbm
- matplotlib
- numpy
- scipy
- PyTorch
- scikit-learn
- tqdm

1.1.1 Install using pip (recommended)

ParSNIP is available on PyPI. To install the latest release:

```bash
pip install astro-parsnip
```
1.1.2 Install development version

The ParSNIP source code can be found on github.

To install it:

```
git clone git://github.com/kboone/parsnip
cd parsnip
pip install -e .
```

1.2 Usage

1.2.1 Overview

ParSNIP is a generative model of astronomical transient light curves. It is designed to work with light curves in 
\texttt{sncosmo} format using the \texttt{lcdata} package to handle large datasets. See the \texttt{lcdata} documentation for details on how
to download or ingest different datasets.

1.2.2 Training a model

ParSNIP provides a built-in script called \texttt{parsnip\_train} that can be used to train a model on an \texttt{lcdata} dataset. It
takes as input the path that the model will be saved to along with a list of paths to datasets. For example:

```
$ parsnip\_train ./model.pt ./dataset\_1.h5 ./dataset\_2.h5
```

will train a model named \texttt{model.pt} using the datasets \texttt{dataset\_1.h5} and \texttt{dataset\_2.h5}.

1.2.3 Generating predictions

The \texttt{parsnip\_predict} script can be used to generate predictions given an \texttt{lcdata} dataset and a pretrained ParSNIP model. To run it:

```
$ parsnip\_predict ./predictions.h5 ./model.h5 ./dataset.h5
```

will generate predictions to the file named \texttt{predictions.h5} using the dataset \texttt{dataset.h5} and the model \texttt{model.h5}.

1.2.4 Loading a dataset in Python

ParSNIP is designed to work with \texttt{lcdata} datasets. \texttt{lcdata} datasets are guaranteed to be in a specific format, but
they may include instrument-specific quirks, light curves that are not compatible with ParSNIP, or metadata in unusual
formats (e.g. PLAsTiCC types are random integers). ParSNIP includes tools to clean up datasets from a range of
different surveys and reject invalid light curves. Given an \texttt{lcdata} dataset, this can be done with:

```
>>> dataset = parsnip.parse_dataset(raw_dataset, kind='ps1')
```

Here \texttt{kind} specifies the type of dataset, in this case one from PanSTARRS-1. Currently supported options include:

- \texttt{ps1}
- \texttt{ztf}
- \texttt{plasticc}
A convenience function is also included to read lcdata datasets in HDF5 format and parse them automatically:

```python
>>> dataset = parsnip.load_dataset('/path/to/data.h5')
```

This function will attempt to determine the dataset kind from the filename. This can be overridden with the kind keyword as in the previous example.

### 1.2.5 Loading a model in Python

Once a model has been trained, ParSNIP has a vast Python API for manipulating it and using it to generate predictions and plots. To load a model in Python:

```python
>>> import parsnip
>>> model = parsnip.load_model('/path/to/model.h5')
```

There are several built-in models included that can be loaded by specifying their name. Currently, these are:

- `plasticc` trained on the PLAsTiCC dataset.
- `ps1` trained on the PS1 dataset from Villar et al. 2020.
- `plasticc_photoz` trained on the PLAsTiCC dataset. Uses the photometric redshifts instead of the true redshifts.

To load one of these built-in models:

```python
>>> model = parsnip.load_model('plasticc')
```

Assuming that you have a light curve in `sncosmo` format, some examples of what can be done with a model include:

Predict the latent representation of a light curve:

```python
>>> model.predict(light_curve)
```

```json
{
'object_id': 'PS0909006',
...
's1': 0.19424194,
's1_error': 0.44743112,
's2': -0.051611423,
's2_error': 1.0143535,
...
}
```

Plot the predicted light curve:

```python
>>> parsnip.plot_light_curve(light_curve, model)
```

Plot the predicted spectrum at a given time:

```python
>>> parsnip.plot_spectrum(light_curve, model, time=53000.)
```

See the Reference / API page for a list of all of the built-in methods, or the notebooks that were used to make figures for Boone et al. 2021 for examples.
1.2.6 Classifying light curves

To classify light curves, we first need to predict their representations using a ParSNIP model. This can be done either with the `parsnip_predict` script described previously or by operating in memory on an `lcdata` Dataset object:

```python
>>> predictions = model.predict_dataset(dataset)
>>> print(predictions)
object_id   ra    dec    ...    s3          s3_error
--------    ------ ------    ...    ----------    ------------
PS0909006 333.9503 1.1848    ...    0.19424233  0.4474311
PS0909010 37.1182 -4.0789    ...    -0.40881702  0.59658796
PS0910012 52.4718 -28.0867    ...    -2.142636  0.08176677
PS0910016 35.3073 -3.9189    ...    -0.31671444  0.5740286
...       ...      ...    ...    ...            ...
```

A classifier can be trained on a set of predictions with:

```python
>>> classifier = parsnip.Classifier()
>>> classifier.train(predictions)
```

The classifier can be used to generate predictions for a new dataset with:

```python
>>> classifier.predict(new_predictions)
object_id   SLSN  SNII  SNIIn  SNIa  SNIbc
--------    ----- ----- ----- ----- ----- 
PS0909006  0.009   0.025  0.031  0.858  0.077
PS0909010  0.001   0.002  0.017  0.954  0.024
PS0910016  0.002   0.002  0.017  0.948  0.032
PSc000001  0.003   0.936  0.038  0.003  0.021
PSc0900022  0.960   0.001  0.037  0.001  0.000
...       ...      ...    ...    ...
```

For more details and examples, see the classification demo notebook.

1.3 SNCosmo Interface

1.3.1 Overview

ParSNIP provides an SNCosmo interface with an implementation of the `sncosmo.Source` class. To load the built-in ParSNIP model trained on the PLAsTiCC dataset:

```python
>>> import parsnip
>>> source = parsnip.ParsnipSncosmoSource('plasticc')
```

This source can be used in any SNCosmo models or methods. For example:

```python
>>> import sncosmo
>>> model = sncosmo.Model(source=source)

>>> model.param_names
['z', 't0', 'amplitude', 'color', 's1', 's2', 's3']
```
>>> data = sncosmo.load_example_data()
>>> result, fitted_model = sncosmo.fit_lc(
    ...    data, model,
    ...    ['z', 't0', 'amplitude', 's1', 's2', 's3', 'color'],
    ...    bounds={'z': (0.3, 0.7)},
    ... )

Note that ParSNIP is a generative model in that it predicts the full spectral time series of each transient. When used with the SNCosmo interface, it can operate on light curves observed in any bands, not just the ones that it was trained on.

### 1.3.2 Predicting the model parameters with variational inference

The ParSNIP model uses variational inference to predict the posterior distribution over all of the parameters of the model. An SNCosmo model can be initialized with the result of this prediction:

```python
>>> parsnip_model = parsnip.load_model( ... )
>>> sncosmo_model = parsnip_model.predict_sncosmo(light_curve)
```

### 1.4 Reproducing Boone 2021

#### 1.4.1 Overview

The details of the ParSNIP model are documented in Boone 2021. To reproduce all of the results in that paper, follow the following steps.

#### 1.4.2 Installing ParSNIP

Install the ParSNIP software package following the instructions on the [Installation](#) page.

#### 1.4.3 Downloading the data

From the desired working directory, run the following scripts on the command line to download the PLaSTiCC and PS1 datasets to `/data/` directory.

Download PS1:

```bash
$ lcdata_download_ps1
```

Download PLaSTiCC (warning, this can take a long time):

```bash
$ lcdata_download_plasticc
```

Build a combined PLaSTiCC training set for ParSNIP:

```bash
$ parsnip_build_plasticc_combined
```
1.4.4 Training the ParSNIP model

Note: Model training is much faster if a GPU is available. By default, ParSNIP will attempt to use the GPU if there is one and fallback to CPU if not. This can be overriden by passing e.g. --device cpu to the parsnip_train script where cpu is the desired PyTorch device.

Train a PS1 model using the full dataset (1 hour):

```bash
$ parsnip_train
  ./models/parsnip_ps1.pt
  ./data/ps1.h5
```

Train a PS1 model with a held-out validation set (1 hour):

```bash
$ parsnip_train
  ./models/parsnip_ps1_validation.pt
  ./data/ps1.h5
  --split_train_test
```

Train a PLAsTiCC model using the full dataset (1 day):

```bash
$ parsnip_train
  ./models/parsnip_plasticc.pt
  ./data/plasticc_combined.h5
```

Train a PLAsTiCC model with a held-out validation set (1 day):

```bash
$ parsnip_train
  ./models/parsnip_plasticc_validation.pt
  ./data/plasticc_combined.h5
  --split_train_test
```

1.4.5 Generate predictions

Generate predictions for the PS1 dataset (< 1 min):

```bash
parsnip_predict
  ./predictions/parsnip_predictions_ps1.h5
  ./models/parsnip_ps1.pt
  ./data/ps1.h5
```

Generate predictions for the PS1 dataset with 100-fold augmentation (3 min):

```bash
parsnip_predict
  ./predictions/parsnip_predictions_ps1_aug_100.h5
  ./models/parsnip_ps1.pt
  ./data/ps1.h5
  --augments 100
```

Generate predictions for the PLAsTiCC combined training dataset (7 min):

```bash
parsnip_predict
  ./predictions/parsnip_predictions_plasticc_combined.h5
  ./models/parsnip_plasticc.pt
  ./data/plasticc_combined.h5
```

Generate predictions for the PLAsTiCC training set with 100-fold augmentation (4 min):
1.5 Including Photometric Redshifts

1.5.1 Overview

The base ParSNIP model described in Boone 2021 assumes that the redshift of each transient is known. In Boone et al. 2022 (in prep.), ParSNIP was extended to handle datasets that only have photometric redshifts available. ParSNIP uses the photometric redshift as a prior and predicts the redshift of each transient. Currently ParSNIP only supports Gaussian photometric redshifts like the ones in the PLAsTiCC dataset, but it is straightforward to include more complex photometric redshift priors.

The `plasticc_photoz` built-in model was trained on the PLAsTiCC dataset and uses photometric redshifts instead of true redshifts. It can be loaded with the following command:

```python
>>> model = parsnip.load_model('plasticc_photoz')
```

This model assumes that each transient has metadata with a `hostgal_photoz` key containing the mean photometric redshift prediction and a `hostgal_photoz_err` key containing the photometric redshift uncertainty.

1.5.2 Training ParSNIP with photometric redshifts

The following steps can be used to train a model that uses photometric redshifts on the PLAsTiCC dataset and generate predictions for both the training and test datasets. You should first follow the steps in Reproducing Boone 2021 to download the PLAsTiCC dataset.

Photometric redshifts are enabled by passing the `--predict_redshift` flag to `parsnip_train`. Model training can be unstable at early epochs when the redshift is being predicted, so we recommend using larger batch sizes and starting the training with a lower learning rate. A batch size of 256 and a learning rate of $5 \times 10^{-4}$ is stable for the PLAsTiCC dataset.

Note: Model training is much faster if a GPU is available. By default, ParSNIP will attempt to use the GPU if there is one and fallback to CPU if not. This can be overridden by passing `--device cpu` to the `parsnip_train` script where `cpu` is the desired PyTorch device.

Train the PLAsTiCC model using the full dataset (1 day):

```
parsnip_predict ./predictions/parsnip_predictions_plasticc_train_aug_100.h5 \
   ./models/parsnip_plasticc.pt \
   ./data/plasticc_train.h5 \
   --augments 100
```

Generate predictions for the full PLAsTiCC dataset (1 hour):

```
parsnip_predict ./predictions/parsnip_predictions_plasticc_test.h5 \
   ./models/parsnip_plasticc.pt \
   ./data/plasticc_test.h5
```

1.4.6 Figures and analysis

All of the figures and analysis in Boone 2021 were done with Jupyter notebooks that are available on GitHub. To rerun these notebooks, copy the notebooks folder to the working directory and run the notebooks from within that folder.
```bash
$ parsnip_train \
  ./models/parsnip_plasticc_photoz.pt \ 
  ./data/plasticc_combined.h5 \ 
  --batch_size 256 \ 
  --learning_rate 5e-4 \ 
  --predict_redshift
```

Generate predictions for the PLAsTiCC training set with 100-fold augmentation (4 min):

```bash
parsnip_predict ./predictions/parsnip_predictions_plasticc_photoz_train_aug_100.h5 \ 
  ./models/parsnip_plasticc_photoz.pt \ 
  ./data/plasticc_train.h5 \ 
  --augments 100
```

Generate predictions for the full PLAsTiCC dataset (1 hour):

```bash
parsnip_predict ./predictions/parsnip_predictions_plasticc_photoz_test.h5 \ 
  ./models/parsnip_plasticc_photoz.pt \ 
  ./data/plasticc_test.h5
```

By default, ParSNIP uses a spectroscopic redshift prior with a width of 0.01 during training. This can be adjusted using the `specz_error` flag to `parsnip_train`. For example, running `parsnip_train ... --specz_error 0.05` will use a prior with a width of 0.05 instead.

1.5.3 Figures and analysis

All of the figures and analysis in Boone et al. 2022 (in prep.) can be reproduced with a Jupyter notebook that is available on GitHub. To rerun this notebook on a newly trained model, copy the notebooks folder to the working directory and run the notebook from within that folder.

1.6 Reference / API

1.6.1 Models

**Loading/saving a model**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ParsnipModel(path, bands[, device, threads, ...])</td>
<td>Generative model of transient light curves</td>
</tr>
<tr>
<td>load_model([path, device, threads])</td>
<td>Load a ParSNIP model.</td>
</tr>
<tr>
<td>ParsnipModel.save()</td>
<td>Save the model</td>
</tr>
<tr>
<td>ParsnipModel.to(device[, force])</td>
<td>Send the model to the specified device</td>
</tr>
</tbody>
</table>
parsnip.ParsnipModel

class parsnip.ParsnipModel(path, bands, device='cpu', threads=8, settings={},
ignore_unknown_settings=False)

Generative model of transient light curves

This class represents a generative model of transient light curves. Given a set of latent variables representing
a transient, it can predict the full spectral time series of that transient. It can also use variational inference to
predict the posterior distribution over the latent variables for a given light curve.

Parameters

- **path** (str) – Path to where the model should be stored on disk.
- **bands** (List[str]) – Bands that the model uses as input for variational inference
- **device** (str) – PyTorch device to use for the model
- **threads** (int) – Number of threads to use
- **settings** (dict) – Settings for the model. Any settings specified here will override the
defaults set in settings.py
- **ignore_unknown_settings** (bool) – If True, ignore any settings that are specified that
are unknown. Otherwise, raise a KeyError if an unknown setting is specified. By default
False.

__init__(path, bands[, device, threads, ...])

Initializes internal Module state, shared by both nn.Module and ScriptModule.

Methods

__init__(path, bands[, device, threads, ...])

Initializes internal Module state, shared by both nn.Module and ScriptModule.

add_module(name, module)

Adds a child module to the current module.

apply(fn)

Applies fn recursively to every submodule (as
returned by .children()) as well as self.

augment_light_curves(light_curves[, as_table])

Augment a set of light curves

bfloat16()

Casts all floating point parameters and buffers to
bfloa16 datatype.

buffers([recurse])

Returns an iterator over module buffers.

children()

Returns an iterator over immediate children modules.

cpu()

Moves all model parameters and buffers to the CPU.

cuda([device])

Moves all model parameters and buffers to the GPU.

decode(encoding, ref_times, color, times, ...)

Predict the light curves for a given set of latent vari-
ables

decode_spectra(encoding, phases, color[, ...])

Predict the spectra at a given set of latent variables
double()

Casts all floating point parameters and buffers to
double datatype.

encode(input_data)

Predict the latent variables for a set of light curves
eval()

Sets the module in evaluation mode.

extra_repr()

Set the extra representation of the module

fit(dataset[, max_epochs, augment, test_dataset])

Fit the model to a dataset

float()

Casts all floating point parameters and buffers to
float datatype.

continues on next page
### Table 1 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>forward(light_curves[, sample, to_numpy])</code></td>
<td>Run a set of light curves through the full ParSNIP model</td>
</tr>
<tr>
<td><code>get_buffer(target)</code></td>
<td>Returns the buffer given by target if it exists, otherwise throws an error.</td>
</tr>
<tr>
<td><code>get_data_loader(dataset[, augment])</code></td>
<td>Get a PyTorch DataLoader for an lcdata Dataset</td>
</tr>
<tr>
<td><code>get_extra_state()</code></td>
<td>Returns any extra state to include in the module's state_dict.</td>
</tr>
<tr>
<td><code>get_parameter(target)</code></td>
<td>Returns the parameter given by target if it exists, otherwise throws an error.</td>
</tr>
<tr>
<td><code>get_submodule(target)</code></td>
<td>Returns the submodule given by target if it exists, otherwise throws an error.</td>
</tr>
<tr>
<td><code>half()</code></td>
<td>Casts all floating point parameters and buffers to half datatype.</td>
</tr>
<tr>
<td><code>ipu(device)</code></td>
<td>Moves all model parameters and buffers to the IPU.</td>
</tr>
<tr>
<td><code>load_state_dict(state_dict[, strict])</code></td>
<td>Copies parameters and buffers from state_dict into this module and its descendants.</td>
</tr>
<tr>
<td><code>loss_function(result[, return_components, ...])</code></td>
<td>Compute the loss function for a set of light curves modules()</td>
</tr>
<tr>
<td><code>named_buffers([prefix, recurse])</code></td>
<td>Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.</td>
</tr>
<tr>
<td><code>named_children()</code></td>
<td>Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.</td>
</tr>
<tr>
<td><code>named_modules([memo, prefix, remove_duplicate])</code></td>
<td>Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.</td>
</tr>
<tr>
<td><code>named_parameters([prefix, recurse])</code></td>
<td>Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.</td>
</tr>
<tr>
<td><code>parameters([recurse])</code></td>
<td>Returns an iterator over module parameters.</td>
</tr>
<tr>
<td><code>predict(light_curves[, augment])</code></td>
<td>Generate predictions for a light curve or set of light curves.</td>
</tr>
<tr>
<td><code>predict_dataset(dataset[, augment])</code></td>
<td>Generate predictions for a dataset</td>
</tr>
<tr>
<td><code>predict_dataset_augmented(dataset[, augmentations])</code></td>
<td>Generate predictions for a dataset with augmentation</td>
</tr>
<tr>
<td><code>predict_light_curve(light_curve[, sample, ...])</code></td>
<td>Predict the flux of a light curve on a grid</td>
</tr>
<tr>
<td><code>predict_redshift(light_curve[, ...])</code></td>
<td>Predict the redshift of a light curve.</td>
</tr>
<tr>
<td><code>predict_redshift_distribution(light_curve[, ...])</code></td>
<td>Predict the redshift distribution for a light curve.</td>
</tr>
<tr>
<td><code>predict_sncosmo(light_curve[, sample])</code></td>
<td>Package the predictions for a light curve as an sncosmo model</td>
</tr>
<tr>
<td><code>predict_spectrum(light_curve, time[, ...])</code></td>
<td>Predict the spectrum of a light curve at a given time</td>
</tr>
<tr>
<td><code>preprocess(dataset[, chunksize, verbose])</code></td>
<td>Preprocess an lcdata dataset</td>
</tr>
<tr>
<td><code>register_backward_hook(hook)</code></td>
<td>Registers a backward hook on the module.</td>
</tr>
<tr>
<td><code>register_buffer(name, tensor[, persistent])</code></td>
<td>Adds a buffer to the module.</td>
</tr>
<tr>
<td><code>register_forward_hook(hook)</code></td>
<td>Registers a forward hook on the module.</td>
</tr>
<tr>
<td><code>register_forward_pre_hook(hook)</code></td>
<td>Registers a forward pre-hook on the module.</td>
</tr>
<tr>
<td><code>register_full_backward_hook(hook)</code></td>
<td>Registers a backward hook on the module.</td>
</tr>
<tr>
<td><code>register_load_state_dict_post_hook(hook)</code></td>
<td>Registers a post hook to be run after module's load_state_dict is called.</td>
</tr>
<tr>
<td><code>register_module(name, module)</code></td>
<td>Alias for add_module().</td>
</tr>
</tbody>
</table>

continues on next page
Table 1 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>register_parameter(name, param)</code></td>
<td>Adds a parameter to the module.</td>
</tr>
<tr>
<td><code>requires_grad_(requires_grad)</code></td>
<td>Change if autograd should record operations on parameters in this module.</td>
</tr>
<tr>
<td><code>save()</code></td>
<td>Save the model</td>
</tr>
<tr>
<td><code>score(dataset[, rounds, return_components, ...])</code></td>
<td>Evaluate the loss function on a given dataset.</td>
</tr>
<tr>
<td><code>set_extra_state(state)</code></td>
<td>This function is called from <code>load_state_dict()</code> to handle any extra state found within the state_dict.</td>
</tr>
<tr>
<td><code>share_memory()</code></td>
<td>See <code>torch.Tensor.share_memory()</code></td>
</tr>
<tr>
<td><code>state_dict(*args[, destination, prefix, ...])</code></td>
<td>Returns a dictionary containing a whole state of the module.</td>
</tr>
<tr>
<td><code>to(device[, force])</code></td>
<td>Send the model to the specified device</td>
</tr>
<tr>
<td><code>to_empty(*, device)</code></td>
<td>Moves the parameters and buffers to the specified device without copying storage.</td>
</tr>
<tr>
<td><code>train([mode])</code></td>
<td>Sets the module in training mode.</td>
</tr>
<tr>
<td><code>type(dst_type)</code></td>
<td>Casts all parameters and buffers to dst_type.</td>
</tr>
<tr>
<td><code>xpu([device])</code></td>
<td>Moves all model parameters and buffers to the XPU.</td>
</tr>
<tr>
<td><code>zero_grad(set_to_none)</code></td>
<td>Sets gradients of all model parameters to zero.</td>
</tr>
</tbody>
</table>

**Attributes**

- **T_destination**
  alias of `TypeVar(T_destination, bound=Dict[str, Any])`

- **dump_patches**

**parsnip.load_model**

`parsnip.load_model(path=None, device='cpu', threads=8)`

Load a ParSNIP model.

**Parameters**

- **path (str, optional)** – Path to the model on disk, or name of a model. If not specified, the default_model specified in settings.py is loaded.
- **device (str, optional)** – Torch device to load the model to, by default ‘cpu’
- **threads (int, optional)** – Number of threads to use, by default 8

**Returns**

- Loaded model

**Return type**

*ParsnipModel*
parsnip.ParsnipModel.save

ParsnipModel.save()

Save the model

parsnip.ParsnipModel.to

ParsnipModel.to(device, force=False)

Send the model to the specified device

**Parameters**

- `device` (*str*) – PyTorch device
- `force` (*bool, optional*) – If True, force the model to be sent to the device even if it is there already (useful if only parts of the model are there), by default False

---

**Interacting with a dataset**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ParsnipModel.preprocess(dataset[, ...])</td>
<td>Preprocess an lcdata dataset</td>
</tr>
<tr>
<td>ParsnipModel.augment_light_curves(light_curves)</td>
<td>Augment a set of light curves</td>
</tr>
<tr>
<td>ParsnipModel.get_data_loader(dataset[, augment])</td>
<td>Get a PyTorch DataLoader for an lcdata Dataset</td>
</tr>
<tr>
<td>ParsnipModel.fit(dataset[, max_epochs, ...])</td>
<td>Fit the model to a dataset</td>
</tr>
<tr>
<td>ParsnipModel.score(dataset[, rounds, ...])</td>
<td>Evaluate the loss function on a given dataset.</td>
</tr>
</tbody>
</table>

parsnip.ParsnipModel.preprocess

ParsnipModel.preprocess(dataset, chunksize=64, verbose=True)

Preprocess an lcdata dataset

The preprocessing will be done over multiple threads. Set ParsnipModel.threads to change how many are used. If the dataset is already preprocessed, then nothing will be done and it will be returned as is.

**Parameters**

- `dataset` (*Dataset*) – Dataset to preprocess
- `chunksize` (*int, optional*) – Number of light curves to process at a time, by default 64
- `verbose` (*bool, optional*) – Whether to show a progress bar, by default True

**Returns**

Preprocessed dataset

**Return type**

Dataset
parsnip.ParsnipModel.augment_light_curves

ParsnipModel.augment_light_curves(light_curves, as_table=True)

Augment a set of light curves

Parameters

- **light_curves** (List[Table]) – List of light curves to augment
- **as_table** (bool, optional) – Whether to return the light curves as astropy Tables, by default True. Constructing new tables is relatively slow, so internally we skip this step when training the ParSNIP model.

Returns

Augmented light curves

Return type

List

parsnip.ParsnipModel.get_data_loader

ParsnipModel.get_data_loader(dataset, augment=False, **kwargs)

Get a PyTorch DataLoader for an lcdata Dataset

Parameters

- **dataset** (Dataset) – Dataset to load
- **augment** (bool, optional) – Whether to augment the dataset, by default False

Returns

PyTorch DataLoader for the dataset

Return type

DataLoader

parsnip.ParsnipModel.fit

ParsnipModel.fit(dataset, max_epochs=1000, augment=True, test_dataset=None)

Fit the model to a dataset

Parameters

- **dataset** (Dataset) – Dataset to fit to
- **max_epochs** (int, optional) – Maximum number of epochs, by default 1000
- **augment** (bool, optional) – Whether to use augmentation, by default True
- **test_dataset** (Dataset, optional) – Test dataset that will be scored at the end of each epoch, by default None
parsnip.ParsnipModel.score

**ParsnipModel.score**: `(dataset, rounds=1, return_components=False, sample=True)`
Evaluate the loss function on a given dataset.

**Parameters**
- **dataset** *(Dataset)* – Dataset to run on
- **rounds** *(int, optional)* – Number of rounds to use for evaluation. VAEs are stochastic, so the loss function is not deterministic. By running for multiple rounds, the uncertainty on the loss function can be decreased. Default 1.
- **return_components** *(bool, optional)* – Whether to return the individual parts of the loss function, by default False. See `loss_function` for details.

**Returns**
Computed loss function

**Return type**
loss

Generating model predictions

<table>
<thead>
<tr>
<th>parsnip.ParsnipModel.predict</th>
<th><strong>ParsnipModel.predict</strong> *(light_curves[, augment])</th>
<th>Generate predictions for a light curve or set of light curves.</th>
</tr>
</thead>
<tbody>
<tr>
<td>parsnip.ParsnipModel.predict_dataset</td>
<td><strong>ParsnipModel.predict_dataset</strong> *(dataset[, augment])</td>
<td>Generate predictions for a dataset</td>
</tr>
<tr>
<td>parsnip.ParsnipModel.predict_dataset_augmented</td>
<td><strong>ParsnipModel.predict_dataset_augmented</strong> <em>(dataset)</em></td>
<td>Generate predictions for a dataset with augmentation</td>
</tr>
<tr>
<td>parsnip.ParsnipModel.predict_light_curve</td>
<td><strong>ParsnipModel.predict_light_curve</strong> <em>(light_curve)</em></td>
<td>Predict the flux of a light curve on a grid</td>
</tr>
<tr>
<td>parsnip.ParsnipModel.predict_spectrum</td>
<td><strong>ParsnipModel.predict_spectrum</strong> <em>(light_curve, time)</em></td>
<td>Predict the spectrum of a light curve at a given time</td>
</tr>
<tr>
<td>parsnip.ParsnipModel.predict_sncosmo</td>
<td><strong>ParsnipModel.predict_sncosmo</strong> <em>(light_curve[, ...])</em></td>
<td>Package the predictions for a light curve as an sncosmo model</td>
</tr>
</tbody>
</table>

parsnip.ParsnipModel.predict

**ParsnipModel.predict**: `(light_curves, augment=False)`
Generate predictions for a light curve or set of light curves.

**Parameters**
- **light_curves** *(Table or List[Table])* – Light curve(s) to generate predictions for.
- **augment** *(bool, optional)* – Whether to augment the light curve(s), by default False

**Returns**
Table (for multiple light curves) or dict (for a single light curve) containing the predictions.

**Return type**
Table or dict
parsnip.ParsnipModel.predict_dataset

ParsnipModel.predict_dataset(dataset, augment=False)
Generate predictions for a dataset

Parameters
- **dataset** (Dataset) – Dataset to generate predictions for.
- **augment** (bool, optional) – Whether to perform augmentation, False by default.

Returns
- **predictions** – astropy Table with one row for each light curve and columns with each of the predicted values.

Return type
  Table

parsnip.ParsnipModel.predict_dataset_augmented

ParsnipModel.predict_dataset_augmented(dataset, augments=10)
Generate predictions for a dataset with augmentation

This will first generate predictions for the dataset without augmentation, and will then generate predictions for the dataset with augmentation the given number of times. This returns a dataframe in the same format as predict_dataset, but with the following additional columns: - original_object_id: the original object_id for each augmentation. - augmented: True for augmented light curves, False for original ones.

Parameters
- **dataset** (Dataset) – Dataset to generate predictions for.
- **augments** (int, optional) – Number of times to augment the dataset, by default 10

Returns
- **predictions** – astropy Table with one row for each light curve and columns with each of the predicted values.

Return type
  Table

parsnip.ParsnipModel.predict_light_curve

ParsnipModel.predict_light_curve(light_curve, sample=False, count=None, sampling=1.0, pad=50.0)
Predict the flux of a light curve on a grid

Parameters
- **light_curve** (Table) – Light curve to predict
- **sample** (bool, optional) – If True, sample from the latent variable posteriors. Otherwise, use the MAP. By default False.
- **count** (int, optional) – Number of light curves to predict, by default None (single prediction)
- **sampling** (int, optional) – Grid sampling in days, by default 1.
- **pad** (int, optional) – Number of days before and after the light curve observations to predict the light curve at, by default 50.
Returns

- ndarray – Times that the model was sampled at
- ndarray – Flux of the model in each band
- ndarray – Model result from ParsnipModel.forward

`parsnip.ParsnipModel.predict_spectrum`

`ParsnipModel.predict_spectrum(light_curve, time, sample=False, count=None)`

Predict the spectrum of a light curve at a given time

Parameters

- `light_curve` (Table) – Light curve
- `time` (float) – Time to predict the spectrum at
- `sample` (bool, optional) – If True, sample from the latent variable posteriors. Otherwise, use the MAP. By default False.
- `count` (int, optional) – Number of spectra to predict, by default None (single prediction)

Returns

Predicted spectrum at the wavelengths specified by `model_wave`

Return type

ndarray

`parsnip.ParsnipModel.predict_sncosmo`

`ParsnipModel.predict_sncosmo(light_curve, sample=False)`

Package the predictions for a light curve as an sncosmo model

This method performs variational inference on a light curve to predict its latent representation. It then initializes an SNCosmo model with that representation.

Parameters

- `light_curve` (Table) – Light curve
- `sample` (bool, optional) – If True, sample from the latent variable posteriors. Otherwise, use the MAP. By default False.

Returns

SNCosmo model initialized with the light curve’s predicted latent representation

Return type

ParsnipSncosmoModel

Individual parts of the model

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ParsnipModel.forward(light_curves[, sample, ...])</code></td>
<td>Run a set of light curves through the full ParSNIP model</td>
</tr>
<tr>
<td><code>ParsnipModel.encode(input_data)</code></td>
<td>Predict the latent variables for a set of light curves</td>
</tr>
<tr>
<td><code>ParsnipModel.decode(encoding, ref_times, ...)</code></td>
<td>Predict the light curves for a given set of latent variables</td>
</tr>
<tr>
<td><code>ParsnipModel.decode_spectra(encoding, ...[, ...])</code></td>
<td>Predict the spectra at a given set of latent variables</td>
</tr>
<tr>
<td><code>ParsnipModel.loss_function(result[, ...])</code></td>
<td>Compute the loss function for a set of light curves</td>
</tr>
</tbody>
</table>
parsnip.ParsnipModel.forward

ParsnipModel.forward(light_curves, sample=True, to_numpy=False)

Run a set of light curves through the full ParSNIP model.
We use variational inference to predict the latent representation of each light curve, and we then use the generative model to predict the light curves for those representations.

Parameters

- **light_curves** (List[Table]) – List of light curves
- **sample** (bool, optional) – If True (default), sample from the posterior distribution. If False, use the MAP.
- **to_numpy** (bool, optional) – Whether to convert the outputs to numpy arrays, by default False

Returns

Result dictionary. If to_numpy is True, all of the elements will be numpy arrays. Otherwise, they will be PyTorch tensors on the model’s device.

Return type
dict

parsnip.ParsnipModel.encode

ParsnipModel.encode(input_data)

Predict the latent variables for a set of light curves.
We use variational inference, and predict the parameters of a posterior distribution over the latent space.

Parameters

- **input_data** (FloatTensor) – Input data representing a set of gridded light curves

Returns

- FloatTensor – Mean predictions for each latent variable
- FloatTensor – Log-variance predictions for each latent variable

parsnip.ParsnipModel.decode

ParsnipModel.decode(encoding, ref_times, color, times, redshifts, band_indices, amplitude=None)

Predict the light curves for a given set of latent variables.

Parameters

- **encoding** (FloatTensor) – Coordinates in the ParSNIP intrinsic latent space for each light curve
- **ref_times** (FloatTensor) – Reference time for each light curve
- **color** (FloatTensor) – Color of each light curve
- **times** (FloatTensor) – Times to predict each light curve at
- **redshifts** (FloatTensor) – Redshift of each light curve
- **band_indices** (LongTensor) – Band indices for each observation
• **amplitude** *(FloatTensor, optional)* – Amplitude to scale each light curve by, by default no scaling will be applied

**Returns**

• FloatTensor – Model spectra
• FloatTensor – Model photometry

**parsnip.ParsnipModel.decode_spectra**

`ParsnipModel.decode_spectra(encoding, phases, color, amplitude=None)`

Predict the spectra at a given set of latent variables

**Parameters**

• **encoding** *(FloatTensor)* – Coordinates in the ParSNIP intrinsic latent space for each light curve
• **phases** *(FloatTensor)* – Phases to decode each light curve at
• **color** *(FloatTensor)* – Color of each light curve
• **amplitude** *(FloatTensor, optional)* – Amplitude to scale each light curve by, by default no scaling will be applied.

**Returns**

Predicted spectra

**Return type**

FloatTensor

**parsnip.ParsnipModel.loss_function**

`ParsnipModel.loss_function(result, return_components=False, return_individual=False)`

Compute the loss function for a set of light curves

**Parameters**

• **result** *(dict)* – Output of `forward`
• **return_components** *(bool, optional)* – Whether to return the individual parts of the loss function, by default False.
• **return_individual** *(bool, optional)* – Whether to return the loss function for each light curve individually, by default False.

**Returns**

If `return_components` and `return_individual` are False, return a single value representing the loss function for a set of light curves. If `return_components` is True, then we return a set of four values representing the negative log likelihood, the KL divergence, the regularization penalty, and the amplitude probability. If `return_individual` is True, then we return the loss function for each light curve individually.

**Return type**

float or FloatTensor
1.6.2 Datasets

Loading datasets

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>load_dataset(path[, kind, in_memory, ...])</code></td>
<td>Load a dataset using the lcdata package.</td>
</tr>
<tr>
<td><code>load_datasets(dataset_paths[, ...])</code></td>
<td>Load a list of datasets and merge them.</td>
</tr>
<tr>
<td><code>parse_dataset(dataset[, path_or_name, kind, ...])</code></td>
<td>Parse a dataset from the lcdata package.</td>
</tr>
</tbody>
</table>

`parsnip.load_dataset`

`parsnip.load_dataset(path, kind=None, in_memory=True, reject_invalid=True, require_redshift=True, verbose=True)`

Load a dataset using the lcdata package.

This can be any lcdata HDF5 dataset. We use `parse_dataset` to clean things up for ParSNIP by rejecting irrelevant light curves (e.g. galactic ones) and updating class labels.

We try to guess the dataset type from the filename. If this doesn’t work, specify the filename explicitly instead.

**Parameters**
- `path (str)` – Path to the dataset on disk
- `kind (str, optional)` – Kind of dataset, by default we will attempt to determine it from the filename
- `in_memory (bool, optional)` – If False, don’t load the light curves into memory, and only load the metadata. See `lcdata.Dataset` for details.
- `reject_invalid (bool, optional)` – Whether to reject invalid light curves, by default True
- `verbose (bool, optional)` – If True, print parsing information, by default True

**Returns**
- Loaded dataset

**Return type**
- Dataset

`parsnip.load_datasets`

`parsnip.load_datasets(dataset_paths, reject_invalid=True, require_redshift=True, verbose=True)`

Load a list of datasets and merge them

**Parameters**
- `dataset_paths (List[str])` – Paths to each dataset to load
- `verbose (bool, optional)` – If True, print parsing information, by default True

**Returns**
- Loaded dataset

**Return type**
- Dataset
parsnip, Release 1.3.1

parsnip.parse_dataset

parsnip.parse_dataset(dataset, path_or_name=None, kind=None, reject_invalid=True, require_redshift=True, verbose=True)

Parse a dataset from the lcdata package.

We cut out observations that are not relevant for the ParSNIP model (e.g. galactic ones), and update the class labels.

We try to guess the kind of dataset from the filename. If this doesn’t work, specify the kind explicitly instead.

Parameters

• **dataset** (Dataset) – Dataset to parse

• **path_or_name** (str, optional) – Name of the dataset, or path to it, by default None

• **kind** (str, optional) – Kind of dataset, by default None

• **reject_invalid** (bool, optional) – Whether to reject invalid light curves, by default True

• **verbose** (bool, optional) – If true, print parsing information, by default True

Returns

Parsed dataset

Return type

Dataset

Parsers for specific instruments

| parse_plasticc(dataset[, reject_invalid, ...]) | Parse a PLAsTiCC dataset |
| parse_ps1(dataset[, reject_invalid, verbose]) | Parse a PanSTARRS-1 dataset |
| parse_ztf(dataset[, reject_invalid, verbose]) | Parse a ZTF dataset |

parsnip.parse_plasticc

parsnip.parse_plasticc(dataset, reject_invalid=True, verbose=True)

Parse a PLAsTiCC dataset

Parameters

• **dataset** (Dataset) – PLAsTiCC dataset to parse

Returns

Parsed dataset

Return type

lcdata.Dataset
parsnip.parse_ps1

parsnip.parse_ps1(dataset, reject_invalid=True, verbose=True)
Parse a PanSTARRS-1 dataset

Parameters
- dataset (Dataset) – PanSTARRS-1 dataset to parse

Returns
- Parsed dataset

Return type
- Dataset

parsnip.parse_ztf

parsnip.parse_ztf(dataset, reject_invalid=True, verbose=True)
Parse a ZTF dataset

Parameters
- dataset (Dataset) – ZTF dataset to parse

Returns
- Parsed dataset

Return type
- Dataset

Tools for manipulating datasets

split_train_test(dataset) Split a dataset into training and testing parts.
get_bands(dataset) Retrieve a list of bands in a dataset

parsnip.split_train_test

parsnip.split_train_test(dataset)
Split a dataset into training and testing parts.
We train on 90%, and test on 10%. We use a fixed algorithm to split the train and test so that we don’t have to keep track of what we did.

Parameters
- dataset (Dataset) – Dataset to split

Returns
- Dataset – Training dataset
- Dataset – Test dataset
parsnip, Release 1.3.1

parsnip.get_bands

parsnip.get_bands(dataset)
Retrieve a list of bands in a dataset

Parameters
    dataset (Dataset) – Dataset to retrieve the bands from

Returns
    List of bands in the dataset sorted by effective wavelength

Return type
    List[str]

1.6.3 Plotting

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>plot_light_curve(light_curve[, model, ...])</td>
<td>Plot a light curve</td>
</tr>
<tr>
<td>plot_representation(predictions, plot_labels)</td>
<td>Plot the representation of a ParSNIP model</td>
</tr>
<tr>
<td>plot_spectrum(light_curve, model, time[, ...])</td>
<td>Plot the spectrum of a light curve predicted by a ParSNIP model</td>
</tr>
<tr>
<td>plot_spectra(light_curve, model[, times, ...])</td>
<td>Plot the spectral time series of a light curve predicted by a ParSNIP model</td>
</tr>
<tr>
<td>plot_sne_space(light_curve, model, name[, ...])</td>
<td>Compare a ParSNIP spectrum prediction to a real spectrum from sne.space</td>
</tr>
<tr>
<td>plot_confusion_matrix(predictions, ...[, ...])</td>
<td>Plot a confusion matrix</td>
</tr>
<tr>
<td>get_band_plot_color(band)</td>
<td>Return the plot color for a given band.</td>
</tr>
<tr>
<td>get_band_plot_marker(band)</td>
<td>Return the plot marker for a given band.</td>
</tr>
</tbody>
</table>

parsnip.plot_light_curve

parsnip.plot_light_curve(light_curve, model=None, count=100, show_uncertainty_bands=True, show_missing_bandpasses=False, percentile=68, normalize_flux=False, sncosmo_model=None, sncosmo_label='SNCosmo Model', ax=None)

Plot a light curve

Parameters
    • light_curve (Table) – Light curve to plot
    • model (ParsnipModel, optional) – ParSNIP model to show, by default None
    • count (int, optional) – Number of samples from the ParSNIP model, by default 100
    • show_uncertainty_bands (bool, optional) – If True (default), show uncertainty bands. Otherwise, show individual draws.
    • show_missing_bandpasses (bool, optional) – Whether to show model predictions for bandpasses where there is no data, by default False
    • percentile (int, optional) – Percentile for the uncertainty bands, by default 68
    • normalize_flux (bool, optional) – Whether to normalize the flux, by default False
    • sncosmo_model (Model, optional) – SNCosmo model to show, by default None
    • sncosmo_label (str, optional) – Legend label for the SNCosmo model, by default ‘SNCosmo Model’
parsnip.plot_representation

parsnip.plot_representation(predictions, plot_labels, mask=None, idx1=1, idx2=2, idx3=None, max_count=1000, show_legend=True, legend_ncol=1, marker='o', markersize=5, ax=None)

Plot the representation of a ParSNIP model

Parameters

• predictions (Table) – Predictions for a dataset from predict_dataset
• plot_labels (List[str]) – Labels for each of the classes
• mask (array, optional) – Mask to apply to the predictions, by default None
• idx1 (int, optional) – Intrinsic latent variable to plot on the x axis, by default 1
• idx2 (int, optional) – Intrinsic latent variable to plot on the y axis, by default 2
• idx3 (int, optional) – If specified, show a three paneled plot with this latent variable in the extra two panels plotted against the other ones
• max_count (int, optional) – Maximum number of light curves to show of each type, by default 1000
• show_legend (bool, optional) – Whether to show the legend, by default True
• legend_ncol (int, optional) – Number of columns to use in the legend, by default 1
• marker (str, optional) – Matplotlib marker to use, by default None
• markersize (int, optional) – Matplotlib marker size, by default 5
• ax (axis, optional) – Matplotlib axis, by default None

parsnip.plot_spectrum

parsnip.plot_spectrum(light_curve, model, time, count=100, show_uncertainty_bands=True, percentile=68, ax=None, c=None, label=None, offset=None, normalize_flux=False, normalize_min_wave=5500.0, normalize_max_wave=6500.0, spectrum_label=None, spectrum_label_wave=7500.0, spectrum_label_offset=0.2, flux_scale=1.0)

Plot the spectrum of a light curve predicted by a ParSNIP model

Parameters

• light_curve (Table) – Light curve
• model (ParsnipModel) – Model to use for the prediction
• time (float) – Time to predict the spectrum at
• count (int, optional) – Number of spectra to sample, by default 100
• show_uncertainty_bands (bool, optional) – Whether to show uncertainty bands, by default True
• percentile (int, optional) – Percentile for the uncertainty bands, by default 68
• ax (axis, optional) – Matplotlib axis to use, by default None
• c (str, optional) – Color for the plot, by default None
parsnip.plot_spectra

parsnip.plot_spectra(light_curve, model, times=[0.0, 10.0, 20.0, 30.0], flux_scale=1.0, ax=None, sncosmo_model=None, sncosmo_label='SNCosmo Model', spectrum_label_offset=0.2)

Plot the spectral time series of a light curve predicted by a ParSNIP model

Parameters

- **light_curve (Table)** – Light curve
- **model (ParsnipModel)** – Model to use for the predictions
- **times (list, optional)** – Times to predict the spectra at, by default [0., 10., 20., 30.]
- **flux_scale (float, optional)** – Scale to multiply the flux by, by default 1.
- **ax (axis, optional)** – Matplotlib axis, by default None
- **sncosmo_model (Model, optional)** – SNCosmo model to overplot, by default None
- **sncosmo_label (str, optional)** – Label for the SNCosmo model, by default ‘SNCosmo Model’
- **spectrum_label_offset (float, optional)** – Offset of the time labels for each spectrum, by default 0.2

parsnip.plot_sne_space

parsnip.plot_sne_space(light_curve, model, name, min_wave=10000.0, max_wave=0.0, time_diff=0.0, min_time=-10000.0, max_time=100000.0, source=None, kernel=5, flux_scale=0.5, label_wave=9000.0, label_offset=0.2, figsize=(5, 6))

Compare a ParSNIP spectrum prediction to a real spectrum from sne.space

Parameters

- **light_curve (Table)** – Light curve
- **model (ParsnipModel)** – ParSNIP Model to use for the prediction
parsnip, Release 1.3.1

- name (str) – Name of the light curve on sne.space
- min_wave (float, optional) – Ignore any spectra that don’t have data bluer than this wavelength, by default 10000.
- max_wave (float, optional) – Ignore any spectra that don’t have data redder than this wavelength, by default 0.
- time_diff (float, optional) – Minimum time between spectra, by default 0.
- min_time (float, optional) – Ignore any spectra before this time, by default -10000.
- max_time (float, optional) – Ignore any spectra after this time, by default 100000.
- source (str, optional) – Ignore any spectra not from this source, by default None
- kernel (int, optional) – Smooth the spectra by a median filter kernel of this size, by default 5
- flux_scale (float, optional) – Scale the flux by this amount, by default 0.5
- label_wave (float, optional) – Show labels with the times of each spectrum at this wavelength, by default 9000.
- label_offset (float, optional) – Y offset to use for the labels, by default 0.2
- figsize (tuple, optional) – Figure size, by default (5, 6)

parsnip.plot_confusion_matrix

parsnip.plot_confusion_matrix(predictions, classifications, figsize=(5, 4), title=None, verbose=True)

Plot a confusion matrix

Adapted from example that used to be at http://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html

Parameters
- predictions (Table) – Predictions from predict_dataset
- classifications (Table) – Classifications from a Classifier
- figsize (tuple, optional) – Figure size, by default (5, 4)
- title (str, optional) – Figure title, by default None
- verbose (bool, optional) – Whether to print additional statistics, by default True

parsnip.get_band_plot_color

parsnip.get_band_plot_color(band)

Return the plot color for a given band.

If the band does not yet have a color assigned to it, then a random color will be assigned (in a systematic way).

Parameters
- band (str) – Name of the band to use.

Returns
Matplotlib color to use when plotting the band

Return type
str
parsnip.get_band_plot_marker

parsnip.get_band_plot_marker(band)
Return the plot marker for a given band.
If the band does not yet have a marker assigned to it, then we use the default circle.

Parameters
    band (str) – Name of the band to use.

Returns
    Matplotlib marker to use when plotting the band

Return type
    str

1.6.4 Classification

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classifier()</td>
<td>LightGBM classifier that operates on ParSNIP predictions</td>
</tr>
<tr>
<td>extract_top_classifications(classifications)</td>
<td>Extract the top classification for each row a classifications Table.</td>
</tr>
<tr>
<td>weighted_multi_logloss(true_types, ...)</td>
<td>Calculate a weighted log loss metric.</td>
</tr>
</tbody>
</table>

parsnip.Classifier

class parsnip.Classifier
    LightGBM classifier that operates on ParSNIP predictions
    __init__()

Methods

__init__()

classify(predictions)  Classify light curves using predictions from a ParsnipModel
extract_features(predictions)  Extract features used for classification
load(path)  Load a classifier that was saved to disk
train(predictions[, num_folds, labels, ...])  Train a classifier on the predictions from a ParSNIP model
write(path)  Write the classifier out to disk
parsnip.extract_top_classifications

parsnip.extract_top_classifications(classifications)

Extract the top classification for each row a classifications Table.
This is a bit complicated when working with astropy Tables.

Parameters

classifications (Table) – Classifications table output from a Classifier

Returns

numpy array with the top type for each light curve

Return type

numpy.array

parsnip.weighted_multi_logloss

parsnip.weighted_multi_logloss(true_types, classifications)

Calculate a weighted log loss metric.
This is the metric used for the PLAsTiCC challenge (with class weights set to 1) as described in Malz et al. 2019

Parameters

• true_types (ndarray) – True types for each object
• classifications (Table) – Classifications table output from a Classifier

Returns

[description]

Return type

[type]

1.6.5 SNCosmo Interface

<table>
<thead>
<tr>
<th>ParsnipSncosmoSource([model])</th>
<th>SNCosmo interface for a ParSNIP model</th>
</tr>
</thead>
<tbody>
<tr>
<td>ParsnipModel.predict_sncosmo(light_curve[, ...])</td>
<td>Package the predictions for a light curve as an sncosmo model</td>
</tr>
</tbody>
</table>

parsnip.ParsnipSncosmoSource

class parsnip.ParsnipSncosmoSource(model=None)

SNCosmo interface for a ParSNIP model

Parameters

model (ParsnipModel or str, optional) – ParSNIP model to use, or path to a model on disk.

__init__(model=None)
## Methods

- **__init__(model)**
  
  Flux through the given bandpass(es) at the given phase(s).

- **bandflux(band, phase[, zp, zpsys])**
  
  Magnitude at the given phase(s) through the given bandpass(es), and for the given magnitude system(s).

- **bandmag(band, magsys, phase)**
  
  The spectral flux density at the given phase and wavelength values.

- **flux(phase, wave)**
  
  Get parameter of the model by name.

- **get(name)**
  
  Max phase.

- **maxphase()**

- **maxwave()**

- **minphase()**

- **minwave()**

- **peakmag(band, magsys[, sampling])**
  
  Peak apparent magnitude in rest-frame bandpass.

- **peakphase(band_or_wave[, sampling])**
  
  Determine phase of maximum flux for the given band/wavelength.

- **set(**param_dict)**
  
  Set parameters of the model by name.

- **set_peakmag(m, band, magsys[, sampling])**
  
  Set peak apparent magnitude in rest-frame bandpass.

- **update(param_dict)**
  
  Set parameters of the model from a dictionary.

## Attributes

- **param_names**
  
  List of parameter names.

- **parameters**
  
  Parameter value array.

### 1.6.6 Custom Neural Network Layers

- **ResidualBlock(in_channels, out_channels, ...)**
  
  1D residual convolutional neural network block

- **Conv1dBlock(in_channels, out_channels, dilation)**
  
  1D convolutional neural network block

- **GlobalMaxPoolingTime()**
  
  Time max pooling layer for 1D sequences

**parsnip.ResidualBlock**

```python
class parsnip.ResidualBlock(in_channels, out_channels, dilation)
    1D residual convolutional neural network block

This module operates on 1D sequences. The input will be padded so that length of the sequences is be left unchanged.
```

### Parameters

- **in_channels** *(int)* – Number of channels for the input
- **out_channels** *(int)* – Number of channels for the output
• **dilation** *(int)* – Dilation to use in the convolution

**__init__**(in_channels, out_channels, dilation)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

**Methods**

**__init__**(in_channels, out_channels, dilation) Initializes internal Module state, shared by both nn.Module and ScriptModule.

**add_module**(name, module) Adds a child module to the current module.

**apply**(fn) Applies fn recursively to every submodule (as returned by .children()) as well as self.

**bfloat16()** Casts all floating point parameters and buffers to bfloat16 datatype.

**buffers** *(recurse)* Returns an iterator over module buffers.

**children** *(recurse)* Returns an iterator over immediate children modules.

**cpu()** Moves all model parameters and buffers to the CPU.

**cuda** *(device)* Moves all model parameters and buffers to the GPU.

**double()** Casts all floating point parameters and buffers to double datatype.

**eval()** Sets the module in evaluation mode.

**extra_repr()** Set the extra representation of the module

**float()** Casts all floating point parameters and buffers to float datatype.

**forward**(x) Defines the computation performed at every call.

**get_buffer**(target) Returns the buffer given by target if it exists, otherwise throws an error.

**get_extra_state()** Returns any extra state to include in the module’s state_dict.

**get_parameter**(target) Returns the parameter given by target if it exists, otherwise throws an error.

**get_submodule**(target) Returns the submodule given by target if it exists, otherwise throws an error.

**half()** Casts all floating point parameters and buffers to half datatype.

**ipu** *(device)* Moves all model parameters and buffers to the IPU.

**load_state_dict**(state_dict[, strict]) Copies parameters and buffers from state_dict into this module and its descendants.

**modules()** Returns an iterator over all modules in the network.

**named_buffers**(prefix, recurse) Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

**named_children()** Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

**named_modules**(prefix, remove_duplicate) Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

**named_parameters**(prefix, recurse) Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

**parameters**(recurse) Returns an iterator over module parameters.

continues on next page
Table 2 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>register_backward_hook(hook)</code></td>
<td>Registers a backward hook on the module.</td>
</tr>
<tr>
<td><code>register_buffer(name, tensor[, persistent])</code></td>
<td>Adds a buffer to the module.</td>
</tr>
<tr>
<td><code>register_forward_hook(hook)</code></td>
<td>Registers a forward hook on the module.</td>
</tr>
<tr>
<td><code>register_forward_pre_hook(hook)</code></td>
<td>Registers a forward pre-hook on the module.</td>
</tr>
<tr>
<td><code>register_full_backward_hook(hook)</code></td>
<td>Registers a backward hook on the module.</td>
</tr>
<tr>
<td><code>register_load_state_dict_post_hook(hook)</code></td>
<td>Registers a post hook to be run after module’s <code>load_state_dict</code> is called.</td>
</tr>
<tr>
<td><code>register_module(name, module)</code></td>
<td>Alias for <code>add_module()</code>.</td>
</tr>
<tr>
<td><code>register_parameter(name, param)</code></td>
<td>Adds a parameter to the module.</td>
</tr>
<tr>
<td><code>requires_grad_([requires_grad])</code></td>
<td>Change if autograd should record operations on parameters in this module.</td>
</tr>
<tr>
<td><code>set_extra_state(state)</code></td>
<td>This function is called from <code>load_state_dict()</code> to handle any extra state found within the state_dict.</td>
</tr>
<tr>
<td><code>share_memory()</code></td>
<td>See <code>torch.Tensor.share_memory_()</code></td>
</tr>
<tr>
<td><code>state_dict(*args[, destination, prefix, ...])</code></td>
<td>Returns a dictionary containing a whole state of the module.</td>
</tr>
<tr>
<td><code>to(*args, **kwargs)</code></td>
<td>Moves and/or casts the parameters and buffers.</td>
</tr>
<tr>
<td><code>to_empty(*, device)</code></td>
<td>Moves the parameters and buffers to the specified device without copying storage.</td>
</tr>
<tr>
<td><code>train([mode])</code></td>
<td>Sets the module in training mode.</td>
</tr>
<tr>
<td><code>type(dst_type)</code></td>
<td>Casts all parameters and buffers to dst_type.</td>
</tr>
<tr>
<td><code>xpu([device])</code></td>
<td>Moves all model parameters and buffers to the XPU.</td>
</tr>
<tr>
<td><code>zero_grad([set_to_none])</code></td>
<td>Sets gradients of all model parameters to zero.</td>
</tr>
</tbody>
</table>

**Attributes**

- `T_destination` alias of `TypeVar(T_destination, bound=Dict[str, Any])`
- `dump_patches`

**parsnip.Conv1dBlock**

**class** parsnip.Conv1dBlock(`in_channels, out_channels, dilation`)

1D convolutional neural network block

This module operates on 1D sequences. The input will be padded so that length of the sequences is be left unchanged.

**Parameters**

- `in_channels (int)` – Number of channels for the input
- `out_channels (int)` – Number of channels for the output
- `dilation (int)` – Dilation to use in the convolution

**init** (`in_channels, out_channels, dilation`)

Initializes internal Module state, shared by both nn.Module and ScriptModule.
Methods

__init__(in_channels, out_channels, dilation) Initializes internal Module state, shared by both nn.Module and ScriptModule.

add_module(name, module) Adds a child module to the current module.

apply(fn) Applies fn recursively to every submodule (as returned by .children()) as well as self.

bfloat16() Casts all floating point parameters and buffers to bfloat16 datatype.

buffers([recurse]) Returns an iterator over module buffers.

children() Returns an iterator over immediate children modules.

cpu() Moves all model parameters and buffers to the CPU.

cuda([device]) Moves all model parameters and buffers to the GPU.

double() Casts all floating point parameters and buffers to double datatype.

eval() Sets the module in evaluation mode.

extra_repr() Set the extra representation of the module

float() Casts all floating point parameters and buffers to float datatype.

forward(x) Defines the computation performed at every call.

get_buffer(target) Returns the buffer given by target if it exists, otherwise throws an error.

get_extra_state() Returns any extra state to include in the module's state_dict.

get_parameter(target) Returns the parameter given by target if it exists, otherwise throws an error.

get_submodule(target) Returns the submodule given by target if it exists, otherwise throws an error.

half() Casts all floating point parameters and buffers to half datatype.

ipu([device]) Moves all model parameters and buffers to the IPU.

load_state_dict(state_dict[, strict]) Copies parameters and buffers from state_dict into this module and its descendants.

modules() Returns an iterator over all modules in the network.

named_buffers([prefix, recurse]) Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

named_children() Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

named_modules([memo, prefix, remove_duplicate]) Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

named_parameters([prefix, recurse]) Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

parameters([recurse]) Returns an iterator over module parameters.

register_backward_hook(hook) Registers a backward hook on the module.

register_buffer(name, tensor[, persistent]) Adds a buffer to the module.

register_forward_hook(hook) Registers a forward hook on the module.

register_forward_pre_hook(hook) Registers a forward pre-hook on the module.

register_full_backward_hook(hook) Registers a backward hook on the module.

continues on next page
Table 3 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>register_load_state_dict_post_hook(hook)</code></td>
<td>Registers a post hook to be run after module's <code>load_state_dict</code> is called.</td>
</tr>
<tr>
<td><code>register_module(name, module)</code></td>
<td>Alias for <code>add_module()</code>.</td>
</tr>
<tr>
<td><code>register_parameter(name, param)</code></td>
<td>Adds a parameter to the module.</td>
</tr>
<tr>
<td><code>requires_grad_(requires_grad)</code></td>
<td>Change if autograd should record operations on parameters in this module.</td>
</tr>
<tr>
<td><code>set_extra_state(state)</code></td>
<td>This function is called from <code>load_state_dict()</code> to handle any extra state found within the <code>state_dict</code>.</td>
</tr>
<tr>
<td><code>share_memory()</code></td>
<td>See <code>torch.Tensor.share_memory()</code></td>
</tr>
<tr>
<td><code>state_dict(*args[, destination, prefix, ...])</code></td>
<td>Returns a dictionary containing a whole state of the module.</td>
</tr>
<tr>
<td><code>to(*args, **kwargs)</code></td>
<td>Moves and/or casts the parameters and buffers.</td>
</tr>
<tr>
<td><code>to_empty(*, device)</code></td>
<td>Moves the parameters and buffers to the specified device without copying storage.</td>
</tr>
<tr>
<td><code>train([mode])</code></td>
<td>Sets the module in training mode.</td>
</tr>
<tr>
<td><code>type(dst_type)</code></td>
<td>Casts all parameters and buffers to <code>dst_type</code>.</td>
</tr>
<tr>
<td><code>xpu([device])</code></td>
<td>Moves all model parameters and buffers to the XPU.</td>
</tr>
<tr>
<td><code>zero_grad([set_to_none])</code></td>
<td>Sets gradients of all model parameters to zero.</td>
</tr>
</tbody>
</table>

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>T_destination</code></td>
<td>alias of <code>TypeVar(T_destination', bound=Dict[str, Any])</code></td>
</tr>
<tr>
<td><code>dump_patches</code></td>
<td></td>
</tr>
</tbody>
</table>

parsnip.GlobalMaxPoolingTime

class parsnip.GlobalMaxPoolingTime

Time max pooling layer for 1D sequences

This layer applies global max pooling over all channels to eliminate the channel dimension while preserving the time dimension.

**init**() → None

Initializes internal Module state, shared by both nn.Module and ScriptModule.

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong>()</td>
<td>Initializes internal Module state, shared by both nn.Module and ScriptModule.</td>
</tr>
<tr>
<td><code>add_module(name, module)</code></td>
<td>Adds a child module to the current module.</td>
</tr>
<tr>
<td><code>apply(fn)</code></td>
<td>Applies <code>fn</code> recursively to every submodule (as returned by <code>children()</code>) as well as self.</td>
</tr>
<tr>
<td><code>bfloat16()</code></td>
<td>Casts all floating point parameters and buffers to <code>bfloat16</code> datatype.</td>
</tr>
<tr>
<td><code>buffers([recurse])</code></td>
<td>Returns an iterator over module buffers.</td>
</tr>
<tr>
<td><code>children()</code></td>
<td>Returns an iterator over immediate children modules.</td>
</tr>
</tbody>
</table>
**Table 4 – continued from previous page**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cpu()</code></td>
<td>Moves all model parameters and buffers to the CPU.</td>
</tr>
<tr>
<td><code>cuda([device])</code></td>
<td>Moves all model parameters and buffers to the GPU.</td>
</tr>
<tr>
<td><code>double()</code></td>
<td>Casts all floating point parameters and buffers to double datatype.</td>
</tr>
<tr>
<td><code>eval()</code></td>
<td>Sets the module in evaluation mode.</td>
</tr>
<tr>
<td><code>extra_repr()</code></td>
<td>Set the extra representation of the module</td>
</tr>
<tr>
<td><code>float()</code></td>
<td>Casts all floating point parameters and buffers to float datatype.</td>
</tr>
<tr>
<td><code>forward(x)</code></td>
<td>Defines the computation performed at every call.</td>
</tr>
<tr>
<td><code>get_buffer(target)</code></td>
<td>Returns the buffer given by target if it exists, otherwise throws an error.</td>
</tr>
<tr>
<td><code>get_extra_state()</code></td>
<td>Returns any extra state to include in the module's state_dict.</td>
</tr>
<tr>
<td><code>get_parameter(target)</code></td>
<td>Returns the parameter given by target if it exists, otherwise throws an error.</td>
</tr>
<tr>
<td><code>get_submodule(target)</code></td>
<td>Returns the submodule given by target if it exists, otherwise throws an error.</td>
</tr>
<tr>
<td><code>half()</code></td>
<td>Casts all floating point parameters and buffers to half datatype.</td>
</tr>
<tr>
<td><code>ipu([device])</code></td>
<td>Moves all model parameters and buffers to the IPU.</td>
</tr>
<tr>
<td><code>load_state_dict(state_dict[, strict])</code></td>
<td>Copies parameters and buffers from state_dict into this module and its descendants.</td>
</tr>
<tr>
<td><code>modules()</code></td>
<td>Returns an iterator over all modules in the network.</td>
</tr>
<tr>
<td><code>named_buffers([prefix, recurse])</code></td>
<td>Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.</td>
</tr>
<tr>
<td><code>named_children()</code></td>
<td>Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.</td>
</tr>
<tr>
<td><code>named_modules([memo, prefix, remove_duplicate])</code></td>
<td>Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.</td>
</tr>
<tr>
<td><code>named_parameters([prefix, recurse])</code></td>
<td>Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.</td>
</tr>
<tr>
<td><code>parameters([recurse])</code></td>
<td>Returns an iterator over module parameters.</td>
</tr>
<tr>
<td><code>register_backward_hook(hook)</code></td>
<td>Registers a backward hook on the module.</td>
</tr>
<tr>
<td><code>register_buffer(name, tensor[, persistent])</code></td>
<td>Adds a buffer to the module.</td>
</tr>
<tr>
<td><code>register_forward_hook(hook)</code></td>
<td>Registers a forward hook on the module.</td>
</tr>
<tr>
<td><code>register_forward_pre_hook(hook)</code></td>
<td>Registers a forward pre-hook on the module.</td>
</tr>
<tr>
<td><code>register_full_backward_hook(hook)</code></td>
<td>Registers a backward hook on the module.</td>
</tr>
<tr>
<td><code>register_load_state_dict_post_hook(hook)</code></td>
<td>Registers a post hook to be run after module's load_state_dict is called.</td>
</tr>
<tr>
<td><code>register_module(name, module)</code></td>
<td>Alias for add_module().</td>
</tr>
<tr>
<td><code>register_parameter(name, param)</code></td>
<td>Adds a parameter to the module.</td>
</tr>
<tr>
<td><code>requires_grad([requires_grad])</code></td>
<td>Change if autograd should record operations on parameters in this module.</td>
</tr>
<tr>
<td><code>set_extra_state(state)</code></td>
<td>This function is called from load_state_dict() to handle any extra state found within the state_dict.</td>
</tr>
<tr>
<td><code>share_memory()</code></td>
<td>See torch.Tensor.share_memory_()</td>
</tr>
<tr>
<td><code>state_dict(*args[, destination, prefix, ...])</code></td>
<td>Returns a dictionary containing a whole state of the module.</td>
</tr>
</tbody>
</table>
### Table 4 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>to(*args, **kwargs)</code></td>
<td>Moves and/or casts the parameters and buffers.</td>
</tr>
<tr>
<td><code>to_empty(*, device)</code></td>
<td>Moves the parameters and buffers to the specified device without copying storage.</td>
</tr>
<tr>
<td><code>train([mode])</code></td>
<td>Sets the module in training mode.</td>
</tr>
<tr>
<td><code>type(dst_type)</code></td>
<td>Casts all parameters and buffers to <code>dst_type</code>.</td>
</tr>
<tr>
<td><code>xpu([device])</code></td>
<td>Moves all model parameters and buffers to the XPU.</td>
</tr>
<tr>
<td><code>zero_grad([set_to_none])</code></td>
<td>Sets gradients of all model parameters to zero.</td>
</tr>
</tbody>
</table>

#### Attributes

- `T_destination`  
  - alias of `TypeVar(T_destination', bound=Dict[str, Any])`
- `dump_patches`

### 1.6.7 Settings

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>parsnip.parse_settings(bands[, settings, ...])</code></td>
<td>Parse the settings for a ParSNIP model</td>
</tr>
<tr>
<td><code>parsnip.parse_int_list(text)</code></td>
<td>Parse a string into a list of integers</td>
</tr>
<tr>
<td><code>build_default_argparse(description)</code></td>
<td>Build an argparser object that can handle all of the ParSNIP model settings.</td>
</tr>
<tr>
<td><code>update_derived_settings(settings)</code></td>
<td>Update the derived settings for a model</td>
</tr>
<tr>
<td><code>update_settings_version(settings)</code></td>
<td>Update settings to a new version</td>
</tr>
</tbody>
</table>

#### `parsnip.parse_settings`

`parsnip.parse_settings(bands, settings={}, ignore_unknown_settings=False)`

Parse the settings for a ParSNIP model

**Parameters**

- `bands (List[str])` – Bands to use in the encoder model
- `settings (dict, optional)` – Settings to override, by default `{}`
- `ignore_unknown_settings (bool, optional)` – If False (default), raise an `KeyError` if there are any unknown settings. Otherwise, do nothing.

**Returns**

Parsed settings dictionary

**Return type**

`dict`

**Raises**

- `KeyError` – If there are unknown keys in the input settings
**parsnip.parse_int_list**

*parsnip.parse_int_list(text)*

Parse a string into a list of integers

For example, the string “1,2,3,4” will be parsed to [1, 2, 3, 4].

**Parameters**
- **text (str)** – String to parse

**Returns**
- Parsed integer list

**Return type**
- List[int]

**parsnip.build_default_argparse**

*parsnip.build_default_argparse(description)*

Build an argparse object that can handle all of the ParSNIP model settings.

The resulting parsed namespace can be passed to `parse_settings` to get a ParSNIP settings object.

**Parameters**
- **description (str)** – Description for the argument parser

**Returns**
- Argument parser with the ParSNIP model settings added as arguments

**Return type**
- ArgumentParser

**parsnip.update_derived_settings**

*parsnip.update_derived_settings(settings)*

Update the derived settings for a model

This calculate the Milky Way extinctions in each band, and determines whether background correction should be applied.

**Parameters**
- **settings (dict)** – Input settings

**Returns**
- Updated settings with derived settings calculated

**Return type**
- dict
parsnip, Release 1.3.1

parsnip.update_settings_version

parsnip.update_settings_version(**settings**)  
Update settings to a new version

Parameters

**settings** (*dict*) – Old settings

Returns

Updates settings

Return type

dict

1.6.8 Light curve utilities

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>preprocess_light_curve</td>
<td>Preprocess a light curve for the ParSNIP model</td>
</tr>
<tr>
<td>time_to_grid</td>
<td>Convert a time in the original units to one on the internal ParSNIP grid</td>
</tr>
<tr>
<td>grid_to_time</td>
<td>Convert a time on the internal grid to a time in the original units</td>
</tr>
<tr>
<td>get_band_effective_wavelength</td>
<td>Calculate the effective wavelength of a band</td>
</tr>
<tr>
<td>calculate_band_mw_extinctions</td>
<td>Calculate the Milky Way extinction corrections for a set of bands</td>
</tr>
<tr>
<td>should_correct_background</td>
<td>Determine if we should correct the background levels for a set of bands</td>
</tr>
</tbody>
</table>

parsnip.preprocess_light_curve

parsnip.preprocess_light_curve(**light_curve, settings, raise_on_invalid=True, ignore_missing_redshift=False**)  
Preprocess a light curve for the ParSNIP model

Parameters

• **light_curve** (*Table*) – Raw light curve

• **settings** (*dict*) – ParSNIP model settings

• **raise_on_invalid** (*bool*) – Whether to raise a ValueError for invalid light curves. If False, None is returned instead. By default, True.

• **ignore_missing_redshift** (*bool*) – Whether to ignore missing redshifts, by default False. If False, a missing redshift value will cause a light curve to be invalid.

Returns

Preprocessed light curve

Return type

table

Raises

* ValueError – For any invalid light curves that cannot be handled by ParSNIP if raise_on_invalid is True. The error message will describe why the light curve is invalid.*
parsnip.time_to_grid

```python
class parsnip.time_to_grid:
    def __init__(self, time, reference_time):
        # Implementation of the method
        pass
```

Convert a time in the original units to one on the internal ParSNIP grid

**Parameters**

- `time` (*float*) – Real time to convert
- `reference_time` (*float*) – Reference time for the grid

**Returns**

Time on the internal grid

**Return type**

*float*

parsnip.grid_to_time

```python
class parsnip.grid_to_time:
    def __init__(self, grid_time, reference_time):
        # Implementation of the method
        pass
```

Convert a time on the internal grid to a time in the original units

**Parameters**

- `grid_time` (*float*) – Time on the internal grid
- `reference_time` (*float*) – Reference time for the grid

**Returns**

Time in original units

**Return type**

*float*

parsnip.get_band_effective_wavelength

```python
class parsnip.get_band_effective_wavelength:
    def __init__(self, band):
        # Implementation of the method
        pass
```

Calculate the effective wavelength of a band

The results of this calculation are cached, and the effective wavelength will only be calculated once for each band.

**Parameters**

- `band` (*str*) – Name of a band in the sncosmo band registry

**Returns**

Effective wavelength of the band.

**Return type**

*float*
parsnip, Release 1.3.1

parsnip.calculate_band_mw_extinctions

parsnip.calculate_band_mw_extinctions(bands)

Calculate the Milky Way extinction corrections for a set of bands

Multiply mwebv by these values to get the extinction that should be applied to each band for a specific light curve. For bands that have already been corrected, we set this value to 0.

Parameters

bands (List[str]) – Bands to calculate the extinction for

Returns

Milky Way extinction in each band

Return type

ndarray

Raises

KeyError – If any bands are not available in band_info in instruments.py

parsnip.should_correct_background

parsnip.should_correct_background(bands)

Determine if we should correct the background levels for a set of bands

Parameters

bands (List[str]) – Bands to lookup

Returns

Boolean for each band indicating if it needs background correction

Return type

ndarray

Raises

KeyError – If any bands are not available in band_info in instruments.py

1.6.9 General utilities

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nmad(x)</td>
<td>Calculate the normalize median absolute deviation (NMAD)</td>
</tr>
<tr>
<td>frac_to_mag(fractional_difference)</td>
<td>Convert a fractional difference to a difference in magnitude.</td>
</tr>
<tr>
<td>parse_device(device)</td>
<td>Figure out which PyTorch device to use</td>
</tr>
</tbody>
</table>

parsnip.nmad

parsnip.nmad(x)

Calculate the normalize median absolute deviation (NMAD)

Parameters

x (ndarray) – Data to calculate the NMAD of

Returns

NMAD of the input
**parsnip.frac_to_mag**

```python
class FractionalDifference:
    def frac_to_mag(self, fractional_difference):
        
        Convert a fractional difference to a difference in magnitude.
        
        Because this transformation is asymmetric for larger fractional changes, we take the average of positive and negative differences.
        
        This supports numpy broadcasting.

        Parameters
        ----------
        fractional_difference : float
            Fractional flux difference

        Returns
        -------
        Difference in magnitudes

        Return type
        ----------
        float
```

**parsnip.parse_device**

```python
class ParseDevice:
    def parse_device(self, device):
        
        Figure out which PyTorch device to use

        Parameters
        ----------
        device : str
            Requested device

        Returns
        -------
        Device to use

        Return type
        ----------
        str
```

Source code: https://github.com/kboone/parsnip
predict_spectrum() *(parsnip.ParsnipModel method)*, 16
preprocess() *(parsnip.ParsnipModel method)*, 12
preprocess_light_curve() *(in module parsnip)*, 36

**R**
ResidualBlock *(class in parsnip)*, 28

**S**
save() *(parsnip.ParsnipModel method)*, 12
score() *(parsnip.ParsnipModel method)*, 14
should_correct_background() *(in module parsnip)*, 38
split_train_test() *(in module parsnip)*, 21

**T**
time_to_grid() *(in module parsnip)*, 37
to() *(parsnip.ParsnipModel method)*, 12

**U**
update_derived_settings() *(in module parsnip)*, 35
update_settings_version() *(in module parsnip)*, 36

**W**
weighted_multi_logloss() *(in module parsnip)*, 27