Time Series Analysis

Time Series Databases

- A time series is a sequence of real numbers, representing the measurements of a real variable at (equal) time intervals
  - Stock prices
  - Volume of sales over time
  - Daily temperature readings
  - ECG data
- A time series database is a large collection of time series

Time Series Problems
(from a database perspective)

- The Similarity Problem

  $X = x_1, x_2, \ldots, x_n$ and $Y = y_1, y_2, \ldots, y_n$

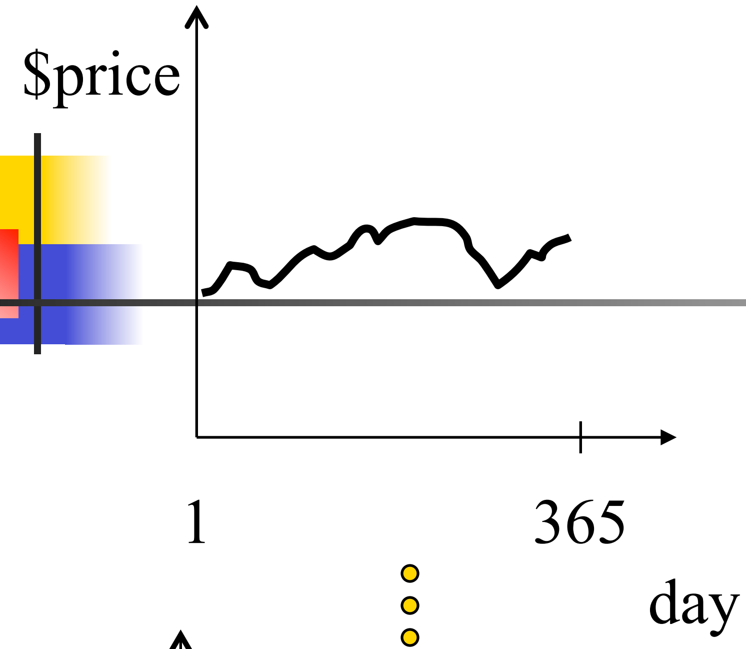
- Define and compute Sim(X, Y)
  - E.g. do stocks X and Y have similar movements?
- Retrieve efficiently similar time series (Indexing for Similarity Queries)

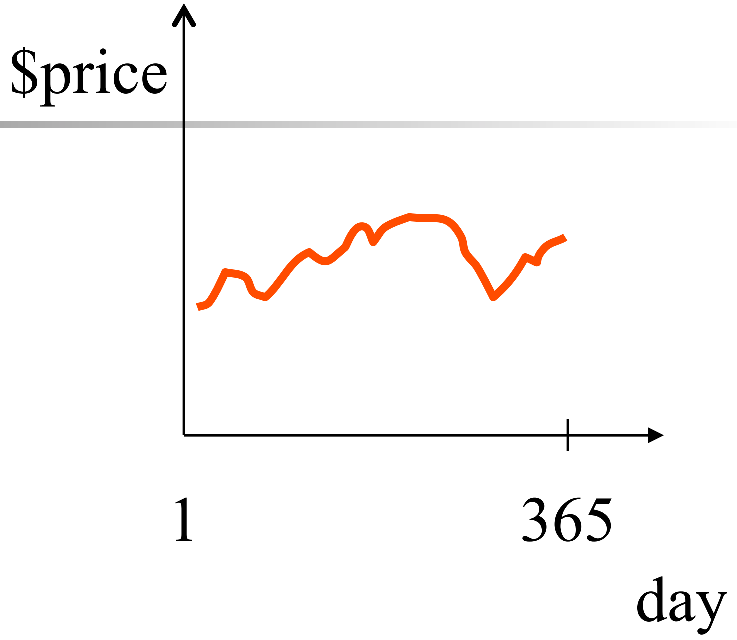
Types of queries

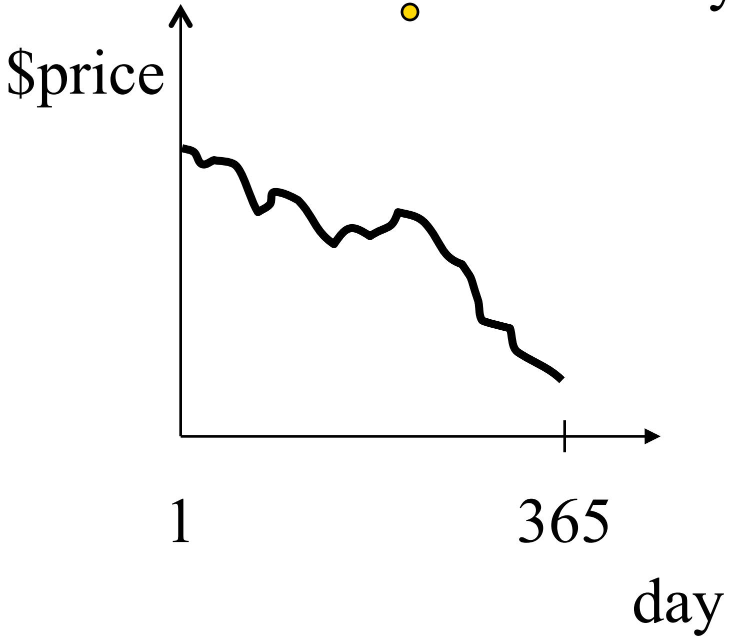
- whole match vs sub-pattern match
- range query vs nearest neighbors
- all-pairs query

Examples

- Find companies with similar stock prices over a time interval
- Find products with similar sell cycles
- Cluster users with similar credit card utilization
- Find similar subsequences in DNA sequences
- Find scenes in video streams







distance function: by expert

(eg, Euclidean distance)

Problems

- Define the similarity (or distance) function
- Find an efficient algorithm to retrieve similar time series from a database
    - (Faster than sequential scan)

The Similarity function depends on the Application

Euclidean Similarity Measure

- View each sequence as a point in d-dimensional Euclidean space (d = length of each sequence)
- Define (dis-)similarity between sequences X and Y as

$$\boxed{L_p = (\sum_{i=1}^{d} | x_i - y_i |^p)^{1/p}}$$

p=1 Manhattan distance

p=2 Euclidean distance

Advantages

- Easy to compute: O(n)
- Metric
- Allows scalable solutions to other problems, such as
  - indexing
  - clustering
  - etc...

Similarity Retrieval

- Range Query
  - Find all time series S where $D(Q,S) \leq \varepsilon$
- Nearest Neighbor query
  - Find all the k most similar time series to Q
- A method to answer the above queries: Linear scan … very slow

- A better approach GEMINI

GEMINI

Solution: Quick-and-dirty' filter:

- extract *k* features (numbers, eg., avg., etc.)
- map into a point in *k*-dimensional feature space, where k << d
- retrieve the answer using a NN query
- discard false alarms

GEMINI

- GEMINI works when:

  $D_{feature}(F(x), F(y)) <= D(x, y)$

- *Note that, the closer the feature distance to the actual one, the better.*
- *GEMINI can be used also for any multimedia dataset, not only time series*

Map to Feature space

- How to extract the features? How to define the feature space?

Answer:

- Fourier transform
- Wavelets transform
- Averages of segments (Histograms or APCA)
- …..

Problems of Euclidean distance

Consider the TS in the picture. If we use the Euclidean distance:
d(t1,t2) = 26.9
d(t1,t3) = 23.2!!!!

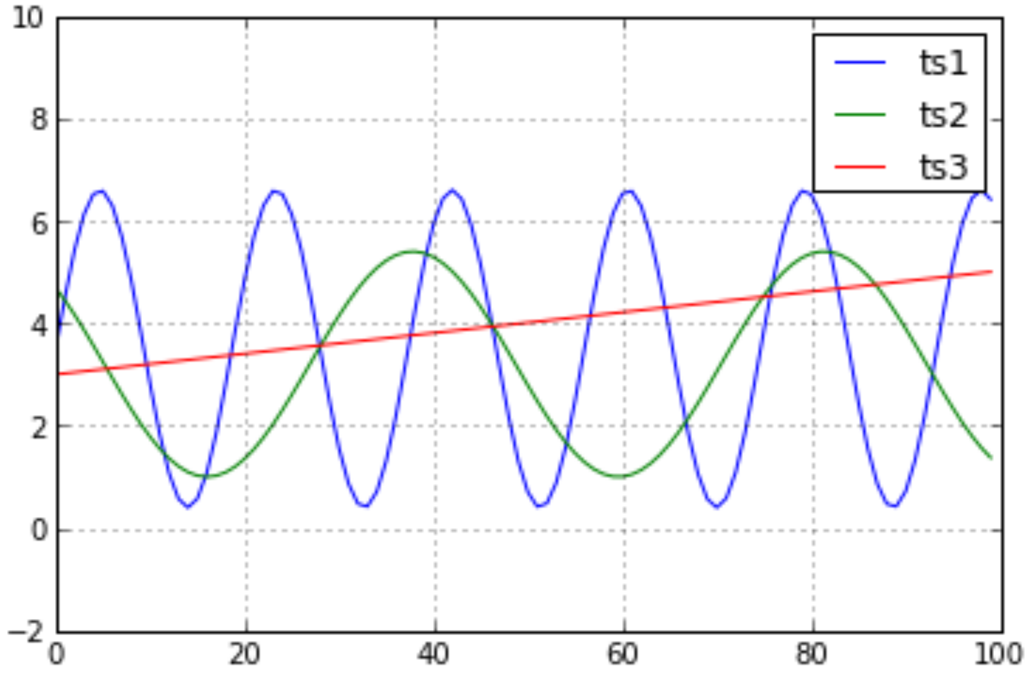
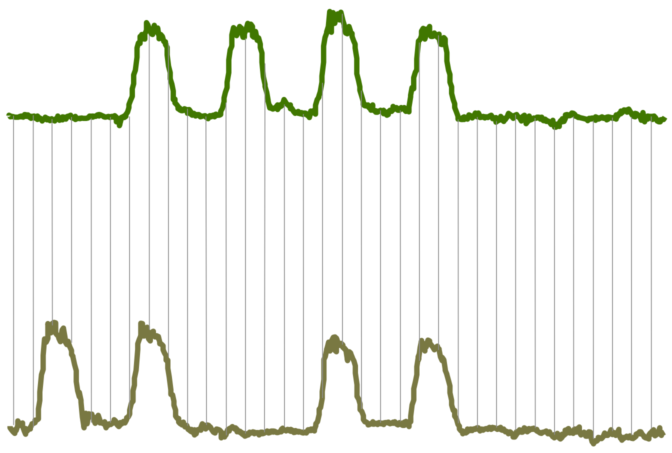


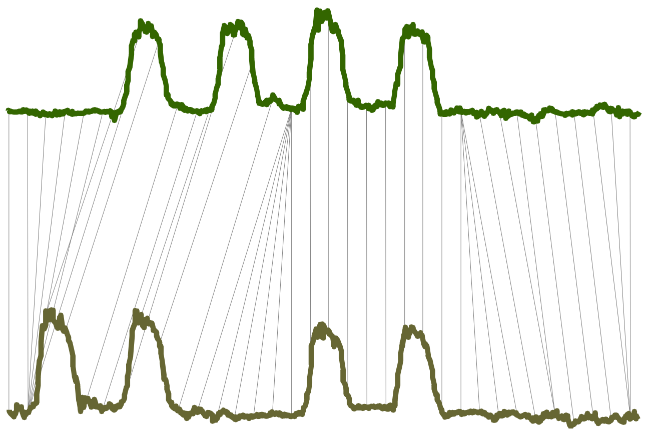
Image from:
http://alexminnaar.com/time-series-classification-and-clustering-with-python.html

Dynamic Time Warping

- Allows acceleration-deceleration of signals along the time dimension
- Basic idea
  - Consider X = $x_1, x_2, \ldots, x_n$, and Y = $y_1, y_2, \ldots, y_m$
  - We are allowed to extend each sequence by repeating elements
  - Euclidean distance now calculated between the extended sequences X' and Y'
  - Matrix M, where $m_{ij} = d(x_i, y_j)$

Example





Images from: Eamonn Keogh's slides

Euclidean distance vs DTW

Formulation

- Let D(i, j) refer to the dynamic time warping distance between the subsequences (L_1 based distance)

  $x_1, x_2, \ldots, x_i$

  $y_1, y_2, \ldots, y_j$

  $$D(i, j) = d(x_i - y_j) + \min \{ D(i-1, j), \; D(i-1, j-1), \; D(i, j-1) \}$$

  Dynamic programming!

  - Optimal substructure

  - Overlapping sub problems

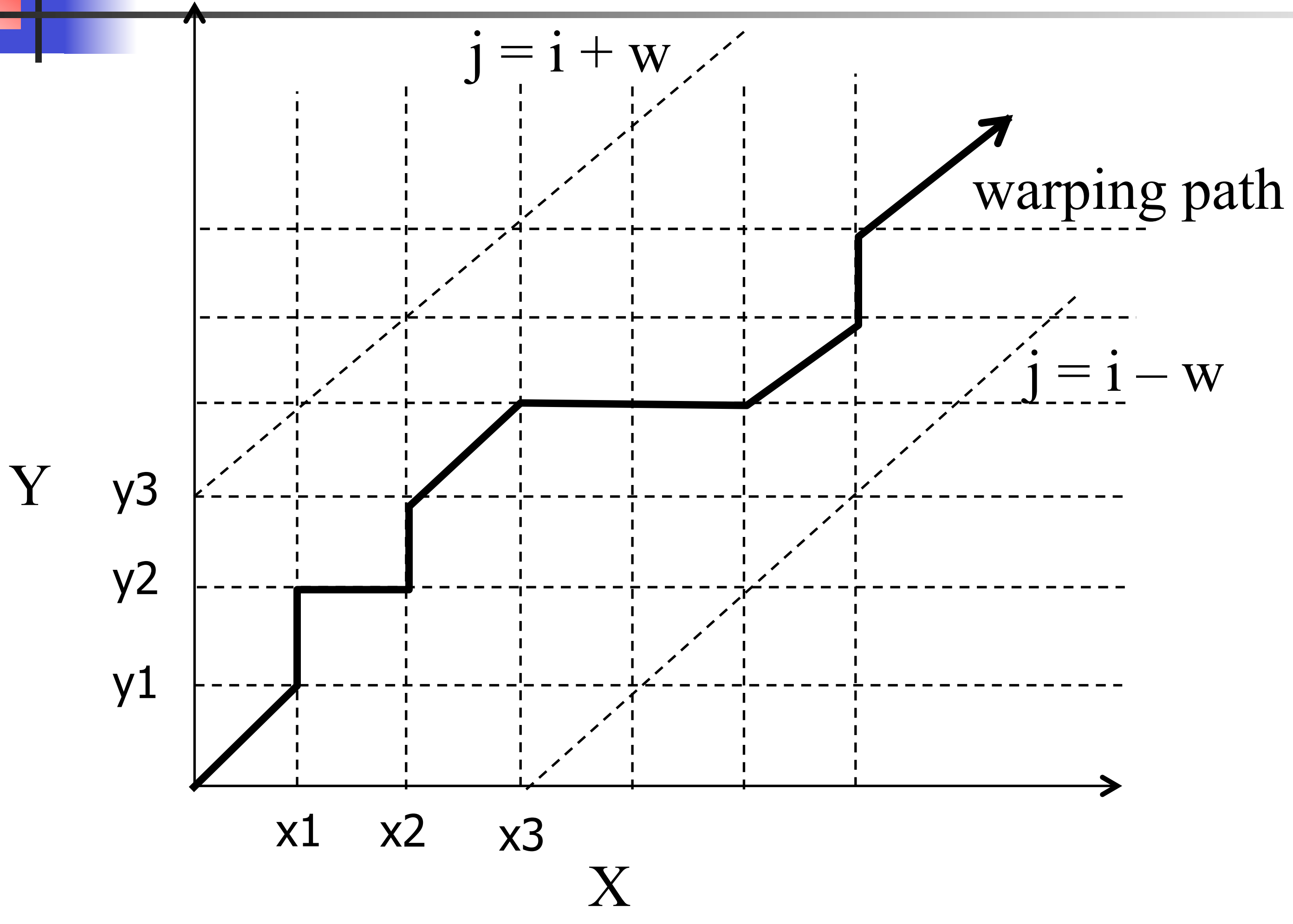
DTW variations

- If we use Euclidean distance to compute the difference between two elements x_i and y_j then we can use:

$$D(i, j) = L_2(x_i - y_j) + \min \{ \begin{array}{l} D(i-1, j), \\ D(i-1, j-1), \\ D(i, j-1) \} \end{array}$$

and take the square root of D(n,m) at the end.

Dynamic Time Warping



Restrictions on Warping Paths

- Monotonicity
  - Path should not go down or to the left
- Continuity
  - No elements may be skipped in a sequence
- Warping Window

  $|i - j| \le w$

Solution by Dynamic Programming

- Basic implementation = O(nm) where n, m are the lengths of the sequences
  - will have to solve the problem for each (i, j) pair

- If warping window is specified, then O(nw)
  - Only solve for the (i, j) pairs where | i – j | <= w

Edit Distance for Strings

- Given two strings sequences we define the distance between the sequences as the minimum number of edit operation that need to be performed to transform one sequence to the other. Edit operation are insertion, deletion and substitution of single characters.

- This is also called **Levenshtein distance**

Computing Edit distance

- We can compute edit distance using the following function:

$$ed(i,j) = \begin{cases} ed(i-1, j-1) & \text{if } x_i = y_j \\ \min\big(ed(i-1,j)+1, \\ \quad ed(i, j-1)+1, \\ \quad ed(i-1, j-1)+1\big) & \text{if } x_i \neq y_j \end{cases}$$

for two strings $x_1x_2x_3\ldots x_i$ and $y_1y_2y_3\ldots y_j$

Edit distance variation

- In some edit distance definitions, substitution costs 2 operations. In that case, you need to use:

$$ed(i,j) = \begin{cases} ed(i-1, j-1) & \text{if } x_i = y_j \\ \min\big(ed(i-1,j)+1, & \text{if } x_i \neq y_j \\ \quad ed(i, j-1)+1, & \\ \quad ed(i-1, j-1)+2\big) & \end{cases}$$

DP for Edit distance

- Given the function, we can design a Dynamic Programming algorithm to find the distance in O(mn) where m and n are the sizes of the two strings.

- More here: http://en.wikipedia.org/wiki/Levenshtein_distance

```
int LevenshteinDistance(char s[1..m], char t[1..n])
{
  // for all i and j, d[i,j] will hold the Levenshtein distance between
  // the first i characters of s and the first j characters of t;
  // note that d has (m+1)x(n+1) values
  declare int d[0..m, 0..n]

  for i from 0 to m
    d[i, 0] := i  // the distance of any first string to an empty second string
  for j from 0 to n
    d[0, j] := j  // the distance of any second string to an empty first string

  for j from 1 to n
  {
    for i from 1 to m
    {
      if s[i] = t[j] then
        d[i, j] := d[i-1, j-1]       // no operation required
      else
        d[i, j] := minimum
                   (
                     d[i-1, j] + 1,  // a deletion
                     d[i, j-1] + 1,  // an insertion
                     d[i-1, j-1] + 1 // a substitution
                   )
    }
  }

  return d[m,n]
}
```

From wikipedia page

Metric properties and Indexing

- Edit distance is a metric! So, we can use metric indexes
- DTW is not a metric… we have to use specialized indexes

Links and References

https://en.wikipedia.org/wiki/Dynamic_time_warping

https://pypi.python.org/pypi/dtw/1.0

http://alexminnaar.com/time-series-classification-and-clustering-with-python.html

http://mlpy.sourceforge.net/docs/3.5/dtw.html

https://pypi.python.org/pypi/editdistance