Notes for FastEnt (GetLambdasFunction) Implementation

Bridget Smart
bridget.smart@adelaide.edu.au

10 Feb 2023

0.1 Overview

1 Notation

Let the string from the source and target processes be denoted, S and T respectively.

The realisations from each process are drawn from a random process with a finite sample space containing values from the set $\mathcal{V}$.

When applied to language, $\mathcal{V}$ represents the vocabulary with each value representing a distinct word and $|\mathcal{V}|$ representing the true vocabulary size.

For the sequence S, we let $S_i^j, i \leq j$ represent the subsequence given by $(S_i, S_{i+1}, ..., S_j)$.

To estimate the Shannon cross entropy rate between S and T, at an instant i, we estimate the entropy rate between S_0^i and T_{i+1}^n. That is, we consider the cross entropy between $S_0, ..., S_i$ and $T_{i+1}, ..., T_n$.

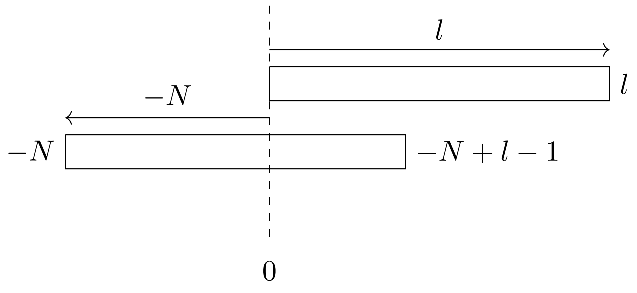

Figure 1: $\tilde{N}_l(x)$ as defined by Wyner Ziv Wyner & Ziv (1989).

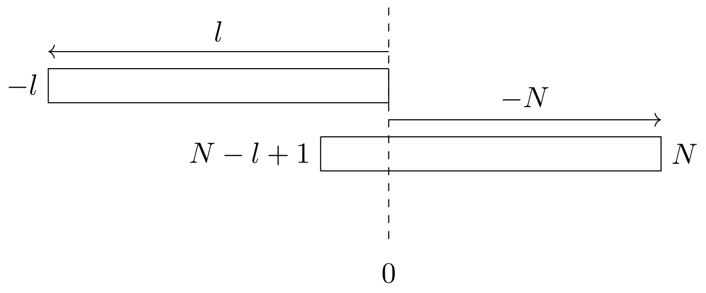

Figure 2: $N_l(x)$ as defined by Wyner Ziv Wyner & Ziv (1989).

Variable	Definition
n	length of 'history'
l	length of matching sequence
L_n (WZ)	L_n is smallest integer l such that x_0^{l-1} does not start in the past X_{-n}^{-1}. $$X_0^{l-1} \neq X_{-m}^{-m+L-1},$$ for $1 \leq m \leq n$. These sequences CAN overlap, so L_n represents length of the sequence (ie so far unseen sequence) for a history length of n where the sequence can start anywhere prior to 0.
L_n (K)	L_n is smallest integer l such that x_0^{l-1} does not occur in the past X_{-n}^{-1}. Alternatively, this is equal to one greater than the longest match length $$L_n = 1 + \max\{l : 0 \leq l \leq n, X_0^{l-1} = X_{-m}^{-m+l-1}\}$$ for $1 \leq m \leq n$. These sequences CANNOT overlap, so L_n represents length of the sequence (ie so far unseen sequence) for a history length of n
$\tilde{N}_l(x)$ (WZ)	smallest integer $N > 0$ which describes the size of the gap between matches of length l. $\boldsymbol{x}_0^{l-1} = \boldsymbol{x}_{-N}^{l-1-N}$ (match from 0 to l-1 back in time N places)
$N_l(x)$ (WZ)	$\tilde{N}_l(x)$ with 'time reversed' ie $\boldsymbol{x}_{-l+1}^0 = \boldsymbol{x}_{N-l+1}^N$, ie slide sequence into the future to match
Λ_i^n (K)	Λ_i^n is the length of the shortest substring X_i^{i+l-1} (length l) starting at position i that does not appear as a contiguous substring of the previous n symbols X_{i-n}^{i-1}. This is a mis-quote from Wyner Ziv (seems to lose its boundary crossing abilities between WZ to OW and K).
$\ddot{\Lambda}_i^n$ (Proposed to simplify notation)	Λ_i^n is the length of the shortest substring X_i^{i+l-1} (length l) starting at position i that does not appear as a contiguous substring of the previous n symbols X_{i-n}^{i-1}. This is a mis-quote from Wyner Ziv (seems to lose its boundary crossing abilities between WZ to OW and K).

Table 1: Overview of notation

2 Entropy estimators

For each value of m, we calculate the match lengths l, such that $S_m, ..., S_{m+l} = T_i, ..., T_{i+l}$ for $m + l \leq i$. We then set, $\Lambda_i = \max(l) + 1$, which represents the length of the longest unseen sequence beginning at instant i.

To estimate the entropy, we calculate each value of Λ_i for $i \leq n$, and use one of the following approximations proposed by Kontoyiannis:

(a)

$$\lim_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} \frac{\Lambda_i^n}{\log n} \to \frac{1}{H}$$

(b)

$$\lim_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} \frac{\Lambda_i^i}{\log i} \to \frac{1}{H}$$

(c)

$$\lim_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} \frac{\Lambda_i^i}{\log n} \to \frac{1}{H}$$

which converges almost surely Ornstein & Weiss (1993), Wyner & Ziv (1989), Kontoyiannis et al. (1998).

The implementation in ProcessEntropy uses (c).

3 Algorithm for Process Ent

- Generally we do not allow overlaps here

The goal for S and T is to find the longest match length. i.e. we find $\max L_n + 1$. L_n is smallest integer l such that the target sequence T_0^{l-1} does not start in the past of the source S_{-n}^{-1}.

$$T_0^{l-1} \neq S_{-m}^{-m+l-1},$$

for $1 \leq m \leq n$. Where n is the history size of the source.

Generally, the process is:

- Iterates through possible i values (start of target sequence)
- For each i, returns the longest match length for any sequence in the source which starts before the start of the target sequence.

3.1 Self Lambdas

Listing 1: Code for calculating self lambdas (ProcessEntropy)

```
def get_all_self_lambdas(source, lambdas):
    """
    Internal function.
    Finds the Lambda value for each index in the source.
    Lambda value denotes the longest subsequence of the source,
    starting from the index, that in contained contiguously in the source,
    before the index.

    Args:
        source: Arry of ints, usually corresponding to hashed words.

        lambdas: A premade array of length(target), usually filled with zeros.
            Used for efficiency reasons.

    Return:
        A list of ints, denoting the value for Lambda for each index in the target.

    """

    N = len(source)

    for i in prange(1, N):

        # The target process is everything ahead of i.
        t_max = 0
        c_max = 0

        for j in range(0, i): # Look back at the past
            if source[j] == source[i]: # Check if matches future's next element
                c_max = 1
                for k in range(1,min(N-i, i-j)): # Look through more of future
                    if source[j+k] != source[i+k]:
                        break
                    else:
                        c_max = c_max+1

                if c_max > t_max:
                    t_max = c_max

        lambdas[i] = t_max+1

    return lambdas
```

3.2 Pseudocode

Algorithm 1 Outline of data simulation procedure for N processes

```
**Variables**
$N \leftarrow$ length of source
**for** $i$ in $1, ..., N$ **do**
    {Target process is everything ahead of $i$ (inclusive)}
    {Source $= X_0, ..., X_{i-1}$}
    {Target $= X_i, ..., X_N$}
    $l \leftarrow 0$
    {Current length is 0}
    **for** $j = 0, ..., i-1$ **do**
        **if** $X_i == X_j$ **then**
            $\lambda = 1$
            **for** $k$ in $1, ..., \min(N-i, i-j)$ **do**
                {conditions stop impossible matches}
                {$(i-j)$ is the max length of source before it overlaps target}
                {$(N-i)$ is the max length of target before we run out of symbols}
                Continue incrementing $\lambda$ while $X_{i+k} = X_{j+k}$
            **end for**
        **end if**
        $\lambda_m$, Save $\lambda$ if greater than current max $\lambda$.
    **end for**
    return $\lambda_m + 1$
**end for**
```

3.3 Notes

Essentially, we don't allow overlaps. I.e. we never check for a target which extends to i, with $j + k < i$.

But we do check the very last element in the list, i.e. $i + k \leq N$.

3.4 Cross Ent version

Listing 2: Code for calculating cross-entropy lambdas (ProcessEntropy)

```python
@jit(nopython=True, fastmath=True)
def find_lambda_jit(target, source):
    """
    Finds the longest subsequence of the target array,
    starting from index 0, that is contained in the source array.
    Returns the length of that subsequence + 1.

    i.e. returns the length of the shortest subsequence starting at 0
    that has not previously appeared.

    Args:
        target: NumPy array, preferable of type int.
        source: NumPy array, preferable of type int.

    Returns:
        Integer of the length.

    """

    source_size = source.shape[0]-1
    target_size = target.shape[0]-1
    t_max = 0
    c_max = 0

    for si in range(0, source_size+1):
        if source[si] == target[0]:
            c_max = 1
            for ei in range(1,min(target_size, source_size - si+1)):
                if(source[si+ei] != target[ei]):
                    break
                else:
                    c_max = c_max+1

            if c_max > t_max:
                t_max = c_max

    return t_max+1



@jit(nopython=True, parallel=True)
def get_all_lambdas(target, source, relative_pos, lambdas):
    """
    Finds all the the longest subsequences of the target,
    that are contained in the sequence of the source,
    with the source cut-off at the location set in relative_pos.

    See function find_lambda_jit for description of
        Lambda_i(target|source)

    Args:
        target: Array of ints, usually corresponding to hashed words.

        source: Array of ints, usually corresponding to hashed words.

        relative_pos: list of integers with the same length as target denoting the
            relative time ordering of target vs. source. These integers tell us the
            position relative_pos[x] = i in source such that all symbols in source[:↘
                i]
            occurred before the x-th word in target.

        lambdas: A pre-made array of length(target), usually filled with zeros.
            Used for efficiency reasons.

    Return:
        A list of ints, denoting the value for Lambda for each index in the target.

    """
    i = 0
    while relative_pos[i] == 0: # Preassign first values to avoid check
```

```
            lambdas[i] = 1
            i+=1

        # Calculate lambdas
        for i in prange(i, len(target)):
            lambdas[i] = find_lambda_jit(target[i:], source[:relative_pos[i]])

        return lambdas
```

3.5 Pseudocode

Basically the same

starting with data in the form:

$$[t_i^a, a, [X_0^a, ..., X_{N_a}^a]]$$

where a is ‘source’ or ‘target’, we have [time, label, set of symbols in each event (tweet)]

we process this into

list of all symbols for source list of all symbols for target list of indicies such that relpos$[x] = i$ if source$[: i]$ (excluding at i) occur prior to target$[x :]$

Then apply similar process to self entropy except:

source$[: i]$ target$[x :]$

Ns - max index in source takes the place of N (we iterate to the value Ns and use it) j is fixed at 0 for k in range(1, min(Nt (length of target), $Ns - i + 1$)) (max possible length of sequence in source)

3.6 Bugs

but doesn’t ever check the last element? should be - does check the last element since using N_T (length of target) $=$ target_size $- 1$ N_S (length of source) $=$ source_size $- 1$

in the code, we iterate the length of the matching sequence up to:

$$min(\text{target\_size}, \text{source\_size} - si + 1),$$

which is equal to

$$= \min(N_T - 1, N_s - si).$$

To understand this, we have condition 1 $(N_T - 1) = 1$ - length of target

BUT SHOULD BE LENGTH OF TARGET

where condition 2 $(N_s - si)=$ length of source

To see this, we find that

```
        find_lambda_jit(np.arange(10), np.arange(10)) = find_lambda_jit(np.array↘
            ([0,1,2,3,4,5,6,7,8,8]),t np.arange(10))
```

i.e. last element of target is never checked.

Code shoule be:

```
    def find_lambda_jit(target, source):
        """
        Finds the longest subsequence of the target array,
        starting from index 0, that is contained in the source array.
        Returns the length of that subsequence + 1.

        i.e. returns the length of the shortest subsequence starting at 0
        that has not previously appeared.

        Args:
            target: NumPy array, preferable of type int.
            source: NumPy array, preferable of type int.

        Returns:
            Integer of the length.

        """

        source_size = source.shape[0]-1
        target_size = target.shape[0]-1
        t_max = 0
        c_max = 0

        for si in range(0, source_size+1):
            if source[si] == target[0]:
                c_max = 1
                for ei in range(1,min(target_size+1, source_size - si+1)):
                    if(source[si+ei] != target[ei]):
                        break
                    else:
                        c_max = c_max+1

                if c_max > t_max:
                    t_max = c_max

        return t_max+1
```

This doesn't become a problem because in `timeseries_cross_entropy`, the relative position always puts target after source if times are equal.

4 Algorithm for Fast Ent (math version)

5 Algorithm Outline

This algorithm employs properties of a sorted suffix array to allow the longest match length to be found in O(1) with O(N) precomputation. This is a property of range minimum queries using a sorted suffix array.

Recall that a sorted suffix array produces a vector of indices which each refer to a suffix sorted in lexicographical order.

Eg. string aabab has the following suffixes, which we can refer to using the following indices of the string (aabab) as described below:

Suffix:	Index:
b	5
ab	4
bab	3
abab	2
aabab	1

So, we could sort these suffixes and represent this order as follows:

sorted suffix array : 1 4 2 5 3

In parallel, we can also compute the longest common prefix between adjacent suffixes in the sorted suffix array. Eg the first value, 1, represents the length of the longest common prefix between suffixes starting at index 1 (aabab) and 4 (ab).

longest common prefix table : 1 2 0 1 0 (the final zero is chosen arbitrarily).

On the longest common prefix table we can use a range minimum query (RMQ) to find the length of the longest common prefix between two suffixes. For example the longest common prefix between suffixes 4 and 3 is given by the minimum of the corresponding section of the longest common prefix table (inclusive at the start and exclusive at the end).

SSA : 1 **[4 2 5 3)**
LCP : 1 **[2 0 1 0)**

which is 0.

Prefix doubling allows the LCP, SSA and RMQ to be constructed simultaneously.

Given the two strings of interest S and T, where we are looking for a match starting at i in S which starts prior to index j in T.

Begin by constructing a mega string given by $S + n{+}1 + T$ where n is the length of the alphabet and $n+1$ is a symbol not in our alphabet. This prevents a match from starting in S and ending in T.

With this string we can construct the LCP, SSA and RMA and resolve the query with (i,j) by finding the first element which corresponds to a suffix in T starting before j to both the left and right of the suffix corresponding to i. By the range minimum property of the SSA, this guarantees us that we will get the longest possible prefix as one of these two values. We call these a and b. By taking the $\max(a, b)$, we can guarantee that we have the LCP which matches our query.

Then after, Max does some clever stuff using the fact that we constrain i and j to be monotonically increasing to add elements to a binary tree to make sure we aren't doing extra calculations than what we need to do.

6 Make equivalent to Process Entropy

In ProcessEntropy, 'slices' are defined using the vector relative_pos. We find that in ProcessEntropy, relative_pos[x] = i

which gives us that everything in source[:x] occured before target[i:].

To make this equivalent to the `FastEntropy` implementation, we take: `l_t = list of tuples with (x,i)`

The following function makes the two equivalent:

```python
def timeseries_cross_entropy_FE_entropy(time_tweets_target, time_tweets_source):

    '''
    ** FAST ENTROPY ***

    Modified version of the timeseries_cross_entropy function (removed please sanitize), to make equivalent between the original and the FastEntropy implementation.

    Function expects two inputs in the form of a list of time-tweet tuples:

    [(time, [list of integers]),....]
    '''

    decorated_target = [ (time,"target",tweet) for time,tweet in time_tweets_target ]
    decorated_source = [ (time,"source",tweet) for time,tweet in time_tweets_source ]

    # Join time series:
    time_tweets = decorated_target + decorated_source

    # Sort in place by time:
    time_tweets.sort()

    # Loop over combined tweets and build word vectors and target->source relative_pos:
    target, source, relative_pos = [], [], []
    for time,user,tweet in time_tweets:
        words = tweet
        if user == "target":
            target.extend(words)
            relative_pos.extend( [len(source)]*len(words) )
        else:
            source.extend(words)

    target = np.array(target, dtype = np.uint32)
    source = np.array(source, dtype = np.uint32)
    relative_pos = np.array(relative_pos, dtype = np.uint32)

    # set up objects
    source = lcs.Vector1D([int(x) for x in ([np.floor(x) for x in source])])
    target = lcs.Vector1D([int(x) for x in ([np.floor(x) for x in target])])

    ob = lcs.LCSFinder(target,source) # s1 and then s2

    l_t =  lcs.Vector2D(tuple((i,int(relative_pos[i])) for i in range(len(relative_pos))))

    fastentropy_lambdas = np.array([x+1 for x in ob.ComputeAllLCSs(l_t)])

    return fastentropy_lambdas
```

We can pretty quickly see the speed up. For small sequence lengths, `ProcessEntropy` is slightly faster, but once N gets large, the improved complexity wins.

7 Testing

Max's first version of the code, which has been wrapped in the LCSFinder code performs the same as the modified (debugged) Process Entropy.

This is a good example of a write up: https://jgaa.info/accepted/2022/588.pdf

The code in 'LCS.py' This code functions the same as lcs_finder2.cpp (Sent by Max on 8th April 2022) This isn't the same as process entropy as it allows the sequence in the source to overlap the starting index of the target sequence.

There currently exists 1. working version of the C++ code which seems to be causing very intermittent kernel crashes (no overlap) 2. making a version in python (to see if crashes are due to SWIG) (which allows overlap)

Got a working version in python (LCS) then tried to use Cython to speed things up These are located in the RewriteinPython Folder.

There is an implementation which only uses LISTS (no numpy arrays) and which has cython types (LIST_typed)

These sped up the code but it is still slower than the C++ code.

Then I went about checking the speed (see in TestingLCSFindervsCythonvsOriginal)

and found 1. Results don't match between cython implementation and the PE / LCSFinder (Max's code)

I then started checking code against lcs_finder2.cpp which was the original version of code sent by Max on the 8th of April 2022. This matches the python implementation (Cython) but doesn't match the PE and LCS implementations.

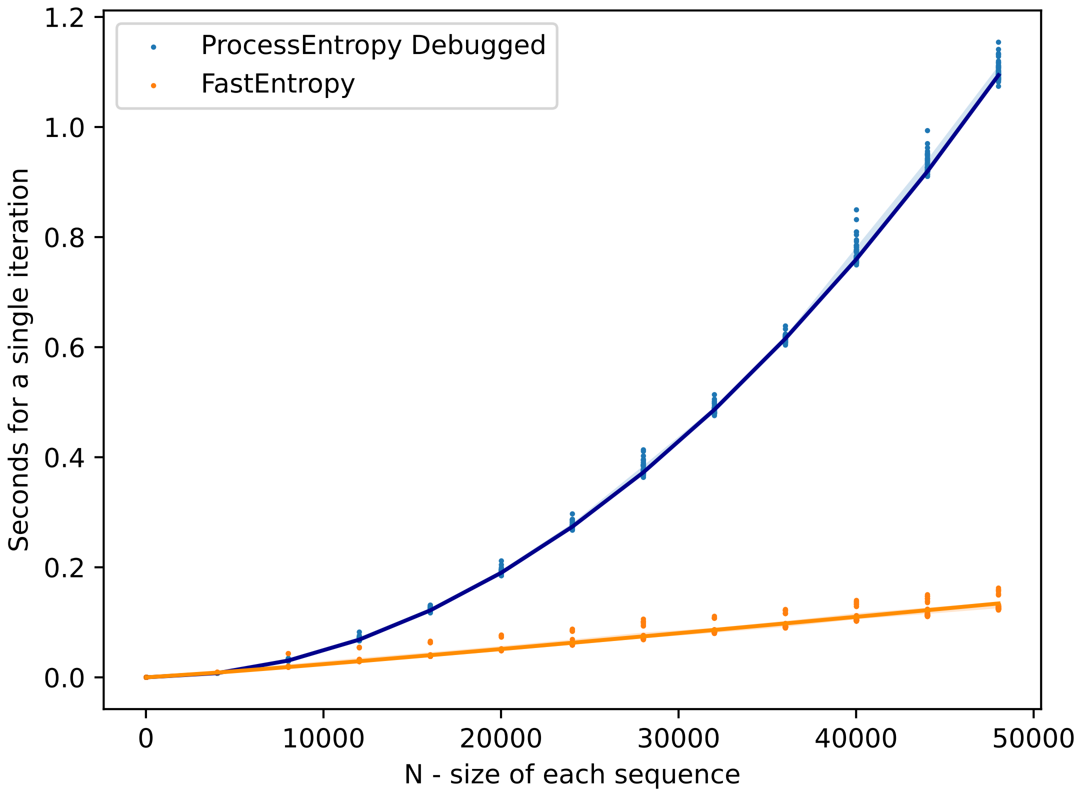

Figure 3: Against debugged code

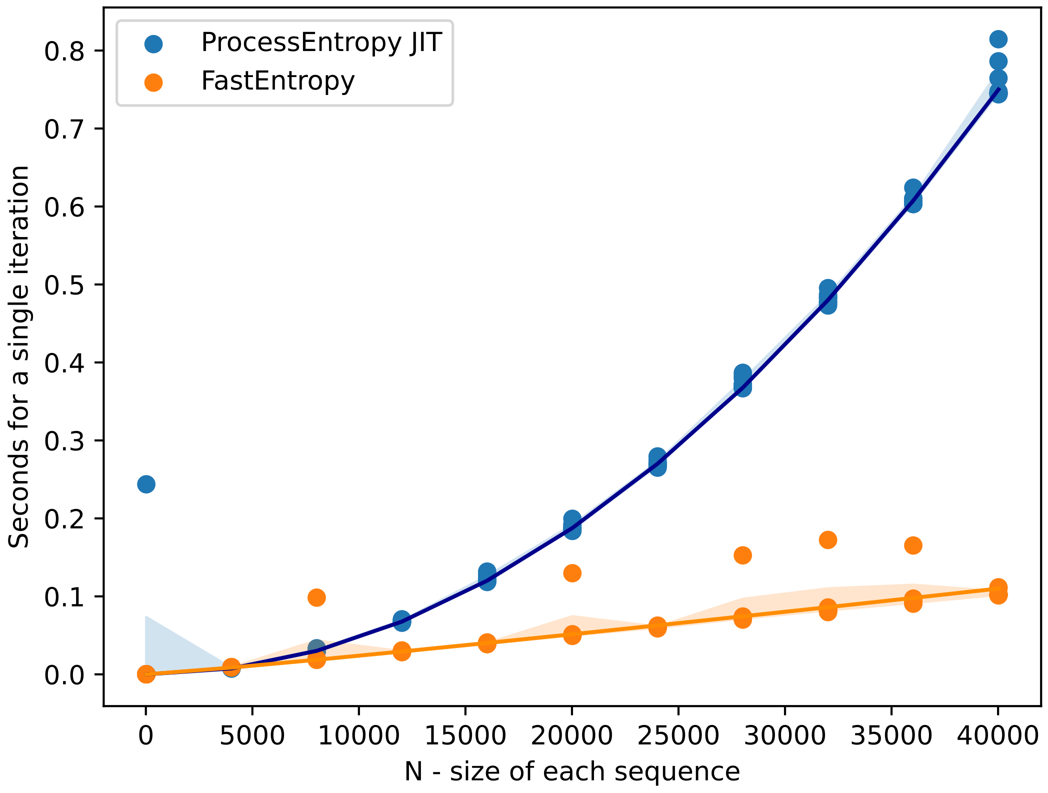

Figure 4: Against JIT code

Found a minimal broken example

s1 = np.array([1, 1, 3, 0, 4, 1, 4, 4, 2, 1]) s2 = np.array([0, 1, 0, 1, 3, 2, 1, 4, 0, 0])

See details in TestingLCSFindervsCythonvsOriginal > testing.ipynb

The true result should give lambda = length of longest sub sequence

The goal for S and T is to find the longest match length. i.e. we find $\max L_n + 1$. L_n is smallest integer l such that the target sequence T_0^{l-1} does not start in the past of the source S_{-n}^{-1}.

$$T_0^{l-1} \neq S_{-m}^{-m+l-1},$$

From max's comments, the LCS finder code gives: The corresponding entry in the return list is the length of the longest string that starts in the range s2[0..j) and that matches a prefix of the string s1[i..n). Note that the right end point of the matching substring in s2 may go up to or past j.

So T = s1, S = s2.

inds = [(0,1),(1,2),(2,3),...,(8,9)]

So, for (3,4), we want to find the length of match between s2[0,4) and s1(3,...)

ind 0 1 2 3 s2 : [0, 1, 0, 1)

ind 3, ... s1 : [0, 4, 1, 4, 4, 2, 1)

Longest match has length 1.

Process Entropy gives: 2 (lambda = 3)

For the FastEnt LCS

So we are comparing source 0 to n, [1 1 3 0 4 1 4 4 2 1] is compared to target to 1, [0] source 1 to n, [1 3 0 4 1 4 4 2 1] is compared to target to 2, [0 1] source 2 to n, [3 0 4 1 4 4 2 1] is compared to target to 3, [0 1 0] source 3 to n, [0 4 1 4 4 2 1] is compared to target to 4, [0 1 0 1] source 4 to n, [4 1 4 4 2 1] is compared to target to 5, [0 1 0 1 3] source 5 to n, [1 4 4 2 1] is compared to target to 6, [0 1 0 1 3 2] source 6 to n, [4 4 2 1] is compared to target to 7, [0 1 0 1 3 2 1] source 7 to n, [4 2 1] is compared to target to 8, [0 1 0 1 3 2 1 4] source 8 to n, [2 1] is compared to target to 9, [0 1 0 1 3 2 1 4 0] source 9 to n, [1] is compared to target to 10, [0 1 0 1 3 2 1 4 0 0] match lengths of (0, 1, 0, 2, 1, 0, 1, 1, 1, 1)

Output does match both ProcessEntropy and FastEnt.

This is because we have the '2' in index 7

source 7 to n, [4 2 1] is compared to target to 8, [0 1 0 1 3 2 1 4]

It does not match lcs_finder2.cpp which gives (0, 1, 0, 2, 1, 0, 2, 1, 1, 1) and LCSV2 which gives array([1, 2, 1, 3, 2, 1, 3, 2, 2, 2]) (and LCS)

This code isn't constraining the target to not overlap. For the given example: With source = [1, 1, 3, 0, 4, 1, 4, 4, 2, 1] target = [0, 1, 0, 1, 3, 2, 1, 4, 0, 0]

looking at matches from target [i:] with anything starting in source[:i]

should be

1,... compared with .., 1, 0, 1, 3, 2, 1, 4, 0, 0 match of 1

1,1,... compared with .., 0, 1, 3, 2, 1, 4, 0, 0 match of 0

1,1,3,... compared with ...,1,3,2,1,4,0,0 match of 2

1,1,3,0,... compared with ...,3, 2, 1, 4, 0, 0 match of 1

1,1,3,0,4,... compared with ..., 2, 1, 4, 0, 0 match of 0

1,1,3,0,4,1... compared with ..., 1, 4, 0, 0 match of 1

1,1,3,0,4,1,4... compared with ..., 4, 0, 0 match of 1

1,1,3,0,4,1,4,4... compared with ..., 0, 0 match of 1

1,1,3,0,4,1,4,4,2... compared with ..., 0 match of 1

giving (0),1,0,2,1,0,1,1,1,1 lambdas -> 1,2,1,3,2,1,2,2,2,2

*** WITH OVERLAP *** i.e. matching target to anything in source stating before i

1,(1, 3, 0, 4, 1, 4, 4, 2, 1)... compared with .., 1, 0, 1, 3, 2, 1, 4, 0, 0 match of 1

1,1,(3, 0, 4, 1, 4, 4, 2, 1)... compared with .., 0, 1, 3, 2, 1, 4, 0, 0 match of 0

1,1,3,(0, 4, 1, 4, 4, 2, 1)... compared with ...,1,3,2,1,4,0,0 match of 2

1,1,3,0,(4, 1, 4, 4, 2, 1)... compared with ...,3, 2, 1, 4, 0, 0 match of 1

1,1,3,0,4,(1, 4, 4, 2, 1)... compared with ..., 2, 1, 4, 0, 0 match of 0

1,1,3,0,4,1,(4, 4, 2, 1)... compared with ..., 1, 4, 0, 0 match of 2 ***** DIFFERENT

1,1,3,0,4,1,4,(4,2,1)... compared with ..., 4, 0, 0 match of 1

1,1,3,0,4,1,4,4,(2,1)... compared with ..., 0, 0 match of 1

1,1,3,0,4,1,4,4,2,(1)... compared with ..., 0 match of 1

giving (0),1,0,2,1,0,2,1,1,1 lambdas -> 1,2,1,3,2,1,3,2,2,2

Ultimately, this seems to be a 'sense' making choice.

BUT - within the functions we only let things overlap to an index, not a time. So for time based applications we need to remove the overlap.

8 Included Code

Testing ProcessEntropy vs FastEnt This folder includes all code to reproduce the results in this notebook. It performs speed and similarity testing between the LCSFinder package and the Process Entropy package.

Cython Versions This folder includes all working cython versions. The version LCS_Cython is the oldest version (code doesn't work properly. LCS_Cython_LIST_typed contains a version of LCS_Cython which doesn't use any numpy arrays and implements Cython typing for a speed boost. LCSV2 contains a version which matches lcs_finder2.cpp (allows overlaps).

LCS.py contains the working python implementation which matches lcs_finder2.cpp.

LCSlistonly.py contains the version of python using only lists, but it is an old version (matches LCS_Cython_LIST)

OriginalCCodeWithComments Contains the old version of Max's code which has explanatory comments (used to write python version)

processentfunctions.py contains process entropy functions to allow for local testing and removing the bugs.

testingCythonVersions performs a speed test on the three Cython versions.

TestingProcessEntLCSFinderCython compares how the three approaches perform.

OriginalLCSCCode contains lcs_finder2.cpp the newer version of Max's code which allows overlap.

References

Kontoyiannis, I., Algoet, P., Suhov, Y. & Wyner, A. (1998), 'Nonparametric entropy estimation for stationary processes and random fields, with applications to English text', *IEEE Transactions on Information Theory* **44**(3), 1319–1327. Conference Name: IEEE Transactions on Information Theory.

Ornstein, D. & Weiss, B. (1993), 'Entropy and data compression schemes', *IEEE Transactions on Information Theory* **39**(1), 78–83. Conference Name: IEEE Transactions on Information Theory.

Wyner, A. & Ziv, J. (1989), 'Some asymptotic properties of the entropy of a stationary ergodic data source with applications to data compression', *IEEE Transactions on Information Theory* **35**(6), 1250–1258. Conference Name: IEEE Transactions on Information Theory.