

---

# **TrendyPy**

***Release 0.2.1***

**Dogan Askan**

**Sep 24, 2020**



# CONTENTS

<b>1</b>	<b>TrendyPy</b>	<b>1</b>
1.1	Installation . . . . .	1
1.2	Quickstart . . . . .	1
1.3	Post . . . . .	2
<b>2</b>	<b>See in Action</b>	<b>3</b>
2.1	Stock Data . . . . .	3
2.2	Image Clustering . . . . .	7
2.3	Custom Metric . . . . .	8
<b>3</b>	<b>API Reference</b>	<b>11</b>
3.1	trendy . . . . .	11
3.2	algos . . . . .	13
3.3	utils . . . . .	15
<b>4</b>	<b>Indices and tables</b>	<b>17</b>
	<b>Python Module Index</b>	<b>19</b>
	<b>Index</b>	<b>21</b>



## TRENDYPY

TrendyPy is a small Python package for sequence clustering. It is initially developed to create time series clusters by calculating trend similarity distance with [Dynamic Time Warping](#).

## 1.1 Installation

You can install TrendyPy with pip.

```
pip install trendy
```

TrendyPy depends on Pandas, Numpy and fastdtw and works in Python 3.6+.

## 1.2 Quickstart

Trendy has scikit-learn like api to allow easy integration to existing programs. Below is a quick example to show how it clusters increasing and decreasing trends.

```
>>> from trendy.trendy import Trendy
>>> a = [1, 2, 3, 4, 5] # increasing trend
>>> b = [1, 2.1, 2.9, 4.4, 5.1] # increasing trend
>>> c = [6.2, 5, 4, 3, 2] # decreasing trend
>>> d = [7, 6, 5, 4, 3, 2, 1] # decreasing trend
>>> trendy = Trendy(n_clusters=2)
>>> trendy.fit([a, b, c, d])
>>> print(trendy.labels_)
[0, 0, 1, 1]
>>> trendy.predict([[0.9, 2, 3.1, 4]]) # another increasing trend
[0]
```

It can also be utilized to cluster strings by using string similarity metrics.

```
>>> from trendy.trendy import Trendy
>>> from trendy.algos import levenshtein_distance
>>> company_names = [
...     'apple inc',
...     'Apple Inc.',
...     'Microsoft Corporation',
...     'Microsft Corp.']
>>> trendy = Trendy(n_clusters=2, algorithm=levenshtein_distance)
```

(continues on next page)

(continued from previous page)

```
>>> trendy.fit(company_names)
>>> print(trendy.labels_)
[0, 0, 1, 1]
>>> trendy.predict(['Apple'])
[0]
```

Refer to [extensive demo](#) to see it in clustering [stock trends](#), [images](#) or to see how to [define your own metric](#) or just check [API Reference](#) for details.

## 1.3 Post

The idea is originated from the post [Trend Clustering](#).

## SEE IN ACTION

Let's see how TrendyPy works with a few use cases.

### 2.1 Stock Data

In this demo, I'd like to show you how to use TrendyPy in some `stock` data between 2018-01-01 and 2020-06-28. You can download the data from [here](#) to reproduce the demo.

Let's say we have some stock data from a combination of tech and banking. And, we want to identify an unknown trend if it's a tech stock or banking. For this purpose, we'll use FB (i.e. Facebook), GOOGL (i.e. Google), AMZN (i.e. Amazon), BAC (i.e. Bank of America) and WFC (i.e. Wells Fargo) for training data then AAPL (i.e. Apple) and c (i.e. Citigroup) for prediction data.

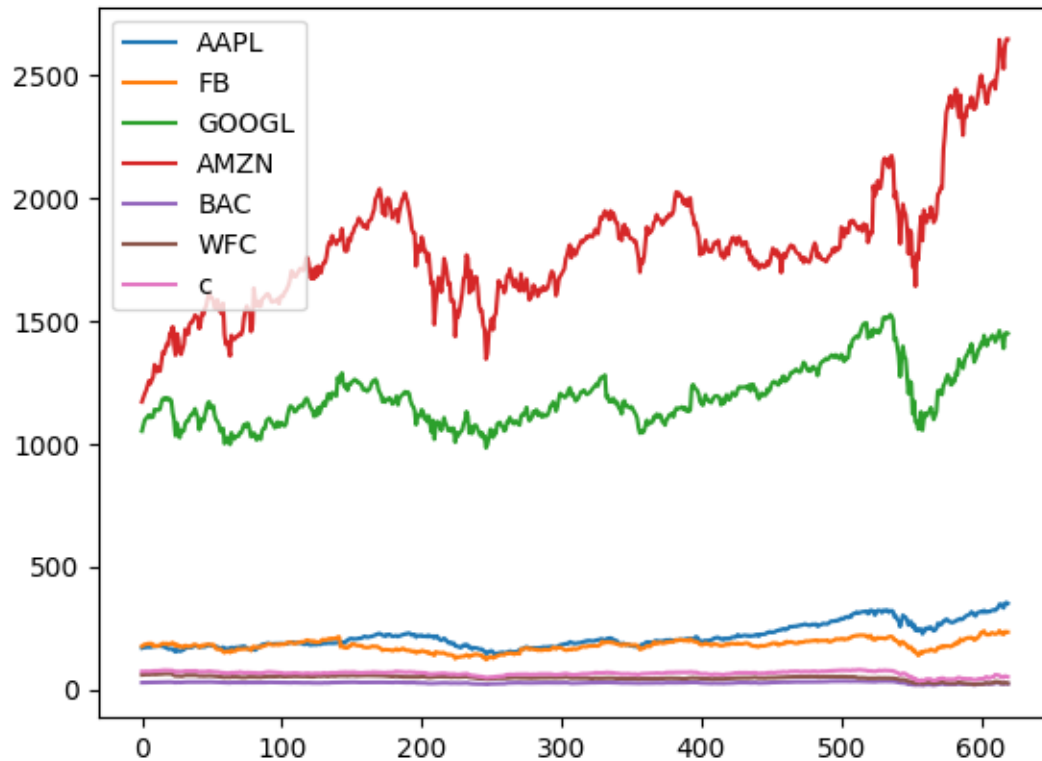
But first, here is how the data looks.

```
In [1]: import pandas as pd

In [2]: import matplotlib.pyplot as plt

In [3]: df = pd.read_csv('stock_data.csv')

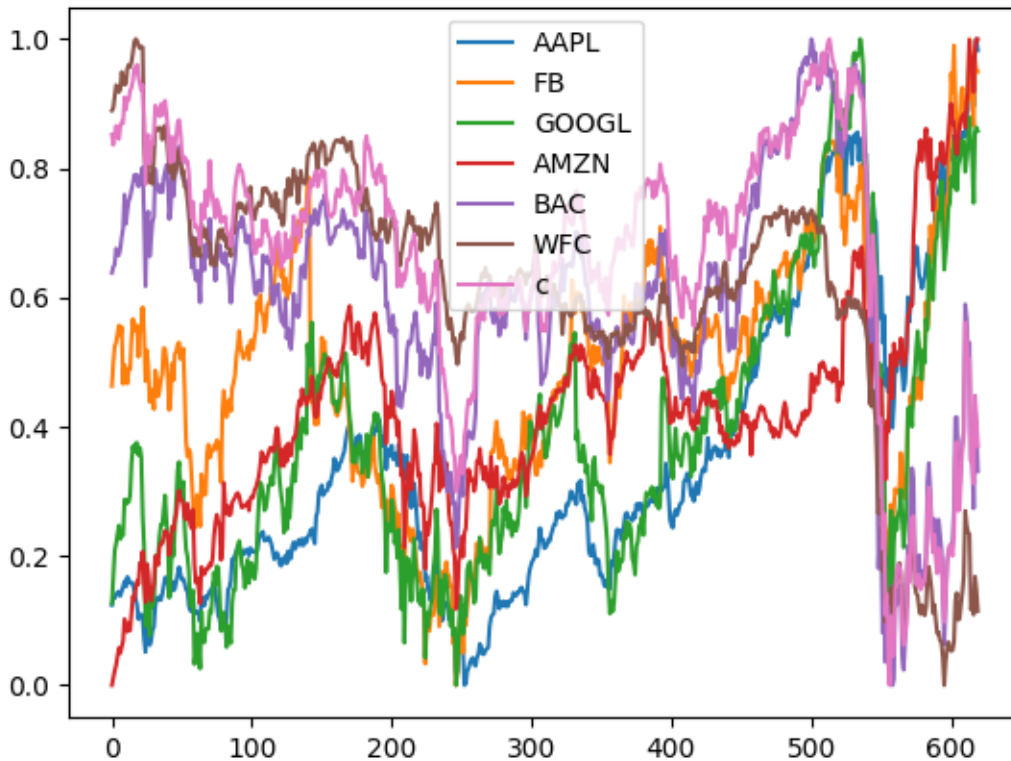
In [4]: df.plot()
Out[4]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3f35dfcd30>
```



If we cluster like this, the expensive stocks like GOOGL and AMZN will alone constitute one cluster which it's clearly not intended. So, let's scale first.

```
In [5]: from trendy import utils
In [6]: df = df.apply(utils.scale_01)
In [7]: df.plot()
Out[7]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3f3585c860>
```





It's a bit apparent that BAC, WFC and c are different than the others. Let's put sectors side by side to see the difference better.

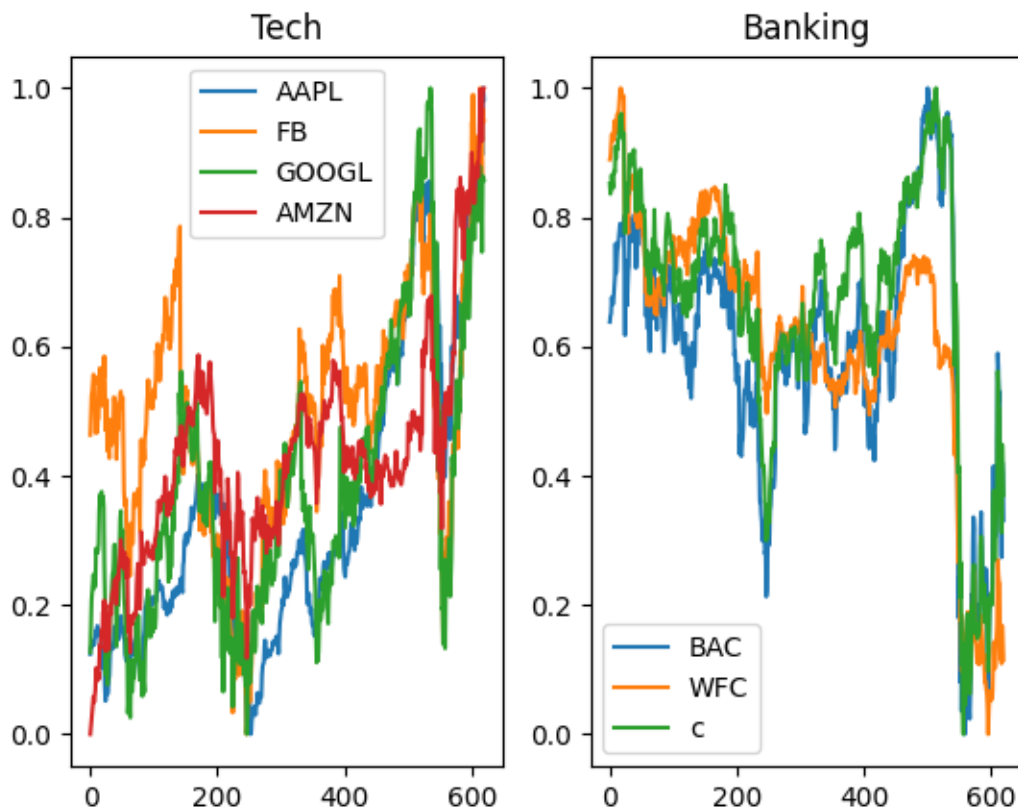
```
In [8]: fig, axes_ = plt.subplots(nrows=1, ncols=2)

In [9]: axes_[0].set_title('Tech')
Out[9]: Text(0.5, 1.0, 'Tech')

In [10]: axes_[1].set_title('Banking')
Out[10]: Text(0.5, 1.0, 'Banking')

In [11]: df[['AAPL', 'FB', 'GOOGL', 'AMZN']].plot(ax=axes_[0])
Out[11]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3f357ee6a0>

In [12]: df[['BAC', 'WFC', 'c']].plot(ax=axes_[1])
Out[12]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3f3579bc88>
```



Now, we can use the training data to fit. Remember, we're setting AAPL and c aside to predict later and only fit by using the rest.

```
In [13]: from trendy.trendy import Trendy
In [14]: trendy = Trendy(n_clusters=2) # 2 for tech and banking
In [15]: trendy.fit([df.FB, df.GOOGL, df.AMZN, df.BAC, df.WFC])
In [16]: trendy.labels_
Out[16]: [0, 0, 0, 1, 1]
```

You can also use `fit_predict` method for this purpose, it's essentially the same.

```
In [17]: trendy.fit_predict([df.FB, df.GOOGL, df.AMZN, df.BAC, df.WFC])
Out[17]: [0, 0, 0, 1, 1]
```

As expected, it successfully assigns FB, GOOGL and AMZN into the first cluster (i.e. 0) and BAC and WFC into the second (i.e. 1). So, we can name 0 as tech and 1 as banking.

Now, let's make predictions on the prediction data that we set aside earlier (i.e. AAPL, c).

```
In [18]: trendy.predict([df.AAPL]) # expecting `0` since AAPL is a part of tech
Out[18]: [0]
In [19]: trendy.predict([df.c]) # expecting `1` since c is a part of banking
```

(continues on next page)

(continued from previous page)

```
Out [19]: [1]
```

As seen above, it correctly predicts trends.

You can easily pickle the model object to be used later with `to_pickle` method.

```
In [20]: trendy.to_pickle('my_first_trendy.pkl')
```

And, that's all.

## 2.2 Image Clustering

If you have the proper distance metric function for the right data, you can use TrendyPy to even cluster images. In this demo, I'll use black & white images from [MPEG7 CE Shape-1 Part B](#) database. The goal is to correctly cluster the images and assign new ones to the appropriate clusters. Here are some simple images that'll be used to create the clusters. Each image is slightly different than the others in the same group. You can download the images if you want to reproduce the demo.

Fig. 1: car-01.gif	Fig. 2: car-02.gif	Fig. 3: car-03.gif
Fig. 4: carriage-02.gif	Fig. 5: carriage-03.gif	Fig. 6: carriage-04.gif
Fig. 7: chopper-01.gif	Fig. 8: chopper-02.gif	Fig. 9: chopper-03.gif

Define a function to read the image and convert to a numpy array.

```
In [21]: from PIL import Image
In [22]: import numpy as np
In [23]: def load_image(file):
.....:     img = Image.open(file)
.....:     img.load()
.....:     return np.asarray(img, dtype="int32")
.....:
```

Read images and assign them into lists.

```
In [24]: cars = [
.....:     load_image('image_data/car-01.gif'),
.....:     load_image('image_data/car-02.gif'),
.....:     load_image('image_data/car-03.gif')]
.....:

In [25]: carriages = [
.....:     load_image('image_data/carriage-02.gif'),
.....:     load_image('image_data/carriage-03.gif'),
.....:     load_image('image_data/carriage-04.gif')]
.....:

In [26]: choppers = [
.....:     load_image('image_data/chopper-01.gif'),
.....:     load_image('image_data/chopper-02.gif'),
```

(continues on next page)

(continued from previous page)

```
.....: load_image('image_data/chopper-03.gif')
.....:
```

Euclidean Distance can be used to calculate the similarity between images. So, let's import `euclidean_distance` from `utils` module, then assign it as `algorithm` argument during the initialization.

```
In [27]: from trendy.trendy import Trendy
In [28]: from trendy.utils import euclidean_distance
In [29]: trendy = Trendy(n_clusters=3, algorithm=euclidean_distance)
In [30]: trendy.fit(cars + carriages + choppers)
In [31]: trendy.labels_
Out[31]: [0, 0, 0, 1, 1, 1, 2, 2, 2]
```

As expected, it correctly clusters these simple images. Let's see if it predicts new data correctly.

Fig. 10: car-20.gif	Fig. 11: carriage-20.gif	Fig. 12: chopper-08.gif
---------------------	--------------------------	-------------------------

```
In [32]: new_car = load_image('image_data/car-20.gif')
In [33]: new_carriage = load_image('image_data/carriage-20.gif')
In [34]: new_chopper = load_image('image_data/chopper-08.gif')
In [35]: trendy.predict([new_car, new_carriage, new_chopper])
Out[35]: [0, 1, 2]
```

Looks like it correctly predicts new data as well.

**Note:** Because of the limitation of the selected metric function (i.e. [Euclidean Distance](#)), I had to cherry pick images with exact same sizes (i.e. 352×288). Depending on the function you choose, you may or may not do the same.

## 2.3 Custom Metric

TrendyPy is flexible enough to be able to utilize user defined metrics. In this example, I'll show how to create your own metric and use it during the clustering.

Let's say we want to cluster DNA sequences and need a metric to do that. [Needleman–Wunsch algorithm](#) is an algorithm used in bioinformatics to align protein or nucleotide sequences. It's not a metric but it inspires us to create our own metric. The metric basically compares two sequences with same length and it penalizes each mismatch by increasing the distance by  $p$  then divides it to total length.

```
In [36]: def my_metric(x, y, p=1):
.....:     assert len(x) == len(y)
.....:     dist = 0
.....:     for i in range(len(x)):
.....:         if x[i] != y[i]:
```

(continues on next page)

(continued from previous page)

```

.....:         dist += p
.....:     return dist/len(x)
.....:

```

As you can see, you just need to consider inputs and output of your custom function. Specifically,

1. Input must have  $x$  and  $y$  for two data points to compare. You may have other default arguments (e.g.  $p$ ).
2. Output must be a float. 0 indicates same and greater is farther.

**Note:** Technically, any float range should work as the output of the custom function as long as greater is farther. However, it won't be named as *metric* in that case.

Anyway, let's use it.

```

In [37]: set_of_sequences = [
.....:     'AAATTT', 'AAACTT', 'AAATCT', # group 1
.....:     'GACTAG', 'GGCTAG', 'GACAAG' # group 2
.....: ]
.....:

```

```

In [38]: from trendy.trendy import Trendy

In [39]: trendy = Trendy(
.....:     n_clusters=2, # there are 2 groups
.....:     algorithm=my_metric # this is where to set custom metric
.....: )
.....:

In [40]: trendy.fit(set_of_sequences)

In [41]: trendy.labels_
Out[41]: [0, 0, 0, 1, 1, 1]

```

It clearly clusters first and second group. Now, let's see on new data.

```

In [42]: new_seq1 = 'AAAGGT' # similar to group 1

In [43]: new_seq2 = 'GTCCAG' # similar to group 2

In [44]: trendy.predict([new_seq1, new_seq2])
Out[44]: [0, 1]

```

Very simple.



## API REFERENCE

### 3.1 trendy

**class** trendy.Trendy(*n\_clusters*, *algorithm*=<function fastdtw\_distance>)

Bases: object

Estimator to cluster trend-lines and assign new lines accordingly.

#### Notes

Scaling and missing values need to be handled externally.

#### Parameters

- **n\_clusters** (*int*) – The number of clusters to form.
- **algorithm** (*callable*) – Algorithm to calculate the difference. Default is [fast DTW](#) with Euclidean.

#### Example

```
>>> a = [1, 2, 3, 4, 5] # increasing trend
>>> b = [1, 2.1, 2.9, 4.4, 5.1] # increasing trend
>>> c = [6.2, 5, 4, 3, 2] # decreasing trend
>>> d = [7, 6, 5, 4, 3, 2, 1] # decreasing trend
>>> trendy = Trendy(n_clusters=2)
>>> trendy.fit([a, b, c, d])
>>> print(trendy.labels_)
[0, 0, 1, 1]
>>> trendy.predict([[0.9, 2, 3.1, 4]]) # another increasing trend
[0]
```

**fit** (*X*)

Compute clustering based on given distance algorithm.

**Parameters** *X* (*array of arrays*) – Training instances to cluster.

### Example

```
>>> a = [1, 2, 3, 4, 5] # increasing
>>> b = [1, 2.1, 2.9, 4.4, 5.1] # increasing
>>> c = [6.2, 5, 4, 3, 2] # decreasing
>>> d = [7, 6, 5, 4, 3, 2, 1] # decreasing
>>> trendy = Trendy(2)
>>> trendy.fit([a, b, c, d])
>>> print(trendy.labels_)
[0, 0, 1, 1]
```

#### **predict** (*X*)

Predict the closest cluster each sample in *X* belongs to.

**Parameters** *X* (*array of arrays*) – New data to predict.

**Returns** Index of the cluster each sample belongs to.

**Return type** list

### Example

```
>>> a = [1, 2, 3, 4, 5] # increasing
>>> b = [1, 2.1, 2.9, 4.4, 5.1] # increasing
>>> c = [6.2, 5, 4, 3, 2] # decreasing
>>> d = [7, 6, 5, 4, 3, 2, 1] # decreasing
>>> trendy = Trendy(2)
>>> trendy.fit([a, b, c, d])
>>> trendy.predict([[0.9, 2, 3.1, 4]])
[0]
>>> trendy.predict([[0.9, 2, 3.1], [7, 6.6, 5.5, 4.4]])
[0, 1]
```

#### **assign** (*X*)

Alias of *predict()*

#### **fit\_predict** (*X*)

Compute cluster centers and predict cluster index for each sample.

**Parameters** *X* (*array of arrays*) – Training instances to cluster.

**Returns** predicted labels

**Return type** list

### Example

```
>>> a = [1, 2, 3, 4, 5] # increasing
>>> b = [1, 2.1, 2.9, 4.4, 5.1] # increasing
>>> c = [6.2, 5, 4, 3, 2] # decreasing
>>> d = [7, 6, 5, 4, 3, 2, 1] # decreasing
>>> trendy = Trendy(2)
>>> trendy.fit_predict([a, b, c, d])
[0, 0, 1, 1]
```

#### **to\_pickle** (*path*)

Pickle (serialize) object to a file.



**Parameters** `path` (*str*) – file path where the pickled object will be stored

### Example

To save a *\*.pkl* file:

```
>>> t1 = Trendy(n_clusters=2)
>>> t1.fit([[1, 2, 3], [2, 3, 3]])
>>> t1.to_pickle(path='trendy.pkl')
```

To load the same object later:

```
>>> import pickle, os
>>> pkl_file = open('trendy.pkl', 'rb')
>>> t2 = pickle.load(pkl_file)
>>> pkl_file.close()
>>> os.remove('trendy.pkl')
```

## 3.2 algos

Algorithms for the package.

`algos.dtw_distance` (*x*, *y*, *d*=<function *euclidean\_distance*>, *scaled*=*False*)

Returns the distance of two arrays with dynamic time warping method.

#### Parameters

- ***x*** (*iter*) – input array 1
- ***y*** (*iter*) – input array 2
- ***d*** (*func*) – distance function, default is euclidean
- ***scaled*** (*bool*) – should arrays be scaled (i.e. 0-1) before calculation

**Returns** distance, 0.0 means arrays are exactly same, upper limit is positive infinity

**Return type** float

### References

[https://en.wikipedia.org/wiki/Dynamic\\_time\\_warping](https://en.wikipedia.org/wiki/Dynamic_time_warping)

### Examples

```
>>> dtw_distance([1, 2, 3, 4], [1, 2, 3, 4])
0.0
>>> dtw_distance([1, 2, 3, 4], [0, 0, 0])
10.0
>>> dtw_distance([1, 2, 3, 4], [0, 2, 0, 4])
4.0
>>> dtw_distance([1, 2, 3, 4], [10, 20, 30, 40])
90.0
>>> dtw_distance([1, 2, 3, 4], [10, 20, 30, 40], scaled=True)
0.0
```

`algos.fastdtw_distance(x, y, d=<function euclidean_distance>)`

Dynamic Time Warping (DTW) algorithm with an O(N) time and memory complexity.

**Parameters**

- **x** (*iter*) – input array 1
- **y** (*iter*) – input array 2
- **d** (*func*) – distance function, default is euclidean

**Returns** distance, 0.0 means arrays are exactly same, upper limit is positive infinity

**Return type** float

**References**

<https://pypi.org/project/fastdtw/>

**Examples**

```
>>> fastdtw_distance([1, 2, 3, 4], [1, 2, 3, 4])
0.0
>>> fastdtw_distance([1, 2, 3, 4], [0, 0, 0])
10.0
>>> fastdtw_distance([1, 2, 3, 4], [0, 2, 0, 4])
4.0
>>> fastdtw_distance([1, 2, 3, 4], [10, 20, 30, 40])
90.0
```

`algos.levenshtein_distance(x, y)`

Levenshtein distance for string similarity.

**Parameters**

- **x** (*str*) – input string 1
- **y** (*str*) – input string 2

**Returns** distance, 0 means strings are exactly same, upper limit is positive infinity

**Return type** int

**References**

[https://en.wikipedia.org/wiki/Levenshtein\\_distance](https://en.wikipedia.org/wiki/Levenshtein_distance)

**Examples**

```
>>> levenshtein_distance('Apple', 'Apple')
0
>>> levenshtein_distance('Apple', 'apple')
1
>>> levenshtein_distance('Apple Inc.', 'apple inc')
3
```

### 3.3 utils

Utility functions for the package.

`utils.scale_01(x)`

Scales array to 0-1.

**Parameters** `x` (*iter*) – 1d array of float

**Returns** scaled 1d array

**Return type** `np.array`

#### Example

```
>>> scale_01([1, 2, 3, 5]).tolist()
[0.0, 0.25, 0.5, 1.0]
```

`utils.abs_distance(x, y)`

Returns absolute distance.

**Parameters**

- `x` (*float*) – input 1
- `y` (*float*) – input 2

**Returns** `|x-y|`

**Return type** `float`

#### Example

```
>>> abs_distance(5, 7)
2.0
>>> abs_distance(4, 1)
3.0
```

`utils.euclidean_distance(x, y)`

Returns Euclidean distance.

**Parameters**

- `x` (*float or iter*) – input 1
- `y` (*float or iter*) – input 2

**Returns** Euclidean distance

**Return type** `float`

## References

<https://numpy.org/doc/stable/reference/generated/numpy.linalg.norm.html>

## Examples

```
>>> x, y = 1, 2
>>> euclidean_distance(x, y)
1.0
>>> x, y = [1, 2], [4, 6]
>>> euclidean_distance(x, y)
5.0
```

## INDICES AND TABLES

- `genindex`
- `modindex`
- `search`



## PYTHON MODULE INDEX

### a

algos, [13](#)

### t

trendy, [11](#)

### u

utils, [15](#)





## A

`abs_distance()` (*in module utils*), 15  
`algos`  
    module, 13  
`assign()` (*trendy.Trendy method*), 12

## D

`dtw_distance()` (*in module algos*), 13

## E

`euclidean_distance()` (*in module utils*), 15

## F

`fastdtw_distance()` (*in module algos*), 13  
`fit()` (*trendy.Trendy method*), 11  
`fit_predict()` (*trendy.Trendy method*), 12

## L

`levenshtein_distance()` (*in module algos*), 14

## M

module  
    algos, 13  
    trendy, 11  
    utils, 15

## P

`predict()` (*trendy.Trendy method*), 12

## S

`scale_01()` (*in module utils*), 15

## T

`to_pickle()` (*trendy.Trendy method*), 12  
`trendy`  
    module, 11  
`Trendy` (*class in trendy*), 11

## U

`utils`  
    module, 15