skedm Documentation

Release 0.1

Nick Cortale

Aug 15, 2017

Contents:

Scikit Empirical Dynamic Modeling

Scikit Empirical Dynamic Modeling (skedm) can be used as a way to forecast time series, spatio-temporal 2D or 3D arrays, and even discrete spatial arrangements. More importantly, skedm can provide insight into the underlying dynamics of a system, specifically whether a system is nonlinear and deterministic or whether it is dominated by noise.

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

1. Time Series and Dynamic Manifolds
2. Reconstructed Shadow Manifold

For a more complete background, I suggest checking out Nonlinear Analysis by Kantz as well as Practical implementation of nonlinear time series methods: The TISEAN package.

This software is useful both for forecasting and exploring underlying dynamical processes in a broad range of systems. The target audience is also wide-ranging as the software can be used to explore any dynamical system. Previous work using similar analyses has explored ecological systems, physical systems, and physiological applications.

CHAPTER 1

Install

pip

```
pip install skedm
```

Conda (Recommended)

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

```
name: skedm_env
dependencies:
  - python=3
  - numpy
  - numba
  - scikit-learn
  - scipy
  - pip:
    - skedm
```

Then you can simply create the environment with:

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

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

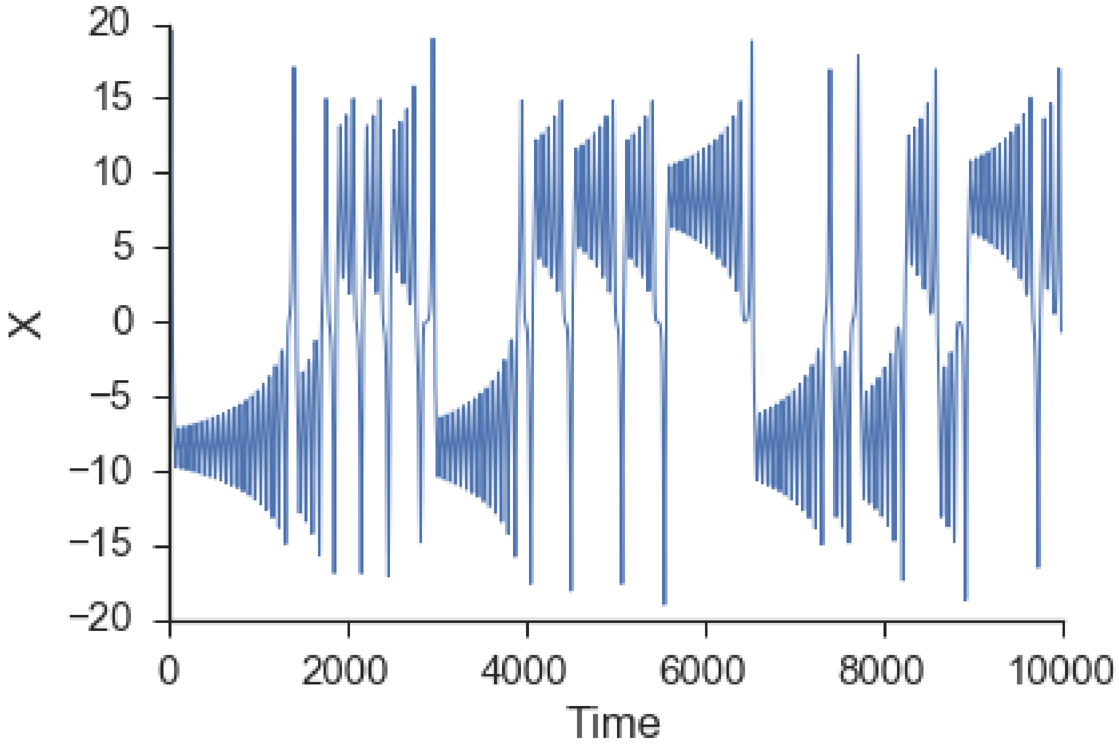
To illustrate the utility of this package we will work with the Lorenz system. The Lorenz system takes the form of :

$$\frac{dx}{dt} = \sigma(y - x)$$
$$\frac{dy}{dt} = x(\rho - z) - y$$
$$\frac{dz}{dt} = xy - \beta z$$

After numerically solving this system of equations, we are going to make forecasts of the x time series. Note that this series, while completely deterministic (there is no randomness associated with any of the above equations), is a classic chaotic system. Chaotic systems are notoriously difficult to forecast (e.g. weather) due to their sensitive dependance on initial conditions and corresponding positive Lyapunov exponents.

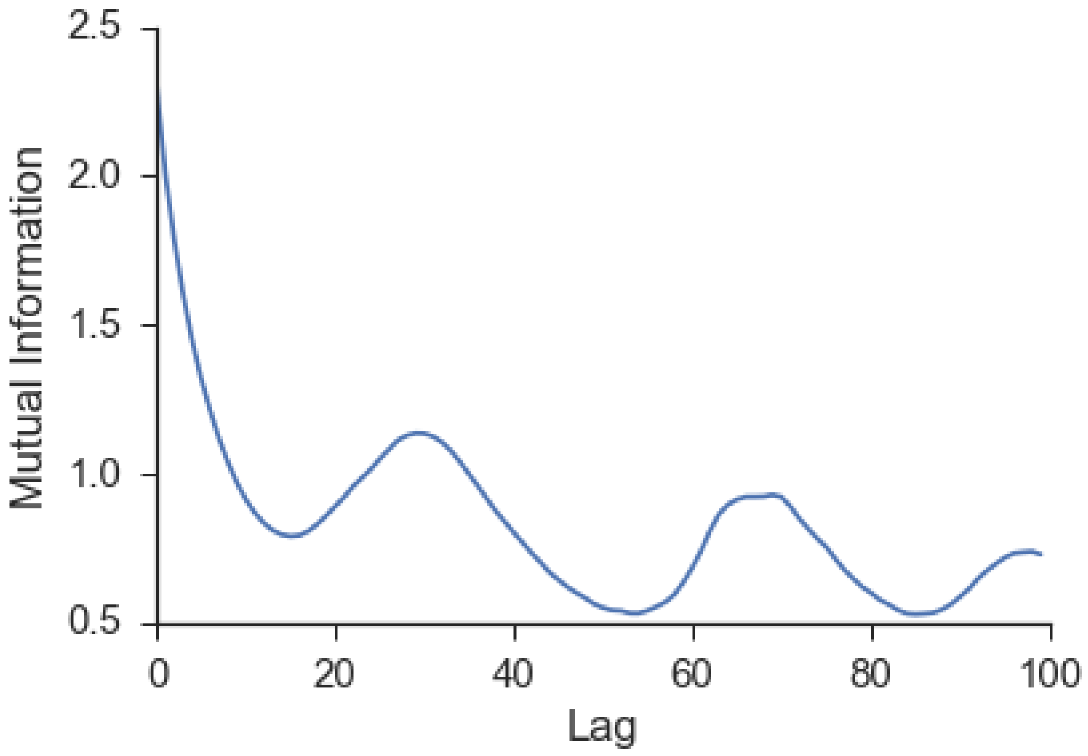
There is a function in `skedm.data` that numerically solves the lorenz system using scipy's built in integrator with a step size of 0.1 seconds. For example:

```
import skedm.data as data

X = data.lorenz(sz=10000)[:,0] #only going to use the x values from the lorenz system
```



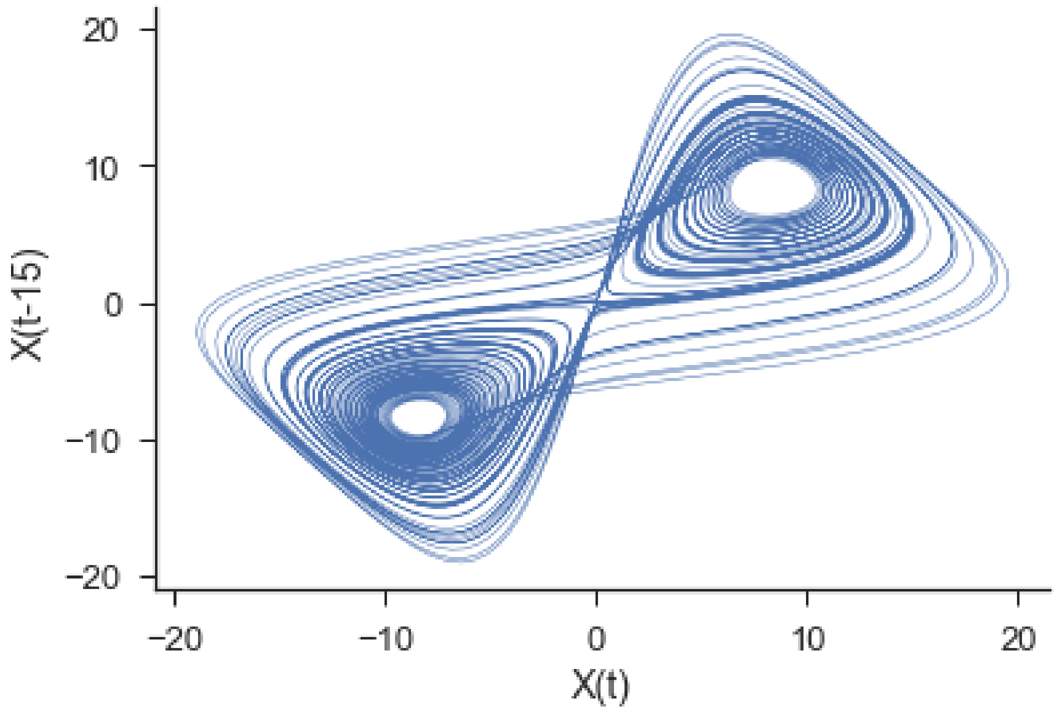
In attempting to use the x time series to reconstruct the state space behavior of the complete lorenz system, a lag is needed to form the embedding vector. This lag is most commonly found from the first minimum in the mutual information between the time series and a shifted version of itself. The first 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 skedm.

```
import skedm as edm

E = edm.Embed(X) #initiate the class

max_lag = 100
mi = E.mutual_information(max_lag)
```



The first minimum of the mutual information is at $n = 15$. This is the lag that will be used to rebuild a shadow manifold. This is done by the `embed_vectors_1d` method. A longer discussion about embedding dimension (how the value for `embed` is chosen) is found in the *Embed* section.

```
lag = 15
embed = 3
predict = 30 #predicting out to double to lag
X,y = E.embed_vectors_1d(lag,embed,predict)
```



The plot above is showing only `X[:,0]` and `X[:,1]`. The full three dimensional embedding preserves the geometric features of the original lorenz attractor.

Now that we have embed the time series, we can use trajectories in the state space near a point in question to generate forecasts of system behavior and see if the system contains the hallmarks of nonlinearity, namely high forecast skill when using local neighbor trajectories in the reconstructed space. First we split the data into a training set and testing set. Additionally, we will initiate the Regression class.

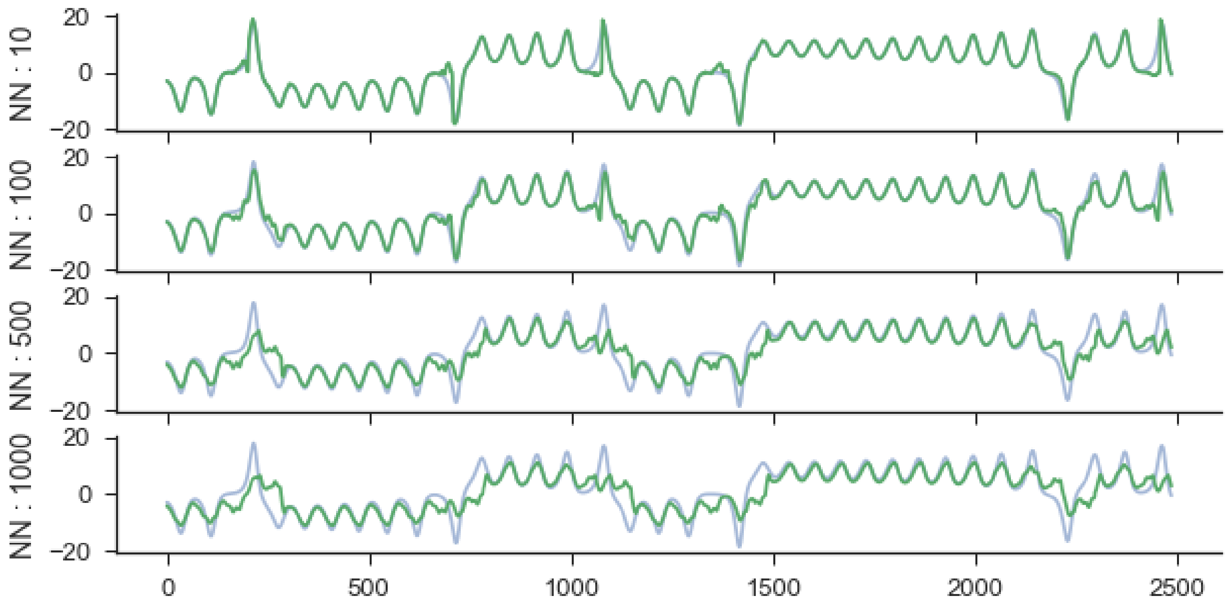
```
#split it into training and testing sets
train_len = int(.75*len(X))
Xtrain = X[0:train_len]
ytrain = y[0:train_len]
Xtest = X[train_len:]
ytest = y[train_len:]

weights = 'distance' #use a distance weighting for the near neighbors
M = edm.Regression(weights) # initiate the nonlinear forecasting class
```

Next, we need to fit the training data (rebuild the shadow manifold) and make predictions for the test set (22 seconds on a macbook air).

```
M.fit(Xtrain, ytrain) #fit the data (rebuilding the attractor)

nn_list = [10, 100, 500, 1000]
ypred = M.predict(Xtest,nn_list)
```

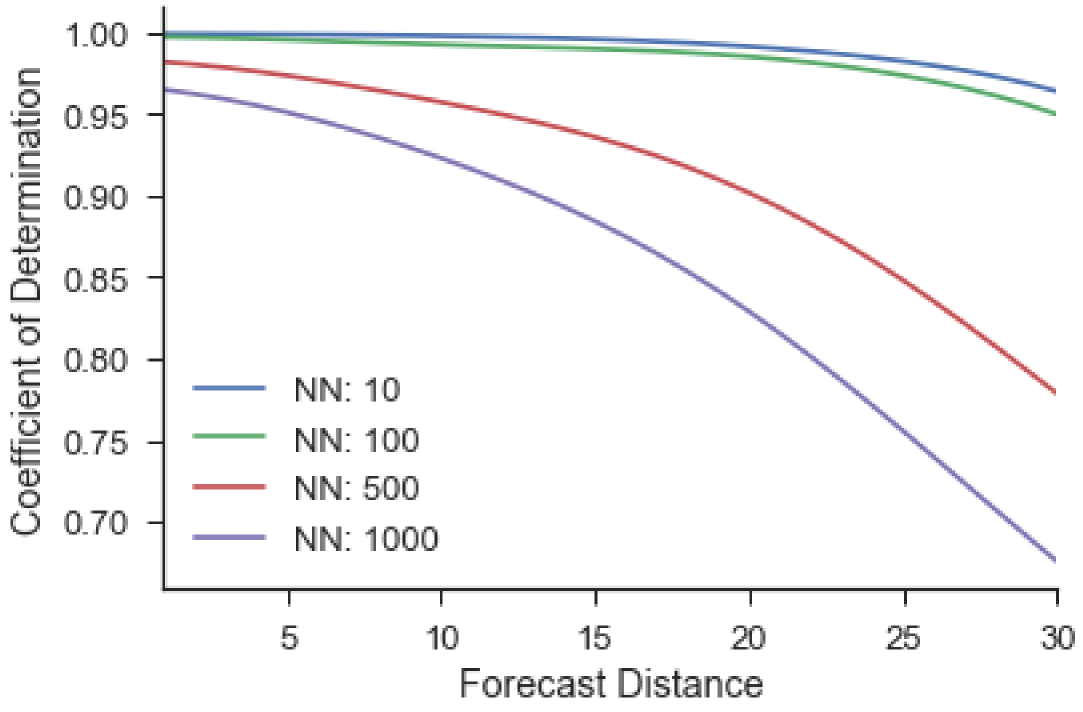
`ypred` is a list of 2d arrays that is the same length as nn_list. Each 2d array is of shape `(len(Xtest), predict)`. For example, the second item in `ypred` is the predictions made by taking a weighted average of the closest 100 neighbor's trajectories. We can view the predictions using 10, 100, 500, and 1000 neighbors at a forecast distance of 30 by doing the following:

```
fig,axes = plt.subplots(4,figsize=(10,5),sharex=True,sharey=True)
ax = axes.ravel()

ax[0].plot(ytest[:,29],alpha=.5)
ax[0].plot(ypred[0][:,29])
ax[0].set_ylabel('NN : ' + str(nn_list[0]))

ax[1].plot(ytest[:,29],alpha=.5)
ax[1].plot(ypred[1][:,29])
ax[1].set_ylabel('NN : ' + str(nn_list[1]))

ax[2].plot(ytest[:,29],alpha=.5)
ax[2].plot(ypred[2][:,29])
ax[2].set_ylabel('NN : ' + str(nn_list[2]))


ax[3].plot(ytest[:,29],alpha=.5)
ax[3].plot(ypred[3][:,29])
ax[3].set_ylabel('NN : ' + str(nn_list[3]))

sns.despine()
```



The next step is to evaluate the predictions with the `score` method. The score method defaults to using the coefficient of determination between the actual values and predicted values.

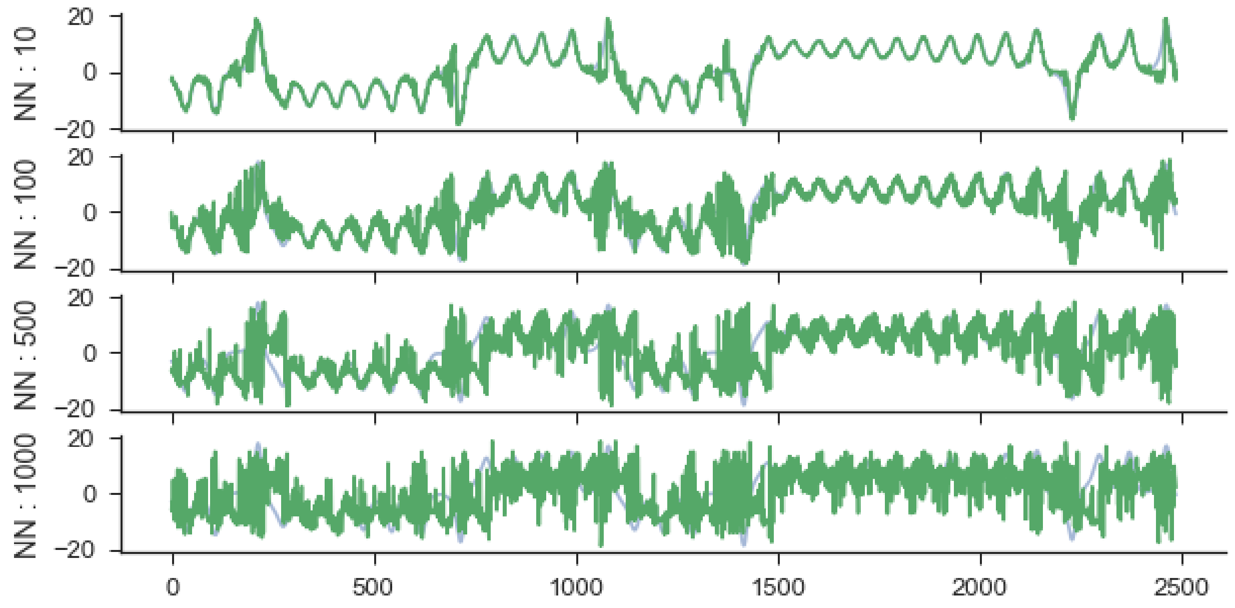
```
scores = M.score(ytest) #score

fig,ax = plt.subplots()

for i in range(4):
    label = 'NN: ' + str(nn_list[i])
    ax.plot(range(1,31),scores[i],label=label)

plt.legend(loc='lower left')
ax.set_ylabel('Coefficient of Determination')
ax.set_xlabel('Forecast Distance')
ax.set_xlim(1,30)
sns.despine()
```



`scores` has shape `(len(nn_list),predictions)`. So this example will have a shape that is `(4, 30)`. For example, the first spot in the score array will be the coefficient of determination between the actual values one time step ahead and the predicted values one time step ahead using 10 near neighbors. As expected, the forecast accuracy decreases as more near neighbor trajectories are averaged together to make a prediction and as we increase the forecast distance.

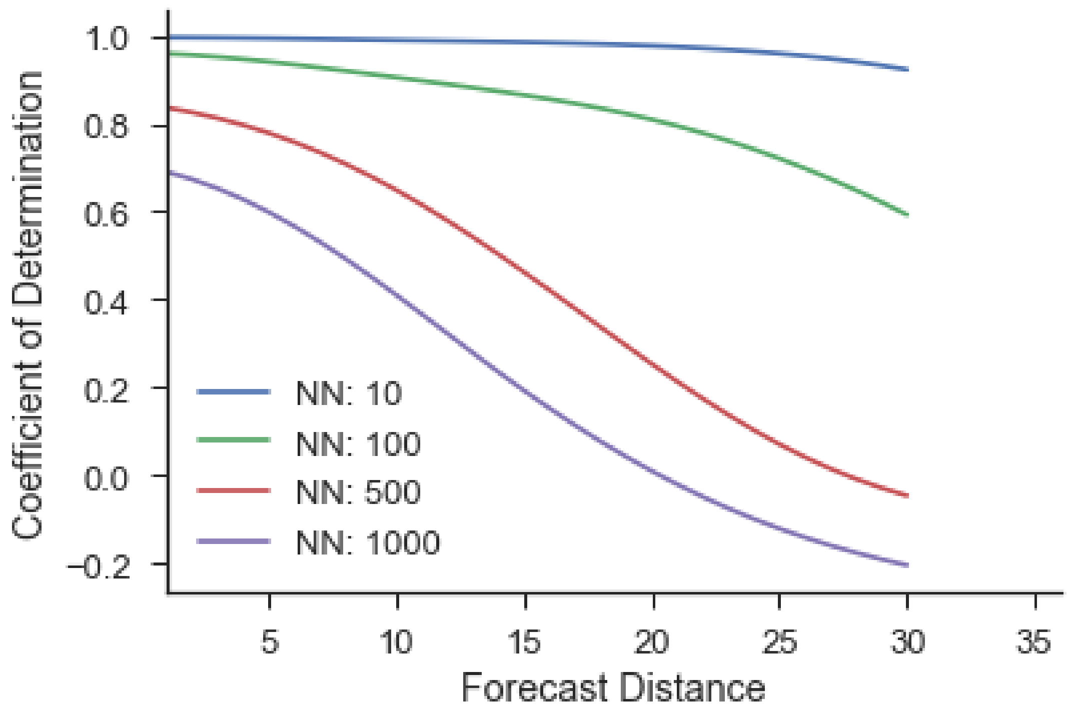
Additionally, instead of averaging near neighbor trajectories, it is possible to look at the forecast of each neighbor individually. This is done by simply calling the `predict_individual` method as below.

```
ypred = M.predict_individual(Xtest,nn_list)
```



Then again, we can calculate the score and visualize it as:

```
score = M.score(ytest)
fig,ax = plt.subplots()

for i in range(4):
    label = 'NN: ' + str(nn_list[i])
    ax.plot(range(1,31),score[i],label=label)

plt.legend(loc='lower left')
sns.despine()
ax.set_ylabel('Coefficient of Determination')
ax.set_xlabel('Forecast Distance')
ax.set_xlim(1,36);
```



As we can see, by not averaging the near neighbors, the forecast skill decreases and the actual forecast made becomes quite noisy. This is because we are now using single trajectories that are not nearby in the reconstructed space to make predictions. This should intuitively do worse than picking nearby regions.

CHAPTER 3

Generate Data

skedm comes with both 1D and 2D test data. These built in functions allow you to quickly explore the behavior of different systems. Check the skedm_example_data notebook for visual examples.

1 Dimensional

`skedm.data.logistic_map`(*sz=256*, *A=3.99*, *seed=36*, *noise=0*)

Solutions to the logistic map for a given amount of time steps.

$$X_{t+1} = AX_t(1 - X_t) + \alpha\eta$$

where the value of the parameter A determines if the system is chaotic, η is uncorrelated white noise and α is the amplitude of the noise.

Parameters

- **sz** (*int*) – Length of the time series.
- **A** (*float*) – Parameter for the logistic map. Values beyond 3.56995 exhibit chaotic behaviour.
- **seed** (*int*) – Sets the random seed for numpy's random number generator.
- **noise** (*float*) – Amplitude of the noise.

Returns **X** – Logistic map of size (sz).

Return type 1D array

`skedm.data.noisy_periodic`(*sz=256*, *freq=52*, *noise=0.5*, *seed=36*)

A simple periodic equation with a a specified amplitude of noise.

$$X(t) = sin(2\pi ft) + 0.5cos(2\pi ft) + \alpha\eta$$

Where f is the frequency η is uncorrelated white noise and α is the amplitude of the noise.

Parameters

- **sz** (*int*) – Length of the time series.
- **freq** (*int*) – Frequency of the periodic equation.
- **noise** (*float*) – Amplitude of the noise.
- **seed** (*int*) – Sets the random seed for numpy's random number generator.

Returns **X** – Periodic array of size (sz) with values between 0 and 1.

Return type 1D array

skedm.data.**noise_1d**(*sz=256*, *seed=36*)

White noise with values between 0 and 1. Uses numpy's random number generator.

Parameters

- **sz** (*int*) – Length of the time series.
- **seed** (*int*) – Sets the random seed for numpy's random number generator.

Returns **X** – Random array of size (sz) with values between 0 and 1.

Return type 1D array

skedm.data.**lorenz**(*sz=10000*, *max_t=100.0*, *noise=0*, *parameters=(10*, *2.6666666666666665*, *28.0)*)

Integrates the lorenz equations. Which are defined as:

$$\frac{dx}{dt} = \sigma(y - x)$$
$$\frac{dy}{dt} = x(\rho - z) - y$$
$$\frac{dz}{dt} = xy - \beta z$$

Where $\sigma = 10$, $\beta = 8/3$, and $\rho = 28$ lead to chaotic behavior.

Parameters

- **sz** (*int*) – Length of the time series to be integrated.
- **max_t** (*float*) – Length of time to solve the lorenz equation over,
- **noise** (*float*) – Amplitude of noise to be added to the lorenz equation.
- **parameters** (*tuple*) – Sigma, beta, and rho parameters for the lorenz equations.

Returns **X** – X solutions in the first column, Y in the second, and Z in the third.

Return type 2D array

2 Dimensional

skedm.data.**chaos_2d**(*sz=128*, *A=3.99*, *eps=1.0*, *noise=None*, *seed=36*)

Logistic map diffused in space. It takes the following form:

$$x_{t+1} = Ax_t(1 - x_t) \equiv f(x_t)$$

$$x_{t+1,s} = \frac{1}{1+2\epsilon}[f(x_{t,s}) + \ \epsilon f(x_{t,s\pm1})] + \alpha\eta$$

Where A is the parameter that controls chaos, η is uncorrelated white noise, α is the amplitude of the noise and ϵ is the strength of the spatial coupling.

Parameters

- **sz** (*int*) – Row and column size of the spatio-temporal series to be generated.
- **A** (*float*) – Parameter for the logistic map. Values beyond 3.56995 exhibit chaotic behaviour.
- **eps** (*float*) – Spatial coupling strength.
- **seed** (*int*) – Sets the random seed for numpy's random number generator.
- **noise** (*float*) – Amplitude of the noise.

Returns **X** – Spatio-temporal logistic map of size (sz,sz).

Return type 2D array

`skedm.data.`**`periodic_2d`**`(`*sz=128*, *freq=36*, *noise=0.5*, *seed=36*`)`

A simple 2D periodic equation with a specified amplitude of noise. This is a sine wave down the rows, added to a cosine wave across the columns.

$$X(r,c) = sin(2\pi fr) + 0.5cos(2\pi fc) + \alpha\eta$$

Where r and c are the row and column values, f is the spatial frequency, η is uncorrelated white noise and α is the amplitude of the noise.

Parameters

- **sz** (*int*) – Length of the time series.
- **freq** (*int*) – Frequency of the periodic equation.
- **noise** (*float*) – Amplitude of the noise.
- **seed** (*int*) – Sets the random seed for numpy's random number generator.

Returns **X** – 2D periodic equaiton of size (sz,sz). Values between 0 and 1.

Return type 2D array

`skedm.data.`**`brown_noise`**`(`*sz=128*, *num_walks=500*, *walk_sz=100000*, *spread=1000*, *seed=3*`)`

Creates brown noise by summing many random walks.

Subsamples to generate sizes: 128, 256, or 512. 512 is the full size.

Parameters

- **sz** (*int*) – Row and column size of the array to be returned.
- **num_walks** (*int*) – Number of random walks.
- **walk_sz** (*int*) – Length of the random walk to take.
- **spread** (*int*) – Normal distribution of walks. Sizes randn*spread.
- **seed** (*int*) – Sets the random seed for numpy's random number generator.

Returns **X** – 2D brown noise array size (sz,sz).

Return type 2D array

`skedm.data.`**`periodic_brown`**`(`*sz=128*, *freq=36*, *seed=36*`)`

A periodic equation with a specified amplitude of brown noise. Calls the function brown_noise.

$$X(r,c) = sin(2\pi fr + \eta)$$

Where r and c are the row and column values, f is the spatial frequency, and η is the brown noise.

Parameters

- **sz** (*int*) – Length of the spatiotemporal series.
- **freq** (*int*) – Frequency of the periodic equation.
- **seed** (*int*) – Sets the random seed for numpy's random number generator.

Returns **X** – Array containing the periodic equation with brown noise added.

Return type 2D array

`skedm.data.`**`noise_2d`**(*sz=128*, *seed=36*)

2D array of white noise values between 0 and 1. Uses numpy's random number generator.

Parameters

- **sz** (*int*) – row and column size of array.
- **seed** (*int*) – Sets the random seed for numpy's random number generator.

Returns **X** – Array of size (sz,sz) with values between 0 and 1.

Return type 2D array

`skedm.data.`**`overlapping_circles`**(*sz=256*, *rad=20.0*, *sigma=1*, *num_circles=1000*)

Randomly places circles that have been gaussian blurred. Overlapping circles are summed together. Uses scipy's gaussian filter.

Calls _gauss_circle_create to make the circles.

Parameters

- **sz** (*int*) – Row and column size of the space.
- **rad** (*float*) – Radius of the circles.
- **sigma** (*float*) – Constant that controlls the strength of the filter.
- **num_circles** (*int*) – Number of circles to place down randomly.

Returns **X** – Summed circles. Size (sz,sz).

Return type 2D array

`skedm.data.`**`concentric_circles`**(*rad_list*, *sz=256*, *num_circs=1000*)

Create circles inside larger circles. These circles cannot overlap. Calls the function _concentric_circle.

For example, if rad_list=[2,4,6], and sz=256; _concentric_circle will generate concentric circles in a 256x256 matrix with the innermost radius equal to 2 and the outermost radius equal to six. The values of the circles will be 1, 2, and 3 respectively.

Parameters

- **rad_list** (*list of ints*) – Radii of the cocentric circles circles. Must be increasing.
- **sz** (*int*) – Row and column size of the returned space.
- **num_circs** (*int*) – Number of circles to create.

Returns **blobs** – Array of integers with shape (sz,sz).

Return type 2D array

`skedm.data.`**`small_and_large_circles`**(*sz=256*, *rad1=5*, *rad2=8*, *num_circs=1000*)

Create larger circles with smaller circles spread randomly in space.

Same number of large circles and smaller circles. Calls _circle_create.

Parameters

- **`sz`** (*`int`*) – Row and column size of the space in which the circle is placed.
- **`rad1`** (*`int`*) – Radius of the smaller circle.
- **`rad2`** (*`int`*) – Radius of the larger circle.
- **`num_circs`** (*`int`*) – Number of large and small circles to attempt to create.

Returns **blobs** – Array of size (sz,sz) containing all the circles.

Return type 2D array

`skedm.data.`**`random_sized_circles`**(*rad_list*, *val_list*, *sz=512*, *num_circs=3000*)

Create random sized circles spread randomly in space and assign them to classes in val_list.

For example, rad_list=[1,2,3] and val_list=[4,5,6] will create circles with radius 1, 2, and 3 and values 4, 5, and 6 respectively.

Parameters

- **`sz`** (*`int`*) – Row and column size of the space in which the circle is placed.
- **`rad_list`** (*`list of ints`*) – List of radii.
- **`val_list`** (*`list ints`*) – List of values associated with the radii.
- **`num_circs`** (*`int`*) – Total number of circles to create.

Returns **blobs** – Array of integers corresponding to the values given in val_list. Areas without a circle will be zero.

Return type 2D array

`skedm.data.`**`voronoi_matrix`**(*sz=512*, *percent=0.01*, *num_classes=27*)

Create voronoi polygons.

Parameters

- **`sz`** (*`int`*) – Row and column size of the space.
- **`percent`** (*`float`*) – Percent of the space to place down centers of the voronoi polygons. Smaller percent makes the polygons larger.
- **`num_classes`** (*`int`*) – Number of classes to assign to each of the voronoi polygons.

Returns **X** – 2D array of size (sz,sz) containing the voronoi polygons.

Return type 2D array

`skedm.data.`**`chaos_3d`**(*sz=128*, *A=3.99*, *eps=1.0*, *steps=100*, *tstart=50*)

Logistic map diffused in space and taken through time. Chaos evolves in 3rd dimension.

$$x_{t+1} = Ax_t(1 - x_t) \equiv f(x_t)$$

$$x_{t+1,r,c} = \frac{1}{1+4\epsilon}[f(x_{t,r,c}) + \epsilon f(x_{t,r\pm 1,c}) + \epsilon f(x_{t,r,c\pm 1})]$$

Where A is the parameter that controls chaos and ϵ is the strength of the spatial coupling.

Parameters

- **`sz`** (*`int`*) – Row and column size of the spatio-temporal series to be generated.
- **`A`** (*`float`*) – Parameter for the logistic map. Values beyond 3.56995 exhibit chaotic behaviour.
- **`eps`** (*`float`*) – Amount of coupling/diffusion between adjecent cells.

- **`seed`** (*`int`*) – Sets the random seed for numpy's random number generator.
- **`tstart`** (*`int`*) – When to start collecting the data. This allows the chaos to be fully developed before collection.

Returns **X** – Spatiotemporal logistic map of size (sz, sz, steps).

Return type 2D array

CHAPTER 4

Embed

Embedding the time series, spatio-temporal series, or a spatial pattern is required before attempting to forecast or understand the dynamics of the system.

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.

1D Embedding

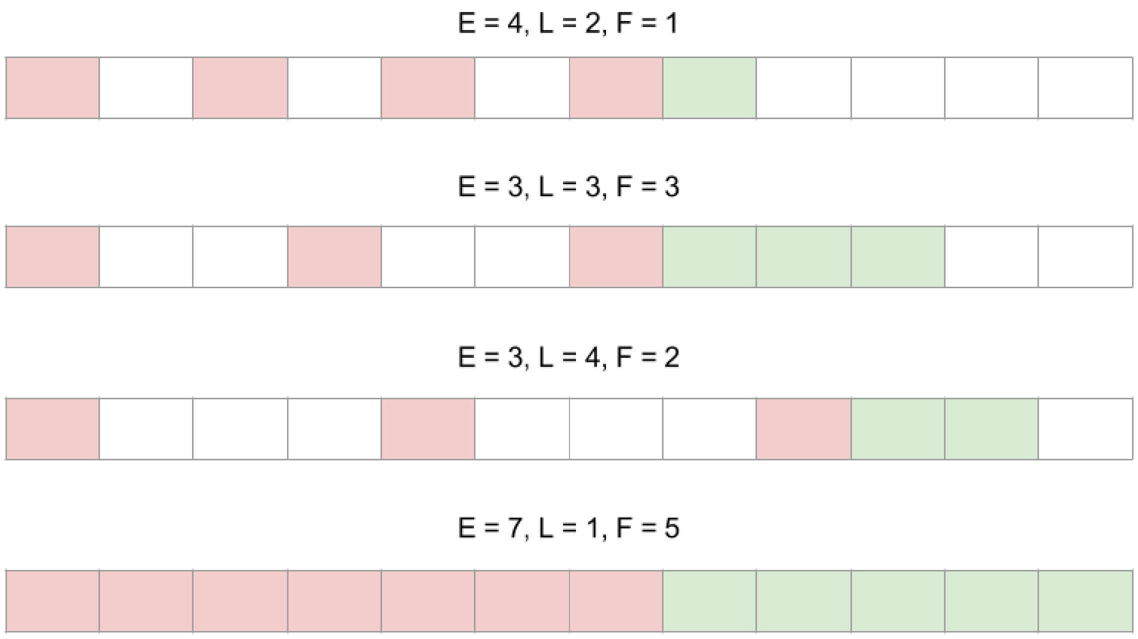
An example of a 1D embedding is shown in the gif below. It shows a lag of 2, an embedding dimension of 3 and a forecast distance of 2. Setting the problem up this way allows us to use powerful near neighbor libraries such as the one implemented in scikit-learn.

This is the same thing as rebuilding the attractor and seeing where the point traveled to next.

Using the skedm package, this would be represented as:

```
E = edm.Embed(X)

lag = 2
embed = 3
predict = 2
X, y = E.embed_vectors_1d(lag, emb, predict)
```

Here `X` is the red matrix (features) and `y` is the green matrix above (targets). More examples of 1d embeddings are shown below. E is the embedding dimension, L is the lag, and F is the prediction distance.



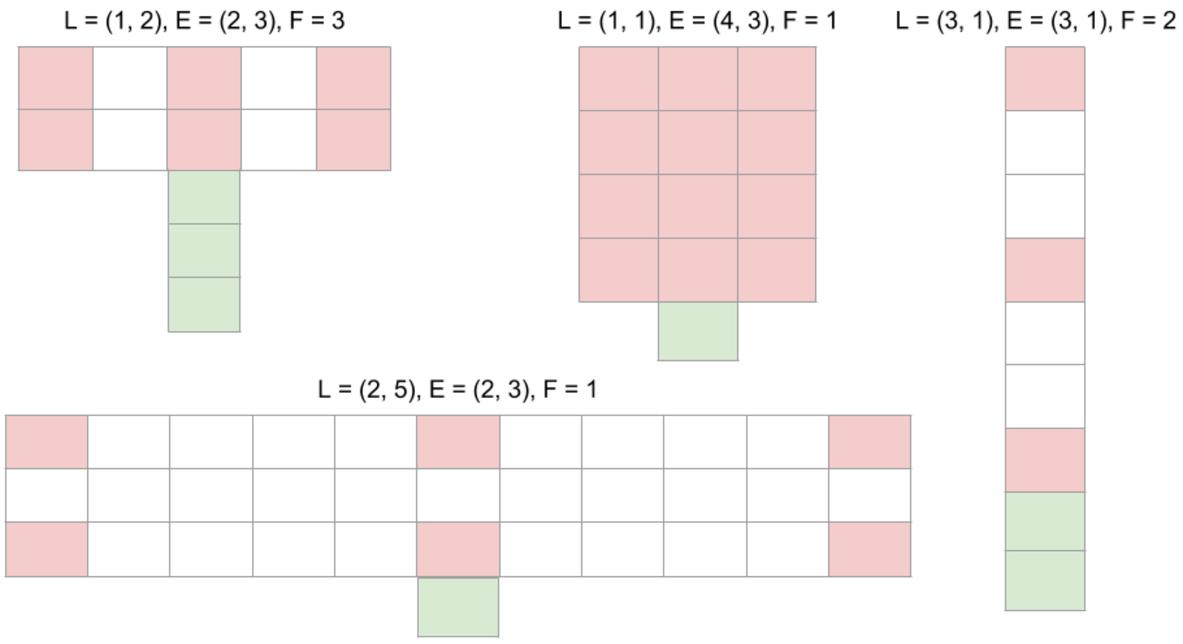
2D Embedding

An example of a 2D embedding is shown in the gif below where there is a lag of 2 in both the rows and columns, an embedding dimension of two down the rows, and an embedding dimension of three across the columns.

This would be implemented in code as:

```
E = edm.Embed(X)

lag = (2,2)
emb = (2,3)
predict = 2
X,y = E.embed_vectors_2d(lag, emb, predict)
```

Here `X` is the red matrix (features) and `y` is the green matrix above (targets). More examples of 2d embeddings are shown below. L is the lag, E is the embedding dimension, and F is the prediction distance.

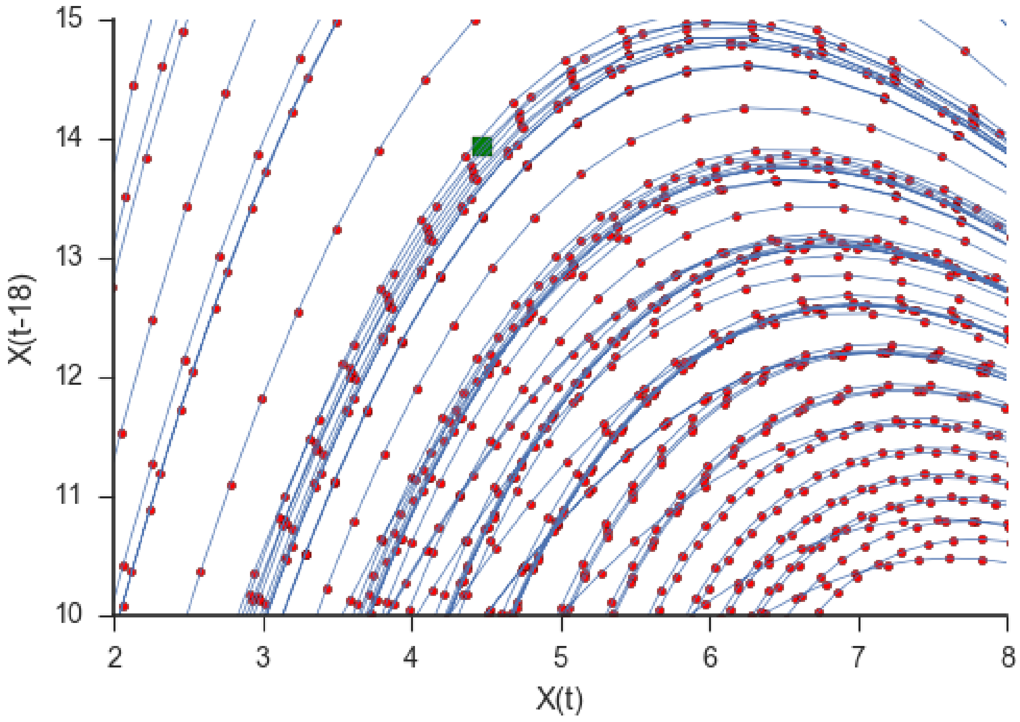

L = (1, 2), E = (2, 3), F = 3
L = (1, 1), E = (4, 3), F = 1
L = (3, 1), E = (3, 1), F = 2
L = (2, 5), E = (2, 3), F = 1

CHAPTER 5

Predict

At the heart of this software package for emperical dynamic modeling 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.

For the regression case, we will look at a zoomed in version of trajectories from the lorenz system projected in two-dimenisonal space. The red dots are the actual points that make up the trajectory depicted by the blue line and the green box is the point that we want to forecast. The trajectory is clockwise.

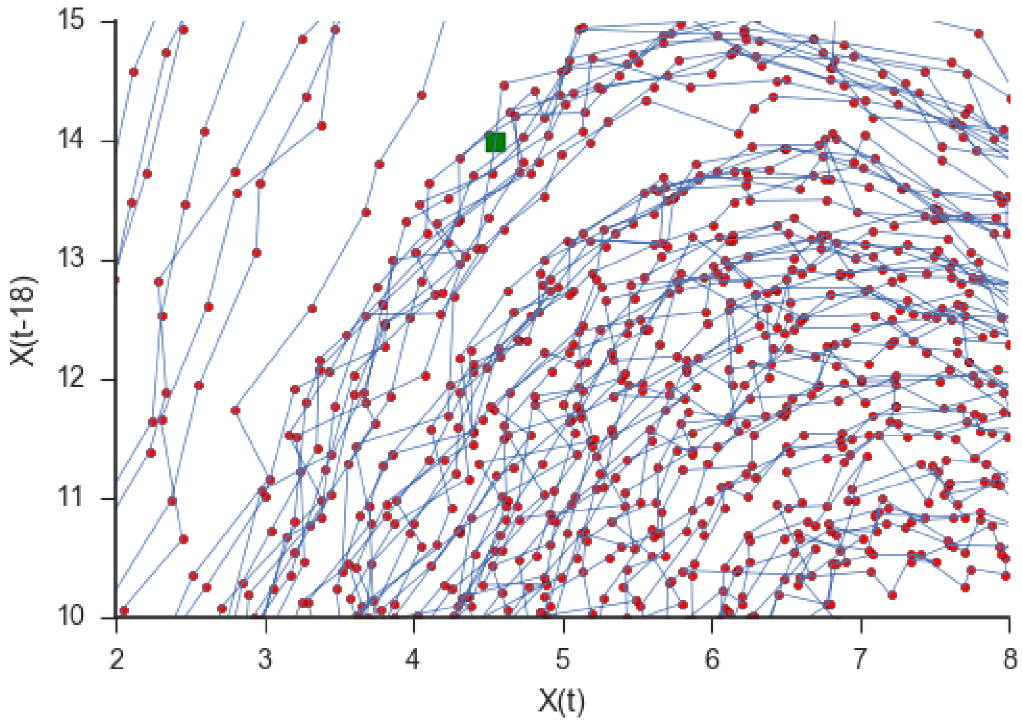


In this section of the Lorenz attractor, we can see that the red points closest to the green box all follow the same trajectory. If we wanted to forecast this green box, we could grab the closest red point and see where that ends up. We would then say that this is where the green box will end up.

Grabbing more points, however, might prove to be useful since our box lies between a couple of the points. It might be better to average the trajectories of, for example, the three nearest points to make a forecast.

For this case, grabbing more and more near neighbor trajectory points will be detrimental to the forecast as those far away points are providing information from regions of the space that are not useful for projecting the local dynamics of our test, green point.

It is illustrative to see the effects of adding noise to this system as shown in the plot below.



To the extent that noise becomes dominant relative to the local space dynamics, the gain from using only local trajectories goes away and predictability levels off or increases as one grabs more and more neighbor trajectories.

CHAPTER 6

Score

The next step is to examine the forecast skill. This is done by comparing the actual trajectories to the forecasted trajectories. We will see different patterns in the forecast skill depending on whether the system is dominated by deterministic or noisy dynamics.

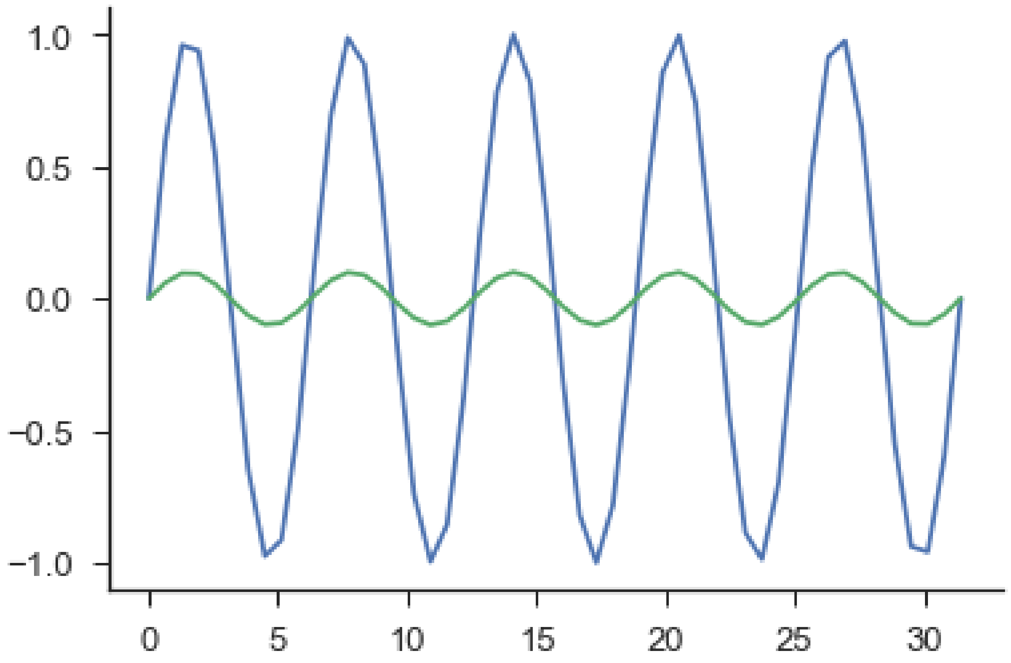
Regression

For regression, there are two ways to measure the accuracy of the results. The default `score` method is the coefficient of determination. This is the default for many of scikit-learn's scoring functions as well. Additionally, the correlation coefficient can also be used to score a continuous variable.

These can be called by:

```
score1 = M.score(ytest, how='score') # Coefficient of determination
score2 = M.score(ytest, how='coeff') # Correlation coefficient
```

Coefficient is useful when you are interested in making precise forecasts and correlation coefficient is useful when you are interested in accurate forecasts. For example, consider the following plot where blue are the actual values and green are the predicted values.



The correlation coefficient for these predictions would be equal to one, while the coefficient of determination would be 0.19. Depending on on your interest, either statistic could be useful.

Classification

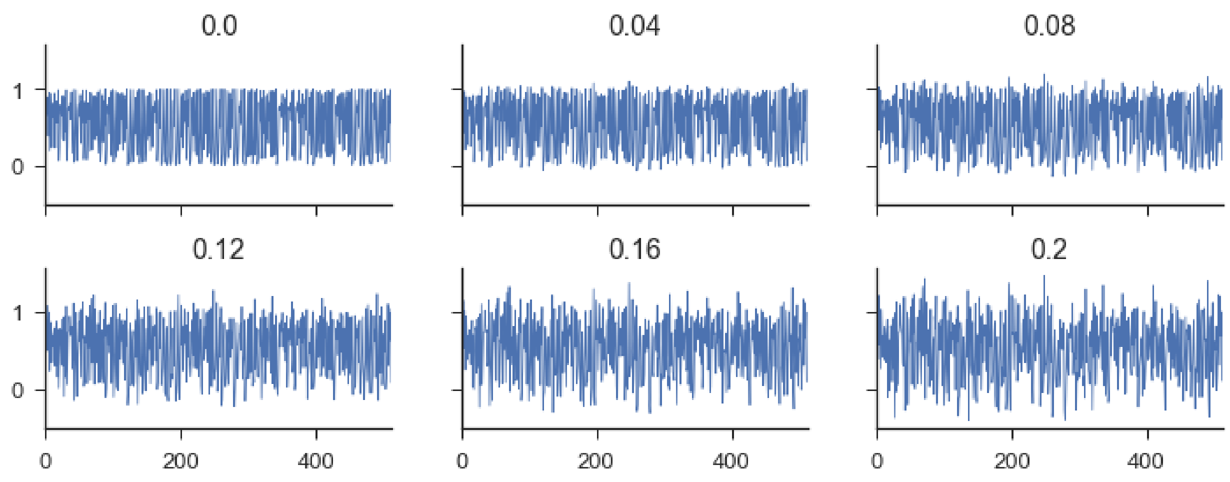
For classification, there are two ways to measure prediction skill. The first is by a simple percent accuracy. This calculates what percent were correctly predicted. The second way is by klecka's tau which attempts to normalize by the distribution of classes.

```
score1 = M.score(ytest, how='compare') # Percent correct
score2 = M.score(ytest, how='tau') # Kleck's tau
```

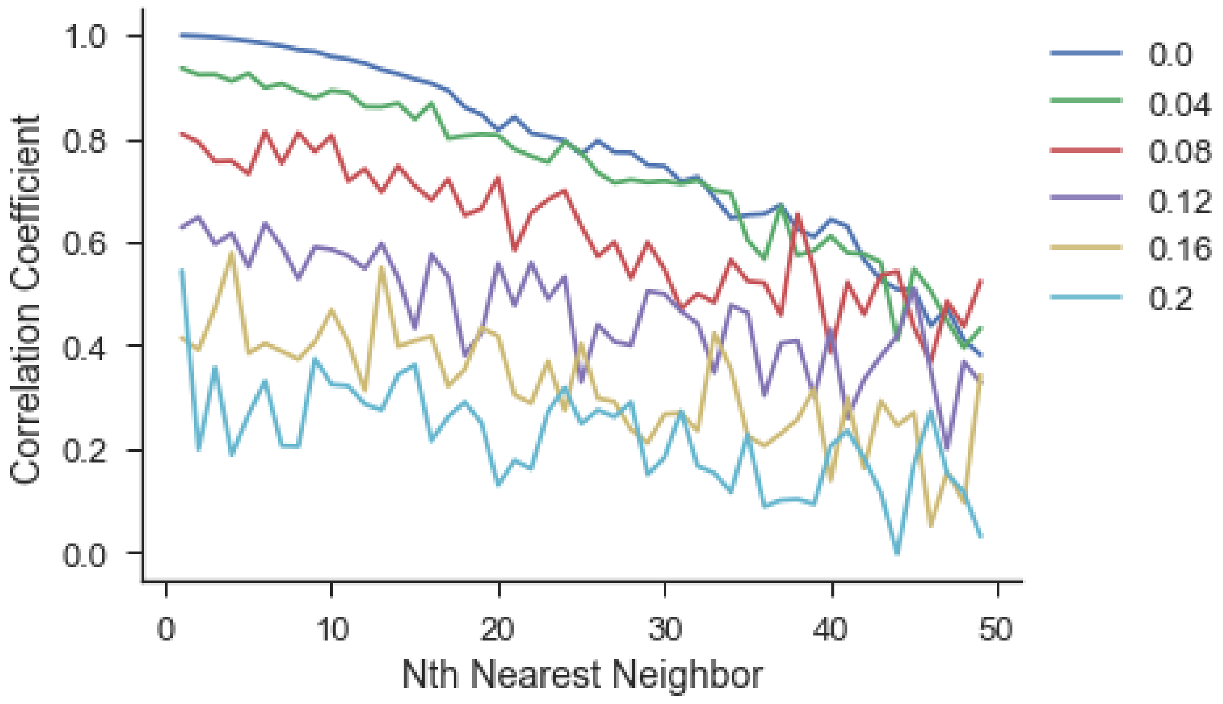
Percent correct is useful when you have balanced classes, but not useful when the classes are skewed. For example, if there are two classes and class 1 makes up 95% of the data. Predicting a 1 everywhere would show you 95% accuracy while klecka's tau would show an accuracy of about -8.8. Again, both stastics could be useful in the correct context.

Distinguishing Determinism

Here we analyze the logistic map with varying levels of noise as indicated by the title.



To begin, we follow the `fit`, `predict_individual`, and `score` routine as usual for every time series. Interested readers can find the code in the jupyter notebook. Next we look at the correlation coefficient for each series for one prediction forward in time. The legend refers to the level of noise that was added to the time series.



Analyzing the plot above, we can see that as the amount of noise increases, the forecast skill decreases. This is to be expected as adding stochasticity to a system makes it inherently more difficult to forecast. The next thing to notice is that the forecast skill decreases more rapidly for the series with less noise. It is more important in deterministic systems to grab neighbors that are nearby in space.

CHAPTER 7

Module Reference

class skedm.skedm.**Classification** (*weights='uniform'*)

Classification using a k-nearest neighbors method. Predictions can be made for each nearest neighbor (*predict_individual*) or by averaging the k nearest neighbors (*predict*).

Parameters **weights** (*str*) – Procedure to weight the near neighbors. Options:

- 'uniform' : uniform weighting
- 'distance' : weighted as 1/distance

Example

```
>>> X # embed series of shape (nsamples, embedding dimension)
>>> y # future trajectory of a point of shape (nsamples, num predictions)
>>> import skedm as edm
>>> R = edm.Classification()
>>> train_len = int(len(X)*.75) # train on 75 percent
>>> R.fit(X[0:train_len], y[0:train_len])
>>> preds = R.predict(X[train_len:], [0,10,20]) # test at 1, 10, and 20 nn
>>> score = M.score(ytest) # Calculate klecka's tau
```

dist_calc (*Xtest*)

Calculates the distance from the testing set to the training set.

Parameters **Xtest** (*2d array*) – Test features (nsamples, nfeatures).

dist_stats (*nn_list*)

Returns the mean and std of the distances for the given nn_list

fit (*Xtrain, ytrain*)

Fit the training data. Can also be thought of as reconstructing the attractor.

Parameters

- **Xtrain** (*2D array*) – Features of shape (nsamples,nfeatures).

- **ytrain** (*2D array*) – Targets of shape (nsamples,ntargets).

predict (*Xtest*, *nn_list*)

Make a prediction for a certain value of near neighbors

Parameters

- **Xtest** (*2d array*) – Contains the test features.
- **nn_list** (*1d array of ints*) – Neighbors to be tested.

Returns **Ypred** – Predictions returned for each nn value in nn_list. It is the same length as nn_list.

Return type list

predict_individual (*Xtest*, *nn_list*)

Make a prediction for each neighbor.

Parameters

- **Xtest** (*2d array*) – Contains the test features.
- **nn_list** (*1d array of ints*) – Neighbors to be tested.

Returns **Ypred** – Predictions returned for each nn value in nn_list. It is the same length as nn_list.

Return type list

score (*ytest*, *how='tau'*)

Evalulate the predictions.

Parameters

- **ytest** (*2d array*) – Contains the target values.
- **how** (*str*) – How to score the predictions. Possible values:
  - **'compare'** [Percent correctly predicted. For more info, see] utilities.class_compare.
  - **'error'** [Percent correct scaled by the most common prediction] of the series. See utilities.classification_error for more.
  - 'tau' : Kleckas tau

Returns **scores** – Scores for the predicted values. Shape (len(nn_list),num_preds)

Return type 2d array

class skedm.skedm.**Embed**(*X*)

Embed a 1d, 2d array, or 3d array in n-dimensional space. Assists in choosing an embedding dimension and a lag value.

Parameters **X** (*1d, 2d, or 3d array*) – Array to be embedded in n-dimensional space.

embed_vectors_1d (*lag*, *embed*, *predict*)

Embeds vectors from a one dimensional array in m-dimensional space.

Parameters

- **X** (*array*) – A 1-D array representing the training or testing set.
- **lag** (*int*) – Lag values 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** (*array of shape [num_vectors,embed]*) – A 2-D array containing all of the embedded vectors.
- **targets** (*array of shape [num_vectors,predict]*) – A 2-D array containing the evolution of the embedded vectors.

Example

```
>>> X = [0,1,2,3,4,5,6,7,8,9,10]
>>> em = 3
>>> lag = 2
>>> predict=3
>>> features, targets = embed_vectors_1d(lag, embed, predict)
>>> features # [[0,2,4], [1,3,5], [2,4,6], [3,5,7]]
>>> targets # [[5,6,7], [6,7,8], [7,8,9], [8,9,10]]
```

embed_vectors_2d(*lag*, *embed*, *predict*, *percent=0.1*)
Embeds vectors from a two dimensional image in m-dimensional space.

Parameters

- **X** (*array*) – A 2-D array representing the training set or testing set.
- **lag** (*tuple of ints (r,c)*) – Row and column lag values (r,c) can think of as (height,width).
- **embed** (*tuple of ints (r,c)*) – Row and column embedding shape (r,c) can think of as (height,width). c must be odd.
- **predict** (*int*) – Distance in the space to forecast (see example).
- **percent** (*float (default = None)*) – Percent of the space to embed. Used for computation efficiency.

Returns

- **features** (*array of shape [num_vectors,r*c]*) – A 2-D array containing all of the embedded vectors.
- **targets** (*array of shape [num_vectors,predict]*) – A 2-D array containing the evolution of the embedded vectors.

Example

```
>>> lag = (3,4)
>>> embed = (2,5)
>>> predict = 2
>>> features, targets = embed_vectors_2d(lag, embed, predict)
```

Notes

The embed space above looks like the following:

```
[f] _ _ _ [f] _ _ _ [f] _ _ _ [f] _ _ _ [f]
 |         |         |         |         |
 |         |         |         |         |
[f] _ _ _ [f] _ _ _ [f] _ _ _ [f] _ _ _ [f]
                    [t]
                    [t]
```

embed_vectors_3d(*lag*, *embed*, *predict*, *percent=0.1*)

Embeds vectors from a 3-dimensional matrix in n-dimensional space.

Parameters

- **X** (*array*) – A 3-D array representing the training set or testing set.
- **lag** (*tuple of ints (r,c)*) – Row and column lag values (r,c) can think of as (height,width).
- **embed** (*tuple of ints (r,c,t)*) – Row and column, and time embedding shape (r,c,t) can think of as (height,width,time). c must be odd.
- **predict** (*int*) – Distance in the space to forecast (see example).
- **percent** (*float (default = None)*) – Percent of the space to embed. Used for computation efficiency.

Returns

- **features** (*array of shape [num_vectors,r*c]*) – A 2-D array containing all of the embedded vectors
- **targets** (*array of shape [num_vectors,predict]*) – A 2-D array containing the evolution of the embedded vectors

Example

```
>>> lag = (3,4,2) #height,width,time
>>> embed = (3,3)
>>> predict = 2
>>> features, targets = embed_vectors_3d(lag, embed, predict)
```

Notes

The above example would look like the following:

```
[f] _ _ _ [f] _ _ _ [f]
 |         |         |
 |         |         |
[f] _ _ _ [f] _ _ _ [f]
 |         |         |
 |         |         |
[f] _ _ _ [f] _ _ _ [f]
```

The targets would be directly below the center [f].

mutual_information(*max_lag*)

Calculates the mutual information between a time series and a shifted version of itself. Uses numpy's mutual information for the calculation.

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

Returns **mi** – Mutual information values for every shift value. Shape (max_lag,).

Return type 1d array

`mutual_information_3d`(*max_lag*, *percent_calc=0.5*, *digitize=True*)

Calculates the mutual information along the rows and down columns at a certain number of indices (percent_calc) and returns the sum of the mutual informaiton along the columns and along the rows.

Parameters

- **`M`** (*`3-D array`*) – Input three-dimensional array.
- **`max_lag`** (*`integer`*) – Maximum amount to shift the space.
- **`percent_calc`** (*`float`*) – Percent of rows and columns to use for the mutual information calculation.

Returns

- **R_mut** (*1-D array*) – The mutual inforation averaged down the rows (vertical)
- **C_mut** (*1-D array*) – The mutual information averaged across the columns (horizontal)
- **Z_mut** (*1-D array*) – The mutual information averaged along the depth.

`mutual_information_spatial`(*max_lag*, *percent_calc=0.5*, *digitize=True*)

Calculates the mutual information along the rows and down columns at a certain number of indices (percent_calc) and returns the sum of the mutual informaiton along the columns and along the rows.

Parameters

- **`M`** (*`2-D array`*) – Input two-dimensional image.
- **`max_lag`** (*`integer`*) – Maximum amount to shift the space.
- **`percent_calc`** (*`float`*) – Percent of rows and columns to use for the mutual information calculation.

Returns

- **R_mut** (*1-D array*) – The mutual inforation averaged down the rows (vertical).
- **C_mut** (*1-D array*) – The mutual information averaged across the columns (horizontal).
- **r_mi** (*2-D array*) – The mutual information down each row (vertical).
- **c_mi** (*2-D array*) – The mutual information across the columns (horizontal).

class `skedm.skedm.`**`Regression`**(*weights='uniform'*)

Regression using a k-nearest neighbors method. Predictions can be made for each nearest neighbor (*predict_individual*) or by averaging the k nearest neighbors (*predict*).

Parameters **`weights`** (*`str`*) – How to weight the near neighbors. Options are:

- 'uniform' : uniform weighting
- 'distance' : weighted as 1/distance

Example

```
>>> X # embed time series of shape (nsamples, embedding dimension)
>>> y # future trajectory of a point of shape (nsamples, num predictions)
>>> import skedm as edm
>>> R = edm.Regression()
>>> train_len = int(len(X)*.75) # train on 75 percent
>>> R.fit(X[0:train_len], y[0:train_len])
>>> preds = R.predict(X[train_len:], [0,10,20]) # test at 1, 10, and 20 nn
>>> score = M.score(ytest) # Calculate coefficient of determination
```

`dist_calc` (*Xtest*)
Calculates the distance from the testing set to the training set.

Parameters **`Xtest`** (*`2D array`*) – Test features (nsamples, nfeatures).

`dist_stats` (*nn_list*)
Calculates the mean and std of the distances for the given nn_list.

Parameters **`nn_list`** (*`1d array of ints`*) – Neighbors to have their mean distance and std returned.

Returns

- **mean** (*1d array*) – Mean of the all the test distances corresponding to the nn_list.
- **std** (*1d array*) – Std of all the test distances corresponding to the nn_list.

`fit` (*Xtrain*, *ytrain*)
Fit the training data. Can also be thought of as populating the phase space.

Parameters

- **`Xtrain`** (*`2D array`*) – Embed training time series. Features shape (nsamples, nfeatures).
- **`ytrain`** (*`2D array`*) – Future trajectory of the points. Targets Shape (nsamples,ntargets).

`predict` (*Xtest*, *nn_list*)
Make a prediction for a certain value of near neighbors

Parameters

- **`Xtest`** (*`2d array`*) – Testing samples of shape (nsamples,nfeatures)
- **`nn_list`** (*`1d array`*) – Values of Near Neighbors to use to make predictions

Returns **ypred** – Predictions for ytest of shape(nsamples,num predictions).

Return type 2d array

`predict_individual` (*Xtest*, *nn_list*)
Make a prediction for each neighbor.

Parameters

- **`Xtest`** (*`2d array`*) – Contains the test features.
- **`nn_list`** (*`1d array of ints`*) – Neighbors to be tested.

`score` (*ytest*, *how='score'*)
Score the predictions.

Parameters

- **ytest** (*2d array*) – Target values.
- **how** (*str*) – How to score the predictions. Options include:

  -'score' : Coefficient of determination. -'corrcoef' : Correlation coefficient.

Returns **score** – Scores for the corresponding near neighbors.

Return type 2d array

utilities

`skedm.utilities.`**`class_compare`**(*preds*, *actual*)

Percent correct between predicted values and actual values.

Parameters

- **preds** (*1D array*) – Predicted values of shape (num samples,).
- **actual** (*1D array*) – Actual values from the testing set. Shape (num samples,).

Returns **cc** – Returns the correlation coefficient.

Return type float

`skedm.utilities.`**`classification_error`**(*preds*, *actual*)

Percent correct between predicted values and actual values scaled to the most common prediction of the space.

Parameters

- **preds** (*1D array*) – Predicted values of shape (num samples,).
- **actual** (*1D array*) – Actual values of shape (num samples,).

Returns **cc** – Returns the correlation coefficient

Return type float

`skedm.utilities.`**`cohens_kappa`**(*preds*, *actual*)

Calculates cohens kappa.

Parameters

- **preds** (*1D array*) – Predicted values of shape (num samples,).
- **test** (*array of shape (num samples,)*) – Actual values from the testing set.

Returns **c** – Returns the cohens_kappa.

Return type float

`skedm.utilities.`**`corrcoef`**(*preds*, *actual*)

Correlation Coefficient of between predicted values and actual values

Parameters

- **preds** (*1D array*) – Predicted values of shape (num samples,).
- **test** (*1D array*) – Actual values from the testing set of shape (num samples,).

Returns **cc** – Returns the correlation coefficient.

Return type float

`skedm.utilities.`**`keep_diversity`**(*X*, *thresh=1.0*)

Returns indices where the columns are not a single class.

Parameters

- **X** (*2d array of ints*) – Array to evaluate for diversity
- **thresh** (*float*) – Percent of species that need to be unique.

Returns **keep** – Array where true means there is more than one class in that row.

Return type 1d boolean array

Examples

```
>>> x = np.array([[1 1 1 1]
[2 1 2 3]
[2 2 2 2]
[3 2 1 4]])
>>> keep_diversity(x)
array([F,T,F,T])
```

`skedm.utilities.`**`kleckas_tau`**(*preds*, *actual*)

Calculates kleckas tau

Parameters

- **preds** (*1D array*) – Predicted values of shape (num samples,).
- **actual** (*1D array*) – Actual values of shape (num samples,).

Returns **tau** – Returns kleckas tau

Return type float

`skedm.utilities.`**`klekas_tau_spatial`**(*X*, *max_lag*, *percent_calc=0.5*)

Calculates the kleckas tau value between a shifted and unshifted slice of the space.

Parameters

- **X** (*2D array*) – Spatail image.
- **max_lag** (*integer*) – Maximum amount to shift the space.
- **percent_calc** (*float*) – How many rows and columns to use average over. Using the whole space is overkill.

Returns

- **R_mut** (*1D array*) – Klekas tau averaged down the rows (vertical).
- **C_mut** (*1-D array*) – Klekas tau averaged across the columns (horizontal).
- **r_mi** (*2-D array*) – Klekas tau down each row (vertical).
- **c_mi** (*2-D array*) – Klekas tau across each columns (horizontal).

`skedm.utilities.`**`mi_digitize`**(*X*)

Digitize a time series for mutual information analysis

Parameters **X** (*1D array*) – Array to be digitized of length m.

Returns **Y** – Digitized array of length m.

Return type 1D array

`skedm.utilities.`**`score`**(*preds*, *actual*)

Calculates the coefficient of determination.

The coefficient R^2 is defined as (1 - u/v), where u is the regression sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual sum of squares ((y_true - y_true.mean()) ** 2).sum(). Best possible score is 1.0, lower values are worse.

Parameters

- **`preds`** (*`1D array`*) – Predicted values of shape (num samples,)
- **`test`** (*`1D array`*) – Actual values of shape (num samples,)

Returns **cc** – Returns the coefficient of determination.

Return type float

`skedm.utilities.`**`variance_explained`**(*preds*, *actual*)

Explained variance between predicted values and actual values.

Parameters

- **`preds`** (*`1D array`*) – Predict values of shape (num samples,).
- **`actual`** (*`1D array`*) – Actual values of shape (num samples,).

Returns **cc** – Returns the correlation coefficient

Return type float

`skedm.utilities.`**`weighted_mean`**(*X*, *distances*)

Calculates the weighted mean given a set of values and their corresponding distances.

Only 1/distance is implemented. This essentially is just a weighted mean down axis=1.

Parameters

- **`X`** (*`2d array`*) – Training values of shape(nsamples,number near neighbors).
- **`distances`** (*`2d array`*) – Sorted distances to the near neighbors for the indices. Shape(nsamples,number near neighbors).

Returns **w_mean** – Weighted predictions.

Return type 2d array

`skedm.utilities.`**`weighted_mode`**(*a*, *w*, *axis=0*)

This function is borrowed from sci-kit learn's extmath.py

Returns an array of the weighted modal (most common) value in a

If there is more than one such value, only the first is returned. The bin-count for the modal bins is also returned.

This is an extension of the algorithm in scipy.stats.mode.

Parameters

- **`a`** (*`array_like`*) – n-dimensional array of which to find mode(s).
- **`w`** (*`array_like`*) – n-dimensional array of weights for each value.
- **`axis`** (*`int, optional`*) – Axis along which to operate. Default is 0, i.e. the first axis.

Returns

- **vals** (*ndarray*) – Array of modal values.
- **score** (*ndarray*) – Array of weighted counts for each mode.

Examples

```
>>> from sklearn.utils.extmath import weighted_mode
>>> x = [4, 1, 4, 2, 4, 2]
>>> weights = [1, 1, 1, 1, 1, 1]
>>> weighted_mode(x, weights)
(array([ 4.]), array([ 3.]))
```

The value 4 appears three times: with uniform weights, the result is simply the mode of the distribution.

```
>>> weights = [1, 3, 0.5, 1.5, 1, 2] # deweight the 4's
>>> weighted_mode(x, weights)
(array([ 2.]), array([ 3.5]))
```

The value 2 has the highest score: it appears twice with weights of 1.5 and 2: the sum of these is 3.

See also:

`scipy.stats.mode()`

CHAPTER 8

Performance

Here we provide some basic guidelines on performance. All of these were run on an early 2015 macbook air. The code can be found in the accompanying jupyter notebook.

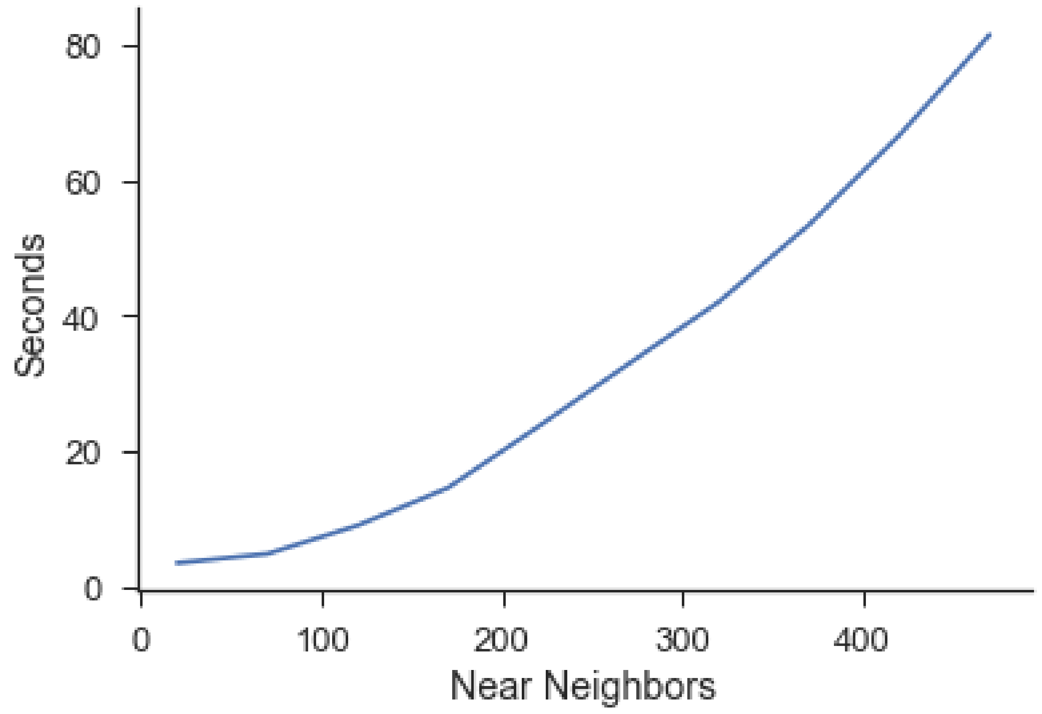
As a baseline, the example given in the Quick Example section completes in 4.2 seconds.

```
X = data.lorenz(sz=10000)[:,0] #only going to use the x values
E = edm.Embed(X)

lag = 15
embed = 3
predict = 30 #predicting out to double to lag
X,y = E.embed_vectors_1d(lag,embed,predict)

train_len = int(.75*len(X))
Xtrain = X[0:train_len]
ytrain = y[0:train_len]
Xtest = X[train_len:]
ytest = y[train_len:]

M = edm.Regression() # initiate the nonlinear forecasting class
M.fit(Xtrain,ytrain) #fit the training data

nn_list = [10,100,500,1000]
ypred = M.predict(Xtest,nn_list)
score = M.score(ytest) #score the predictions against the actual values
```

In the following sections we expand this example by changing the amount of near neighbors, training size, and testing size. This will give us an idea of the performance impacts of exploring different data sets.

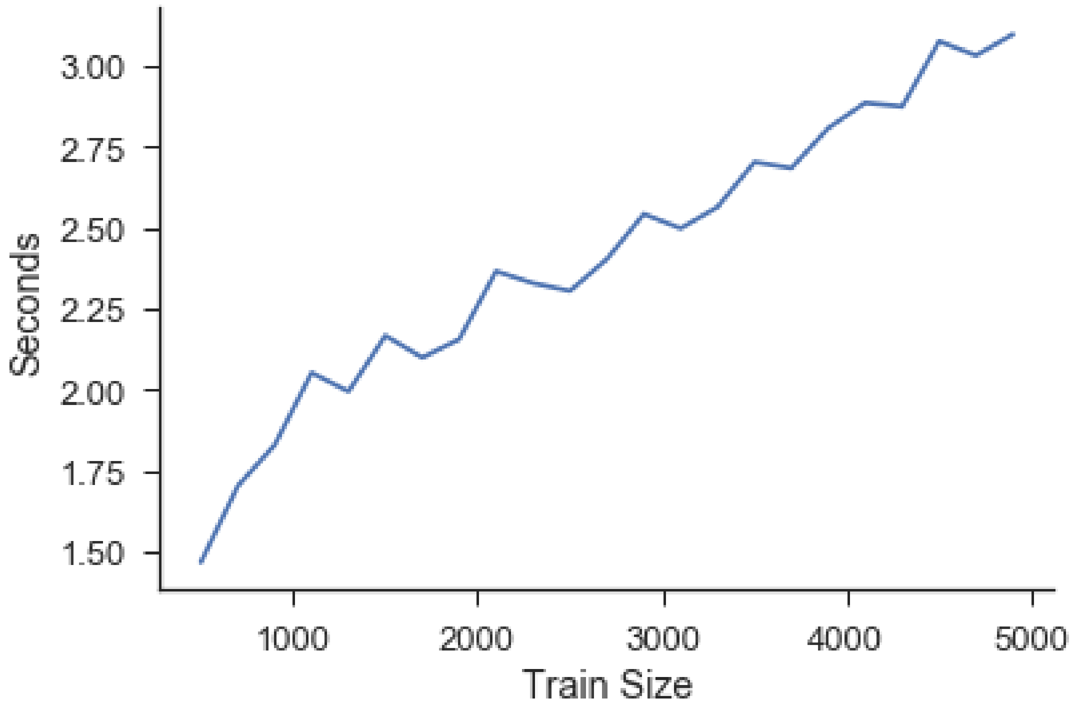
Near Neighbor

Here we increase the amount of near neighbors with everything else held constant. This is the time it takes to complete every near neighbor value between the one given on the x axis and 1. For example, the first point is the amount of time it takes to complete the near neighbor calculation for [1,2,3,4,5,6,7,8,9,10] near neighbors.



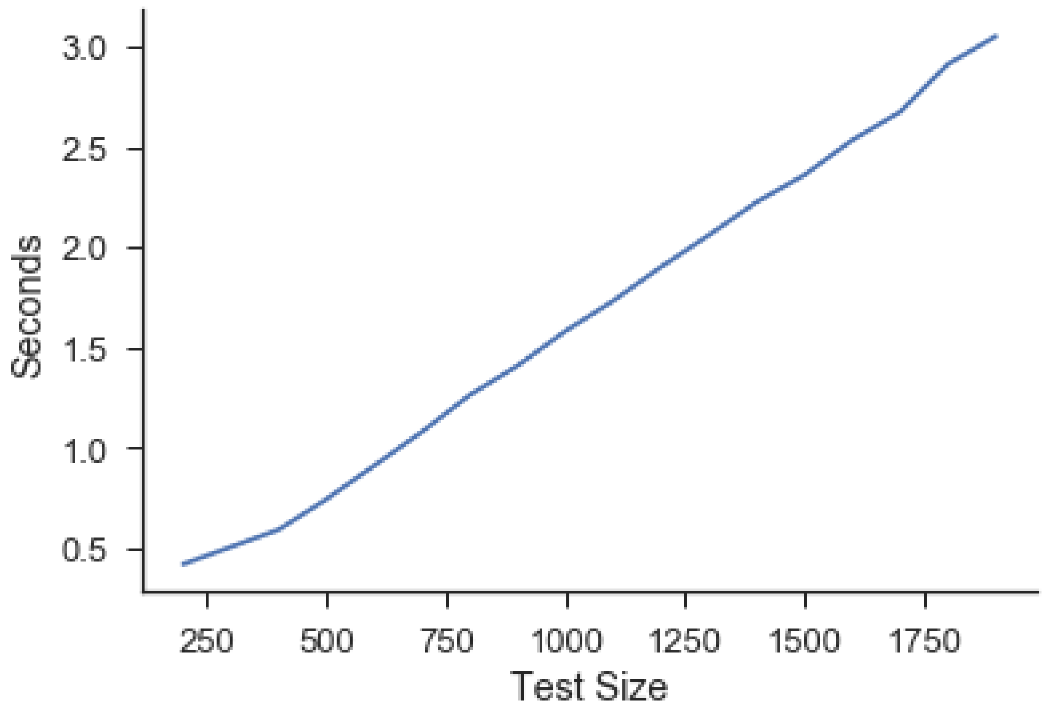
Train Size

Here the testing size, and the number of near neighbors is held constant. We simply iterate through the length of the training set.



Test Size

Here the training size and the number of near neighbors is held constant. We simply iterate through the length of the testing set.


Seconds
3.0
2.5
2.0
1.5
1.0
0.5
250
500
750
1000
1250
1500
1750
Test Size

CHAPTER 9

Acknowledgements

This work resulted from the Reefs Tomorrow Intiative funded by the Gordon and Betty Moore Foundation. I would also like to thank Dr. Dylan McNamara for continous feedback and for introducing me to emperical dynamic modeling. I would also like to thank Dr. Kenneth Ells and the rest of the CASL group for proofreading and testing this package.

Python Module Index

Index