Time series analysis: Time series classification and clustering

ΕΠΛ 428: IOT PROGRAMMING

Dr. Panayiotis Kolios
Assistant Professor, Dept. Computer Science,
KIOS CoE for Intelligent Systems and Networks
Office: FST 01, 116
Telephone: +357 22893450 / 22892695
Web: https://www.kios.ucy.ac.cy/pkolios/

Πανεπιστήμιο Κύπρου

- Intro to classification and clustering
- Similarity and dissimilarity measures and their impact in classification and clustering
- Similarity measures for time series.
- Classification and clustering of time series.

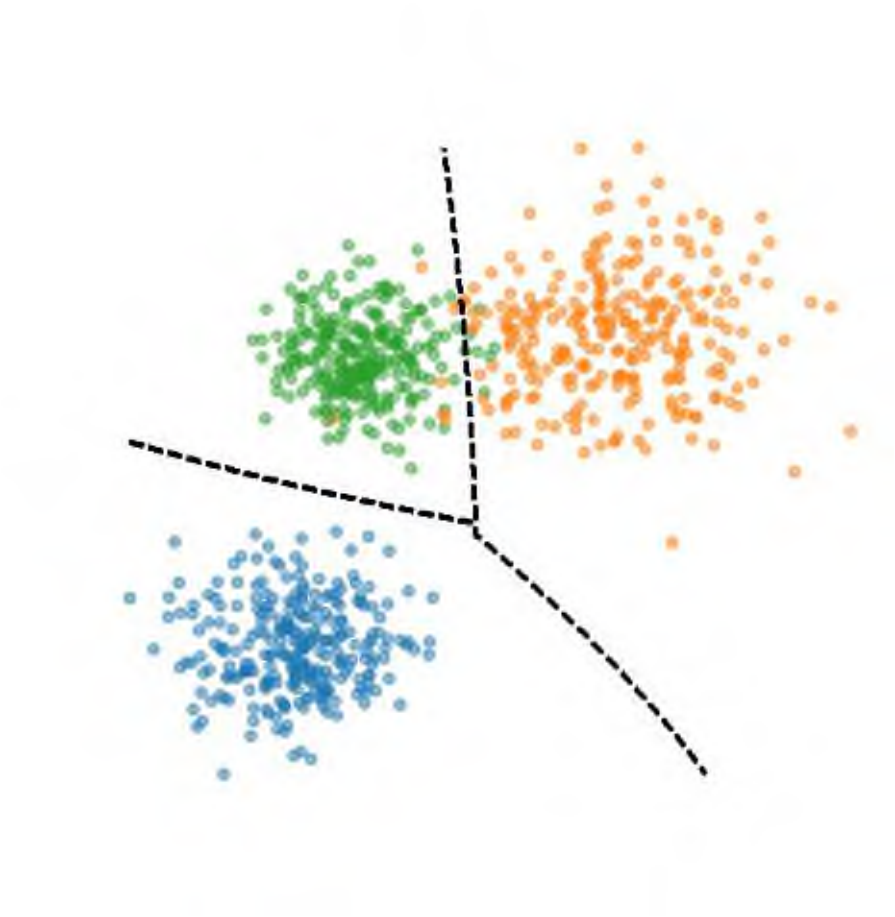
```
# Imports
import warnings
warnings.filterwarnings("ignore")
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
                                    ⋮
from sklearn.mixture import GaussianMixture
from sklearn.neighbors import KNeighborsClassifier
from scipy.cluster.hierarchy import linkage, dendrogram, fcluster
import scipy.spatial.distance as ssd
import statsmodels.api as sm
from dtaidistance import dtw, dtw_ndim
from dtaidistance import dtw_visualisation as dtwvis
from tck.TCK import TCK
from tck.datasets import DataLoader
from reservoir_computing.reservoir import Reservoir
from reservoir_computing.tensorPCA import tensorPCA
from reservoir_computing.modules import RC_model
from reservoir_computing.utils import compute_test_scores
```

- Classification is a **supervised** task: a classifier uses external supervision to learn a task
- Use classes of information (labels)
- A classifier fits its parameters to predict the correct label
- This can be seen as learning where to put a decision boundary that separates the classes

```
# Generate some toy data
data = datasets.make_blobs(n_samples=800, centers=[[-5,-7], [5,5], [-3,4]],
cluster_std=[1.7, 2.5, 1.5], random_state=8)
X, y = data
X = StandardScaler().fit_transform(X)
```

- Most classifiers trade an accurate fit of the training data with generalization capabilities on out-of-sample data
- The behavior is controlled by hyperparameters that are set through a validation procedure (and usually some experience)
- Here we use SVC, which is a Support Vector Machine (SVM) for classification

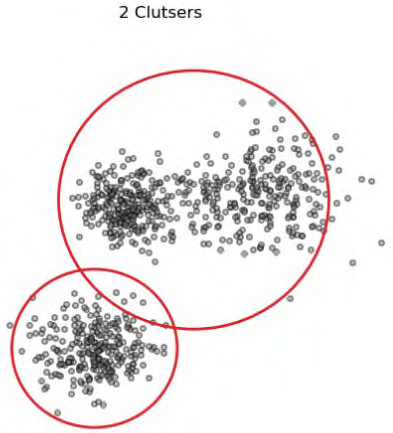
```
plot_class_example(X, y, gamma=0.1)
```



- Change gamma to investigate performance

- Clustering is an unsupervised task
- It only looks at the structure of the data without using additional information (class labels, extra data, human knowledge, etc…)
- A clustering algorithm groups data together so that each group is compact and separated from the others
- Clustering gives insights about the structure of the data
- The number of clusters is often not given

```
# We use the same data (X) as before, but not the labels (y)
plot_cluster_example(X, K=2)
```



- There are many metrics to evaluate the performance in classification and clustering tasks
- Depend on the problem at hand
- Often use more than metric for evaluation
- Consider:
  - accuracy and F1 score for classification
  - NMI for clustering

- Classification accuracy is the simplest way to measure of how well a classification model performs
- Ratio of correctly predicted observations to the total observations:

$$Accuracy = \frac{Correct\ Predictions}{Total\ Number\ of\ Predictions} = \frac{TP + TN}{TP + TN + FN + FP}$$

- TP: true positive, TN: true negative, FP: false positive, FN: false negative

```
# Split the data in training and test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=0)

# Fit the classifier
clf = svm.SVC(kernel="linear")
clf.fit(X_train, y_train)

# Compute predictions and accuracy
y_pred = clf.predict(X_test)
print(f"Accuracy: {accuracy_score(y_test, y_pred):.2f}")
```

Accuracy: 0.99

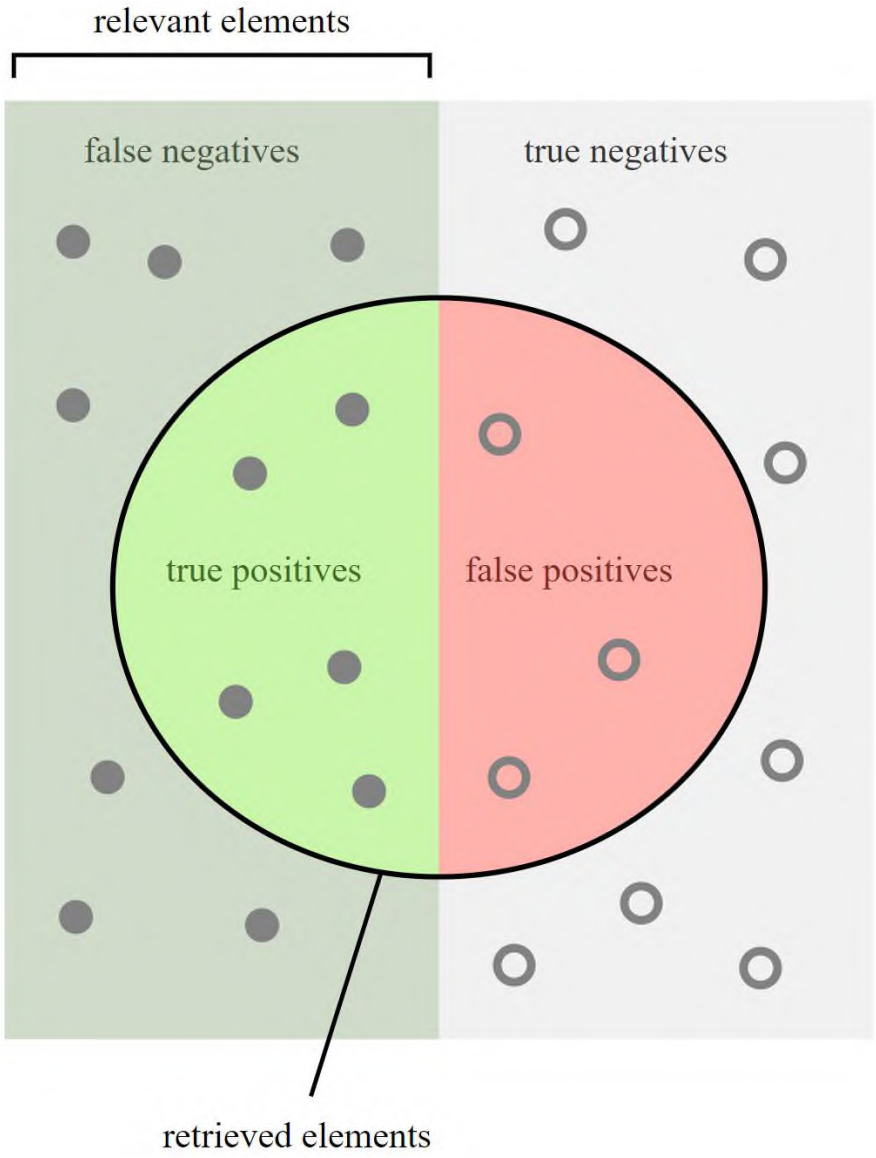
- F1 Score is the *harmonic mean* of precision and recall, providing a balance between them
- Used when class distribution is uneven and you need a measure that takes both FP and FN into account

$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

- where

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$



How many retrieved items are relevant?

Precision =

How many relevant items are retrieved?

Recall =

```
# Create an imbalanced dataset
n_samples_1 = 2000 # Samples of class 0
n_samples_2 = 100  # Samples of class 1

X_imb, y_imb = datasets.make_blobs(
    n_samples=[n_samples_1, n_samples_2],
    centers=[[0.0, 0.0], [2.0, 2.0]],
    cluster_std=[1.5, 0.5],
    random_state=0, shuffle=False)
```

```
# Split the data in training and test set
X_tr_imb, X_te_imb, y_tr_imb, y_te_imb = train_test_split(X_imb, y_imb,
test_size=0.2, random_state=0)

# Fit the classifier
clf = svm.SVC(kernel="linear", class_weight={1: 20}) # Try setting
class_weight={1: 20}
clf.fit(X_tr_imb, y_tr_imb)

# Compute predictions and accuracy
y_pred_imb = clf.predict(X_te_imb)
print(f"Accuracy: {accuracy_score(y_te_imb, y_pred_imb):.2f}")
```

Accuracy: 0.89

Predictions of class 0: 355
Predictions of class 1: 65
F1 score: 0.45

- High accuracy score… but if we look closely, the classifier simply assigned all labels to the majority class
- This is not acceptable in cases where the minority class is of interest
    - For example in anomaly detection

Πανεπιστήμιο Κύπρου

- NMI is a normalization of the Mutual Information (MI) score to scale the results between 0 (no mutual information) and 1 (perfect correlation)
- It measures the agreement between the cluster assignments C, and the class labels Y

$$NMI(C,Y) = \frac{2 \times I(C;Y)}{H(C) + H(Y)}$$

- where $I(C;Y)$ is the MI between clusters and labels, and $H(C)$ and $H(Y)$ are the entropies of X and Y

```
# NMI for k-means with different values of k
clust_lab = KMeans(n_clusters=2, random_state=0, n_init="auto").fit(X).labels_
print(f"K=2, NMI: {v_measure_score(clust_lab, y):.2f}")

clust_lab = KMeans(n_clusters=3, random_state=0, n_init="auto").fit(X).labels_
print(f"K=3, NMI: {v_measure_score(clust_lab, y):.2f}")

clust_lab = KMeans(n_clusters=4, random_state=0, n_init="auto").fit(X).labels_
print(f"K=4, NMI: {v_measure_score(clust_lab, y):.2f}")
```

K=2, NMI: 0.73, K=3, NMI: 0.92, K=4, NMI: 0.84

Dissimilarity measures

- Quantify how different two objects are: the higher their dissimilarity, the more different they are.
- Dissimilarity measures are crucial to distinguish between distinct groups of data or identify outliers.
- The most common linear dissimilarity measure is the Euclidean distance:

$$d(x, y) = \|x - y\|_2$$

- Mahalanobis distance:

$$d(x, y) = \sqrt{(x - y)^T \Sigma^{-1} (x - y)}$$

- which reduces to the Euclidean distance when covariance matrix $\Sigma^{-1} = I$

Similarity measures

- A similarity measure quantifies how similar two objects are: the higher the value, the more similar the objects
- These measures are essential in algorithms that rely on the concept of closeness or similarity to make decisions, such as recommender systems
- An example of similarity measures is the cosine similarity:

$$s(x, y) = \frac{x^1 y}{\|x\|\|y\|}$$

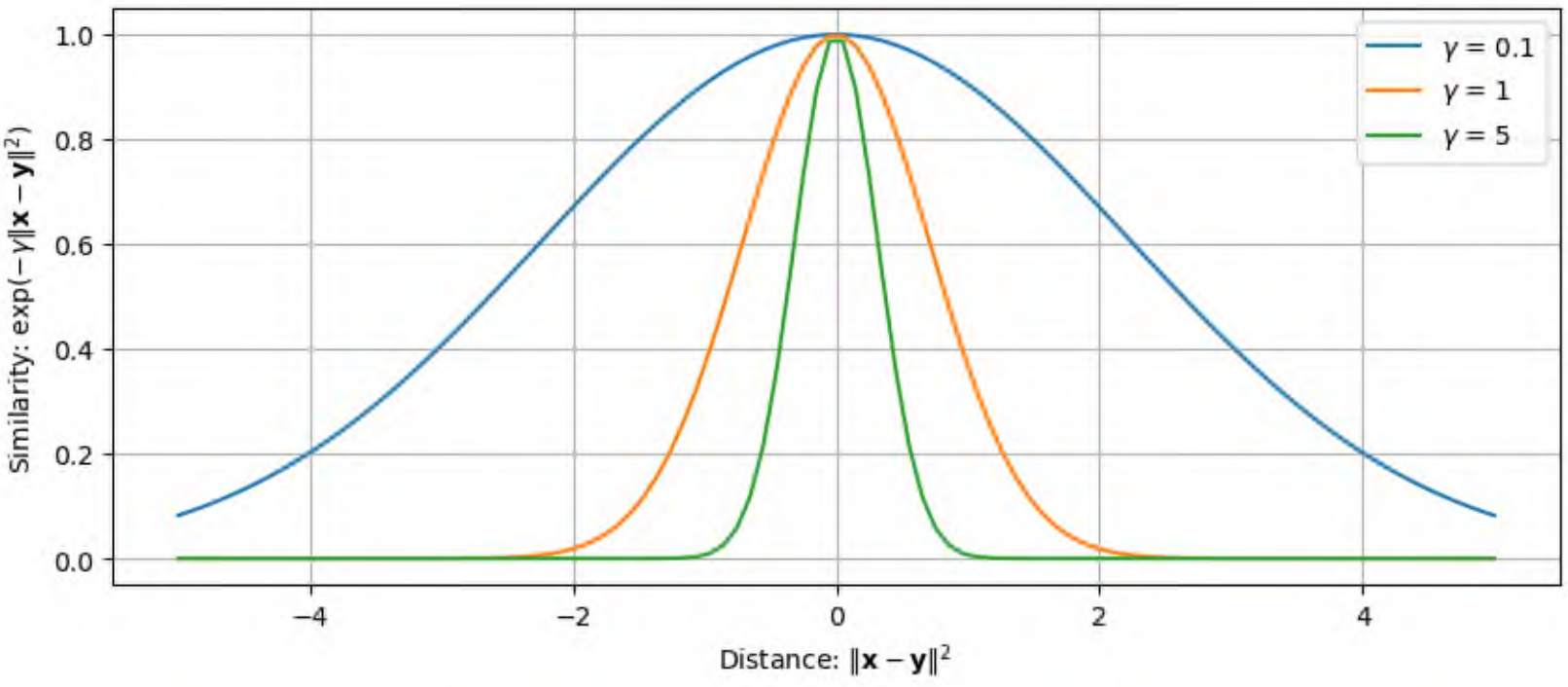
- Another example is the Pearson correlation coefficient, used in statistics to measure the linear correlation between two variables:

$$s(x, y) = \frac{cov(x, y)}{\sigma_x \sigma_y}$$

- The measures we described so far are linear
- It means that their computation follows a linear relationship with respect to the data.
- Some measures, instead, are non-linear, meaning the relationship between the measure and the data does not follow a straight line.
- Kernels are examples of non-linear similarity measures.
- The most famous kernel is the Radial Basis Function (RBF):

$$s(x, y) = \exp(-\gamma \|x - y\|^2$$

- The parameter γ is the kernel width, which controls the std. dev. of the Gaussian
- Setting it properly is crucial for defining distances: smaller values will account for relationships between distant objects

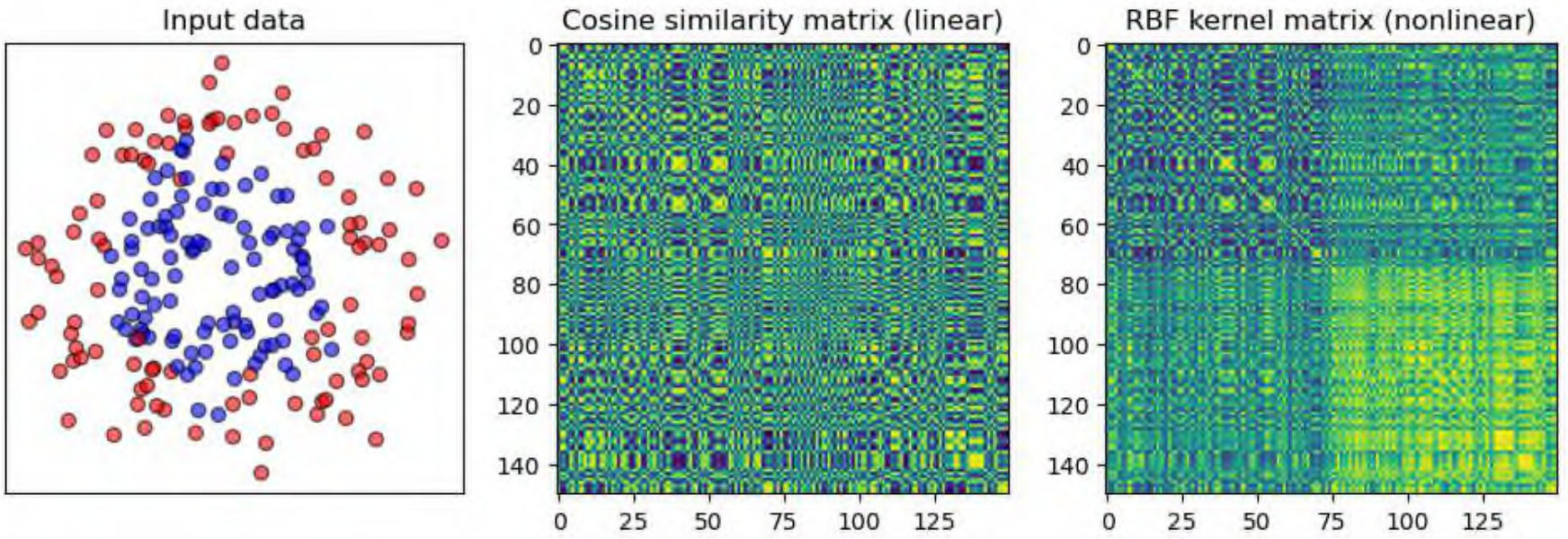


- The choice of similarity or dissimilarity measure is critical in classification and clustering problems
- The measure directly affects how well an algorithm can identify the structure of the data
- An inappropriate choice might lead to poor classification or clustering performance because the measure may not capture the actual relationships among data points

Πανεπιστήμιο Κύπρου

- Explore measures for the following example
- Compute a similarity matrix using a linear and a nonlinear measure
    - Linear measure: cosine similarity
    - Non-linear measure: RBF kernel

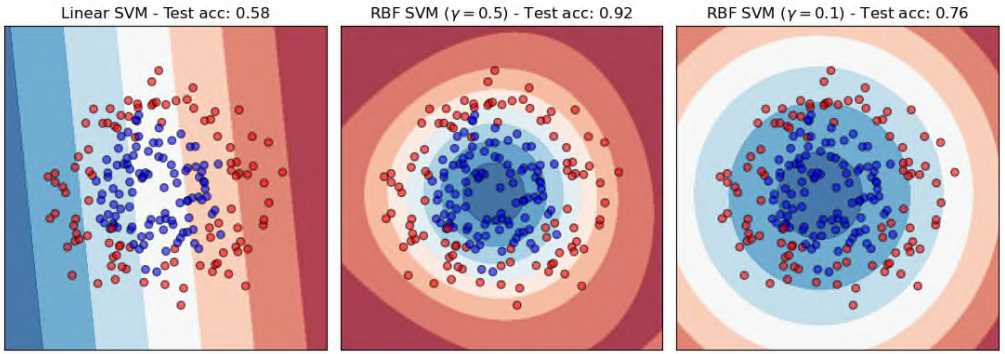
```
X, y = datasets.make_circles(noise=0.2, factor=0.5, random_state=1,
n_samples=200) # Create toy data
X_train , X_test , y_train, y_test = train_test_split(X, y, random_state=0)
#Train-test split

# Cosine similarity matrix
cosine_train = cosine_similarity(X_train)

# RBF similarity matrix
rbf_kernel_train = pairwise_kernels(X_train, metric='rbf', gamma=0.5)
```



- The two classes appear more separated when using a nonlinear similarity

- Next, train an SVM classifier that uses the two similarity measures
  - Look at the decision boundaries learned by the classifier and compute the performance on the test data
- For the RBF kernel, we will use both γ=0.5 and γ=0.1

```
classifiers = [svm.SVC(kernel="linear"), svm.SVC(gamma=0.5), svm.SVC(gamma=0.1)]
names = ["Linear SVM", "RBF SVM ($\gamma=0.5$)", "RBF SVM ($\gamma=0.1$)"]
figure = plt.figure(figsize=(11, 4))
for i, (name, clf) in enumerate(zip(names, classifiers)):
    ax = plt.subplot(1,3,i+1)
    clf.fit(X_train, y_train)
    score = clf.score(X_test, y_test)
    DecisionBoundaryDisplay.from_estimator(
        clf, X, cmap=plt.cm.RdBu, alpha=0.8, ax=ax, eps=0.5)
    ax.scatter(X[:, 0], X[:, 1], c=y, cmap=cm_bright, edgecolors="k", alpha=0.6)
    ax.set_xticks(()) ax.set_yticks(())
    ax.set_title(name + f" - Test acc: {score:.2f}")
plt.tight_layout() plt.show()
```



Πανεπιστήμιο Κύπρου

- (dis)similarity measures are more profound in clustering (unsupervised task)
  - The lack of training cannot compensate for the effect of choosing a bad (dis)similarity
- Samples get grouped very differently based on what makes them (dis)similar

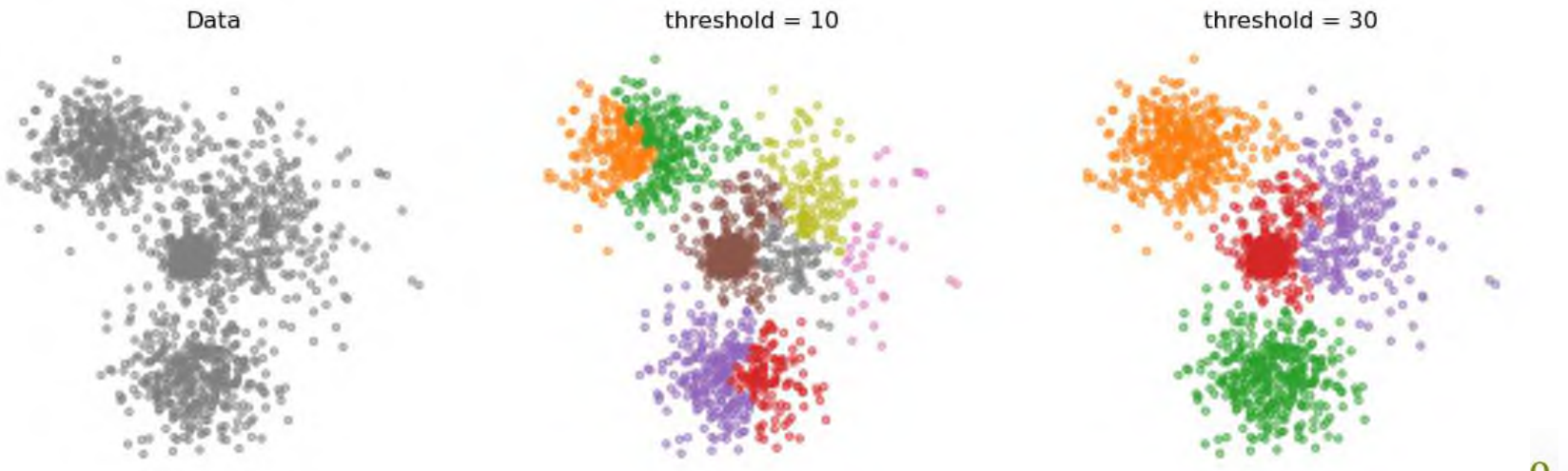
- In the next example, we look at hierarchical clustering (HC).
- HC progressively form clusters by grouping together samples within a certain distance radius
  - In the beginning, the radius is very small and many distinct clusters are formed
  - As the radius increases, further points are grouped together and the number of clusters decreases

- Consider the following data and compute a squared Euclidean distance matrix
- As HC algorithm we will use the Ward Linkage, which gradually aggregate clusters by optimizing the minimum variance criterion
- At each step it finds the pair of clusters that leads to minimum increase in total within-cluster variance after merging

```
X, y = datasets.make_blobs(n_samples=1500, centers=4,
                           cluster_std=[1.7, 2.5, 0.5, 1.5], random_state=2)
X = StandardScaler().fit_transform(X) # Normalizing the data facilitates setting
the radius

# Compute the distance matrix
Dist = pairwise_distances(X, metric="sqeuclidean")
distArray = ssd.squareform(Dist)

# Compute the hierarchy of clusters
Z = linkage(distArray, 'ward')
```

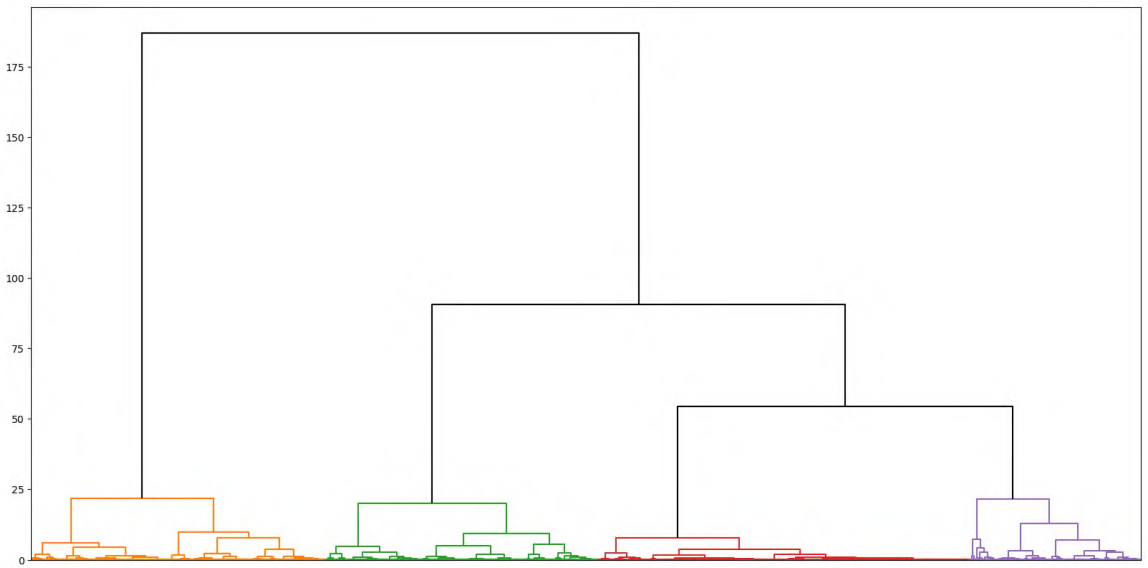
- By setting a different radius (threshold) we obtain different partitions
- For example, set thresholds t=10 and t=30

```
partition_1 = fcluster(Z, t=10, criterion="distance")
print("Partition 1: %d clusters"%len(np.unique(partition_1)))

partition_2 = fcluster(Z, t=30, criterion="distance")
print("Partition 1: %d clusters"%len(np.unique(partition_2)))

_, axes = plt.subplots(1,3, figsize=(14,4))
plot_clusters(X, title="Data", ax=axes[0])
plot_clusters(X, title="threshold = 10", clusters=partition_1, ax=axes[1])
plot_clusters(X, title="threshold = 30", clusters=partition_2, ax=axes[2])
```



- Possible to visualize the hierarchy generated by HC or to color the branches based on the threshold value
- Useful tool to examine the structure in the data at different resolutions
- Partitions that persist for broad ranges of values of the threshold are those that characterize the dataset the most

```
fig = plt.figure(figsize=(20, 10))
dn = dendrogram(Z, color_threshold=30, above_threshold_color='k',
                show_leaf_counts=False)
plt.xticks([])
plt.show()
```

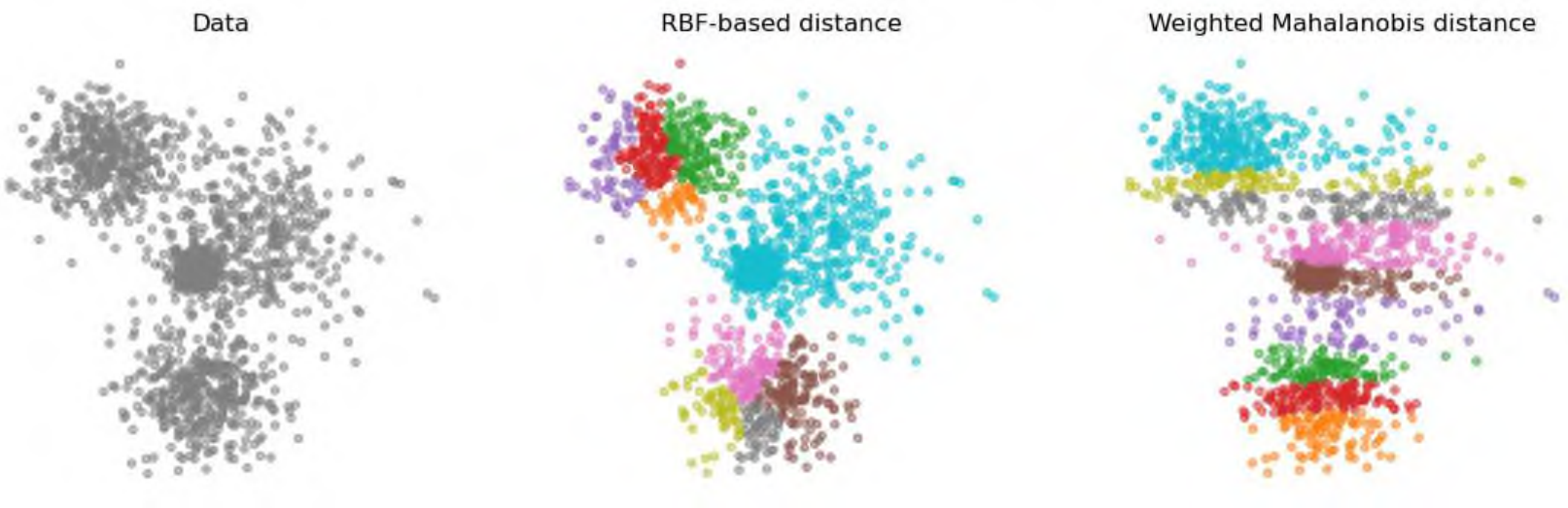


- Examine different distance metrics
  - e.g. RBF kernel and a Mahalanobis distance that weights each feature differently

```
# Compute the RBF (note that we must convert it to a distance)
rbf_kernel = pairwise_kernels(X, metric='rbf', gamma=1.0) # compute the rbf
similarity
rbf_kernel = rbf_kernel + rbf_kernel.T # make symmetric
rbf_kernel /= rbf_kernel.max() # normalize to [0, 1]
rbf_dist = 1.0 - rbf_kernel # convert to distance
np.fill_diagonal(rbf_dist, 0) # due to numerical errors, the diagonal might not
be 0

# Compute the partition
distArray = ssd.squareform(rbf_dist)
Z = linkage(distArray, 'ward')
partition_3 = fcluster(Z, t=3, criterion="distance")
```

```
# Mahalanobis distance that assigns different weights to the features
weights = np.array([1, 0.1])
Dist = pairwise_distances(X, metric="mahalanobis", VI=np.diag(1/weights**2))

# Compute the partition
distArray = ssd.squareform(Dist)
Z = linkage(distArray, 'ward')
partition_4 = fcluster(Z, t=30, criterion="distance")
```

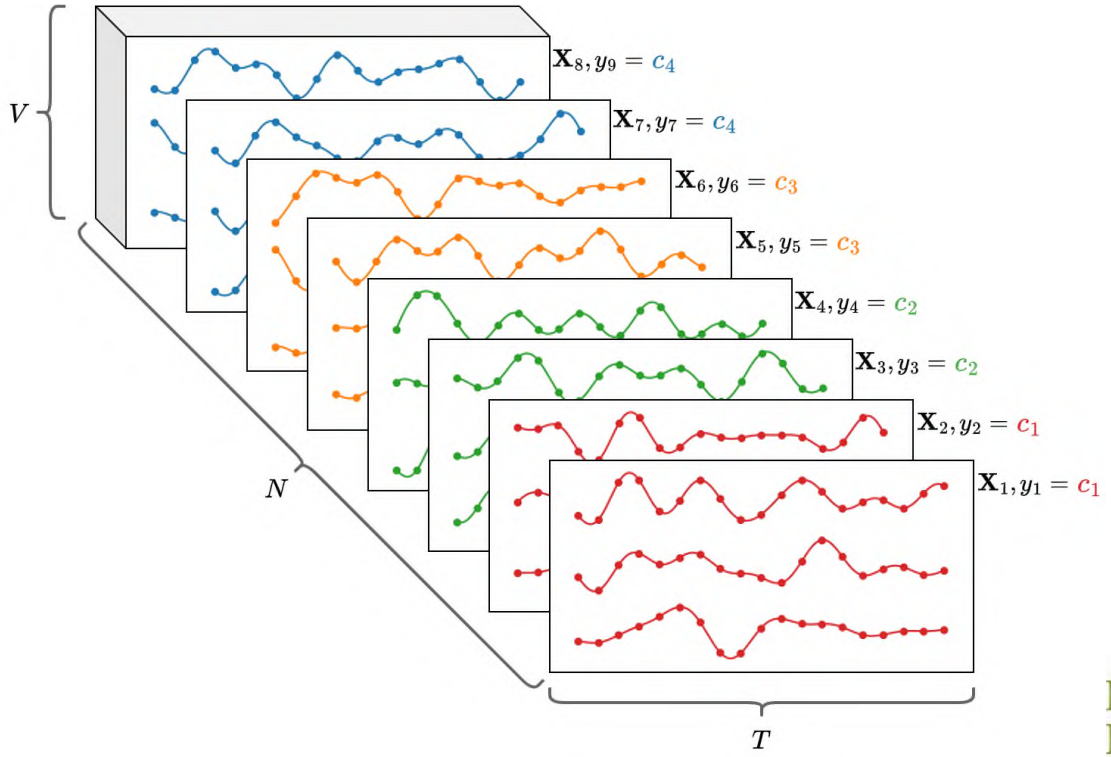
```
_, axes = plt.subplots(1,3, figsize=(14,4))
plot_clusters(X, title="Data", ax=axes[0])
plot_clusters(X, title="RBF-based distance", clusters=partition_3, ax=axes[1])
plot_clusters(X, title="Weighted Mahalanobis distance", clusters=partition_4,
ax=axes[2])
```



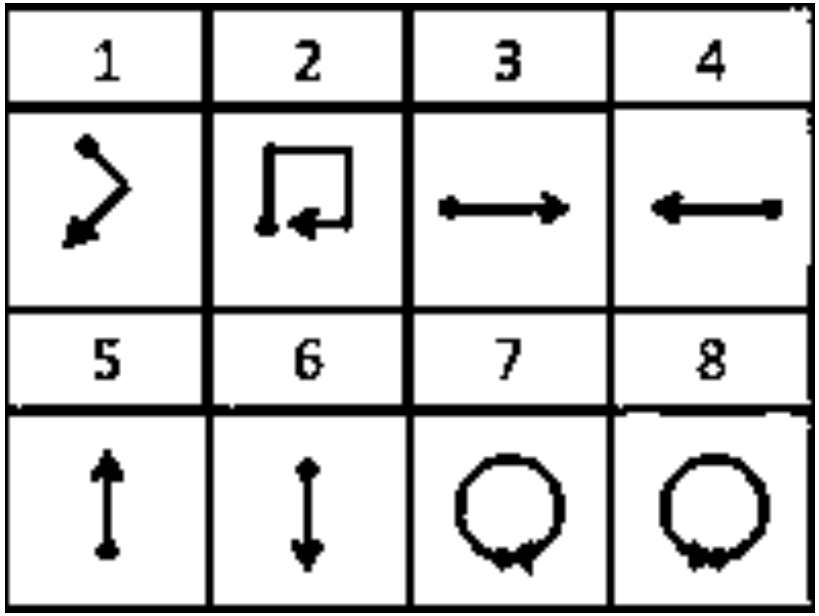
- Distance measure greatly impacts both classification and clustering

- Investigate 3 families of approaches to compute a distance between time series:
  - Alignment-based metrics
  - Time series kernels
  - Vector distance on time series embeddings
- Each approach comes with pros and cons
- We will look at one approach from each family

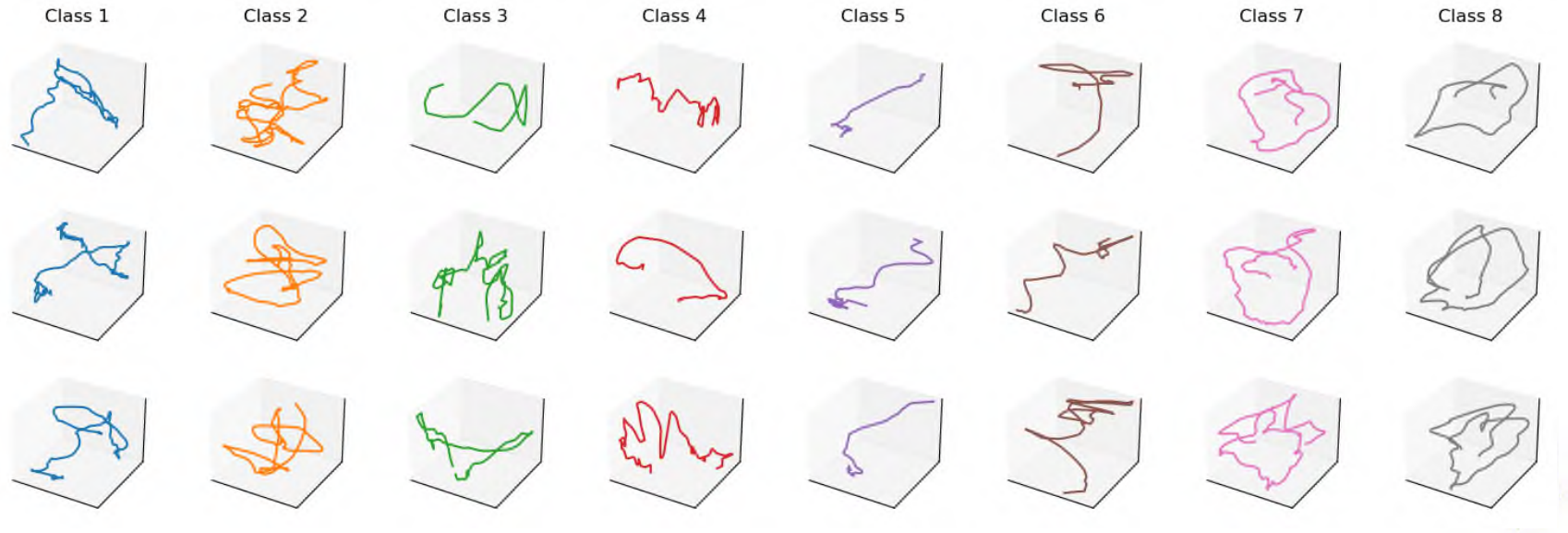
- An MTS is represented by a matrix $X \in \mathbb{R}^{T \times V}$, where T is the number of time steps and V is the number of variables
- The whole dataset can be represented by a 3-dimensional array x of size $[N, T, V]$
- In a classification setting, the i-th MTS X[i,:,:] is associated with with a class label y[i]



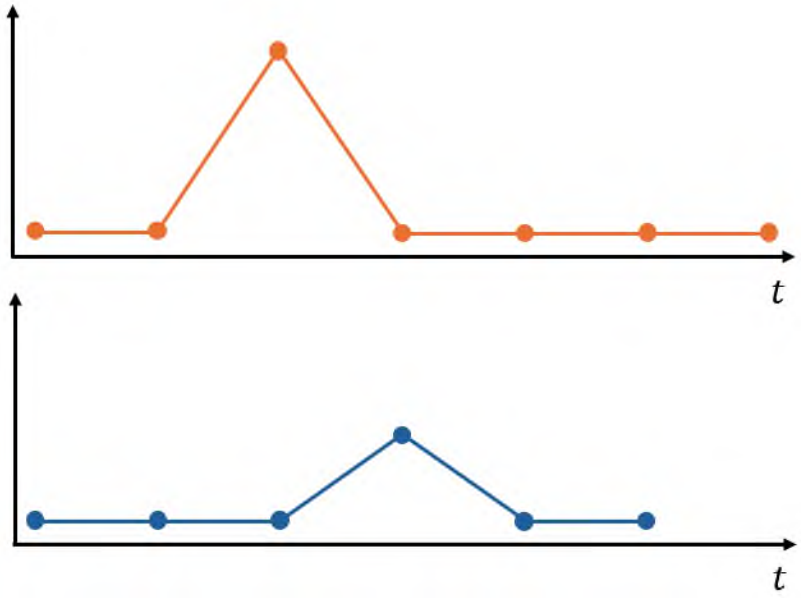
- Accelerometer-based personalized gesture recognition (uWave)
- Each MTS represents the measurements of an accelerometer wore when doing one of the following gestures
    - The dot is the starting point, the arrow the end point



```
def plot_uwave():
    X, Y, _, _ = DataLoader().get_data('UWAVE')
    _, axes = plt.subplots(3, 8, figsize=(15, 5), subplot_kw={'projection':
'3d'})
    for i in range(len(np.unique(Y))):
        idx = np.where(Y == i+1)[0][:3]
        for j, id in enumerate(idx):
            axes[j,i].plot(X[id, :, 0], X[id, :, 1], X[id, :, 2],
color=plt.cm.tab10(i))
            axes[j,i].set_xticks(())
            axes[j,i].set_yticks(())
            axes[j,i].set_zticks(())
            axes[j,i].spines[['right', 'left', 'top',
'bottom']].set_visible(False)
            if j == 0:
                axes[j,i].set_title(f"Class {i+1}")
    plt.tight_layout()
    plt.show()
```



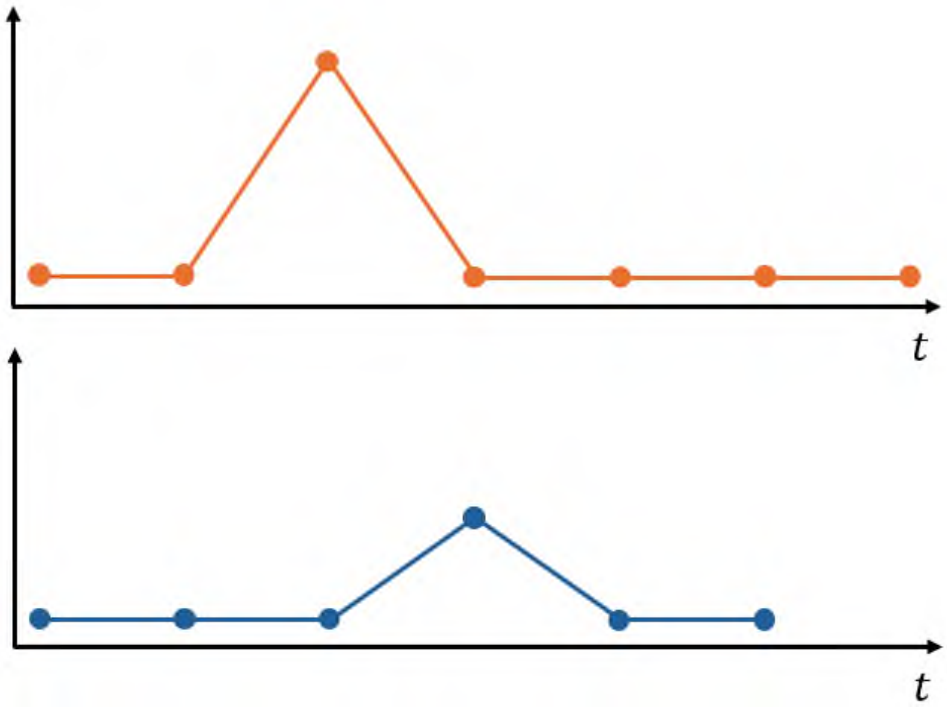
- An alignment-based metric relies on a temporal alignment of two time series to assess their similarity
- One of the most prominent representatives of this class is Dynamic Time Warping (DTW)
- The idea of DTW is to first align two time series and then compute a Euclidean distance between the matched elements
    - Naive approach: Compute distance of raw time series



$$d(x,y) = \sum_{t=1}^{\min(T_x,T_y)} \|x(t) - y(t)\|_2$$

If the two time series are very similar but slightly disaligned, it will produce a large distance.

- DTW disregards the exact timestamps at which the observations occur
- DTW seeks for the temporal alignment (a matching between time indexes of the two time series) that minimizes Euclidean distance between the aligned series



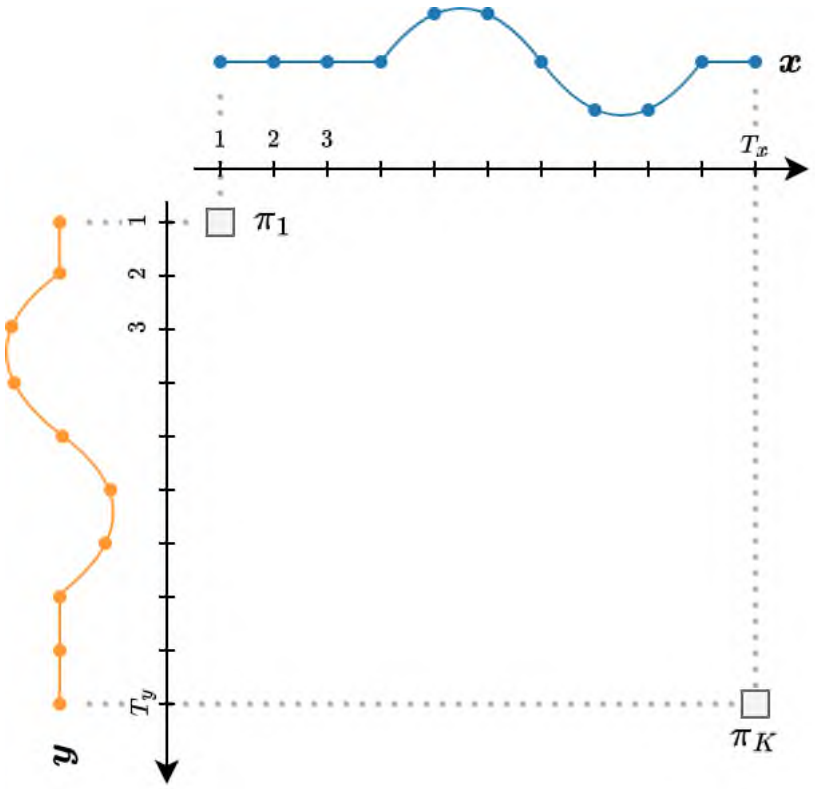
Πανεπιστήμιο Κύπρου

- DTW solves the following optimization problem:

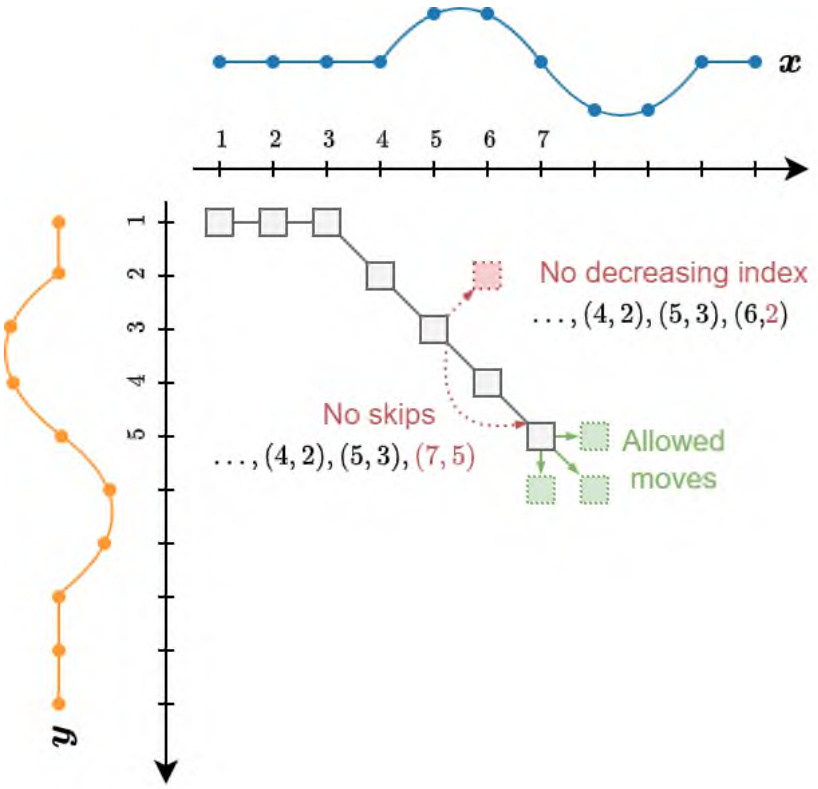
$$DTW(x,y) = \min_{\pi \in \mathrm{P}(x,y)} \left(\sum_{(i,j)\in\pi} d\big(x(i), y(j)\big) \right)$$

- π is an alignment path of length K, i.e., a sequence of index pairs
- $\mathrm{P}(x,y)$ is the set of all admissible paths
  - An admissible path should satisfy the following conditions:
  1. The beginning and the end of x, y are matched together $\pi_1 = (1,1), \pi_K = (T_x, T_y)$
  2. The sequence is monotonically increasing in both i and j and all time series indexes should appear at least once $i_{k-1} \leq i_k \leq i_{k-1} + 1, j_{k-1} \leq j_k \leq j_{k-1} + 1$

The two time series are matched together



Sequence is monotonically increasing



Πανεπιστήμιο Κύπρου

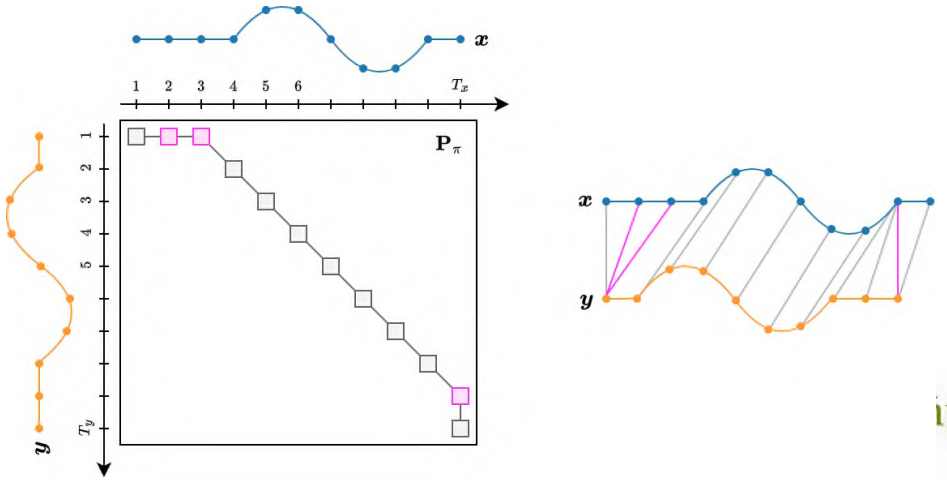
- The DTW path can be represented by a binary matrix P_π whose non-zero entries are those corresponding to a matching between time series elements:

$$P_\pi = \begin{cases} 1 = if\ (i,j) \in \pi \\ 0 = otherwise \end{cases}$$

- Using the matrix notation, DTW can be rewritten as:

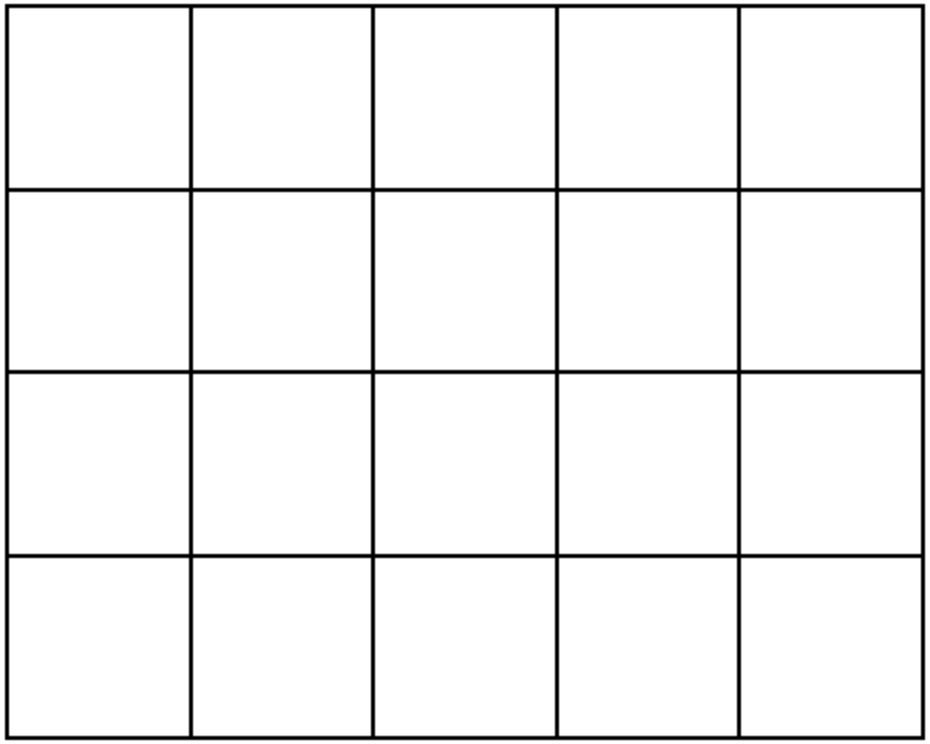
$$DTW(x,y) = \min_{\pi \in P(x,y)} \langle P_\pi, D_{x,y} \rangle$$

- where the $(i,j)^{th}$ element of $D_{x,y} \in \mathbb{R}^{T_x,T_y}$ stores the distance $d\big(x(i),y(j)\big)$



- Compute cost matrix $D_{x,y}$

$$d\big(x(0), y(0)\big) = 0$$



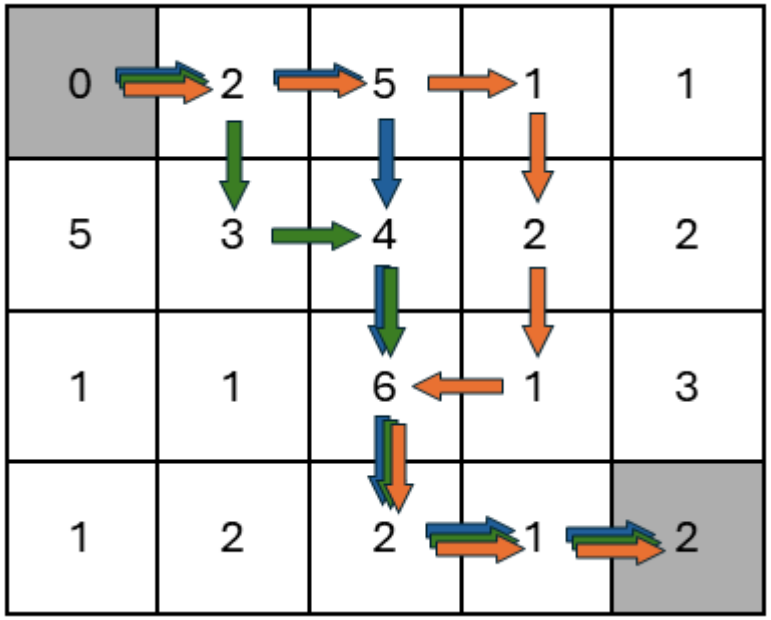
- Then, traverse the matrix from the top-left corner $(1,1)$ to the bottom-right one (T_x, T_y)
- Excluding the borders, at each step we have three options to decide where to move with a different cost
- The optimal path is the one with minimum cost

0	2	5	1	1
5	3	4	2	2
1	1	6	1	3
1	2	2	1	2

Cost = 0 + + + + + + 2

Πανεπιστήμιο Κύπρου

- Need to compute the cost for each possible path
- For T_x, T_y the total number of paths is $O\left(\frac{\left(3+2\sqrt{2}\right)^{Tx}}{\sqrt{T_x}}\right)$
- That is a big number: calculating all of them is intractable
- There are many paths that share the same sections
- To make the problem tractable, we must avoid recomputing the same paths over and over



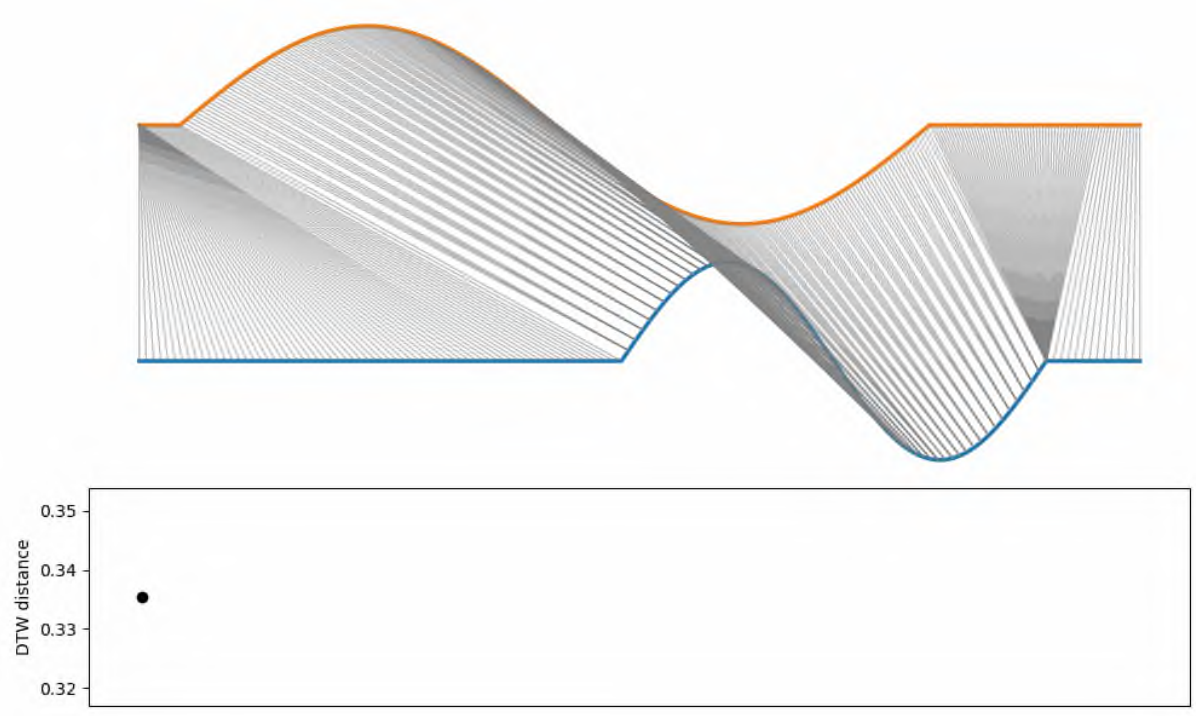
- Use recursion, a dynamic programming technique that breaks down a complex problem into simpler subproblems
- Each subproblem is solved just once and the solution is stored in memory
- When a subproblem is encountered again, its solution is retrieved instead of being recomputed
- This significantly reduces the number of computations cutting the redundancies
- The recursive algorithm has complexity $O(T_x T_y)$ and is formulated as follow:

```
def DTWDistance(x, y):
  for i in range(len(x)):
    for j in range(len(y)):
      DTW[i, j] = d(x[i], y[j])
      if i > 0 or j > 0:
        DTW[i, j] += min(
          DTW[i-1, j  ] if i > 0             else inf,
          DTW[i  , j-1] if j > 0             else inf,
          DTW[i-1, j-1] if (i > 0 and j > 0) else inf
        )
  return DTW[-1, -1]
```

- Basic idea is that each block (i, j) recursively ask its predecessors $(i-1, j)$, $(i, j-1)$ and $(i-1, j-1)$ the cost to reach them
- The request is propagated back to the origin which returns the first answer
- The answer is then propagated forward to all the requesters, which update the answer with their own cost

0	2	5	1	1
5	3	4	2	2
1	1	6	1	3
1	2	2	1	2

- Example shows how the DTW changes when two curves are translated and stretched/squeezed at the same time

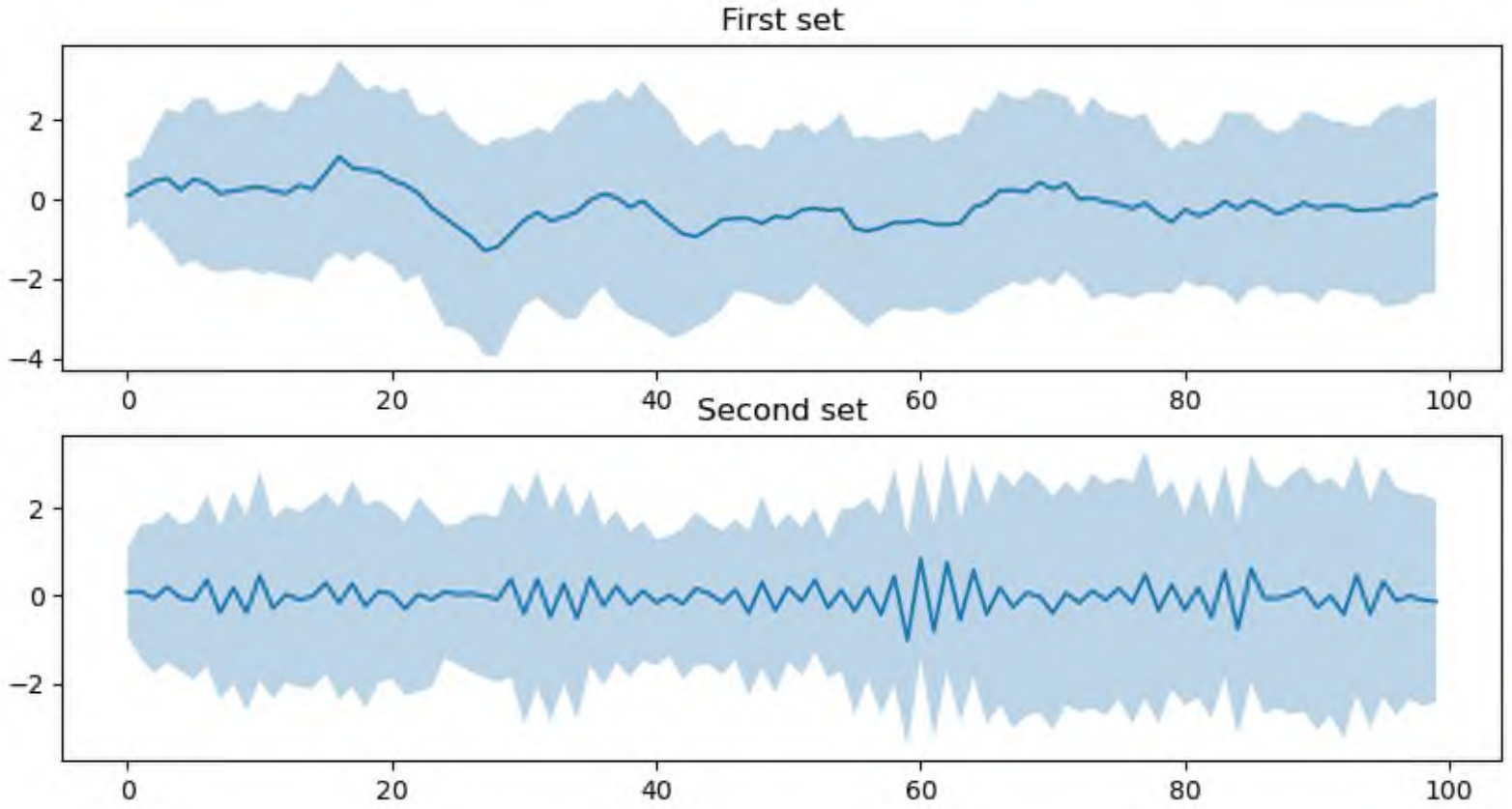


- DTW is invariant to translation

- Example by generating two groups of time series using two different AR(1) processes

```
T = 100 # Length of the time series
N = 30  # Time series per set

# Generate the first set of time series
Y1 = np.zeros((T, N))
for i in range(N):
    Y1[:,i] = sm.tsa.arma_generate_sample(ar=[1, -.9], ma=[1], nsample=T,
scale=1)

# Generate the second set of time series
Y2 = np.zeros((T, N))
for i in range(N):
    Y2[:,i] = sm.tsa.arma_generate_sample(ar=[1, .9], ma=[1], nsample=T, scale=1)
```

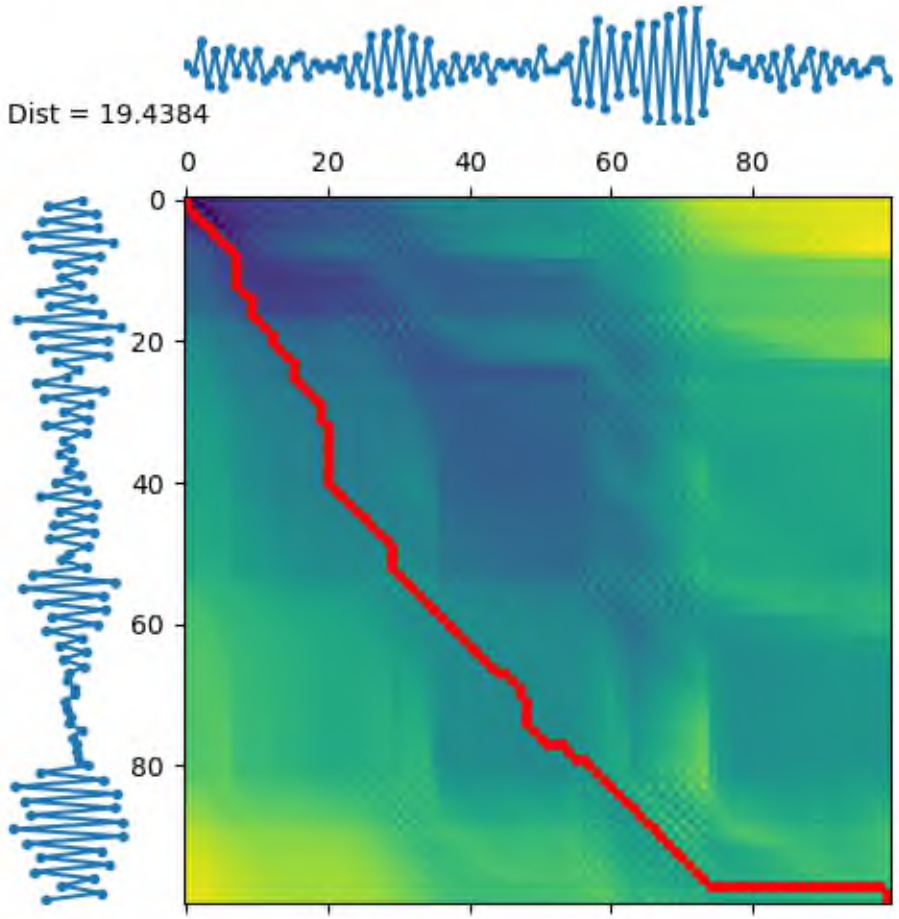
```
fig, axes = plt.subplots(2,1, figsize=(10, 5))
axes[0].plot(np.mean(Y1, axis=1))
axes[0].fill_between(range(T), np.mean(Y1, axis=1) - np.std(Y1, axis=1),
np.mean(Y1, axis=1) + np.std(Y1, axis=1), alpha=0.3)
axes[0].set_title("First set")
axes[1].plot(np.mean(Y2, axis=1))
axes[1].fill_between(range(T), np.mean(Y2, axis=1) - np.std(Y2, axis=1),
np.mean(Y2, axis=1) + np.std(Y2, axis=1), alpha=0.3)
axes[1].set_title("Second set")
plt.show()
```



- Visualize the path π on the cost matrix $D_{x,y}$ for the time series we generated
- First, we let x, y be two time series from the same group
- Notice how the path π crosses the darker areas, corresponding to smaller dissimilarity values

Πανεπιστήμιο Κύπρου

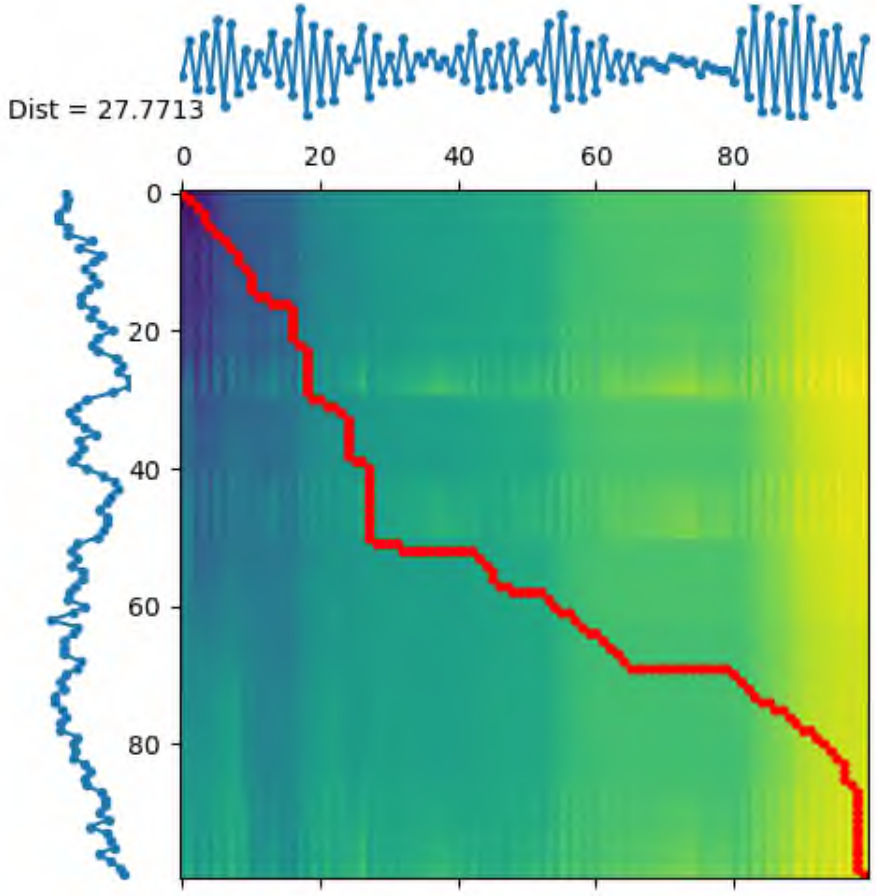
```
s1 = Y2[:,1]
s2 = Y2[:,2]
fig = plt.figure(figsize=(5, 5))
d, paths = dtw.warping_paths(s1, s2)
best_path = dtw.best_path(paths)
dtwvis.plot_warpingpaths(s1, s2, paths, best_path, figure=fig);
```



Πανεπιστήμιο Κύπρου

- Then, we select x and y from two different groups
- We see how the dissimilarity is much higher in this case and the path changes significantly

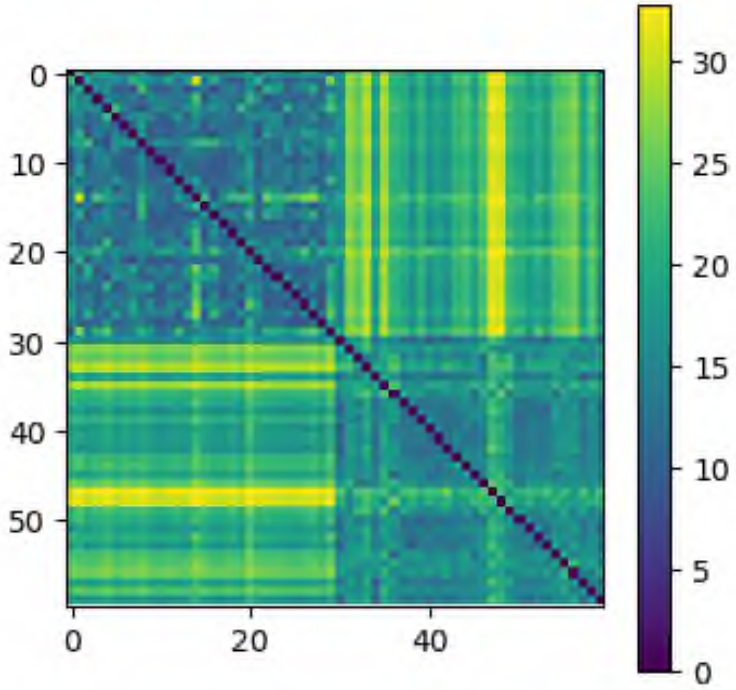
```
s1 = Y1[:,1]
s2 = Y2[:,1]
fig = plt.figure(figsize=(5, 5))
d, paths = dtw.warping_paths(s1, s2)
best_path = dtw.best_path(paths)
dtwvis.plot_warpingpaths(s1, s2, paths, best_path, figure=fig);
```



Πανεπιστήμιο Κύπρου

- Compute the DTW distance between all the time series in the two sets

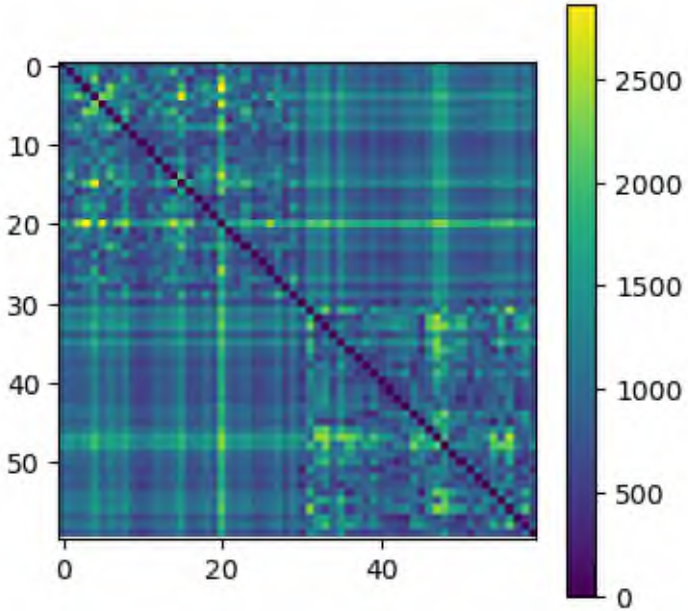
```
# Concatenate the two sets of time series
Y = np.concatenate((Y1, Y2), axis=1).T
# Compute the distance matrix
dtw_dist = dtw.distance_matrix_fast(Y)
# Plot the distance matrix
plt.figure(figsize=(4,4))
plt.imshow(dtw_dist, cmap='viridis')
plt.colorbar()
plt.show()
```



Πανεπιστήμιο Κύπρου

- For comparison, compute the Euclidean distance between the time series

```
# compute euclidean distance between the time series
euc_dist = pairwise_distances(Y, metric="sqeuclidean")
plt.figure(figsize=(4,4))
plt.imshow(euc_dist, cmap='viridis')
plt.colorbar()
plt.show()
```



- This time the dissimilarity matrix is less structured and is harder to see the division in two groups
- As expected, the Euclidean distance is less suitable for these type of data

Πανεπιστήμιο Κύπρου

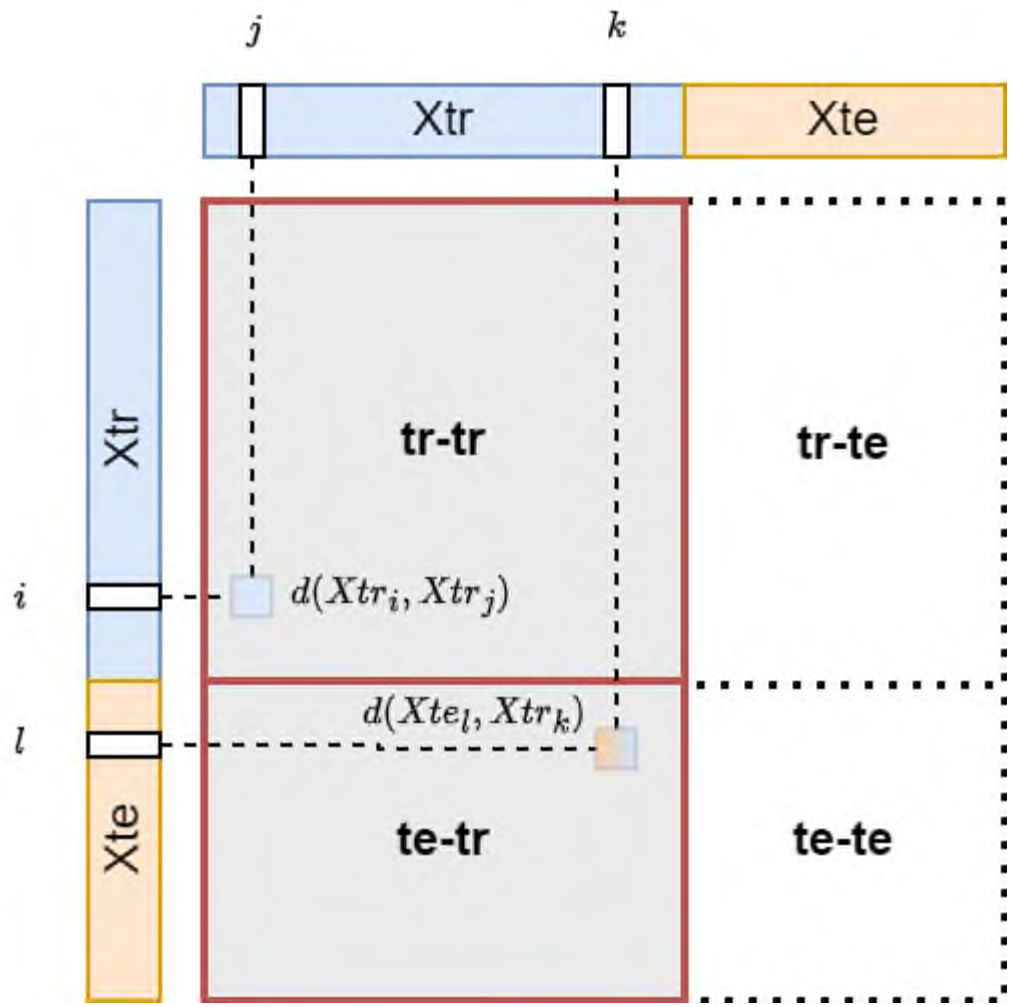
- Will use Japanese Vowels dataset in the next example

```
DataLoader().available_datasets()
Xtr, Ytr, Xte, Yte = DataLoader().get_data('Japanese_Vowels’)
# Concatenate X and Xte
X = np.concatenate((Xtr, Xte), axis=0)
Y = np.concatenate((Ytr, Yte), axis=0)
# Compute the dissimilarity matrix
dtw_dist = dtw_ndim.distance_matrix_fast(X)
print("dist shape:", dtw_dist.shape)
```

Loaded Japanese_Vowels dataset.
Number of classes: 9
Data shapes:
Xtr: (270, 29, 12)
Ytr: (270, 1)
Xte: (370, 29, 12)
Yte: (370, 1)

dist shape: (640, 640)

- Note that we have concatenated the training and test set before computing the distances
- In the following, we will use an SVC with a pre-computed kernel, i.e., a custom similarity matrix
- There are two sets of distances we need to compute to provide the information to the classifier
- For training, we need the distances between elements in the training set
    - Let's call the matrix containing these distances tr-tr
- For testing, we need the distances between the training and test set elements
    - These distances are in the matrix te-tr

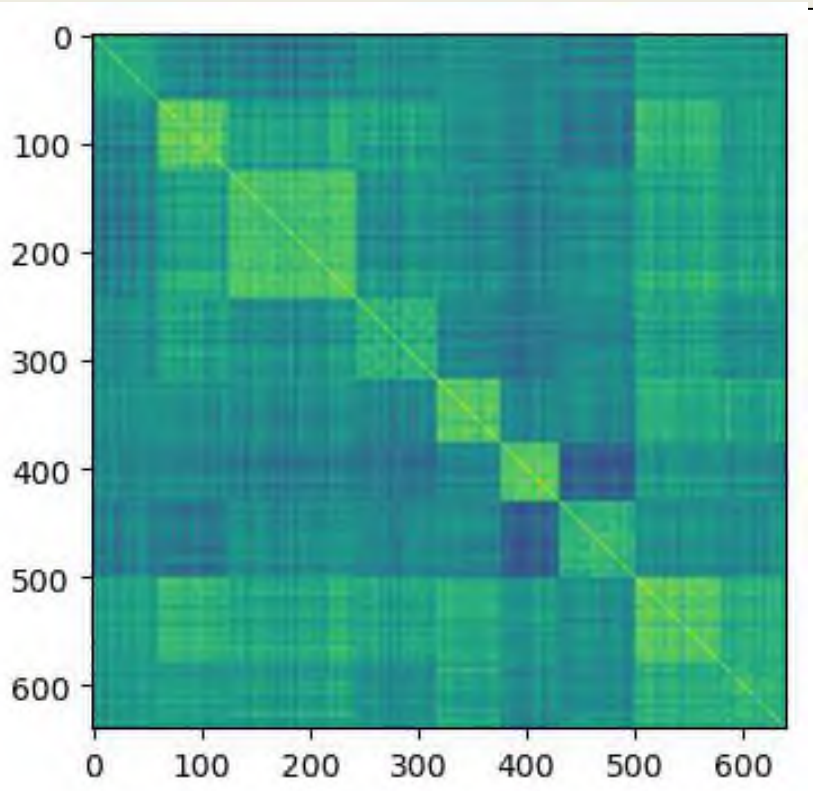


Πανεπιστήμιο Κύπρου

- Unfortunately, dtw_ndim.distance_matrix_fast does not compute distances across two different sets, meaning that it cannot compute te-tr explicitly
- In our case, since the dataset is small, we can compute the whole distance matrix and keep only the parts we need
- Note that some computations (tr-te and te-te) are wasted
- If the computing resources are limited or the dataset is too large, it's better to iterate through the elements of the training and test set and compute only the distances that we actually need

- The kernel is a similarity matrix with elements in [0,1]
- We can obtain it with the following transformation

```
dtw_sim = 1.0 - dtw_dist/dtw_dist.max()

# Plot the similarity matrix
idx_sorted = np.argsort(Y[:,0])
dtw_sim_sorted = dtw_sim[:,idx_sorted][idx_sorted,:]
fig =  plt.figure(figsize=(4,4))
plt.imshow(dtw_sim_sorted);
```



Πανεπιστήμιο Κύπρου

```
# Extract only kernels that we need
sim_trtr = dtw_sim[:Xtr.shape[0], :Xtr.shape[0]]
print(sim_trtr.shape)

sim_tetr = dtw_sim[Xtr.shape[0]:, :Xtr.shape[0]]
print(sim_tetr.shape)
```

(270, 270)
(370, 270)

```
clf = svm.SVC(kernel='precomputed', C=1).fit(sim_trtr, Ytr.ravel())
y_pred = clf.predict(sim_tetr)
accuracy = accuracy_score(Yte.ravel(), y_pred)
print(f"Accuracy: {accuracy*100:.2f}%")
```

Accuracy: 97.30%

Πανεπιστήμιο Κύπρου

- Once computed, the distance/similarity matrix can seamlessly be used with other classifiers
- One example is the classic k-NN classifier, which simply implements a majority vote:
    - A test sample is assigned the most frequent class among its nearest neighbors (k-NN) in the training set
- In our case, the neighbors are identified based on the DTW distance

```
# In this case we use the DTW distance directly
dtw_trtr = dtw_dist[:Xtr.shape[0], :Xtr.shape[0]]
dtw_tetr = dtw_dist[Xtr.shape[0]:, :Xtr.shape[0]]
```

```
neigh = KNeighborsClassifier(n_neighbors=3, metric='precomputed') # specify k=3
neigh.fit(dtw_trtr, Ytr.ravel())
y_pred = neigh.predict(dtw_tetr)
accuracy = accuracy_score(Yte.ravel(), y_pred)
print(f"Accuracy: {accuracy*100:.2f}%")
```

Accuracy: 96.49%

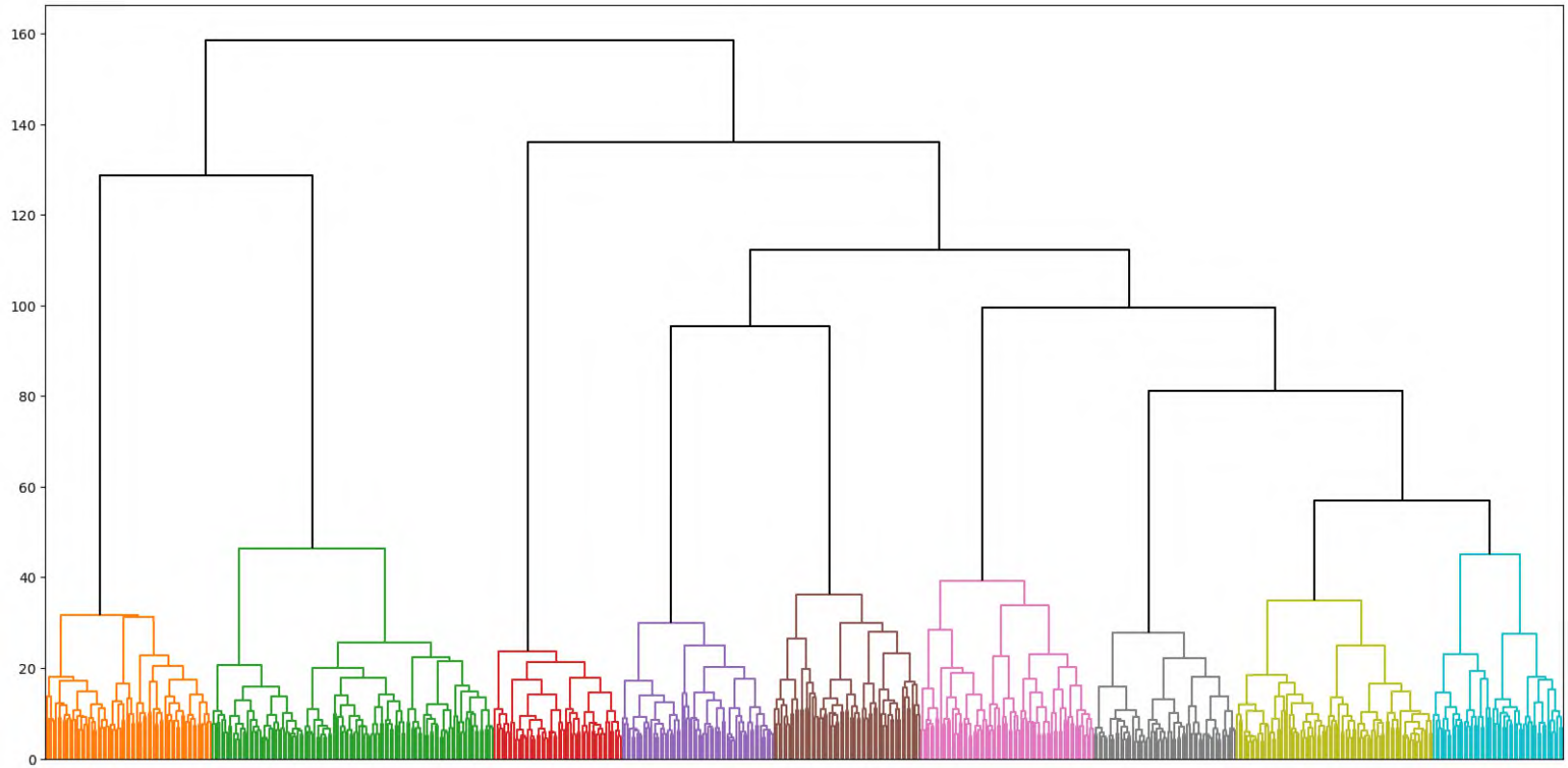
- To perform clustering we can use HC
- As before, we will use Ward Linkage to generate the hierarchy Z
- We need to pass the DTW dissimilarity matrix

```
distArray = ssd.squareform(dtw_dist)
Z = linkage(distArray, 'ward')
```

- To obtain the actual clusters we need to put a threshold
- To select the threshold, we look at the hierarchy Z with a dendrogram plot

```
fig = plt.figure(figsize=(20, 10))
dn = dendrogram(Z, color_threshold=50, above_threshold_color='k',
                show_leaf_counts=False)
plt.xticks([]);
```

- A value between 50 and 60 seems to give a stable partition



```
partition = fcluster(Z, t=55, criterion="distance")
print(f"Found {len(np.unique(partition))} clusters")
```

Found 9 clusters

```
print(f"DTW-based clustering NMI: {v_measure_score(partition, Y.ravel()):.2f}")
```

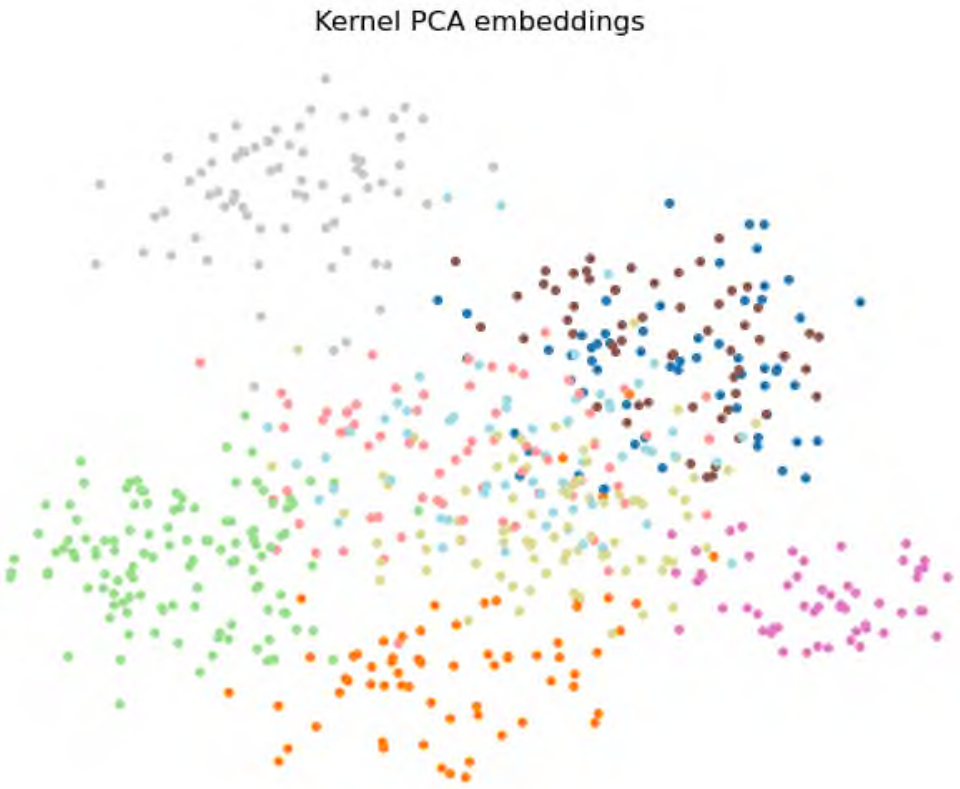
DTW-based clustering NMI: 0.95

Πανεπιστήμιο Κύπρου

- We have N samples, each one representing a multivariate time series of size $T \times V$
- Visualizing them directly is **impossible**
- Use PCA to reduce the data dimensionality
    - Reducing data to 2 or 3 dimension would make visualization possible
- If the data are vectors, i.e., $X \in \mathbb{R}^{N \times V}$, PCA first computes the empirical correlation matrix $X^T X \in \mathbb{R}^{V \times V}$ that captures the covariance among the features in the dataset
- PCA uses this information to project the data onto the directions (principal components) that maximize variance
- Unfortunately, the empirical correlation is meaningless if the data are time series because we are not interested in the correlation of individual time steps
- The problem further complicates if the time series are multivariate

- Thankfully PCA can compute principal components also through the eigendecomposition of the Gram matrix $XX^T \in \mathbb{R}^{N \times N}$
- The Gram matrix is a covariance matrix, i.e., a similarity matrix that captures linear relationships between data samples
    - As with the Euclidean distance, a linear covariance matrix is not suitable for time series
    - However, we can replace the covariance with another kernel matrix, such as the one derived from DTW
- The PCA algorithm that uses kernel matrices is called KernelPCA

```
# Kernel PCA
kpca = KernelPCA(n_components=2, kernel='precomputed')
embeddings_pca = kpca.fit_transform(dtw_sim)

fig =  plt.figure(figsize=(8,6))
plt.scatter(embeddings_pca[:,0], embeddings_pca[:,1], c=Y.ravel(), s=10,
cmap='tab20')
plt.title("Kernel PCA embeddings")
plt.gca().spines[['right', 'left', 'top', 'bottom']].set_visible(False)
plt.xticks(())
plt.yticks(());
```



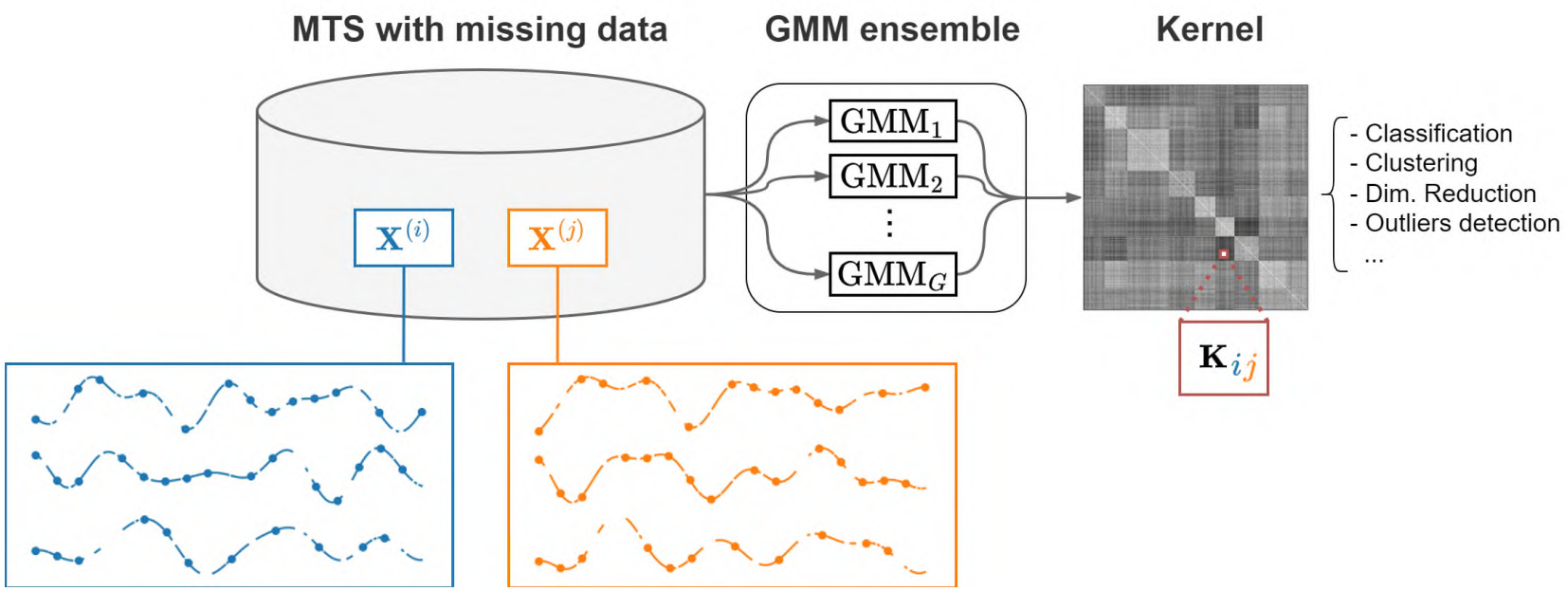
- KernelPCA maps each time series to a 2D point
- The DTW-based similarity caputures well the structure of the data and the relationships among the time series
- As a result, KernelPCA projects the classes in relatively well-separated groups

Πανεπιστήμιο Κύπρου

- A kernel for time series data is a mathematical function used to measure the similarity between two time series

$$k(x, y) \geq 0$$

- Consider the Time Series Cluster Kernel (TCK), which offers the following advantages:
  - Suitable for multi-variate time series
  - Can deal with missing data
  - Robust to hyperparameters selection
- TCK combines Gaussian Mixture Models (GMM) with an ensemble learning approach to build a kernel

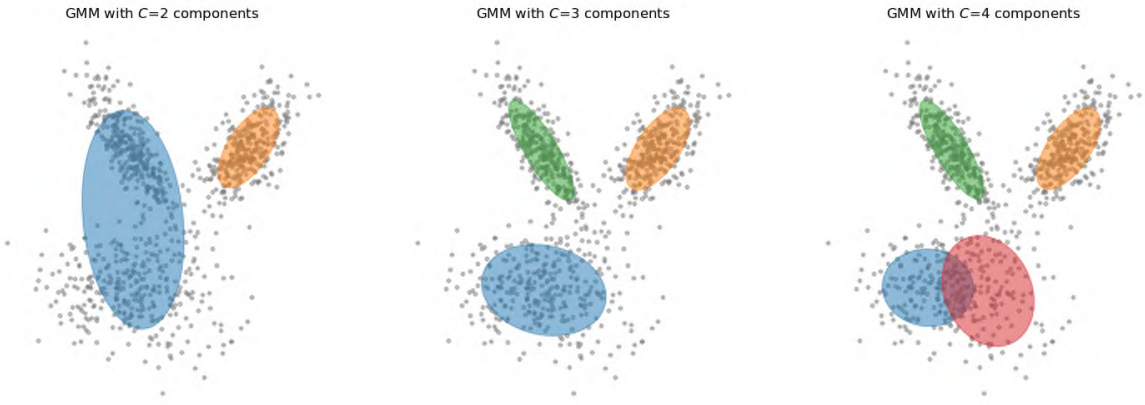


Πανεπιστήμιο Κύπρου

- A GMM assumes that all the data points are generated from a mixture of C Gaussian distributions (components) with unknown parameters $\{\mu_c, \Sigma_c\}_{c=1}^{C}$
- GMMs are used for clustering
- Each cluster is modeled by a Gaussian distribution.

```
# Create toy data
X, y = datasets.make_classification(n_samples=950, n_features=2, n_informative=2,
                                    n_redundant=0, n_clusters_per_class=1,
random_state=4,
                                    n_classes=3, class_sep=1.5, flip_y=0.1)
```

```
_, axes = plt.subplots(1, 3, figsize=(18, 6))
for i, n in enumerate([2, 3, 4]):
    plot_gmm(X, n, ax=axes[i])
    axes[i].set_title(f"GMM with $C$={n} components")
```



- Mathematically, a GMM is defined as:

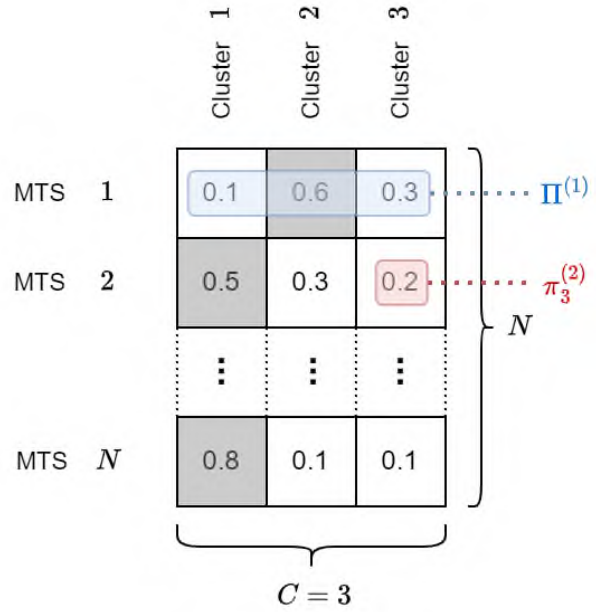
$$p(x) = \sum_{c=1}^{C} \pi_c \mathrm{N}\left(x \middle| \mu_c, \Sigma_c\right)$$

- π_c is the mixing coefficient of the c-th Gaussian, with $0 \leq \pi_c \leq 1$ and $\sum_{c=1}^{C} \pi_c = 1$
- π_c indicates how much a data point x belongs to the c-th Gaussian $\mathrm{N}\left(x \middle| \mu_c, \Sigma_c\right)$

- Clustering with GMM involves estimating the parameters π_c, μ_c, Σ_c using the Expectation-Maximization (EM) algorithm:
    1. iteratively assigns data points to clusters (expectation step)
    2. Then, updates the parameters to maximize the likelihood of the data given the current clusters (maximization step)

- In the end, each data point is assigned with a probability of belonging to each cluster
- The output is a soft clustering where points can belong to multiple clusters with different probabilities

- The output can be represented by a soft cluster assignment matrix $\Pi \in \mathbb{R}^{N \times C}$
- The (i, c)-th element of is the membership of the MTS i to cluster c: $\Pi[i, c] = \pi_c{}^{(i)}$
- The i-th row of Π represent all the memberships of the -th MTS: $\Pi[i, :] = \Pi^{(i)}$
- Crisp cluster assignments (cells in gray in the figure) are obtained by taking the maximum value in $\Pi^{(i)}$



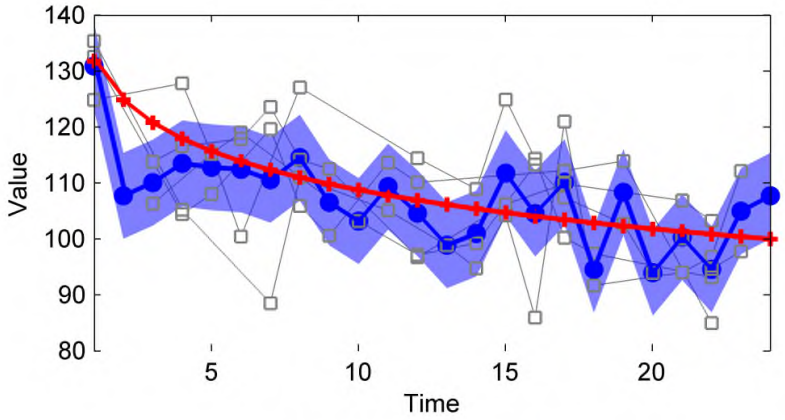
Πανεπιστήμιο Κύπρου

- TCK modifies the standard GMM model in two ways

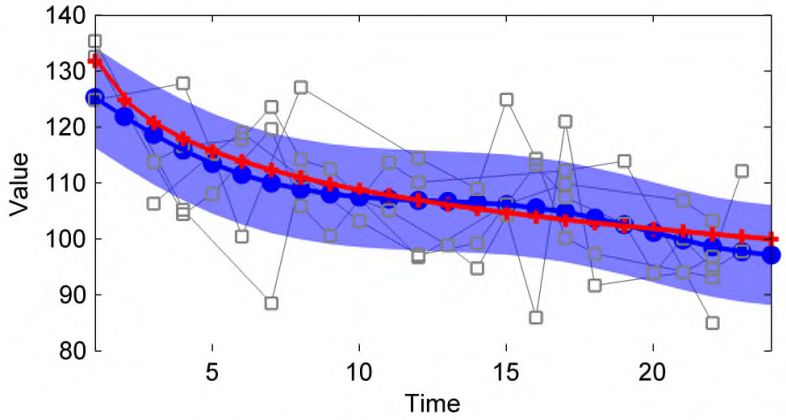
1. To handle time series data, the means of the GMM model become multi-variate time series.
   - Standard GMM: $\mu_c \in \mathbb{R}^V$
   - TCK GMM: $\mu_c \in \mathbb{R}^{T \times V}$
2. To handle missing values, TCK puts priors on the GMM parameters and replaces the EM algorithm with Maximum a-posteriori (MAP) estimation.
   - EM: $\hat{\mu}, \hat{\Sigma} = argmax_{\mu,\Sigma} p(X|\mu,\Sigma)$
   - MAP with priors $\hat{\mu}, \hat{\Sigma} = argmax_{\mu,\Sigma} p(X|\mu,\Sigma)p(\mu,\Sigma)$

- The resulting means are smoother and the parameters are similar to the overall mean and covariance in clusters with few samples

- The resulting means are smoother and the parameters are similar to the overall mean and covariance in clusters with few samples



(a) ML Estimate



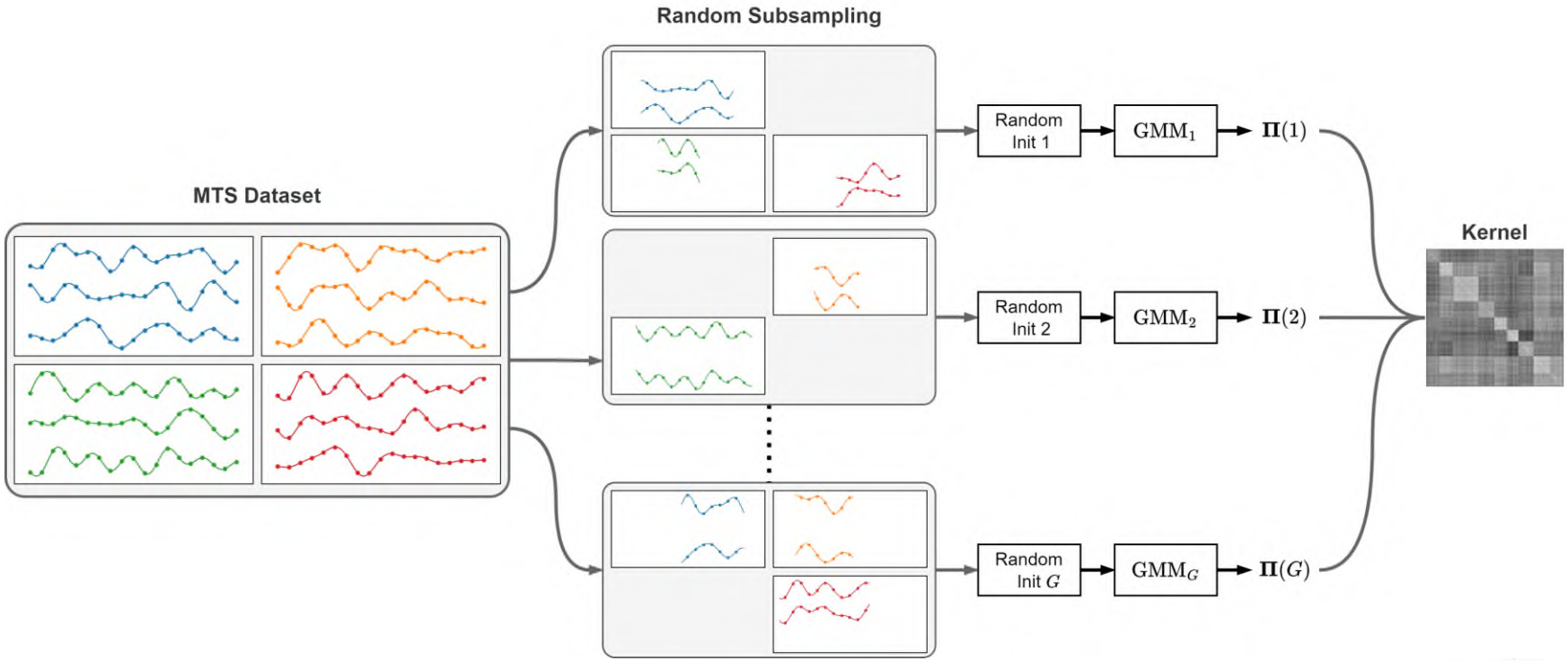
(b) MAP Estimate

- Ensemble approach: combine many weak models to obtain a more powerful one.
- Reduces the sensitivity to hyperparameters selection.
- Can produce a well-defined kernel
- The ensemble of GMMs is obtained by fitting G different GMMs on:
  - A subset of the N MTS in the dataset
  - A subset of the V variables
  - A segment of indices of length $\leq T$ in the time series

The kernel matrix is obtained by combining the clustering results from the G GMMs in the ensemble

Specifically, $k(x, y)$ is proportional to how many times x and y are assigned to the same GMM:

$$k(x,y) = \sum_{g \in G} \Pi^{(x)}(g)^T \Pi^{(y)}(g)$$



```
# Load the data
Xtr, Ytr, Xte, Yte = DataLoader().get_data('Japanese_Vowels')
# Randomly remove 40% to show how TCK handles missing values
mask_tr = np.random.choice([0, 1], size=Xtr.shape, p=[0.6, 0.4])
Xtr[mask_tr == 1] = np.nan
mask_te = np.random.choice([0, 1], size=Xte.shape, p=[0.6, 0.4])
Xte[mask_te == 1] = np.nan
```

- Create a TCK kernel with G=30 GMMs, each with C=15 components
  - Higher G the better the performance but the higher the computation time
  - C controls the model complexity. Too low underfit, too high overfit.
- Then, fit TCK on the training data
- As in the DTW case, we compute the two kernels K_{tr-tr} and K_{tr-te} to train and test our classifier

```
tck = TCK(G=30, C=15)
Ktr = tck.fit(Xtr).predict(mode='tr-tr')
Kte = tck.predict(Xte=Xte, mode='tr-te').T
print(f"Ktr shape: {Ktr.shape}\nKte shape: {Kte.shape}")
```

The dataset contains missing data

Training the TCK using the following parameters:

C = 15, G = 30

Number of MTS for each GMM: 216 - 270 (80 - 100 percent)

Number of attributes sampled from [2, 11]

Length of time segments sampled from [6, 23]

Ktr shape: (270, 270)

Kte shape: (370, 270)

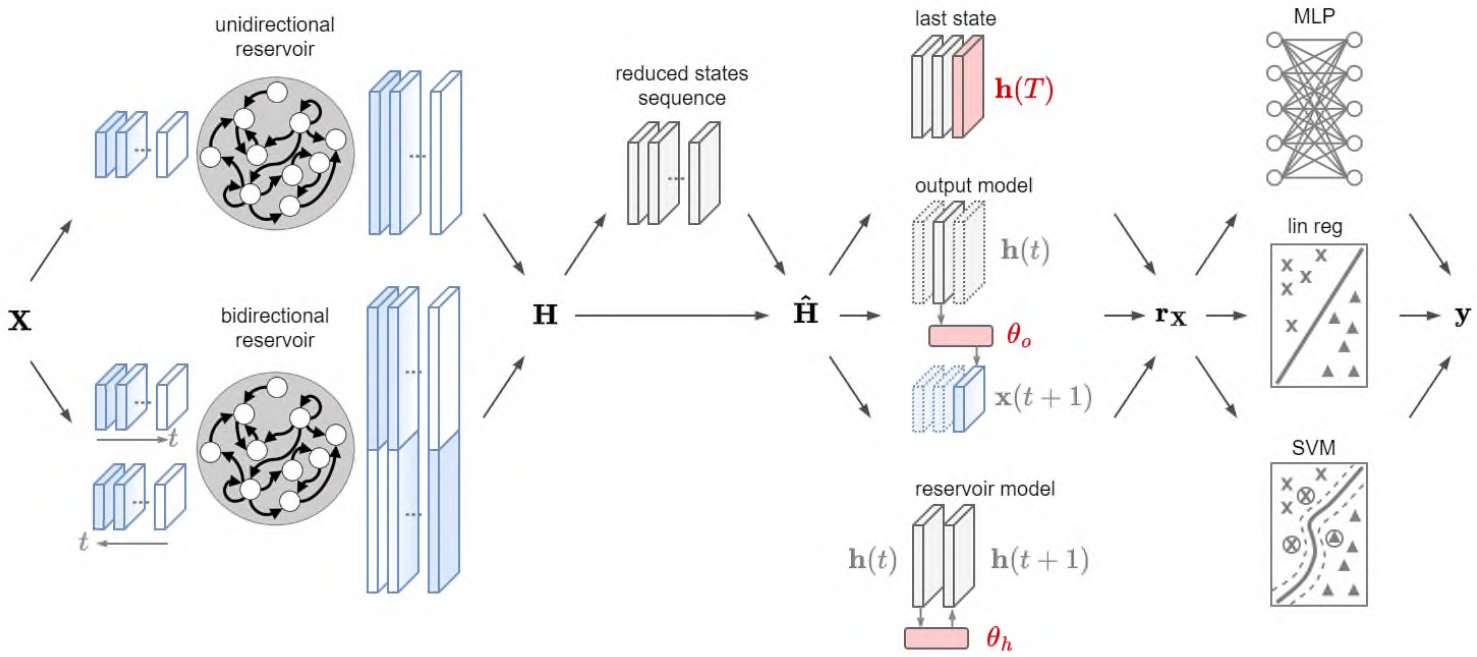
```
clf = svm.SVC(kernel='precomputed').fit(Ktr, Ytr.ravel()) # Train
Ypred = clf.predict(Kte) # Test
print(f" Test accuracy: {accuracy_score(Yte, Ypred):.2f}")
```

Test accuracy: 0.92

- Even with 40% of missing data, with TCK we maintain a good classification performance

Πανεπιστήμιο Κύπρου

- Another approach is to embed the whole MTS into a real-valued vector
- Allows us to use standard (dis)similarity measures for vectorial data (cosine similarity, Euclidean distance, etc)
- The key problem is how to embed the temporal information into a static vector in a meaningful way
- There are many approaches for extracting static features from a time series
- Here we will use Reservoir Computing (RC) to generate embeddings of MTS



- A RC framework consisting of 4 modules:
  - Reservoir module
  - Dimensionality reduction module
  - MTS representation module
  - Readout module

Πανεπιστήμιο Κύπρου

- This module generates a sequence of Reservoir states from a given MTS x

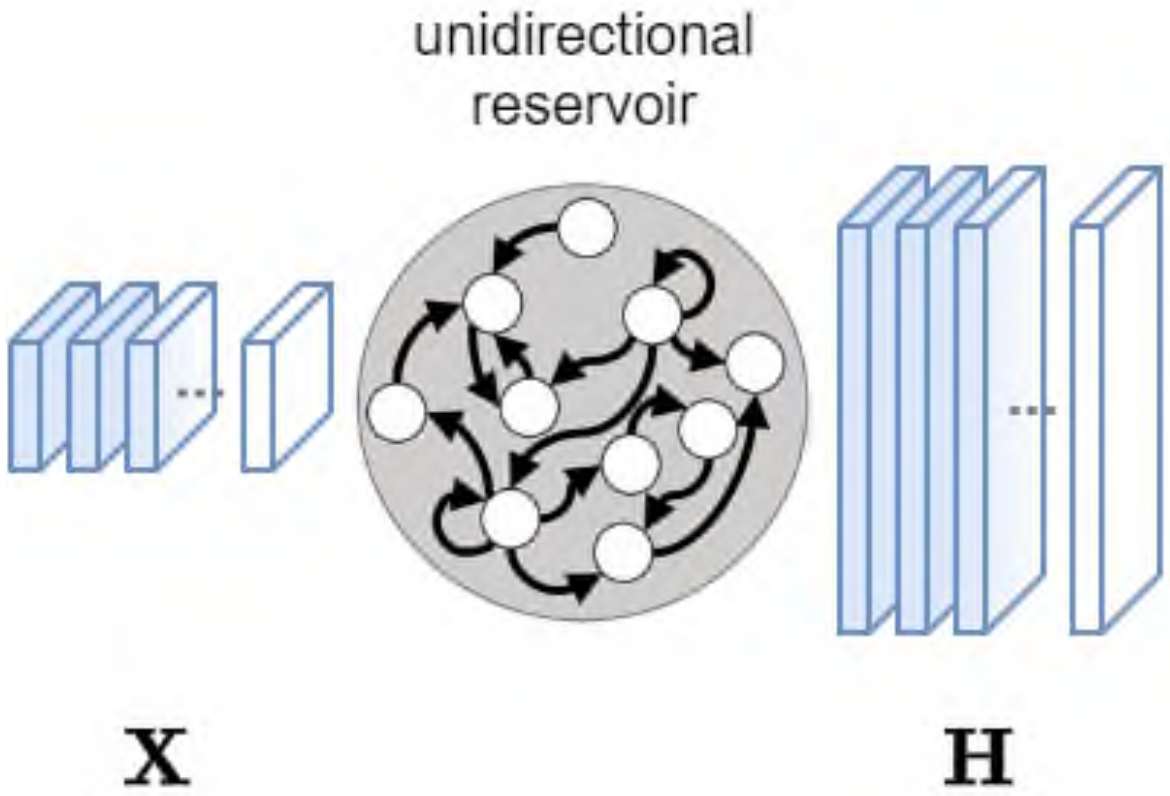
$$h(t) = \sigma\left(W_i x(t) + W_h h(t-1)\right)$$

- If we have N MTS they can be processed in parallel by the Reservoir and generate the sequence of states

$$\{H(1), H(2), \dots, H(T)\}$$

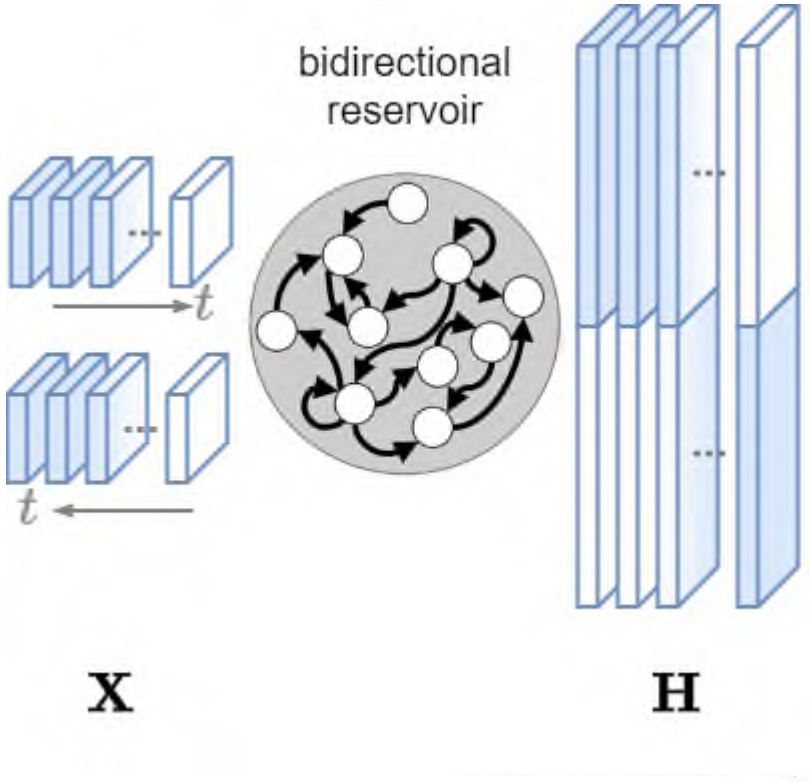
- with $H(t) \in \mathbb{R}^{N \times H}$
- The Reservoir can operate in two modalities:
  - Unidirectional
  - Bidirectional

- This is the same Reservoir we saw previously
  - Input: MTS data X of shape [N, T, V]
  - Output: a sequence of states H of shape [N, T, H].



- This Reservoir processes the time series also backwards
- Allows to retrieve context from both past and future data points capture more complex and longer temporal dependencies
- A Bidirectional Reservoir is not causal: it cannot be used for forecasting but is suitable for classification and clustering

- Input: MTS data X of shape [N, T, V].
- Output: a sequence of states H of shape [N, T, 2*H].



```
Xtr, Ytr, Xte, Yte = DataLoader().get_data('Japanese_Vowels’)

H_uni = Reservoir(n_internal_units=300).get_states(Xtr, bidir=False)
print(f"Unidir\n  H: {H_uni.shape}")

H_bi = Reservoir(n_internal_units=300).get_states(Xtr, bidir=True)
print(f"Bidir\n  H: {H_bi.shape}")
```

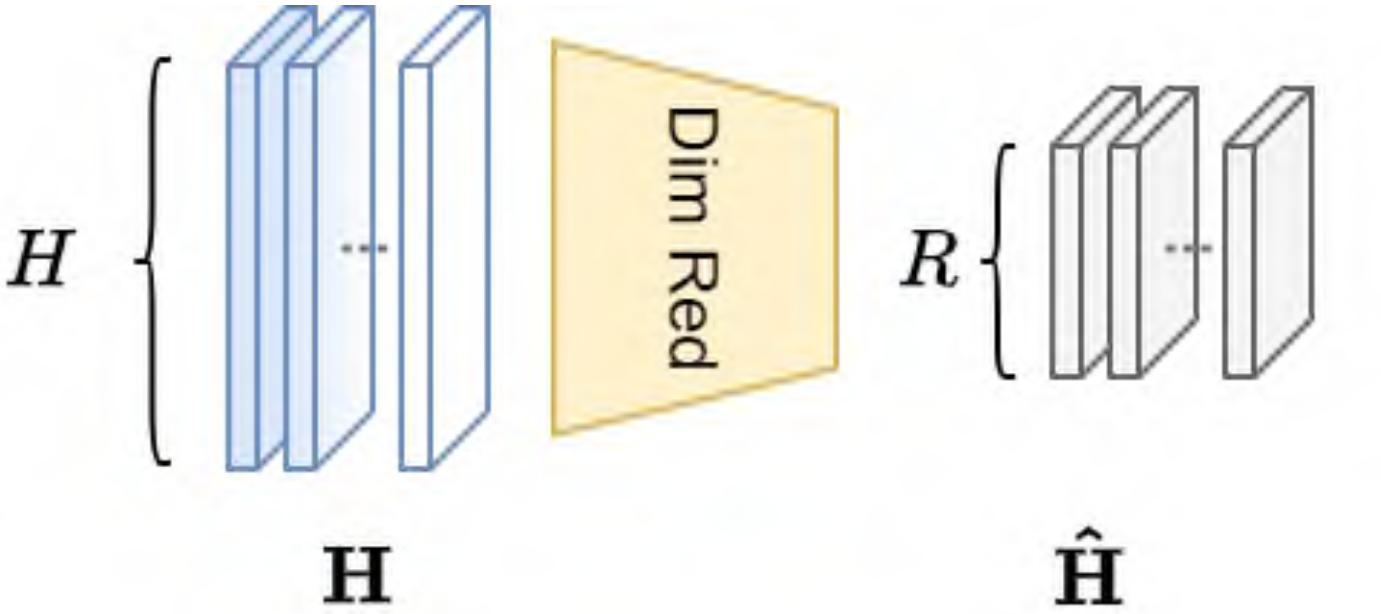
Unidir

H: (270, 29, 300)

Bidir

H: (270, 29, 600)

- This module reduces the dimensionality of the Reservoir states from [N,T,H] (or [N,T,2*H]) to [N,T,R]
- Dramatically speeds up computations especially when using more **advanced representations**
- Dimensionality reduction can be implemented by:
  - Standard PCA
  - Tensor PCA



- Standard PCA
  - Only works on uni-dimensional data
  - Reshape H from [N,T,H] to [N*T,H]
  - Apply PCA and keep the first R components
  - Reshape [N*T,R] back to [N,T,R]
- Tensor PCA
  - Compute the following covariance matrix

$$S = \frac{1}{N-1}\sum_{n=1}^{N}(H_n - \bar{H})^T(H_n - \bar{H}) \in \mathbb{R}^{H\times H}$$

  - where $H_n \in \mathbb{R}^{H\times H}$ is obtained as H[n,:,:] and $\bar{H} = \frac{1}{N}\sum_{n=1}^{N} H_n \in \mathbb{R}^{T\times H}$
  - This allows to compute the variations across the Reservoir dimension by keeping sample- and time-dimension separated
  - Then take the first R eigenvectors of S: $D = [u_1, u_2, \dots, u_R] \in \mathbb{R}^{H\times R}$ and the reduced state $\widehat{H}_n = H_n D$

```
H_red = tensorPCA(n_components=75).fit_transform(H_bi)
print(f"H_red: {H_red.shape}")
```

H_red: (270, 29, 75)

- This module is responsible of transforming the sequence of Reservoir states into a vectorial representation r_x
- The Reservoir extracts and separates the dynamical features
- In addition, it keeps a memory of all the past input
- Therefore, in some cases, is sufficient to keep only the last Reservoir state to represent the whole MTS

```
rx_last = H_red[:,-1,:]
print(f"rx_last: {rx_last.shape}")
```

rx_last: (270, 75)

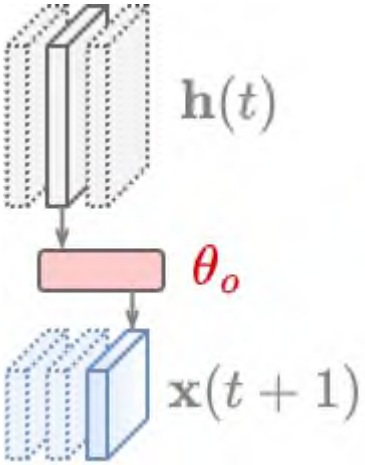
- A more effective representation is obtained as follows
  - Train a linear readout to predict the MTS one step-ahead

$$x(t+1) = h(t)W_0 + w_0$$

  - The parameters of the linear model

$$\theta_0 = [vec(W_0); w_0] \in \mathbb{R}^{V(R+1)}$$

  - become the r_x



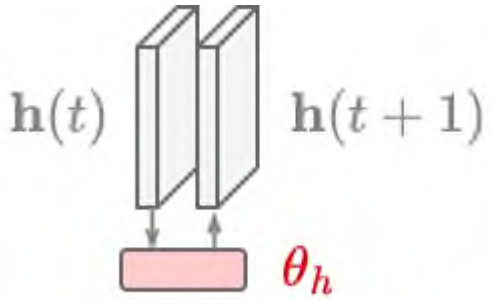

```
out_pred = Ridge(alpha=1.0)
# If we use a bidirectional Reservoir we also need to predict the time series
backwards
X = np.concatenate((Xtr, Xtr[:, ::-1, :]), axis=2)
coeff, biases = [], []
for i in range(X.shape[0]):
    out_pred.fit(H_red[i, 0:-1, :], X[i, 1:, :])
    coeff.append(out_pred.coef_.ravel())
    biases.append(out_pred.intercept_.ravel())
rx_out = np.concatenate((np.vstack(coeff), np.vstack(biases)), axis=1)

print(f"rx_out: {rx_out.shape}") # [N, 2*V*(R+1)]
```

rx_out: (270, 1824)

- A similar approach is to use the coefficients of a linear model that predicts the next state of the Reservoir

$$h(t+1) = h(t)W_h + w_h$$

- The MTS representation r_x becomes $\theta_h = [vec(W_h); w_h] \in \mathbb{R}^{R(R+1)}$




```
res_pred = Ridge(alpha=1.0)
coeff, biases = [], []
for i in range(H_red.shape[0]):
    res_pred.fit(H_red[i, 0:-1, :], H_red[i, 1:, :])
    coeff.append(res_pred.coef_.ravel())
    biases.append(res_pred.intercept_.ravel())
rx_res = np.concatenate((np.vstack(coeff), np.vstack(biases)), axis=1)

print(f"rx_res: {rx_res.shape}") # [N, R*(R+1)]
```

rx_res: (270, 5700)

- In principle, the Reservoir model space θ_h provides a better representation than the Output model space θ_0
- The Reservoir generates a large pool of dynamics but only few are needed to predict the input at a specific forecast horizon, i.e., 1 if we predict $x(t+1)$
- Not being useful for the task, the other dynamics are discarded even if they are still useful to characterize the MTS
- On the other hand, to predict the next Reservoir state $h(t+1)$ is necessary to consider all Reservoir dynamics
- This makes θ_h a more powerful representation as it fully characterizes the MTS

- The readout module is responsible to classify or cluster the MTS representations
- Being each representation r_x a vector, any standard classifier for vectorial data can be used
- Similarly, we can use standard (dis)similarity measures for vectorial data
- For example, clustering can be done using the Linkage aglorithm on the Euclidean distances between MTS representations

- We can use the high-level function RC_model to perform the classification
- The function takes as input the hyperparameters to configure:
    1. the Reservoir module,
    2. the dimensionality reduction module,
    3. the representation module,
    4. the readout module

Store the hyperparameters in a Python dictionary

```
config = {}
# Hyperarameters of the reservoir
config['n_internal_units'] = 450          # size of the reservoir
config['spectral_radius'] = 0.9           # largest eigenvalue of the reservoir
config['leak'] = None                     # amount of leakage in the reservoir
state update (None or 1.0 --> no leakage)
config['connectivity'] = 0.25             # percentage of nonzero connections in
the reservoir
config['input_scaling'] = 0.1             # scaling of the input weights
config['noise_level'] = 0.01              # noise in the reservoir state update
config['n_drop'] = 5                      # transient states to be dropped
config['bidir'] = True                    # if True, use bidirectional reservoir
config['circle'] = False                  # use reservoir with circle topology

# Dimensionality reduction hyperparameters
config['dimred_method'] = 'tenpca'        # options: {None (no dimensionality
reduction), 'pca', 'tenpca'}
config['n_dim'] = 75                      # number of resulting dimensions after
the dimensionality reduction procedure

# Type of MTS representation
config['mts_rep'] = 'reservoir'           # MTS representation: {'last', 'mean',
'output', 'reservoir'}
config['w_ridge_embedding'] = 10.0        # regularization parameter of the ridge
regression

# Type of readout
config['readout_type'] = 'lin'            # readout used for classification:
{'lin', 'mlp', 'svm'}
config['w_ridge'] = 5.0                   # regularization of the ridge regression
readout
```

- Create a RC classifier by passing the configuration parameters

```
classifier =  RC_model(**config)

Xtr, Ytr, Xte, Yte = DataLoader().get_data('Japanese_Vowels')
```

- The RC model expects class labels to be one-hot-encoded, e.g.
- $1 \rightarrow [1,0,0,0,\ldots,0], 2 \rightarrow [0,1,0,0,\ldots,0], 3 \rightarrow [0,0,1,0,\ldots,0]$, etc

```
# One-hot encoding for labels
onehot_encoder = OneHotEncoder(sparse_output=False)
Ytr = onehot_encoder.fit_transform(Ytr)
Yte = onehot_encoder.transform(Yte)
# Train the model
tr_time = classifier.fit(Xtr, Ytr)
# Compute predictions on test data
pred_class = classifier.predict(Xte)
accuracy, f1 = compute_test_scores(pred_class, Yte)
print(f"Accuracy = {accuracy:.3f}, F1-score = {f1:.3f}")
```

Accuracy = 0.978, F1-score = 0.978

- We will perform the following steps:
    1. Generate the vectorial representations for all MTS
    2. Compute a dissimilarity matrix
    3. Perform clustering with the distance-based HC algorithm Linkage
- Since we are doing clustering, we do not need the train/test split.
- Can concatenate the data from both sets together

```
Xtr, Ytr, Xte, Yte = DataLoader().get_data('Japanese_Vowels')

X = np.concatenate((Xtr, Xte), axis=0)
Y = np.concatenate((Ytr, Yte), axis=0)
```

- Can re-use the same RC_model as before
- The only difference is that we do not want to apply the readout module
- Instead, it should return the MTS representations
- This is achieved by setting 'readout_type'to None

```
config['readout_type'] = None # We update this entry from the previous config
dict
# Instantiate the RC model
rcm =  RC_model(**config)

# Generate representations of the input MTS
rcm.fit(X)
mts_representations = rcm.input_repr
```

- Then compute the distance matrix using standard metrics for vectorial data
    - For example, use the Euclidean or a Cosine distance
- The latter is often preferred in high-dimensional spaces because Euclidean distance can become inflated and less meaningful ("curse of dimensionality")

```
# Compute Dissimilarity matrix
Dist = cosine_distances(mts_representations)
distArray = ssd.squareform(Dist)

# Hierarchical clustering
Z = linkage(distArray, 'ward')
clust = fcluster(Z, t=4.0, criterion="distance")
print(f"Found {len(np.unique(clust))} clusters")

# Evaluate the agreement between class and cluster labels
nmi = v_measure_score(Y[:,0], clust)
print(f"Normalized Mutual Information (v-score): {nmi:.3f}")
```

Found 9 clusters
Normalized Mutual Information (v-score): 0.906

Πανεπιστήμιο Κύπρου

- Introduced the problem of classification and clustering
- The importance of choosing a proper measure for computing (dis)similarities
- Introduced three approaches to compute (dis)similarities across multivariate time series
    1. DTW, an alignment based-metric
    2. TCK, a kernel similarity
    3. RC embeddings, an approach to embed time series into vectorial data
- These (dis)similarity measures are the cornerstone in time series classification and clustering
- Once computed, we saw how they can be easily plugged into the same classification and clustering method we saw for vectorial data
- We conclude by highlighting the main pros and cons of the three approaches to compute MTS (dis)similarity

DTW

- ✅ In most cases, works well with default hyperparameters
- ✅ Invariant to translations in time
- ❌ Slow
- ❌ Does not account for complex dynamical features

TCK

- ✅ Hyperparameters are easy to set
- ✅ Handles missing data
- ❌ Very slow

RC-embedding

- ✅ Fast
- ✅ Captures complex dynamical features
- ❌ Many hyperparameters to set.

- Each approach can achieve better or worse performance depending on the data and the problem at hand.
- Selecting the optimal (dis)similarity measure, classification/clustering algorithm, and hyperparameters is often a difficult procedure.
- It requires experience and should be performed with systematic approaches such as cross-validation