snfpy

Release 0.2.1+17.gfd65e38.dirty

Oct 04, 2019

Contents

CHAPTER 1

Installation and setup

1.1 Basic installation

This package requires Python >= 3.5. Assuming you have the correct version of Python installed, you can install `snfpy` by opening a terminal and running the following:

```
git clone https://github.com/rmarkello/snfpy.git
cd snfpy
python setup.py install
```

You can also install the latest release from PyPi, with:

```
pip install snfpy
```

CHAPTER 2

User guide

2.1 Brief example

A brief example for those who just want to get started:

```
# load raw data / labels for supplied dataset
>>> from snf import datasets
>>> simdata = datasets.load_simdata()
>>> sorted(simdata.keys())
['data', 'labels']

# this dataset has two data arrays representing features from 200 samples
>>> len(simdata.data)
2
>>> len(simdata.labels)
200

# convert raw data arrays into sample x sample affinity matrices
>>> from snf import compute
>>> affinities = compute.make_affinity(simdata.data, metric='euclidean')

# fuse the similarity matrices with SNF
>>> fused = compute.snf(affinities)

# estimate the number of clusters present in the fused matrix, derived via
# an "eigengap" method (i.e., largest difference in eigenvalues of the
# laplacian of the graph). note this function returns the top two options;
# we'll only use the first
>>> first, second = compute.get_n_clusters(fused)
>>> first, second
(2, 5)

# apply clustering procedure
# you can use any clustering method here, but since SNF returns an affinity
```

(continues on next page)

(continued from previous page)

```
# matrix (i.e., all entries are positively-valued and indicate similarity)
# spectral clustering makes a lot of sense
>>> from sklearn import cluster
>>> fused_labels = cluster.spectral_clustering(fused, n_clusters=first)

# compute normalized mutual information for clustering solutions
>>> from snf import metrics
>>> labels = [simdata.labels, fused_labels]
>>> for arr in affinities:
...     labels += [cluster.spectral_clustering(arr, n_clusters=first)]
>>> nmi = metrics.nmi(labels)

# compute silhouette score to assess goodness-of-fit for clustering
>>> silhouette = metrics.silhouette_score(fused, fused_labels)
```

2.2 In-depth example

Using SNF is pretty straightforward. There are only a handful of commands that you'll need, and the output (a subject x subject array) can easily be carried forward to any number of analysis pipelines.

Nonetheless, for a standard scenario, this package comes bundled with two datasets provided by the original authors of SNF which can be quite illustrative.

First, we'll load in the data; data arrays should be (N x M), where *N* is samples and *M* are features.

```
>>> from snf import datasets
>>> simdata = datasets.load_simdata()
>>> sorted(simdata.keys())
['data', 'labels']
```

The loaded object `simdata` is a dictionary with two keys containing our data arrays and the corresponding labels:

```
>>> n_dtypes = len(simdata.data)
>>> n_samp = len(simdata.labels)
>>> print('Simdata has {} datatypes with {} samples each.'.format(n_dtypes, n_samp))
Simdata has 2 datatypes with 200 samples each.
```

Once we have our data arrays loaded we need to create affinity matrices. Unlike distance matrices, a higher number in an affinity matrix indicates increased similarity. Thus, the highest numbers should always be along the diagonal, since subjects are always most similar to themselves!

To construct our affinity matrix, we'll use `snf.make_affinity`, which first constructs a distance matrix (using a provided distance metric) and then converts this into an affinity matrix based on a given subject's similarity to their `K` nearest neighbors. As such, we need to provide a few hyperparameters: `K` and `mu`. `K` determines the number of nearest neighbors to consider when constructing the affinity matrix; `mu` is a scaling factor that weights the affinity matrix. While the appropriate numbers for these varies based on scenario, a good rule is that `K` should be around `N // 10`, and `mu` should be in the range (0.2 - 0.8).

```
>>> from snf import compute
>>> affinities = compute.make_affinity(simdata.data, metric='euclidean', K=20, mu=0.5)
```

Note that we specified `metric='euclidean'`, specifying that we wanted to use euclidean distance in the generation of the initial distance array before constructing the affinity matrix.

Once we have our affinity arrays, we can run them through the SNF algorithm. We need to carry forward our `K` hyperparameter to this algorithm, as well.

```
>>> fused = compute.snf(affinities, K=20)
```

The array output by SNF is a fused affinity matrix; that is, it represents data from all the inputs. It's designed to be full rank, and can thus be subjected to clustering and classification. We'll do the former, now, by estimating the number of clusters in the data via the "eigengap" method:

```
>>> first, second = compute.get_n_clusters(fused)
>>> first, second
(2, 5)
```

By default, `compute.get_n_clusters` returns two values. We'll use the first for our clustering:

```
>>> from sklearn import cluster
>>> fused_labels = cluster.spectral_clustering(fused, n_clusters=first)
```

Now we can compare the clustering of our fused matrix to what would happen if we had used the data from either of the original matrices, individually. To do this we need to generate cluster labels from the individual affinity matrices:

```
>>> labels = [simdata.labels, fused_labels]
>>> for arr in affinities:
...     labels += [cluster.spectral_clustering(arr, n_clusters=first)]
```

Then, we can calculate the normalized mutual information score (NMI) between the labels generated by SNF and the ones we just obtained:

```
>>> from snf import metrics
>>> nmi = metrics.nmi(labels)
>>> print(nmi)
[[1.         1.         0.25266274 0.07818002]
 [1.         1.         0.25266274 0.07818002]
 [0.25266274 0.25266274 1.         0.0355961 ]
 [0.07818002 0.07818002 0.0355961  1.        ]]
```

The output array is symmetric and the values range from 0 to 1, where 0 indicates no overlap and 1 indicates a perfect correspondence between the two sets of labels.

The entry in (0, 1) indicates that the fused array generated by SNF has perfect overlap with the "true" labels from the datasets. The entries in (0, 2) and (0, 3) indicate the shared information from the individual (unfused) data arrays (`simdata.data`) with the true labels.

While this example has the true labels to compare against, in unsupervised clustering we would not have such information. In these instances, the NMI cannot tell us that the fused array is **superior** to the individual data arrays. Rather, it can only help distinguish how much data from each of the individual arrays is contributing to the fused network.

We can also assess how well the clusters are defined using the silhouette score. These values range from -1 to 1, where -1 indicates a poor clustering solution and 1 indicates a fantastic solution. We set the diagonal of the fused network to zero before construction because it was artifically inflated during the fusion process; thus, this returns a *conservative* estimate of the cluster goodness-of-fit.

```
>>> import numpy as np
>>> np.fill_diagonal(fused, 0)
>>> sil = metrics.silhouette_score(fused, fused_labels)
>>> print('Silhouette score for the fused matrix is: {:.2f}'.format(sil))
Silhouette score for the fused matrix is: 0.28
```

This indicates that the clustering solution for the data is not too bad! We could try playing around with the hyperparameters to see if we can improve our fit (being careful to do so in a way that won't overfit to the data). It's worth noting that the silhouette score here is slightly modified to deal with the fact that we're working with affinity matrices instead of distance matrices. See the *API reference* for more information.

CHAPTER 3

Reference API

This is the primary reference of `snfpy`. Please refer to the *user guide* for more information on how to best implement these functions in your own workflows.

List of modules

- `snf.compute` - *Primary SNF functionality*
- `snf.metrics` - *Evaluation metrics*
- `snf.cv` - *Cross-validation functions*
- `snf.datasets` - *Load tests datasets*

3.1 `snf.compute` - Primary SNF functionality

Contains the primary functions for conducting similarity network fusion workflows.

make_affinity(*data[, metric, K, mu, normalize])	Constructs affinity (i.e., similarity) matrix from *data*
get_n_clusters(arr[, n_clusters])	Finds optimal number of clusters in *arr* via eigengap method
snf(*aff[, K, t, alpha])	Performs Similarity Network Fusion on *aff* matrices
group_predict(train, test, labels, *[, K, mu, t])	Propagates *labels* from *train* data to *test* data via SNF

3.1.1 snf.compute.make_affinity

`snf.compute.`**`make_affinity`**(**data*, *metric='sqeuclidean'*, *K=20*, *mu=0.5*, *normalize=True*)

Constructs affinity (i.e., similarity) matrix from *data*

Performs columnwise normalization on *data*, computes distance matrix based on provided *metric*, and then

constructs affinity matrix. Uses a scaled exponential similarity kernel to determine the weight of each edge based on the distance matrix. Optional hyperparameters *K* and *mu* determine the extent of the scaling (see *Notes*).

Parameters

- ***data** (*(N, M) array_like*) – Raw data array, where *N* is samples and *M* is features. If multiple arrays are provided then affinity matrices will be generated for each.
- **metric** (*str or list-of-str, optional*) – Distance metric to compute. Must be one of available metrics in :py:func‘scipy.spatial.distance.pdist‘. If multiple arrays a provided an equal number of metrics may be supplied. Default: ‘sqeuclidean’
- **K** (*(0, N) int, optional*) – Number of neighbors to consider when creating affinity matrix. See *Notes* of :py:func‘snf.compute.affinity_matrix‘ for more details. Default: 20
- **mu** (*(0, 1) float, optional*) – Normalization factor to scale similarity kernel when constructing affinity matrix. See *Notes* of :py:func‘snf.compute.affinity_matrix‘ for more details. Default: 0.5
- **normalize** (*bool, optional*) – Whether to normalize (i.e., zscore) *arr* before constructing the affinity matrix. Each feature (i.e., column) is normalized separately. Default: True

Returns **affinity** – Affinity matrix (or matrices, if multiple inputs provided)

Return type (N, N) numpy.ndarray or list of numpy.ndarray

Notes

The scaled exponential similarity kernel, based on the probability density function of the normal distribution, takes the form:

$$\mathbf{W}(i,j) = \frac{1}{\sqrt{2\pi\sigma^2}}\, exp^{-\frac{\rho^2(x_i,x_j)}{2\sigma^2}}$$

where $\rho(x_i, x_j)$ is the Euclidean distance (or other distance metric, as appropriate) between patients x_i and x_j. The value for
$sigma$ is calculated as:

$$\sigma = \mu\, \frac{\overline{\rho}(x_i, N_i) + \overline{\rho}(x_j, N_j) + \rho(x_i, x_j)}{3}$$

where $\overline{\rho}(x_i, N_i)$ represents the average value of distances between x_i and its neighbors $N_{1..K}$, and $\mu \in (0, 1) \subset \mathbb{R}$.

Examples

```
>>> from snf import datasets
>>> simdata = datasets.load_simdata()
```

```
>>> from snf import compute
>>> aff = compute.make_affinity(simdata.data[0], K=20, mu=0.5)
>>> aff.shape
(200, 200)
```

3.1.2 snf.compute.get_n_clusters

snf.compute.**get_n_clusters**(*arr*, *n_clusters=range(2, 6)*)
Finds optimal number of clusters in *arr* via eigengap method

Parameters

- **arr** (*(N, N) array_like*) – Input array (e.g., the output of :py:func‘snf.compute.snf‘)
- **n_clusters** (*array_like*) – Numbers of clusters to choose between

Returns

- **opt_cluster** (*int*) – Optimal number of clusters
- **second_opt_cluster** (*int*) – Second best number of clusters

3.1.3 snf.compute.snf

snf.compute.**snf**(**aff*, *K=20*, *t=20*, *alpha=1.0*)
Performs Similarity Network Fusion on *aff* matrices

Parameters

- ***aff** (*(N, N) array_like*) – Input similarity arrays; all arrays should be square and of equal size.
- **K** (*(0, N) int, optional*) – Hyperparameter normalization factor for scaling. Default: 20
- **t** (*int, optional*) – Number of iterations to perform information swapping. Default: 20
- **alpha** (*(0, 1) float, optional*) – Hyperparameter normalization factor for scaling. Default: 1.0

Returns **W** – Fused similarity network of input arrays

Return type (N, N) np.ndarray

Notes

In order to fuse the supplied m arrays, each must be normalized. A traditional normalization on an affinity matrix would suffer from numerical instabilities due to the self-similarity along the diagonal; thus, a modified normalization is used:

$$\mathbf{P}(i,j) = \begin{cases} \frac{\mathbf{W}(i,j)}{2\sum_{k\neq i}\mathbf{W}(i,k)}, & j \neq i \\ 1/2, & j = i \end{cases}$$

Under the assumption that local similarities are more important than distant ones, a more sparse weight matrix is calculated based on a KNN framework:

$$\mathbf{S}(i,j) = \begin{cases} \frac{\mathbf{W}(i,j)}{\sum_{k\in N_i}\mathbf{W}(i,k)}, & j \in N_i \\ 0, & \text{otherwise} \end{cases}$$

The two weight matrices $\mathbf{P}$ and $\mathbf{S}$ thus provide information about a given patient's similarity to all other patients and the *K* most similar patients, respectively.

These m matrices are then iteratively fused. At each iteration, the matrices are made more similar to each other via:

$$\mathbf{P}^{(v)} = \mathbf{S}^{(v)} \times \frac{\sum_{k \neq v} \mathbf{P}^{(k)}}{m-1} \times (\mathbf{S}^{(v)})^T, v = 1, 2, ..., m$$

After each iteration, the resultant matrices are normalized via the equation above. Fusion stops after *t* iterations, or when the matrices $\mathbf{P}^{(v)}, v = 1, 2, ..., m$ converge.

The output fused matrix is full rank and can be subjected to clustering and classification.

3.1.4 snf.compute.group_predict

snf.compute.**group_predict**(*train*, *test*, *labels*, ***, *K=20*, *mu=0.4*, *t=20*)

Propagates *labels* from *train* data to *test* data via SNF

Parameters

- **train** (*m*-list of (S1, F) array_like) – Input subject x feature training data. Subjects in these data sets should have been previously labelled (see: *labels*).
- **test** (*m*-list of (S2, F) array_like) – Input subject x feature testing data. These should be similar to the data in *train* (though the first dimension can differ). Labels will be propagated to these subjects.
- **labels** (*(S1,) array_like*) – Cluster labels for *S1* subjects in *train* data sets. These could have been obtained from some ground-truth labelling or via a previous iteration of SNF with only the *train* data (e.g., the output of `sklearn.cluster.spectral_clustering()` would be appropriate).
- **K** (*(0, N) int, optional*) – Hyperparameter normalization factor for scaling. See *Notes* of *snf.affinity_matrix* for more details. Default: 20
- **mu** (*(0, 1) float, optional*) – Hyperparameter normalization factor for scaling. See *Notes* of *snf.affinity_matrix* for more details. Default: 0.5
- **t** (*int, optional*) – Number of iterations to perform information swapping during SNF. Default: 20

Returns **predicted_labels** – Cluster labels for subjects in *test* assigning to groups in *labels*

Return type (S2,) np.ndarray

3.2 snf.metrics - Evaluation metrics

Functions for computing various metrics to aid interpretation of similarity network fusion outputs.

nmi(labels)	Calculates normalized mutual information for all combinations of *labels*
rank_feature_by_nmi(inputs, W, *[, K, mu, ...])	Calculates NMI of each feature in *inputs* with *W*
silhouette_score(arr, labels)	Calculates modified silhouette score from affinity matrix
affinity_zscore(arr, labels[, n_perms, seed])	Calculates z-score of silhouette (affinity) score by permutation

3.2.1 snf.metrics.nmi

snf.metrics.**nmi**(*labels*)

Calculates normalized mutual information for all combinations of *labels*

Uses `sklearn.metrics.v_measure_score()` for calculation; refer to that codebase for information on algorithm.

Parameters **labels** (*m-length list of (N,) array_like*) – List of label arrays

Returns **nmi** – NMI score for all combinations of *labels*

Return type (m x m) np.ndarray

Examples

```
>>> import numpy as np
>>> label1 = np.array([1, 1, 1, 2, 2, 2])
>>> label2 = np.array([1, 1, 2, 2, 2, 2])
```

```
>>> from snf import metrics
>>> metrics.nmi([label1, label2])
array([[1.        , 0.47870397],
       [0.47870397, 1.        ]])
```

3.2.2 snf.metrics.rank_feature_by_nmi

snf.metrics.**rank_feature_by_nmi**(*inputs*, *W*, ***, *K=20*, *mu=0.5*, *n_clusters=None*)

Calculates NMI of each feature in *inputs* with *W*

Parameters

- **inputs** (*list-of-tuple*) – Each tuple should contain (1) an (N, M) data array, where N is samples M is features, and (2) a string indicating the metric to use to compute a distance matrix for the given data. This MUST be one of the options available in `scipy.spatial.distance.cdist()`
- **W** (*(N, N) array_like*) – Similarity array generated by *snf.compute.snf()*
- **K** (*(0, N) int, optional*) – Hyperparameter normalization factor for scaling. Default: 20
- **mu** (*(0, 1) float, optional*) – Hyperparameter normalization factor for scaling. Default: 0.5
- **n_clusters** (*int, optional*) – Number of desired clusters. Default: determined by eigengap (see *snf.get_n_clusters()*)

Returns **nmi** – Normalized mutual information scores for each feature of input arrays

Return type list of (M,) np.ndarray

3.2.3 snf.metrics.silhouette_score

snf.metrics.**silhouette_score**(*arr*, *labels*)

Calculates modified silhouette score from affinity matrix

The Silhouette Coefficient is calculated using the mean intra-cluster affinity (*a*) and the mean nearest-cluster affinity (*b*) for each sample. The Silhouette Coefficient for a sample is *(b - a) / max(a,b)*. To clarify, *b* is the distance between a sample and the nearest cluster that the sample is not a part of. This corresponds to the cluster with the next *highest* affinity (opposite how this metric would be computed for a distance matrix).

Parameters

- **arr** (*(N, N) array_like*) – Array of pairwise affinities between samples
- **labels** (*(N,) array_like*) – Predicted labels for each sample

Returns **silhouette_score** – Modified (affinity) silhouette score

Return type float

Notes

Code is *lightly* modified from the `sklearn` implementation. See: *sklearn.metrics.silhouette_score*

3.2.4 snf.metrics.affinity_zscore

snf.metrics.**affinity_zscore**(*arr*, *labels*, *n_perms=1000*, *seed=None*)

Calculates z-score of silhouette (affinity) score by permutation

Parameters

- **arr** (*(N, N) array_like*) – Array of pairwise affinities between samples
- **labels** (*(N,) array_like*) – Predicted labels for each sample
- **n_perms** (*int, optional*) – Number of permutations. Default: 1000
- **seed** (*int, optional*) – Random seed. Default: None

Returns **z_aff** – Z-score of silhouette (affinity) score

Return type float

3.3 snf.cv - Cross-validation functions

Code for implementing cross-validation of similarity network fusion. Useful for determining the "optimal" number of clusters in a dataset within a cross-validated, data-driven framework.

snf_gridsearch(*data[, metric, mu, K, ...])	Performs grid search for SNF hyperparameters *mu*, *K*, and *n_clusters*
get_optimal_params(zaff, labels[, neighbors])	Finds optimal parameters for SNF based on K-folds grid search

3.3.1 snf.cv.snf_gridsearch

snf.cv.**snf_gridsearch**(**data*, *metric='sqeuclidean'*, *mu=None*, *K=None*, *n_clusters=None*, *t=20*, *folds=3*, *n_perms=1000*, *normalize=True*, *seed=None*)

Performs grid search for SNF hyperparameters *mu*, *K*, and *n_clusters*

Uses *folds*-fold CV to subsample *data* and performs grid search on *mu*, *K*, and *n_clusters* hyperparameters for SNF. There is no testing on the left-out sample for each CV fold—it is simply removed.

Parameters

- ***data** (*(N, M) array_like*) – Raw data arrays, where *N* is samples and *M* is features.
- **metric** (*str or list-of-str, optional*) – Distance metrics to compute on *data*. Must be one of available metrics in `scipy.spatial.distance.pdist`. If a list is provided for *data* a list of equal length may be supplied here. Default: 'sqeuclidean'
- **mu** (*array_like, optional*) – Array of *mu* values to search over. Default: np.arange(0.35, 1.05, 0.05)
- **K** (*array_like, optional*) – Array of *K* values to search over. Default: np.arange(5, N // 2, 5)
- **n_clusters** (*array_like, optional*) – Array of cluster numbers to search over. Default: np.arange(2, N // 20)
- **t** (*int, optional*) – Number of iterations for SNF. Default: 20
- **folds** (*int, optional*) – Number of folds to use for cross-validation. Default: 3
- **n_perms** (*int, optional*) – Number of permutations for generating z-score of silhouette (affinity) to assess reliability of SNF clustering output. Default: 1000
- **normalize** (*bool, optional*) – Whether to normalize (z-score) *data* arrrays before constructing affinity matrices. Each feature is separately normalized. Default: True
- **seed** (*int, optional*) – Random seed. Default: None

Returns

- **grid_zaff** (*(F,) list of (S, K, C) np.ndarray*) – Where *S* is mu, *K* is K, *C* is n_clusters, and *F* is the number of folds for CV. The entries in the individual arrays correspond to the z-scored silhouette (affinity).
- **grid_labels** (*(F,) list of (S, K, C, N) np.ndarray*) – Where *S* is mu, *K* is K, *C* is n_clusters, and *F* is the number of folds for CV. The *N* entries along the last dimension correspond to the cluster labels for the given parameter combination.

3.3.2 snf.cv.get_optimal_params

`snf.cv.`**get_optimal_params**(*zaff*, *labels*, *neighbors='edges'*)

Finds optimal parameters for SNF based on K-folds grid search

Parameters

- **zaff** (*(F,) list of (S, K, C) np.ndarray*) – Where *S* is mu, *K* is K, *C* is n_clusters, and *F* is the number of folds for CV. The entries in the individual arrays correspond to the z-scored silhouette (affinity).
- **labels** (*(F,) list of (S, K, C, N) np.ndarray*) – Where *S* is mu, *K* is K, *C* is n_clusters, and *F* is the number of folds for CV. The *N* entries along the last dimension correspond to the cluster labels for the given parameter combination.
- **neighbors** (*str, optional*) – How many neighbors to consider when calculating z-Rand kernel. Must be in ['edges', 'corners']. Default: 'edges'

Returns

- **mu** (*int*) – Index along S indicating optimal mu parameter
- **K** (*int*) – Index along K indicating optimal K parameter

3.4 snf.datasets - Load tests datasets

Functions for loading test data setss

load_simdata()	Loads “similarity” data with two datatypes
load_digits()	Loads “digits” dataset with four datatypes

3.4.1 snf.datasets.load_simdata

snf.datasets.**load_simdata**()
Loads “similarity” data with two datatypes

Returns **sim** – Dictionary-like object with keys [‘data’, ‘labels’]

Return type sklearn.utils.Bunch

3.4.2 snf.datasets.load_digits

snf.datasets.**load_digits**()
Loads “digits” dataset with four datatypes

Returns **digits** – Dictionary-like object with keys [‘data’, ‘labels’]

Return type sklearn.utils.Bunch

Python Module Index

S

Index