skccm Documentation

Release 0.1

Nick Cortale

Oct 09, 2018

Contents:

Scikit Convergent Cross Mapping

Scikit Convergent Cross Mapping (skccm) can be used as a way to detect causality between time series.

For a quick explanation of this package, I suggest checking out the *Quick Example* section as well as the wikipedia article on convergent cross mapping . Additionally, Dr. Sugihara's lab has produced some good summary videos about the topic:

1. Time Series and Dynamic Manifolds
2. Reconstructed Shadow Manifold
3. State Space Reconstruction: Convergent Cross Mapping

For a more complete background, I suggest checking out the following papers:

1. Detecting Causality in Complex Ecosystems by Sugihara
2. Distinguishing time-delayed causal interactions using convergent cross mapping by Ye

Sugihara also has a good talk about about Correlation and Causation.

CHAPTER 1

Install

1.1 pip

```
pip install skccm
```

1.2 Conda (Recommended)

To create a conda environment, you can use the following conda environment.yml file:

```
name: skccm_env
dependencies:
  - python=3
  - numpy
  - scikit-learn
  - scipy
  - pip:
    - skccm
```

Then you can simply create the environment with:

```
conda env create -f environment.yml
```

And activate it with:

```
source activate skccm_env
```

1.3 Contribute, Report Issues, Support

To contribute, we suggest making a pull request.

To report issues, we suggest opening an issue.

For support, email cortalen@uncw.edu.

CHAPTER 2

Quick Example

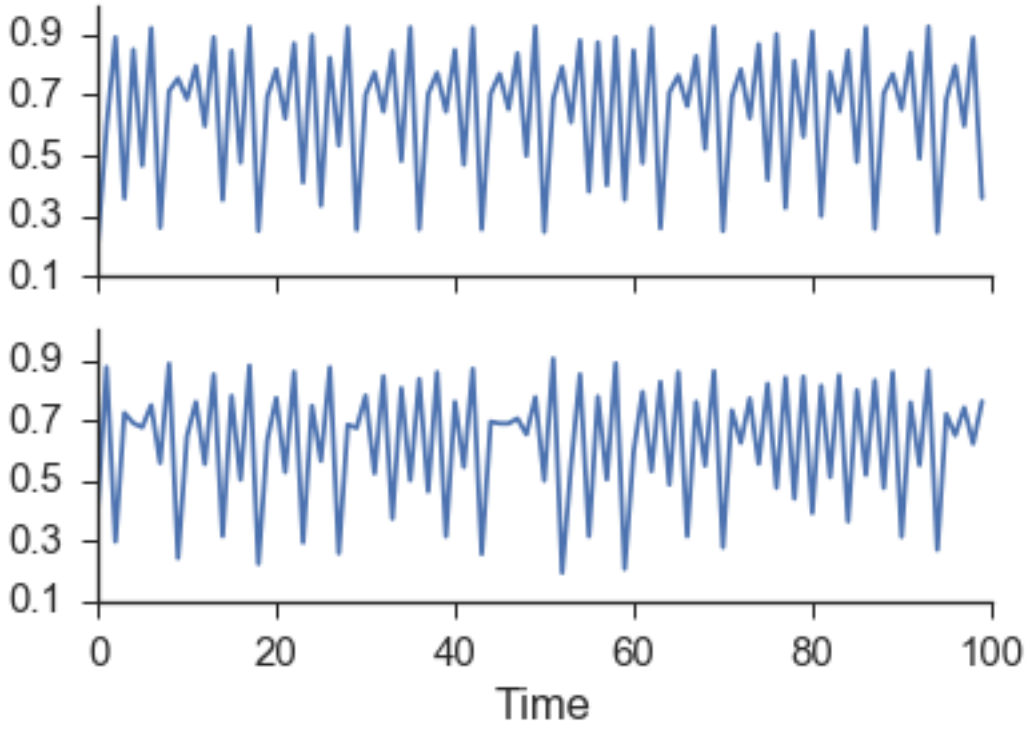
In order to illustrate how this package works, we will use the coupled logistic map from the original convergent cross mapping paper. The equations the take form:

$$X(t+1) = X(t)[r_x - r_x X(t) - \beta_{x,y} Y(t)]$$
$$Y(t+1) = Y(t)[r_y - r_y Y(t) - \beta_{y,x} X(t)]$$

Notice that $\beta_{x,y}$ controls the amount of information from the Y time series that is being injected into the X time series. Likewise, $\beta_{y,x}$ controls the amount of information injected into the Y time series from the X time series. These parameters control how much one series influences the other. There is a function in `skccm.data` to reproduce these time series. For example:

```
import skccm.data as data

rx1 = 3.72 #determines chaotic behavior of the x1 series
rx2 = 3.72 #determines chaotic behavior of the x2 series
b12 = 0.2 #Influence of x1 on x2
b21 = 0.01 #Influence of x2 on x1
ts_length = 1000
x1,x2 = data.coupled_logistic(rx1,rx2,b12,b21,ts_length)
```

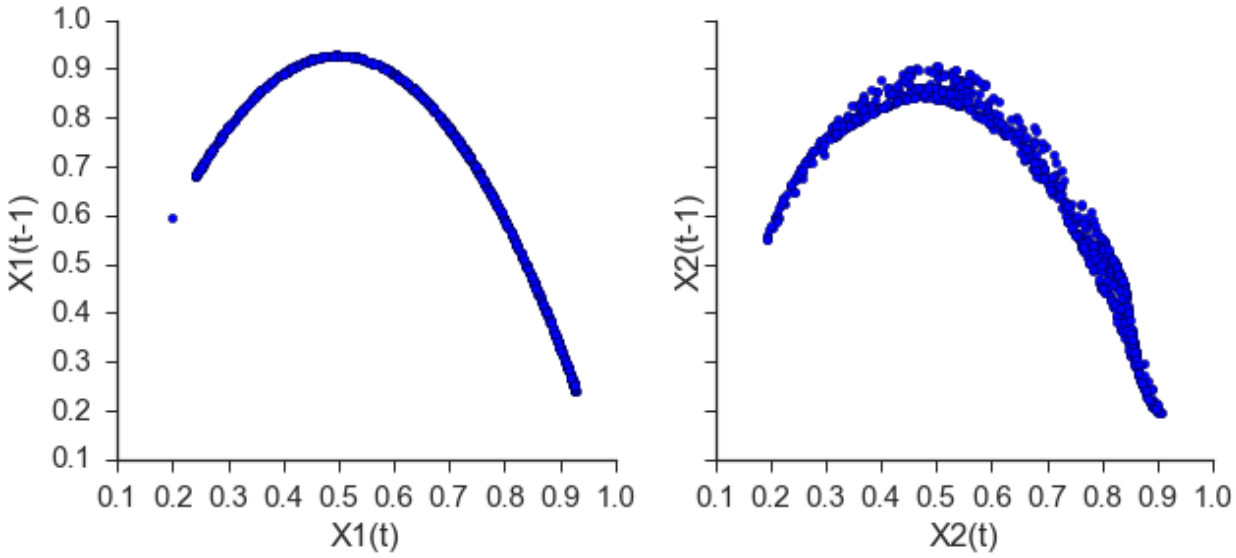
Here we opt to use `x1` and `x2` instead of `X` and `Y`, but the equations are the same. Using these parameters, `x1` has more of an influence on `x2` than `x2` has on `x1`. This produces the coupled logistic map as seen in the figure below where the top plot is `x1` and the bottom is `x2`.



As is clearly evident from the figure above, there is no way to tell if one series is influencing the other just by examining the time series.

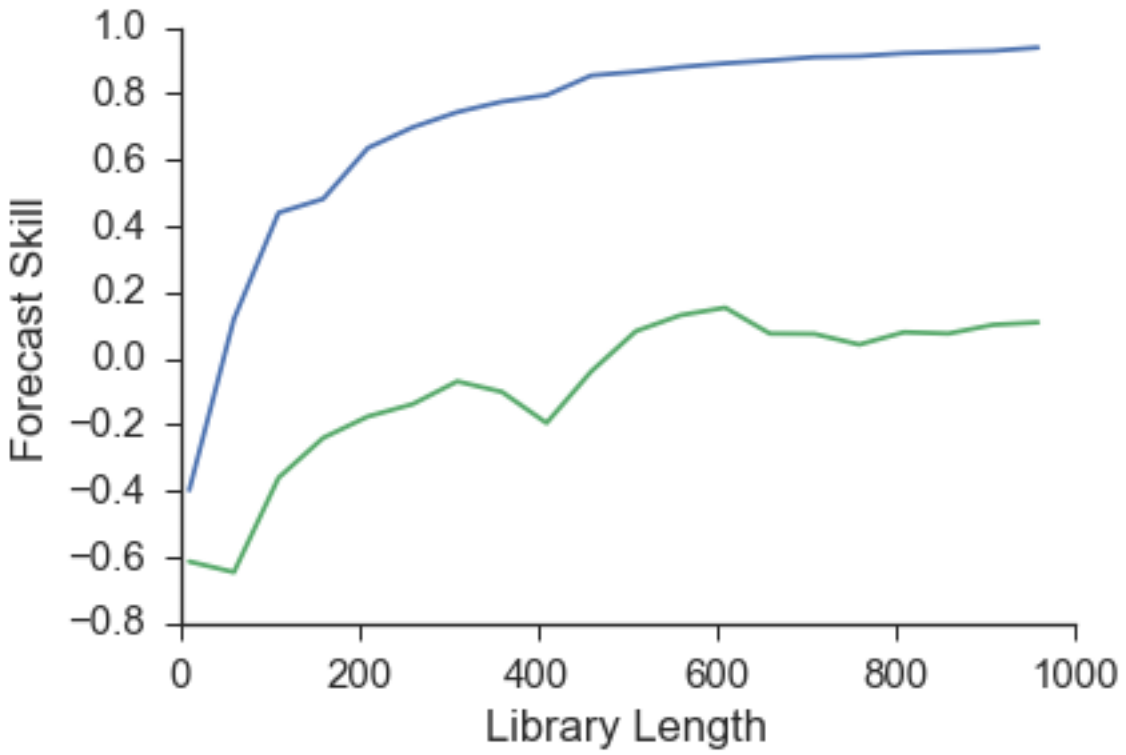
The next step is to embed both time series. An in-depth discussion about appropriately picking the lag and the embedding dimension can be found in the *Embed* section. For this example, an embedding dimension of 2 and a lag of 1 successfully rebuilds the shadow manifold.

```
import skccm as ccm
lag = 1
embed = 2
e1 = ccm.Embed(x1)
e2 = ccm.Embed(x2)
X1 = e1.embed_vectors_1d(lag,embed)
X2 = e2.embed_vectors_1d(lag,embed)
```



Now that we have embed the time series, all that is left to do is check the forecast skill as a function of library length. This package diverges from the paper above in that a training set is used to rebuild the shadow manifold and the testing set is used to see if nearby points on one manifold can be used to make accurate predictions about the other manifold. This removes the problem of autocorrelated time series. The original implementation can be found in `ccm.paper`.

```
from skccm.utilities import train_test_split

#split the embedded time series
x1tr, x1te, x2tr, x2te = train_test_split(X1,X2, percent=.75)

CCM = ccm.CCM() #initiate the class

#library lengths to test
len_tr = len(x1tr)
lib_lens = np.arange(10, len_tr, len_tr/20, dtype='int')

#test causation
CCM.fit(x1tr,x2tr)
x1p, x2p = CCM.predict(x1te, x2te,lib_lengths=lib_lens)

sc1,sc2 = CCM.score()
```



As can be seen from the image above, `x1` has a higher prediction skill. Another way to view this is that information about `x1` is present in the `x2` time series. This leads to better forecasts for `x1` using `x2`'s reconstructed manifold. This means that `x1` is driving `x2` which is exactly how we set the initial conditions when we generated these time series.

To make sure that this algorithm is robust we test a range of β values similar to the original paper. The results below show the difference between `sc1` and `sc2`.

2.1 Summary

The workflow for convergent cross mapping is as follows:

1. Calculate the mutual information of both time series to find the appropriate lag value
2. Embed the time series using the calculated lag and best embedding dimension
3. Split each embedded time series into a training set and testing set
4. Calculate the distance from each test sample to each training sample
5. Use the near neighbor time indices from X_1 to make a prediction about X_2
6. Repeat the prediction for multiple library lengths
7. Evaluate the predictions

1. Calculate mutual information for both time series to find the appropriate lag value.

Mutual information is used as a way to jump far enough in time that new information about the system can be gained. A similar idea is calculating the autocorrelation. Systems that don't change much from one time step to the next would have higher autocorrelation and thus a larger lag value would be necessary to gain new information about the system. It turns out that using mutual information over autocorrelation allows for better predictions to be made [CITATION].

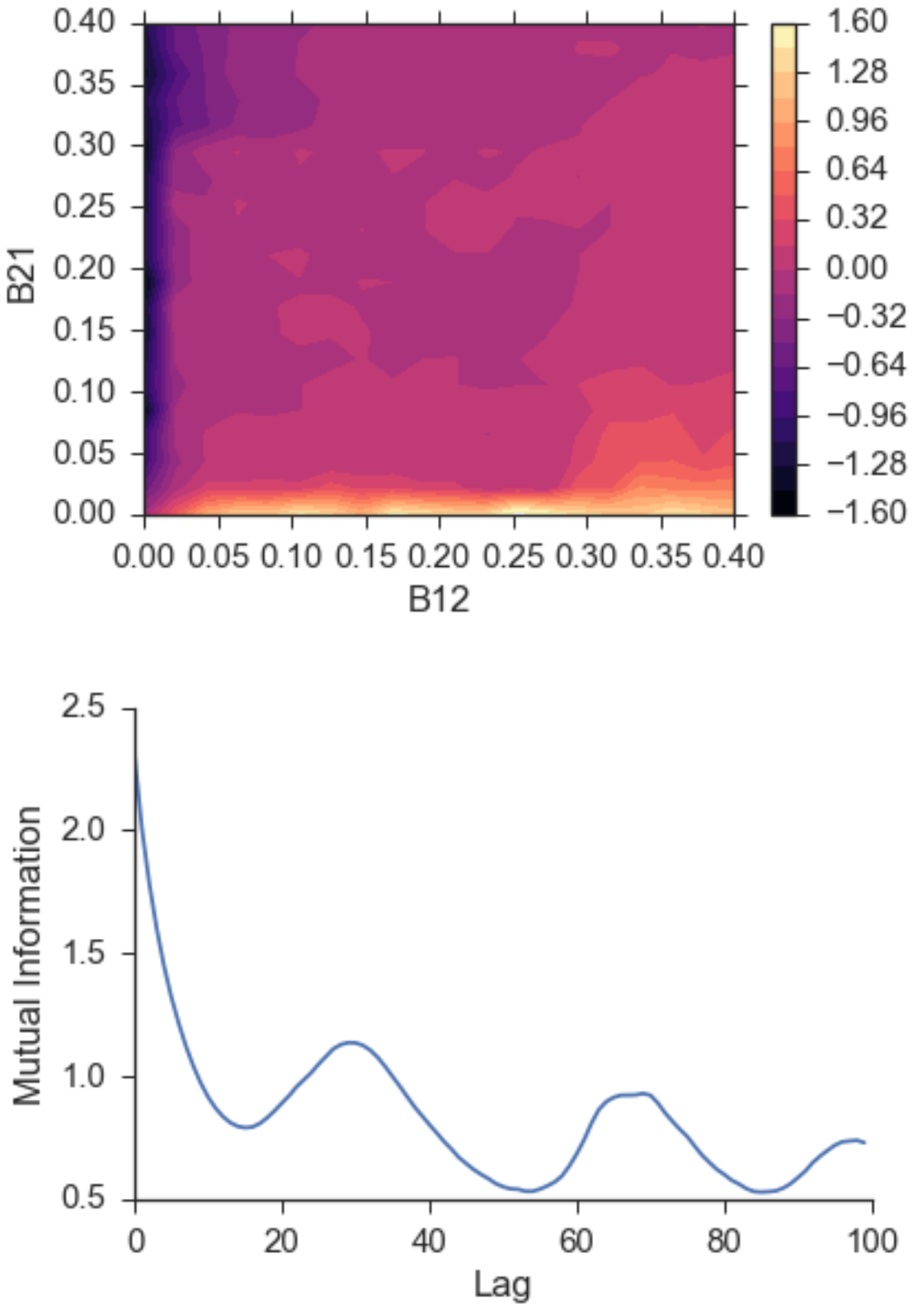


Figure: The image above shows the mutual information for the $$x$$ values of the lorenz time series. We can see a minimum around 16.

2. Determine the embedding dimension by finding which gives the highest prediction skill.

Ideally you want to find the best embedding dimension for a specific time series. A good rule of thumb is to use an embedding dimension of three as your first shot. After the initial analysis, you can tweak this hyperparameter until you achieve the best prediction skill.

Alternatively, you can use a [false near neighbor][fnn] test when the reconstructed attractor is fully "unfolded". This functionality is not in skccm currently, but will be added in the future.

Figure: An example of an embedding dimension of three and a lag of two.

3. Split each embedded time series into a training set and testing set.

This protects against highly autocorrelated time series. For example, random walk time series can seem like they are coupled if they are not split into a training set and testing set.

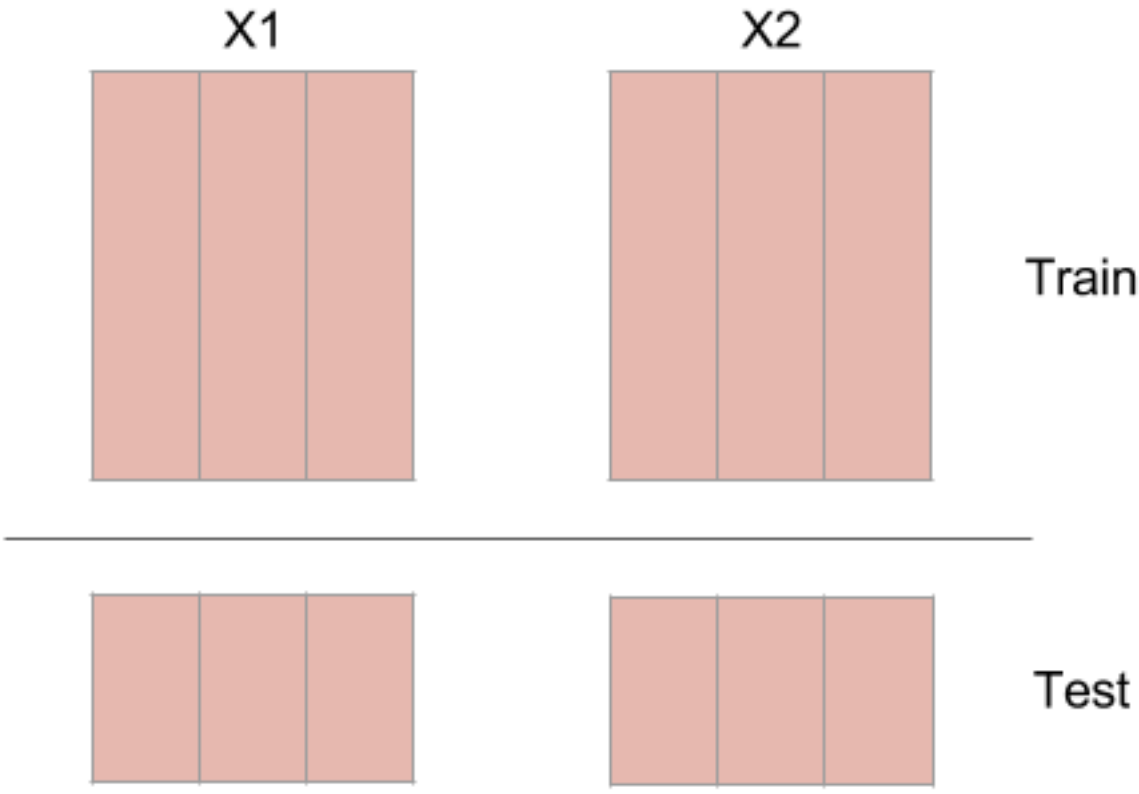


Figure: Splitting an embedded time series into a training set and a testing set.

5. Calculate the distance from each test sample to each training sample

At this point, you will have these four embedded time series:

1. X1tr
2. X1te
3. X2tr

4. X2te

The distance is calculated from every sample in X1te to every sample in X1tr. The same is then done for X2tr and X2te. The distances are then sorted and the closest $$k$$ indices are kept to make a prediction in the next step. $$k$$ is the embedding dimension plus 1. So if your embedding dimension was three, then the amount of near neighbors used to make a prediction will be four.

6. Use the near neighbor time indices from $$X_1$$ to make a prediction about $$X_2$$

The next step is to use the near neighbor indices and weights to make a prediction about the other time series. The indices that were found by calculating the distance from every sample in X1te to every sample in X1tr, are used on X2tr to make a prediction about X2te. This seems a little counterintuitive, but it is expected that if one time series influences the other, the system being forced should be in a similar state when the system doing the forcing is in a certain configuration.

INSERT THOUGHT EXPERIMENT

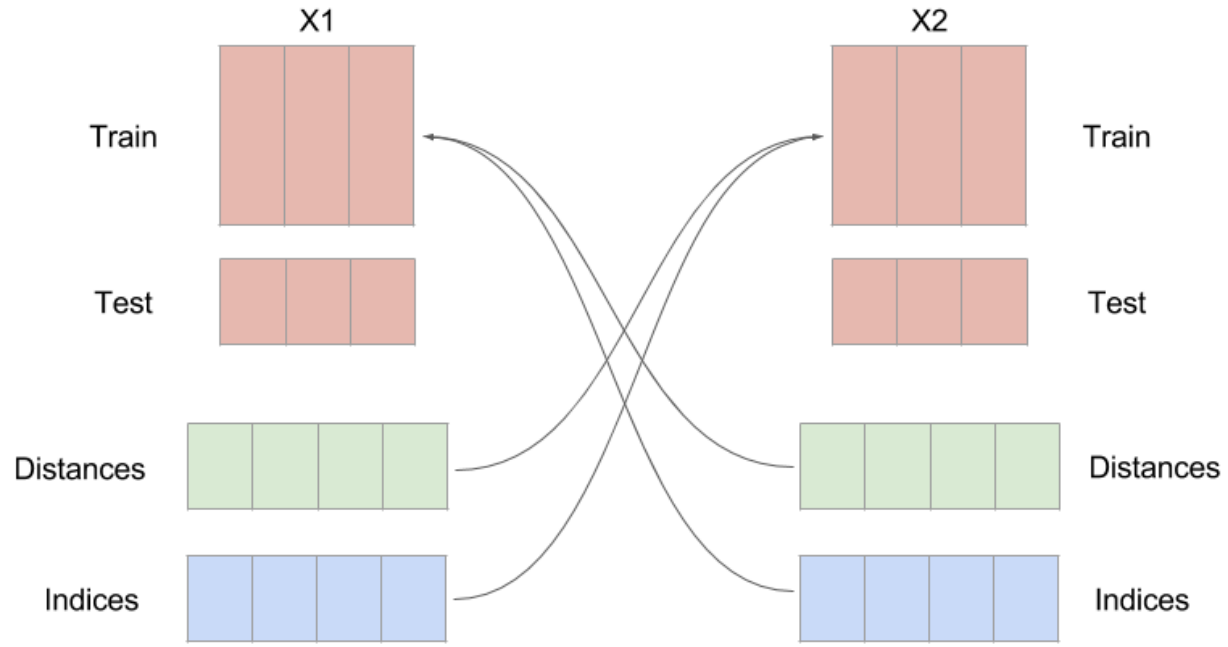


Figure: An example of switching the indices. Notice the distances and indices have the same number of samples as the testing set, but an extra dimension. This is because you need $$K+1$$ near neighbors in order to surround a point.

7. Repeat the prediction for multiple library lengths

The hope is we see convergence as the library length is increased. By increasing the library length, the density of the rebuilt attractor is increasing. As that attractor becomes more and more populated, better predictions should be able to be made.

8. Finally, evaluate the predictions

The way the predictions are evaluated in the paper is by using the [$$R^2$$][r2] (coefficient of determination) value between the predictions and the actual value. This is done for all the predictions at multiple library lengths. If the predictions for $$X_1$$ are better than $$X_2$$ than it is said that $$X_1$$ influences $$X_2$$.

Caveats

- Simple attractors can fool this technique (sine waves)
- Can't be used on non-steady state time series.
- Lorenz equation doesn't work?

[paper]: http://science.sciencemag.org/content/338/6106/496 [skccm]:https://github.com/NickC1/skccm [r2]: https://www.wikiwand.com/en/Coefficient_of_determination [fnn]: http://www.mpipks-dresden.mpg.de/~tisean/TISEAN_2.1/docs/chaospaper/node9.html

CHAPTER 3

Generate Data

skccm comes with function to generate test data. Quickly explore the behavior of different systems.

skccm.data.**coupled_logistic**(*rx1*, *rx2*, *b12*, *b21*, *ts_length*, *random_start=False*)

Coupled logistic map.

Parameters

- **rx1** (*float*) – Parameter that determines chaotic behavior of the x1 series.
- **rx2** (*float*) – Parameter that determines chatotic behavior of the x2 series.
- **b12** (*float*) – Influence of x1 on x2.
- **b21** (*float*) – Influence of x2 on x1.
- **ts_length** (*int*) – Length of the calculated time series.
- **random_start** (*bool*) – Random initialization of starting conditions.

Returns

- **x1** (*1d array*) – Array of length (ts_length,) that stores the values of the x series.
- **x2** (*1d array*) – Array of length (ts_length,) that stores the values of the y series.

skccm.data.**driven_rand_logistic**(*rx2*, *b12*, *ts_length*, *random_start=False*)

Logistic map with random forcing. x1 is the random array and x2 is the logistic map.

Parameters

- **rx2** (*float*) – Parameter that determines chatotic behavior of the x2 series.
- **b12** (*float*) – Influence of x1 on x2.
- **ts_length** (*int*) – Length of the calculated time series.
- **random_start** (*Boolean*) – Random initialization of starting conditions.

Returns

- **x1** (*array*) – Array of length (ts_length,)

- **x2** (*array*) – Array of length (ts_length,)

skccm.data.**driving_sin**(*rx2*, *b12*, *ts_length*, *random_start=False*)
Sine wave driving a logistic map.

Parameters

- **rx2** (*float*) – Parameter that determines chatotic behavior of the x2 series.
- **b12** (*float*) – Influence of x1 on x2.
- **ts_length** (*int*) – Length of the calculated time series.
- **random_start** (*Boolean*) – Random initialization of starting conditions.

Returns

- **x1** (*array*) – Array of length (ts_length,) that stores the values of the x series.
- **x2** (*array*) – Array of length (ts_length,) that stores the values of the y series.

skccm.data.**lagged_coupled_logistic**(*rx1*, *rx2*, *b12*, *b21*, *ts_length*, *random_start=False*)
Coupled logistic map. x1 is driven by random lags of x2.

Parameters

- **rx1** (*float*) – Parameter that determines chaotic behavior of the x1 series.
- **rx2** (*float*) – Parameter that determines chatotic behavior of the x2 series.
- **b12** (*float*) – Influence of x1 on x2.
- **b21** (*float*) – Influence of x2 on x1.
- **ts_length** (*int*) – Length of the calculated time series.
- **random_start** (*Boolean*) – Random initialization of starting conditions.

Returns

- **x1** (*array*) – Array of length (ts_length,) that stores the values of the x series.
- **x2** (*array*) – Array of length (ts_length,) that stores the values of the y series.

skccm.data.**lorenz**(*sz=10000*, *noise=0*, *max_t=100.0*)
Integrates the lorenz equation.

Parameters

- **sz** (*int*) – Length of the time series to be integrated.
- **noise** (*float*) – Amplitude of noise to be added to the lorenz equation.
- **max_t** (*float*) – Length of time to solve the lorenz equation over.

Returns **X** – Solutions to the Lorenz equations. Columns are X,Y,Z.

Return type 2D array

CHAPTER 4

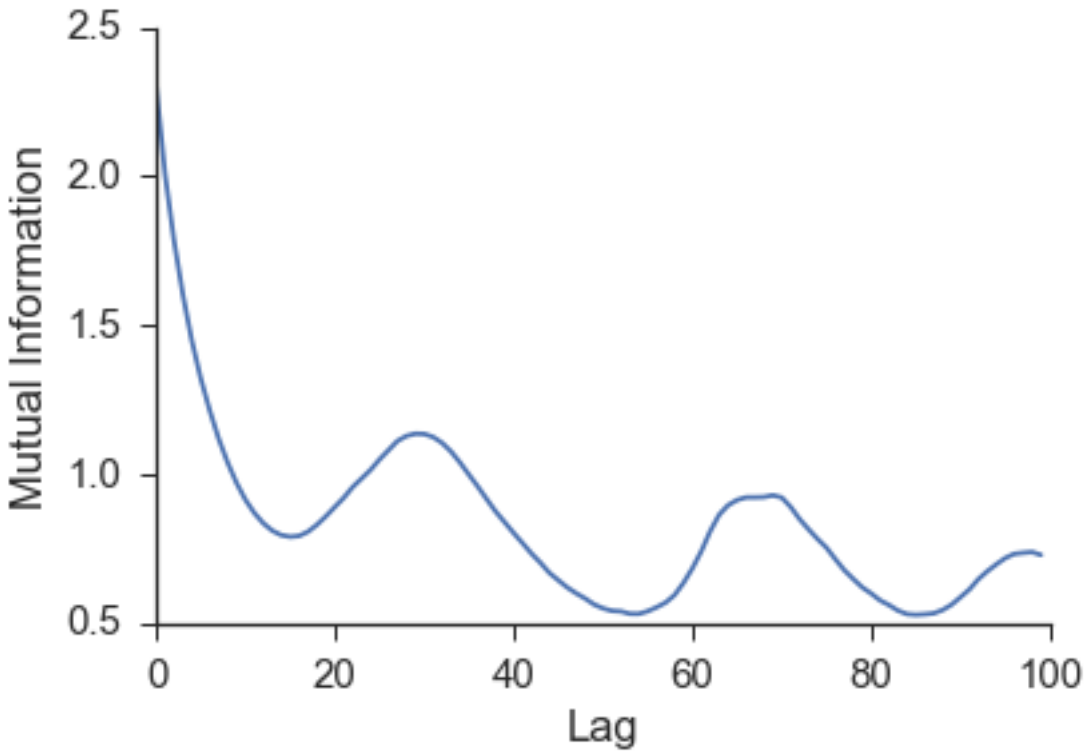
Embed

Before performing convergent cross mapping, both time series must be appropriately embed. As a quick recap, the lag is picked as the first minimum in the mutual information and the embedding dimension is picked using a false near neighbors test. In practice, however, it is acceptable to use the embedding that gives the highest forecast skill.

4.1 Lag Value

In attempting to use the time series to reconstruct the state space behavior of a complete system, a lag is needed to form the embedding vector. This lag is most commonly found from the first minimum in the mutual information between the time series and a shifted version of itself. The first minimum in the mutual information can be thought of as jumping far enough away in the time series that new information is gained. A more inituitive but less commonly used procedure to finding the lag is using the first minimum in the autocorrelation. The mutual information calculation can be done using the embed class provided by skccm.

```
from skccm import Embed
lag = 1
embed = 2
e1 = Embed(x1)
e2 = Embed(x2)
X1 = e1.embed_vectors_1d(lag,embed)
X2 = e2.embed_vectors_1d(lag,embed)
```



The figure above shows the mutual information for the x values of the lorenz time series. We can see a minimum around 16. If we were interested in reconstructing the state space behavior we would use a lag of 16.

4.2 Embedding Dimension

Traditionally, the embedding dimension is chosen using a false near neighbor test. This checks to see when the reconstructed attractor is fully "unfolded". This functionality is not in skccm currently, but will be added in the future. In practice, the embedding dimension that gives the highest forecast skill is chosen. The false near neighbor test can be noisy for real world systems.

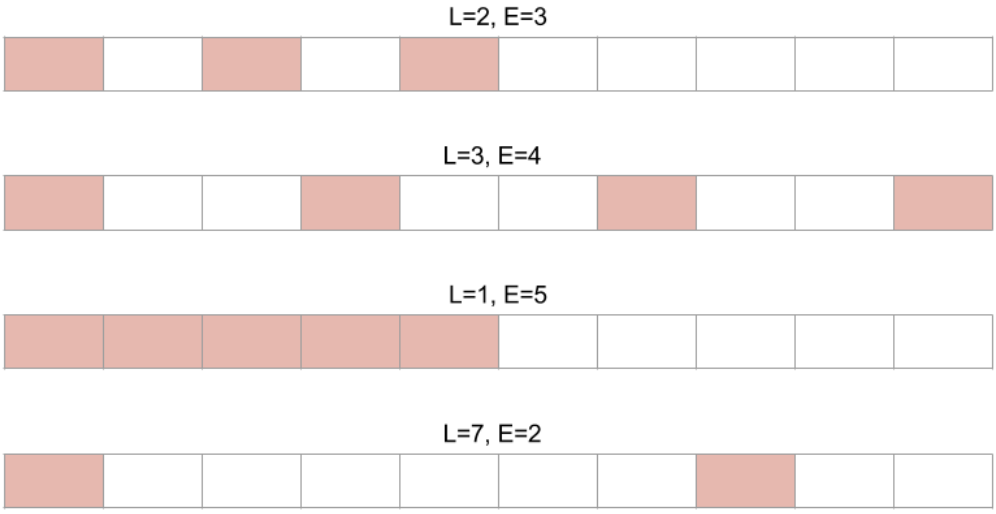
4.3 Examples

An example of a 1D embedding is shown in the gif below. This is the same thing as rebuilding the attractor. It shows a lag of 2 and an embedding dimension of 3. Setting the problem up this way allows us to use powerful near neighbor libraries such as the one implemented in scikit-learn.

Using this package, this would be represented as:

```
from skccm import Embed
lag = 1
embed = 2
e1 = Embed(x1)
e2 = Embed(x2)
X1 = e1.embed_vectors_1d(lag,embed)
X2 = e2.embed_vectors_1d(lag,embed)
```

More examples of 1d embeddings are shown below. L is the lag and E is the embedding dimension.



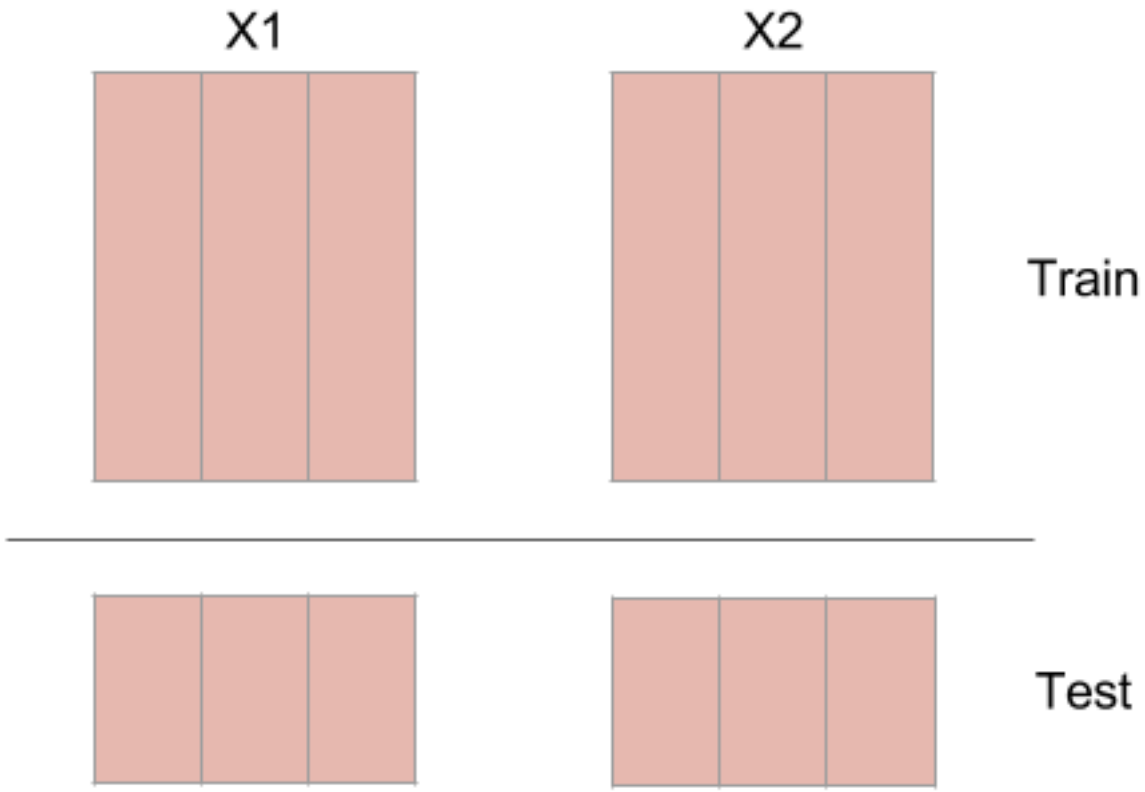
CHAPTER 5

Predict

At the heart of convergent cross mapping is the k-nearest neighbors algorithm. In fact, the package uses scikit-learn's nearest neighbor implementation for efficient calculation of distances and to retrieve the indices of the nearest neighbors. It is a good idea to understand the k-nearest neighbor algorithm before interpreting what this package implements.

5.1 Training and Testing

Before making predictions, split the embedded time series into a testing set and a training set. This protects against highly autocorrelated time series. For example, random walk time series can seem like they are coupled if they are not split into a training set and testing set.



The code for this is done with a function in `skccm.utilities`.

```
from skccm.utilities import train_test_split

#split the embedded time series
x1tr, x1te, x2tr, x2te = train_test_split(X1,X2, percent=.75)
```

The code above uses the first 75% as the training set (to rebuild the shadow manifolds) and uses the last 25% as the testing set.

5.2 Distance Calculation

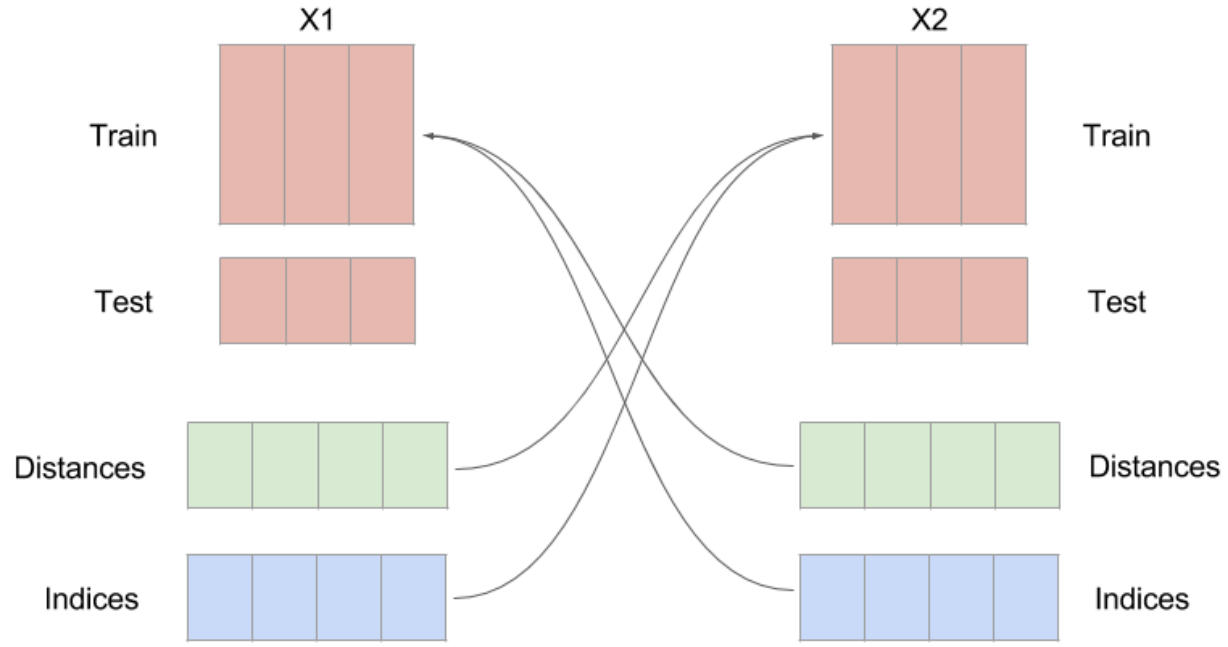
At this point, you will have these four embedded time series:

1. X1tr
2. X1te
3. X2tr
4. X2te

The distance is calculated from every sample in `X1te` to every sample in `X1tr`. The same is then done for `X2tr` and `X2te`. The distances are then sorted and the closest k indices are kept to make a prediction in the next step. k is the embedding dimension plus 1. So if your embedding dimension was three, then the amount of near neighbors used to make a prediction will be four.

5.3 Weighted Average

The next step is to use the near neighbor indices and weights to make a prediction about the other time series. The indices that were found by calculating the distance from every sample in X1te to every sample in X1tr, are used on X2tr to make a prediction about X2te. This seems a little counterintuitive, but it is expected that if one time series influences the other, the system being forced should be in a similar state when the system doing the forcing is in a certain configuration.



Notice the distances and indices have the same number of samples as the testing set, but an extra dimension. This is because you need $K + 1$ near neighbors in order to surround a point.

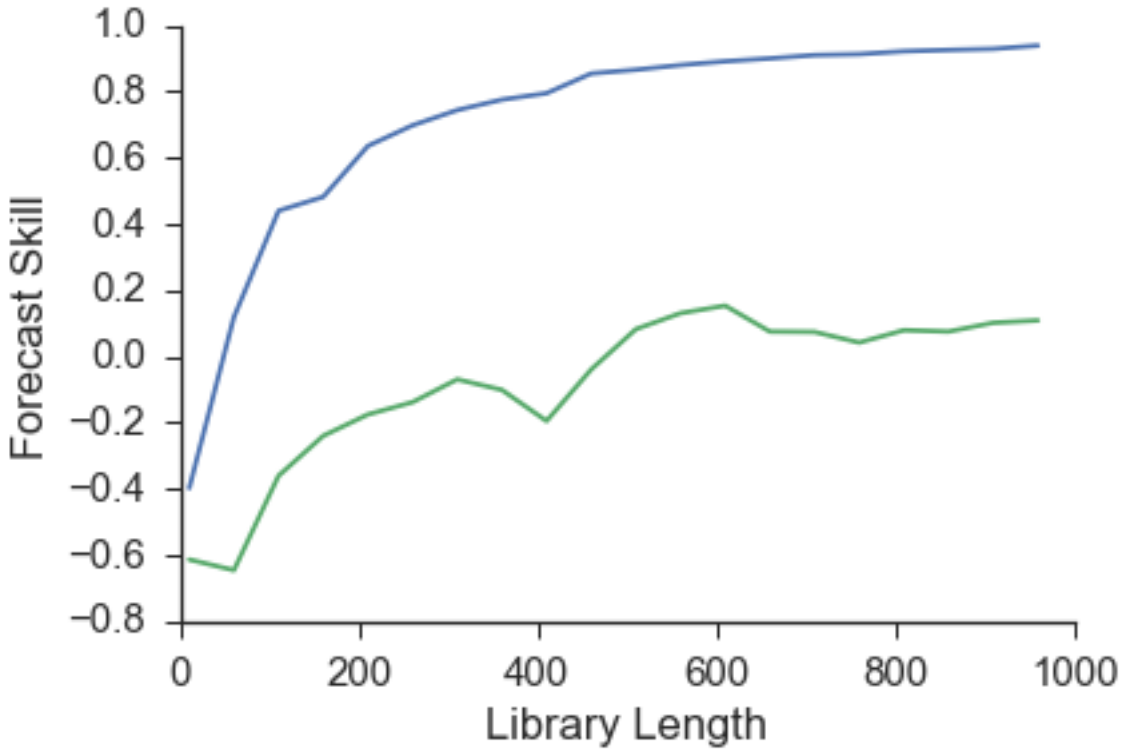
5.4 Library Length

A prediction is made for multiple library lengths. As the library length is increased, the prediction skill should converge. By increasing the library length, the density of the rebuilt attractor is increasing. As that attractor becomes more and more populated, better predictions should be able to be made.

CHAPTER 6

Score

The way the predictions are evaluated in the paper is by using the coefficient of determination between the predictions and the actual value. This is done for all the predictions at multiple library lengths. If the predictions for X_1 are better than X_2 than it is said that X_1 influences X_2.



In the plot above, the blue line has a higher forecast skill than the green line. This indicates that the blue system is forcing the green system. Another important feature is that they converge as the library length is increased. This indicates that the shadow manifold has been appropriately populated and novel predictions can be made.

CHAPTER 7

Module Reference

class skccm.skccm.**CCM**(*weights='exp'*, *verbose=False*)

Convergent cross mapping for two embedded time series.

Parameters

- **weights** (*str*) – Weighting scheme for predictions. Options:
  - 'exp' : exponential weighting
- **score** (*str*) – How to score the predictions. Options:
  - 'score'
  - 'corrcoef'
- **verbose** (*bool*) – Prints out calculation status.

fit(*X1_train*, *X2_train*)

Fit the training data for ccm. Can be thought of as reconstructing the shadow manifolds of each time series.

Amount of near neighbors is set to be embedding dimension plus one. Creates seperate near neighbor regressors for X1 and X2 independently.

Parameters

- **X1_train** (*2d array*) – Embed time series of shape (num_samps,embed_dim).
- **X2_train** (*2d array*) – Embed time series of shape (num_samps,embed_dim).

predict(*X1_test*, *X2_test*, *lib_lengths*)

Make a prediction.

Parameters

- **X1_test** (*2d array*) – Embed time series of shape (num_samps,embed_dim).
- **X2_test** (*2d array*) – Embed time series of shape (num_samps,embed_dim).
- **lib_lengths** (*1d array of ints*) – Library lengths to test.

Returns

- **X1_pred** (*list of 2d arrays*) – Predictions for each library length.
- **X2_pred** (*list of 2d arrays*) – Predictions for each library length.

score(*score_metric='corrcoef'*)
Evalulate the predictions.

Parameters **how** (*string*) – How to score the predictions. Options: - 'score' - 'corrcoef'

Returns

- **score_1** (*2d array*) – Scores for the first time series using the weights from the second time series.
- **score_2** (*2d array*) – Scores for the second time series using the weights from the first time series.

class skccm.skccm.**Embed**(*X*)
Embed a time series.

Parameters **X** (*1D array*) – Time series to be embed.

df_mutual_information(*max_lag*)
Calculates the mutual information along each column of a dataframe.

Ensure that the time series is continuous in time and sampled regularly. You can resample it hourly, daily, minutely etc. if needed.

Parameters **max_lag** (*int*) – maximum amount to shift the time series

Returns **mi** – columns are the columns of the original dataframe with rows being the mutual information. shape(max_lag,num_cols)

Return type dataframe

embed_vectors_1d(*lag*, *embed*)
Embeds vectors from a one dimensional time series in m-dimensional space.

Parameters

- **X** (*1d array*) – Training or testing set.
- **lag** (*int*) – Lag value as calculated from the first minimum of the mutual info.
- **embed** (*int*) – Embedding dimension. How many lag values to take.
- **predict** (*int*) – Distance to forecast (see example).

Returns **features** – Contains all of the embedded vectors. Shape (num_vectors,embed).

Return type 2d array

Example

```
>>> X = [0,1,2,3,4,5,6,7,8,9,10]
em = 3
lag = 2
predict=3
```

```
>>> embed_vectors_1d
features = [[0,2,4], [1,3,5], [2,4,6], [3,5,7]]
```

mutual_information(*max_lag*)

Calculates the mutual information between the an unshifted time series and a shifted time series.

Utilizes scikit-learn's implementation of the mutual information found in sklearn.metrics.

Parameters **max_lag** (*integer*) – Maximum amount to shift the time series.

Returns **m_score** – Mutual information at between the unshifted time series and the shifted time series,

Return type 1-D array

CHAPTER 8

Acknowledgements

I would like to thank the Gordon and Betty Moore Foundation for funding some of this work. I would also like to thank Dylan McNamara for continous feedback and for introducing me to emperical dynamic modeling. I would also like to thank Kenneth Ells and the rest of the CASL group for proofreading and testing this package.

Python Module Index

S

Index