Entropy and Lorenz Model

December 2021

1 Entropy

1.1 Theory svd entropy

We can find online the documentation of the function svd-entropy [[1]].

$$S_{svd} = S(\mathbf{x(t)}, order, delay) \tag{1}$$

We denote with $\mathbf{x}$ the dataset that we want to study.

$$\begin{aligned} \mathbf{x}(t_0) &= \mathbf{x}_0, \\ \mathbf{x}(t) &= (x_1, x_2, x_3, \ldots, x_N) \end{aligned} \tag{2}$$

This function creates a matrix Y with the dataset.

$$\mathbf{y}(i) = (x_i, x_{i+\mathrm{delay}}, ..., x_{i+(\mathrm{order}-1)\mathrm{delay}}) \tag{3}$$

$$Y = \begin{pmatrix} \mathbf{y}(1) \\ \mathbf{y}(2) \\ . \\ . \\ . \\ \mathbf{y}(N-(ord-1)del) \end{pmatrix}$$

If we take as parameters $order = 3$ and $delay = 1$ we obtain the following matrix:

$$Y = \begin{pmatrix} x_1 & x_2 & x_3 \\ x_2 & x_3 & x_4 \\ . & . & . \\ . & . & . \\ . & . & . \\ . & x_{N-1} & x_N \end{pmatrix}$$

[1] `https://raphaelvallat.com/entropy/build/html/generated/entropy.svd_entropy.html`

The next step is to write this matrix using the SVD decomposition.

$$Y = U\Sigma V^* \tag{4}$$

Where U is an unitary matrix, V is also an unitary matrix, and Σ is a diagonal matrix. This is linked with PCA. Now we denotes the eigenvalues of the matrix Σ with σ_i. Now we can compute the average eigenvalues:

$$\bar{\sigma}_k = \frac{\sigma_k}{\sum_j^M \sigma_j} \tag{5}$$

Where M is the number of eigenvalues.

After this we can compute the SVD Entropy:

$$S_{svd} = -\sum_k^M \bar{\sigma}_k log_2(\bar{\sigma}_k) \tag{6}$$

1.2 Shannon Entropy

We analyze the Shannon Entropy function [2].

```python
def shannon_entropy(time_series):
    """Return the Shannon Entropy of the sample data.
    Args:
        time_series: Vector or string of the sample data
    Returns:
        The Shannon Entropy as float value
    """

    # Check if string
    if not isinstance(time_series, str):
        time_series = list(time_series)

    # Create a frequency data
    data_set = list(set(time_series))
    freq_list = []
    for entry in data_set:
        counter = 0.
        for i in time_series:
            if i == entry:
                counter += 1
        freq_list.append(float(counter) / len(time_series))

    # Shannon entropy
    ent = 0.0
    for freq in freq_list:
        ent += freq * np.log2(freq)
    ent = -ent
    return ent
```

$$S_{sh} = S_{sh}(\mathbf{x}) \tag{7}$$

[2] https://github.com/nikdon/pyEntropy/blob/master/pyentrp/entropy.py

Here we find the definition of Shannon Entropy:

$$S_{sh} = -\sum_{i} P(x_i) log_2(P(x_i)) \tag{8}$$

Where $P(x_i)$ is the frequency of x_i in the dataset $\mathbf{x}$.

1.2.1 Technical Issues

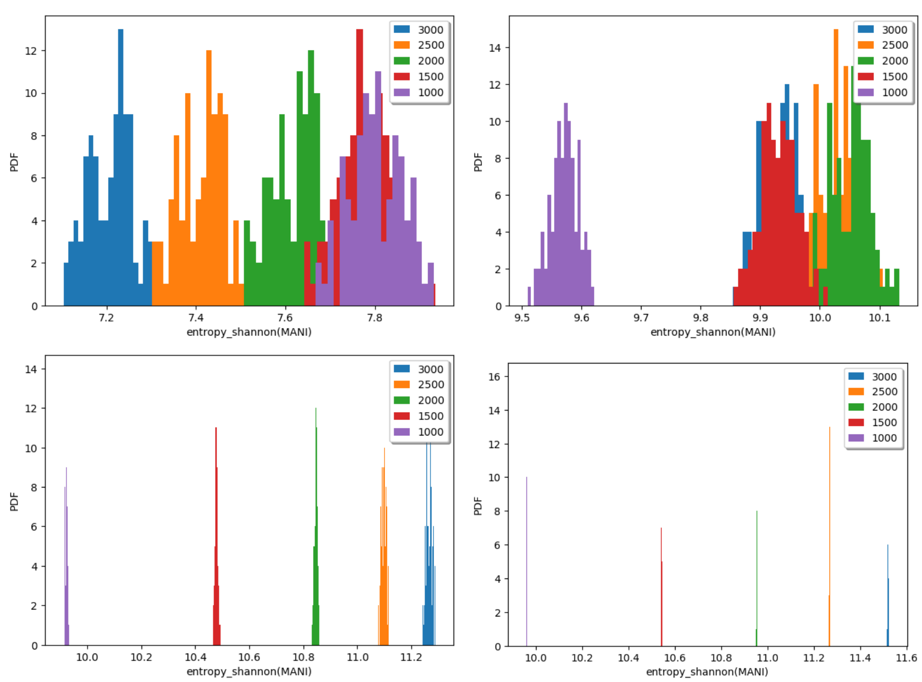
Lorenz's Orbit trajectory is a timeseries with elements of type float64. A classic Shannon Entropy algorithm has to compute the relative frequency of each element in the array, if it's a float64 values vector it's almost impossible to find an equality between nubmers. Infact using the function "torch.load()" creats a vector of loat64 arrays with a precision of 15 decimals. That's why to compute entropy it is useful to use another type of equality.

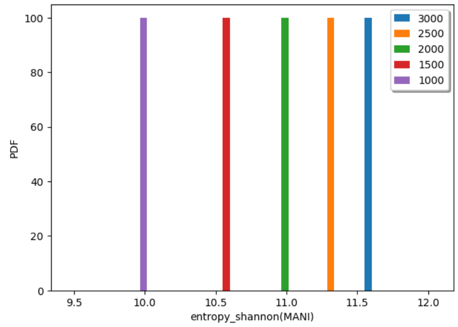
$$\begin{aligned} x(t_a) &= a_0, a_1 a_2 \dots a_{15} \\ x(t_b) &= b_0, b_1 b_2 \dots b_{15} \end{aligned} \tag{9}$$

We say that $x(t_a) = x(t_b)$ with precision p if :

$$a_0 = b_0,\ a_1 = b_1,\ \dots,\ a_p = b_p \tag{10}$$

That's why there are differents plots of Shannon Entropy. With $p = 1, 2, 3, 4, 15$





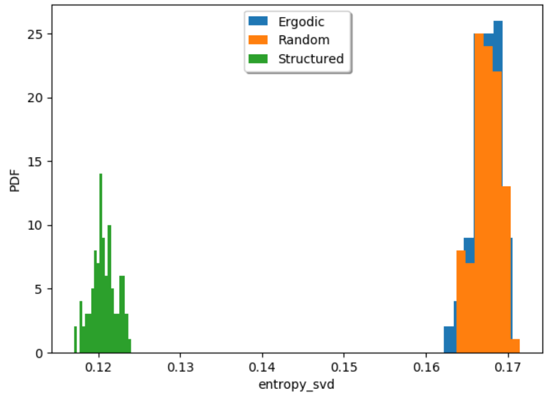
1.3 Approximate Entropy

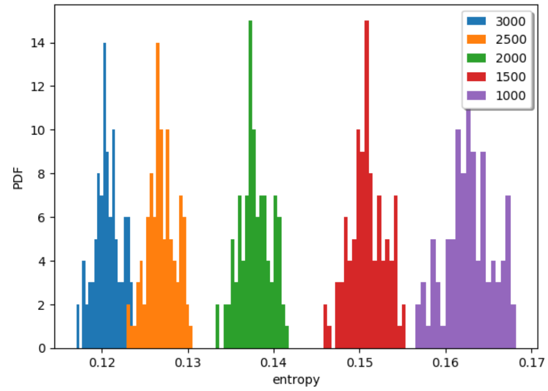
```
def app_entropy(x, order=2, metric='chebyshev'):

    phi = _app_samp_entropy(x, order=order, metric=metric, approximate=True)    # it defines the vector phi
    return np.subtract(phi[0], phi[1])

def _app_samp_entropy(x, order, metric='chebyshev', approximate=True):
    """Utility function for `app_entropy`` and `sample_entropy`.
        """
    #technical stuff

    _all_metrics = KDTree.valid_metrics
    if metric not in _all_metrics:
        raise ValueError('The given metric (%s) is not valid. The valid '
                         'metric names are: %s' % (metric, _all_metrics))



    phi = np.zeros(2) #vector phi
    r = 0.2 * np.std(x, ddof=0) #radium to define the concept of neighbourds (0.2* standard deviation with 0 DF of the timeseries)

    # compute phi(order, r)
    _emb_data1 = _embed(x, order, 1) #creates a matrix with the timeseries Nx2
    if approximate: # TRUE
        emb_data1 = _emb_data1
    else:
        emb_data1 = _emb_data1[:-1]
    count1 = KDTree(emb_data1, metric=metric).query_radius(emb_data1, r,
                                                           count_only=True
                                                           ).astype(np.float64)
```

```
                                                                         # compute phi(order + 1, r)
emb_data2 = _embed(x, order + 1, 1) # same matrix but Nx3
count2 = KDTree(emb_data2, metric=metric).query_radius(emb_data2, r,
                                                       count_only=True
                                                       ).astype(np.float64)            # count the number of neighbours according to Chebyshev norm for
                        # each point and put this number in a vector.

if approximate: #true
    phi[0] = np.mean(np.log(count1 / emb_data1.shape[0]))
    phi[1] = np.mean(np.log(count2 / emb_data2.shape[0]))
    else:
        phi[0] = np.mean((count1 - 1) / (emb_data1.shape[0] - 1))
        phi[1] = np.mean((count2 - 1) / (emb_data2.shape[0] - 1))
    return phi






def _embed(x, order=3, delay=1):

    N = len(x)
    if order * delay > N:
        raise ValueError("Error: order * delay should be lower than x.size")
    if delay < 1:
        raise ValueError("Delay has to be at least 1.")
    if order < 2:
        raise ValueError("Order has to be at least 2.")
    Y = np.zeros((order, N - (order - 1) * delay))
    for i in range(order):
        Y[i] = x[(i * delay):(i * delay + Y.shape[1])]
    return Y.T


@jit('UniTuple(float64, 2)(float64[:], float64[:])', nopython=True)
```

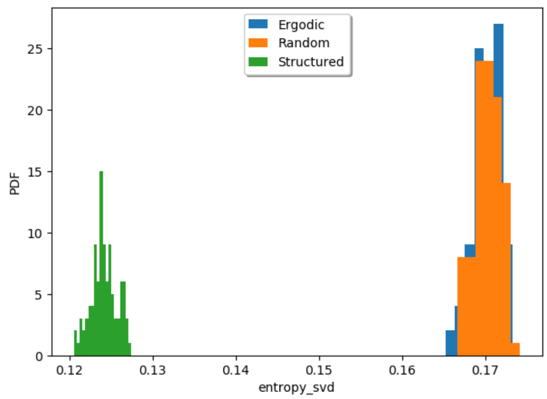
1.4 Plots Entropy

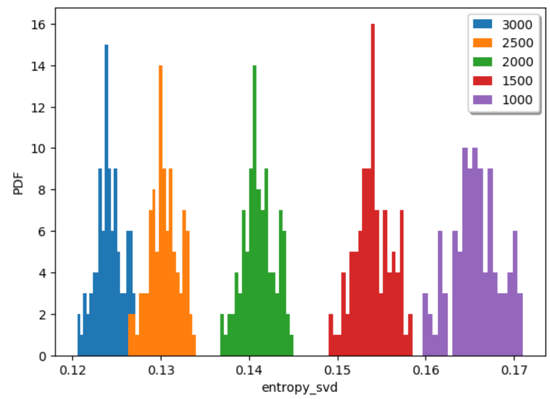
1.4.1 SVD with order 3



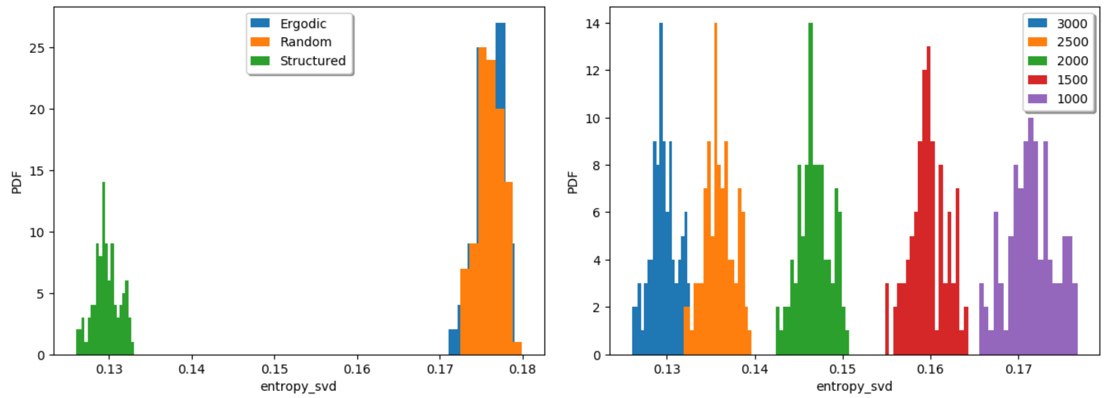


1.4.2 SVD with order 4

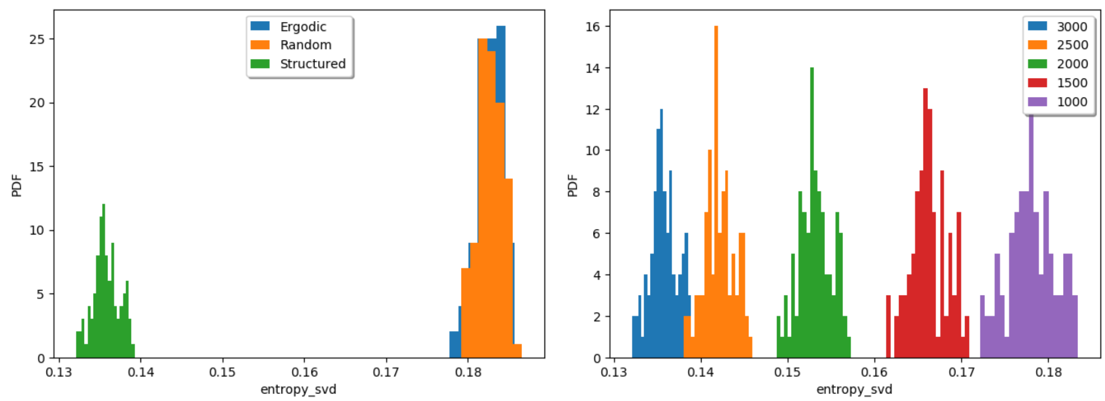




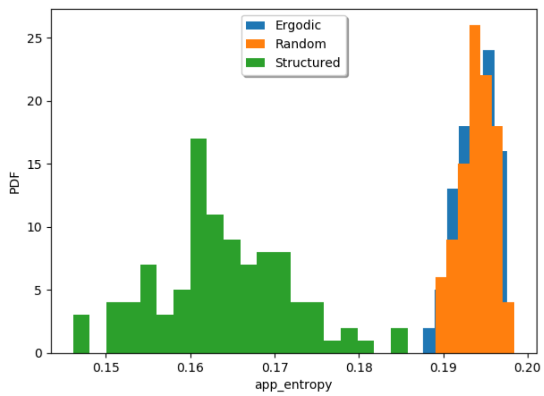
1.4.3 SVD with order 5

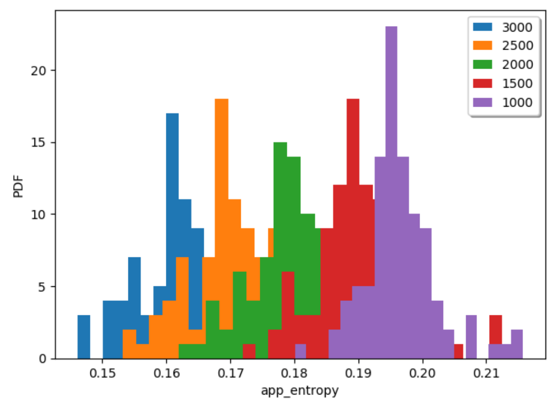


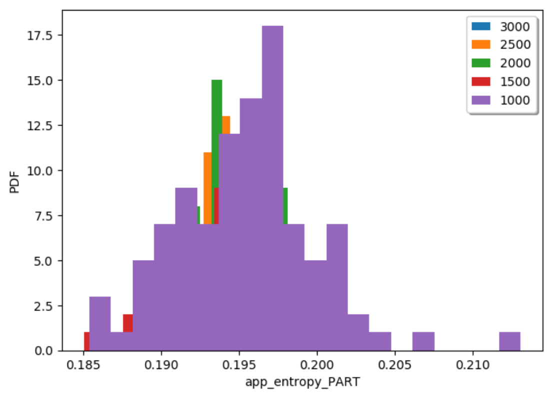
1.4.4 SVD with order 6



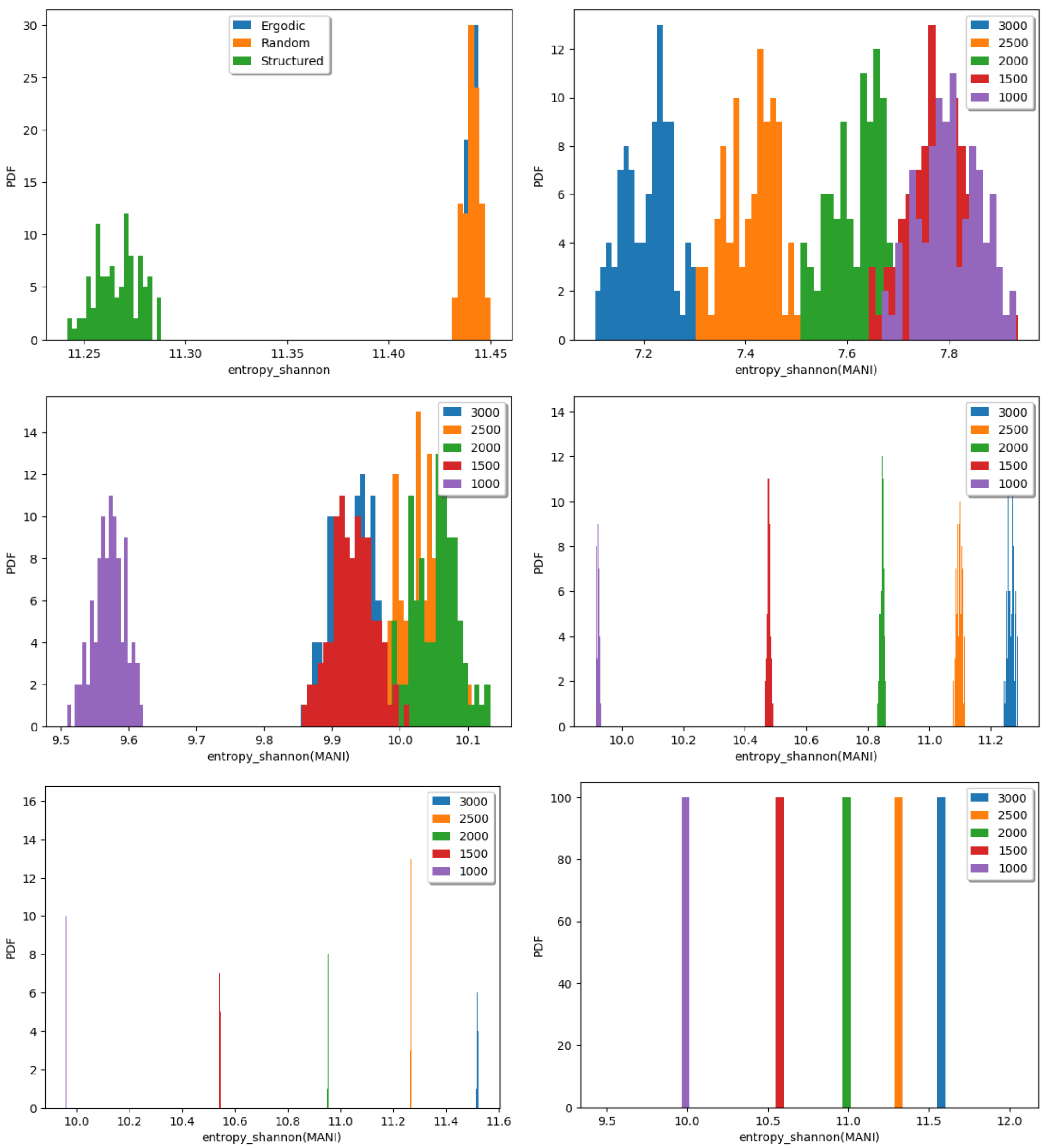
1.4.5 Approx Entropy







1.4.6 Shannon Entropy



1,2,3,4,exact digits precision.

2 Appendix

2.1 Chebyshev Norm