stumpy Documentation

Release 1.7.2

Sean M. Law

Jan 20, 2021

Contents:

CHAPTER 1

Background & Motivation

The following video provides the background and motivation for developing and open sourcing STUMPY:

CHAPTER 2

Installation

Supported Python and NumPy versions are determined according to the NEP 29 deprecation policy.

2.1 Using conda/pip

Conda install (preferred):

```
conda install -c conda-forge stumpy
```

PyPI install with `pip`:

```
python -m pip install stumpy
```

2.2 From source

To install stumpy from source, you'll need to install the dependencies above. For maximum performance, it is recommended that you install all dependencies using *conda*:

```
conda install -c conda-forge -y numpy scipy numba
```

Alternatively, but with lower performance, you can also install these dependencies using the requirements.txt file in the root of this repository:

```
python -m pip install -r requirements.txt
```

Once the dependencies are installed (stay inside of the `stumpy` directory), execute:

```
python -m pip install .
```

CHAPTER 3

STUMPY API

Overview

stumpy.stump	Compute the z-normalized matrix profile
stumpy.stumped	Compute the z-normalized matrix profile with a distributed dask cluster
stumpy.gpu_stump	Compute the z-normalized matrix profile with one or more GPU devices
stumpy.scrump	Compute an approximate z-normalized matrix profile
stumpy.stumpi	Compute an incremental z-normalized matrix profile for streaming data.
stumpy.mstump	Compute the multi-dimensional z-normalized matrix profile
stumpy.mstumped	Compute the multi-dimensional z-normalized matrix profile with a distributed dask cluster
stumpy.subspace	Compute the k-dimensional matrix profile subspace for a given subsequence index and its nearest neighbor index
stumpy.aamp	Compute the non-normalized (i.e., without z-normalization) matrix profile
stumpy.aamped	Compute the non-normalized (i.e., without z-normalization) matrix profile
stumpy.gpu_aamp	Compute the non-normalized (i.e., without z-normalization) matrix profile with one or more GPU devices
stumpy.aampi	Compute an incremental non-normalized (i.e., without z-normalization) matrix profile for streaming data
stumpy.atsc	Compute the anchored time series chain (ATSC)
stumpy.allc	Compute the all-chain set (ALLC)

Continued on next page

Table 1 – continued from previous page

stumpy.fluss	Compute the Fast Low-cost Unipotent Semantic Segmentation (FLUSS) for static data (i.e., batch processing)
stumpy.floss	Compute the Fast Low-cost Online Semantic Segmentation (FLOSS) for streaming data
stumpy.ostinato	Find the z-normalized consensus motif of multiple time series
stumpy.ostinatoed	Find the z-normalized consensus motif of multiple time series with a distributed dask cluster
stumpy.gpu_ostinato	Find the z-normalized consensus motif of multiple time series with one or more GPU devices
stumpy.aamp_ostinato	Find the non-normalized (i.e., without z-normalization) consensus motif of multiple time series
stumpy.aamp_ostinatoed	Find the non-normalized (i.e., without z-normalization) consensus motif of multiple time series with a distributed dask cluster
stumpy.gpu_aamp_ostinato	Find the non-normalized (i.e., without z-normalization) consensus motif of multiple time series with one or more GPU devices
stumpy.mpdist	Compute the z-normalized matrix profile distance (MPdist) measure between any two time series
stumpy.mpdisted	Compute the z-normalized matrix profile distance (MPdist) measure between any two time series with a distributed dask cluster
stumpy.gpu_mpdist	Compute the z-normalized matrix profile distance (MPdist) measure between any two time series with one or more GPU devices
stumpy.aampdist	Compute the non-normalized (i.e., without z-normalization) matrix profile distance (MPdist) measure between any two time series with *stumpy.aamp*.
stumpy.aampdisted	Compute the non-normalized (i.e., without z-normalization) matrix profile distance (MPdist) measure between any two time series with a distributed dask cluster and *stumpy.aamped*.
stumpy.gpu_aampdist	Compute the non-normalized (i.e., without z-normalization) matrix profile distance (MPdist) measure between any two time series with one or more GPU devices and *stumpy.gpu_aamp*.

3.1 stump

stumpy.**stump**(*T_A*, *m*, *T_B=None*, *ignore_trivial=True*)

Compute the z-normalized matrix profile

This is a convenience wrapper around the Numba JIT-compiled parallelized *_stump* function which computes the matrix profile according to STOMPopt with Pearson correlations.

Parameters

- **T_A** (*ndarray*) – The time series or sequence for which to compute the matrix profile
- **m** (*int*) – Window size
- **T_B** (*ndarray, default None*) – The time series or sequence that will be used to

annotate T_A. For every subsequence in T_A, its nearest neighbor in T_B will be recorded. Default is *None* which corresponds to a self-join.

- **ignore_trivial** (*bool, default True*) – Set to *True* if this is a self-join. Otherwise, for AB-join, set this to *False*. Default is *True*.

Returns **out** – The first column consists of the matrix profile, the second column consists of the matrix profile indices, the third column consists of the left matrix profile indices, and the fourth column consists of the right matrix profile indices.

Return type ndarray

Notes

DOI: 10.1007/s10115-017-1138-x

See Section 4.5

The above reference outlines a general approach for traversing the distance matrix in a diagonal fashion rather than in a row-wise fashion.

DOI: 10.1145/3357223.3362721

See Section 3.1 and Section 3.3

The above reference outlines the use of the Pearson correlation via Welford's centered sum-of-products along each diagonal of the distance matrix in place of the sliding window dot product found in the original STOMP method.

DOI: 10.1109/ICDM.2016.0085

See Table II

Timeseries, T_A, will be annotated with the distance location (or index) of all its subsequences in another times series, T_B.

Return: For every subsequence, Q, in T_A, you will get a distance and index for the closest subsequence in T_B. Thus, the array returned will have length T_A.shape[0]-m+1. Additionally, the left and right matrix profiles are also returned.

Note: Unlike in the Table II where T_A.shape is expected to be equal to T_B.shape, this implementation is generalized so that the shapes of T_A and T_B can be different. In the case where T_A.shape == T_B.shape, then our algorithm reduces down to the same algorithm found in Table II.

Additionally, unlike STAMP where the exclusion zone is m/2, the default exclusion zone for STOMP is m/4 (See Definition 3 and Figure 3).

For self-joins, set *ignore_trivial* = *True* in order to avoid the trivial match.

Note that left and right matrix profiles are only available for self-joins.

3.2 stumped

stumpy.**stumped**(*dask_client*, *T_A*, *m*, *T_B=None*, *ignore_trivial=True*)

Compute the z-normalized matrix profile with a distributed dask cluster

This is a highly distributed implementation around the Numba JIT-compiled parallelized *_stump* function which computes the matrix profile according to STOMPopt with Pearson correlations.

Parameters

- **dask_client** (*client*) – A Dask Distributed client that is connected to a Dask scheduler and Dask workers. Setting up a Dask distributed cluster is beyond the scope of this library. Please refer to the Dask Distributed documentation.
- **T_A** (*ndarray*) – The time series or sequence for which to compute the matrix profile
- **m** (*int*) – Window size
- **T_B** (*ndarray, default None*) – The time series or sequence that will be used to annotate T_A. For every subsequence in T_A, its nearest neighbor in T_B will be recorded. Default is *None* which corresponds to a self-join.
- **ignore_trivial** (*bool, default True*) – Set to *True* if this is a self-join. Otherwise, for AB-join, set this to *False*. Default is *True*.

Returns **out** – The first column consists of the matrix profile, the second column consists of the matrix profile indices, the third column consists of the left matrix profile indices, and the fourth column consists of the right matrix profile indices.

Return type ndarray

Notes

DOI: 10.1007/s10115-017-1138-x

See Section 4.5

The above reference outlines a general approach for traversing the distance matrix in a diagonal fashion rather than in a row-wise fashion.

DOI: 10.1145/3357223.3362721

See Section 3.1 and Section 3.3

The above reference outlines the use of the Pearson correlation via Welford's centered sum-of-products along each diagonal of the distance matrix in place of the sliding window dot product found in the original STOMP method.

DOI: 10.1109/ICDM.2016.0085

See Table II

This is a Dask distributed implementation of stump that scales across multiple servers and is a convenience wrapper around the parallelized *stump._stump* function

Timeseries, T_A, will be annotated with the distance location (or index) of all its subsequences in another times series, T_B.

Return: For every subsequence, Q, in T_A, you will get a distance and index for the closest subsequence in T_B. Thus, the array returned will have length T_A.shape[0]-m+1. Additionally, the left and right matrix profiles are also returned.

Note: Unlike in the Table II where T_A.shape is expected to be equal to T_B.shape, this implementation is generalized so that the shapes of T_A and T_B can be different. In the case where T_A.shape == T_B.shape, then our algorithm reduces down to the same algorithm found in Table II.

Additionally, unlike STAMP where the exclusion zone is m/2, the default exclusion zone for STOMP is m/4 (See Definition 3 and Figure 3).

For self-joins, set *ignore_trivial* = *True* in order to avoid the trivial match.

Note that left and right matrix profiles are only available for self-joins.

3.3 gpu-stump

stumpy.**gpu_stump**(*T_A*, *m*, *T_B=None*, *ignore_trivial=True*, *device_id=0*)

Compute the z-normalized matrix profile with one or more GPU devices

This is a convenience wrapper around the Numba *cuda.jit _gpu_stump* function which computes the matrix profile according to GPU-STOMP.

Parameters

- **T_A** (*ndarray*) – The time series or sequence for which to compute the matrix profile
- **m** (*int*) – Window size
- **T_B** (*ndarray, default None*) – The time series or sequence that will be used to annotate T_A. For every subsequence in T_A, its nearest neighbor in T_B will be recorded. Default is *None* which corresponds to a self-join.
- **ignore_trivial** (*bool, default True*) – Set to *True* if this is a self-join. Otherwise, for AB-join, set this to *False*. Default is *True*.
- **device_id** (*int or list, default 0*) – The (GPU) device number to use. The default value is *0*. A list of valid device ids (int) may also be provided for parallel GPU-STUMP computation. A list of all valid device ids can be obtained by executing *[device.id for device in numba.cuda.list_devices()]*.

Returns **out** – The first column consists of the matrix profile, the second column consists of the matrix profile indices, the third column consists of the left matrix profile indices, and the fourth column consists of the right matrix profile indices.

Return type ndarray

Notes

DOI: 10.1109/ICDM.2016.0085

See Table II, Figure 5, and Figure 6

Timeseries, T_A, will be annotated with the distance location (or index) of all its subsequences in another times series, T_B.

Return: For every subsequence, Q, in T_A, you will get a distance and index for the closest subsequence in T_B. Thus, the array returned will have length T_A.shape[0]-m+1. Additionally, the left and right matrix profiles are also returned.

Note: Unlike in the Table II where T_A.shape is expected to be equal to T_B.shape, this implementation is generalized so that the shapes of T_A and T_B can be different. In the case where T_A.shape == T_B.shape, then our algorithm reduces down to the same algorithm found in Table II.

Additionally, unlike STAMP where the exclusion zone is m/2, the default exclusion zone for STOMP is m/4 (See Definition 3 and Figure 3).

For self-joins, set *ignore_trivial* = *True* in order to avoid the trivial match.

Note that left and right matrix profiles are only available for self-joins.

3.4 scrump

stumpy.**scrump**(*T_A*, *m*, *T_B=None*, *ignore_trivial=True*, *percentage=0.01*, *pre_scrump=False*, *s=None*)

Compute an approximate z-normalized matrix profile

This is a convenience wrapper around the Numba JIT-compiled parallelized *_stump* function which computes the matrix profile according to SCRIMP.

Parameters

- **T_A** (*ndarray*) – The time series or sequence for which to compute the matrix profile
- **T_B** (*ndarray*) – The time series or sequence that will be used to annotate T_A. For every subsequence in T_A, its nearest neighbor in T_B will be recorded.
- **m** (*int*) – Window size
- **ignore_trivial** (*bool*) – Set to *True* if this is a self-join. Otherwise, for AB-join, set this to *False*. Default is *True*.
- **percentage** (*float*) – Approximate percentage completed. The value is between 0.0 and 1.0.
- **pre_scrump** (*bool*) – A flag for whether or not to perform the PreSCRIMP calculation prior to computing SCRIMP. If set to *True*, this is equivalent to computing SCRIMP++ and may lead to faster convergence
- **s** (*int*) – The size of the PreSCRIMP fixed interval. If *pre-scrump=True* and *s=None*, then *s* will automatically be set to *s=int(np.ceil(m/4))*, the size of the exclusion zone.

stumpy.**P_**

The updated matrix profile

Type ndarray

stumpy.**I_**

The updated matrix profile indices

Type ndarray

stumpy.**update**()

Update the matrix profile and the matrix profile indices by computing additional new distances (limited by *percentage*) that make up the full distance matrix.

Notes

DOI: 10.1109/ICDM.2018.00099

See Algorithm 1 and Algorithm 2

3.5 stumpi

stumpy.**stumpi**(*T*, *m*, *excl_zone=None*, *egress=True*)

Compute an incremental z-normalized matrix profile for streaming data. This is based on the on-line STOMPI and STAMPI algorithms.

Parameters

- **T** (*ndarray*) – The time series or sequence for which the matrix profile and matrix profile indices will be returned
- **m** (*int*) – Window size
- **excl_zone** (*int, default None*) – The half width for the exclusion zone relative to the current sliding window
- **egress** (*bool, default True*) – If set to *True*, the oldest data point in the time series is removed and the time series length remains constant rather than forever increasing

stumpy.**P_**
The updated matrix profile for *T*

Type ndarray

stumpy.**I_**
The updated matrix profile indices for *T*

Type ndarray

stumpy.**left_P_**
The updated left matrix profile for *T*

Type ndarray

stumpy.**left_I_**
The updated left matrix profile indices for *T*

Type ndarray

stumpy.**T_**
The updated time series or sequence for which the matrix profile and matrix profile indices are computed

Type ndarray

stumpy.**update**(*t*)
Append a single new data point, *t*, to the time series, *T*, and update the matrix profile

Notes

DOI: 10.1007/s10618-017-0519-9

See Table V

Note that line 11 is missing an important *sqrt* operation!

3.6 mstump

stumpy.**mstump**(*T*, *m*, *include=None*, *discords=False*)
Compute the multi-dimensional z-normalized matrix profile

This is a convenience wrapper around the Numba JIT-compiled parallelized *_mstump* function which computes the multi-dimensional matrix profile and multi-dimensional matrix profile index according to mSTOMP, a variant of mSTAMP. Note that only self-joins are supported.

Parameters

- **T** (*ndarray*) – The time series or sequence for which to compute the multi-dimensional matrix profile. Each row in *T* represents data from a different dimension while each column in *T* represents data from the same dimension.

- **m** (*int*) – Window size
- **include** (*list, ndarray, default None*) – A list of (zero-based) indices corresponding to the dimensions in *T* that must be included in the constrained multidimensional motif search. For more information, see Section IV D in:

  DOI: 10.1109/ICDM.2017.66
- **discords** (*bool, default False*) – When set to *True*, this reverses the distance matrix which results in a multi-dimensional matrix profile that favors larger matrix profile values (i.e., discords) rather than smaller values (i.e., motifs). Note that indices in *include* are still maintained and respected.

Returns

- **P** (*ndarray*) – The multi-dimensional matrix profile. Each row of the array corresponds to each matrix profile for a given dimension (i.e., the first row is the 1-D matrix profile and the second row is the 2-D matrix profile).
- **I** (*ndarray*) – The multi-dimensional matrix profile index where each row of the array corresponds to each matrix profile index for a given dimension.

Notes

DOI: 10.1109/ICDM.2017.66

See mSTAMP Algorithm

3.7 mstumped

stumpy.**mstumped**(*dask_client*, *T*, *m*, *include=None*, *discords=False*)

Compute the multi-dimensional z-normalized matrix profile with a distributed dask cluster

This is a highly distributed implementation around the Numba JIT-compiled parallelized *_mstump* function which computes the multi-dimensional matrix profile according to STOMP. Note that only self-joins are supported.

Parameters

- **dask_client** (*client*) – A Dask Distributed client that is connected to a Dask scheduler and Dask workers. Setting up a Dask distributed cluster is beyond the scope of this library. Please refer to the Dask Distributed documentation.
- **T** (*ndarray*) – The time series or sequence for which to compute the multi-dimensional matrix profile. Each row in *T* represents data from a different dimension while each column in *T* represents data from the same dimension.
- **m** (*int*) – Window size
- **include** (*list, ndarray, default None*) – A list of (zero-based) indices corresponding to the dimensions in *T* that must be included in the constrained multidimensional motif search. For more information, see Section IV D in:

  DOI: 10.1109/ICDM.2017.66
- **discords** (*bool, default False*) – When set to *True*, this reverses the distance matrix which results in a multi-dimensional matrix profile that favors larger matrix profile values (i.e., discords) rather than smaller values (i.e., motifs). Note that indices in *include* are still maintained and respected.

Returns

- **P** (*ndarray*) – The multi-dimensional matrix profile. Each row of the array corresponds to each matrix profile for a given dimension (i.e., the first row is the 1-D matrix profile and the second row is the 2-D matrix profile).
- **I** (*ndarray*) – The multi-dimensional matrix profile index where each row of the array corresponds to each matrix profile index for a given dimension.

Notes

DOI: 10.1109/ICDM.2017.66

See mSTAMP Algorithm

3.8 subspace

stumpy.**subspace**(*T*, *m*, *subseq_idx*, *nn_idx*, *k*, *include=None*, *discords=False*)

Compute the k-dimensional matrix profile subspace for a given subsequence index and its nearest neighbor index

Parameters

- **T** (*ndarray*) – The time series or sequence for which the multi-dimensional matrix profile, multi-dimensional matrix profile indices were computed
- **m** (*int*) – Window size
- **subseq_idx** (*int*) – The subsequence index in T
- **nn_idx** (*int*) – The nearest neighbor index in T
- **k** (*int*) – The subset number of dimensions out of *D = T.shape[0]*-dimensions to return the subspace for
- **include** (*ndarray, default None*) – A list of (zero-based) indices corresponding to the dimensions in *T* that must be included in the constrained multidimensional motif search. For more information, see Section IV D in:

  DOI: 10.1109/ICDM.2017.66
- **discords** (*bool, default False*) – When set to *True*, this reverses the distance profile to favor discords rather than motifs. Note that indices in *include* are still maintained and respected.

Returns

- **S** (*ndarray*)
- An array of that contains the ‘k‘th-dimensional subspace for the subsequence
- with index equal to *motif_idx*

3.9 aamp

stumpy.**aamp**(*T_A*, *m*, *T_B=None*, *ignore_trivial=True*)

Compute the non-normalized (i.e., without z-normalization) matrix profile

This is a convenience wrapper around the Numba JIT-compiled parallelized *_aamp* function which computes the matrix profile according to AAMP.

Parameters

- **T_A** (*ndarray*) – The time series or sequence for which to compute the matrix profile
- **m** (*int*) – Window size
- **T_B** (*ndarray, default None*) – The time series or sequence that will be used to annotate T_A. For every subsequence in T_A, its nearest neighbor in T_B will be recorded. Default is *None* which corresponds to a self-join.
- **ignore_trivial** (*bool, default True*) – Set to *True* if this is a self-join. Otherwise, for AB-join, set this to *False*. Default is *True*.

Returns **out** – The first column consists of the matrix profile, the second column consists of the matrix profile indices.

Return type ndarray

Notes

arXiv:1901.05708

See Algorithm 1

Note that we have extended this algorithm for AB-joins as well.

3.10 aamped

stumpy.**aamped**(*dask_client*, *T_A*, *m*, *T_B=None*, *ignore_trivial=True*)

Compute the non-normalized (i.e., without z-normalization) matrix profile

This is a highly distributed implementation around the Numba JIT-compiled parallelized *_aamp* function which computes the non-normalized matrix profile according to AAMP.

Parameters

- **dask_client** (*client*) – A Dask Distributed client that is connected to a Dask scheduler and Dask workers. Setting up a Dask distributed cluster is beyond the scope of this library. Please refer to the Dask Distributed documentation.
- **T_A** (*ndarray*) – The time series or sequence for which to compute the matrix profile
- **m** (*int*) – Window size
- **T_B** (*ndarray, default None*) – The time series or sequence that will be used to annotate T_A. For every subsequence in T_A, its nearest neighbor in T_B will be recorded. Default is *None* which corresponds to a self-join.
- **ignore_trivial** (*bool, default True*) – Set to *True* if this is a self-join. Otherwise, for AB-join, set this to *False*. Default is *True*.

Returns **out** – The first column consists of the matrix profile, the second column consists of the matrix profile indices.

Return type ndarray

Notes

arXiv:1901.05708

See Algorithm 1

Note that we have extended this algorithm for AB-joins as well.

3.11 gpu_aamp

stumpy.**gpu_aamp**(*T_A*, *m*, *T_B=None*, *ignore_trivial=True*, *device_id=0*)

Compute the non-normalized (i.e., without z-normalization) matrix profile with one or more GPU devices

This is a convenience wrapper around the Numba *cuda.jit _gpu_aamp* function which computes the non-normalized matrix profile according to modified version GPU-STOMP.

Parameters

- **T_A** (*ndarray*) – The time series or sequence for which to compute the matrix profile
- **m** (*int*) – Window size
- **T_B** (*ndarray, default None*) – The time series or sequence that contain your query subsequences of interest. Default is *None* which corresponds to a self-join.
- **ignore_trivial** (*bool, default True*) – Set to *True* if this is a self-join. Otherwise, for AB-join, set this to *False*. Default is *True*.
- **device_id** (*int or list, default 0*) – The (GPU) device number to use. The default value is *0*. A list of valid device ids (int) may also be provided for parallel GPU-STUMP computation. A list of all valid device ids can be obtained by executing *[device.id for device in numba.cuda.list_devices()]*.

Returns **out** – The first column consists of the matrix profile, the second column consists of the matrix profile indices, the third column consists of the left matrix profile indices, and the fourth column consists of the right matrix profile indices.

Return type ndarray

Notes

arXiv:1901.05708

See Algorithm 1

Note that we have extended this algorithm for AB-joins as well.

DOI: 10.1109/ICDM.2016.0085

See Table II, Figure 5, and Figure 6

3.12 aampi

stumpy.**aampi**(*T*, *m*, *excl_zone=None*, *egress=True*)

Compute an incremental non-normalized (i.e., without z-normalization) matrix profile for streaming data

Parameters

- **T** (*ndarray*) – The time series or sequence for which the non-normalized matrix profile and matrix profile indices will be returned
- **m** (*int*) – Window size
- **excl_zone** (*int, default None*) – The half width for the exclusion zone relative to the current sliding window
- **egress** (*bool, default True*) – If set to *True*, the oldest data point in the time series is removed and the time series length remains constant rather than forever increasing

stumpy.**P_**
The updated matrix profile for *T*

Type ndarray

stumpy.**I_**
The updated matrix profile indices for *T*

Type ndarray

stumpy.**left_P_**
The updated left matrix profile for *T*

Type ndarray

stumpy.**left_I_**
The updated left matrix profile indices for *T*

Type ndarray

stumpy.**T_**
The updated time series or sequence for which the matrix profile and matrix profile indices are computed

Type ndarray

stumpy.**update**(*t*)
Append a single new data point, *t*, to the time series, *T*, and update the matrix profile

Notes

arXiv:1901.05708

See Algorithm 1

Note that we have extended this algorithm for AB-joins as well.

3.13 atsc

stumpy.**atsc**(*IL*, *IR*, *j*)
Compute the anchored time series chain (ATSC)

Parameters

- **IL** (*ndarray*) – Left matrix profile indices
- **IR** (*ndarray*) – Right matrix profile indices
- **j** (*int*) – The index value for which to compute the ATSC

Returns **output** – Anchored time series chain for index, *j*

Return type ndarray

Notes

DOI: 10.1109/ICDM.2017.79

See Table I

This is the implementation for the anchored time series chains (ATSC).

Unlike the original paper, we've replaced the while-loop with a more stable for-loop.

3.14 allc

stumpy.**allc**(*IL*, *IR*)

Compute the all-chain set (ALLC)

Parameters

- **IL** (*ndarray*) – Left matrix profile indices
- **IR** (*ndarray*) – Right matrix profile indices

Returns

- **S** (*list(ndarray)*) – All-chain set
- **C** (*ndarray*) – Anchored time series chain for the longest chain (also known as the unanchored chain)

Notes

DOI: 10.1109/ICDM.2017.79

See Table II

Unlike the original paper, we've replaced the while-loop with a more stable for-loop.

This is the implementation for the all-chain set (ALLC) and the unanchored chain is simply the longest one among the all-chain set. Both the all-chain set and unanchored chain are returned.

The all-chain set, S, is returned as a list of unique numpy arrays.

3.15 fluss

stumpy.**fluss**(*I*, *L*, *n_regimes*, *excl_factor=5*, *custom_iac=None*)

Compute the Fast Low-cost Unipotent Semantic Segmentation (FLUSS) for static data (i.e., batch processing)

Essentially, this is a wrapper to compute the corrected arc curve and regime locations.

Parameters

- **I** (*ndarray*) – The matrix profile indices for the time series of interest
- **L** (*int*) – The subsequence length that is set roughly to be one period length. This is likely to be the same value as the window size, *m*, used to compute the matrix profile and matrix profile index but it can be different since this is only used to manage edge effects and has no bearing on any of the IAC or CAC core calculations.
- **n_regimes** (*int*) – The number of regimes to search for. This is one more than the number of regime changes as denoted in the original paper.

- **m** (*int*) – The subsequence length. This is expected to be the same value as the window size used to compute the matrix profile and matrix profile index.
- **excl_factor** (*int, default 5*) – The multiplying factor for the regime exclusion zone
- **custom_iac** (*ndarray, default None*) – A custom idealized arc curve (IAC) that will used for correcting the arc curve

Returns

- **cac** (*ndarray*) – A corrected arc curve (CAC)
- **regime_locs** (*ndarray*) – The locations of the regimes

Notes

DOI: 10.1109/ICDM.2017.21

See Section A

This is the implementation for Fast Low-cost Unipotent Semantic Segmentation (FLUSS).

3.16 floss

stumpy.**floss**(*mp*, *T*, *m*, *L*, *excl_factor=5*, *n_iter=1000*, *n_samples=1000*, *custom_iac=None*)

Compute the Fast Low-cost Online Semantic Segmentation (FLOSS) for streaming data

Parameters

- **mp** (*ndarray*) – The first column consists of the matrix profile, the second column consists of the matrix profile indices, the third column consists of the left matrix profile indices, and the fourth column consists of the right matrix profile indices.
- **T** (*ndarray*) – A 1-D time series data used to generate the matrix profile and matrix profile indices found in *mp*. Note that the the right matrix profile index is used and the right matrix profile is intelligently recomputed on the fly from *T* instead of using the bidirectional matrix profile.
- **m** (*int*) – The window size for computing sliding window mass. This is identical to the window size used in the matrix profile calculation. For managing edge effects, see the *L* parameter.
- **L** (*int*) – The subsequence length that is set roughly to be one period length. This is likely to be the same value as the window size, *m*, used to compute the matrix profile and matrix profile index but it can be different since this is only used to manage edge effects and has no bearing on any of the IAC or CAC core calculations.
- **excl_factor** (*int, default 5*) – The multiplying factor for the regime exclusion zone. Note that this is unrelated to the *excl_zone* used in to compute the matrix profile.
- **n_iter** (*int, default 1000*) – Number of iterations to average over when determining the parameters for the IAC beta distribution
- **n_samples** (*int, default 1000*) – Number of distribution samples to draw during each iteration when computing the IAC
- **custom_iac** (*ndarray, default None*) – A custom idealized arc curve (IAC) that will used for correcting the arc curve

stumpy.**cac_1d_**
A 1-dimensional corrected arc curve (CAC) updated as a result of ingressing a single new data point and egressing a single old data point.

Type ndarray

stumpy.**P_**
The matrix profile updated as a result of ingressing a single new data point and egressing a single old data point.

Type ndarray

stumpy.**I_**
The (right) matrix profile indices updated as a result of ingressing a single new data point and egressing a single old data point.

Type ndarray

stumpy.**T_**
The updated time series, *T*

Type ndarray

stumpy.**update**(*t*)
Ingress a new data point, *t*, onto the time series, *T*, followed by egressing the oldest single data point from *T*. Then, update the 1-dimensional corrected arc curve (CAC_1D) and the matrix profile.

Notes

DOI: 10.1109/ICDM.2017.21 <https://www.cs.ucr.edu/~eamonn/Segmentation_ICDM.pdf>'__

See Section C

This is the implementation for Fast Low-cost Online Semantic Segmentation (FLOSS).

3.17 ostinato

stumpy.**ostinato**(*Ts*, *m*)
Find the z-normalized consensus motif of multiple time series

This is a wrapper around the vanilla version of the ostinato algorithm which finds the best radius and a helper function that finds the most central conserved motif.

Parameters

- **Ts** (*list*) – A list of time series for which to find the most central consensus motif
- **m** (*int*) – Window size

Returns

- **central_radius** (*float*) – Radius of the most central consensus motif
- **central_Ts_idx** (*int*) – The time series index in *Ts* which contains the most central consensus motif
- **central_subseq_idx** (*int*) – The subsequence index within time series *Ts[central_motif_Ts_idx]* the contains most central consensus motif

Notes

DOI: 10.1109/ICDM.2019.00140

See Table 2

The ostinato algorithm proposed in the paper finds the best radius in *Ts*. Intuitively, the radius is the minimum distance of a subsequence to encompass at least one nearest neighbor subsequence from all other time series. The best radius in *Ts* is the minimum radius amongst all radii. Some data sets might contain multiple subsequences which have the same optimal radius. The greedy Ostinato algorithm only finds one of them, which might not be the most central motif. The most central motif amongst the subsequences with the best radius is the one with the smallest mean distance to nearest neighbors in all other time series. To find this central motif it is necessary to search the subsequences with the best radius via *stumpy.ostinato._get_central_motif*

3.18 ostinatoed

stumpy.**ostinatoed**(*dask_client*, *Ts*, *m*)

Find the z-normalized consensus motif of multiple time series with a distributed dask cluster

This is a wrapper around the vanilla version of the ostinato algorithm which finds the best radius and a helper function that finds the most central conserved motif.

Parameters

- **dask_client** (*client*) – A Dask Distributed client that is connected to a Dask scheduler and Dask workers. Setting up a Dask distributed cluster is beyond the scope of this library. Please refer to the Dask Distributed documentation.
- **Ts** (*list*) – A list of time series for which to find the most central consensus motif
- **m** (*int*) – Window size

Returns

- **central_radius** (*float*) – Radius of the most central consensus motif
- **central_Ts_idx** (*int*) – The time series index in *Ts* which contains the most central consensus motif
- **central_subseq_idx** (*int*) – The subsequence index within time series *Ts[central_motif_Ts_idx]* the contains most central consensus motif

Notes

DOI: 10.1109/ICDM.2019.00140

See Table 2

The ostinato algorithm proposed in the paper finds the best radius in *Ts*. Intuitively, the radius is the minimum distance of a subsequence to encompass at least one nearest neighbor subsequence from all other time series. The best radius in *Ts* is the minimum radius amongst all radii. Some data sets might contain multiple subsequences which have the same optimal radius. The greedy Ostinato algorithm only finds one of them, which might not be the most central motif. The most central motif amongst the subsequences with the best radius is the one with the smallest mean distance to nearest neighbors in all other time series. To find this central motif it is necessary to search the subsequences with the best radius via *stumpy.ostinato._get_central_motif*

3.19 gpu_ostinato

3.20 aamp_ostinato

stumpy.**aamp_ostinato**(*Ts*, *m*)

Find the non-normalized (i.e., without z-normalization) consensus motif of multiple time series

This is a wrapper around the vanilla version of the ostinato algorithm which finds the best radius and a helper function that finds the most central conserved motif.

Parameters

- **Ts** (*list*) – A list of time series for which to find the most central consensus motif
- **m** (*int*) – Window size

Returns

- **central_radius** (*float*) – Radius of the most central consensus motif
- **central_Ts_idx** (*int*) – The time series index in *Ts* which contains the most central consensus motif
- **central_subseq_idx** (*int*) – The subsequence index within time series *Ts[central_motif_Ts_idx]* the contains most central consensus motif

Notes

DOI: 10.1109/ICDM.2019.00140

See Table 2

The ostinato algorithm proposed in the paper finds the best radius in *Ts*. Intuitively, the radius is the minimum distance of a subsequence to encompass at least one nearest neighbor subsequence from all other time series. The best radius in *Ts* is the minimum radius amongst all radii. Some data sets might contain multiple subsequences which have the same optimal radius. The greedy Ostinato algorithm only finds one of them, which might not be the most central motif. The most central motif amongst the subsequences with the best radius is the one with the smallest mean distance to nearest neighbors in all other time series. To find this central motif it is necessary to search the subsequences with the best radius via *stumpy.ostinato._get_central_motif*

3.21 aamp_ostinatoed

stumpy.**aamp_ostinatoed**(*dask_client*, *Ts*, *m*)

Find the non-normalized (i.e., without z-normalization) consensus motif of multiple time series with a distributed dask cluster

This is a wrapper around the vanilla version of the ostinato algorithm which finds the best radius and a helper function that finds the most central conserved motif.

Parameters

- **dask_client** (*client*) – A Dask Distributed client that is connected to a Dask scheduler and Dask workers. Setting up a Dask distributed cluster is beyond the scope of this library. Please refer to the Dask Distributed documentation.
- **Ts** (*list*) – A list of time series for which to find the most central consensus motif
- **m** (*int*) – Window size

Returns

- **central_radius** (*float*) – Radius of the most central consensus motif
- **central_Ts_idx** (*int*) – The time series index in *Ts* which contains the most central consensus motif
- **central_subseq_idx** (*int*) – The subsequence index within time series *Ts[central_motif_Ts_idx]* the contains most central consensus motif

Notes

DOI: 10.1109/ICDM.2019.00140

See Table 2

The ostinato algorithm proposed in the paper finds the best radius in *Ts*. Intuitively, the radius is the minimum distance of a subsequence to encompass at least one nearest neighbor subsequence from all other time series. The best radius in *Ts* is the minimum radius amongst all radii. Some data sets might contain multiple subsequences which have the same optimal radius. The greedy Ostinato algorithm only finds one of them, which might not be the most central motif. The most central motif amongst the subsequences with the best radius is the one with the smallest mean distance to nearest neighbors in all other time series. To find this central motif it is necessary to search the subsequences with the best radius via *stumpy.ostinato._get_central_motif*

3.22 gpu_aamp_ostinato

`stumpy.`**`gpu_aamp_ostinato`**(*Ts*, *m*, *device_id=0*)

Find the non-normalized (i.e., without z-normalization) consensus motif of multiple time series with one or more GPU devices

This is a wrapper around the vanilla version of the ostinato algorithm which finds the best radius and a helper function that finds the most central conserved motif.

Parameters

- **`Ts`** (*`list`*) – A list of time series for which to find the most central consensus motif
- **`m`** (*`int`*) – Window size
- **`device_id`** (*`int or list, default 0`*) – The (GPU) device number to use. The default value is *0*. A list of valid device ids (int) may also be provided for parallel GPU-STUMP computation. A list of all valid device ids can be obtained by executing *[device.id for device in numba.cuda.list_devices()]*.

Returns

- **central_radius** (*float*) – Radius of the most central consensus motif
- **central_Ts_idx** (*int*) – The time series index in *Ts* which contains the most central consensus motif
- **central_subseq_idx** (*int*) – The subsequence index within time series *Ts[central_motif_Ts_idx]* the contains most central consensus motif

Notes

DOI: 10.1109/ICDM.2019.00140

See Table 2

The ostinato algorithm proposed in the paper finds the best radius in *Ts*. Intuitively, the radius is the minimum distance of a subsequence to encompass at least one nearest neighbor subsequence from all other time series. The best radius in *Ts* is the minimum radius amongst all radii. Some data sets might contain multiple subsequences which have the same optimal radius. The greedy Ostinato algorithm only finds one of them, which might not be the most central motif. The most central motif amongst the subsequences with the best radius is the one with the smallest mean distance to nearest neighbors in all other time series. To find this central motif it is necessary to search the subsequences with the best radius via *stumpy.ostinato._get_central_motif*

3.23 mpdist

stumpy.**mpdist**(*T_A*, *T_B*, *m*, *percentage=0.05*, *k=None*)

Compute the z-normalized matrix profile distance (MPdist) measure between any two time series

The MPdist distance measure considers two time series to be similar if they share many subsequences, regardless of the order of matching subsequences. MPdist concatenates and sorts the output of an AB-join and a BA-join and returns the value of the ‘k‘th smallest number as the reported distance. Note that MPdist is a measure and not a metric. Therefore, it does not obey the triangular inequality but the method is highly scalable.

Parameters

- **T_A** (*ndarray*) – The first time series or sequence for which to compute the matrix profile
- **T_B** (*ndarray*) – The second time series or sequence for which to compute the matrix profile
- **m** (*int*) – Window size
- **percentage** (*float, default 0.05*) – The percentage of distances that will be used to report *mpdist*. The value is between 0.0 and 1.0.

Returns **MPdist** – The matrix profile distance

Return type float

Notes

DOI: 10.1109/ICDM.2018.00119

See Section III

3.24 mpdisted

stumpy.**mpdisted**(*dask_client*, *T_A*, *T_B*, *m*, *percentage=0.05*, *k=None*)

Compute the z-normalized matrix profile distance (MPdist) measure between any two time series with a distributed dask cluster

The MPdist distance measure considers two time series to be similar if they share many subsequences, regardless of the order of matching subsequences. MPdist concatenates and sorts the output of an AB-join and a BA-join and returns the value of the ‘k‘th smallest number as the reported distance. Note that MPdist is a measure and not a metric. Therefore, it does not obey the triangular inequality but the method is highly scalable.

Parameters

- **dask_client** (*client*) – A Dask Distributed client that is connected to a Dask scheduler and Dask workers. Setting up a Dask distributed cluster is beyond the scope of this library. Please refer to the Dask Distributed documentation.
- **T_A** (*ndarray*) – The first time series or sequence for which to compute the matrix profile
- **T_B** (*ndarray*) – The second time series or sequence for which to compute the matrix profile
- **m** (*int*) – Window size
- **percentage** (*float, default 0.05*) – The percentage of distances that will be used to report *mpdist*. The value is between 0.0 and 1.0. This parameter is ignored when *k* is not *None*.
- **k** (*int*) – Specify the *k'th value in the concatenated matrix profiles to return. When 'k* is not *None*, then the *percentage* parameter is ignored.

Returns **MPdist** – The matrix profile distance

Return type float

Notes

DOI: 10.1109/ICDM.2018.00119

See Section III

3.25 gpu_mpdist

stumpy.**gpu_mpdist**(*T_A*, *T_B*, *m*, *percentage=0.05*, *k=None*, *device_id=0*)

Compute the z-normalized matrix profile distance (MPdist) measure between any two time series with one or more GPU devices

The MPdist distance measure considers two time series to be similar if they share many subsequences, regardless of the order of matching subsequences. MPdist concatenates and sorts the output of an AB-join and a BA-join and returns the value of the 'k'th smallest number as the reported distance. Note that MPdist is a measure and not a metric. Therefore, it does not obey the triangular inequality but the method is highly scalable.

Parameters

- **T_A** (*ndarray*) – The first time series or sequence for which to compute the matrix profile
- **T_B** (*ndarray*) – The second time series or sequence for which to compute the matrix profile
- **m** (*int*) – Window size
- **percentage** (*float, default 0.05*) – The percentage of distances that will be used to report *mpdist*. The value is between 0.0 and 1.0. This parameter is ignored when *k* is not *None*.
- **k** (*int, default None*) – Specify the *k'th value in the concatenated matrix profiles to return. When 'k* is not *None*, then the *percentage* parameter is ignored.
- **device_id** (*int or list, default 0*) – The (GPU) device number to use. The default value is *0*. A list of valid device ids (int) may also be provided for parallel GPU-STUMP computation. A list of all valid device ids can be obtained by executing *[device.id for device in numba.cuda.list_devices()]*.

Returns **MPdist** – The matrix profile distance

Return type float

Notes

DOI: 10.1109/ICDM.2018.00119

See Section III

3.26 aampdist

stumpy.**aampdist**(*T_A*, *T_B*, *m*, *percentage=0.05*, *k=None*)

Compute the non-normalized (i.e., without z-normalization) matrix profile distance (MPdist) measure between any two time series with *stumpy.aamp*.

The MPdist distance measure considers two time series to be similar if they share many subsequences, regardless of the order of matching subsequences. MPdist concatenates and sorts the output of an AB-join and a BA-join and returns the value of the ‘k‘th smallest number as the reported distance. Note that MPdist is a measure and not a metric. Therefore, it does not obey the triangular inequality but the method is highly scalable.

Parameters

- **T_A** (*ndarray*) – The first time series or sequence for which to compute the matrix profile
- **T_B** (*ndarray*) – The second time series or sequence for which to compute the matrix profile
- **m** (*int*) – Window size
- **percentage** (*float, default 0.05*) – The percentage of distances that will be used to report *mpdist*. The value is between 0.0 and 1.0.

Returns **MPdist** – The matrix profile distance

Return type float

Notes

DOI: 10.1109/ICDM.2018.00119

See Section III

3.27 aampdisted

stumpy.**aampdisted**(*dask_client*, *T_A*, *T_B*, *m*, *percentage=0.05*, *k=None*)

Compute the non-normalized (i.e., without z-normalization) matrix profile distance (MPdist) measure between any two time series with a distributed dask cluster and *stumpy.aamped*.

The MPdist distance measure considers two time series to be similar if they share many subsequences, regardless of the order of matching subsequences. MPdist concatenates and sorts the output of an AB-join and a BA-join and returns the value of the ‘k‘th smallest number as the reported distance. Note that MPdist is a measure and not a metric. Therefore, it does not obey the triangular inequality but the method is highly scalable.

Parameters

- **dask_client** (*client*) – A Dask Distributed client that is connected to a Dask scheduler and Dask workers. Setting up a Dask distributed cluster is beyond the scope of this library. Please refer to the Dask Distributed documentation.
- **T_A** (*ndarray*) – The first time series or sequence for which to compute the matrix profile
- **T_B** (*ndarray*) – The second time series or sequence for which to compute the matrix profile
- **m** (*int*) – Window size
- **percentage** (*float, default 0.05*) – The percentage of distances that will be used to report *mpdist*. The value is between 0.0 and 1.0. This parameter is ignored when *k* is not *None*.
- **k** (*int*) – Specify the *k'th value in the concatenated matrix profiles to return. When 'k* is not *None*, then the *percentage* parameter is ignored.

Returns **MPdist** – The matrix profile distance

Return type float

Notes

DOI: 10.1109/ICDM.2018.00119

See Section III

3.28 gpu_aampdist

stumpy.**gpu_aampdist**(*T_A*, *T_B*, *m*, *percentage=0.05*, *k=None*, *device_id=0*)

Compute the non-normalized (i.e., without z-normalization) matrix profile distance (MPdist) measure between any two time series with one or more GPU devices and *stumpy.gpu_aamp*.

The MPdist distance measure considers two time series to be similar if they share many subsequences, regardless of the order of matching subsequences. MPdist concatenates and sorts the output of an AB-join and a BA-join and returns the value of the 'k'th smallest number as the reported distance. Note that MPdist is a measure and not a metric. Therefore, it does not obey the triangular inequality but the method is highly scalable.

Parameters

- **T_A** (*ndarray*) – The first time series or sequence for which to compute the matrix profile
- **T_B** (*ndarray*) – The second time series or sequence for which to compute the matrix profile
- **m** (*int*) – Window size
- **percentage** (*float, default 0.05*) – The percentage of distances that will be used to report *mpdist*. The value is between 0.0 and 1.0. This parameter is ignored when *k* is not *None*.
- **k** (*int, default None*) – Specify the *k'th value in the concatenated matrix profiles to return. When 'k* is not *None*, then the *percentage* parameter is ignored.
- **device_id** (*int or list, default 0*) – The (GPU) device number to use. The default value is *0*. A list of valid device ids (int) may also be provided for parallel GPU-STUMP computation. A list of all valid device ids can be obtained by executing *[device.id for device in numba.cuda.list_devices()]*.

Returns **MPdist** – The matrix profile distance

Return type float

Notes

DOI: 10.1109/ICDM.2018.00119

See Section III

CHAPTER 4

Tutorials

4.1 The Matrix Profile

4.1.1 Laying the Foundation

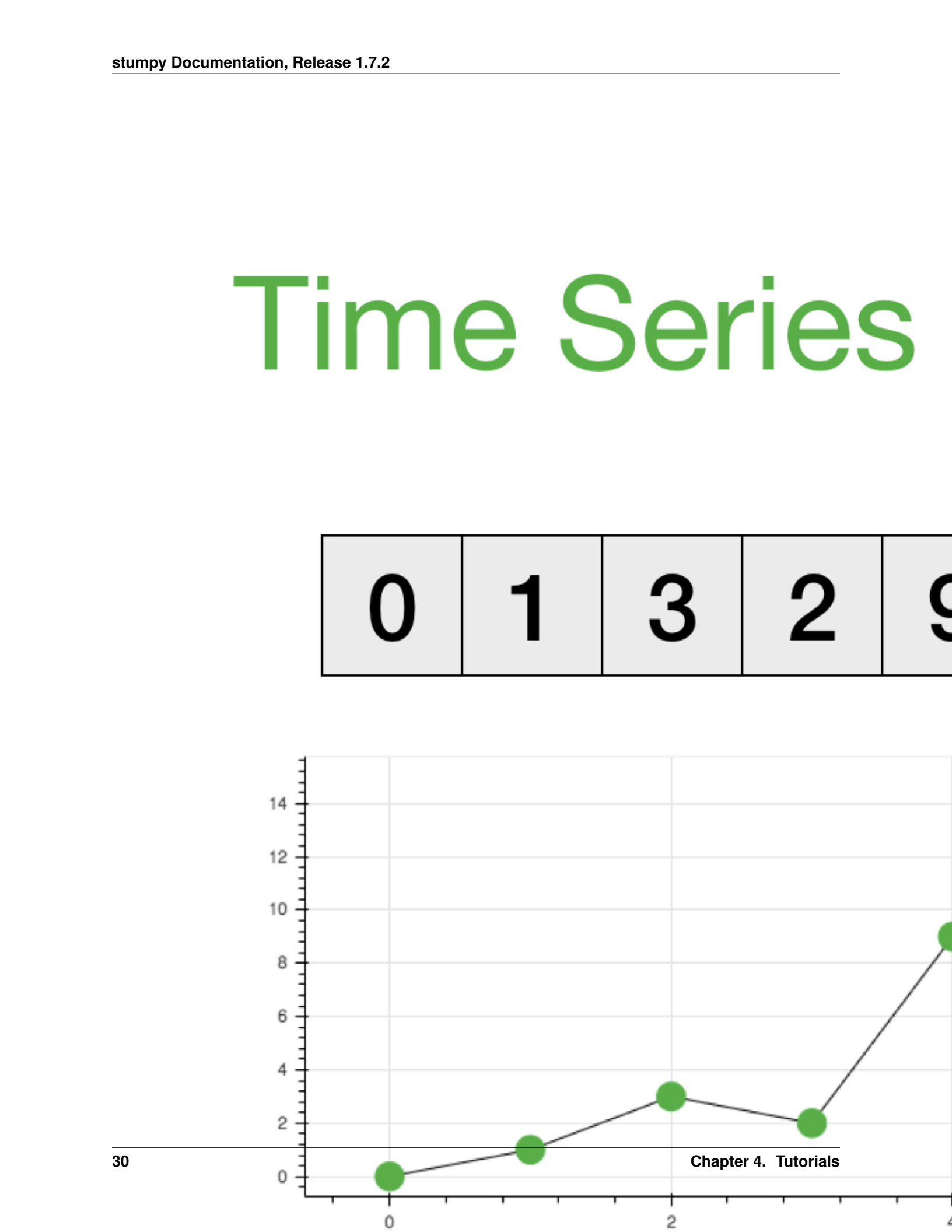
At its core, the STUMPY library efficiently computes something called a matrix profile, a vector that stores the z-normalized Euclidean distance between any subsequence within a time series and its nearest neighbor.

To fully understand what this means, let's take a step back and start with a simple illustrative example along with a few basic definitions:

4.1.2 Time Series with Length n = 13

[1]:
```
time_series = [0, 1, 3, 2, 9, 1, 14, 15, 1, 2, 2, 10, 7]
n = len(time_series)
```

To analyze this time series with length `n = 13`, we could visualize the data or calculate global summary statistics (i.e., mean, median, mode, min, max). If you had a much longer time series, then you may even feel compelled to build an ARIMA model, perform anomaly detection, or attempt a forecasting model but these methods can be complicated and may often have false positives or no interpretable insights.


Time Series
0
1
3
2
14
12
10
8
6
4
2
0
0
2

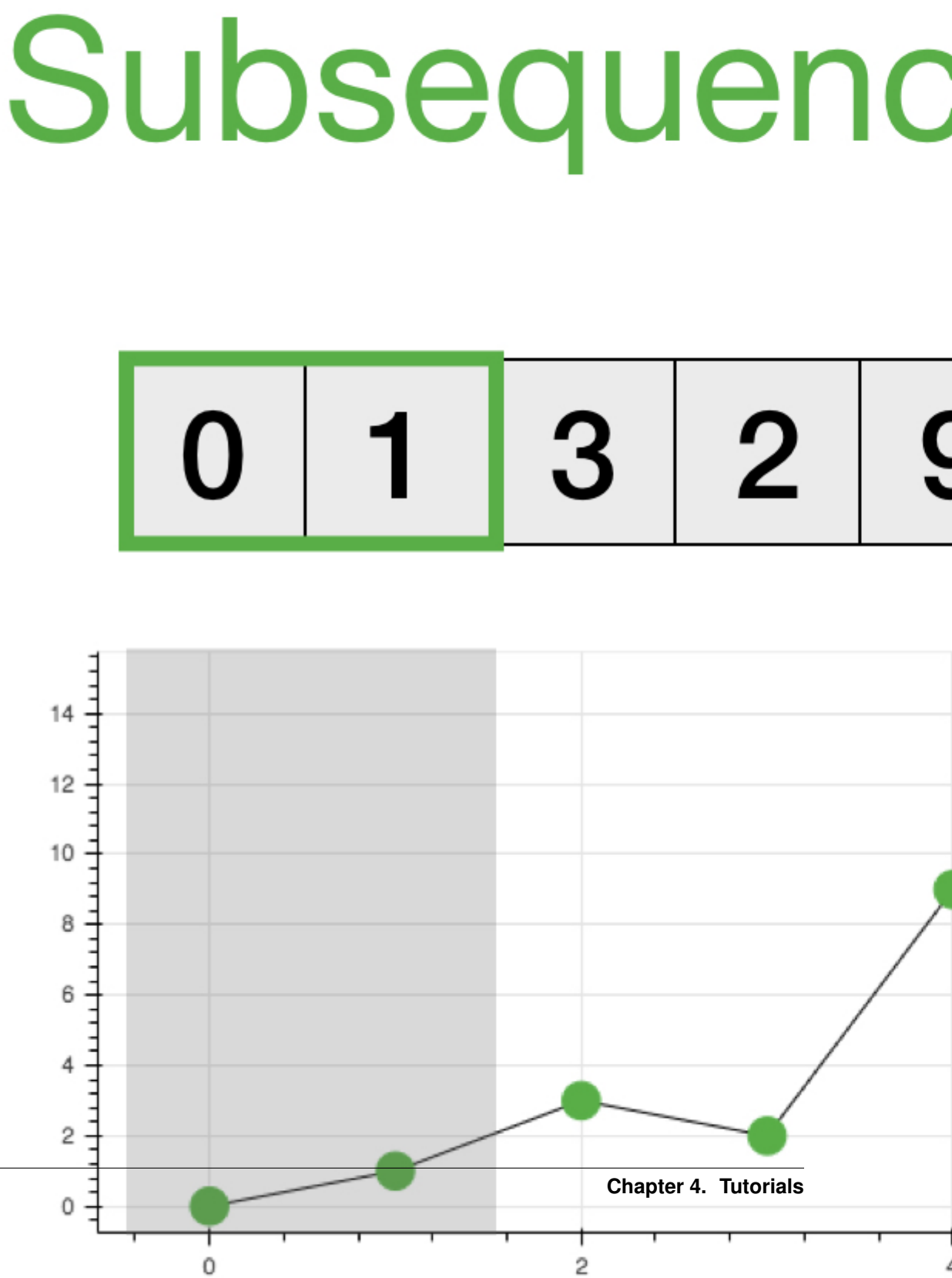
However, if we were to apply Occam's Razor, then what is the most simple and intuitive approach that we could take analyze to this time series?

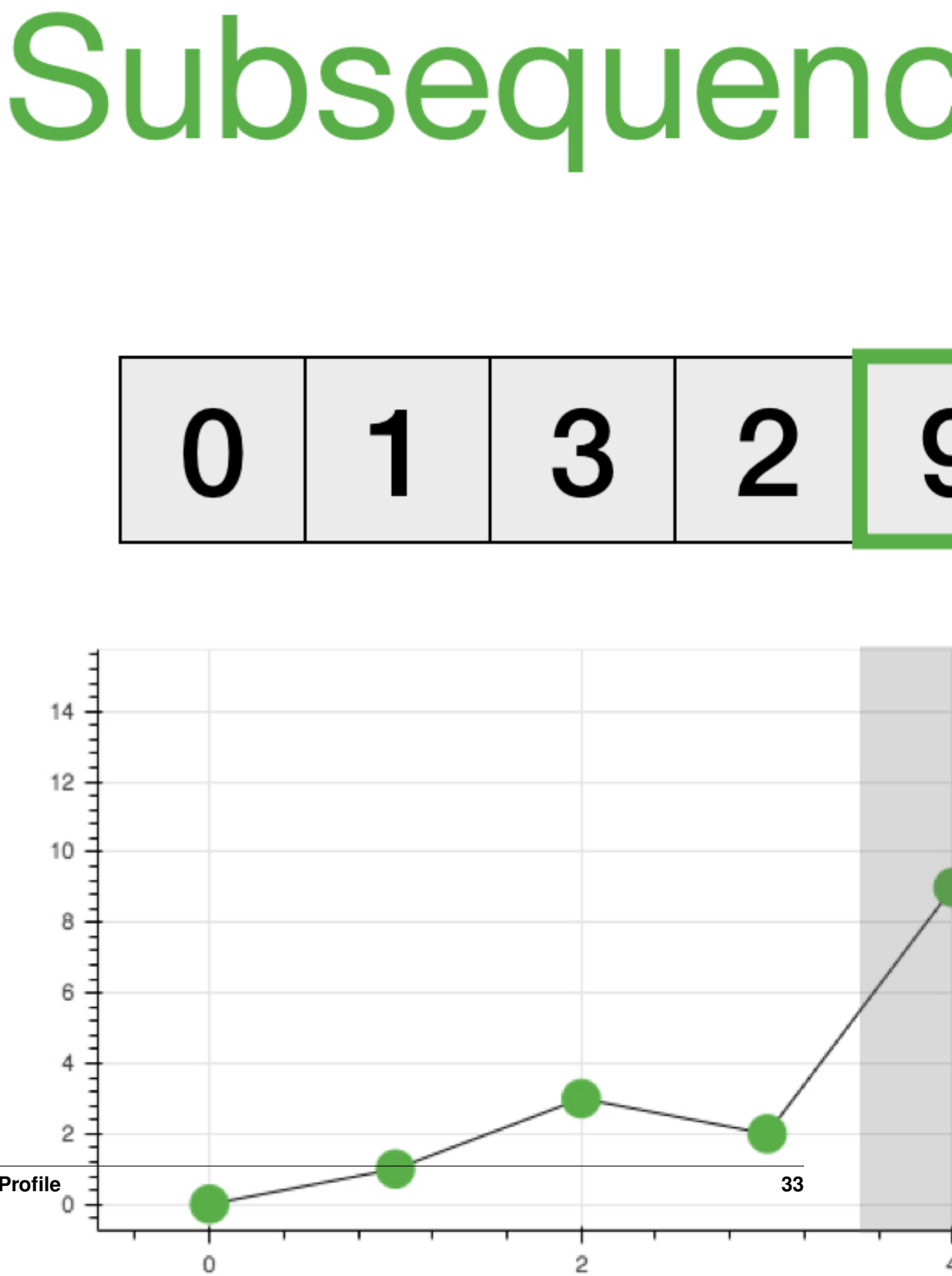
To answer this question, let's start with our first defintion:

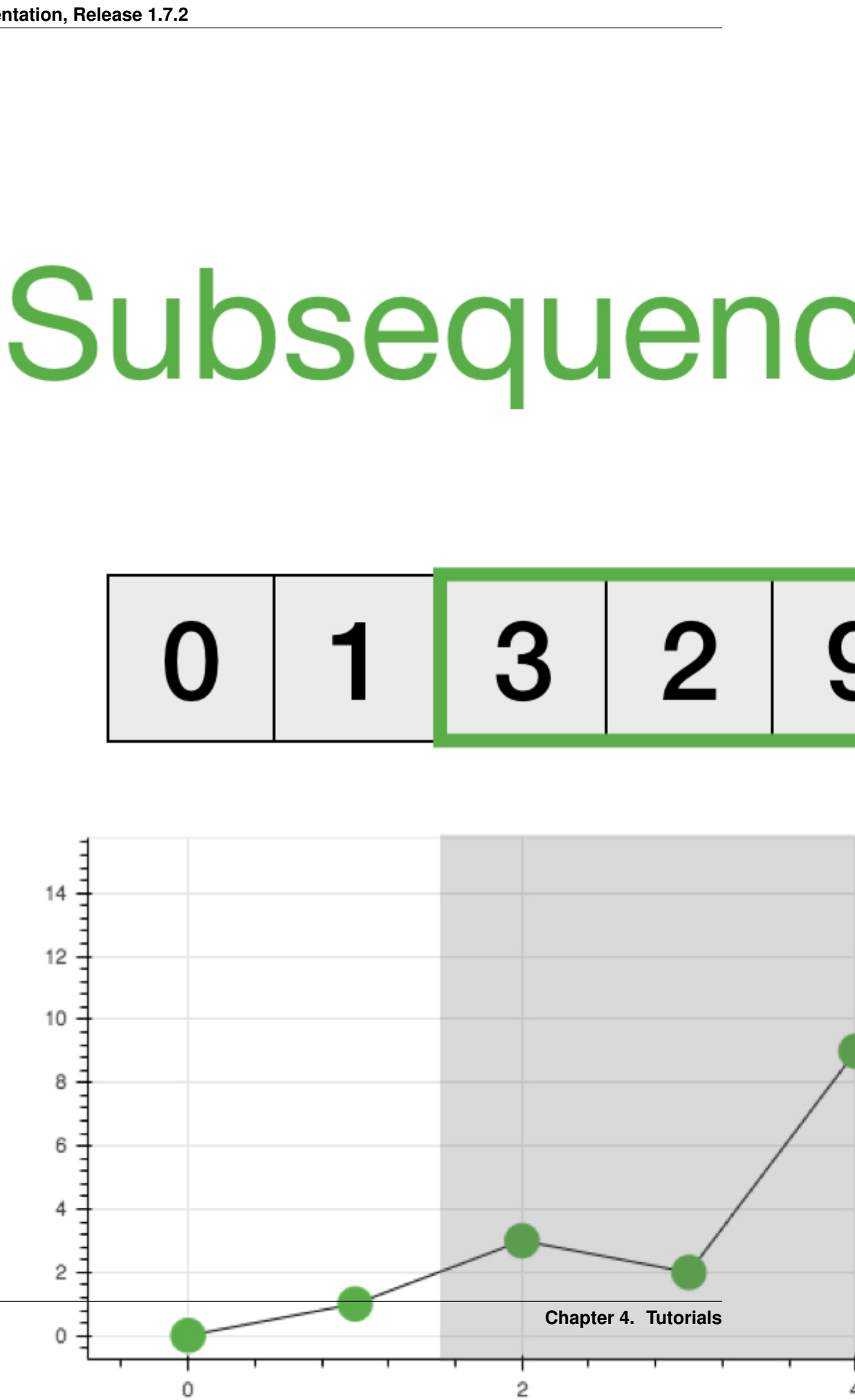
4.1.3 Subsequence /sbskwns/ noun

a part or section of the full time series

So, the following are all considered subsequences of our `time_series` since they can all be found in the time series above.


Subsequenc
0
1
3
2
14
12
10
8
6
4
2
0
0
2


Subsequenc
0
1
3
2
14
12
10
8
6
4
2
0
0
2


Subsequenc
0
1
3
2
14
12
10
8
6
4
2
0
0
2

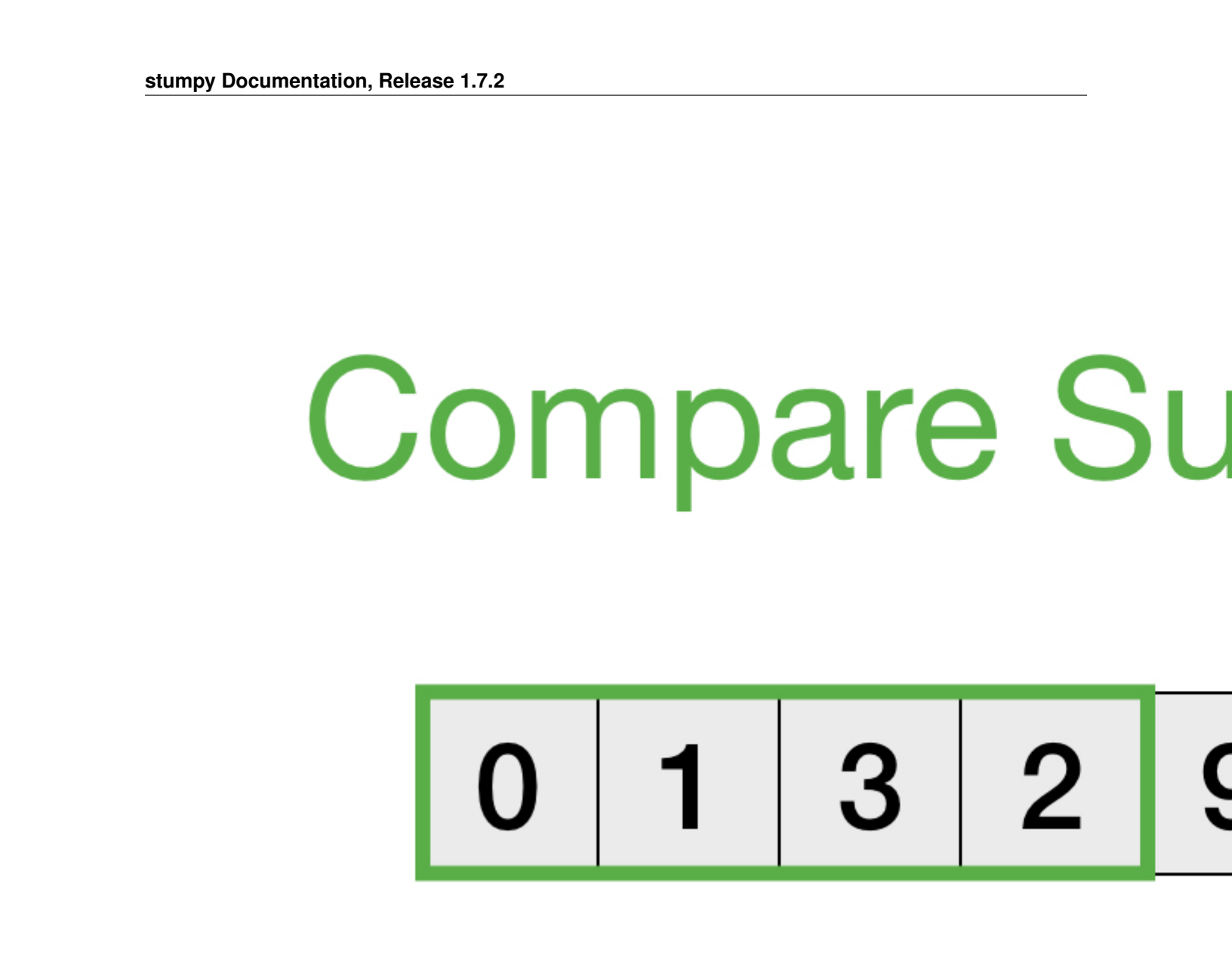
[2]:
```
print(time_series[0:2])
print(time_series[4:7])
print(time_series[2:10])
```

```
[0, 1]
[9, 1, 14]
[3, 2, 9, 1, 14, 15, 1, 2]
```

We can see that each subsequence can have a different sequence length that we'll call `m`. So, for example, if we choose `m = 4`, then we can think about how we might compare any two subsequences of the same length.

[3]:
```
m = 4
i = 0  # starting index for the first subsequence
j = 8  # starting index for the second subsequence

subseq_1 = time_series[i:i+m]
subseq_2 = time_series[j:j+m]

print(subseq_1, subseq_2)
```

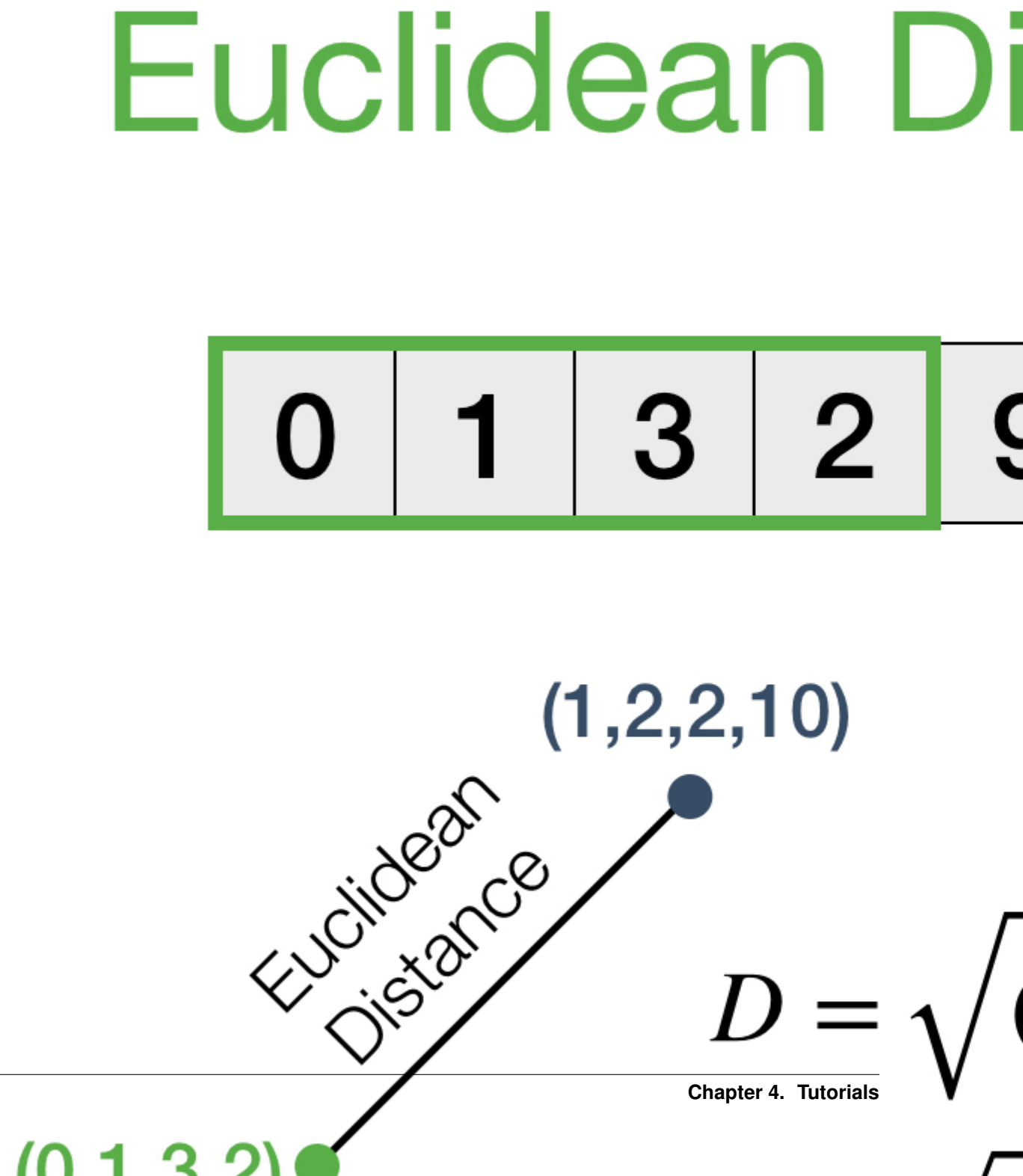
```
[0, 1, 3, 2] [1, 2, 2, 10]
```


Compare Su
0
1
3
2
14
12
10
8
6
4
2
0
0
2

One way to compare any two subsequences is to calculate what is called the Euclidean distance.

4.1.4 Euclidean Distance /yooklidēn/ /distns/ noun

the straight-line distance between two points



[4]:
```
import math

D = 0
for k in range(m):
    D += (time_series[i+k] - time_series[j+k])**2
print(f"The square root of {D} = {math.sqrt(D)}")
```

```
The square root of 67 = 8.18535277187245
```

4.1.5 Distance Profile - Pairwise Euclidean Distances

Now, we can take this a step further where we keep one subsequence the same (reference subsequence), change the second subsequence in a sliding window manner, and compute the Euclidean distance for each window. The resulting vector of pairwise Euclidean distances is also known as a distance profile.

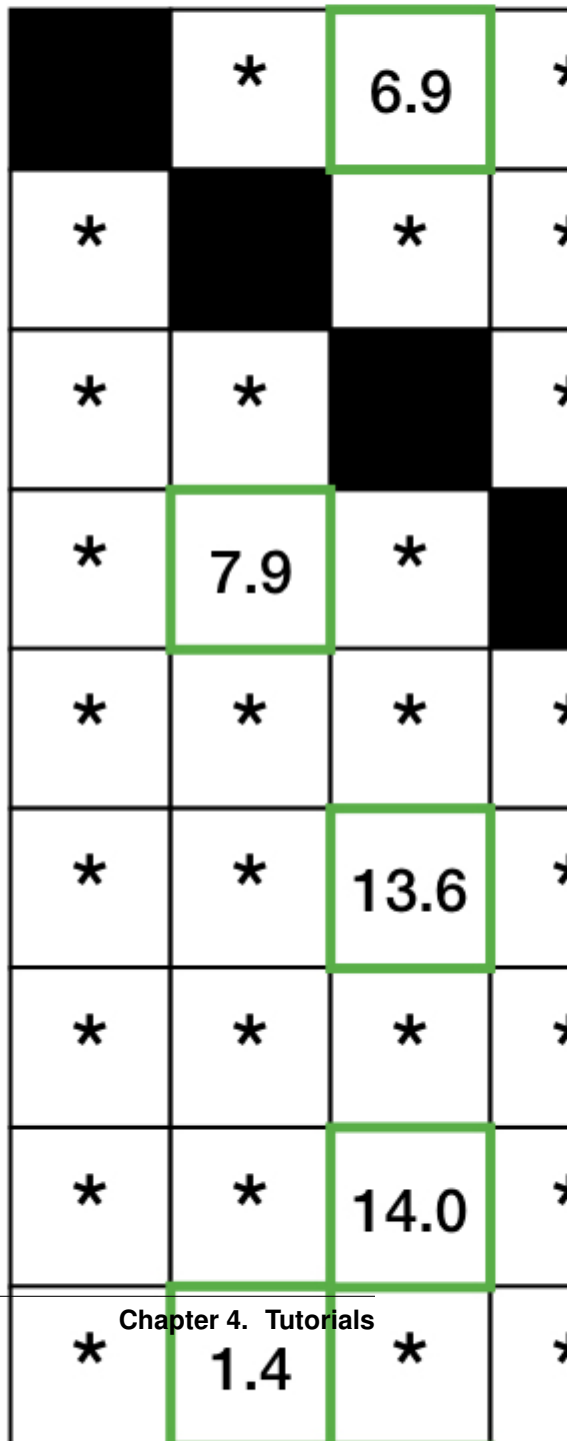
Of course, not all of these distances are useful. Specifically, the distance for the self match (or trivial match) isn't informative since the distance will be always be zero when you are comparing a subsequence with itself. So, we'll ignore it and, instead, take note of the next smallest distance from the distance profile and choose that as our best match:


Pairwise Euc
0 1 3 2
0 1 3 2
7.4 6.9 14.7

Next, we can shift our reference subsequence over one element at a time and repeat the same sliding window process to compute the distance profile for each new reference subsequence.

4.1.6 Distance Matrix

If we take all of the distance profiles that were computed for each reference subsequence and stack them one on top of each other then we get something called a distance matrix


Dis
6.9
7.9
13.6
14.0
1.4

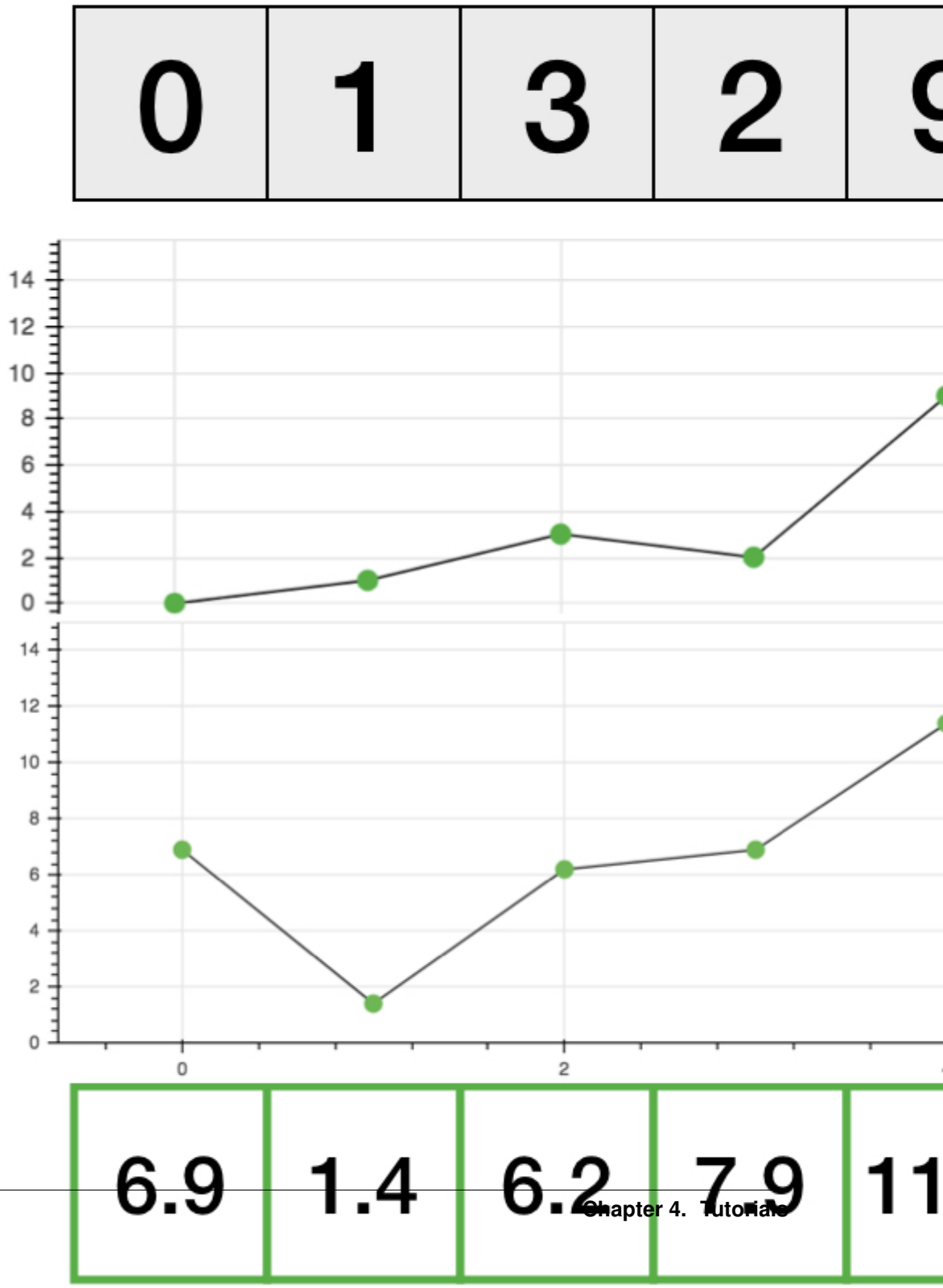
Now, we can simplify this distance matrix by only looking at the nearest neighbor for each subsequence and this takes us to our next concept:

4.1.7 Matrix Profile /mātriks/ /prōfīl/ noun

a vector that stores the (z-normalized) Euclidean distance between any subsequence within a time series and its nearest neighbor

Practically, what this means is that the matrix profile is only interested in storing the smallest non-trivial distances from each distance profile, which significantly reduces the spatial complexity to O(n):

We can now plot this matrix profile underneath our original time series. And, as it turns out, a reference subsequence with a small matrix profile value (i.e., it has a nearest neighbor significantly "closeby") may indicate a possible pattern while a reference subsequence with a large matrix profile value (i.e., its nearest neighbor is significantly "faraway") may suggest the presence of an anomaly.


0
1
3
2
14
12
10
8
6
4
2
0
14
12
10
8
6
4
2
0
0
2
6.9
1.4
6.2
7.9

So, by simply computing and inspecting the matrix profile alone, one can easily pick out the top pattern (global minimum) and rarest anomaly (global maximum). And this is only a small glimpse into what is possible once you've computed the matrix profile!

4.1.8 The Real Problem - The Brute Force Approach

Now, it might seem pretty straightforward at this point but what we need to do is consider how to compute the full distance matrix efficiently. Let's start with the brute force approach:

[5]:
```
for i in range(n-m+1):
    for j in range(n-m+1):
        D = 0
        for k in range(m):
            D += (time_series[i+k] - time_series[j+k])**2
        D = math.sqrt(D)
```

At first glance, this may not look too bad but if we start considering both the computational complexity as well as the spatial complexity then we begin to understand the real problem. It turns out that, for longer time series (i.e., n >> 10,000) the computational complexity is O(n2m) (as evidenced by the three for loops in the code above) and the spatial complexity for storing the full distance matrix is O(n2).

To put this into perspective, imagine if you had a single sensor that collected data 20 times/min over the course of 5 years. This would result:

[6]:
```
n = 20 * 60 * 24 * 364 * 5  # 20 times/min x 60 mins/hour x 24 hours/day x 365 days/
↪year x 5 years
print(f"There would be n = {n} data points")
```

```
There would be n = 52416000 data points
```

Assuming that each calculation in the inner loop takes 0.0000001 seconds then this would take:

[7]:
```
time = 0.0000001 * (n * n - n)/2
print(f"It would take {time} seconds to compute")
```

```
It would take 137371850.1792 seconds to compute
```

Which is equivalent to 1,598.7 days (or 4.4 years) and 11.1 PB of memory to compute! So, it is clearly not feasible to compute the distance matrix using our naive brute force method. Instead, we need to figure out how to reduce this computational complexity by efficiently generating a matrix profile and this is where STUMPY comes into play.

4.1.9 STUMPY

In the fall of 2016, researchers from the University of California, Riverside and the University of New Mexico published a beautiful set of back-to-back papers that described an exact method called STOMP for computing the matrix profile for any time series with a computational complexity of O(n2)! They also further demonstrated this using GPUs and they called this faster approach GPU-STOMP.

With the academics, data scientists, and developers in mind, we have taken these concepts and have open sourced STUMPY, a powerful and scalable library that efficiently computes the matrix profile according to this published research. And, thanks to other open source software such as Numba and Dask, our implementation is highly parallelized (for a single server with multiple CPUs or, alternatively, multiple GPUs), highly distributed (with multiple CPUs across multiple servers). We've tested STUMPY on as many as 256 CPU cores (spread across 32 servers) or 16 NVIDIA GPU devices (on the same DGX-2 server) and have achieved similar performance to the published GPU-STOMP work.

4.1.10 Conclusion

According to the original authors, "these are the best ideas in times series data mining in the last two decades" and "given the matrix profile, most time series data mining problems are trivial to solve in a few lines of code".

From our experience, this is definitely true and we are excited to share STUMPY with you! Please reach out and let us know how STUMPY has enabled your time series analysis work as we'd love to hear from you!

4.1.11 Additional Notes

For the sake of completeness, we'll provide a few more comments for those of you who'd like to compare your own matrix profile implementation to STUMPY. However, due to the many details that are omitted in the original papers, we strongly encourage you to use STUMPY.

In our explanation above, we've only excluded the trivial match from consideration. However, this is insufficient since nearby subsequences (i.e., `i ± 1`) are likely highly similar and we need to expand this to a larger "exclusion zone" relative to the diagonal trivial match. Here, we can visualize what different exclusion zones look like:

Trivial Match Only

	7.4	6.9	14.7	19.3	17.7	19.9	15.0	8.2	8.9
7.4		10.9	7.9	15.7	18.8	19.1	15.8	1.4	8.4
6.9	10.9		16.8	16.1	13.6	18.8	14.0	11.6	6.2
14.7	7.9	16.8		16.8	19.8	18.0	19.4	8.2	13.4
19.3	15.7	16.1	16.8		20.7	23.6	18.7	15.3	11.4
17.7	18.8	13.6	19.8	20.7		19.2	23.1	19.8	14.4
19.9	19.1	18.8	18.0	23.6	19.2		14.1	20.1	20.5
15.0	15.8	14.0	19.4	18.7	23.1	14.1		16.2	16.1
8.2	1.4	11.6	8.2	15.3	19.8	20.1	16.2		8.6
8.9	8.4	6.2	13.4	11.4	14.4	20.5	16.1	8.6	

However, in practice, it has been found that an exclusion zone of `i ± int(np.ceil(m / 4))` works well (where `m` is the subsequence window size) and the distances computed in this region are is set to `np.inf` before the matrix profile value is extracted for the `ith` subsequence. Thus, the larger the window size is, the larger the exclusion zone will be. Additionally, note that, since NumPy indexing has an inclusive start index but an exlusive stop index, the proper way to ensure a symmetrical exclusion zone is:

```
excl_zone = int(np.ceil(m / 4))
zone_start = i - excl_zone
zone_end = i + excl_zone + 1  # Notice that we add one since this is exclusive
distance_profile[zone_start : zone_end] = np.inf
```

4.1.12 Resources

STUMPY Documentation

STUMPY Matrix Profile Github Code Repository

4.2 STUMPY Basics

4.2.1 Analyzing Motifs and Anomalies with STUMP

This tutorial utilizes the main takeaways from the research papers: Matrix Profile I & Matrix Profile II.

To explore the basic concepts, we'll use the workhorse `stump` function to find interesting motifs (patterns) or discords (anomalies/novelties) and demonstrate these concepts with two different time series datasets:

1. The Steamgen dataset
2. The NYC taxi passengers dataset

`stump` is Numba JIT-compiled version of the popular STOMP algorithm that is described in detail in the original Matrix Profile II paper. `stump` is capable of parallel computation and it performs an ordered search for patterns and outliers within a specified time series and takes advantage of the locality of some calculations to minimize the runtime.

4.2.2 Getting Started

Let's import the packages that we'll need to load, analyze, and plot the data.

[1]:
```
%matplotlib inline

import pandas as pd
import stumpy
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.dates as dates
from matplotlib.patches import Rectangle
import datetime as dt

plt.rcParams["figure.figsize"] = [20, 6]  # width, height
plt.rcParams['xtick.direction'] = 'out'
```

4.2.3 What is a Motif?

Time series motifs are approximately repeated subsequences found within a longer time series. Being able to say that a subsequence is "approximately repeated" requires that you be able to compare subsequences to each other. In the case of STUMPY, all subsequences within a time series can be compared by computing the pairwise z-normalized Euclidean distances and then storing only the index to its nearest neighbor. This nearest neighbor distance vector is referred to as the `matrix profile` and the index to each nearest neighbor within the time series is referred to as the `matrix profile index`. Luckily, the `stump` function takes in any time series (with floating point values) and computes the matrix profile along with the matrix profile indices and, in turn, one can immediately find time series motifs. Let's look at an example:

4.2.4 Loading the Steamgen Dataset

This data was generated using fuzzy models applied to mimic a steam generator at the Abbott Power Plant in Champaign, IL. The data feature that we are interested in is the output steam flow telemetry that has units of kg/s and the data is "sampled" every three seconds with a total of 9,600 datapoints.

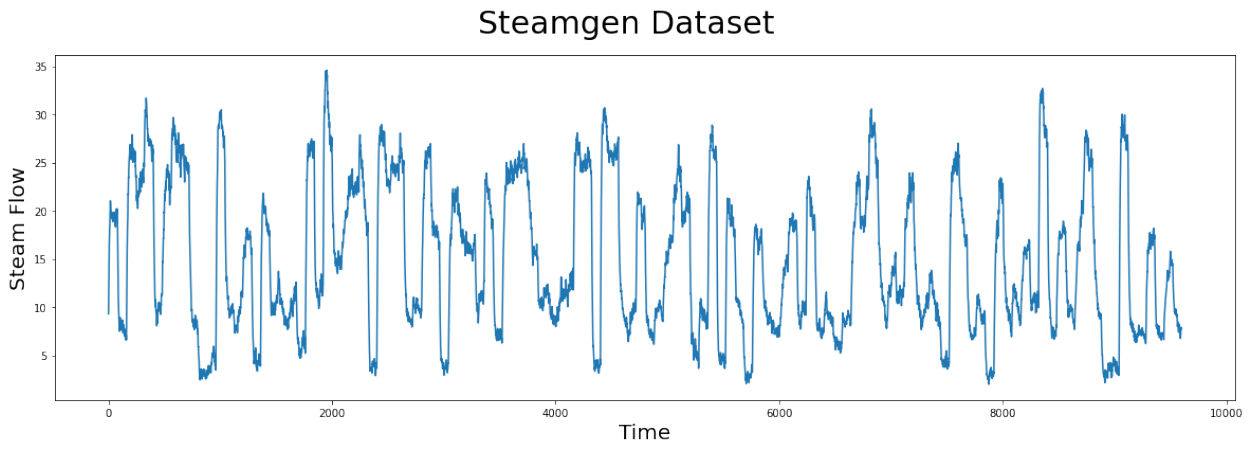
[2]:
```
steam_df = pd.read_csv("https://zenodo.org/record/4273921/files/STUMPY_Basics_
↪steamgen.csv?download=1")
steam_df.head()
```

[2]:

	drum pressure	excess oxygen	water level	steam flow
0	320.08239	2.506774	0.032701	9.302970
1	321.71099	2.545908	0.284799	9.662621
2	320.91331	2.360562	0.203652	10.990955
3	325.00252	0.027054	0.326187	12.430107
4	326.65276	0.285649	0.753776	13.681666

4.2.5 Visualizing the Steamgen Dataset

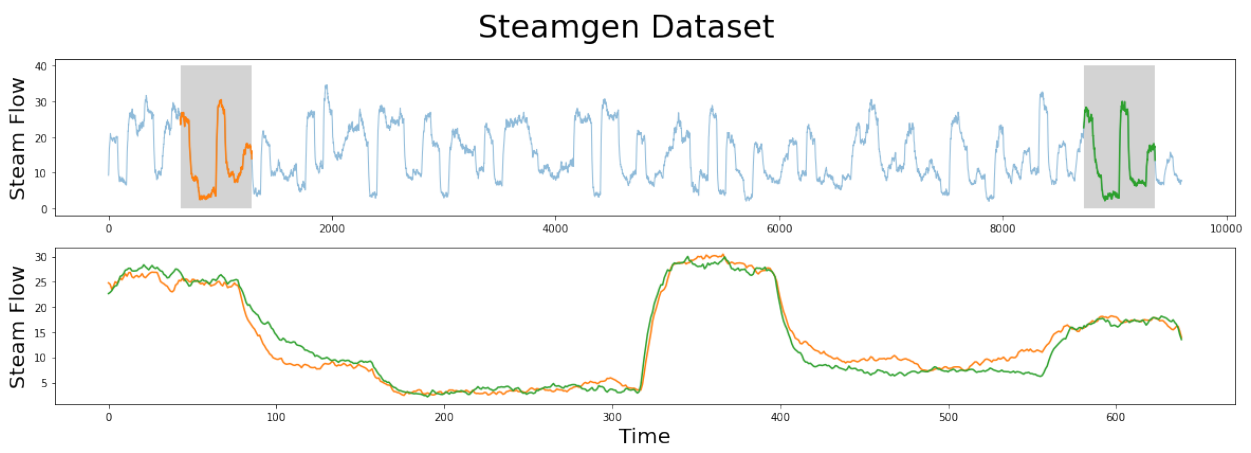
[3]:
```
plt.suptitle('Steamgen Dataset', fontsize='30')
plt.xlabel('Time', fontsize ='20')
plt.ylabel('Steam Flow', fontsize='20')
plt.plot(steam_df['steam flow'].values)
plt.show()
```



Take a moment and carefully examine the plot above with your naked eye. If you were told that there was a pattern that was approximately repeated, can you spot it? Even for a computer, this can be very challenging. Here's what you should be looking for:

4.2.6 Manually Finding a Motif

[4]:
```
m = 640
fig, axs = plt.subplots(2)
plt.suptitle('Steamgen Dataset', fontsize='30')
axs[0].set_ylabel("Steam Flow", fontsize='20')
axs[0].plot(steam_df['steam flow'], alpha=0.5, linewidth=1)
axs[0].plot(steam_df['steam flow'].iloc[643:643+m])
axs[0].plot(steam_df['steam flow'].iloc[8724:8724+m])
rect = Rectangle((643, 0), m, 40, facecolor='lightgrey')
axs[0].add_patch(rect)
rect = Rectangle((8724, 0), m, 40, facecolor='lightgrey')
axs[0].add_patch(rect)
axs[1].set_xlabel("Time", fontsize='20')
axs[1].set_ylabel("Steam Flow", fontsize='20')
axs[1].plot(steam_df['steam flow'].values[643:643+m], color='C1')
axs[1].plot(steam_df['steam flow'].values[8724:8724+m], color='C2')
plt.show()
```



The motif (pattern) that we are looking for is highlighted above and yet it is still very hard to be certain that the orange and green subsequences are a match (upper panel), that is, until we zoom in on them and overlay the subsequences on top each other (lower panel). Now, we can clearly see that the motif is very similar! The fundamental value of computing the matrix profile is that it not only allows you to quickly find motifs but it also identifies the nearest neighbor for all subsequences within your time series. Note that we haven't actually done anything special here to locate the motif except that we grab the locations from the original paper and plotted them. Now, let's take our steamgen data and apply the `stump` function to it:

4.2.7 Find a Motif Using STUMP

[5]:
```
m = 640
mp = stumpy.stump(steam_df['steam flow'], m)
```

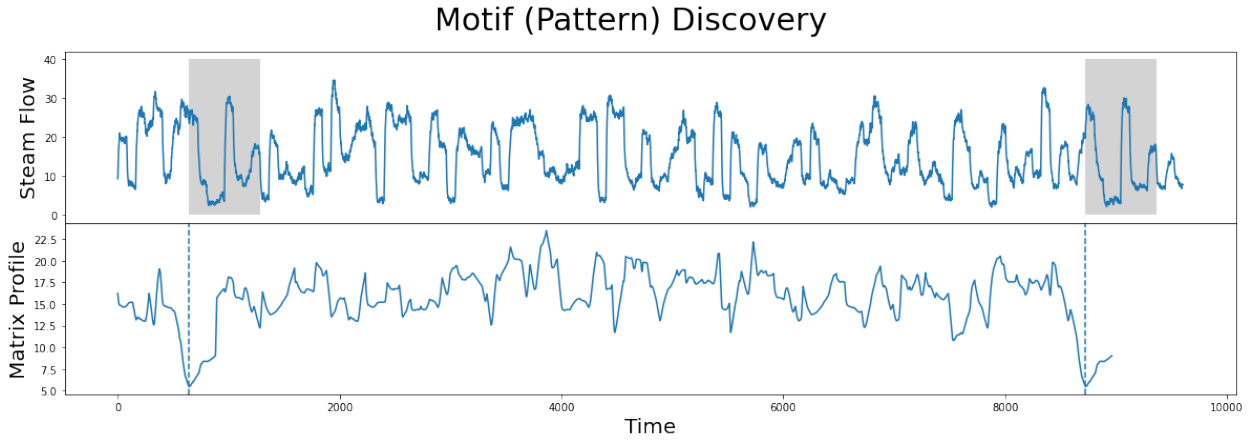
`stump` requires two parameters:

1. A time series
2. A window size, `m`

In this case, based on some domain expertise, we've chosen `m = 640`, which is roughly equivalent to half-hour windows. And, again, the output of `stump` is an array that contains all of the matrix profile values (i.e., z-normalized

Euclidean distance to your nearest neighbor) and matrix profile indices in the first and second columns, respectively (we'll ignore the third and fourth columns for now). Let's plot the matrix profile next to our raw data:

[6]:
```
fig, axs = plt.subplots(2, sharex=True, gridspec_kw={'hspace': 0})
plt.suptitle('Motif (Pattern) Discovery', fontsize='30')

axs[0].plot(steam_df['steam flow'].values)
axs[0].set_ylabel('Steam Flow', fontsize='20')
rect = Rectangle((643, 0), m, 40, facecolor='lightgrey')
axs[0].add_patch(rect)
rect = Rectangle((8724, 0), m, 40, facecolor='lightgrey')
axs[0].add_patch(rect)
axs[1].set_xlabel('Time', fontsize ='20')
axs[1].set_ylabel('Matrix Profile', fontsize='20')
axs[1].axvline(x=643, linestyle="dashed")
axs[1].axvline(x=8724, linestyle="dashed")
axs[1].plot(mp[:, 0])
plt.show()
```



What we learn is that the global minima (vertical dashed lines) from the matrix profile correspond to the locations of the two subsequences that make up the motif pair! And the exact z-normalized Euclidean distance between these two subsequences is:

[7]:
```
mp[:, 0].min()
```

[7]:
```
5.491619827769589
```

So, this distance isn't zero since we saw that the two subsequences aren't an identical match but, relative to the rest of the matrix profile (i.e., compared to either the mean or median matrix profile values), we can understand that this motif is a significantly good match.

4.2.8 Find Anomalies using STUMP

Conversely, the maximum value in the matrix profile (computed from `stump` above) is:

[8]:
```
mp[:, 0].max()
```

[8]:
```
23.476168367301977
```

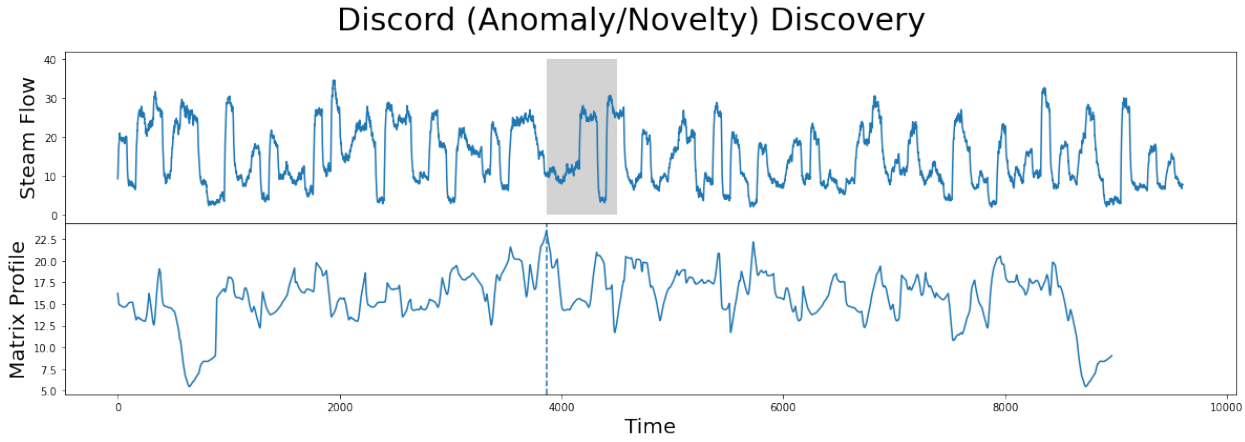
The matrix profile index also tells us which subsequence within the time series does not have nearest neighbor that resembles itself:

[9]:
```
np.argwhere(mp[:, 0] == mp[:, 0].max()).flatten()[0]
```

[9]:
```
3864
```

The subsequence located at this global maximum is also referred to as a discord, novelty, or anomaly:

[10]:
```
fig, axs = plt.subplots(2, sharex=True, gridspec_kw={'hspace': 0})
plt.suptitle('Discord (Anomaly/Novelty) Discovery', fontsize='30')

axs[0].plot(steam_df['steam flow'].values)
axs[0].set_ylabel('Steam Flow', fontsize='20')
rect = Rectangle((3864, 0), m, 40, facecolor='lightgrey')
axs[0].add_patch(rect)
axs[1].set_xlabel('Time', fontsize ='20')
axs[1].set_ylabel('Matrix Profile', fontsize='20')
axs[1].axvline(x=3864, linestyle="dashed")
axs[1].plot(mp[:, 0])
plt.show()
```



Now that you've mastered the STUMPY basics and understand how to discover motifs and anomalies from a time series, we'll leave it up to you to investigate other interesting local minima and local maxima in the steamgen dataset.

To further develop/reinforce our growing intuition, let's move on and explore another dataset!

4.2.9 Loading the NYC Taxi Passengers Dataset

First, we'll download historical data that represents the half-hourly average of the number of NYC taxi passengers over 75 days in the Fall of 2014.

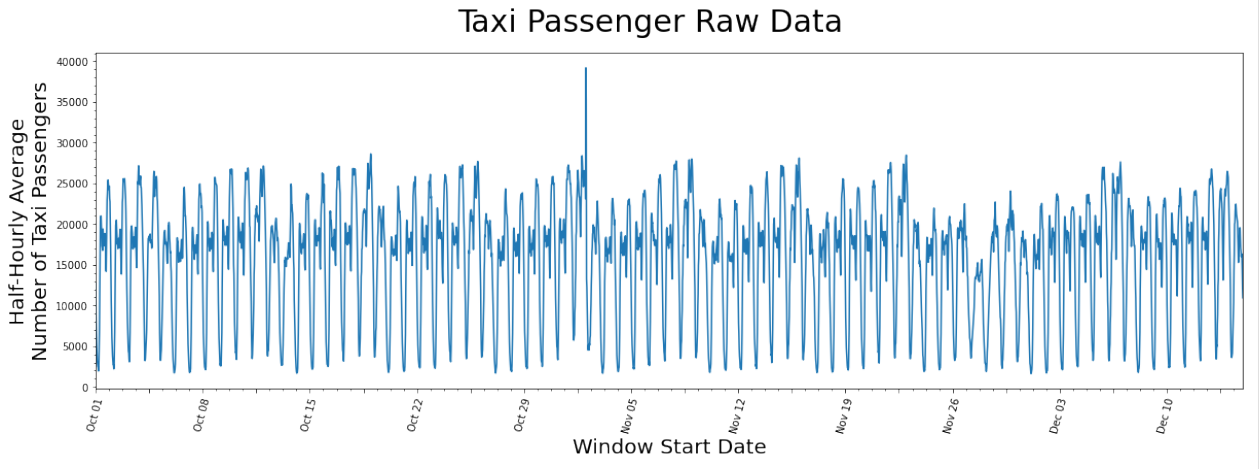
We extract that data and insert it into a pandas dataframe, making sure the timestamps are stored as *datetime* objects and the values are of type *float64*. Note that we'll do a little more data cleaning than above just so you can see an example where the timestamp is included. But be aware that `stump` does not actually use or need the timestamp column at all when computing the matrix profile.

[11]:
```
taxi_df = pd.read_csv("https://zenodo.org/record/4276428/files/STUMPY_Basics_Taxi.csv?
↪download=1")
taxi_df['value'] = taxi_df['value'].astype(np.float64)
taxi_df['timestamp'] = pd.to_datetime(taxi_df['timestamp'])
taxi_df.head()
```

[11]:
```
            timestamp    value
0 2014-10-01 00:00:00  12751.0
1 2014-10-01 00:30:00   8767.0
2 2014-10-01 01:00:00   7005.0
3 2014-10-01 01:30:00   5257.0
4 2014-10-01 02:00:00   4189.0
```

4.2.10 Visualizing the Taxi Dataset

[12]:
```
# This code is going to be utilized to control the axis labeling of the plots
DAY_MULTIPLIER = 7  # Specify for the amount of days you want between each labeled x-
↪axis tick

x_axis_labels = taxi_df[(taxi_df.timestamp.dt.hour==0)]['timestamp'].dt.strftime('%b
↪%d').values[::DAY_MULTIPLIER]
x_axis_labels[1::2] = " "
x_axis_labels, DAY_MULTIPLIER

plt.suptitle('Taxi Passenger Raw Data', fontsize='30')
plt.xlabel('Window Start Date', fontsize ='20')
plt.ylabel('Half-Hourly Average\nNumber of Taxi Passengers', fontsize='20')
plt.plot(taxi_df['value'])

plt.xticks(np.arange(0, taxi_df['value'].shape[0], (48*DAY_MULTIPLIER)/2), x_axis_
↪labels)
plt.xticks(rotation=75)
plt.minorticks_on()
plt.margins(x=0)
plt.show()
```



It seems as if there is a general periodicity between spans of 1-day and 7-days, which can likely be explained by the fact that more people use taxis throughout the day than through the night and that it is reasonable to say most weeks have similar taxi-rider patterns. Also, maybe there is an outlier just to the right of the window starting near the end of October but, other than that, there isn't anything you can conclude from just looking at the raw data.

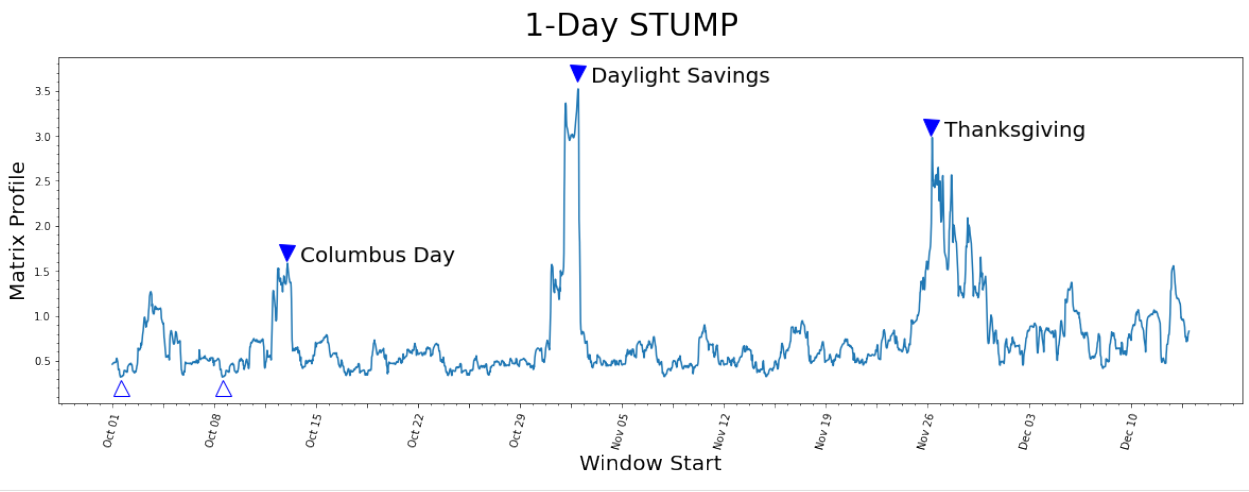
4.2.11 Generating the Matrix Profile

Again, defining the window size, `m`, usually requires some level of domain knowledge but we'll demonstrate later on that `stump` is robust to changes in this parameter. Since this data was taken half-hourly, we chose a value `m = 48` to represent the span of exactly one day:

[13]:
```
m = 48
mp = stumpy.stump(taxi_df['value'], m=m)
```

4.2.12 Visualizing the Matrix Profile

[14]:
```
plt.suptitle('1-Day STUMP', fontsize='30')
plt.xlabel('Window Start', fontsize ='20')
plt.ylabel('Matrix Profile', fontsize='20')
plt.plot(mp[:, 0])

plt.plot(575, 1.7, marker="v", markersize=15, color='b')
plt.text(620, 1.6, 'Columbus Day', color="black", fontsize=20)
plt.plot(1535, 3.7, marker="v", markersize=15, color='b')
plt.text(1580, 3.6, 'Daylight Savings', color="black", fontsize=20)
plt.plot(2700, 3.1, marker="v", markersize=15, color='b')
plt.text(2745, 3.0, 'Thanksgiving', color="black", fontsize=20)
plt.plot(30, .2, marker="^", markersize=15, color='b', fillstyle='none')
plt.plot(363, .2, marker="^", markersize=15, color='b', fillstyle='none')
plt.xticks(np.arange(0, 3553, (m*DAY_MULTIPLIER)/2), x_axis_labels)
plt.xticks(rotation=75)
plt.minorticks_on()
plt.show()
```



4.2.13 Understanding the Matrix Profile

Let's understand what we're looking at.

Lowest Values

The lowest values (open triangles) are considered a motif since they represent the pair of nearest neighbor subsequences with the smallest z-normalized Euclidean distance. Interestingly, the two lowest data points are *exactly* 7 days

apart, which suggests that, in this dataset, there may be a periodicity of seven days in addition to the more obvious periodicity of one day.

Highest Values

So what about the highest matrix profile values (filled triangles)? The subsequences that have the highest (local) values really emphasizes their uniqueness. We found that the top three peaks happened to correspond exactly with the timing of Columbus Day, Daylight Saving Time, and Thanksgiving, respectively.

4.2.14 Different Window Sizes

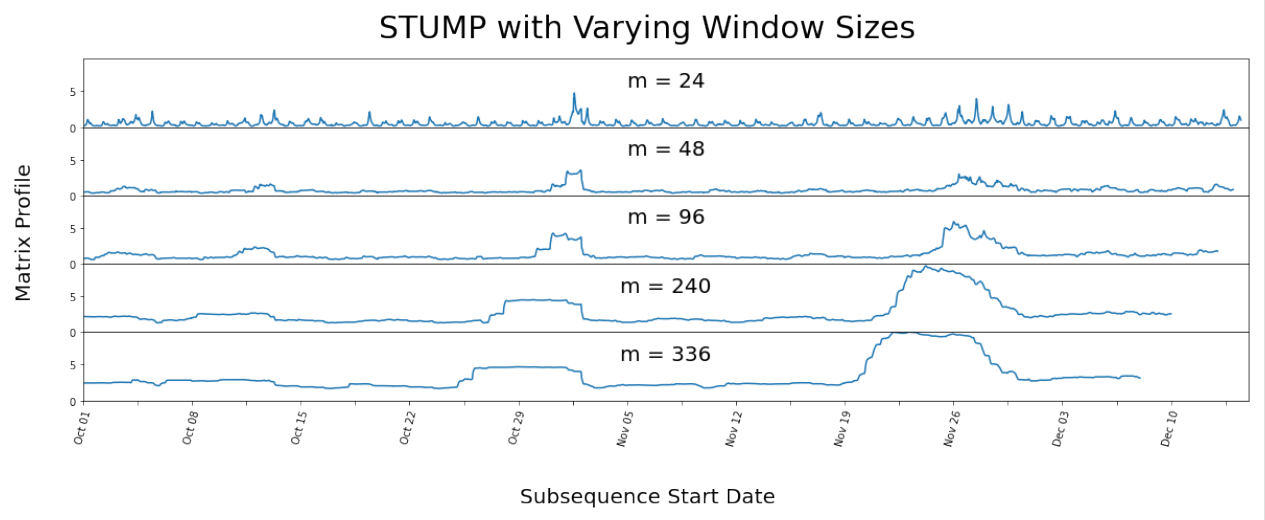
As we had mentioned above, `stump` should be robust to the choice of the window size parameter, `m`. Below, we demonstrate how manipulating the window size can have little impact on your resulting matrix profile by running `stump` with varying windows sizes.

[15]:
```
days_dict ={
  "Half-Day": 24,
  "1-Day": 48,
  "2-Days": 96,
  "5-Days": 240,
  "7-Days": 336,
}

days_df = pd.DataFrame.from_dict(days_dict, orient='index', columns=['m'])
days_df.head()
```

[15]:
```
            m
Half-Day   24
1-Day      48
2-Days     96
5-Days    240
7-Days    336
```

We purposely chose spans of time that correspond to reasonably intuitive day-lengths that could be chosen by a human.

[16]:
```
fig, axs = plt.subplots(5, sharex=True, gridspec_kw={'hspace': 0})
fig.text(0.5, -0.1, 'Subsequence Start Date', ha='center', fontsize='20')
fig.text(0.08, 0.5, 'Matrix Profile', va='center', rotation='vertical', fontsize='20')
for i, varying_m in enumerate(days_df['m'].values):
    mp = stumpy.stump(taxi_df['value'], varying_m)
    axs[i].plot(mp[:, 0])
    axs[i].set_ylim(0,9.5)
    axs[i].set_xlim(0,3600)
    title = f"m = {varying_m}"
    axs[i].set_title(title, fontsize=20, y=.5)
plt.xticks(np.arange(0, taxi_df.shape[0], (48*DAY_MULTIPLIER)/2), x_axis_labels)
plt.xticks(rotation=75)
plt.suptitle('STUMP with Varying Window Sizes', fontsize='30')
plt.show()
```



We can see that even with varying window sizes, our peaks stay prominent. But it looks as if all the non-peak values are converging towards each other. This is why having a knowledge of the data-context is important prior to running `stump`, as it is helpful to have a window size that may capture a repeating pattern or anomaly within the dataset.

4.2.15 GPU-STUMP - Faster STUMP Using GPUs

When you have significantly more than a few thousand data points in your time series, you may need a speed boost to help analyze your data. Luckily, you can try `gpu_stump`, a super fast GPU-powered alternative to `stump` that gives speed of a few hundred CPUs and provides the same output as `stump`:

```
import stumpy

mp = stumpy.gpu_stump(df['value'], m=m)  # Note that you'll need a properly␣
↪configured NVIDIA GPU for this
```

In fact, if you aren't dealing with PII/SII data, then you can try out `gpu_stump` using the this notebook on Google Colab.

4.2.16 STUMPED - Distributed STUMP

Alternatively, if you only have access to a cluster of CPUs and your data needs to stay behind your firewall, then `stump` and `gpu_stump` may not be sufficient for your needs. Instead, you can try `stumped`, a distributed and parallel implementation of `stump` that depends on Dask distributed:

```
import stumpy
from dask.distributed import Client

dask_client = Client()

mp = stumpy.stumped(dask_client, df['value'], m=m)  # Note that a dask client is␣
↪needed
```

4.2.17 Summary

And that's it! You have now loaded in a dataset, ran it through `stump` using our package, and were able to extract multiple conclusions of existing patterns and anomalies within the two different time series. You can now import this package and use it in your own projects. Happy coding!

4.2.18 Resources

Matrix Profile I

Matrix Profile II

STUMPY Documentation

STUMPY Matrix Profile Github Code Repository

4.3 Time Series Chains

4.3.1 Forecasting Web Query Data with Anchored Time Series Chains (ATSC)

This example is adapted from the Web Query Volume case study and utilizes the main takeaways from the Matrix Profile VII research paper.

4.3.2 Getting Started

Let's import the packages that we'll need to load, analyze, and plot the data.

[1]:
```
%matplotlib inline

import pandas as pd
import numpy as np
import stumpy
from scipy.io import loadmat
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle, FancyArrowPatch
import itertools

plt.rcParams["figure.figsize"] = [20, 6]  # width, height
plt.rcParams['xtick.direction'] = 'out'
```

4.3.3 What are Time Series Chains?

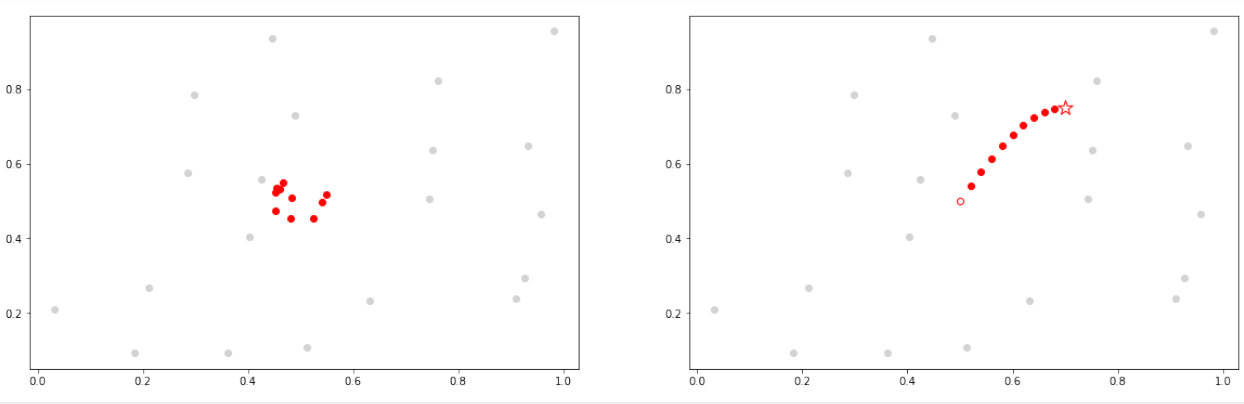
Time series chains may be informally considered as motifs that evolve or drift in some direction over time. The figure below illustrates the difference between time series motifs (left) and time series chains (right).

[2]:
```
x = np.random.rand(20)
y = np.random.rand(20)
n = 10
motifs_x = 0.5 * np.ones(n) + np.random.uniform(-0.05, 0.05, n)
motifs_y = 0.5 * np.ones(n) + np.random.uniform(-0.05, 0.05, n)
sin_x = np.linspace(0, np.pi/2, n+1)
```

(continues on next page)

(continued from previous page)

```
sin_y = np.sin(sin_x)/4
chains_x = 0.5 * np.ones(n+1) + 0.02 * np.arange(n+1)
chains_y = 0.5 * np.ones(n+1) + sin_y
fig, axes = plt.subplots(nrows=1, ncols=2)
axes[0].scatter(x, y, color='lightgrey')
axes[0].scatter(motifs_x, motifs_y, color='red')
axes[1].scatter(x, y, color='lightgrey')
axes[1].scatter(chains_x[0], chains_y[0], edgecolor='red', color='white')
axes[1].scatter(chains_x[1:n], chains_y[1:n], color='red')
axes[1].scatter(chains_x[n], chains_y[n], edgecolor='red', color='white', marker='*',␣
↪s=200)
plt.show()
```



Above, we are visualizing time series subsequences as points in high-dimensional space. Shown on the left is a time series motif and it can be thought of as a collection of points that approximate a platonic ideal. In contrast, depicted on the right, is a time series chain and it may be thought of as an evolving trail of points in the space. Here, the open red circle represents the first link in the chain, the anchor. Both motifs and chains have the property that each subsequence is relatively close to its nearest neighbor. However, the motif set (left) also has a relatively small diameter. In contrast, the set of points in a chain (right) has a diameter that is much larger than the mean of each member's distance to its nearest neighbor and, moreover, the chain has the important property of directionality. For example, in the case of a motif, if an additional member was added to the motif set, its location will also be somewhere near the platonic ideal, but independent of the previous subsequences. In contrast, in the case of a chain, the location of the next member of the chain would be somewhere after the last red circle, possibly where the open red star is located.

4.3.4 A Simplified Example

Adapted from the Matrix Profile VII paper, consider the following time series:

47, 32, 1, 22, 2, 58, 3, 36, 4, -5, 5, 40

Assume that the subsequence length is 1 and the distance between two subsequences is simply the absolute difference between them. To be clear, we are making these simple and pathological assumptions here just for the purposes of elucidation; we are actually targeting much longer subsequence lengths and using z-normalized Euclidean distance in our applications. To capture the directionality of a time series chain, we need to store the left and right nearest neighbor information into the left (IL) and right (IR) matrix profile indices:

Index	Value	Left Index (IL)	Right Index (IR)
1	47	•	12
2	32	1	8
3	1	2	5
4	22	2	8
5	2	3	7
6	58	1	12
7	3	5	9
8	36	2	12
9	4	7	11
10	-5	3	11
11	5	9	12
12	40	8	•

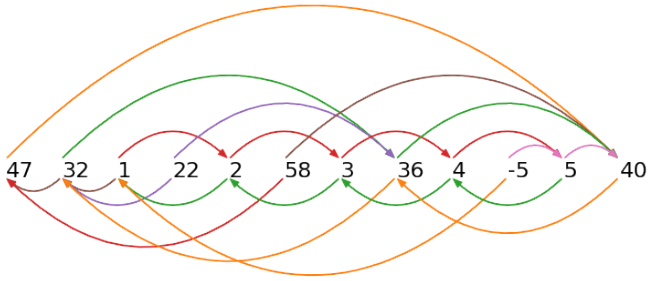
In this vertical/transposed representation, the `index` column shows the location of every subsequence in the time series, the `value` column contains the original numbers from our time series above, the `IL` column shows the left matrix profile indices, and `IR` is the right matrix profile indices. For example, `IR[2] = 8` means the right nearest neighbor of `index = 2` (which has `value = 32`) is at `index = 8` (which has `value = 36`). Similarly, `IL[3] = 2` means that the left nearest neighbor of `index = 3` (with `value = 1`) is at `index = 2` (which has `value = 32`). To better visualize the left/right matrix profile index, we use arrows to link every subsequence in the time series with its left and right nearest neighbors:

[3]:
```
nearest_neighbors = np.array([[1,  47, np.nan,     12],
                              [2,  32,       1,      8],
                              [3,   1,       2,      5],
                              [4,  22,       2,      8],
                              [5,   2,       3,      7],
                              [6,  58,       1,     12],
                              [7,   3,       5,      9],
                              [8,  36,       2,     12],
                              [9,   4,       7,     11],
                              [10, -5,       3,     11],
                              [11,  5,       9,     12],
                              [12, 40,       8, np.nan]])

colors = [['C1', 'C1'],
          ['C2', 'C5'],
          ['C3', 'C5'],
          ['C4', 'C4'],
          ['C3', 'C2'],
          ['C5', 'C3'],
          ['C3', 'C2'],
          ['C2', 'C1'],
          ['C3', 'C2'],
          ['C6', 'C1'],
          ['C6', 'C2'],
          ['C1', 'C1']]

style="Simple, tail_width=0.5, head_width=6, head_length=8"
kw = dict(arrowstyle=style, connectionstyle="arc3, rad=-.5",)

xs = np.arange(nearest_neighbors.shape[0]) + 1
```

(continues on next page)

(continued from previous page)

```
ys = np.zeros(nearest_neighbors.shape[0])
plt.plot(xs, ys, "-o", markerfacecolor="None", markeredgecolor="None", linestyle="None
↪")

x0, x1, y0, y1 = plt.axis()
plot_margin = 5.0
plt.axis((x0 - plot_margin,
          x1 + plot_margin,
          y0 - plot_margin,
          y1 + plot_margin))
plt.axis('off')

for x, y, nearest_neighbor, color in zip(xs, ys, nearest_neighbors, colors):
    plt.text(x, y, str(int(nearest_neighbor[1])), color="black", fontsize=20)

    # Plot right matrix profile indices
    if not np.isnan(nearest_neighbor[3]):
        arrow = FancyArrowPatch((x, 0.5), (nearest_neighbor[3], 0.5), color=color[0],␣
↪**kw)
        plt.gca().add_patch(arrow)

    # Plot left matrix profile indices
    if not np.isnan(nearest_neighbor[2]):
        arrow = FancyArrowPatch((x, 0.0), (nearest_neighbor[2], 0.0), color=color[1],␣
↪**kw)
        plt.gca().add_patch(arrow)

plt.show()
```



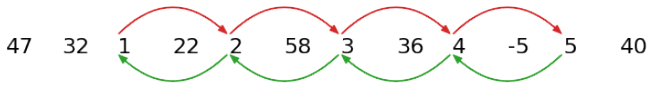
An arrow pointing from a number to its right nearest neighbor (arrows shown above the time series) can be referred to as forward arrow and an arrow pointing from a number to its left nearest neighbor (arrows shown below the time series) can be referred to as a backward arrow. According to the formal definition of a time series chain (see Matrix Profile VII for a thorough definition and discussion), every pair of consecutive subsequences in a chain must be connected by both a forward arrow and a backward arrow. A keen eye will spot the fact that the longest chain in our simplified example is:

[4]:
```
nearest_neighbors = np.array([[1,  47, np.nan, np.nan],
                              [2,  32, np.nan, np.nan],
                              [3,   1, np.nan,      5],
                              [4,  22, np.nan, np.nan],
                              [5,   2,      3,      7],
                              [6,  58, np.nan, np.nan],
                              [7,   3,      5,      9],
```

(continues on next page)

(continued from previous page)

```
                              [8,  36, np.nan, np.nan],
                              [9,   4,      7,     11],
                              [10, -5, np.nan, np.nan],
                              [11,  5,      9, np.nan],
                              [12, 40, np.nan, np.nan]])

colors = [['C1', 'C1'],
          ['C2', 'C5'],
          ['C3', 'C5'],
          ['C4', 'C4'],
          ['C3', 'C2'],
          ['C5', 'C3'],
          ['C3', 'C2'],
          ['C2', 'C1'],
          ['C3', 'C2'],
          ['C6', 'C1'],
          ['C6', 'C2'],
          ['C1', 'C1']]

style="Simple, tail_width=0.5, head_width=6, head_length=8"
kw = dict(arrowstyle=style, connectionstyle="arc3, rad=-.5",)

xs = np.arange(nearest_neighbors.shape[0]) + 1
ys = np.zeros(nearest_neighbors.shape[0])
plt.plot(xs, ys, "-o", markerfacecolor="None", markeredgecolor="None", linestyle="None
↪")

x0, x1, y0, y1 = plt.axis()
plot_margin = 5.0
plt.axis((x0 - plot_margin,
          x1 + plot_margin,
          y0 - plot_margin,
          y1 + plot_margin))
plt.axis('off')

for x, y, nearest_neighbor, color in zip(xs, ys, nearest_neighbors, colors):
    plt.text(x, y, str(int(nearest_neighbor[1])), color="black", fontsize=20)

    # Plot right matrix profile indices
    if not np.isnan(nearest_neighbor[3]):
        arrow = FancyArrowPatch((x, 0.5), (nearest_neighbor[3], 0.5), color=color[0],␣
↪**kw)
        plt.gca().add_patch(arrow)

    # Plot left matrix profile indices
    if not np.isnan(nearest_neighbor[2]):
        arrow = FancyArrowPatch((x, 0.0), (nearest_neighbor[2], 0.0), color=color[1],␣
↪**kw)
        plt.gca().add_patch(arrow)

plt.show()
```



The longest extracted chain is therefore 1 2 3 4 5. Note that we see a gradual monotonic increase in the data but, in reality, the increase or decrease in drift can happen in arbitrarily complex ways that can be detected by the time series chains approach. The key component of drifting is that the time series must contain chains with clear directionality.

STUMPY is capable of computing:

1. anchored time series chains (ATSC) - grow a chain from a user-specified anchor (i.e., specific subsequence)
2. all-chain set (ALLC) - a set of anchored time series chains (i.e., each chain starts with a particular subsequence) that are not subsumed by another longer chain
3. unanchored time series chain(s) - the unconditionally longest chain within a time series (there could be more than one if there were chains with the same length)

So, what does this mean in the context of a real time series? Let's take a look at a real example from web query data!

4.3.5 Retrieve the Data

We will be looking at a noisy dataset that is under-sampled and has a growing trend, which will perfectly illustrate the idea regarding time series chains. The data contains a decade-long GoogleTrend query volume (collected weekly from 2004-2014) for the keyword Kohl's, an American retail chain. First, we'll download the data, extract it, and insert it into a pandas dataframe.

[5]:
```
df = pd.read_csv("https://zenodo.org/record/4276348/files/Time_Series_Chains_Kohls_
↪data.csv?download=1")
df.head()
```

[5]:
```
     volume
0  0.010417
1  0.010417
2  0.010417
3  0.000000
4  0.000000
```

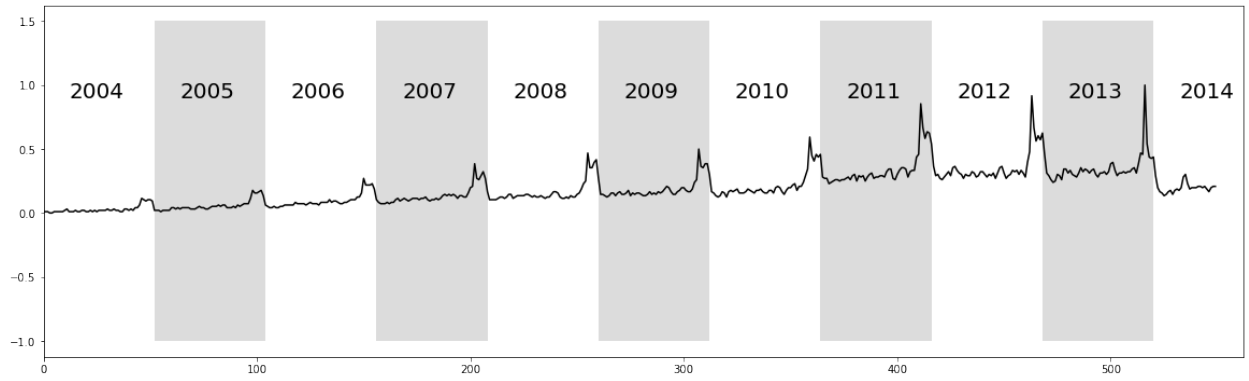
4.3.6 Visualizing the Data

[6]:
```
plt.plot(df['volume'], color='black')
plt.xlim(0, df.shape[0]+12)
color = itertools.cycle(['white', 'gainsboro'])
for i, x in enumerate(range(0, df.shape[0], 52)):
    plt.text(x+12, 0.9, str(2004+i), color="black", fontsize=20)
    rect = Rectangle((x, -1), 52, 2.5, facecolor=next(color))
```

(continues on next page)

(continued from previous page)

```
    plt.gca().add_patch(rect)
plt.show()
```



The raw time series above displays ten years of web query volume for the keyword "Kohl's", where each alternating white and grey vertical band represents a 52 week period starting from 2004 to 2014. As depicted, the time series features a significant but unsurprising "end-of-year holiday bump". Relating back to time series chains, we can see that the bump is generally increasing over time and so we might be able to capture this when we compute the unanchored chain.

However, as we learned above, in order to compute any time series chains, we also need the left and right matrix profile indices. Luckily for us, according to the docstring, the `stump` function not only returns the (bidirectional) matrix profile and the matrix profile indices in the first and second columns of the NumPy array, respectively, but the third and fourth columns consists of the left matrix profile indices and the right matrix profile indices, respectively.

4.3.7 Computing the Left and Right Matrix Profile Indices

So, let's go ahead and compute the matrix profile indices and we'll set the window size, `m = 20`, which is the approximate length of a "bump".

[7]:
```
m = 20
mp = stumpy.stump(df['volume'], m=m)
```

4.3.8 Computing the Unanchored Chain

Now, with our left and right matrix profile indices in hand, we are ready to call the all-chain set function, `allc`, which not only returns the all-chain set but, as a freebie, it also returns the unconditionally longest chain, also know as the unanchored chain. The latter of which is really what we're most interested in.

[8]:
```
all_chain_set, unanchored_chain = stumpy.allc(mp[:, 2], mp[:, 3])
```

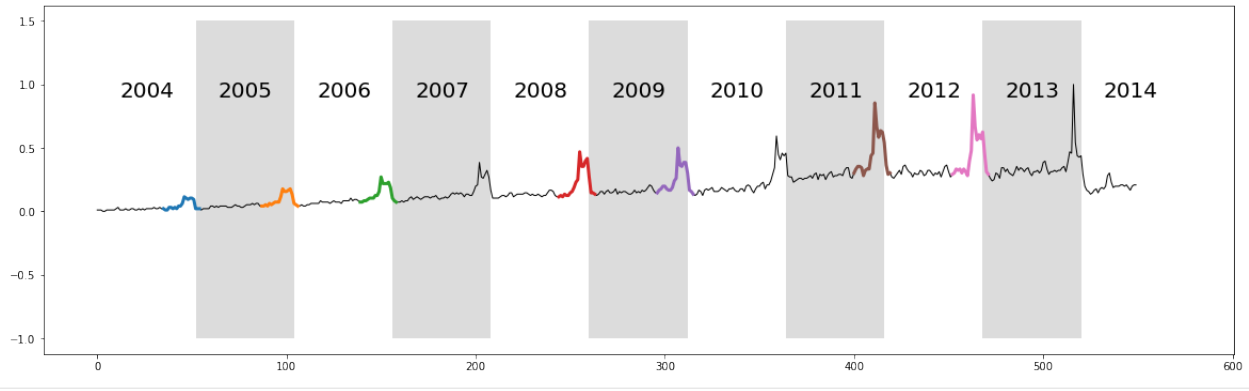
4.3.9 Visualizing the Unanchored Chain

[9]:
```
plt.plot(df['volume'], linewidth=1, color='black')
for i in range(unanchored_chain.shape[0]):
    y = df['volume'].iloc[unanchored_chain[i]:unanchored_chain[i]+m]
    x = y.index.values
    plt.plot(x, y, linewidth=3)
```

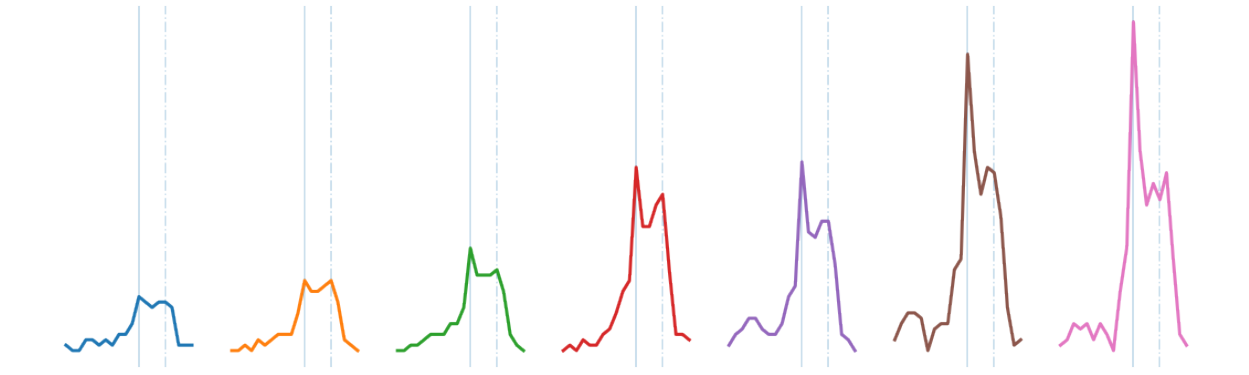
(continues on next page)

(continued from previous page)

```
color = itertools.cycle(['white', 'gainsboro'])
for i, x in enumerate(range(0, df.shape[0], 52)):
    plt.text(x+12, 0.9, str(2004+i), color="black", fontsize=20)
    rect = Rectangle((x, -1), 52, 2.5, facecolor=next(color))
    plt.gca().add_patch(rect)

plt.show()
```



[10]:
```
plt.axis('off')
for i in range(unanchored_chain.shape[0]):
    data = df['volume'].iloc[unanchored_chain[i]:unanchored_chain[i]+m].reset_index().
↪values
    x = data[:, 0]
    y = data[:, 1]
    plt.axvline(x=x[0]-x.min()+(m+5)*i + 11, alpha=0.3)
    plt.axvline(x=x[0]-x.min()+(m+5)*i + 15, alpha=0.3, linestyle='-.')
    plt.plot(x-x.min()+(m+5)*i, y-y.min(), linewidth=3)
plt.show()
```



The discovered chain shows that over the decade, the bump transitions from a smooth bump covering the period between Thanksgiving (solid vertical line) and Christmas (dashed vertical line), to a more sharply focused bump centered on Thanksgiving. This seems to reflect the growing importance of "Cyber Monday", a marketing term for the Monday after Thanksgiving. The phrase was created by marketing companies to persuade consumers to shop online. The term made its debut on November 28th, 2005 in a press release entitled "Cyber Monday Quickly Becoming One of the Biggest Online Shopping Days of the Year". Note that this date coincides with the first glimpse of the sharpening peak in our chain.

It also appears that we may have "missed" a few links in the chain. However, note that the data is noisy and under-sampled, and the "missed" bumps are too distorted to conform with the general evolving trend. This noisy example actually illustrates the robustness of the time series chains technique. As noted before, we don't actually need "per-

fect" data in order to find meaningful chains. Even if some links are badly distorted, the discovered chain will still be able to include all of the other evolving patterns.

One final consideration is the potential use of chains to predict the future. One could leverage the evolving links within the chains in order to forecast the shape of the next bump. We refer the reader to the Matrix Profile VII for further discussions on this topic.

4.3.10 Summary

And that's it! You've just learned the basics of how to identify directional trends, also known as chains, within your data using the matrix profile indices and leveraging `allc`.

4.3.11 Resources

Matrix Profile VII

Matrix Profile VII Supplementary Materials

STUMPY Documentation

STUMPY Matrix Profile Github Code Repository

4.4 Semantic Segmentation

4.4.1 Analyzing Arterial Blood Pressure Data with FLUSS and FLOSS

This example utilizes the main takeaways from the Matrix Profile VIII research paper. For proper context, we highly recommend that you read the paper first but know that our implementations follow this paper closely.

According to the aforementioned publication, "one of the most basic analyses one can perform on [increasing amounts of time series data being captured] is to segment it into homogeneous regions." In other words, wouldn't it be nice if you could take your long time series data and be able to segment or chop it up into `k` regions (where `k` is small) and with the ultimate goal of presenting only `k` short representative patterns to a human (or machine) annotator in order to produce labels for the entire dataset. These segmented regions are also known as "regimes". Additionally, as an exploratory tool, one might uncover new actionable insights in the data that was previously undiscovered. Fast low-cost unipotent semantic segmentation (FLUSS) is an algorithm that produces something called an "arc curve" which annotates the raw time series with information about the likelihood of a regime change. Fast low-cost online semantic segmentation (FLOSS) is a variation of FLUSS that, according to the original paper, is domain agnostic, offers streaming capabilities with potential for actionable real-time intervention, and is suitable for real world data (i.e., does not assume that every region of the data belongs to a well-defined semantic segment).

To demonstrate the API and underlying principles, we will be looking at arterial blood pressure (ABP) data from from a healthy volunteer resting on a medical tilt table and will be seeing if we can detect when the table is tilted from a horizontal position to a vertical position. This is the same data that is presented throughout the original paper (above).

4.4.2 Getting Started

Let's import the packages that we'll need to load, analyze, and plot the data.

[1]:
```
%matplotlib inline

import pandas as pd
import numpy as np
import stumpy
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle, FancyArrowPatch
from matplotlib import animation
from IPython.display import HTML

plt.rcParams["figure.figsize"] = [20, 6]  # width, height
plt.rcParams['xtick.direction'] = 'out'
```

4.4.3 Retrieve the Data

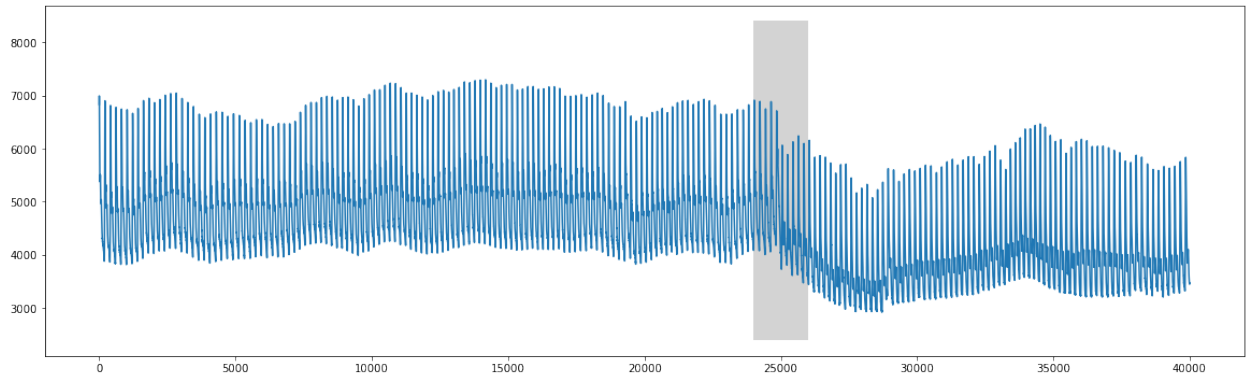
[2]:
```
df = pd.read_csv("https://zenodo.org/record/4276400/files/Semantic_Segmentation_
↪TiltABP.csv?download=1")
df.head()
```

[2]:

	time	abp
0	0	6832.0
1	1	6928.0
2	2	6968.0
3	3	6992.0
4	4	6980.0

4.4.4 Visualizing the Raw Data

[3]:
```
plt.plot(df['time'], df['abp'])
rect = Rectangle((24000,2400),2000,6000,facecolor='lightgrey')
plt.gca().add_patch(rect)
plt.show()
```

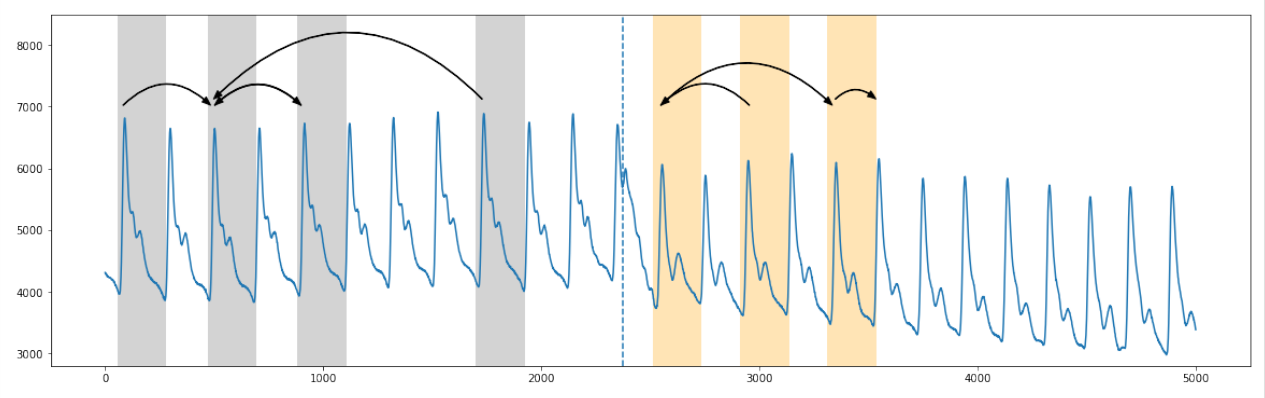


We can clearly see that there is a change around `time=25000` that corresponds to when the table was tilted upright.

4.4.5 FLUSS

Instead of using the full dataset, let's zoom in and analyze the 2,500 data points directly before and after `x=25000` (see Figure 5 in the paper).

[4]:
```
start = 25000 - 2500
stop = 25000 + 2500
abp = df.iloc[start:stop, 1]
plt.plot(range(abp.shape[0]), abp)
plt.ylim(2800, 8500)
plt.axvline(x=2373, linestyle="dashed")

style="Simple, tail_width=0.5, head_width=6, head_length=8"
kw = dict(arrowstyle=style, color="k")

# regime 1
rect = Rectangle((55,2500), 225, 6000, facecolor='lightgrey')
plt.gca().add_patch(rect)
rect = Rectangle((470,2500), 225, 6000, facecolor='lightgrey')
plt.gca().add_patch(rect)
rect = Rectangle((880,2500), 225, 6000, facecolor='lightgrey')
plt.gca().add_patch(rect)
rect = Rectangle((1700,2500), 225, 6000, facecolor='lightgrey')
plt.gca().add_patch(rect)
arrow = FancyArrowPatch((75, 7000), (490, 7000), connectionstyle="arc3, rad=-.5",␣
↪**kw)
plt.gca().add_patch(arrow)
arrow = FancyArrowPatch((495, 7000), (905, 7000), connectionstyle="arc3, rad=-.5",␣
↪**kw)
plt.gca().add_patch(arrow)
arrow = FancyArrowPatch((905, 7000), (495, 7000), connectionstyle="arc3, rad=.5",␣
↪**kw)
plt.gca().add_patch(arrow)
arrow = FancyArrowPatch((1735, 7100), (490, 7100), connectionstyle="arc3, rad=.5",␣
↪**kw)
plt.gca().add_patch(arrow)

# regime 2
rect = Rectangle((2510,2500), 225, 6000, facecolor='moccasin')
plt.gca().add_patch(rect)
rect = Rectangle((2910,2500), 225, 6000, facecolor='moccasin')
plt.gca().add_patch(rect)
rect = Rectangle((3310,2500), 225, 6000, facecolor='moccasin')
plt.gca().add_patch(rect)
arrow = FancyArrowPatch((2540, 7000), (3340, 7000), connectionstyle="arc3, rad=-.5",␣
↪**kw)
plt.gca().add_patch(arrow)
arrow = FancyArrowPatch((2960, 7000), (2540, 7000), connectionstyle="arc3, rad=.5",␣
↪**kw)
plt.gca().add_patch(arrow)
arrow = FancyArrowPatch((3340, 7100), (3540, 7100), connectionstyle="arc3, rad=-.5",␣
↪**kw)
plt.gca().add_patch(arrow)

plt.show()
```



Roughly, in the truncated plot above, we see that the segmentation between the two regimes occurs around `time=2373` (vertical dotted line) where the patterns from the first regime (grey) don't cross over to the second regime (orange) (see Figure 2 in the original paper). And so the "arc curve" is calculated by sliding along the time series and simply counting the number of times other patterns have "crossed over" that specific time point (i.e., "arcs"). Essentially, this information can be extracted by looking at the matrix profile indices (which tells you where along the time series your nearest neighbor is). And so, we'd expect the arc counts to be high where repeated patterns are near each other and low where there are no crossing arcs.

Before we compute the "arc curve", we'll need to first compute the standard matrix profile and we can approximately see that the window size is about 210 data points (thanks to the knowledge of the subject matter/domain expert).

[5]:
```
m = 210
mp = stumpy.stump(abp, m=m)
```

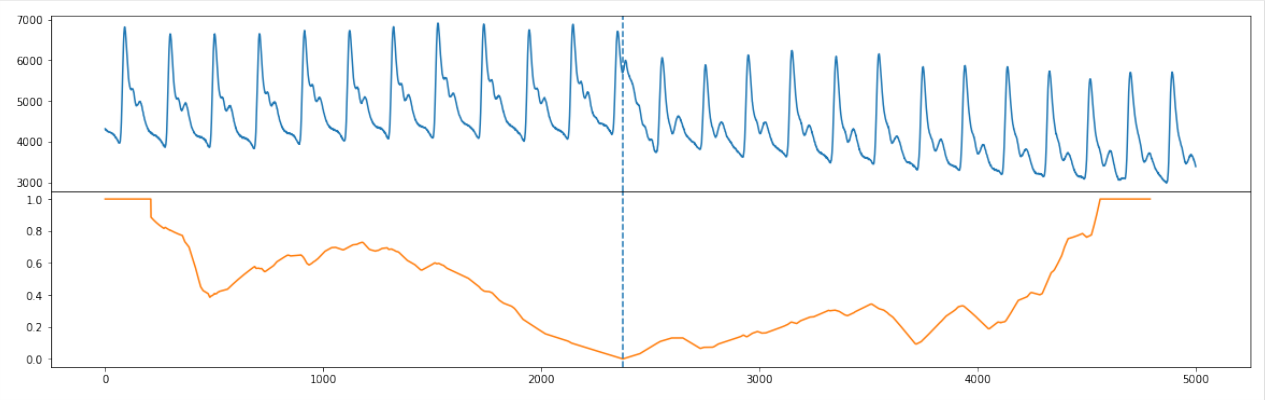
Now, to compute the "arc curve" and determine the location of the regime change, we can directly call the `fluss` function. However, note that `fluss` requires the following inputs:

1. the matrix profile indices `mp[:, 1]` (not the matrix profile distances)
2. an appropriate subsequence length, `L` (for convenience, we'll just choose it to be equal to the window size, `m=210`)
3. the number of regimes, `n_regimes`, to search for (2 regions in this case)
4. an exclusion factor, `excl_factor`, to nullify the beginning and end of the arc curve (anywhere between 1-5 is reasonable according to the paper)

[6]:
```
L = 210
cac, regime_locations = stumpy.fluss(mp[:, 1], L=L, n_regimes=2, excl_factor=1)
```

Notice that `fluss` actually returns something called the "corrected arc curve" (CAC), which normalizes the fact that there are typically less arcs crossing over a time point near the beginning and end of the time series and more potential for cross overs near the middle of the time series. Additionally, `fluss` returns the regimes or location(s) of the dotted line(s). Let's plot our original time series (top) along with the corrected arc curve (orange) and the single regime (vertical dotted line).

[7]:
```
fig, axs = plt.subplots(2, sharex=True, gridspec_kw={'hspace': 0})
axs[0].plot(range(abp.shape[0]), abp)
axs[0].axvline(x=regime_locations[0], linestyle="dashed")
axs[1].plot(range(cac.shape[0]), cac, color='C1')
axs[1].axvline(x=regime_locations[0], linestyle="dashed")
plt.show()
```



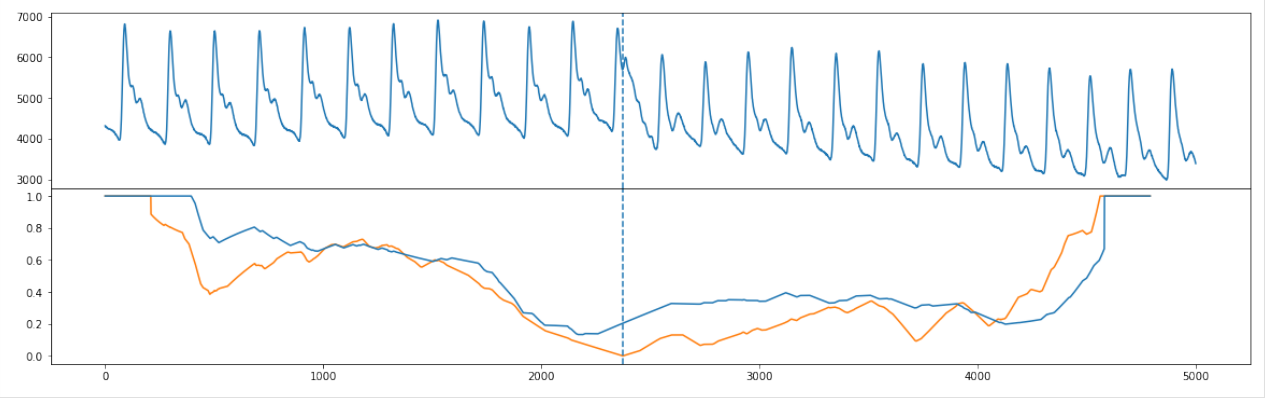
4.4.6 FLOSS

Unlike FLUSS, FLOSS is concerned with streaming data, and so it calculates a modified version of the corrected arc curve (CAC) that is strictly one-directional (CAC_1D) rather than bidirectional. That is, instead of expecting cross overs to be equally likely from both directions, we expect more cross overs to point toward the future (and less to point toward the past). So, we can manually compute the `CAC_1D`

[8]:
```
cac_1d = stumpy._cac(mp[:, 3], L, bidirectional=False, excl_factor=1)  # This is for␣
↪demo purposes only. Use floss() below!
```

and compare the `CAC_1D` (blue) with the bidirectional `CAC` (orange) and we see that the global minimum are approximately in the same place (see Figure 10 in the original paper).

[9]:
```
fig, axs = plt.subplots(2, sharex=True, gridspec_kw={'hspace': 0})
axs[0].plot(np.arange(abp.shape[0]), abp)
axs[0].axvline(x=regime_locations[0], linestyle="dashed")
axs[1].plot(range(cac.shape[0]), cac, color='C1')
axs[1].axvline(x=regime_locations[0], linestyle="dashed")
axs[1].plot(range(cac_1d.shape[0]), cac_1d)
plt.show()
```



4.4.7 Streaming Data with FLOSS

Instead of manually computing `CAC_1D` like we did above on streaming data, we can actually call the `floss` function directly which instantiates a streaming object. To demonstrate the use of `floss`, let's take some `old_data` and compute the matrix profile indices for it like we did above:

[10]:
```
old_data = df.iloc[20000:20000+5000, 1].values  # This is well before the regime␣
↪change has occurred

mp = stumpy.stump(old_data, m=m)
```

Now, we could do what we did early and compute the bidirectional corrected arc curve but we'd like to see how the arc curve changes as a result of adding new data points. So, let's define some new data that is to be streamed in:

[11]:
```
new_data = df.iloc[25000:25000+5000, 1].values
```

Finally, we call the `floss` function to initialize a streaming object and pass in:

1. the matrix profile generated from the `old_data` (only the indices are used)
2. the old data used to generate the matrix profile in 1.
3. the matrix profile window size, `m=210`
4. the subsequence length, `L=210`
5. the exclusion factor

[12]:
```
stream = stumpy.floss(mp, old_data, m=m, L=L, excl_factor=1)
```

You can now update the `stream` with a new data point, `t`, via the `stream.update(t)` function and this will slide your window over by one data point and it will automatically update:

1. the `CAC_1D` (accessed via the `.cac_1d_` attribute)
2. the matrix profile (accessed via the `.P_` attribute)
3. the matrix profile indices (accessed via the `.I_` attribute)
4. the sliding window of data used to produce the `CAC_1D` (accessed via the `.T_` attribute - this should be the same size as the length of the 'old_data)

Let's continuously update our `stream` with the `new_data` one value at a time and store them in a list (you'll see why in a second):

[13]:
```
windows = []
for i, t in enumerate(new_data):
    stream.update(t)

    if i % 100 == 0:
        windows.append((stream.T_, stream.cac_1d_))
```

Below, you can see an animation that changes as a result of updating the stream with new data. For reference, we've also plotted the `CAC_1D` (orange) that we manually generated from above for the stationary data. You'll see that halfway through the animation, the regime change occurs and the updated `CAC_1D` (blue) will be perfectly aligned with the orange curve.

[14]:
```
fig, axs = plt.subplots(2, sharex=True, gridspec_kw={'hspace': 0})

axs[0].set_xlim((0, mp.shape[0]))
axs[0].set_ylim((-0.1, max(np.max(old_data), np.max(new_data))))
axs[1].set_xlim((0, mp.shape[0]))
axs[1].set_ylim((-0.1, 1.1))

lines = []
for ax in axs:
```

(continues on next page)

(continued from previous page)

```
    line, = ax.plot([], [], lw=2)
    lines.append(line)
line, = axs[1].plot([], [], lw=2)
lines.append(line)

def init():
    for line in lines:
        line.set_data([], [])
    return lines

def animate(window):
    data_out, cac_out = window
    for line, data in zip(lines, [data_out, cac_out, cac_1d]):
        line.set_data(np.arange(data.shape[0]), data)
    return lines

anim = animation.FuncAnimation(fig, animate, init_func=init,
                               frames=windows, interval=100,
                               blit=True)

anim_out = anim.to_jshtml()
plt.close()  # Prevents duplicate image from displaying
if os.path.exists("None0000000.png"):
    os.remove("None0000000.png")  # Delete rogue temp file

HTML(anim_out)
# anim.save('/tmp/semantic.mp4')
```

[14]:
```
<IPython.core.display.HTML object>
```

4.4.8 Summary

And that's it! You've just learned the basics of how to identify segments within your data using the matrix profile indices and leveraging `fluss` and `floss`.

4.4.9 Resources

Matrix Profile VIII

STUMPY Documentation

STUMPY Matrix Profile Github Code Repository

4.5 Fast Approximate Matrix Profiles with SCRUMP

In this paper, a new approach called "SCRIMP++", which computes a matrix profile in an incremental fashion, is presented. When only an approximate matrix profile is needed, the this algorithm uses certain properties of the matrix profile calculation to greatly reduce the total computational time and, in this tutorial, we'll demonstrate how this approach may be sufficient for your applications.

`stumpy` implements this approach for both self-joins and AB-joins in the `stumpy.scrump` function and it allows for the matrix profile to be easily refined when a higher resolution output is desired.

4.5.1 Getting started

First, let us import some packages we will use for data loading, analyzing, and plotting.

[1]:
```
%matplotlib inline

import pandas as pd
import stumpy
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle

plt.rcParams["figure.figsize"] = [20, 6]  # width, height
plt.rcParams['xtick.direction'] = 'out'
```

4.5.2 Load the Steamgen Dataset

This data was generated using fuzzy models applied to mimic a steam generator at the Abbott Power Plant in Champaign, IL. The data feature that we are interested in is the output steam flow telemetry that has units of kg/s and the data is "sampled" every three seconds with a total of 9,600 datapoints.

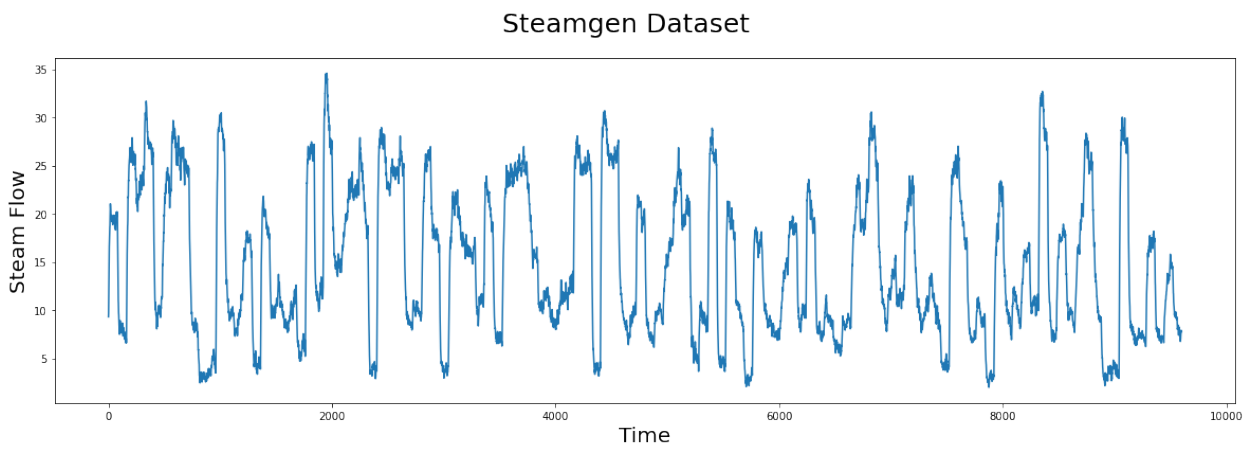
[2]:
```
steam_df = pd.read_csv("https://zenodo.org/record/4273921/files/STUMPY_Basics_
↪steamgen.csv?download=1")
steam_df.head()
```

[2]:

	drum pressure	excess oxygen	water level	steam flow
0	320.08239	2.506774	0.032701	9.302970
1	321.71099	2.545908	0.284799	9.662621
2	320.91331	2.360562	0.203652	10.990955
3	325.00252	0.027054	0.326187	12.430107
4	326.65276	0.285649	0.753776	13.681666

4.5.3 Visualizing the Steamgen Dataset

[3]:
```
plt.suptitle('Steamgen Dataset', fontsize='25')
plt.xlabel('Time', fontsize ='20')
plt.ylabel('Steam Flow', fontsize='20')
plt.plot(steam_df['steam flow'].values)
plt.show()
```

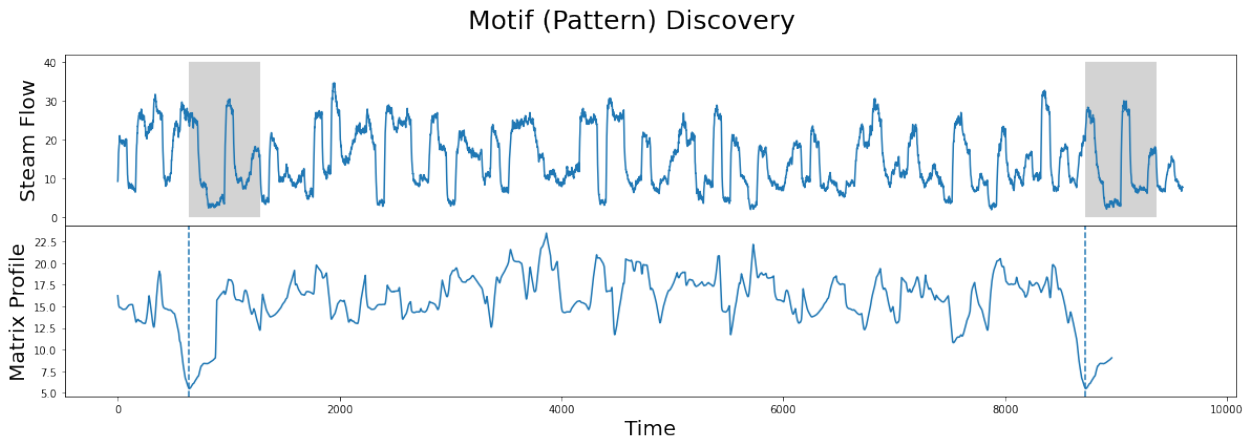


4.5.4 Computing the True Matrix Profile

Now, as a benchmark to compare against, we will compute the full matrix profile using the `stumpy.stump` function along with a window size of `m=640`.

[4]:
```
m = 640
mp = stumpy.stump(steam_df['steam flow'], m)
true_P = mp[:, 0]
```

[5]:
```
fig, axs = plt.subplots(2, sharex=True, gridspec_kw={'hspace': 0})
plt.suptitle('Motif (Pattern) Discovery', fontsize='25')

axs[0].plot(steam_df['steam flow'].values)
axs[0].set_ylabel('Steam Flow', fontsize='20')
rect = Rectangle((643, 0), m, 40, facecolor='lightgrey')
axs[0].add_patch(rect)
rect = Rectangle((8724, 0), m, 40, facecolor='lightgrey')
axs[0].add_patch(rect)
axs[1].set_xlabel('Time', fontsize ='20')
axs[1].set_ylabel('Matrix Profile', fontsize='20')
axs[1].axvline(x=643, linestyle="dashed")
axs[1].axvline(x=8724, linestyle="dashed")
axs[1].plot(true_P)
plt.show()
```



The global minima of the matrix profile, symbolised with the dashed lines, are the indices of the top motif, i.e. the two sub-sequences which are most similar to each other. When using the matrix profile, in many application these are the two most important subsequences, and we will see how we can use `stumpy.scrump` to quickly come up with an approximate matrix profile in a fraction of the time.

Additionally, we will use the helper function below to visually compare the true matrix profile (`true_P` - computed with `stumpy.stump`) with the approximated matrix profile (`approx_P` - computed with `stumpy.scrump`).

[6]:
```
def compare_approximation(true_P, approx_P):
    fig, ax = plt.subplots(gridspec_kw={'hspace': 0})

    ax.set_xlabel('Time', fontsize ='20')
    ax.axvline(x=643, linestyle="dashed")
    ax.axvline(x=8724, linestyle="dashed")
    ax.set_ylim((5, 28))
    ax.plot(approx_P, color='C1', label="Approximate Matrix Profile")
    ax.plot(true_P, label="True Matrix Profile")
```

(continues on next page)

(continued from previous page)

```
    ax.legend()
    plt.show()
```

4.5.5 Computing an Approximate Matrix Profile Using SCRUMP

To calculate the full matrix profile, one has to compute the whole distance matrix (i.e., the pairwise distances between all pairs of subsequences). However, `stumpy.scrump` computes this distance matrix in a diagonal-wise fashion but only using a subset of all diagonals (and, therefore, only using a subset of all distances). How many pairwise distances along the diagonals you want to compute is controlled by the `percentage` argument. The more distance you compute, the better the approximation will be but this also implies a higher computational cost. Choosing a value of `1.0`, or 100% of all distances, produces the full exact matrix profile (equivalent to the output from `stumpy.stump`). It is important to note that even though less pairwise distances are being computed, no pairwise distance is being approximated. That is, you are always guaranteed that `approx_P >= true_P` when `percentage <= 1.0` and, at `percentage=1.0`, `approx_P == true_P` (i.e., it is exact).

Now, let's call `stumpy.scrump` and approximate the full matrix profile by computing only 1% of all distances (i.e., `percentage=0.01`):

[7]:
```
approx = stumpy.scrump(steam_df['steam flow'], m, percentage=0.01, pre_scrump=False)
```

There are a couple of things to notice. First, we passed in a `pre_scrump` argument, a preprocessing step for `stumpy.scrump` that, when set to `True`, can greatly improve the approximation. For now, we turn off the preprocessing step for demonstration purposes and revisit it in the next section. Second, `stumpy.scrump` initializes and returns a `scrump` object and not the matrix profile directly and we will see why this is useful below.

To retrieve the first approximation (i.e., a matrix profile computed from 1% of all distances), we simply call the `.update()` method:

[8]:
```
approx.update()
```

And we can access the updated matrix profile and the matrix profile indices via the `.P_` and `.I_` attributes, respectively:

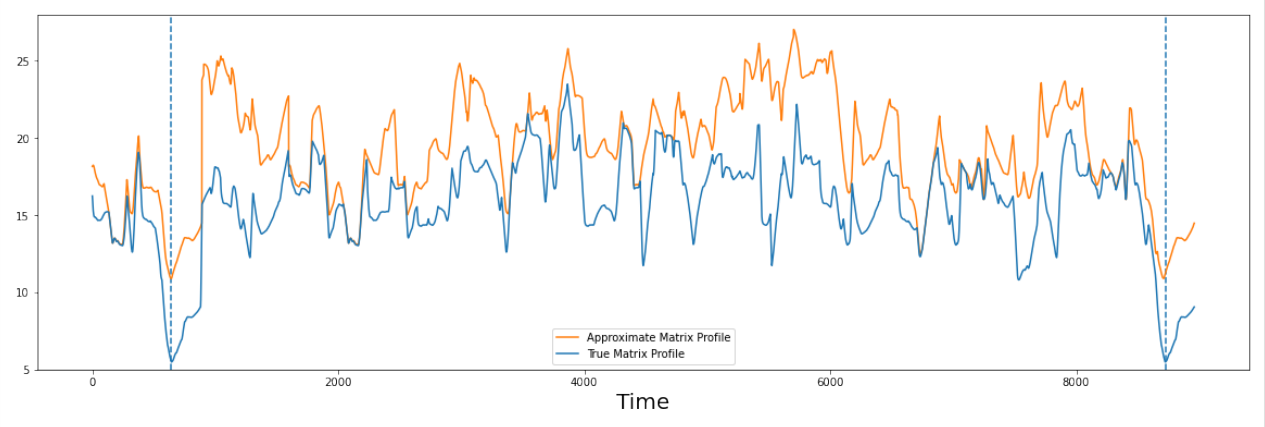
[9]:
```
approx_P = approx.P_
```

Please keep in mind that the approximate matrix profile is computed by randomly computing distances along a subset of diagonals. So, each time you initialize a new `scrump` object by calling `stumpy.scrump`, this will randomly shuffle the order that the distances are computed, which inevitably results in different approximate matrix profiles (except when `percentage=1.0`). Depending on your use case, to ensure reproducible results, you may consider setting random seed prior to calling `stumpy.scrump`:

```
seed = np.random.randint(100000)
np.random.seed(seed)
approx = stumpy.scrump(steam_df['steam flow'], m, percentage=0.01, pre_scrump=False)
```

Next, let's plot `approx_P` on top of the `true_P` to see how well they compare:

[10]:
```
compare_approximation(true_P, approx_P)
```



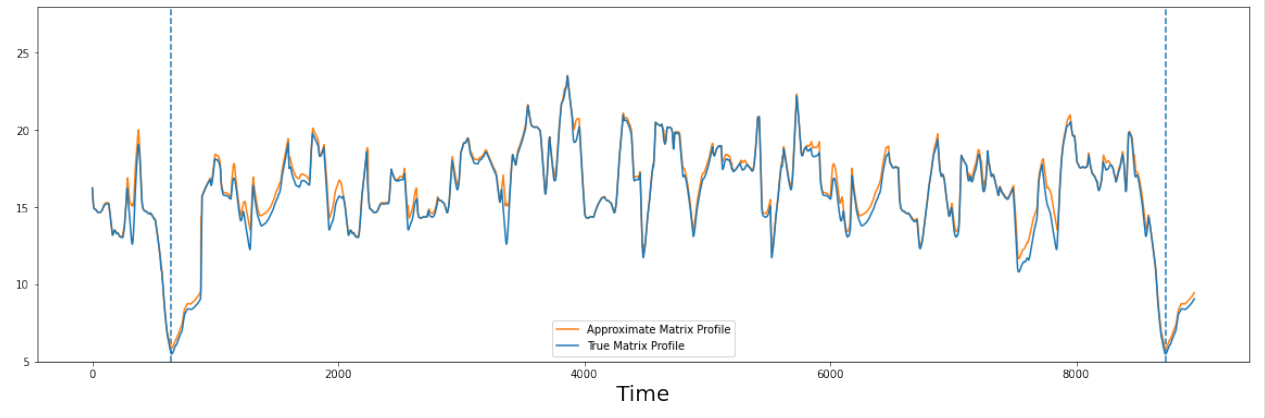
We can see, that this approximation (in orange) is far from perfect but there are some similarities between the two. However, the lowest points in the approximation do not correspond to the true minima yet.

4.5.6 Refining the Matrix Profile

However, we can incrementally refine the approximation by calling `.update()` nine more times (i.e., `10 * 0.01 = 0.10`) , so our new approximated matrix profile will be computed using roughly 10% of all pairwise distances in the full distance matrix.

[11]:
```
for _ in range(9):
    approx.update()

approx_P = approx.P_
```

[12]:
```
compare_approximation(true_P, approx_P)
```



Now, this result is much more convincing and it only required computing just 10% of all of the pairwise distances! We can see that the two profiles are very similar, in particular the important features like global minima are at almost equal, if not equal, positions. For most applications this is enough, as an offset of a few points usually does not matter and the number of distances that had to be calculated was reduced by a factor of ten! In fact, we can do even better this!

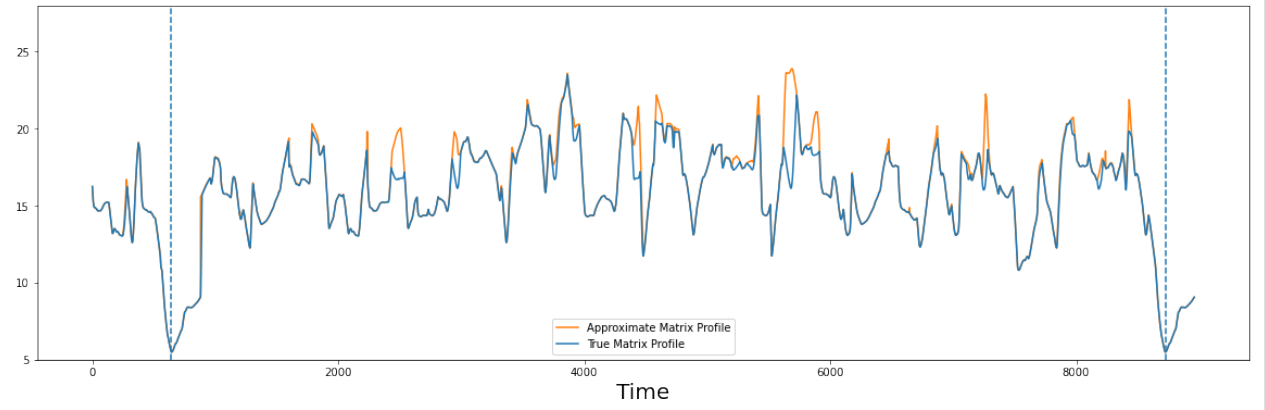
4.5.7 The Power of Preprocessing

Until now, we only ran `stumpy.scrump` without the powerful "PRESCRIMP" preprocessing step. "PRESCRIMP" is an algorithm that preprocesses the time series data with complexity `O(n log(n) / s)`, where `n` is the number of points and `s` is called the sampling rate. `stumpy.stump` and `stumpy.scrump` (without PRESCRIMP) are both `O(n^2)` complexity so, in general, preprocessing is 'cheap'. PRESCRIMP already computes the distances of some pairwise distances and the sampling rate controls how many. Typically, a good value is `s=m/4`, the same size as the typical exclusion zone, and this value is used if you pass `None` to the `scrimp.scrump` function call.

Below we will approximate the matrix profile again with 1% of all diagonals but, this time, we'll enable preprocessing by setting `pre_scrimp=True`. Obviously, this will take a bit longer to compute, since a few more calculations have to be performed:

[13]:
```
approx = stumpy.scrump(steam_df['steam flow'], m, percentage=0.01, pre_scrump=True,␣
↪s=None)
approx.update()
approx_P = approx.P_
```

[14]:
```
compare_approximation(true_P, approx_P)
```



However, one can see that in this example, with preprocessing followed by computing only 1% of the pairwise distances (i.e., calling `.update()` only once), the approximate matrix profile and the true matrix profile are nearly visually indistinguishable. With increasing time series length, the benefit of using preprocessing grows even further. Of course, depending on the size of the time series data that you need to analyze and the available computing resources at your disposal, it may be worth it to compute a higher percentage of distances to be certain that the approximation is converged.

4.5.8 Summary

That's it! You have learned how to approximate the matrix profile using `stumpy.scrump` and hopefully can use the approximated matrix profile in your application.

4.5.9 Resources

Matrix Profile XI

STUMPY Documentation

STUMPY Matrix Profile Github Code Repository

4.6 Incremental Matrix Profiles for Streaming Time Series Data

Now that you have a basic understanding of how to compute a matrix profile, in this short tutorial, we will demonstrate how to incrementally update your matrix profile when you have streaming (on-line) data using the `stumpy.stumpi` ("STUMP Incremental") function. You can learn more about the details of this approach by reading Section G of the Matrix Profile I paper and Section 4.6 and Table 5 this paper.

4.6.1 Getting Started

Let's import the packages that we'll need to create and analyze a randomly generated time series data set.

[1]:
```
import numpy as np
import stumpy
import numpy.testing as npt
import time
```

4.6.2 Generating Some Random Time Series Data

Imagine that we have an IoT sensor that has been collecting data once an hour for the last 14 days. That would mean that we've amassed `14 * 24 = 336` data points up until this point and our data set might look like this:

[2]:
```
T = np.random.rand(336)
```

And, perhaps, we know from experience that an interesting motif or anomaly might be detectable within a 12 hour (sliding) time window:

[3]:
```
m = 12
```

4.6.3 Typical Batch Analysis

To compute the matrix profile using a batch process is straightforward using `stumpy.stump`:

[4]:
```
mp = stumpy.stump(T, m)
```

But as the length of `T` grows with each passing hour, it will take increasingly more time to compute the matrix profile since `stumpy.stump` will actually re-compute all of the pairwise distances between all subsequences within the time series. This is super time consuming! Instead, for streaming data, we want to find a way to take the new incoming (single) data point and compare the subsequence that it resides in with the rest of the time series (i.e., compute the distance profile) and update the existing matrix profile. Luckily, this can be easily accomplished with `stumpy.stumpi` or "STUMP Incremental".

4.6.4 Streaming (On-line) Analysis with STUMPI

As we wait for the next data point, `t`, to arrive, we can take our existing data initialize our streaming object:

[5]:
```
stream = stumpy.stumpi(T, m, egress=False)  # Don't egress/remove the oldest data␣
↪point when streaming
```

And when a new data point, `t`, arrives:

[6]:
```
t = np.random.rand()
```

We can append `t` to our `stream` and easily update the matrix profile, `P`, and matrix profile indices, `I` behind the scenes:

[7]:
```
stream.update(t)
```

In the background, `t` has been appended to the existing time series and it automatically compares the new subsequence with all of the existing ones and updates the historical values. It also determines which one of the existing subsequences is the nearest neighbor to the new subsequence and appends this information to the matrix profile. And this can continue on, say, for another 1,000 iterations (or indefinitely) as additional data is streamed in:

[8]:
```
for i in range(1000):
    t = np.random.rand()
    stream.update(t)
```

It is important to reiterate that incremental `stumpy.stumpi` is different from batch `stumpy.stump` in that it does not waste any time re-computing any of the past pairwise distances. `stumpy.stumpi` only spends time computing new distances and then updates the appropriate arrays where necessary and, thus, it is really fast!

4.6.5 Validation

The Matrix Profile

Now, this claim of "fast updating" with streaming (on-line) data may feel strange or seem magical so, first, let's validate that the output from incremental `stumpy.stumpi` is the same as performing batch `stumpy.stump`. Let's start with the full time series with `64` data points and compute the full matrix profile:

[9]:
```
T_full = np.random.rand(64)
m = 8

mp = stumpy.stump(T_full, m)
P_full = mp[:, 0]
I_full = mp[:, 1]
```

Next, for `stumpy.stumpi`, we'll only start with the first `10` elements from the full length time series and then incrementally stream in the additional data points one at a time:

[10]:
```
# Start with half of the full length time series and initialize inputs
T_stream = T_full[:10].copy()
stream = stumpy.stumpi(T_stream, m, egress=False)  # Don't remove/egress the oldest␣
↪data point when streaming

# Incrementally add one new data point at a time and update the matrix profile
for i in range(len(T_stream), len(T_full)):
    t = T_full[i]
    stream.update(t)
```

Now that we're done, let's check and validate that:

1. `stream.T == T_full`
2. `stream.P == P_full`
3. `stream.I == I_full`

[11]:
```
npt.assert_almost_equal(stream.T_, T_full)
npt.assert_almost_equal(stream.P_, P_full)
npt.assert_almost_equal(stream.I_, I_full)
```

There are no errors! So, this means that `stump.stumpi` indeed produces the correct matrix profile results that we'd expect.

The Performance

We've basically claimed that incrementally updating our matrix profile with `stumpy.stumpi` is much faster (in total computational time) than performing a full pairwise distance calculation with `stumpy.stump` as each new data point arrives. Let's actually compare the timings by taking a full time series that is 10,000 data points in length and we initialize both approaches with the first 2% of the time series (i.e., the first 200 points) and append a single new data point at each iteration before re-computing the matrix profile:

[12]:
```
T_full = np.random.rand(10_000)
T_stream = T_full[:200].copy()
m = 100

# `stumpy.stump` timing
start = time.time()
mp = stumpy.stump(T_stream, m)
for i in range(200, len(T_full)):
    T_stream = np.append(T_stream, T_full[i])
    mp = stumpy.stump(T_stream, m)
stump_time = time.time() - start

# `stumpy.stumpi` timing
stream = stumpy.stumpi(T_stream, m, egress=False)  # Don't egress/remove the oldest␣
↪data point when streaming
start = time.time()
for i in range(200, len(T_full)):
    t = T_full[i]
    stream.update(t)
stumpi_time = time.time() - start

print(f"stumpy.stump: {np.round(stump_time,1)}s")
print(f"stumpy.stumpi: {np.round(stumpi_time, 1)}s")
```

```
stumpy.stump: 799.8s
stumpy.stumpi: 14.8s
```

Setting aside the fact that having more CPUs will speed up both approaches, we clearly see that incremental `stumpy.stumpi` is one to two orders of magnitude faster than batch `stumpy.stump` for processing streaming data. In fact for the current hardware, on average, it is taking roughly 0.1 seconds for `stumpy.stump` to analyze each new matrix profile. So, if you have more than 10 new data point arriving every second, then you wouldn't be able to keep up. In contrast, `stumpy.stumpi` should be able to comfortably handle and process ~450+ new data points per second using fairly modest hardware. Additionally, batch `stumpy.stump`, which has a computational complexity of `O(n^2)`, will get even slower as more and more data points get appended to the existing time series while `stumpy.stumpi`, which is essentially `O(1)`, will continue to be highly performant.

In fact, if you don't care about maintaining the oldest data point and its relationships with the newest data point (i.e., you only care about maintaining a fixed sized sliding window), then you can get slightly improve the performance by telling `stumpy.stumpi` to remove/egress the oldest data point (along with its corresponding matrix profile information) by setting the parameter `egress=True` when we instantiate our streaming object (note that this is actually the default behavior):

[13]:
```
stream = stumpy.stumpi(T_stream, m, egress=True)  # Egressing/removing the oldest␣
↪data point is the default behavior!
```

And now, when we process the same data above:

[14]:
```
# `stumpy.stumpi` timing with egress
stream = stumpy.stumpi(T_stream, m, egress=True)
start = time.time()
for i in range(200, len(T_full)):
    t = T_full[i]
    stream.update(t)
stumpi_time = time.time() - start

print(f"stumpy.stumpi: {np.round(stumpi_time, 1)}s")
```

```
stumpy.stumpi: 12.1s
```

4.6.6 A Visual Example

Now that we understand how to compute and update our matrix profile with streaming data, let's explore this with a real example data set where there is a known pattern and see if `stumpy.stumpi` can correctly identify when the global pattern (motif) is encountered.

Retrieving and Loading the Data

First let's import some additional Python packages and then retrieve our standard "Steamgen Dataset":

[15]:
```
%matplotlib inline

import pandas as pd
import stumpy
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle
from matplotlib import animation
from IPython.display import HTML
import os

plt.rcParams["figure.figsize"] = [20, 6]  # width, height
plt.rcParams['xtick.direction'] = 'out'

steam_df = pd.read_csv("https://zenodo.org/record/4273921/files/STUMPY_Basics_
↪steamgen.csv?download=1")
steam_df.head()
```

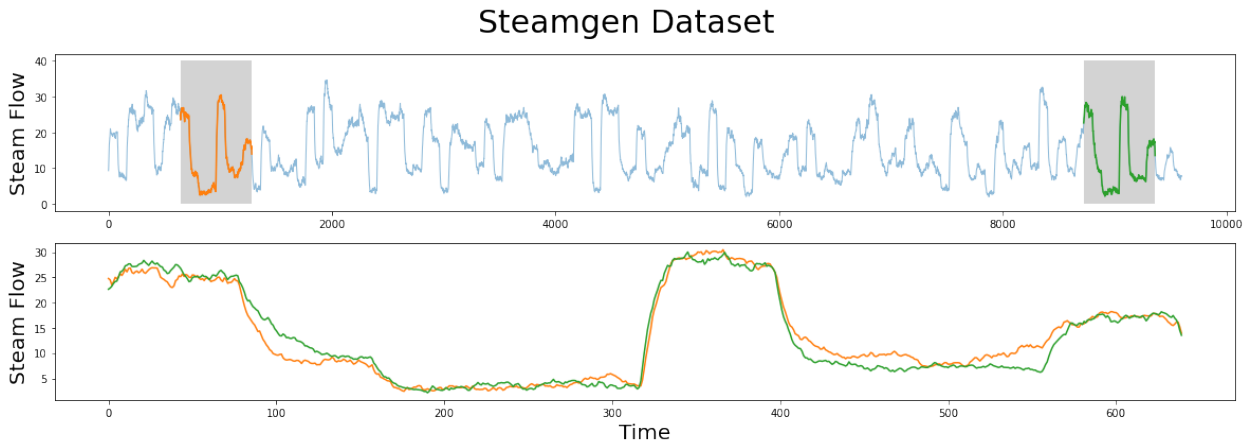
[15]:

	drum pressure	excess oxygen	water level	steam flow
0	320.08239	2.506774	0.032701	9.302970
1	321.71099	2.545908	0.284799	9.662621
2	320.91331	2.360562	0.203652	10.990955
3	325.00252	0.027054	0.326187	12.430107
4	326.65276	0.285649	0.753776	13.681666

This data was generated using fuzzy models applied to mimic a steam generator at the Abbott Power Plant in Champaign, IL. The data feature that we are interested in is the output steam flow telemetry that has units of kg/s and the data is "sampled" every three seconds with a total of 9,600 datapoints.

The motif (pattern) that we are looking for is highlighted below and yet it is still very hard to be certain that the orange and green subsequences are a match, that is, until we zoom in on them and overlay the subsequences on top each other. Now, we can clearly see that the motif is very similar!

[16]:
```
m = 640
fig, axs = plt.subplots(2)
plt.suptitle('Steamgen Dataset', fontsize='30')
axs[0].set_ylabel("Steam Flow", fontsize='20')
axs[0].plot(steam_df['steam flow'], alpha=0.5, linewidth=1)
axs[0].plot(steam_df['steam flow'].iloc[643:643+m])
axs[0].plot(steam_df['steam flow'].iloc[8724:8724+m])
rect = Rectangle((643, 0), m, 40, facecolor='lightgrey')
axs[0].add_patch(rect)
rect = Rectangle((8724, 0), m, 40, facecolor='lightgrey')
axs[0].add_patch(rect)
axs[1].set_xlabel("Time", fontsize='20')
axs[1].set_ylabel("Steam Flow", fontsize='20')
axs[1].plot(steam_df['steam flow'].values[643:643+m], color='C1')
axs[1].plot(steam_df['steam flow'].values[8724:8724+m], color='C2')
plt.show()
```



Using STUMPI

Now, let's take a look at what happens to the matrix profile when we initialize our `stumpy.stumpi` with the first 2,000 data points:

[17]:
```
T_full = steam_df['steam flow'].values
T_stream = T_full[:2000]
stream = stumpy.stumpi(T_stream, m, egress=False)  # Don't egress/remove the oldest␣
↪data point when streaming
```

and then incrementally append a new data point and update our results:

[18]:
```
windows = [(stream.P_, T_stream)]
P_max = -1
for i in range(2000, len(T_full)):
    t = T_full[i]
    stream.update(t)

    if i % 50 == 0:
```

(continues on next page)

(continued from previous page)

```
        windows.append((stream.P_, T_full[:i+1]))
        if stream.P_.max() > P_max:
            P_max = stream.P_.max()
```

When we plot the growing time series (upper panel), `T_stream`, along with the matrix profile (lower panel), `P`, we can watch how the matrix profile evolves as new data is appended:

[19]:
```
fig, axs = plt.subplots(2, sharex=True, gridspec_kw={'hspace': 0})

rect = Rectangle((643, 0), m, 40, facecolor='lightgrey')
axs[0].add_patch(rect)
rect = Rectangle((8724, 0), m, 40, facecolor='lightgrey')
axs[0].add_patch(rect)
axs[0].set_xlim((0, T_full.shape[0]))
axs[0].set_ylim((-0.1, T_full.max()+5))
axs[1].set_xlim((0, T_full.shape[0]))
axs[1].set_ylim((-0.1, P_max+5))
axs[0].axvline(x=643, linestyle="dashed")
axs[0].axvline(x=8724, linestyle="dashed")
axs[1].axvline(x=643, linestyle="dashed")
axs[1].axvline(x=8724, linestyle="dashed")
axs[0].set_ylabel("Steam Flow", fontsize='20')
axs[1].set_ylabel("Matrix Profile", fontsize='20')
axs[1].set_xlabel("Time", fontsize='20')


lines = []
for ax in axs:
    line, = ax.plot([], [], lw=2)
    lines.append(line)
line, = axs[1].plot([], [], lw=2)
lines.append(line)

def init():
    for line in lines:
        line.set_data([], [])
    return lines

def animate(window):
    P, T = window
    for line, data in zip(lines, [T, P]):
        line.set_data(np.arange(data.shape[0]), data)

    return lines

anim = animation.FuncAnimation(fig, animate, init_func=init,
                               frames=windows, interval=100,
                               blit=True, repeat=False)

anim_out = anim.to_jshtml()
plt.close()  # Prevents duplicate image from displaying
if os.path.exists("None0000000.png"):
    os.remove("None0000000.png")  # Delete rogue temp file

HTML(anim_out)
# anim.save('/tmp/stumpi.mp4')
```

[19]:
```
<IPython.core.display.HTML object>
```

Here, the vertical dotted lines mark the position of where the global motif pair is expected to be and the gray box emphasizes the corresponding motif subsequences. As you play through this animation you may notice that the matrix profile is constantly changing since past subsequences may find a new nearest neighbor. However, note that any change in the matrix profile can only move downward (toward zero). Throughout most of this animation, the subsequence highlighted on the left (grey box) has a relatively high matrix profile value. However, as the time series extends past the grey box on the right, the aforementioned matrix profile value drops significantly and stabilizes quickly as soon as its nearest neighbor has fully arrived in the stream. This is really cool! In fact, the authors of the original Matrix Profile I paper point out that, on this dataset, it would be possible to continue monitoring the matrix profile with `stumpy.stumpi` for several decades before running out of time or memory!

4.6.7 Bonus Section - Never Update History

Above, we've gone with the typical definition of a matrix profile. That is, for any given subsequence, `T[i : i + m]`, find the distance to its nearest neighbor, `T[j : j + m]`, regardless of whether `j` is to the left of `i` (i.e., `j < i`) or to the right of `i` (i.e., `j > i`). This means that as new data comes in, even the matrix profiles for the past historical data points also get updated if a "new" nearest neighbor has revealed itself. Essentially, this is "hindsight". So, there may be a case where, the first time you see a unique subsequence, you may identify it as an anomaly due to its relatively high matrix profile value. However, as more and more new data arrive, this originally anomalous subsequence may no longer be unique anymore. Consider observing only the first period of a sine wave, all of these subsequence would be unique. But as the next period of the sine wave starts to stream in, we realize that the data points in the first period are no longer anomalous so we update their matrix profile values accordingly.

Now, this may or may not be beneficial depending on how you choose to define an "anomaly". In fact, you may choose not to update the matrix profiles from the past and you may want to restrict the search for a nearest neighbor, `j`, to always be to the left of `i` (i.e., `j < i`). Luckily, in `stumpy.stumpi` this is already done for you and you can access the left matrix profile and the left matrix profile indices via the `.left_P` and the `.left_I` attributes, respectively, of your streaming object:

[20]:
```
T_full = np.random.rand(64)
m = 8

T_stream = T_full[:10].copy()
stream = stumpy.stumpi(T_stream, m, egress=False)  # Don't egress/remove the oldest␣
↪data point when streaming. To egress, set `egress=True`

for i in range(len(T_stream), len(T_full)):
    t = T_full[i]
    stream.update(t)

print(f"Full Matrix Profile: {np.round(stream.P_, 2)}")
print(f"Left Matrix Profile: {np.round(stream.left_P_, 2)}")
print(f"Full Matrix Profile Indices: {stream.I_}")
print(f"Left Matrix Profile Indices: {stream.left_I_}")
```

```
Full Matrix Profile: [1.62 1.58 1.67 1.12 1.58 1.75 1.53 1.77 1.32 1.83 1.23 1.05 1.
↪88 1.74
 1.61 1.74 1.75 1.66 1.73 1.12 1.05 1.31 1.59 1.46 2.02 2.29 1.32 1.58
 1.67 1.29 1.05 1.31 1.53 1.77 2.41 1.96 1.96 2.09 1.84 1.23 1.05 1.88
 1.74 1.42 1.74 1.75 1.73 2.1  1.99 1.92 1.75 1.46 2.11 2.41 1.82 1.71
 1.5 ]
Left Matrix Profile: [ inf  inf  inf 4.31 3.63 3.65 3.66 3.57 2.13 2.41 2.38 2.04 2.
↪89 1.91
 2.17 2.94 2.37 2.04 2.04 1.12 1.58 2.28 2.37 2.81 2.48 2.32 1.32 1.58
```

(continues on next page)

(continued from previous page)

```
 1.67 1.29 1.05 1.31 1.53 1.77 2.44 1.96 1.98 2.09 1.89 1.23 1.05 1.88
 1.74 1.42 1.74 1.75 1.73 2.1  1.99 1.92 1.75 1.46 2.11 2.41 1.82 1.71
 1.5 ]
Full Matrix Profile Indices: [26 27 28 19 20 50 32 33 26 42 39 40 41 42 43 44 45 27␣
↪40  3 30 31 32 51
 35 46  8  1  2 19 20 21  6  7 44 13 46 15 42 10 11 12 13 10 15 16 17 18
 29  4  5 23 24 25  8  1 31]
Left Matrix Profile Indices: [-1 -1 -1  0  0  1  2  2  0  0  1  2  8  9 10 11  8  1 ␣
↪2  3  4  5  6 12
 14 15  8  1  2 19 20 21  6  7 12 13 14 15 16 10 11 12 13 10 15 16 17 18
 29  4  5 23 24 25  8  1 31]
```

Of course, it is important to point out that a `-1` value in the left matrix profile indices does not correspond to the last subsequence in the time series. Instead, it means that the subsequence in that position has no valid nearest neighbor to its left. Consequently, the corresponding left matrix profile value will be set to `np.inf`.

4.6.8 Summary

And that's it! You've just learned how to incrementally update your matrix profile for streaming (on-line) data.

4.6.9 Resources

Matrix Profile I

Time Series Joins, Motifs, Discords and Shapelets: A Unifying View that Exploits the Matrix Profile (see Section 4.6 and Table 5)

STUMPY Documentation

STUMPY Matrix Profile Github Code Repository

4.7 Fast Pattern Searching

4.7.1 Beyond Matrix Profiles

At the core of STUMPY, one can take any time series data and efficiently compute something called a matrix profile, which essentially scans along your entire times series with a fixed window size, `m`, and finds the exact nearest neighbor for every subsequence within your time series. A matrix profile allows you to determine if there are any conserved behaviors (i.e., conserved subsequences/patterns) within your data and, if so, it can tell you exactly where they are located within your time series. In a previous tutorial, we demonstrated how to use STUMPY to easily obtain a matrix profile, learned how to interpret the results, and discover meaningful motifs and discords. While this brute-force approach may be very useful when you don't know what pattern or conserved behavior you are looking but, for sufficiently large datasets, it can become quite expensive to perform this exhaustive pairwise search.

However, if you already have a specific user defined pattern in mind then you don't actually need to compute the full matrix profile! For example, maybe you've identified an interesting trading strategy based on historical stock market data and you'd like to see if that specific pattern may have been observed in the past within one or more stock ticker symbols. In that case, searching for a known pattern or "query" is actually quite straightforward and can be accomplished quickly by using the wonderful `core.mass` function in STUMPY.

In this short tutorial, we'll take a simple known pattern of interest (i.e., a query subsequence) and we'll search for this pattern in a separate independent time series. Let's get started!

4.7.2 Getting Started

Let's import the packages that we'll need to load, analyze, and plot the data

[1]:
```
%matplotlib inline

import pandas as pd
import stumpy
import numpy as np
import numpy.testing as npt
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle

plt.rcParams["figure.figsize"] = [20, 6]  # width, height
plt.rcParams['xtick.direction'] = 'out'
```

4.7.3 Loading the Sony AIBO Robot Dog Dataset

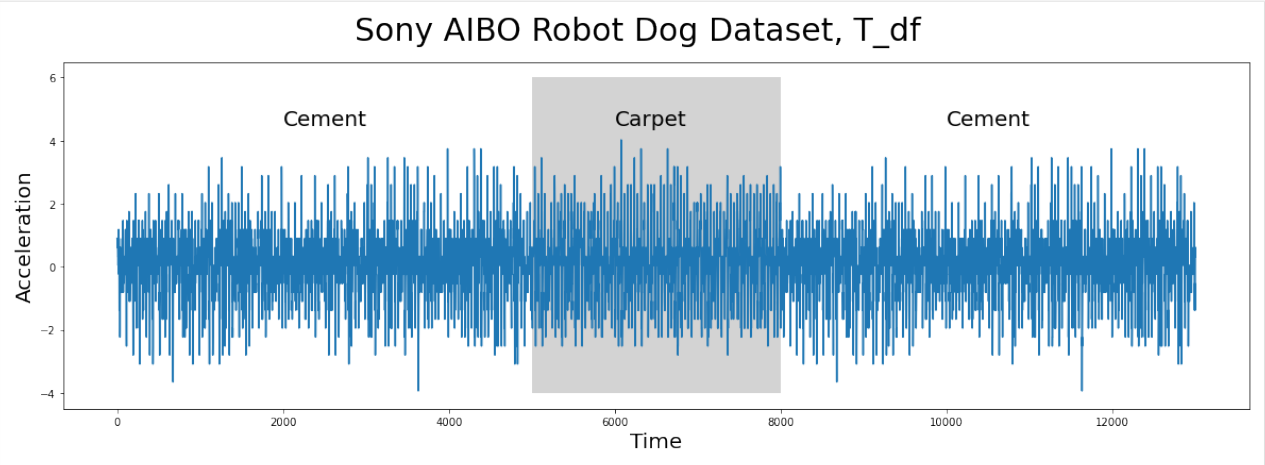
The time series data (below), `T_df`, has `n = 13000` data points and it was collected from an accelerometer inside of a Sony AIBO robot dog where it tracked the robot dog when it was walking from a cement surface onto a carpeted surface and, finally, back to the cement surface:

[2]:
```
T_df = pd.read_csv("https://zenodo.org/record/4276393/files/Fast_Pattern_Searching_
↪robot_dog.csv?download=1")
T_df.head()
```

[2]:
```
   Acceleration
0       0.89969
1       0.89969
2       0.89969
3       0.89969
4       0.89969
```

4.7.4 Visualizing the Sony AIBO Robot Dog Dataset

[3]:
```
plt.suptitle('Sony AIBO Robot Dog Dataset, T_df', fontsize='30')
plt.xlabel('Time', fontsize ='20')
plt.ylabel('Acceleration', fontsize='20')
plt.plot(T_df)
plt.text(2000, 4.5, 'Cement', color="black", fontsize=20)
plt.text(10000, 4.5, 'Cement', color="black", fontsize=20)
ax = plt.gca()
rect = Rectangle((5000, -4), 3000, 10, facecolor='lightgrey')
ax.add_patch(rect)
plt.text(6000, 4.5, 'Carpet', color="black", fontsize=20)
plt.show()
```

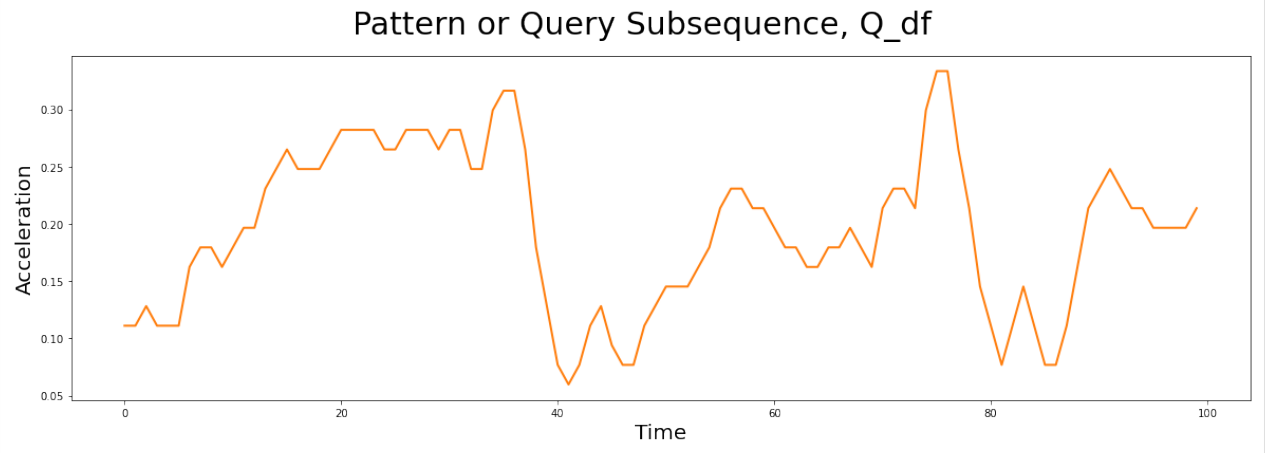


In the plot above, the periods of time when the robot dog was walking on cement is displayed with a white background while the times when the robot dog was walking on carpet is highlighted with a grey background. Do you notice any appreciable difference(s) between walking on the different surfaces? Are there any interesting insights that you can observe with the human eye? Do any conserved patterns exist within this time series and, if so, where are they?

4.7.5 Have You Seen This Pattern?

The subsequence pattern or query (below) that we are interested in searching for in the time series (above) looks like this:

[4]:
```
Q_df = pd.read_csv("https://zenodo.org/record/4276880/files/carpet_query.csv?
↪download=1")

plt.suptitle('Pattern or Query Subsequence, Q_df', fontsize='30')
plt.xlabel('Time', fontsize ='20')
plt.ylabel('Acceleration', fontsize='20')
plt.plot(Q_df, lw=2, color="C1")  # Walking on cement
plt.show()
```



This pattern, `Q_df`, has a window length of `m = 100` and it was taken from a completely independent walking sample. Does it look familiar at all? Does a similar pattern exist in our earlier time series, `T_df`? Can you tell which surface the robot dog was walking on when this query sample was collected?

To answer some of these questions, you can compare this specific query subsequence or pattern with the full time

series by computing something called a "distance profile". Essentially, you take this single query, `Q_df`, and compare it to every single subsequence in `T_df` by computing all possible (z-normalized Euclidean) pairwise distances. So, the distance profile is simply a 1-dimensional vector that tells you exactly how similar/dissimilar `Q_df` is to every subsequence (of the same length) found in `T_df`. Now, a naive algorithm for computing the distance profile would take `O(n*m)` time to process but, luckily, we can do much better than this as there exists a super efficient approach called "Mueen's Algorithm for Similarity Search" (MASS) that is able to compute the distance profile in much faster `O(n*log(n))` time (`log` base 2). Now, this may not be a big deal if you only have a few short time series to analyze but if you need to repeat this process many times with different query subsequences then things can add up quickly. In fact, as the length of the time series, `n`, and/or the length of the query subsequence, `m`, gets much longer, the naive algorithm would take way too much time!

4.7.6 Computing the Distance Profile with MASS

So, given a query subsequence, `Q_df`, and a time series, `T_df`, we can perform a fast similarity search and compute a distance profile using the `core.mass` function in STUMPY:

[5]:
```
distance_profile = stumpy.core.mass(Q_df["Acceleration"], T_df["Acceleration"])
```

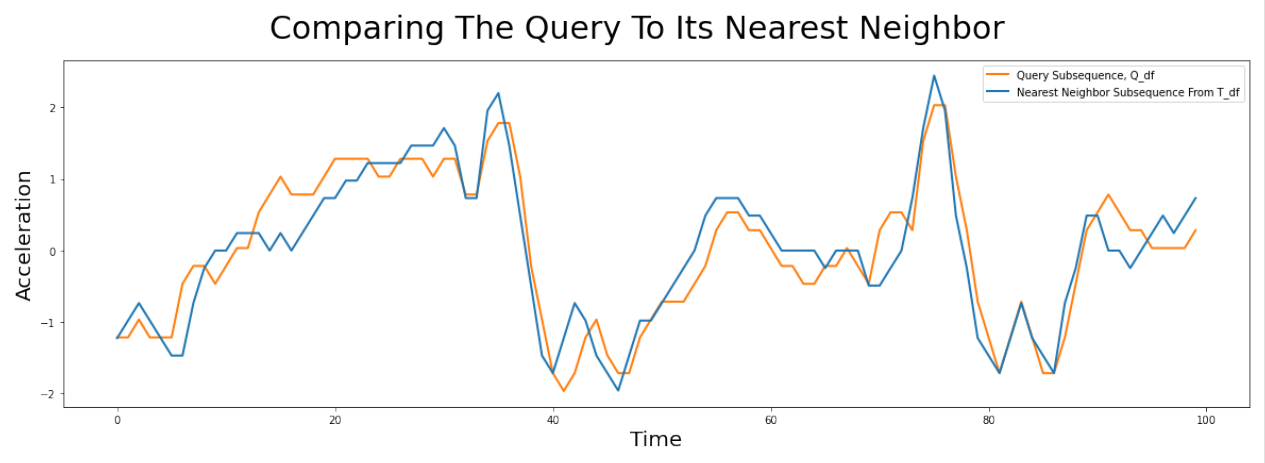
And, since the `distance_profile` contains the full list of pairwise distances between `Q_df` and every subsequence within `T_df`, we can retrieve the most similar subsequence from `T_df` by finding the smallest distance value in `distance_profile` and extracting its positional index:

[6]:
```
idx = np.argmin(distance_profile)

print(f"The nearest neighbor to `Q_df` is located at index {idx} in `T_df`")
```

```
The nearest neighbor to `Q_df` is located at index 7479 in `T_df`
```

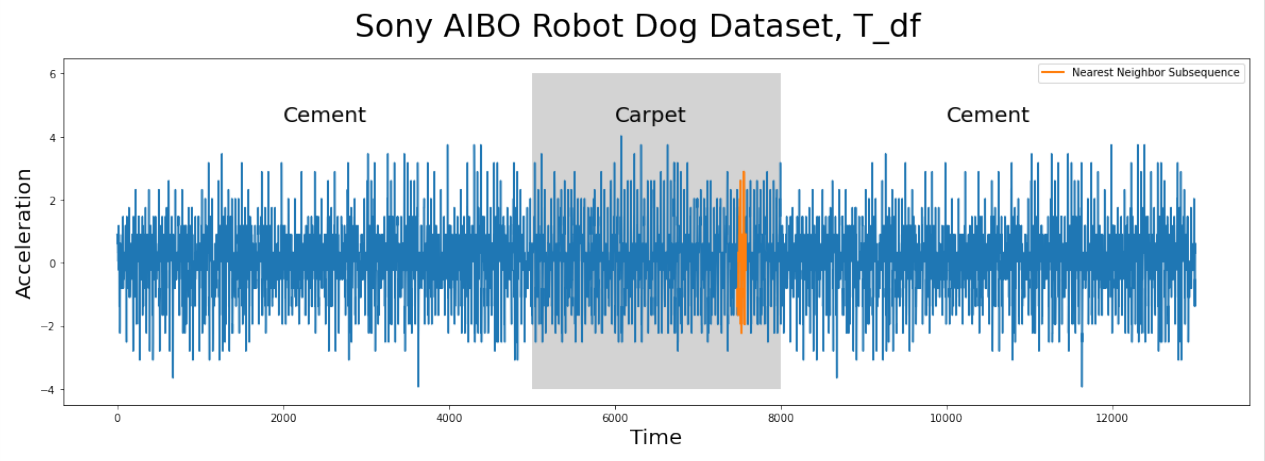
So, to answer our earlier question of "Does a similar pattern exist in our earlier time series, T_df?", let's go ahead and plot the most similar subsequence in `T_df`, which is located at index 7479 (blue), and overlay this with our query pattern, `Q_df`, (orange):

[7]:
```
# Since MASS computes z-normalized Euclidean distances, we should z-normalize our␣
↪subsequences before plotting
Q_z_norm = stumpy.core.z_norm(Q_df.values)
T_z_norm = stumpy.core.z_norm(T_df.values[idx:idx+len(Q_df)])

plt.suptitle('Comparing The Query To Its Nearest Neighbor', fontsize='30')
plt.xlabel('Time', fontsize ='20')
plt.ylabel('Acceleration', fontsize='20')
plt.plot(Q_z_norm, lw=2, color="C1", label="Query Subsequence, Q_df")
plt.plot(T_z_norm, lw=2, label="Nearest Neighbor Subsequence From T_df")
plt.legend()
plt.show()
```



Notice that even though the query subsequence does not perfectly match its nearest neighbor, STUMPY was still able to find it! And then, to answer the second question of "Can you tell which surface the robot dog was walking on when this query sample was collected?", we can look at precisely where `idx` is located within `T_df`:

[8]:
```
plt.suptitle('Sony AIBO Robot Dog Dataset, T_df', fontsize='30')
plt.xlabel('Time', fontsize ='20')
plt.ylabel('Acceleration', fontsize='20')
plt.plot(T_df)
plt.text(2000, 4.5, 'Cement', color="black", fontsize=20)
plt.text(10000, 4.5, 'Cement', color="black", fontsize=20)
ax = plt.gca()
rect = Rectangle((5000, -4), 3000, 10, facecolor='lightgrey')
ax.add_patch(rect)
plt.text(6000, 4.5, 'Carpet', color="black", fontsize=20)
plt.plot(range(idx, idx+len(Q_df)), T_df.values[idx:idx+len(Q_df)], lw=2, label=
↪"Nearest Neighbor Subsequence")
plt.legend()
plt.show()
```

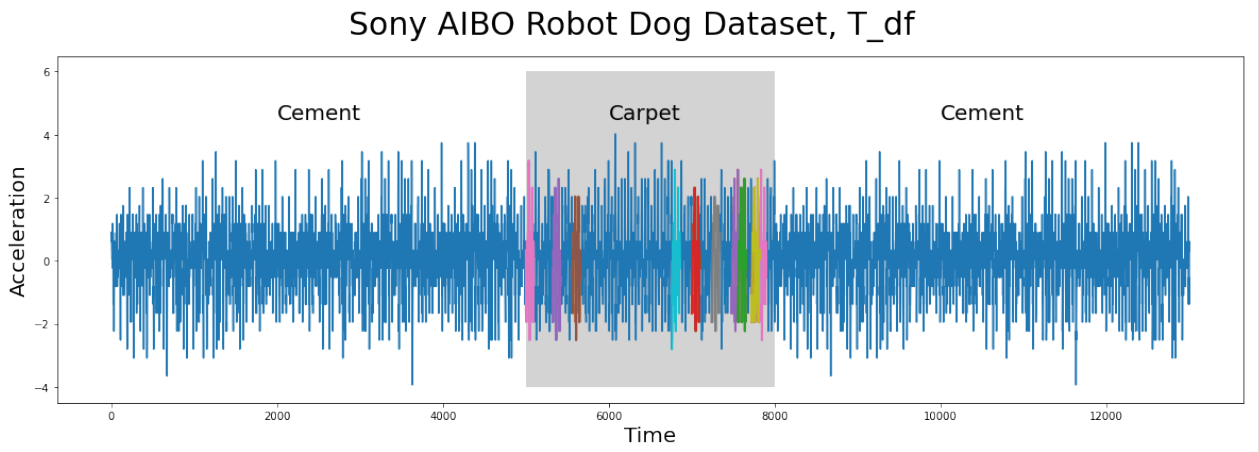


As we can see above, the nearest neighbor (orange) to `Q_df` is a subsequence that is found when the robot dog was walking on carpet and, as it turns out, the `Q_df` was collected from an independent sample where the robot dog was walking on carpet too! To take this a step further, instead of extracting the only top nearest neighbor, we can look at where the top `k = 16` nearest neighbors are located:

[9]:
```
# This simply returns the (sorted) positional indices of the top 16 smallest␣
↪distances found in the distance_profile
k = 16
idxs = np.argpartition(distance_profile, k)[:k]
idxs = idxs[np.argsort(distance_profile[idxs])]
```

And then let's plot all of these subsequences based on their index locations:

[10]:
```
plt.suptitle('Sony AIBO Robot Dog Dataset, T_df', fontsize='30')
plt.xlabel('Time', fontsize ='20')
plt.ylabel('Acceleration', fontsize='20')
plt.plot(T_df)
plt.text(2000, 4.5, 'Cement', color="black", fontsize=20)
plt.text(10000, 4.5, 'Cement', color="black", fontsize=20)
ax = plt.gca()
rect = Rectangle((5000, -4), 3000, 10, facecolor='lightgrey')
ax.add_patch(rect)
plt.text(6000, 4.5, 'Carpet', color="black", fontsize=20)

for idx in idxs:
    plt.plot(range(idx, idx+len(Q_df)), T_df.values[idx:idx+len(Q_df)], lw=2)
plt.show()
```



Unsurprisingly, the top `k = 16` nearest neighbors to `Q_df` (or best matches, shown in multiple colors above) can all be found when the robot dog was walking on the carpet (grey)!

4.7.7 Summary

And that's it! You have now taken a known pattern of interest (or query), ran it through `core.mass` using STUMPY, and you were able to quickly search for this pattern in another time series. With this newfound knowledge, you can now go and search for patterns in your own time series projects. Happy coding!

4.7.8 Additional Note - Distance Profiles with Non-normalized Euclidean Distances

There are times when you may want to use non-normalized Euclidean distance as your measure of similarity/dissimilarity, and so, instead of using `core.mass` (which z-normalizes your subsequences first before computing the pairwise Euclidean distances), you can use the `core.mass_absolute` function. This is provided for those who are interested in computing the non-normalized matrix profiles that are available in the complementary `stumpy.aamp`, `stumpy.aamped`, `stumpy.gpu_aamp`, and `stumpy.ammpi` functions.

4.7.9 Bonus Section - What Makes MASS So Fast?

The reason why MASS is so much faster than a naive approach is because MASS uses Fast Fourier Transforms (FFT) to convert the data into the frequency domain and performs what is called a "convolution", which reduces the `m` operations down to `log(n)` operations. You can read more about this in the original Matrix Profile I paper.

Here's a naive implementation of computing a distance profile:

[11]:
```
def compute_naive_distance_profile(Q, T):
    Q = Q.copy()
    T = T.copy()
    n = len(T)
    m = len(Q)
    naive_distance_profile = np.empty(n - m + 1)

    start = time.time()
    Q = stumpy.core.z_norm(Q)
    for i in range(n - m + 1):
        naive_distance_profile[i] = np.linalg.norm(Q - stumpy.core.z_norm(T[i:i+m]))
    naive_elapsed_time = time.time()-start

    print(f"For n = {n} and m = {m}, the naive algorithm takes {np.round(naive_
↪elapsed_time, 2)}s to compute the distance profile")

    return naive_distance_profile
```

For a random time series, `T_random`, with 1 million data points and a random query subsequence, `Q_random`:

[12]:
```
Q_random = np.random.rand(100)
T_random = np.random.rand(1_000_000)

naive_distance_profile = compute_naive_distance_profile(Q_random, T_random)
```

```
For n = 1000000 and m = 100, the naive algorithm takes 40.89s to compute the distance␣
↪profile
```

The naive algorithm takes over half a minute to compute! However, MASS can handle this (and even larger data sets) in about a 1 second:

[13]:
```
start = time.time()
mass_distance_profile = stumpy.core.mass(Q_random, T_random)
mass_elapsed_time = time.time()-start

print(f"For n = {len(T_random)} and m = {len(Q_random)}, the MASS algorithm takes {np.
↪round(mass_elapsed_time, 2)}s to compute the distance profile")
```

```
For n = 1000000 and m = 100, the MASS algorithm takes 0.12s to compute the distance␣
↪profile
```

And to be absolutely certain, let's make sure and check that the output is the same from both methods:

[14]:
```
npt.assert_almost_equal(naive_distance_profile, mass_distance_profile)
```

Success, no errors! This means that both outputs are identical. Go ahead and give it a try!

4.7.10 Resources

The Fastest Similarity Search Algorithm for Time Series Subsequences Under Euclidean Distance

STUMPY Documentation

STUMPY Matrix Profile Github Code Repository

4.8 Finding Conserved Patterns Across Two Time Series

4.8.1 AB-Joins

This tutorial is adapted from the Matrix Profile I paper and replicates Figures 9 and 10.

Previously, we had introduced a concept called time series motifs, which are conserved patterns found within a single time series, T, that can be discovered by computing its matrix profile using STUMPY. This process of computing a matrix profile with one time series is commonly known as a "self-join" since the subsequences within time series T are only being compared with itself. However, what do you do if you have two time series, T_A and T_B, and you want to know if there are any subsequences in T_A that can also be found in T_B? By extension, a motif discovery process involving two time series is often referred to as an "AB-join" since all of the subsequences within time series T_A are compared to all of the subsequences in T_B.

It turns out that "self-joins" can be trivially generalized to "AB-joins" and the resulting matrix profile, which annotates every subsequence in T_A with its nearest subsequence neighbor in T_B, can be used to identify similar (or unique) subsequences across any two time series. Additionally, as long as T_A and T_B both have lengths that are greater than or equal to the subsequence length, m, there is no requirement that the two time series must be the same length.

In this short tutorial we will demonstrate how to find a conserved pattern across two independent time series using STUMPY.

4.8.2 Getting Started

Let's import the packages that we'll need to load, analyze, and plot the data.

[1]:
```
%matplotlib inline

import stumpy
import pandas as pd
import numpy as np
from IPython.display import IFrame
import matplotlib.pyplot as plt

plt.rcParams["figure.figsize"] = [20, 6]  # width, height
plt.rcParams['xtick.direction'] = 'out'
```

4.8.3 Finding Similarities in Music Using STUMPY

In this tutorial we are going to analyze two songs, "Under Pressure" by Queen and David Bowie as well as "Ice Ice Baby" by Vanilla Ice. For those who are unfamiliar, in 1990, Vanilla Ice was alleged to have sampled the bass line from "Under Pressure" without crediting the original creators and the copyright claim was later settled out of court. Have a look at this short video and see if you can hear the similarities between the two songs:

[2]:
```
IFrame(width="560", height="315", src="https://www.youtube.com/embed/HAA__AW3I1M")
```

[2]:
```
<IPython.lib.display.IFrame at 0x7fd515563a10>
```

The two songs certainly share some similarities! But, before we move forward, imagine if you were the judge presiding over this court case. What analysis result would you need to see in order to be convinced, beyond a shadow of a doubt, that there was wrongdoing?

4.8.4 Loading the Music Data

To make things easier, instead of using the raw music audio from each song, we're only going to use audio that has been pre-converted to a single frequency channel (i.e., the 2nd MFCC channel sampled at 100Hz).

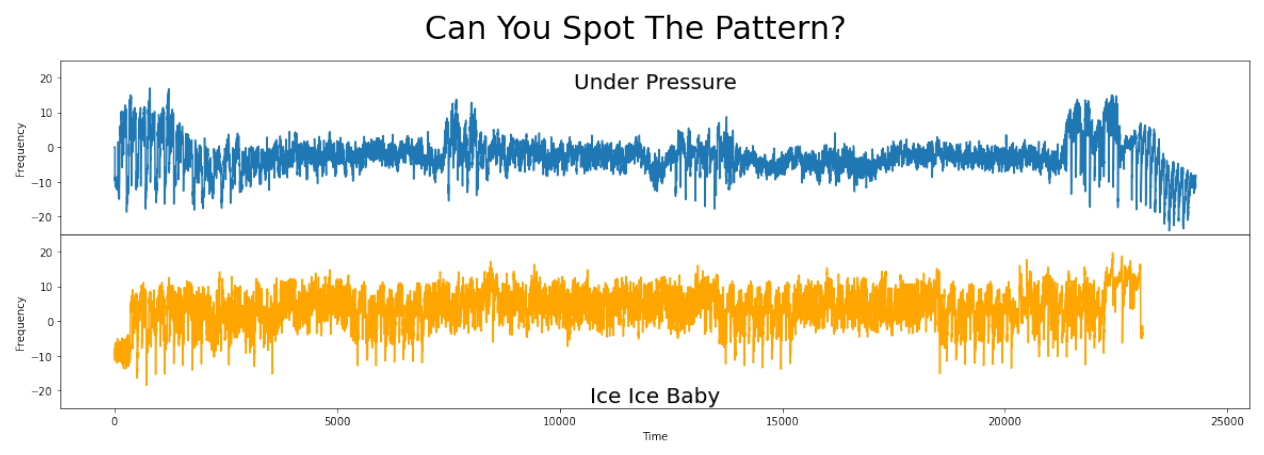
[3]:
```
queen_df = pd.read_csv("https://zenodo.org/record/4294912/files/queen.csv?download=1")
vanilla_ice_df = pd.read_csv("https://zenodo.org/record/4294912/files/vanilla_ice.csv?
↪download=1")

print("Length of Queen dataset : " , queen_df.size)
print("Length of Vanilla ice dataset : " , vanilla_ice_df.size)
```

```
Length of Queen dataset :  24289
Length of Vanilla ice dataset :  23095
```

4.8.5 Visualizing the Audio Frequencies

It was very clear in the earlier video that there are strong similarities between the two songs. However, even with this prior knowledge, it's incredibly difficult to spot the similarities (below) due to the sheer volume of the data:

[4]:
```
fig, axs = plt.subplots(2, sharex=True, gridspec_kw={'hspace': 0})
plt.suptitle('Can You Spot The Pattern?', fontsize='30')

axs[0].set_title('Under Pressure', fontsize=20, y=0.8)
axs[1].set_title('Ice Ice Baby',  fontsize=20, y=0)

axs[1].set_xlabel('Time')

axs[0].set_ylabel('Frequency')
axs[1].set_ylabel('Frequency')

ylim_lower = -25
ylim_upper = 25
axs[0].set_ylim(ylim_lower, ylim_upper)
axs[1].set_ylim(ylim_lower, ylim_upper)

axs[0].plot(queen_df['under_pressure'])
axs[1].plot(vanilla_ice_df['ice_ice_baby'], c='orange')

plt.show()
```



4.8.6 Performing an AB-Join with STUMPY

Fortunately, using the `stumpy.stump` function, we can quickly compute the matrix profile by performing an AB-join and this will help us easily identify and locate the similar subsequence(s) between these two songs:

[5]:
```
m = 500
queen_mp = stumpy.stump(T_A = queen_df['under_pressure'],
                        m = m,
                        T_B = vanilla_ice_df['ice_ice_baby'],
                        ignore_trivial = False)
```

Above, we call `stumpy.stump` by specifying our two time series `T_A = queen_df['under_pressure']` and `T_B = vanilla_ice_df['ice_ice_baby']`. Following the original published work, we use a subsequence window length of `m = 500` and, since this is not a self-join, we set `ignore_trivial = False`. The resulting matrix profile, `queen_mp`, essentially serves as an annotation for `T_A` so, for every subsequence in `T_A`, we find its closest subsequence in `T_B`.

As a brief reminder of the matrix profile data structure, each row of `queen_mp` corresponds to each subsequence within `T_A`, the first column in `queen_mp` records the matrix profile value for each subsequence in `T_A` (i.e., the distance to its nearest neighbor in `T_B`), and the second column in `queen_mp` keeps track of the index location of the nearest neighbor subsequence in `T_B`.

One additional side note is that AB-joins are not symmetrical in general. That is, unlike a self-join, the order of the input time series matter. So, an AB-join will produce a different matrix profile than a BA-join (i.e., for every subsequence in `T_B`, we find its closest subsequence in `T_A`).

4.8.7 Visualizing the Matrix Profile

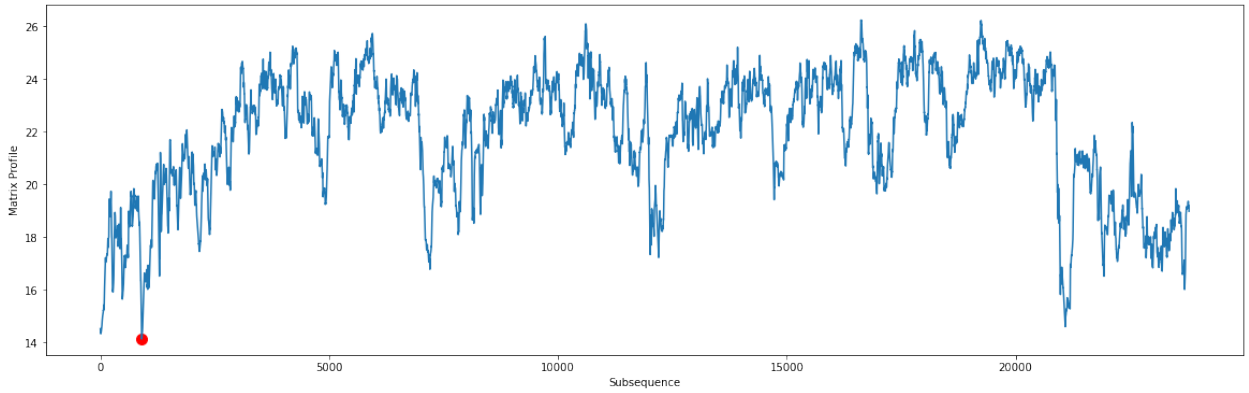
Just as we've done in the past, we can now look at the matrix profile, `queen_mp`, computed from our AB-join:

[6]:
```
queen_motif_index = queen_mp[:, 0].argmin()

plt.xlabel('Subsequence')
plt.ylabel('Matrix Profile')

plt.scatter(queen_motif_index,
            queen_mp[queen_motif_index, 0],
            c='red',
```

(continues on next page)

(continued from previous page)

```
                s=100)

plt.plot(queen_mp[:,0])

plt.show()
```



Now, to discover the global motif (i.e., the most conserved pattern), `queen_motif_index`, all we need to do is identify the index location of the lowest distance value in the `queen_mp` matrix profile (see red circle above).

[7]:
```
queen_motif_index = queen_mp[:, 0].argmin()
print(f'The motif is located at index {queen_motif_index} of "Under Pressure"')
```

```
The motif is located at index 904 of "Under Pressure"
```

In fact, the index location of its nearest neighbor in "Ice Ice Baby" is stored in `queen_mp[queen_motif_index, 1]`:

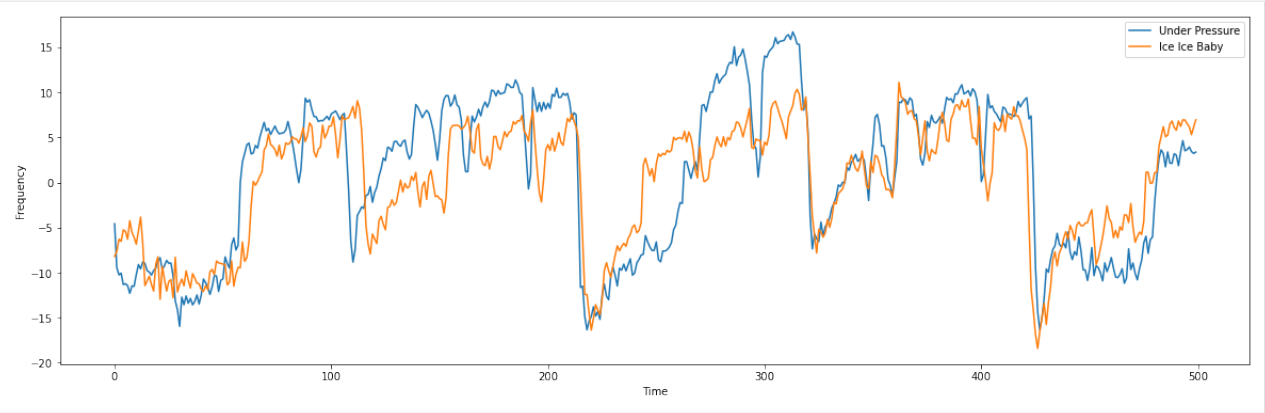
[8]:
```
vanilla_ice_motif_index = queen_mp[queen_motif_index, 1]
print(f'The motif is located at index {vanilla_ice_motif_index} of "Ice Ice Baby"')
```

```
The motif is located at index 288 of "Ice Ice Baby"
```

4.8.8 Overlaying The Best Matching Motif

After identifying the motif and retrieving the index location from each song, let's overlay both of these subsequences and see how similar they are to each other:

[9]:
```
plt.plot(queen_df.iloc[queen_motif_index : queen_motif_index + m].values, label=
↪'Under Pressure')
plt.plot(vanilla_ice_df.iloc[vanilla_ice_motif_index:vanilla_ice_motif_index+m].
↪values, label='Ice Ice Baby')

plt.xlabel('Time')
plt.ylabel('Frequency')

plt.legend()

plt.show()
```



Wow, the resulting overlay shows really strong correlation between the two subsequences! Are you convinced?

4.8.9 Summary

And that's it! In just a few lines of code, you learned how to compute a matrix profile for two time series using STUMPY and identified the top-most conserved behavior between them. While this tutorial has focused on audio data, there are many further applications such as detecting imminent mechanical issues in sensor data by comparing to known experimental or historical failure datasets or finding matching movements in commodities or stock prices, just to name a few.

You can now import this package and use it in your own projects. Happy coding!

4.8.10 Resources

Matrix Profile I

STUMPY Documentation

STUMPY Matrix Profile Github Code Repository

4.9 Consensus Motif Search

This tutorial utilizes the main takeaways from the Matrix Profile XV paper.

Matrix profiles can be used to find conserved patterns within a single time series (self-join) and across two time series (AB-join). In both cases these conserved patterns are often called "motifs". And, when considering a set of three or more time series, one common trick for identifying a conserved motif across the entire set is to:

1. Append a `np.nan` to the end of each time series. This is used to identify the boundary between neighboring time series and ensures that any identified motif will not straddle multiple time series.
2. Concatenate all of the time series into a single long time series
3. Compute the matrix profile (self-join) on the aforementioned concatenated time series

However, this is not guaranteed to find patterns that are conserved across *all* of the time series within the set. This idea of a finding a conserved motif that is common to all of the time series in a set is referred to as a "consensus motif". In this tutorial we will introduce the "Ostinato" algorithm, which is an efficient way to find the consensus motif amongst a set of time series.

4.9.1 Getting started

Let's import the packages that we'll need to load, analyze, and plot the data.

[1]:
```
%matplotlib inline

import stumpy
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from itertools import cycle, combinations
from matplotlib.patches import Rectangle
from scipy.cluster.hierarchy import linkage, dendrogram
from scipy.special import comb

fig_size = plt.rcParams["figure.figsize"]
fig_size[0] = 20
fig_size[1] = 6
plt.rcParams["figure.figsize"] = fig_size
plt.rcParams['xtick.direction'] = 'out'
```

4.9.2 Loading the Eye-tracking (EOG) Dataset

In the following dataset, a volunteer was asked to "spell" out different Japanese sentences by performing eye movements that represented writing strokes of individual Japanese characters. Their eye movements were recorded by an electrooculograph (EOG) and they were given one second to "visually trace" each Japanese character. For our purposes we're only using the vertical eye positions and, conceptually, this basic example reproduced Figure 1 and Figure 2 of the Matrix Profile XV paper.

[2]:
```
sentence_idx = [6, 7, 9, 10, 16, 24]
Ts = [None] * len(sentence_idx)
fs = 50  # eog signal was downsampled to 50 Hz

for i, s in enumerate(sentence_idx):
    Ts[i] = pd.read_csv(f'https://zenodo.org/record/4288978/files/EOG_001_01_{s:03d}.
↪csv?download=1').iloc[:, 0].values

# the literal sentences
sentences = pd.read_csv(f'https://zenodo.org/record/4288978/files/test_sent.jp.csv?
↪download=1', index_col=0)
```

4.9.3 Visualizing the EOG Dataset

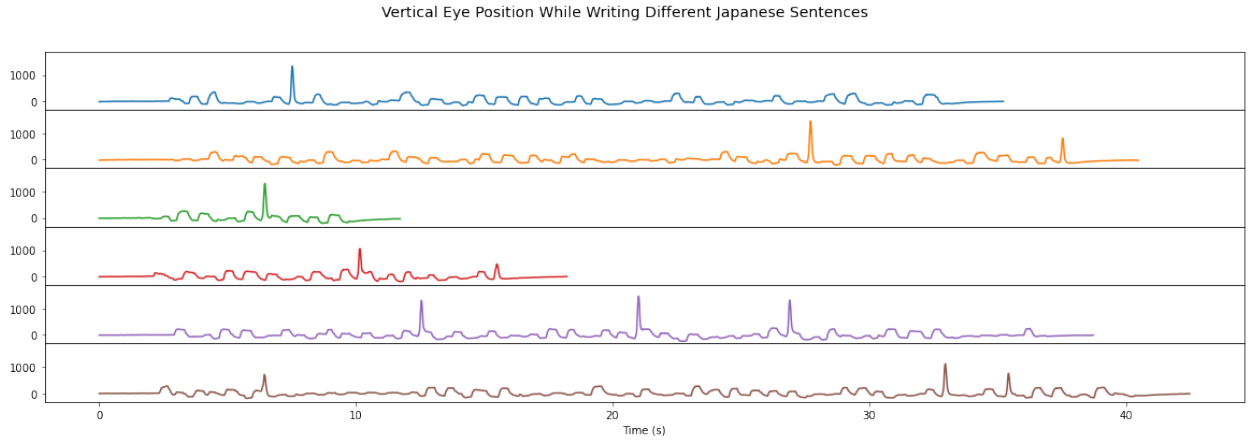
Below, we plotted six time series that each represent the vertical eye position while a person "wrote" Japanese sentences using their eyes. As you can see, some of the Japanese sentences are longer and contain more words while others are shorter. However, there is one common Japanese word (i.e., a "common motif") that is contained in all six examples. Can you spot the one second long pattern that is common across these six time series?

[3]:
```
def plot_vertical_eog():
    fig, ax = plt.subplots(6, sharex=True, sharey=True)
    prop_cycle = plt.rcParams['axes.prop_cycle']
    colors = cycle(prop_cycle.by_key()['color'])
    for i, e in enumerate(Ts):
        ax[i].plot(np.arange(0, len(e)) / fs, e, color=next(colors))
```

(continues on next page)

(continued from previous page)

```
        ax[i].set_ylim((-330, 1900))
    plt.subplots_adjust(hspace=0)
    plt.xlabel('Time (s)')
    return ax

plot_vertical_eog()
plt.suptitle('Vertical Eye Position While Writing Different Japanese Sentences',␣
↪fontsize=14)
plt.show()
```



4.9.4 Consensus Motif Search

To find out, we can use the `stumpy.ostinato` function to help us discover the "consensus motif" by passing in the list of time series, `Ts`, along with the subsequence window size, `m`:

[4]:
```
m = fs
radius, Ts_idx, subseq_idx = stumpy.ostinato(Ts, m)
print(f'Found Best Radius {np.round(radius, 2)} in time series {Ts_idx} starting at␣
↪subsequence index location {subseq_idx}.')
```

```
Found Best Radius 0.87 in time series 4 starting at subsequence index location 1271.
```

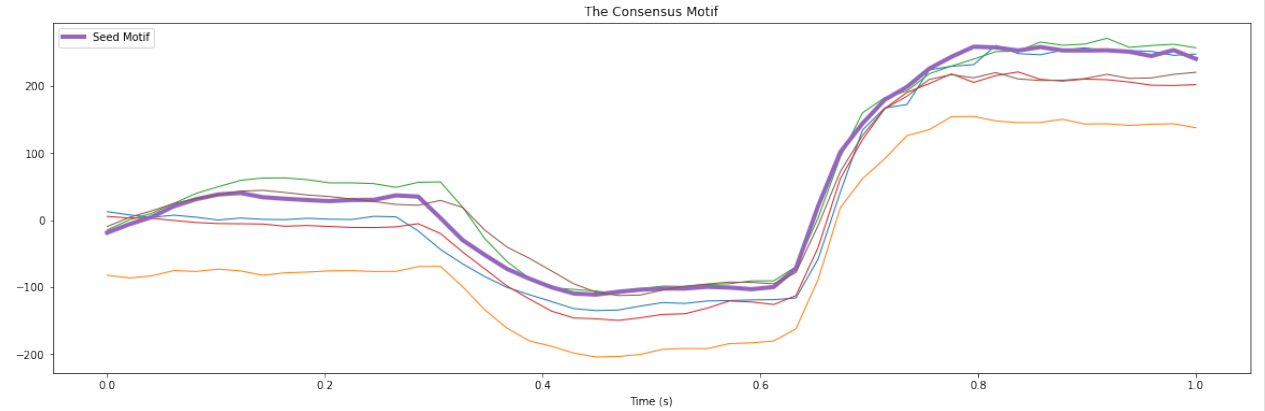
Now, Let's plot the individual subsequences from each time series that correspond to the matching consensus motif:

[5]:
```
seed_motif = Ts[Ts_idx][subseq_idx : subseq_idx + m]
x = np.linspace(0,1,50)
nn = np.zeros(len(Ts), dtype=np.int64)
nn[Ts_idx] = subseq_idx
for i, e in enumerate(Ts):
    if i != Ts_idx:
        nn[i] = np.argmin(stumpy.core.mass(seed_motif, e))
        lw = 1
        label = None
    else:
        lw = 4
        label = 'Seed Motif'
    plt.plot(x, e[nn[i]:nn[i]+m], lw=lw, label=label)
plt.title('The Consensus Motif')
plt.xlabel('Time (s)')
```

(continues on next page)

(continued from previous page)

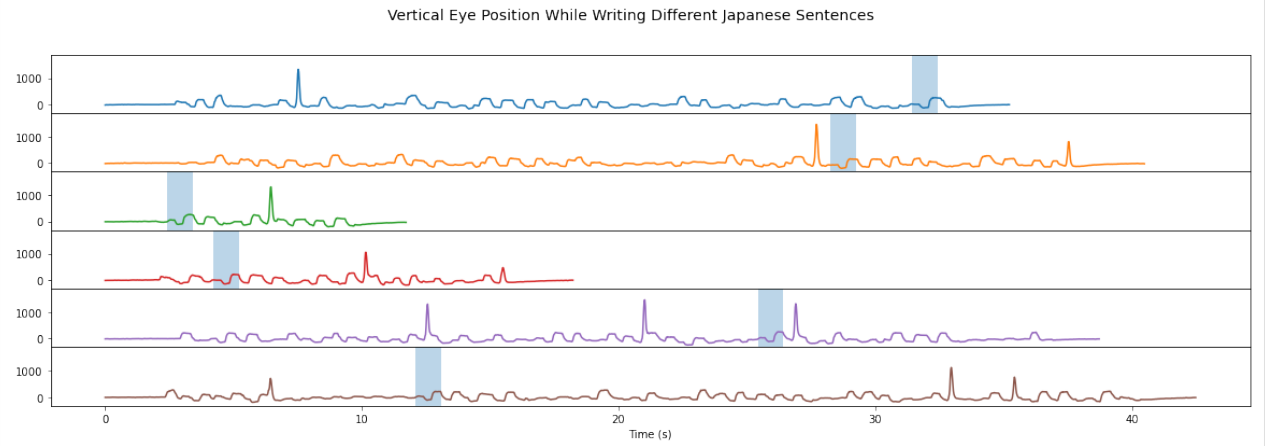
```
plt.legend()
plt.show()
```



There is a striking similarity between the subsequences. The most central "seed motif" is plotted with a thicker purple line.

When we highlight the above subsequences in their original context (light blue boxes below), we can see that they occur at different times:

[6]:
```
ax = plot_vertical_eog()
for i in range(len(Ts)):
    y = ax[i].get_ylim()
    r = Rectangle((nn[i] / fs, y[0]), 1, y[1]-y[0], alpha=0.3)
    ax[i].add_patch(r)
plt.suptitle('Vertical Eye Position While Writing Different Japanese Sentences',␣
↪fontsize=14)
plt.show()
```



The discovered conserved motif (light blue boxes) correspond to writing the Japanese character , which occurs at different times in the different example sentences.

4.9.5 Phylogeny Using Mitochondrial DNA (mtDNA)

In this next example, we'll reproduce Figure 9 from the Matrix Profile XV paper.

Mitochondrial DNA (mtDNA) has been successfully used to determine evolutionary relationships between organisms (phylogeny). Since DNAs are essentially ordered sequences of letters, we can loosely treat them as time series and use all of the available time series tools.

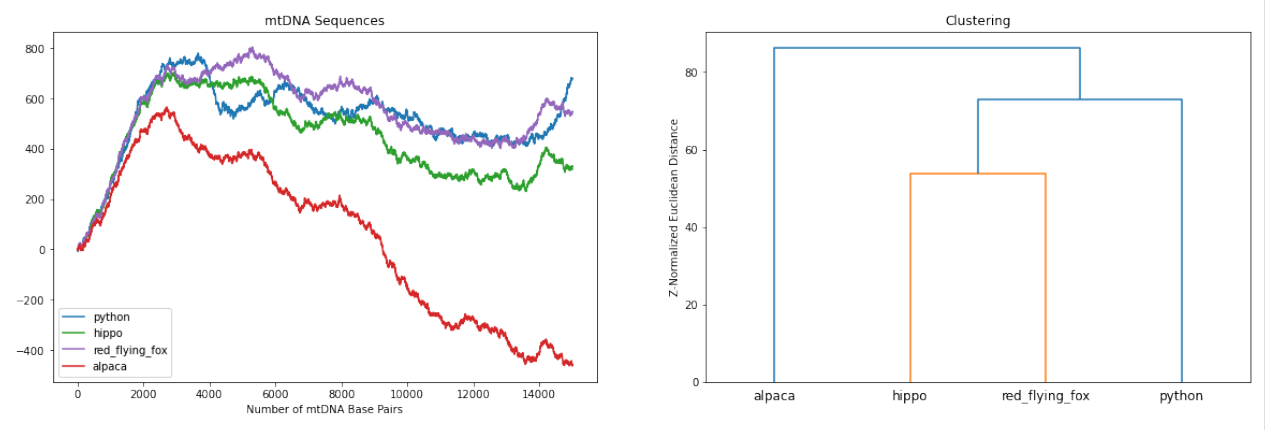
4.9.6 Loading the mtDNA Dataset

[7]:
```
animals = ['python', 'hippo', 'red_flying_fox', 'alpaca']
data = {}
for animal in animals:
    data[animal] = pd.read_csv(f"https://zenodo.org/record/4289120/files/{animal}.csv?
↪download=1").iloc[:,0].values

colors = {'python': 'tab:blue', 'hippo': 'tab:green', 'red_flying_fox': 'tab:purple',
↪'alpaca': 'tab:red'}
```

4.9.7 Clustering Using Large mtDNA Sequences

Naively, using `scipy.cluster.hierarchy` we can cluster the mtDNAs based on the majority of the sequences. A correct clustering would place the two "artiodactyla", hippo and alpaca, closest and, together with the red flying fox, we would expect them to form a cluster of "mammals". Finally, the python, a "reptile", should be furthest away from all of the "mammals".

[8]:
```
fig, ax = plt.subplots(ncols=2)

# sequences in Fig 9 left
truncate = 15000
for k, v in data.items():
    ax[0].plot(v[:truncate], label=k, color=colors[k])
ax[0].legend()
ax[0].set_xlabel('Number of mtDNA Base Pairs')
ax[0].set_title('mtDNA Sequences')

# clustering in Fig 9 left
truncate = 16000
dp = np.zeros(int(comb(4, 2)))
for i, a_c in enumerate(combinations(data.keys(), 2)):
    dp[i] = stumpy.core.mass(data[a_c[0]][:truncate], data[a_c[1]][:truncate])
Z = linkage(dp, optimal_ordering=True)
dendrogram(Z, labels=[k for k in data.keys()], ax=ax[1])
ax[1].set_ylabel('Z-Normalized Euclidean Distance')
ax[1].set_title('Clustering')
plt.show()
```



Uh oh, the clustering is clearly wrong! Amongst other problems, the alpaca (a mammal) should not be most closely related to the python (a reptile).

4.9.8 Consensus Motif Search

In order to obtain the correct relationships, we need to identify and then compare the parts of the mtDNA that is the most conserved across the mtDNA sequences. In other words, we need to cluster based on their consensus motif. Let's limit the subsequence window size to 1,000 base pairs and identify the consensus motif again using the `stumpy.ostinato` function:

[9]:
```
m = 1000
bsf_radius, bsf_Ts_idx, bsf_subseq_idx = stumpy.ostinato(list(data.values()), m)
print(f'Found best radius {np.round(bsf_radius, 2)} in time series {bsf_Ts_idx}␣
↪starting at subsequence index location {bsf_subseq_idx}.')
```

```
Found best radius 2.73 in time series 1 starting at subsequence index location 602.
```

4.9.9 Clustering Using the Consensus mtDNA Motif

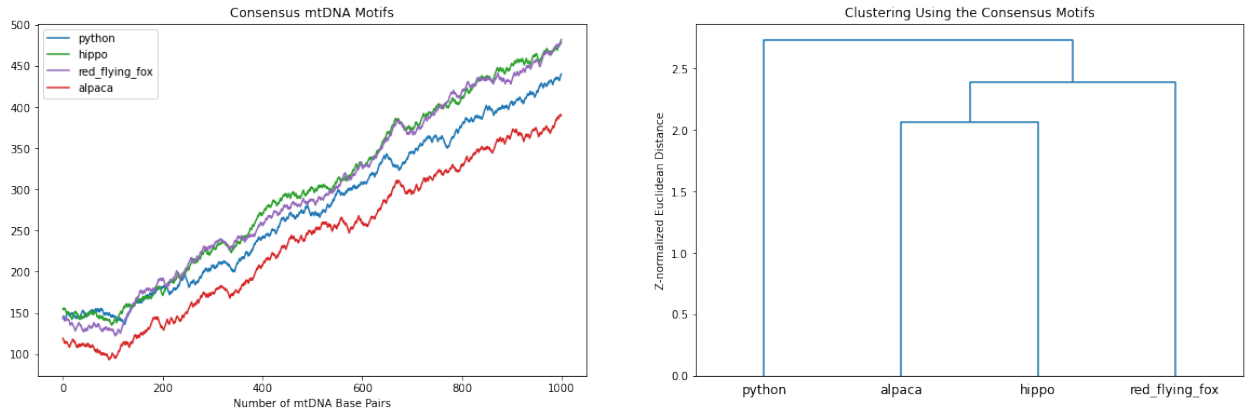
Now, let's perform the clustering again but, this time, using the consensus motif:

[10]:
```
consensus_motifs = {}
best_motif = list(data.items())[bsf_Ts_idx][1][bsf_subseq_idx : bsf_subseq_idx + m]
for i, (k, v) in enumerate(data.items()):
    if i == bsf_Ts_idx:
        consensus_motifs[k] = best_motif
    else:
        idx = np.argmin(stumpy.core.mass(best_motif, v))
        consensus_motifs[k] = v[idx : idx + m]

fig, ax = plt.subplots(ncols=2)
# plot the consensus motifs
for animal, motif in consensus_motifs.items():
    ax[0].plot(motif, label=animal, color=colors[animal])
ax[0].legend()
# cluster consensus motifs
dp = np.zeros(int(comb(4, 2)))
for i, motif in enumerate(combinations(list(consensus_motifs.values()), 2)):
    dp[i] = stumpy.core.mass(motif[0], motif[1])
```

(continues on next page)

(continued from previous page)

```
Z = linkage(dp, optimal_ordering=True)
dendrogram(Z, labels=[k for k in consensus_motifs.keys()])

ax[0].set_title('Consensus mtDNA Motifs')
ax[0].set_xlabel('Number of mtDNA Base Pairs')
ax[1].set_title('Clustering Using the Consensus Motifs')
ax[1].set_ylabel('Z-normalized Euclidean Distance')
plt.show()
```



Now this looks much better! Hierarchically, the python is "far away" from the other mammals and, amongst the mammalia, the red flying fox (a bat) is less related to both the alpaca and the hippo which are the closest evolutionary relatives in this set of animals.

4.9.10 Summary

And that's it! You have now learned how to search for a consensus motif amongst a set of times series using the awesome `stumpy.ostinato` function. You can now import this package and use it in your own projects. Happy coding!

4.9.11 Resources

Matrix Profile XV

STUMPY Documentation

STUMPY Matrix Profile Github Code Repository

4.10 Multidimensional Motif Discovery

4.10.1 Finding a Motif in Multidimensional Time Series Data with MSTUMP

This tutorial utilizes the main takeaways from the Matrix Profile VI research paper and requires STUMPY v1.6.1 or newer. Also, the word "dimensionality" is overloaded for multi-dimensional time series since it is often used to refer to both the number of time series and to the number of data points in a subsequence. For clarity, we restrict our use of "dimensions" to refer only to the number of time series and not to the number of data points.

Previously, we had introduced a concept called time series motifs, which are conserved patterns found within a 1-dimensional time series, T, that can be discovered by computing its matrix profile using STUMPY. This process of

computing a matrix profile with one time series is commonly known as a "self-join" since the subsequences within time series T are only being compared with itself. Since the first 1-dimensional motif discovery algorithm was introduced in 2002, a lot of effort has been made to generalize motif-finding to the multi-dimensional case but producing multi-dimensional matrix profiles are computationally expensive and so extra care must be taken to minimize the added time complexity. Also, while it may be tempting to find motifs in all available dimensions (i.e., a motif must exist in all dimensions and occur simultaneously), it has been shown that this rarely produces meaningful motifs except in the most contrived situations. Instead, given a set of time series dimensions, we should filter them down to a subset of "useful" dimensions before assigning a subsequence as a motif. For example, take a look at this motion capture of a boxer throwing some punches:

[1]:
```
from IPython.display import IFrame

IFrame(width="560", height="315", src="https://www.youtube.com/embed/2CQttFf2OhU")
```

[1]:
```
<IPython.lib.display.IFrame at 0x7f87f06c5e90>
```

If we strictly focus on the boxer's right arm in both cases, the two punches are almost identical. The position of the right shoulder, right elbow, and right hand (a three dimensional time series) are virtually the indistinguishable. So, identifying this punching motif is relatively straightforward when we limit ourselves to only a subset of all of the available body movement dimensions. However, if we incorporate the full set of motion capture markers from all of the limbs (i.e., increasing the number of dimensions from three), the differences captured by the left arm and the subtle noise in the footwork actually drowns out the the similarity of the right arm motions, making the previous punching motif impossible to find. This example demonstrates how classic multidimensional motif discovery algorithms are likely to fail since they try to use all of the available dimensions. So, not only do we need an efficient algorithm for computing the multi-dimensional matrix profile but we also need to establish an informed approach to guide us in selecting the relevant subset of dimensions that are to be used in identifying multi-dimensional motifs.

In this tutorial, we will explain precisely what a multi-dimensional matrix profile is and then we'll learn how to compute it using the `mstump` function (i.e., "multi-dimensional STUMP") by exploring a simple toy dataset. To conclude, we'll see if we can identify a meaningful sub-dimensional motif (i.e., that only uses a subset of dimensions) in this multi-dimensional time series data.

4.10.2 Getting Started

Let's import the packages that we'll need to load, analyze, and plot the data.

[2]:
```
%matplotlib inline

import pandas as pd
import numpy as np
import stumpy
import matplotlib.pyplot as plt

plt.rcParams["figure.figsize"] = [20, 6]  # width, height
plt.rcParams['xtick.direction'] = 'out'
```

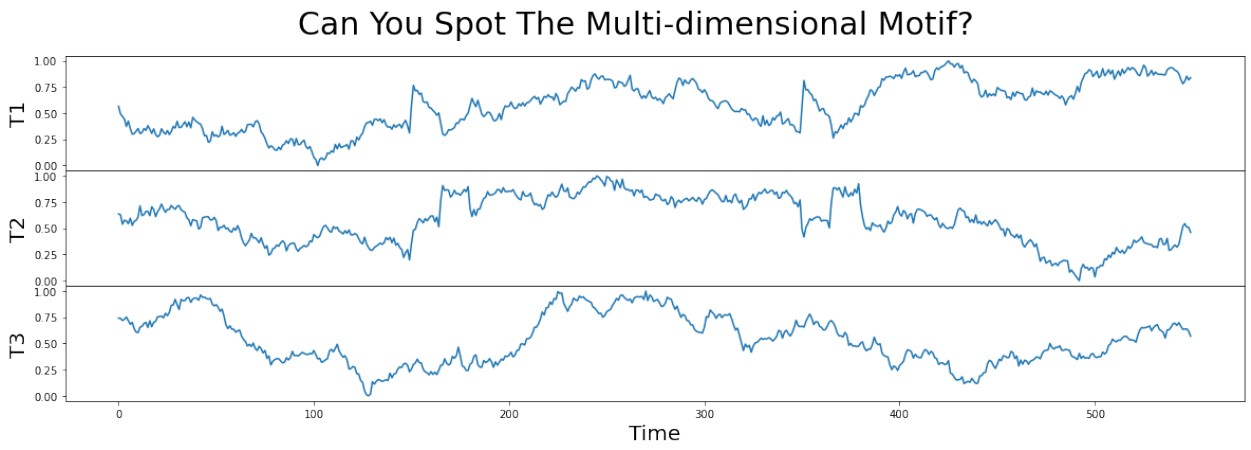
4.10.3 Loading and Visualizing the Toy Data

In this example data, we have a 3-dimensional time series labeled `T1`, `T2`, and `T3`. Can you spot where the motif is? Does that motif exist in one, two, or all three dimensions?

[3]:
```
df = pd.read_csv("https://zenodo.org/record/4328047/files/toy.csv?download=1")
df.head()
```

[3]:
```
         T1        T2        T3
0  0.565117  0.637180  0.741822
1  0.493513  0.629415  0.739731
2  0.469350  0.539220  0.718757
3  0.444100  0.577670  0.730169
4  0.373008  0.570180  0.752406
```

[4]:
```
fig, axs = plt.subplots(df.shape[1], sharex=True, gridspec_kw={'hspace': 0})
plt.suptitle('Can You Spot The Multi-dimensional Motif?', fontsize='30')

for i in range(df.shape[1]):
    axs[i].set_ylabel(f'T{i + 1}', fontsize='20')
    axs[i].set_xlabel('Time', fontsize ='20')
    axs[i].plot(df[f'T{i + 1}'])

plt.show()
```



4.10.4 A Quick 1-Dimensional Exploration

Before diving into a multi-dimensional matrix profile analysis, let's take a naive approach and simply run the classic 1-dimensional motif discovery algorithm, `stumpy.stump`, on each of the dimensions independently (using a window size of `m = 30`) and extract 1-dimensional motif pairs:

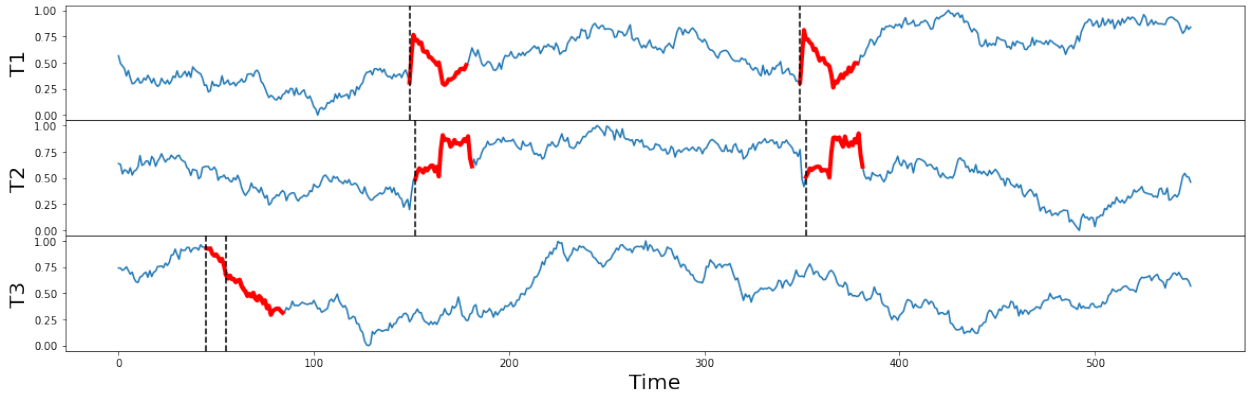
[5]:
```
m = 30
mps = {}  # Store the 1-dimensional matrix profiles
motifs_idx = {}  # Store the index locations for each pair of 1-dimensional motifs (i.
↪e., the index location of two smallest matrix profile values within each dimension)
for dim_name in df.columns:
    mps[dim_name] = stumpy.stump(df[dim_name], m)
    motif_distance = np.round(mps[dim_name][:, 0].min(), 1)
    print(f"The motif pair matrix profile value in {dim_name} is {motif_distance}")
    motifs_idx[dim_name] = np.argsort(mps[dim_name][:, 0])[:2]
```

```
The motif pair matrix profile value in T1 is 1.1
The motif pair matrix profile value in T2 is 1.0
The motif pair matrix profile value in T3 is 1.1
```

And when we plot the raw times series again (below) along with their independently discovered motifs (thick red lines), we can correctly match the visually obvious motif pairs in `T1` and `T2` starting near locations `150` and `350` (dotted vertical lines). Notice that these two motifs aren't perfectly aligned in time (i.e., they aren't occurring simultaneously)

but they are reasonably close to each and their motif pair values are `1.1` and `1.0`, respectively. This is a great start!

[6]:
```
fig, axs = plt.subplots(len(mps), sharex=True, gridspec_kw={'hspace': 0})

for i, dim_name in enumerate(list(mps.keys())):
    axs[i].set_ylabel(dim_name, fontsize='20')
    axs[i].plot(df[dim_name])
    axs[i].set_xlabel('Time', fontsize ='20')
    for idx in motifs_idx[dim_name]:
        axs[i].plot(df[dim_name].iloc[idx:idx+m], c='red', linewidth=4)
        axs[i].axvline(x=idx, linestyle="dashed", c='black')

plt.show()
```



However, when we examine `T3`, its motif pair are overlapping each other starting near position `50` (dotted vertical lines) and they are relatively far away from the motifs discovered in `T1` and `T2`. Oh, no!

In fact, `T3` is actually a "random walk" time series that was purposely included in this set as a decoy and, unlike `T1` and `T2`, `T3` does not contain any conserved behavior whatsoever despite the fact that the distance between its motif pair is `1.1`, a red herring. This illustrates a few important points:

1. if there are additional irrelevant dimensions (i.e., `T3`), you will do about as well as random chance at discovering multi-dimensional motifs if you don't ignore/dismiss those distracting dimensions
2. if you suspect that there are motifs in only a subset of the time series, how do you know which dimensions are involved, or how do you even know how many dimensions are relevant?
3. doing motif search on all dimensions is almost always guaranteed to produce meaningless results, even if a subset of dimensions has clear and unambiguous motifs (like our example above)

A quick survey of all current multi-dimensional motif discovery algorithms in the literature (see Section II in Matrix Profile VI) reveals that they are slow, approximate, and brittle to irrelevant dimensions. In contrast, what we need is an algorithm that is fast, exact, and robust to hundreds of irrelevant dimensions and to spurious data. And this is where `stumpy.mstump` can help!

4.10.5 Multi-dimensional Matrix Profiles

There is no substitution for the multi-dimensional matrix profile definitions provided in the Matrix Profile VI paper (see Section III and Section IV) and so we refer the reader to this quintessential resource for a detailed walkthrough. However, to develop some basic intuition, we'll share an oversimplified description for computing a multi-dimensional matrix profile but know that the `stumpy.mstump` function provides a highly efficient, accurate, and scalable variant to the naive explanation provided here.

First and foremost, we must start by dispelling a common misconception regarding multi-dimensional matrix profiles:

Multi-dimensional matrix profiles are not 1-dimensional matrix profiles stacked one on top of each other!

So, what is a multi-dimensional matrix profile? To answer this question, let's step away from our toy data example for a minute and suppose that we have a "multi-dimensional time series", $T = [T1, T2, T3, T4]$, which has $d = 4$ dimensions and length $n = 7$ (i.e., there are seven data points or "positions" within each dimension). Then, T simply has shape $d \times n$ (or 4×7). If we choose a window size, $m = 3$, then we can define the i^{th} "multi-dimensional subsequence" as a continuous subset of the values from T of length m starting from position i that has an overall shape of $d \times m$ (or 4×3). You can think of each multi-dimensional subsequence as a rectangular slice of T and T can only have exactly $l = n - m + 1$ multi-dimensional subsequences. In our example, T has exactly $l = 5$ multi-dimensional subsequences (i.e., we can incrementally slide a 4×3-shaped rectangle across the length of T only 5 times before we reach the end of T) and, for the i^{th} multi-dimensional subsequence, we can iterate over each of its dimensions independently and compute an aggregated "multi-dimensional distance profile" (i.e., three 1-dimensional distance profiles stacked one on top of each other). Essentially, the i^{th} multi-dimensional distance profile has shape $d \times l$ (or 3×5) and gives you the pairwise distances between the i^{th} multi-dimensional subsequence and all other possible multi-dimensional subsequences within T.

Recall that our ultimate goal is to output something called the "multi-dimensional matrix profile" (and its corresponding "multi-dimensional matrix profile indices"), which has an overall shape of $d \times l$ (i.e., one set of d values for each of the l multi-dimensional subsequences). As it turns out, the values in the i^{th} column of the multi-dimensional matrix profile is directly derived from the i^{th} multi-dimensional distance profile. Continuing with our example, let's illustrate this process using the fictitious array below to represent a typical multi-dimensional distance profile for the i^{th} multi-dimensional subsequence:

```
ith_distance_profile = np.ndarray([[0.4, 0.2, 0.6, 0.5, 0.2, 0.1, 0.9],
                                   [0.7, 0.0, 0.2, 0.6, 0.1, 0.2, 0.9],
                                   [0.6, 0.7, 0.1, 0.5, 0.8, 0.3, 0.4],
                                   [0.7, 0.4, 0.3, 0.1, 0.2, 0.1, 0.7]])
```

With this, we can now identify the set of d values that form the i^{th} column vector of the multi-dimensional matrix profile with shape $d \times 1$ (or 4×1). The value for the first dimension is found by extracting the smallest value in each column of the `ith_distance_profile` and then returning the minimum value in the reduced set. Then, the value for the second dimension is found by extracting the two smallest values in each column of the `ith_distance_profile`, averaging these two values, and then returning the minimum averaged value in the reduced set. Finally, the value for the k^{th} out of d dimensions is found by extracting the k smallest values in each column of the `ith_distance_profile`, averaging these k values, and then returning the minimum averaged value in the reduced set. A naive algorithm might look something like this:

```
ith_matrix_profile = np.full(d, np.inf)
ith_indices = np.full(d, -1, dtype=np.int64)

for k in range(1, d + 1):
    smallest_k = np.partition(ith_distance_profile, k, axis=0)[:k]  # retrieves the␣
↪smallest k values in each column
    averaged_smallest_k = smallest_k.mean(axis=0)
    min_val = averaged_smallest_k.min()
    if min_val < ith_matrix_profile[k - 1]:
        ith_matrix_profile[k - 1] = min_val
        ith_indices[k - 1] = averaged_smallest_k.argmin()
```

Therefore, by simply advancing the i^{th} multi-dimensional subsequence along the entire length of T and then computing its corresponding i^{th} multi-dimensional matrix profile (and indices), we can easily populate the full multi-dimensional matrix profile and multi-dimensional matrix profile indices. And, hopefully, you'd agree with our initial statement that:

multi-dimensional matrix profiles are not 1-dimensional matrix profiles stacked one on top of each other!

But then what exactly does each dimension of the multi-dimensional matrix profile tell us? Essentially, the k^{th} dimension (or row) of the multi-dimensional matrix profile stores the distance between each subsequence and its

nearest neighbor (the distance is computed using a k-dimensional distance function as we saw above). We should point out that, for the k^{th} dimension of the multi-dimensional matrix profile, only a subset of time series dimensions (i.e., k out of d dimensions) are selected and this subset of chosen dimensions can change as you vary either the i^{th} multi-dimensional subsequence and/or k.

Now that we have a better understanding of what a multi-dimensional matrix profile is, let's go ahead and compute it by simply calling the `stumpy.mstump` function on our original toy dataset:

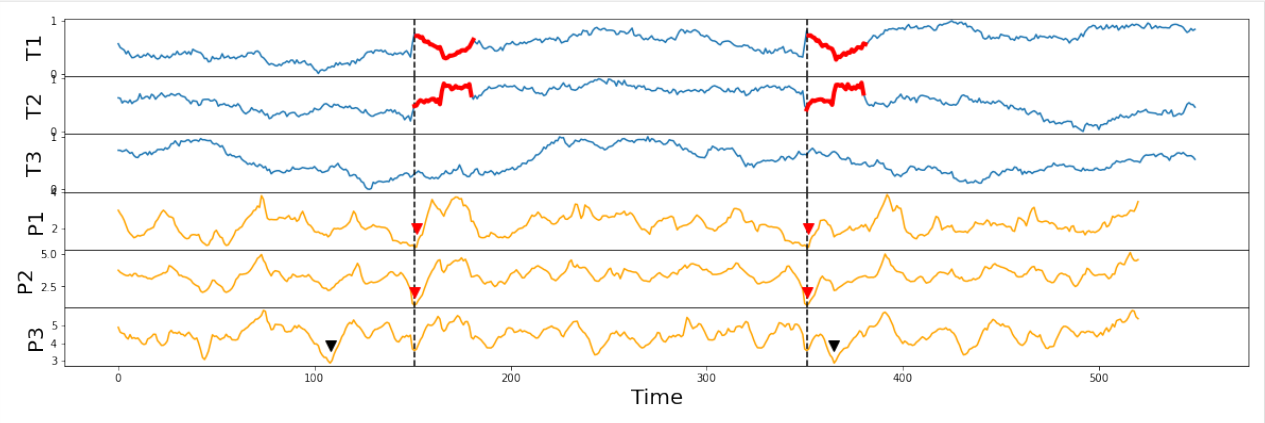
[7]:
```
mps, indices = stumpy.mstump(df, m)
```

Consequently, the "k-dimensional motif" can be found by locating the two lowest values in the correspond k-dimensional matrix profile, `mps`, (these two lowest values must be a tie).

[8]:
```
motifs_idx = np.argsort(mps, axis=1)[:, :2]
```

Finally, we can plot the k-dimensional matrix profile (orange lines) for all possible values of k (i.e., `P1`, `P2`, and `P3`) alongside the original time series data (blue lines):

[9]:
```
fig, axs = plt.subplots(mps.shape[0] * 2, sharex=True, gridspec_kw={'hspace': 0})

for k, dim_name in enumerate(df.columns):
    axs[k].set_ylabel(dim_name, fontsize='20')
    axs[k].plot(df[dim_name])
    axs[k].set_xlabel('Time', fontsize ='20')

    axs[k + mps.shape[0]].set_ylabel(dim_name.replace('T', 'P'), fontsize='20')
    axs[k + mps.shape[0]].plot(mps[k], c='orange')
    axs[k + mps.shape[0]].set_xlabel('Time', fontsize ='20')

    axs[k].axvline(x=motifs_idx[1, 0], linestyle="dashed", c='black')
    axs[k].axvline(x=motifs_idx[1, 1], linestyle="dashed", c='black')
    axs[k + mps.shape[0]].axvline(x=motifs_idx[1, 0], linestyle="dashed", c='black')
    axs[k + mps.shape[0]].axvline(x=motifs_idx[1, 1], linestyle="dashed", c='black')

    if dim_name != 'T3':
        axs[k].plot(range(motifs_idx[k, 0], motifs_idx[k, 0] + m), df[dim_name].
↪iloc[motifs_idx[k, 0] : motifs_idx[k, 0] + m], c='red', linewidth=4)
        axs[k].plot(range(motifs_idx[k, 1], motifs_idx[k, 1] + m), df[dim_name].
↪iloc[motifs_idx[k, 1] : motifs_idx[k, 1] + m], c='red', linewidth=4)
        axs[k + mps.shape[0]].plot(motifs_idx[k, 0], mps[k, motifs_idx[k, 0]] + 1,␣
↪marker="v", markersize=10, color='red')
        axs[k + mps.shape[0]].plot(motifs_idx[k, 1], mps[k, motifs_idx[k, 1]] + 1,␣
↪marker="v", markersize=10, color='red')
    else:
        axs[k + mps.shape[0]].plot(motifs_idx[k, 0], mps[k, motifs_idx[k, 0]] + 1,␣
↪marker="v", markersize=10, color='black')
        axs[k + mps.shape[0]].plot(motifs_idx[k, 1], mps[k, motifs_idx[k, 1]] + 1,␣
↪marker="v", markersize=10, color='black')

plt.show()
```



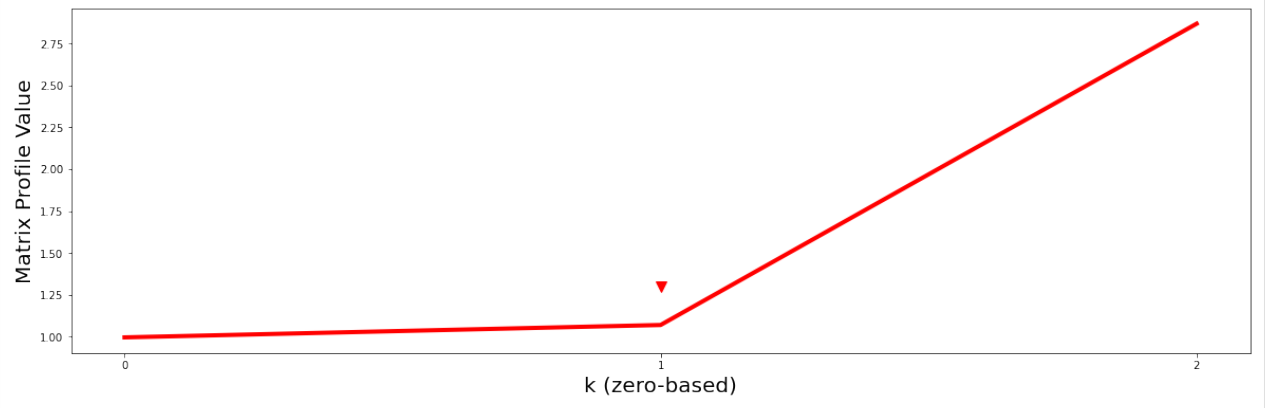
Notice that the (implanted) semantically meaningful motif (thick red lines) can be spotted visually by inspecting the locations of the lowest points (red arrowheads) in either the `P1` or `P2` matrix profiles but the `P3` case has identified the motif (black arrowheads) in an effectively random location, which further reinforces the point that we had made earlier:

> if there are additional irrelevant dimensions (i.e., `T3`), you will do about as well as random chance at discovering multi-dimensional motifs if you don't ignore/dismiss those distracting dimensions

Additionally, it may seem counterintuitive, but as demonstrated above, the lower dimensional motif(s) may or may not necessarily be a subset of the higher dimensional motif, since the lower dimensional motif pair could be closer than any subset of dimensions in the higher dimensional motif pair. In general, this is a subtle but important point to keep in mind.

So then how do we choose the "right" k? One straightforward approach is to turn this into a classic elbow/knee finding problem by plotting the minimum matrix profile value in each dimension against k and then you look for the "turning point" (i.e., the point of maximum curvature):

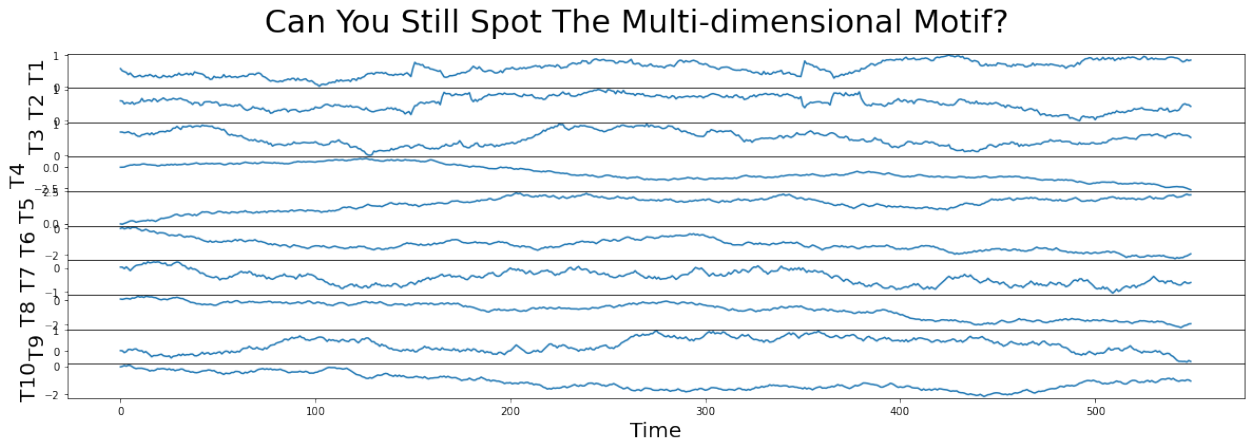
[10]:
```
plt.plot(mps[range(mps.shape[0]), motifs_idx[:, 0]], c='red', linewidth='4')
plt.xlabel('k (zero-based)', fontsize='20')
plt.ylabel('Matrix Profile Value', fontsize='20')
plt.xticks(range(mps.shape[0]))
plt.plot(1, 1.3, marker="v", markersize=10, color='red')
plt.show()
```



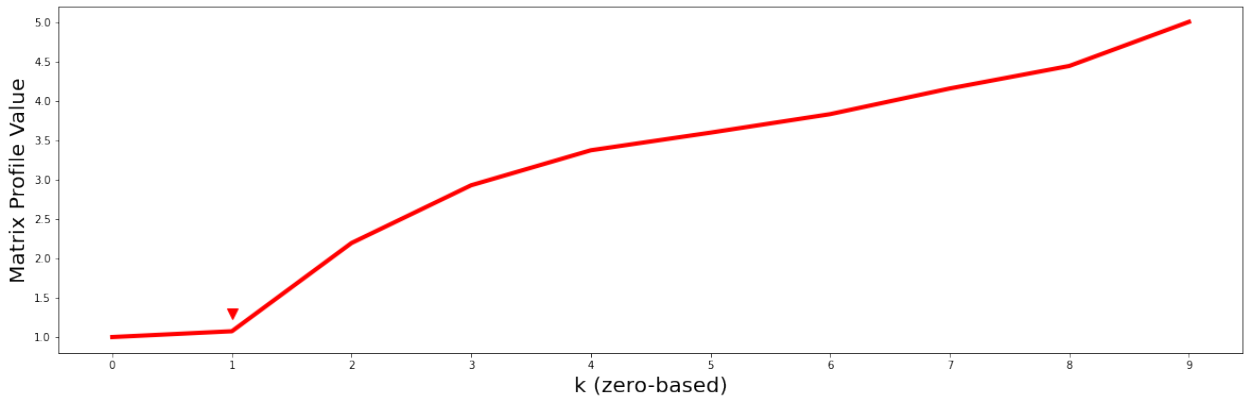
Notice that the thick red line curves up sharply right after $k = 1$ (red arrowhead) like a "hockey stick" and so, naturally, we should choose $P2$ and its motif as the best motif out of all possible k-dimensional motifs.

To really drive home this point, let's add a few more random walk decoys to our toy data set and compare the smallest k-dimensional matrix profile values again:

[11]:
```
for i in range(4, 11):
    df[f'T{i}'] = np.random.uniform(0.1, -0.1, size=df.shape[0]).cumsum()

fig, axs = plt.subplots(df.shape[1], sharex=True, gridspec_kw={'hspace': 0})
plt.suptitle('Can You Still Spot The Multi-dimensional Motif?', fontsize='30')

for i in range(df.shape[1]):
    axs[i].set_ylabel(f'T{i + 1}', fontsize='20')
    axs[i].set_xlabel('Time', fontsize ='20')
    axs[i].plot(df[f'T{i + 1}'])

plt.show()
```



[12]:
```
mps, indices = stumpy.mstump(df, m)
motifs_idx = np.argsort(mps, axis=1)[:, :2]
plt.plot(mps[range(mps.shape[0]), motifs_idx[:, 0]], c='red', linewidth='4')
plt.xlabel('k (zero-based)', fontsize='20')
plt.ylabel('Matrix Profile Value', fontsize='20')
plt.xticks(range(mps.shape[0]))
plt.plot(1, 1.3, marker="v", markersize=10, color='red')
plt.show()
```



Again, the "point of maximum curvature" occurs right after $k = 1$ (red arrowhead) and so we should continue to choose $P2$ and its motif as the best motif out of all possible k-dimensional motifs. While this might seem like a rather manual task, we recommend automating this turning point selection process using the kneed Python package.

The astute reader may have also recognized that the k-dimensional matrix profile really only reveals the location of a motif in time but it fails to disclose which k out of the d time series dimensions contains the desired motif pair. To

recover this information, we must compute something called the "k-dimensional matrix profile subspace". Luckily, the STUMPY convenience function, `stumpy.subspace`, can help us with this!

4.10.6 Matrix Profile Subspace

To use `stumpy.subspace`, we simply pass in:

1. the multi-dimensional time series, `df`, used to compute the multi-dimensional matrix profile
2. the window size, `m`, used to compute the multi-dimensional matrix profile
3. the indices for the k^{th}-dimensional motif pair, `motifs_idx[k][0]` and `indices[k][motifs_idx[k][0]]`
4. the desired (zero-based) dimension, `k`

[13]:
```
k = 1
S = stumpy.subspace(df, m, motifs_idx[k][0], indices[k][motifs_idx[k][0]], k)

print(f"For k = {k}, the {k + 1}-dimensional subspace includes subsequences from {df.
↪columns[S].values}")
```

```
For k = 1, the 2-dimensional subspace includes subsequences from ['T2' 'T1']
```

So, after computing the multi-dimensional matrix profile using `mstump` and selecting the second dimension (i.e., $k = 1$), according to the k-dimensional subspace, the motif should be extracted from `T2` and `T1` out of all of the possible time series dimensions.

4.10.7 Bonus Section

MSTUMPED - Distributed MSTUMP

Dependent upon the total number of dimensions and the length of your time series dat, it may be computationally expensive to produce the multi-dimensional matrix profile. Thus, you can overcome this by trying `mstumped`, a distributed and parallel implementation of `mstump` that depends on Dask distributed:

```
import stumpy
from dask.distributed import Client

dask_client = Client()

mps, indices = stumpy.mstumped(dask_client, df, m)  # Note that a dask client is␣
↪needed
```

Constrained Search

There may be situations where you want to find the best motif on k dimensions but you want to explicitly "include" or "exclude" a given subset of dimensions. In the trivial exclusion case, one just needs to omit the undesired time series dimensions before calling `mstump`:

```
mps, indices = stumpy.mstump(df[df.columns.difference(['T3'])], m)  # This excludes␣
↪the `T3` dimension
```

However, in the case of inclusion, the user may have specific time series dimensions that they'd always want to be included or prioritized in the multi-dimensional matrix profile output and so you can provide a list of (zero-based) dimensions as an parameter:

```
mps, indices = stumpy.mstump(df, m, include=[0, 1])
```

So, in this example where we've instructed `mstump` to `include=[0, 1]` (i.e., `T1` and `T2`), when $k \geq 1$, the k^{th} dimensional matrix profile subspace will always include `T1` and `T2`. Similarly, $k1$ will include either `T1` or `T2`.

Finally, instead of searching for motifs, it is also possible to have `mstump` search for discords by simply passing in the `discords=True` parameter:

```
mps, indices = stumpy.mstump(df, m, discords=True)
```

Instead of returning the smallest average distance, this returns the largest average distance across k dimensions. This ability to return discords is unique to STUMPY and was not published in the original paper. Also note that it is possible to include specific dimensions and search for discords at the same time:

```
mps, indices = stumpy.mstump(df, m, include=[0, 1], discords=True)
```

In this case, the dimensions listed in `include` are honored first and then all subsequent dimensions are sorted by their largest average distance across k dimensions.

4.10.8 Summary

And that's it! You've just learned the basics of how to analyze multi-dimensional time series data using `stumpy.mstump` (or `stumpy.mstumped`). Happy coding!

4.10.9 Resources

Matrix Profile VI

STUMPY Documentation

STUMPY Matrix Profile Github Code Repository

CHAPTER 5

Contributing Guide

So you're interested in making your first open source contribution to STUMPY? We're excited to start this journey with you. Here are some things that will help you get started.

5.1 Git and GitHub

You don't need a GitHub account to use STUMPY, but you do if you want to make a code or documentation contribution. This account will enable you to contribute to many of the most popular open source projects. Additionally, it will give you a place to store, track, and coordinate progress on your own projects. GitHub is a `remote repository` for the version control system `git`. `Git` is a tool that tracks changes in projects by constructing a directed acyclic graph (fancy way of allowing us to create `branches`, `revert` changes, and identify `conflicts`). You use `git` on your local machine, but when you want to save your work or collaborate with other team members, you `push` your work to a `remote` repository. In this case, that `remote` will be GitHub.

With all of that in mind, you'll also need to install git.

> Warning: Don't confuse git and GitHub. Git is a tool used for version control. GitHub is an online platform used as a remote repository for git projects.

Checklist:
[ ] Create GitHub account
[ ] Install Git

5.2 Find your contribution

You've decided that you want to contribute, but how do you approach a new project and figure out where you can help? This will feel like like trying to jump into a conversation that's been happening for months (or years) and can often be intimidating. If you've used the project before, you'll be more familiar with its structure and API, but you probably haven't "peeked behind the curtain" before. The best place to get started is the list of Issues. These are

requests/changes/bugs that other people have identified. Feel free to peruse the list to get a feel for all of the work ongoing in the project. Often, maintainers will have a label system to organize the issues. These labels may include things like `documentation` or `enhancement`. For new contributors, many projects have a good first issue label.

Your next stop (if it exists), should always be CONTRIBUTING.md. Here, the maintainers outline any guidance they have for contributors,

If you click on any issue, you'll see a running history of chats. This serves as the record of thoughts around that specific issue. For some issues, you may see an ongoing conversation. In others, you may just see the initial issue. This is your chance to have a dialogue with the maintainers. If you've found an open issue that interests you and you think you may be able to solve, feel free to leave a message. Remember that maintainers are people too and at STUMPY, they're excited to help new contributors. Here's an example of a potential message:

> First time contributor here! I see that this issue is related to a bug caused by a collision in two variable names *here* and *here*. Does that seem right to you? If so, I'd like to take this issue and submit a pull request.

This message does a couple of things. It shows that you've done a little bit of research into the issue (demonstrates respect for the maintainers' time), gets their input (helps you solve the problem) and makes sure they will accept a pull request (saves your time/effort). Remember how you're jumping into an ongoing conversation? Sometimes an issue will become irrelevant or obsolete based on changes happening somewhere else in the package. A quick note like this lets the maintainer know that you're going to work on this issue and confirms that it is still a valuable issue.

> Warning: If you can tell someone else is actively working on an issue, don't take it.

Checklist:
[ ] Identify an issue
[ ] Read CONTRIBUTING.md
[ ] Post your proposal in the issue

5.3 Setup your Branch

Okay, so you've picked an issue and the maintainers are excited to have your help. Now, let's get to work!

First, you need to create copy of the repository for you to work off of; this is called a `fork`. Here are instructions on forking a repository. Now you have your own copy associated with your GitHub account.

Next, you need to `clone` this copy of the repository. This simply downloads it to your computer so that you can work on it. Here are instructions on cloning a repository. Now you have a copy downloaded on your computer. *Remember to clone your fork, not STUMPY.

Next, you want to create a `branch`. Here's an overview of how git branches work, but if you're working in the command line, you probably just need to type `git checkout -b branch_name`. `branch_name` should be something descriptive about the change you are making like `change_index_slice` or `document_x`.

Now, any changes you make in the project will be reflected in your branch. Later, You'll want your changes to be `merged` into the main STUMPY repository (so that everyone can enjoy your craftsmanship). Your branch reflects all of these changes.

> Warning: Create your branch before making any changes.

Checklist:
[ ] Fork STUMPY to your own account
[ ] Clone a local copy

[ ] Create a branch for your work

5.4 Make your Changes

When working on a new project, there are often going to be dependencies. In order to isolate dependencies between different projects, it's a good practice to use a virtual environment. STUMPY supports both venv and conda. After creating and activating either of these virtual environments, any dependencies you install will be isolated (so they don't break anything else on your system).

Next, install the dependencies using the From Source Section of the instructions. A good check to make sure everything is working 100% is to run the unit tests. For STUMPY, we have scripts to help you do that. You'll run `./setup.sh && ./test.sh`. Ideally, you'll see the excellent STUMPY test coverage passing. If things start failing or breaking, you have a dependency problem. There's nothing worse than finding this out *after* you've made changes.

If all of the tests pass, you know that you have a working copy of STUMPY to start your development on. Go ahead and implement your feature or change!

Checklist:
[ ] Create a virtual environment
[ ] Install dependencies
[ ] Run the unit tests

5.5 Adhere to CONTRIBUTING.md Guidance

One of the great benefits of open source is the ability to collaborate with developers from around the world. However, you can imagine that combining their contributions into one coherent codebase while maintaining consistency can be challenging. Luckily, recent gains in automated tooling have made this a lot easier. Remember CONTRIBUTING.md? There are a couple of things we want to make sure we do before we submit a `pull request`.

First, if you implemented a new feature or changed an existing feature, you are also responsible for providing the unit test. This can often be just as much work as the feature, so make sure you account for it.

Next, run `flake8` and `black`. flake8 is a linter that checks the style and quality of the code. black makes any necessary changes to ensure consistent code format.

Checklist:
[ ] Write/update any unit tests
[ ] Run `black` to reformat any python files you changed
[ ] Run `flake8` to identify any formatting errors before you submit your `pull request`
[ ] Run the unit tests. In STUMPY, you'll run `./setup.sh && ./tests.sh`

5.6 Push your code to GitHub

Now your `fork` contains updated code and unit tests that adhere to the STUMPY formatting standards. If you're comfortable with `git` or have been doing work between multiple workstations, you may have `pushed` your `commits` to GitHub already. If not, now is the time!

When you were working on your change, you probably saved your files all the time. Those changes were saved on your local machine, but never updated in version control. To do that, you'll need to `git add <files>` ("hey git, please track these files"), `git commit -m <message>` ("hey git, at this point, I want to save") and `git push` ("Hey git, take the last commit and move it to GitHub") as described here. Remember to write a good commit message.

> Warning: If this is your first time using git, it may through a warning about global settings. Luckily, that warning tells you exactly how to fix it.

Checklist:
[ ] `git add your_file`
[ ] `git commit -m 'Great Commit Message'`
[ ] `git push`

5.7 Create a Pull Request

Now, your updated code is on GitHub in your fork of STUMPY. If you want the STUMPY maintainers to see it, you need to create a `Pull Request`. There are instructions here, but github is also pretty smart. If you navigate to your fork of the repository shortly after you push a commit, you'll often see a message like "Your recently pushed branches: X ". Clicking that will automatically create your pull request. Like a commit, make sure to give it a descriptive name. If your change isn't quite working yet, you can preface your pull request with `[WIP]` which stands for "Work in Progress." In the body of your Pull Request, reference the Issue Number that you're working on. GitHub will automatically link your pull request to that issue. This just makes housekeeping easier.

Here's your next chance to communicate with the maintainers. Let them know what changes you made and if you need any help. This pull request will now be the running dialogue between you and the maintainers as you work on the issue.

Continuous integration systems automatically determine the suitability of merging pull requests. They check the formatting, test coverage, and test success of your code. After you submit a pull request, you'll see these running (as comments in your pull request). If they fail, your code will not be merged until the failure is fixed. In STUMPY, locally passing `flake8`, `black`, and `./setup.sh && ./tests.sh` should ensure your continuous integration tests pass.

Checklist:
[ ] Create a Pull Request
[ ] Write an informative Pull Request Message
[ ] Monitor your Pull Request for continuous integration checks and messages from the maintainers

5.8 Work on your Pull Request

It is extremely likely that the maintainer will have questions, comments, or suggestions on your pull request. This is an opportunity for collaboration and should benefit both parties. Continue refining your contribution based on their feedback. Now, every commit you make will automatically be refected in this pull request; there's no need to create another. After you make any change in your local code, just `add`, `commit`, and `push` like before.

> Warning: After every change, make sure to run your formatting checks and unit tests.

Checklist:

[ ] Be open minded, responsive, and polite

5.9 Merge!

When the contribution is nice and polished, the maintainers will `merge` it into STUMPY. **Success!**

Checklist:
[ ] Celebrate
[ ] Go find another issue

5.10 What to do when it goes wrong

Invariably, something goes wrong. Remember, you're not in this alone. The open source community is a *community* and we appreciate your interest and contribution to STUMPY. If your barrier is related to an issue already filed in GitHub, that issue probably a great place to request additional information and help. If your barrier is related to your solution to an issue, go ahead and submit your progress as a pull request. As always, be polite and be patient.

5.11 Final Checklist:

[ ] Create GitHub account
[ ] Install Git
[ ] Identify an issue
[ ] Read CONTRIBUTING.md
[ ] Post your proposal in the issue
[ ] Fork STUMPY to your own account
[ ] Clone a local copy
[ ] Create a branch for your work
[ ] Create a virtual environment
[ ] Install dependencies
[ ] Run the unit tests
[ ] Write/update any unit tests
[ ] Run `black` to reformat any python files you changed
[ ] Run `flake8` to identify any formatting errors before you submit your `pull request`
[ ] Run the unit tests. In STUMPY, you'll run `./setup.sh && ./tests.sh`
[ ] `git add your_file`
[ ] `git commit -m 'Great Commit Message'`
[ ] `git push`
[ ] Create a Pull Request
[ ] Write an informative Pull Request Message
[ ] Monitor your Pull Request for continuous integration checks and messages from the maintainers
[ ] Be open minded, responsive, and polite
[ ] Celebrate
[ ] Go find another issue

CHAPTER 6

Getting Help

First, please check the discussions and issues on Github to see if your question has already been answered there. If no solution is available there feel free to open a new discussion or issue and the authors will attempt to respond in a reasonably timely fashion.


stumpy

CHAPTER 7

STUMPY

STUMPY is a powerful and scalable library that efficiently computes something called the matrix profile, which can be used for a variety of time series data mining tasks such as:

- pattern/motif (approximately repeated subsequences within a longer time series) discovery
- anomaly/novelty (discord) discovery
- shapelet discovery
- semantic segmentation
- streaming (on-line) data
- fast approximate matrix profiles
- time series chains (temporally ordered set of subsequence patterns)
- and more . . .

Whether you are an academic, data scientist, software developer, or time series enthusiast, STUMPY is straightforward to install and our goal is to allow you to get to your time series insights faster. See documentation for more information.

7.1 How to use STUMPY

Please see our API documentation for a complete list of available functions and see our informative tutorials for more comprehensive example use cases. Below, you will find code snippets that quickly demonstrate how to use STUMPY.

Typical usage (1-dimensional time series data) with STUMP:

```
import stumpy
import numpy as np

your_time_series = np.random.rand(10000)
window_size = 50  # Approximately, how many data points might be found in a pattern

matrix_profile = stumpy.stump(your_time_series, m=window_size)
```

Distributed usage for 1-dimensional time series data with Dask Distributed via STUMPED:

```
import stumpy
import numpy as np
from dask.distributed import Client
dask_client = Client()

your_time_series = np.random.rand(10000)
window_size = 50  # Approximately, how many data points might be found in a pattern

matrix_profile = stumpy.stumped(dask_client, your_time_series, m=window_size)
```

GPU usage for 1-dimensional time series data with GPU-STUMP:

```
import stumpy
import numpy as np
from numba import cuda

your_time_series = np.random.rand(10000)
window_size = 50  # Approximately, how many data points might be found in a pattern
all_gpu_devices = [device.id for device in cuda.list_devices()]  # Get a list of all␣
↪available GPU devices

matrix_profile = stumpy.gpu_stump(your_time_series, m=window_size, device_id=all_gpu_
↪devices)
```

Multi-dimensional time series data with MSTUMP:

```
import stumpy
import numpy as np

your_time_series = np.random.rand(3, 1000)  # Each row represents data from a␣
↪different dimension while each column represents data from the same dimension
window_size = 50  # Approximately, how many data points might be found in a pattern

matrix_profile, matrix_profile_indices = stumpy.mstump(your_time_series, m=window_
↪size)
```

Distributed multi-dimensional time series data analysis with Dask Distributed MSTUMPED:

```
import stumpy
import numpy as np
from dask.distributed import Client
dask_client = Client()

your_time_series = np.random.rand(3, 1000)   # Each row represents data from a␣
↪different dimension while each column represents data from the same dimension
window_size = 50  # Approximately, how many data points might be found in a pattern

matrix_profile, matrix_profile_indices = stumpy.mstumped(dask_client, your_time_
↪series, m=window_size)
```

Time Series Chains with Anchored Time Series Chains (ATSC):

```
import stumpy
import numpy as np

your_time_series = np.random.rand(10000)
```

(continues on next page)

(continued from previous page)

```
window_size = 50  # Approximately, how many data points might be found in a pattern

matrix_profile = stumpy.stump(your_time_series, m=window_size)

left_matrix_profile_index = matrix_profile[:, 2]
right_matrix_profile_index = matrix_profile[:, 3]
idx = 10  # Subsequence index for which to retrieve the anchored time series chain for

anchored_chain = stumpy.atsc(left_matrix_profile_index, right_matrix_profile_index,␣
↪idx)

all_chain_set, longest_unanchored_chain = stumpy.allc(left_matrix_profile_index,␣
↪right_matrix_profile_index)
```

Semantic Segmentation with Fast Low-cost Unipotent Semantic Segmentation (FLUSS):

```
import stumpy
import numpy as np

your_time_series = np.random.rand(10000)
window_size = 50  # Approximately, how many data points might be found in a pattern

matrix_profile = stumpy.stump(your_time_series, m=window_size)

subseq_len = 50
correct_arc_curve, regime_locations = stumpy.fluss(matrix_profile[:, 1],
                                                   L=subseq_len,
                                                   n_regimes=2,
                                                   excl_factor=1
                                                  )
```

7.2 Dependencies

Supported Python and NumPy versions are determined according to the NEP 29 deprecation policy.

- NumPy
- Numba
- SciPy

7.3 Where to get it

Conda install (preferred):

```
conda install -c conda-forge stumpy
```

PyPI install, presuming you have numpy, scipy, and numba installed:

```
python -m pip install stumpy
```

To install stumpy from source, see the instructions in the documentation.

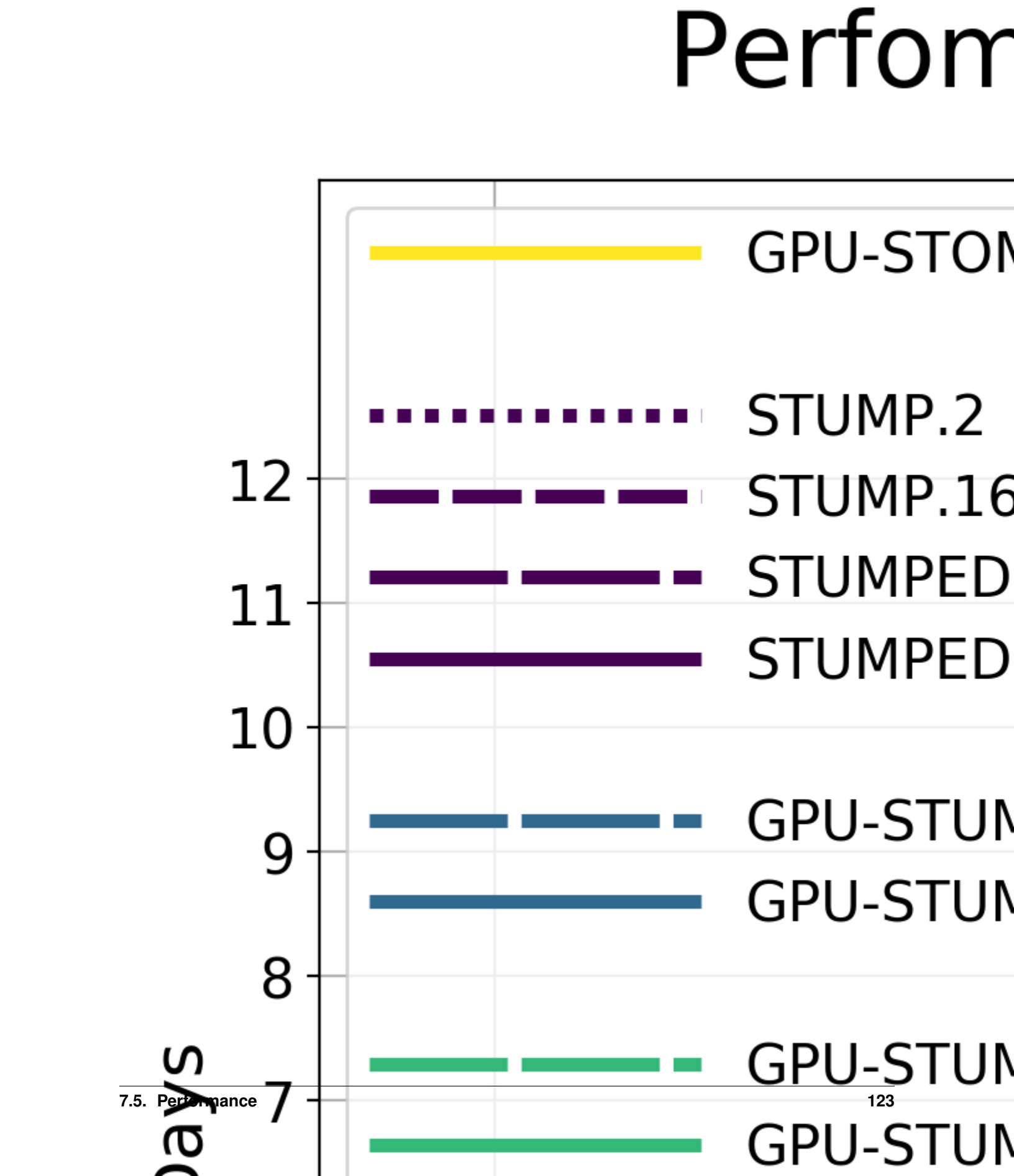
7.4 Documentation

In order to fully understand and appreciate the underlying algorithms and applications, it is imperative that you read the original *publications*. For a more detailed example of how to use STUMPY please consult the latest documentation or explore the following tutorials:

1. The Matrix Profile
2. STUMPY Basics
3. Time Series Chains
4. Semantic Segmentation

7.5 Performance

We tested the performance of computing the exact matrix profile using the Numba JIT compiled version of the code on randomly generated time series data with various lengths (i.e., `np.random.rand(n)`) along with different *CPU and GPU hardware resources*.


Perfom
GPU-STOM
STUMP.2
STUMP.16
STUMPED
STUMPED
GPU-STUM
GPU-STUM
GPU-STUM
GPU-STUM
12
11
10
9
8
7
DaYs

The raw results are displayed in the table below as Hours:Minutes:Seconds.Milliseconds and with a constant window size of $m = 50$. Note that these reported runtimes include the time that it takes to move the data from the host to all of the GPU device(s). You may need to scroll to the right side of the table in order to see all of the runtimes.

i	n = 2^i	GPU-STOMP	STUMP.2	STUMP.16	STUMPED.128	STUMPED.256	GPU-STUMP.1	GPU-STUMP.2	GPU-STUMP.DGX1	GPU-STUMP.DGX2
6	64	00:00:10.00	00:00:00.00	00:00:00.00	00:00:05.77	00:00:06.08	00:00:00.03	00:00:01.63	NaN	NaN
7	128	00:00:10.00	00:00:00.00	00:00:00.00	00:00:05.93	00:00:07.29	00:00:00.04	00:00:01.66	NaN	NaN
8	256	00:00:10.00	00:00:00.00	00:00:00.01	00:00:05.95	00:00:07.59	00:00:00.08	00:00:01.69	00:00:06.68	00:00:25.68
9	512	00:00:10.00	00:00:00.00	00:00:00.02	00:00:05.97	00:00:07.47	00:00:00.13	00:00:01.66	00:00:06.59	00:00:27.66
10	1024	00:00:10.00	00:00:00.00	00:00:00.04	00:00:05.69	00:00:07.64	00:00:00.24	00:00:01.72	00:00:06.70	00:00:30.49
11	2048	NaN	00:00:00.05	00:00:00.09	00:00:05.60	00:00:07.83	00:00:00.53	00:00:01.88	00:00:06.87	00:00:31.09
12	4096	NaN	00:00:00.22	00:00:00.19	00:00:06.26	00:00:07.90	00:00:01.04	00:00:02.19	00:00:06.91	00:00:33.93
13	8192	NaN	00:00:00.86	00:00:00.41	00:00:06.29	00:00:07.73	00:00:01.97	00:00:02.49	00:00:06.61	00:00:33.81
14	16384	NaN	00:00:01.70	00:00:00.99	00:00:06.24	00:00:08.18	00:00:03.69	00:00:03.29	00:00:07.36	00:00:35.23
15	32768	NaN	00:00:06.07	00:00:02.89	00:00:06.48	00:00:08.29	00:00:07.45	00:00:04.93	00:00:07.02	00:00:36.09
16	65536	NaN	00:00:22.94	00:00:06.42	00:00:07.33	00:00:09.01	00:00:14.89	00:00:08.12	00:00:08.10	00:00:36.54
17	131072	00:00:10.00	00:01:29.27	00:00:19.62	00:00:09.75	00:00:10.53	00:00:29.97	00:00:15.42	00:00:09.45	00:00:37.33
18	262144	00:00:18.00	00:05:56.50	00:01:08.44	00:00:33.38	00:00:24.07	00:00:59.62	00:00:27.41	00:00:13.18	00:00:39.30
19	524288	00:00:46.00	00:25:34.58	00:03:56.82	00:01:35.27	00:03:43.66	00:01:56.67	00:00:54.05	00:00:19.65	00:00:41.45
20	1048576	00:02:30.00	01:51:13.40	00:19:54.75	00:04:37.15	00:03:01.16	00:05:06.48	00:02:24.73	00:00:32.95	00:00:46.14
21	2097152	00:09:15.00	09:25:47.63	03:05:07.64	00:13:36.51	00:08:47.47	00:20:27.94	00:09:41.43	00:01:06.51	00:01:02.67
22	4194304	NaN	36:12:23.74	10:37:51.02	01:55:44.43	00:32:06.70	01:21:12.33	00:38:30.86	00:04:03.26	00:02:23.47
23	8388608	NaN	143:16:09.84	39:42:51.03	07:33:30.53	02:00:49.37	05:11:44.45	02:33:14.60	00:15:46.26	00:08:03.76
24	16777216	NaN	NaN	NaN	14:39:11.99	07:13:47.12	20:43:03.80	09:48:43.42	01:00:24.06	00:29:07.84
NaN	17729800	09:16:12.00	NaN	NaN	15:31:31.75	07:18:42.54	23:09:22.43	10:54:08.64	01:07:35.39	00:32:51.55
25	33554432	NaN	NaN	NaN	56:03:46.81	26:27:41.29	83:29:21.06	39:17:43.82	03:59:32.79	01:54:56.52
26	67108864	NaN	NaN	NaN	211:17:37.60	106:40:17.13	328:58:04.68	157:18:30.56	15:42:15.94	07:18:52.91
NaN	100000000	291:07:12.00	NaN	NaN	NaN	234:51:35.39	NaN	NaN	35:03:44.61	16:22:40.81
27	134217728	NaN	NaN	NaN	NaN	NaN	NaN	NaN	64:41:55.09	29:13:48.12

7.5.1 Hardware Resources

GPU-STOMP: These results are reproduced from the original Matrix Profile II paper - NVIDIA Tesla K80 (contains 2 GPUs) and serves as the performance benchmark to compare against.

STUMP.2: stumpy.stump executed with 2 CPUs in Total - 2x Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz processors parallelized with Numba on a single server without Dask.

STUMP.16: stumpy.stump executed with 16 CPUs in Total - 16x Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz processors parallelized with Numba on a single server without Dask.

STUMPED.128: stumpy.stumped executed with 128 CPUs in Total - 8x Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz processors x 16 servers, parallelized with Numba, and distributed with Dask Distributed.

STUMPED.256: stumpy.stumped executed with 256 CPUs in Total - 8x Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz processors x 32 servers, parallelized with Numba, and distributed with Dask Distributed.

GPU-STUMP.1: stumpy.gpu_stump executed with 1x NVIDIA GeForce GTX 1080 Ti GPU, 512 threads per block, 200W power limit, compiled to CUDA with Numba, and parallelized with Python multiprocessing

GPU-STUMP.2: stumpy.gpu_stump executed with 2x NVIDIA GeForce GTX 1080 Ti GPU, 512 threads per block, 200W power limit, compiled to CUDA with Numba, and parallelized with Python multiprocessing

GPU-STUMP.DGX1: stumpy.gpu_stump executed with 8x NVIDIA Tesla V100, 512 threads per block, compiled to CUDA with Numba, and parallelized with Python multiprocessing

GPU-STUMP.DGX2: stumpy.gpu_stump executed with 16x NVIDIA Tesla V100, 512 threads per block, compiled to CUDA with Numba, and parallelized with Python multiprocessing

7.6 Running Tests

Tests are written in the `tests` directory and processed using PyTest and requires `coverage.py` for code coverage analysis. Tests can be executed with:

```
./test.sh
```

7.7 Python Version

STUMPY supports Python 3.7+ and, due to the use of unicode variable names/identifiers, is not compatible with Python 2.x. Given the small dependencies, STUMPY may work on older versions of Python but this is beyond the scope of our support and we strongly recommend that you upgrade to the most recent version of Python.

7.8 Getting Help

First, please check the discussions and issues on Github to see if your question has already been answered there. If no solution is available there feel free to open a new discussion or issue and the authors will attempt to respond in a reasonably timely fashion.

7.9 Contributing

We welcome contributions in any form! Assistance with documentation, particularly expanding tutorials, is always welcome. To contribute please fork the project, make your changes, and submit a pull request. We will do our best to work through any issues with you and get your code merged into the main branch.

7.10 Citing

If you have used this codebase in a scientific publication and wish to cite it, please use the Journal of Open Source Software article.

> S.M. Law, (2019). *STUMPY: A Powerful and Scalable Python Library for Time Series Data Mining*. Journal of Open Source Software, 4(39), 1504.

```
@article{law2019stumpy,
  title={{STUMPY: A Powerful and Scalable Python Library for Time Series Data Mining}}
↪,
  author={Law, Sean M.},
  journal={{The Journal of Open Source Software}},
  volume={4},
  number={39},
```

(continues on next page)

(continued from previous page)

```
  pages={1504},
  year={2019}
}
```

7.11 References

Yeh, Chin-Chia Michael, et al. (2016) Matrix Profile I: All Pairs Similarity Joins for Time Series: A Unifying View that Includes Motifs, Discords, and Shapelets. ICDM:1317-1322. Link

Zhu, Yan, et al. (2016) Matrix Profile II: Exploiting a Novel Algorithm and GPUs to Break the One Hundred Million Barrier for Time Series Motifs and Joins. ICDM:739-748. Link

Yeh, Chin-Chia Michael, et al. (2017) Matrix Profile VI: Meaningful Multidimensional Motif Discovery. ICDM:565-574. Link

Zhu, Yan, et al. (2017) Matrix Profile VII: Time Series Chains: A New Primitive for Time Series Data Mining. ICDM:695-704. Link

Gharghabi, Shaghayegh, et al. (2017) Matrix Profile VIII: Domain Agnostic Online Semantic Segmentation at Superhuman Performance Levels. ICDM:117-126. Link

Zhu, Yan, et al. (2017) Exploiting a Novel Algorithm and GPUs to Break the Ten Quadrillion Pairwise Comparisons Barrier for Time Series Motifs and Joins. KAIS:203-236. Link

Zhu, Yan, et al. (2018) Matrix Profile XI: SCRIMP++: Time Series Motif Discovery at Interactive Speeds. ICDM:837-846. Link

Yeh, Chin-Chia Michael, et al. (2018) Time Series Joins, Motifs, Discords and Shapelets: a Unifying View that Exploits the Matrix Profile. Data Min Knowl Disc:83-123. Link

Gharghabi, Shaghayegh, et al. (2018) "Matrix Profile XII: MPdist: A Novel Time Series Distance Measure to Allow Data Mining in More Challenging Scenarios." ICDM:965-970. Link

Zimmerman, Zachary, et al. (2019) Matrix Profile XIV: Scaling Time Series Motif Discovery with GPUs to Break a Quintillion Pairwise Comparisons a Day and Beyond. SoCC '19:74-86. Link

Akbarinia, Reza, and Betrand Cloez. (2019) Efficient Matrix Profile Computation Using Different Distance Functions. arXiv:1901.05708. Link

Kamgar, Kaveh, et al. (2019) Matrix Profile XV: Exploiting Time Series Consensus Motifs to Find Structure in Time Series Sets. ICDM:1156-1161. Link

7.12 License & Trademark

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Index