Sub-time-series Clustering

Alexander Whitefield

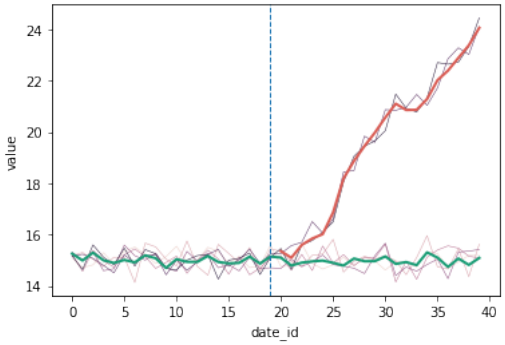
April 2022

Abstract

We present a model for *sub-time-series clustering* (stsc) and propose two greedy algorithms that attempt to approximate the model's objective function. We then conduct experiments to evaluate the model and the algorithms. Our main contributions are a new model for *sub-time-series clustering* and two greedy algorithms that efficiently obtain promising results.

1 Introduction

Sub-time-series clustering (stsc) attempts to find clusters of time-series at given intervals of time. Unlike time-series clustering, the clustering is allowed to change through time to better fit the data. An illustration a of a sub-time-series clustering is presented in Figure 1. Standard time-series clustering methods (such as K-means using a pairwise Euclidean distance metric) will cluster the time-series objects in the figure into two clusters if applied to the whole series. However, if a subset of the time periods are used (e.g $t < 25$), it is unlikely that standard methods will pick up on two distinct clusters. As *sub-time-series clustering* is more flexible than time-series clustering, it raises additional difficulties such as an increased risk of overfitting the data and increased computational complexity.



We restrict our focus to model free clustering methods that detect 'interesting' patterns in historical time-series data. As a result, our algorithms make minimal assumptions about the data generating processes. We view our approach as being complementary to the model based approaches discussed in section 3.

There are a number of motivations to sub-time-series clustering. First, historical time-series data contains patterns that are of interest in of themselves. Applying algorithms that allow for different patterns overtime can paint a richer picture of the nature of historical time-series compared to a whole time-series clustering. For example, identifying when a previously existing cluster broke apart is interesting in and of itself, and can spark further research into the qualitative events that may have occurred. Second, prediction for future relationships between time-series objects should be seen in context of the rise and fall of previous clusters. In many practical applications of clustering, the objective is to predict baskets of time-series objects that will behave similarly in the future. Observing the most recent clustering in a *sub-time-series clustering* may be able to identify clusterings that are relevant now, rather than over the course of a historical dataset. While standard time-series clustering methods can be applied to recent data (and exclude old data), it is not obvious how much old data should be excluded.

Throughout this study, we focus discussions on clustering data seen in social sciences, with a special focus on stock prices. The nature of time-series data in the social sciences is broad, ranging from high frequency and well recorded stock price data, to irregular, multidimensional household survey data. Clustering units is useful for a number of reasons including identifying heterogeneous effects of policies and partitioning units to groups to conduct separate qualitative and quantitative analysis for each. There is a substantive literature on the topics above, for example in Finance [20], Psychology [13] and Economics [16].

The paper is structured as follows: Section 2 specifies a model for *sub-time series clustering* and introduces notation. Section 3 provides an overview of related literature on clustering time-series related literature such as sub-trajectory clustering. Section 4 explains why *sub-time-series clustering* is hard. We then introduce some simple algorithms for *sub-time-series clustering* in section 5. We test our model and the proposed algorithms in section 6 using real and synthetic data. Section 7 concludes.

2 Model

In this section we describe our model for *sub-time-series clustering*. We first provide some definitions.

A time-series object is a sequence of points, each indexed by time. In this paper, we restrict our focus to time-series objects with one dimensional points. Therefore, when we refer to time-series object, we use the following defintion.

Definition (Time-series object)**.** *A time-series object x of length T is a series of points $p \in \mathbb{R}$ ordered by a time index i. We write this concisely as $x = \{p_1, p_2, \ldots, p_T\}$. x can be stored in a $T \times 1$ matrix.*

A time-series dataset usually contains multiple time-series objects. We restrict our focus to time-series datasets that contain time-series objects with identical time indexing: $t = (1, 2, \ldots, T)$. This is not prohibative in many time-series datasets in the social sciences. While missing data may occur, and sequences can start at different times, it is straightforward to extend our model to these situations. However, for simplicity, we leave generalisations to future research. As such, we define a balanced time-series dataset as follows.

Definition (Balanced time-series dataset)**.** *A balanced time-series dataset $\mathcal{X}$ of size n is a set of time-series objects with identical time indexing. We denote a balanced time-series dataset as $\mathcal{X} = \{x_j, \ldots, x_n\}$. As time-series objects are the same length T, we can store $\mathcal{X}$ in a $T \times n$ matrix.*

Next, we give a definition of time-series clustering.

Definition (Time-series clustering)**.** *Given a balanced time-series dataset $\mathcal{X}$, time-series clustering partitions the time-series objects into clusters $C = \{C_1, C_2, \ldots, C_k\}$. We define a clustering to be crisp if membership to a cluster is binary, and each time-series object is assigned to at most one cluster.*

Our definition for *stsc* uses the notation of a pathlet, introduced to the sub-trajectory clustering by Agarwal et al (2018) [1]. The authors define a pathlet P as a sequence of points that is not necessarily a subsequence of an input trajectory - for example a highway. In our case, a pathlet is a series of points indexed by time, that does not nessarility map onto a subsequence of time-series object, and covers a contiguous subset of the time index. We denote the time indicies pathet P exists over as $b(P)$. The set of all pathlets is denoted by $\mathcal{P}$

Definition (Sub-time-series clustering (stsc))**.** *Given a balanced time-series dataset $\mathcal{X}$, a sub-time-series clustering $\mathcal{S}$ allocates contiguous sequences of points in time-series objects to pathlets $P = \{P_1, P_2, \ldots, P_k\}$. If a point p_k in a time-series object x is in a sequence of points $x[a : b]$ that is allocated to a pathlet P_i, we say that that point p_k is allocated to pathlet P_i. We define a time-series clustering to be crisp if i) for each time-series object, each point in that time-series object is partitioned to at most one pathlet, and ii) allocation to pathlets is binary.*

We restrict our focus to crisp *sub-time-series clusterings*. When we refer to *sub-time-series clusterings* (or *stsc*), we refer to crisp *sub-time-series clusterings*.

2.1 Model for sub-time-series clustering

Our proposed model consists of five components, each contributing to a loss function $\mu(.)$.

The first, $|\mathcal{P}|$ is the total number of pathlets in $\mathcal{S}$. The second is the sum of unassigned points of $\mathcal{X}$ where $\tau(x)$ is the proportion of points in $x \in \mathcal{X}$ that are unassigned to a pathlet. While our model allows portions of time-series objects to be unassigned, we want to control this by making unassigned points costly. The third component, sums the distances of all points of the time-series objects in $\mathcal{X}$ to their assigned path S. Given a distance function d, the cost assigned to a point p is given by $d(p, P)$ where P_p is its assigned pathlet. If a point is unassinged, then it is not included in the sum. Next, our model allows time-series objects to switch from pathlet to pathlet through t. However, we want control the amount of switching so we impose a cost per switch, per time-series object. We denote $\psi(x)$ as the number of pathlets time-series object x passes through. Our fifth components seeks to control the number of pathlets at given point in time. While the first component controls for this, it will not incentivise removing pathlets that started early t, but at later points only contain only small number of time-series objects. We scale each component by user inputs that can be optimised. The model is presented below.

Definition (Sub-time-series clustering objective)**.** *Let $c = (c_1, c_2, c_3, c_4, c_5)$ be user defined parameters. Given a stsc $\mathcal{S}$, let the tuple $(\mathcal{X}, P, d, c)$ denote the balanced time-series dataset, the set of pathlets used in $\mathcal{S}$ and a distance function. The loss function is then given by*

$$\mu(\mathcal{X}, P, d, c) = c_1|P| + c_2 \sum_{x \in \mathcal{X}} \tau(x) + c_3 \sum_{x \in \mathcal{X}} \sum_{p \in P_p} d(x, P_p) + c_4 \sum_{x \in \mathcal{X}} \psi(x) + c_5 \sum_{t \in T} \sum_{P_i \in P} \mathbb{I}(t \in b(P_i)) \quad (1)$$

A key challenge in sub-time-series clustering overfitting the pathlets to the observed data. Our model attemps to control overfitting, while also allowing flexibility in clustering overtime. The number of pathlets is not fixed, but too a larger number of pathlets incurs a larger cost in the first term (if $c_1 > 0$[1]). Each time-series object, does not need to be mapped to a pathlet in all (or any) index of time, however not being assigned incurs a loss in the second term. The third term is standard for incentivising good clustering. The first three terms, closely follow Agarwal et al [1] objective function for sub-trajectory clustering. The fourth term allows time-series objects to switch clusters (by switching pathlets), or join new clustering (joining a newly formed pathlets). The final term is introduced to prevent lingering pathlets [2]. Given the flexibility in *stsc*, the first, fourth and fifth term play key roles in regularising the fit to the data.

3 Prior work

Sub-time-series clustering borrows ideas from two branches of literature: i) time-series clustering and ii) subtrajectory clustering. In section we briefly outline the ideas that are most related to *stsc*.

For a detailed overview of timeseries clustering, see [2]. For a survey on trajectory clustering (including sub-trajectory) see [48].

3.1 Time-series clustering

Aghabozorgi *et al.* (2015) [2] group of time-series clustering methods into three buckets: i) whole time-series clustering, ii) sub-sequence time-series clustering and iii) time point clustering. The first is the most relevant to *stsc*. The second, sub-sequence time-series clustering seeks to cluster a set of subsequences of a single time-series through a sliding window [50]. The third clusters time points, rather than time-series objects.

Within the category of whole time-series clustering, we identify four key steps in a clustering: 1) pre-processing the time-series data, or data representation 2) computing a distances between time-series objects 3) clustering time-series objects 4) evaluating the quality of clustering. Note that each of the first three steps rely on their preceding step. In the following subsections, we provide brief overviews on the state of the literature regarding the four steps.

[1] While values in c can take negative values, we expect them to positive almost all use cases.

[2] The other terms in the objective function will not penalise accumulating pathlets. For example, they will not penalise maintaining a cluster that was originally relevant, but only covers a few time series objects at later time periods. Note that the first term only penalises the number of pathlets, not their size.

3.1.1 Data representation

A time-series object of length T in a balance time-series dataset can be viewed as a T dimensional vector. To compare the similarity of two time-series objects we could apply a given distance metric (e.g. Euclidean, L_∞ metric). Indeed this is common approach. However, the distance between raw vectors may not be representative of the time-series objects. For example, consider the stock prices of Pepsi, Coca Cola and Apple (tickers: PEP, KO, AAPL respectively) plotted in Figure 1 from January to March 2022. Intuitively, we would expect Pepsi and Coca Cola to be similar, and hence have a small distance between then. However, calculating, say the Euclidean distance, between the three series will suggest that Apple is much 'closer' to Pepsi, than Coca Cola is. This example is trivial, as the level of a stock price at a given point in time carries little meaning. Note that this is unlike many types of time-series [3] Correlation in the changes of the stock prices are likely to carry much more meaning about the relationship between the stock prices.

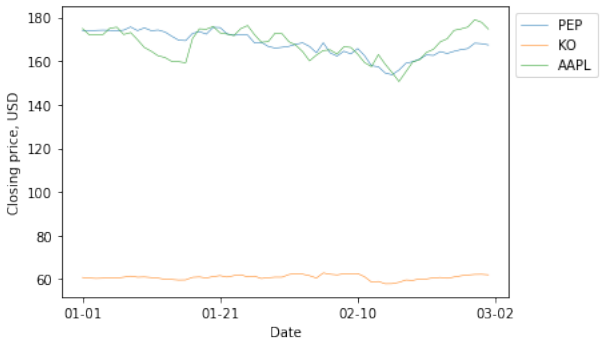


Figure 1: Raw stock prices for Apple (AAPL), Coca Cola (KO), Pepsi (PEP)

There are multiple commonly used normalisations. For example, Z score ($x = \frac{x-\bar{x}}{std(x)}$) [40] min-max ($x = \frac{x-\min x}{\max x - \min x}$) and unit length normalisation ($x = \frac{x}{||x||}$). Other normalisations include Mean normalization, Median normalization, Adaptive scaling [11], Logistic or sigmoid normalization and Hyperbolic tangent normalization [45]. Note that out of the datasets in the UCR archive, a repository of time-series datasets used to evaluate time-series cluster, around 80% are z-score normalized [24]. Paparrizos et al. (2020) [45] compare eight normalisation methods across a range of distance metrics. They claim to de-bunk a myth that z-scores are the best normalisation method by showing that when combined with some distance measures, other normalisations can perform better. A reasonable takeaway from this section is that normalisations should chosen carefully, and simply taking a z-score may not be sufficient.

Indeed, for the application of clustering stock prices, there are multiple specific normalisation to consider. A standard approach in the finance literature is to compute log daily returns: $r_{it} = 100 \times log(P_{i,t}/P_{i,t1})$ [12].

An alternative or complementary approach to normalising raw data is to reduce the dimension of the a time-series object, thereby reducing noise and extracting key features. Examples include Piecewise Aggregate Approximation (PAA) [29], Discrete Wavelet Transform (DWT) and Discrete Fourier Transformations (DFT) [3]. These methods come with the added benefit of reduced computational burden.

For our experiments, we use two normalisations i) z-score normalisation, and ii) abnormal daily returns. We expect the latter to be more useful, but include the former as it is the most commonly used normalisation method for time-series clustering in general. For the example, of Pepsi, Coca Cola and Apple, we present the z-score normalised stock prices (from 2010 to 2022) [4] and daily abnormal returns for the same time period as Figure 1 in Figure 2. With both pre-processing methods, Pepsi and Coke are closer to each other than Apple - in line within their known similarity. Note that proposed algorithms run on pre-processed data, and can work with any normalisations. Rigorously comparing our algorithms across many different pre-processing steps is beyond the scope of this study.

[3] This is not the case in other asset markets such as bond or foreign exchange market.

[4] As the normalised uses the mean from 2010 to 2022, the y axis is not centered at 0

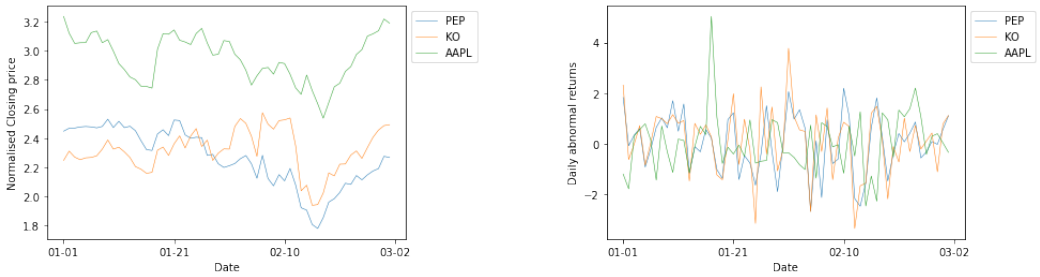


Figure 2: Comparison of z-score (left) and daily abnormal returns (right)

3.1.2 Similarity measures

The objective of clustering is to identify structure in an unlabeled dataset by grouping together objects in clusters such that objects in the same cluster are similar, and objects in different clusters are different [35]. Implicit in this definition is a notion of similarity, which is non-obvious when objects are time-series. We restrict our attention to clustering time-series that are *similar in time* [2] (also referred to as being *in lockstep* [45]). For example, stock price data is largely recorded at regular time intervals[5]. When considering distance between time-series objects that are *similar in time*, two individual time-series can be viewed as a T dimensional vector. Given the number of ways to measure differences between two vectors, there are huge number of candidate distance measures for time-series. As well as normalisation methods, Paparrizos et al. (2020) [45] compare 52 *lock-step* distance measures for time-series. We refer readers to this review when choosing a reasonable distance metric between time-series vectors

Alternatively, distance measures that were orginally devised for time-series that are not *similar in time* can also be applied to series that are *similar in time*. Commonly used methods include longest common subsquence (LCSS) [8, 18], dynamic time warping (DTW) [41], Hausdorff distance [7], and modified Hausdorff distance [2]. We can therefore easily extend *stsc* to time-series objects of differing length using these distance measures.

3.1.3 Clustering methods

There are two intrinsic difficulties to all clustering methods. First, while useful heuristics exist (see 3.1.4), it is often hard to quantify to goodness of a clustering. An objective function can be helpful in evaluating a clustering, and it also required for many greedy clustering algorithms. Second, given an objective function, clustering is almost always *NP-hard* [38].

A clustering is considered crisp if clusters are non-overlapping, and fuzzy otherwise [35]. As described in section 2 we focus on crisp clustering methods in this paper.

As with static clustering, categories of time-series clustering algorithms include partitioning, hierarchical, grid-based, model-based, and density-based clustering algorithms [39]. There are also approaches, sometimes referred to as multi-step algorithms that combine these approaches [2]. Another notable category are randomised algorithms that optimise the objective on a subset of the data [22, 23].

Partitioning methods require a user defined number of partitions k. Each of the n objects are assigned to a partition, and it is often required that all partitions contain at least one object. Two of the most common partitioning algorithms: *k-means* [36] and *k-centers* [21], are both greedy. Given a distance measure [6], these algorithms run for time-series data as they do for static data. *k-means*, also called Lloyd's algorithm, first randomly selects *centroids*: k points (in $n-$dimensional space, e.g. a time-series object). It then assigns datapoints to their nearest centroid. Centroids are then recalculated based on their assignments so that they best fit the datapoints assigned to them. Then points are again re-assigned to their nearest centroid. The process continues until a maximum iteration limit is reached, or there is a round where no datapoints are re-assigned. While sometimes effective, there a no theoretic bounds on the quality of the k-means solution,

[5]There are some exceptions, such as when companies first enter the stock market in an initial public offering, when a regulator suspends trading a certain stock or when comparing stocks across exchanges that observe different holidays

[6]Including distance measures for time-series objects with varying time indices

and in many situations it performs poorly. It is also sensitive to the choice of initial centroids. *K-centers* first random selects random point in the data as centroid. It then selects the point in the data that is furthest from the centroids already selected. This process continues until K centroids have been chosen. It turns out that *K-centers* is a 2-approximation to the optimal k-center clustering [21]. These algorithms are often combined in an algorithm named *k-means++* [5]. Due to its simplicity and efficiency, we use k-means++ in our proposed algorithms. There are interesting extensions to partition based clustering that impose further constraints on the nature of each partition [37]. While partitioning methods require a user to select k, it is straightforward to optimise k if the user has a loss function (such the one provided in section 2). This is the basis of our first naive algorithm.

Hierarchical clustering creates a hierarchy of clustering using either agglomerative or divisive steps [46]. Agglomerative algorithms start with each datapoint as its own cluster, and then gradually merges them. These methods are closely related to Kruskal's algorithm for finding minimum spanning trees. Divisive algorithms start with one large cluster, and then gradually split clusters into smaller clusters. Both methods create a hierarchy of clusters. A benefit of hierarchical methods is that they do not require the user to specify number of clusters as the hierarchy constructed can inform the cluster choice [31]. However, it is often infeasible with large datasets it requires computing all pairwise distances between time-series objects in $\mathcal{O}(n^2)$. For large n these methods are often infeasible. That said, they are closely linked to the idea of evolving clusters as they contain information in their structure as to when the number of clusters changes [6].

Density-based clustering, such as OPTICS [4] and DBSCAN [14] seek to cluster together points that are connected by high density regions. **Grid-based methods** such as [49] discretise the space into a grid and perform clustering on the grid cells. **Model-based** clustering methods impose a structure on the processes that generate the observed data and select clusters that best fit the model. Close to the motivation of our paper are hidden Markov models (e.g. [42, 12]) that allow for changes in underlying data generating processes over time. For example, [12] propose a regime-switching model (hidden-Markov model) for stock markets (whole markets, not the stocks within them) that model latent states of the world that oscillate over time. Another popular method in the data mining community is Self Organizing Maps (SOM) [30] that use neural networks to create low dimensional representations of the data. SOM does not work well with time-series of unequal length [35], but this does not exclude its use for time-series that are *similar in time*. Gaussian mixture models are also a popular approach to cluster time series [23].

3.1.4 Assessing the quality of clustering

By the nature of clustering being unsupervised, it is non-trivial how the quality of clustering should be evaluated. We discuss four types of methods.

First, some authors compare the clustering to expert picked categories that are considered to be the 'ground truth' [17]. For the stock market example, a natural 'ground truth' is expert picked industry groups. There are various measures that place a score on difference between two clustering such as the Rand index and Modified Rand Index, F-measure, Cluster Similarity Measure (CSM) and Jaccard index. These measures are not just used for evaluating clustering, and we draw on the Jaccard index in our greedy algorithm as part of the decision as to merge pathlets. Overall, while these measures compared to ground truth can be credible, they rely on us having group truth - which is rarely the case if we are looking for the ground truth. Moreoever, expert picks may not be the ground truth as authors such [43] observe.

A second method to evaluate clustering is out-of-sample predictive power. The key idea is that if the clustering is good, it should be valid on realisations of data other than the data it is trained on. For example, a common use of clustering stocks is to assist investors in managing risk. Therefore, a natural measure is out-of-sample forecast comparisons, for example see [43].

Thirdly, there are a number of metrics that measure the quality of clustering on the training data alone. These are also called internal indices [2]. Common measures include Sum of Squared Error and the Silhouette index [47]. In our model, we use the sum of all points to their nearest pathlets as an interval quality of clustering method.

Finally, in many datasets, visualisation is helpful in ascertaining if a clustering is reasonable. Of course, visualisation is not always scalable.

Measuring the quality of a *sub-time-series clustering* can draw of each of these methods. Indeed our

model setup draws on the third. In our experimental section on synthetic data, we draw on the first and last. We leave the second for future research in relation to *stsc* clustering.

3.2 Sub-trajectory clustering and other approaches

While clustering algorithms can detect time-series objects that are similar across the whole time-series, they are not always well suited to to detect objects the moved in a similar way for some portion of time. This is especially relevant in the field of trajectory clustering [32]. For example objects such as cars have common subtrajectories. To identify common subroutes, where a car starts or ends its whole journey may not be relevant to detect major traffic flows. Identifying these subtractories can be used to detect common routes or even detect roads if maps are unavailable. Another application of subtrajectory clustering is to detect hurricanes, that only follow common routes towards the end of their lifetimes [9].

A related concept is a pathlet: a portion of a path that tends to be traversed as a whole by many trajectories [1]. Pathlets have been used to impute missing data, reduce noise and identify outliers [33, 34]. As discussed in section 2, we lean on pathlets in our model.

The focus of this paper is mining patterns in time-series data that *differ in time*, and accordingly, the motivations for subtrajectory algorithms are not immediately relevant for time-series data with a one dimensional outcome. However, we can transform time-series data into a two dimensional trajectory by setting time as a dimension. This reduces the sub-time-series clustering problem to a subtrajectory clustering problem. Running subtrajectory clustering algorithms may be able to answer questions such as what period of time did the relationship between different time-series objects form or breakdown. However, as *stsc* has a specific structure, more specialised methods may be more efficient.

An alternative approach is to partition each time-series by time, and run a separate time-series clustering on each time period. This has a number of drawbacks, including being sensitive to the time window selected. If we are only interested in one period of time, we could apply existing methods to the window of time we are interested and prune the rest of the data. However this requires that we know the window of time we are interested in, and does not facilitate automated pattern mining. This is motivation therefore similar to subtrajectory clustering [32].

3.3 Other methods

The are a number of other methods related to *sub-time-series clustering* that are also relevant to *sub-time-series clustering*. We give brief overviews of ideas from these methods.

First there are methods that seek to locate objects that move together over for a subset of the time index of a dataset. A **flock** is defined as a group of objects that move together within a circular disk, define by a parameter value [19]. A related idea is a *moving cluster* that must maintain a certain amount of objects through its lifetime [27]. Note that a moving cluster allows the objects within it to be completely different at either ends of its lifetime. Finally, a **convoy** is a group of objects that are connected by other objects through its lifetime. Convoy clustering, first proposed by [25], seeks to cluster points that are 'density connected': where a given a set of points S, a point $p \in S$ is density–connected to a point $q \in S$, with respect to a distance threshold $e-$ and an integer m determining the number of connects, if there exists a point $x \in S$ such that both p and q are density–reachable from x [26]. Our proposed greedy algorithms allow for time-series objects to hop from pathlet to pathlet, and we allow the constituents of pathlets to change. Hence our notion of a pathlet is related to a moving cluster. Time-series-objects may be related over the course of a long period of time, but may deviate from their cluster in given periods. While we do not directly use methods from identifying flocks and convoy clustering in this study, applying these methods to time-series objects may be a direction for future work.

Semantic based trajectory data mining incorporates domain-specific information in the trajectories in a pre-processing step, so that information such as important places, stops and moves etc. can be extracted from the trajectories by applying data mining algorithms. For example [28] use an incremental algorithm to extracting significant places from a trace of coordinates. Palma *et al.* (2008) add semantics to trajectory based on speed [44]. Our algorithms can easily be extended to include semantic information when picking partitioning windows.

There are a number model based approach used in Economics and Finance to detect underlying shifts in patterns. Casini and Perron (2018) [10] provide an overview of structural break models that can complement our approach by suggesting windows for segmentation. As mentioned previously, the aim of the study is to complement model based approaches with almost completely model free methods to identify patterns. Therefore, the core of our algorithms do not draw from these methods.

The most similar study to ours in aim is [6] that seeks to identify evolving clusters in a financial time-series. They propose the PETRA Method that uses hierarchical clustering (single-linkage) combined with a dynamic tree cut (DTC) to first find a suitable number of clusters. They iterate the algorithm through sliding windows, and at each step, calculate if the difference in clustering is 'significant'. The output of their model is a notification of when a significant change occurs. Our first greedy algorithm (greedy1) is similar in that it measures the similarity of clusterings across time periods. However, our approach differs in the following ways. First, our goal is broader in that we want to describe the history of clustering over a time-series dataset. Our method generalises to multiple types of data, not just financial data. Second, we rely on partition based clustering methods rather than hierarchical methods. This is so that our method can scale to datasets with a large number of time-series objects.

4 Hardness of sub-time-series clustering

It is not hard to see that pathlet cover is at least as hard as sub-time-series clustering. This is because sub-time-series clustering is a special case of pathlet cover. Specifically, we can reduce sub-time-series clustering to pathlet cover by converting a one dimensional time-series, into a 2 dimensional trajectory, with time as the additional dimension. If we restrict distance measures to *similar in time* distance measures, then we can solve the instance of sub-time-series clustering with an algorithm for subtrajectory clustering. Note, we cannot make the opposite reduction as sub-time-series clustering cannot take on distance measures that are not similar in time. Solving subtrajectory clustering (with constant dimension) with a specialised algorithm for sub-time-series clustering would require all trajectories are the same length.

On top of the computational hardness associated with clustering methods (in more than one dimension), the source of hardness of *stsc* is that there are an infinite number of potential pathlet combinations, and a huge number of reasonable pathlet combinations.

5 New Algorithms

In this section we propose new algorithms for sub-time-series clustering. First we propose some naive routines that will provide helpful benchmarks. They are also be used in the main greedy algorithms. We then propose our main greedy algorithms: greedy1 and greedy2.

5.1 Naive 1

Our first naive algorithm clusters the whole time-series. As this imposes an additional constraint on the clustering, it is clearly not generally optimal for our objective function. However, it is useful to keep as a base case. The algorithm will perform k-means++ on various values of k, and pick the value that minimises the objective function. We provide pseudo-code in Algorithm 1.

5.2 Naive 2

The motivation for our second algorithm is that clusterings may substantively change over time. Perhaps the simplest way to account for this is to split up the time indices into intervals, and compute an independent clustering on each. This is exactly what our second algorithm does: it first partitions T into a set of I intervals. It then applies naive1 to each interval, using the clusters in each problem as pathlets for the full solution. As it is unclear what the optimal set of intervals is, it can test out various sets of intervals and return the best solution. Pseudocode is is presented in Algorithm 2.

If one of the intervals entered into I includes $t = (1, \ldots, T)$, then naive2 will do strictly better than naive1.

Algorithm 1 naive1($\mathcal{X}$,d,c,K)

Input: $\mathcal{X}$, d: a distance function, c: vector containing model parameters: $c1, \ldots, c5$, K: a list of cluster numbers
Output: $\mathcal{S}$: a *stsc* object

```
l ← inf
S ← ∅
for k ∈ K do
    s ← Kmeans++(X, k, d)
    if loss(s) < l then
        S ← s
        l ← μ(X, s.P, d, c)        ▷ Loss applies the loss function as specified by the model
    end if
end for
return S
```

Algorithm 2 naive2($\mathcal{X}$,d,c,K,$\mathcal{I}$,$GP = False$)

Input: $\mathcal{X}$,d,c,K defined in naive1. $\mathcal{I}$: candidate sets of intervals
Output: $\mathcal{S}$: a *stsc* object

```
l ← inf
S ← ∅
for I ∈ I do
    res ← ∅
    for i ∈ I do                   ▷ Each i is an interval of the form (t_start, t_end)
        res.append(naive1(X[i],d,c,K))   ▷ X[i] denotes the subset of the panel dataset that lies in i
        count += 1
    end for
    s ← comb(res)                  ▷ comb(.) combines converts clusters in multiple intervals into pathlets
    if μ(X, s.P, d, c) < l then    ▷ c is augmented if GP is set to true
        S ← s
        l ← μ(X, s.P, d, c)        ▷ c is augmented if GP is set to true
    end if
end for
return S
```

For the greedy algorithms, we add an argument: GP=True that modifies the loss function called at the end of each iteration of $\mathcal{I}$. If this is turned on, the loss function that compares across iterations sets c_1, c_4 and c_5 to zero. A a result, calling naive2 will return the optimal clustering within each segment as if that segment was the whole series.

5.3 Naive 3

Naive 3 takes in an *stsc* and greedily drops the sub-time-series sub-series if the benefit outweights the cost. It can be applied after any of the algorithms described, such as naive1 or naive2. Note that naive3 will always do at least as well as the *stsc* entered into it. We do not use this algorithm in our experiments, however we mention it to demonstrate how the clustering can be locally optimised post-hoc.

We next present two greedy algorithms that start with or iterate across version of naive2, and then take greedy steps to improve the clustering

5.4 Greedy 1

Recall the objective function in equation 1.

$$\mu(\mathcal{X}, P, d, c) = c_1|P| + c_2 \sum_{x \in \mathcal{X}} \tau(x) + c_3 \sum_{x \in \mathcal{X}} \sum_{p \in P_p} d(x, P_p) + c_4 \sum_{x \in \mathcal{X}} \psi(x) + c_5 \sum_{t \in T} \sum_{P_i \in P} \mathbb{I}(t \in b(P_i))$$

Compared to naive1, given a single input of K and single set of intervals $\mathcal{I}$, naive2 reduces the third component of the objective function, at the expense of the first component (number of paths) and fourth component (number of path switches). Both our greedy algorithms attempt to take a solution that leads to a low third component, and then merge paths and augment pathlet assignments to reduce the first and fourth component. The second greedy algorithm also attempts to reduce the fifth component.

Making greedy moves to improve the solution given by naive2 is non-trivial. We propose two approaches in greedy1 and greedy2.

The essence of Greedy1 is to take a result from naive2, and then seek to improve objective function by merging pathlets that are similar and thereby decreasing $|P|$, without substantially increase the distances of points to their assigned pathlet. Moreoever, it allows time-series objects to switch pathlets if the benefit (reducing component 3: $d(.)$) outweights the cost (component 4: $\psi(.)$). It takes advantage specific structure of $\mathcal{X}$ by starting at the leftmost intervals and scanning left to right.

To simplify the algorithm, it calls on naive2 with a single k as an input. Therefore, the *stsc* object it works off has an equal number of clusters in each interval. Psuedo code is provided in Algorithm 3.

The notation *s.i.leftpaths* and *s.i.rightpaths* respectively refer to the paths that start and end at the end of the interval i. For example, suppose $i = (5, \ldots, 10)$. *s.i.leftpaths* are the pathlets in s that end at time index 10. *s.i.rightpaths* are the pathlets that start at time index 10.

The function J(pth,j) computes the Jaccard similarly [7] between the time-series objects assigned to *pth* and j. If the members are identical, J(pth,j) = 1, and if they have no overlap, J(pth,j=0).

The *cont()* method extends a left path into the next period, and removes its matched pathlet in the right period. The method *greedilyAssign[i]* considers all the time-series objects in $\mathcal{X}$ in the next period (also referred to as period j). It assigns greedy as follows: calculate the cost associated with moving to each path. If the path is the same as their previous path, calculate the distance for period j. If the path is different, add the switching cost: 1*c_4 and the distance for period j. [8]. Then assign the time-series object to the path with the lowest cost.

While greedy1 takes single inputs for I and k, we can iterate it over multiple values of k and multiple sets of intervals.

greedy1 has a number of limitations. The major limitation is that while it allows clusters to change over time, it does not allow the number of clusters to change over time. We can iterate the whole algorithm across different values of k, but for each iteration, the number of clusters is fixed. Therefore, it cannot pickup on

[7] Jaccard similarity between sets A and B is calculate as $J(A, B) = \frac{|A \cap B|}{|A \cup B|}$

[8] As membership may change, the pathlets in period j can then recalculate their centroid according to their new assignments. However, we leave aside this feature for simplicity.

Algorithm 3 greedy1($\mathcal{X}$,d,c,k,I)

Input: $\mathcal{X}$,d,c,k: single number. I: candidate interval set
Output: $\mathcal{S}$: a *stsc* object

```
c' ← (0, c_2, c_3, 0, 0)                          ▷ set c_1, c_4, c_5 to zero to get best fit
s ← naive2(X,d,c',k,I)
for i in I do                                      ▷ intervals are ordered left to right
    left ← s.i.leftpaths                           ▷ at the end of i, obtain paths that end at the left
    right ← s.i.rightpaths
    cont_paths ← ∅
    sort(left)                                     ▷ Sort by the Jaccard score of their best match
    for pth in left do
        match ← arg max_{j∈right} (J(pth,j))       ▷ J is the Jaccard index
        if J(pth,match) > ρ then
            cont_paths.append((pth,match))         ▷ a tuple is added each successful iteration
            right.remove(match)
        end if
    end for
    s.cont(cont_paths)   ▷ .cont() extends paths into the next interval and removes their match
    s.greedilyAssign[i]                            ▷ Greedily assign time-series in the next period
end for
return s
```

the phenomena such as the number of clusters increasing or decreasing over time. Another limitation is that the merging cluster step is not sensitive to the loss function. In fact, its sensitivity to c_1 is purely covered by the optimal choice of k and I. It is sensitive to c_4 in the .greedilyAssign method. A minor limitation is that it introduces a new parameter ρ: when the merge clusters, that requires optimisation.

5.4.1 Runtime of greedy1

When implementing the algorithm, at each interval we compute a matrix between the left and right paths for the Jaccard index between them. This takes $\mathcal{O}(k^2)$. It takes $\mathcal{O}(nTk)$ to run .greedilyAssign, and there are at most $|I|$ iterations after naive2 is run. Naive2 runs K-means++ on $|I|$ segments and K-means++ is $\mathcal{O}(nkT)$. Naive2 therefore takes $\mathcal{O}(|I|nkT)$. The worst case runtime is therefore $\mathcal{O}(|I|k^2nT)$

5.5 Greedy 2

While similar in spirit to greedy1 (scanning through the naive2 result from left to right and making greedy steps), greedy2 is fairly different in how it makes greedy steps. At a high level, at the end of each interval $i \in I$: greedy2 projects the left pathlets into the interval to the right of i. It then ranks the cost of the right pathlets and the extrapolated left pathlets, and chooses the best pathlets. The cost of each pathlet depend on the model parameters $c = (c_1, \ldots, c_5)$. Right pathlets are punished as they increase $|P|$. Pathlets that cause lots of time-series objects to switch paths are costly as they increase the fourth component of the objective. Extrapolated left pathlets that cover only a few time-series objects are punished by the fifth term in the objective function. Pathlets that badly represent their members are punished by the third term in the objective function ($d()$ and c_3). We present pseudo-code for the algorithm in Algorithm 4. Most of the work is done by the s.merge() method that we detail further below.

The merge function has three steps: 1) extrapolate the left paths, 2) assign a score to all the extrapolated left paths 3) out of all the extrapolated left paths and right paths, pick the best. These three steps are described in detail below.

1. Extrapolate the left paths: For each left path: *pth* (pathlets that ends at the end of interval i), store the time-series objects that are assigned to *pth* in period i. We refer to these time-series objects as member of *pth*, and we refer to the period the right of i as period j. Calculate the point series in j that minimises the metric d applied to members if they were assigned to that point series in j. For

Algorithm 4 greedy2($\mathcal{X}$,d,c,K,I)

Input: $\mathcal{X}$,d,c,K,I
Output: $\mathcal{S}$: a *stsc* object

```
c' ← (c_1, c_2, c_3, 0, 0)
s ← naive2(X,d,c',K,I,GP=True)        ▷ may be different number of clusters across intervals
for i in I do
    left ← s.i.leftpaths               ▷ at the end of i, obtain paths that end at the left
    right ← s.i.rightpaths
    s.merge(left,right)
    s.greedilyAssign[i]                ▷ as described in greedy1
end for
return s
```

example, if we use the euclidean distance metric, calculate the mean path of members j. This point series, combined with assignments to its members, is the extrapolated left path.

2. Assign scores to candidate paths: There are two types of candidates paths i) extrapolated left paths ii) right paths i.e. the optimal clustering according to naive1 if we only considered period j. The cost for right paths include the cost of creating a new path, and causing time-series objects to switch paths from i to j. Both type of path cost include the cost from the third component (distance of member to its assigned pathlet centroid). Both path costs also include the fifth component. The costs are divided by the total number of time-series objects assigned to it. We then rank all candidate pathlets by score. Note that well fitting extrapolated left pathlets will always rank highly. Well fitting right pathlets will rank highly if they are a good fit (i.e. cover lots of points at a low cost) compared to extrapolated left paths, and this is enough to overcome to cost of creating a new path.

3. First we unassign all the right paths (as they are already given assignments before iteration i by naive2). We then select the top ranked paths. However it is not straightforward how to do this - for example, the top two paths may be a left and right path that cover exactly the same objects. Therefore, in order of rank, we iterate through each of the candidate paths, and add that path if the fraction of their members that overlap with already assigned members for period j is less than a function $\delta(|left|, |right|)$. We stop assigning new clusters until one of following three conditions becomes true: i) total paths assigned to period j is equal to the max(|left|,|right|) ii) number of extrapolated left paths assigned = |left| iii) number of right paths assigned = |right|.

Like greedy1, greedy2 introduces an additional parameter δ that requires tuning. Both greedy1 and greedy2 require a user to input a set of K. Iterating over a large set of K is computationally expensive. Furthermore, I needs to be selected in both greedy1 and greedy2. While this is straightforward to optimise, it is computationally expensive. In the next section we describe how domain specific knowledge can be use to accelerate the search for the optimal I.

The limitations of both greedy1 and greedy2 are areas for future research.

5.5.1 Runtime of greedy2

On top of naive2, greedy2 iterates through $|I|$ steps. In each step it runs .greedilyAssign in $\mathcal{O}(nTk)$ and .merge() in $\mathcal{O}(nTk)$. Therefore, total runtime is $\mathcal{O}(|I|nTk)$, the same as naive2 [9].

5.6 Semantic modifications

So far, we have not specified how sets of k and $\mathcal{I}$ are selected. Domain specific knowledge is likely to be useful in informing these decisions. For example, in the context of stock prices, clusters are likely to change at periods of high volatility. A reasonable approach to choosing candidates for $\mathcal{I}$ is to record volatility of stocks (using the input data), and select the y most volatile periods (that are not too close together).

[9] assuming a single entry of k

6 Experiments

We conduct two sets of experiments. First we evaluate the algorithms on synthetic datasets, each of which with specific features that our algorithm should detect. We then apply our algorithm to stock price data for over 400 stocks [10] from the *S&P500* from 2010 to 2022.

In all simulations we set $d(.)$ to $L1$ distance. We set $\rho = 0.6$ for greedy1, and $\delta(x) = \frac{1}{x}$ in greedy2. Moreover, in some experiments we loop the greedy algorithms through different sets of $\mathcal{I}$ (and k for greedy2) and choose the best solution according the objective function. Implementations of our algorithms are available here.

6.1 Synthetic data experiments

To better understand the nature of our algorithms, we conduct three sets of experiments on synthetic datasets. The first experiments demonstrate that with the correct parameterisation, our greedy algorithms can achieve parsimonious explanations of the data. The purpose of the first experiment is to demonstrate how the algorithms work, rather than rigorously evaluate their performance. The third dataset demonstrates the sensitivity of our algorithms to choices of c_1.

6.1.1 Experiment 1: Change in cluster composition

A feature of our algorithm is that it is designed to detect shifts in clustering over time. We simulate the following timeseries dataset that mimick a shift in clustering.

- $T = 80$, $N = 30$
- 2 data-generating processes at given point in time: N(0,1), N(5,1).
- 4 almost evenly sized groups (eight in the first two, seven in the second two)
    1. $N(0, 1)$ for first 80 periods
    2. $N(0, 1)$ for first 60 periods, then $N(5, 1)$ for 20 periods
    3. $N(5, 1)$ for first 60 periods, then $N(0, 1)$ for 20 periods
    4. $N(5, 1)$ for first 60 periods, then $N(5, 1)$ for 20 periods

The data is plotted in Figure 3.

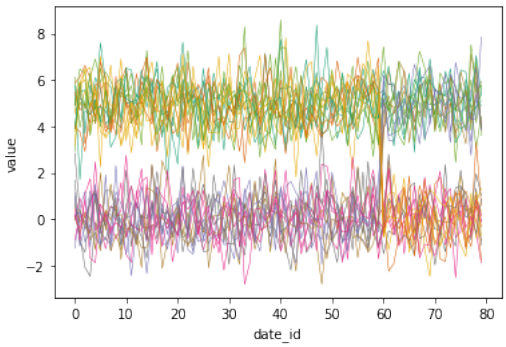


Figure 3: Simulated shift in cluster composition

We set $K = \{2\}, I = \{(1, 20), (21, 40), (41, 60), (61, 80)\}$, and C = $(c_1 = 10, c_2 = 1, c_3 = 1, c_4 = 1, c_5 = 1)$. We then run naive1, naive2, greedy1 and greedy2. The purpose of setting $k = 2$ is that we force the clusterings to have at most two clusters at a given point in time.

We plot the pathlets in from each clustering Figure 4 where the solid lines represent pathlets, and the faded gray lines represent raw time-series objects.

[10] We take stocks that are in the $S\&P500$ today, and omit those who do not have stock prices back to 2010

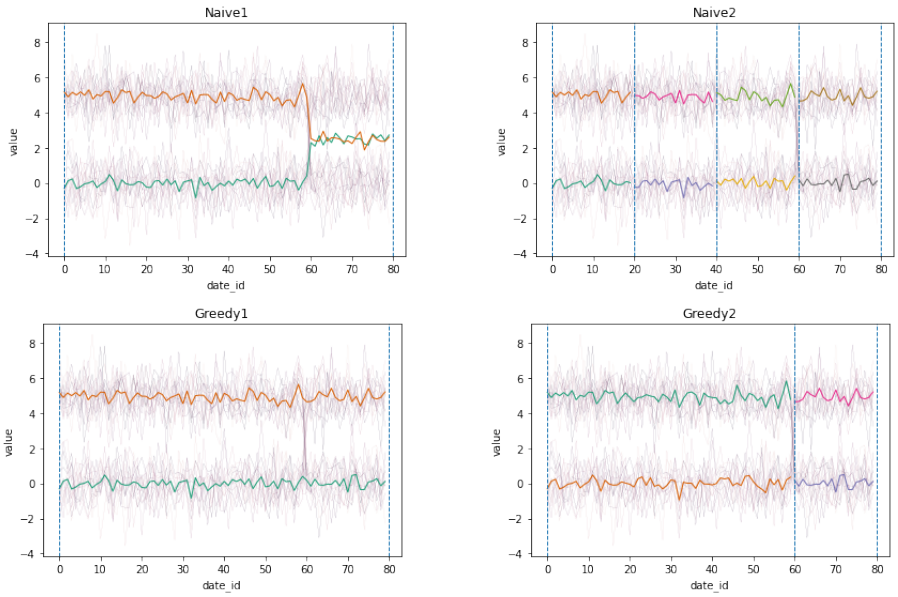


Figure 4: Algorithm comparison

Naive 1 (essential Kmeans++) could perfectly cluster the data with k=4. However we add the constraint to demonstrate how our greedy algorithms can achieve a good fit even when the number of clusters in each period in constrained to 2. In this case, naive1 is unable to explain the data well in the final 20 time periods. The other algorithms have equal distance costs in the objective function (i.e they almost perfectly fit the data) but differ in the quality of data representation they provide. naive2 achieves a well fitted clustering, but it does not link up pathlets in the first 3 intervals (from $t = 0$ to 60) and hence cannot identify a shared structure across periods. greedy1 maintains two pathlets throughout the time periods. It achieves an almost perfect fit by allowing time-series objects to switch clusters (15 objects—half the time-series objects in the sample—switch cluster at t=60). Finally greedy2 generates two new pathlets at (t=60). In terms of distance of pathlet centroids to points, naive2, greedy1 and greedy2 all do almost perfectly. However greedy1 and greedy2 are more parsimonious in data description. Moreoever, stored in the data structures generated by running greedy1 and greedy2, is useful information about how the clusterings change.

6.1.2 Experiment 2: Change in number of clusters

Next we simulate data where the number of clusters, and the constituents of clusters change over time. We generate data coming from one of 6 data generating processes—each with mean at a given point in time following Figure 6. We generate data for 30 observations, 5 from each data generating process, and add iid noise from $N(0, 1)$ to each point in each series. One could argue that there are 6 clusters, and that their nature does change over time. However, the predominant aim of clustering is to generate a simplified representation of the data. In this simple example, the goal is to cluster points in under 4 clusters at a given period of time.

We set $K = \{1, 2, 3, 4\}, I = \{(1, 20), (21, 40), (41, 60), (61, 80)\}$, and C $= (c_1 = 10, c_2 = 1, c_3 = 1, c_4 = 1, c_5 = 1)$. We then run naive1, naive2, greedy1 and greedy2. Results are presented in Figure 6.

As we only allow a maximum 4 clusters at a given time interval, naive1 could not fit the data well for the first 20 periods. The other algorithms had pathlets that covered all the observable paths. Naive2 shows the optimal clustering in each time interval (the third segments only contain one center as more centers incurs cost in the objective function). greedy1 and greedy2 are more parsimonious, but in different ways. greedy1 maintains 3 pathlets and allows time-series objects to hop from cluster to cluster. In total there are 38 pathlet switches in greedy1 compared to 90 for naive2. Recall that N=30, which implies that some time-series objects switched clusters more than once. greedy2 maintains 5 pathlets, but allows fewer switches across pathlets (30 in total - specifically, each time-series object moved cluster once when they moved to

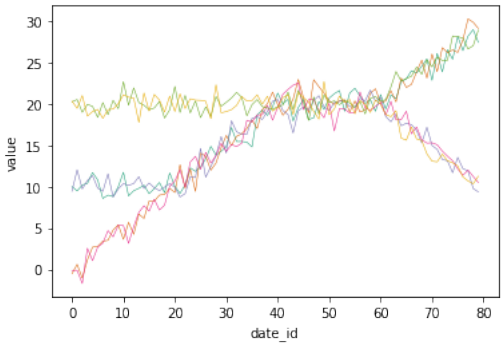


Figure 5: Means for 6 generated clusters

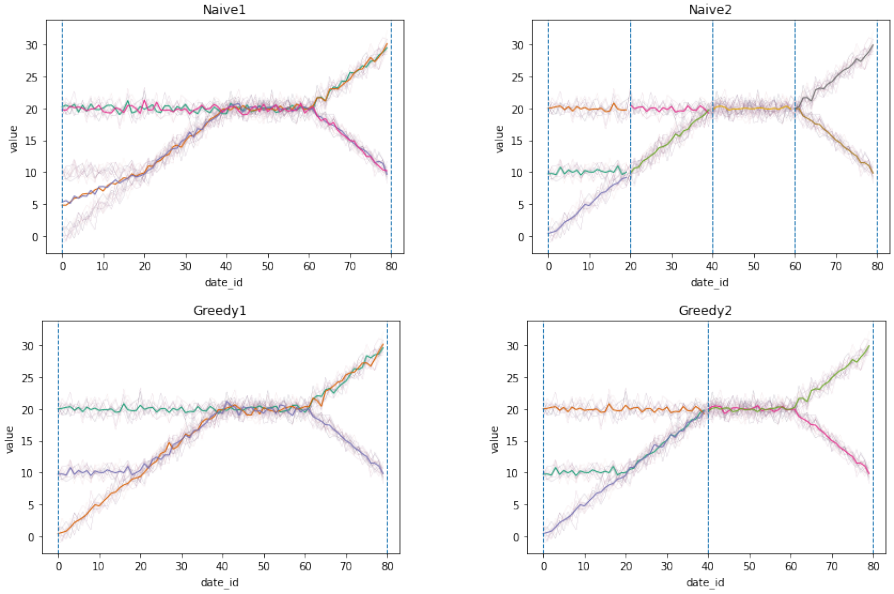


Figure 6: Algorithm comparison

a new palthlet in period 40). Our main takeaway is that greedy2 allows the number of pathlets at a time index to change across time indices, while greedy1 does not. At all time indices, greedy1 has 3 pathlets, while greedy2 first has three pathlets, and then switches to two pathlets after t=40.

6.1.3 Experiment 3: No structural shift

The aim of experiment 3 is to test what parameterisation of our model prevents the greedy algorithms from incorrectly generating more than two clusters (i.e. overfitting the data). We therefore simulate data where there are two clusters whose composition does not shift over time. We make this dataset noiser than in experiment 1 and 2.

We generate the following dataset

- $T = 80$, $N = 30$
- There are two underlying generating process: i) $N\left(-5 + \frac{t}{80}, 10\right)$, ii) $N\left(5 - \frac{t}{80}, 10\right)$
- An equal number of participants are selected from each data generating process

The raw data is plotted in top left of Figure 7 where each true cluster is colour coded. The top right figure contains the desired fit, achieved by naive1, and the bottom figure presents the fit by naive2 which is overfitted by construction.

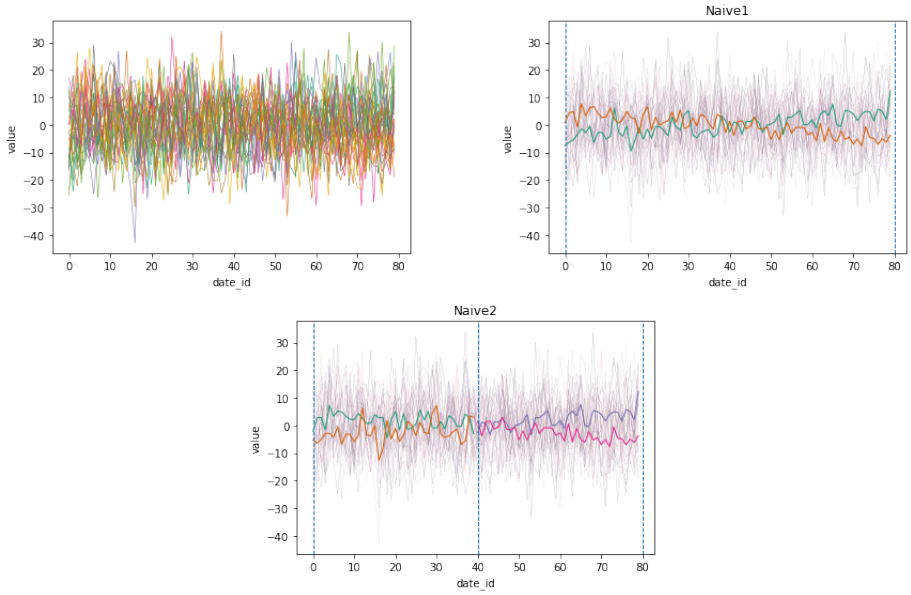


Figure 7: 2 noisy clusters with no underlying shift

We keep $c_2, c_3, c_4, c_5 = 1$, and vary c_1 from 0 to 900 in 100 intervals for greedy1 and greedy2 - repeating each for 50 iterations of the data. We set $K = \{2, 3, 4\}$ and $\mathcal{I} = \{I_1, I_2\}$ where $I_1 = \{(0, 40), (41, 80)\}$ and $I_2 = \{(0, 20), (21, 40), (41, 60), (61, 80)\}$. We record the number of pathlets selected for each iteration for each value of I and report the result in Table 1. Note that the correct number of pathlets is 2, and the maximum possible recorded pathlets is 8 with I_1 and 16 for I_2. For greedy 1, for a given I and c_1, we iterate over K (as greedy1 only takes in a single k).

As we would expect, as c_1 rises, fewer pathlets are chosen. The mechanisms for why this occur differ in greedy1 and greedy2. greedy1 merges paths whenever it can, irrespective of the objective function. As c_1 rises, the driver of fewer pathlets in greedy1 is the iterations across values of K and choosing iteration with the best objective function score (lowest cost). As such, when $I = I_2$, greedy1 falters as it cannot distinguish between clusters within $t \in (21, 60)$. As k must be the same in all segments, its best result is achieved when there are two clusters within each segment. However, the two sets of cluster assignments in

c_1	greedy1, I_1	greedy2, I_1	greedy1, I_2	greedy2, I_2
0	6.96	6.06	15.84	11.82
100	7.06	4.90	15.82	6.20
200	4.44	4.10	7.98	3.36
300	2.84	3.10	6.88	2.04
400	2.64	2.10	6.94	2.00
500	2.18	2.02	6.84	2.00
600	2.22	2.00	7.08	2.00
700	2.12	2.00	6.88	2.00
800	2.14	2.00	7.14	2.00
900	2.16	2.00	6.92	2.00

Table 1: Experiment 3 results: Average pathlets across generated datasets (optimal number is 2)

$t \in (21, 60)$ are noisy as the data is almost the same—hence the cluster assignments are almost random. Therefore, unsurprisingly, greedy1 does not merge clusters as they are unlikely to be similar from the initial clusters generated in $(0, 20)$ and $(61, 80)$. greedy2 is less sensitive to the choice of I. This is because it rather than merging similar clusters, it greedily picks clusters that optimise the objective function. This suggests the greedy1 is especially sensitive to the choice of I.

6.2 Experiments on stock market data

Next, we apply our algorithms to real stock market data.

6.2.1 Data collection

We construct a balanced time-series dataset consisting of 441 stock prices from the $S\&P500$ from 2010 to 2022. Stock price data is collected using Yahoo Finance's API [11]. For each company, we collect industry groups classifications data from Refinitiv workspace. [12].

We normalise the data in two ways: i) taking z-scores of each time-series object ii) computing daily 'abnormal' returns. We abuse the term 'abnormal returns' [15] by computing the following for each point in a time-series: $ar_i = r_{it} - \bar{r}_t$ where r_{it} is the log daily returns [13] for object i at time t and r_t is the mean log returns on day t for all time-series objects in our sample. The purpose of this normalisation is to remove common movements among all stocks. The results of both pre-processing steps are plotted in Figure 8. As the figure suggests, both of these datasets are noisy, and it is tricky to pick out clusters by eye. Picking out changes in clusterings is even harder.

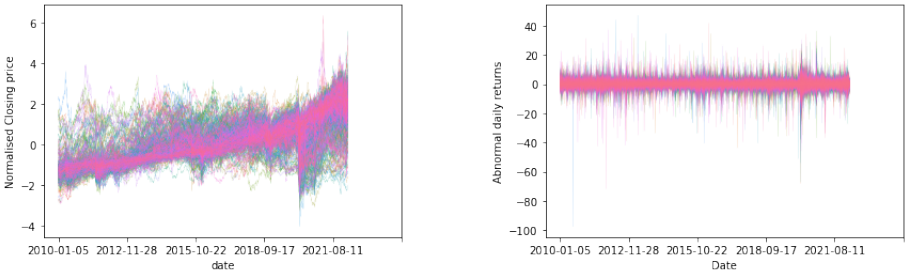


Figure 8: z-score normalised stock prices (left) and daily abnormal returns (right)

[11] More information on the Yahoo finance API is available here
[12] More information on Refinitiv workspace is availibe here here
[13] As defined in section 3

6.2.2 Parameter tuning

Before we run the experiment, we use the data and domain specific knowledge to determine reasonable choices for $c, K, \mathcal{I}$. In our data, we have 12 unique ICB industry codes. Therefore, we set k to $\{10, 11, 12, 13, 14\}$. For $\mathcal{I}$ we pick a single interval set of I that consists of 12 evenly sized periods (roughly 1 per year). We choose intervals of this length so that our algorithms given the opportunity to pick up changes in clusters at regular intervals. This is essential to experiment 4. Running more frequent intervals would increase the computational cost, and potentially introduce more noise into the clustering. Given K and I, we then run our greedy algorithms over different choices of c_1 and c_4, and set $c_2, c_3, c_5 = 1$. This is because c_1 and c_4 are the most intuitive way to control overfitting. We then search for combinations of c_1 and c_4 that lead to reasonable clusterings. We define a reasonable clusterings to have the following properties: i) under 1800 switches ii) less than 24 pathlets. As our dataset contains 441 stocks, we think it would be unreasonable for the average number of times a stock to switch clusters to exceed four (over the 12 year period our data covers). Therefore, we cap the number of switches at 1800. Given that we are searching for around 12 clusters in a given period of time, we think that having clusters completely change more than 2 times would be unreasonable. There are two reasons for this. First, too many cluster changes will increase noise into experiment 4. Second, one of the aims of clustering stocks (and most other time series objects) is that the clustering at a given point in time has some predictive power. Therefore, we want to find durable clusterings. Given the average pathlets per period is 12, a clustering where there are 144 different pathlets would imply that on average, the clustering completely changes every year. This suggests that the clusters are not durable.

For the both datasets, we vary c_1 from 2000 to 8000 in intervals of 2000. We vary c_4 from 20 in 80 in intervals of 20. We then choose the parameters that minimise the third component of the objective function (quality of fit) and satisfy the containt: # pathlets < 24, # switches < 1800.

For greedy1, we were unable to produce reasonable clusterings with 12 evenly sized intervals. Even for high regularisation parameters, greedy1 struggled to merge pathelts. Given the noise in the datase and how greedy1 merges clusters this is not surprising, and in line with our findings in experiment 3. Therefore, we focus on greedy2 in experiment 4. For greedy2, the best fitting parameters were $(c_1, c_2) \approx (6000, 20)$ for the abnormal returns dataset and $(c_1, c_2) \approx (6000, 40)$. Therefore we set $c_1 = 6000$ for greedy2 in both datasets and set $c_2 = 20, 40$ for the abnormal returns and normalised prices datasets respectively. Note that the results for z-score normalised price datasets were not promising, so there is scope for further optimisations of this dataset.

6.2.3 Experiment 4: when do clusters change?

As discussed in section 3, it is tricky to evaluate our algorithm as there is limited labeled data on the evolution of clusters in the *S&P500*. It is also hard to pick out clusters and changes in clusters visually. That said, one might expect clusterings to change in periods of volatility. Stock market volatility can be measured by the VIX index [14], that we plot in Figure 9. The figure suggests peaks in volatility on centered around the following dates: t_1: September-2015 [15]; t_2: February-2018 (Fears of interest rate hike); t_3: April-2020 (start of COVID pandemic [16]). The corresponding date ids (when starting the series in January 2010) are: $t_1 \approx 1536$, $t_2 \approx 2048$ and $t_3 \approx 2560$. Given the sizes of of spikes in volatility, we might expect that largest shift in clustering to be centered around the start of the COVID pandemic ($t_3 = 2560$).

Therefore, we run our greedy algorithms over the two processed *S&P500* datasets and identify if they identify changes in clustering on these dates relative to other dates where we allow the algorithm to change clusters. We set $K = \{10, 11, 12, 13, 14\}$ to match the 12 unique ICB Industry code in our dataset, while allowing some flexibility in the number of clusters over time. We set I as 12 evenly sized intervals from 2010 to 2022 (so that each interval consistent of a year-—256 working days [17]). We set $c = (c_1 = 6000, c_2 = 1, c_3 = 1, c_4 = 20, c_5 = 1)$ for the abnormal returns dataset and $c - (c_1 = 6000, c_2 = 1, c_3 = 1, c_4 = 40, c_5 = 1)$ for the z-score normalised stock price dataset. As clusterings for daily abnormal returns data are hard to visualise for large T, we summarise both results in Table 2.

[14] we apply a 14 day moving average. Data downloaded from Yahoo finance that can be found here

[15] 2015-2016 stock market sell-off

[16] More details on stock market reaction, see here

[17] the stock prices only have closing prices on working days

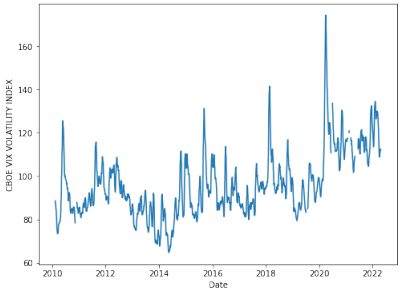


Figure 9: CBOE VIX Volatility Index from 2010 to 2022

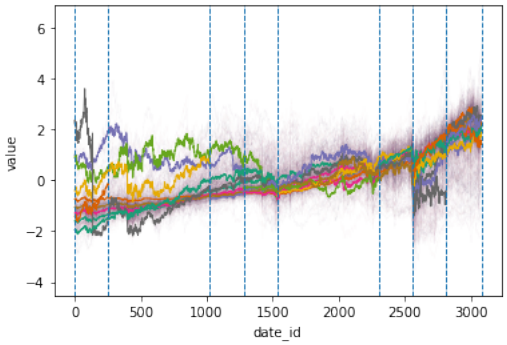


Figure 10: *stsc* $c_1 = 5000, c_4 = 50$: z-score normalised stock prices (greedy2)

	greedy2, abnormal returns	greedy2, normalised price
# joined new cluster in $t = 256$	184	46
# joined new cluster in $t = 512$	0	382
# joined new cluster in $t = 768$	0	89
# joined new cluster in $t = 1024$	10	81
# joined new cluster in $t = 1280$	9	98
# joined new cluster in *t=1536*	**174**	**60**
# joined new cluster in $t = 1792$	0	0
# joined new cluster in *t=2048*	**0**	**0**
# joined new cluster in $t = 2304$	0	25
# joined new cluster in *t=2560*	**366**	**95**
# joined new cluster in $t = 2816$	13	25
# pathlets	17	21
# switches	707	1440
# goodness of fit %	102%	174%

Table 2: Stock market *stsc* results

The data in Table 2 can be interpreted as follows: *# joined cluster in* $t = k$ indicates the number of time-series objects that were assigned to a newly formed cluster in that time period. *# pathlets* indicates the total number of pathlets in the *sub-time-series* clustering. *# switches* is the number of times that a time-series object switches to an already existing path. *# goodness of fit %* is the third component of the objective function (total distance of assigned points to their assigned pathlet) divided by corresponding distance when we run naive2 on the data (that fits the optimal k clustering at each interval with $k = 12$, and $GP = True$).

Experiment 4 results: When applied to the abnormal returns dataset, the result for greedy2 are promising. Aside from $t = 256$, (the first-segment where re calibrating clusters occurs) the two most

concentrated time periods for cluster changes are where we would expect them: at $t = 1536$ and $t = 2560$. Moreover, the largest change, is at $t = 2560$, when the VIX was at it highest (at the start of the COVID pandemic). However, there were no changes in $t = 2048$. Given the nature of the volatility in $t = 2048$ it is reasonable clustering did not change. What is most promising is that exactly when we expected the biggest shift in clustering (after the start of the COVID pandemic $t = 2560$), our algorithm detected the largest shift in clustering. The results for normalised price dataset is more nuanced. As the fit is less regularised, more switches and cluster formations occur earlier on the process. Therefore, the largest shifts in clustering occur in the first few periods. That said, we still observe an increase in the number of time-series objects that join a newly formed pathlet in $t = 2560$. It is also worth noting that the fit for normalised data is far worse relative to naive2. Unfortunately, we did not have time to tune the parameters further for the normalised price dataset. We present the results for a different parameterisation of $c_1 = 5000, c_4 = 50$ in Figure 10 for z-score normalised data to illustrate the pathlets. The figure indicates how new clusters form during the covid pandemix, though only of the newly formed clusters was durable. It also suggests how the clustering is particularly volatile further left in the chart. This is expected as the initial clustering is likely overfit to the first segment.

Overall, while this experiment has mixed results, and it is itself ad-hoc, it does suggest that our algorithm can recover trends that we theoretically believe to be true. This is promising evidence that our approach can lead to reasonable *sub-time-series* clusterings. Moreoever, it suggests the weakness of both algorithms, and suggests where they can be improved.

6.3 Future experimentation

Unfortunately, further experimentation was beyond the scope of the project. Future work should continue to test and modify these algorithms on diverse time-series datasets—for example time-series datasets in the UCR Time-series Classification Archive[18]. Moreoever, the algorithms should be tested on datasets where changes in clusterings are labled. Finally, they should be tested on their predictive ability as discussed in section 3. For example, one application is evaluating if longer lasting pathlets are less likely to break up in the future. This is helpful, as in a streaming model, we may be able to assign scores to each active pathlet at a given point in time based on how long it has been a path for.

7 Conclusion

In this paper we introduce a model for *sub-time-series clustering*: a special case of subtrajectory clustering. We describe its place in the literature: in the intersection of *different in time* time-series clustering and subtrajectory clustering. As such, issues such as how to normalise *similar in time* series data, and measure difference between time-series must be taken into considering in a sub-time-series clustering application. We then propose algorithms that seek to reach good solutions according to the models objective function. Our experiments suggest that while the algorithms show some promising results, more work needs to be done to validate their effectiveness.

There are a multiple avenues for future work. First, to what extent a *sub-time-series clustering* is interesting is almost impossible to measure. Future work could identify how to tune to parameters so that they lead to 'interesting' clustering. As mentioned in section 6 the proposed algorithms should be tested on multiple datasets and results compared to experts' view on patterns in the time-series. Another interesting direction is to enhance our approach with pre-processing steps that reduce the dimension of the time series data in each interval. Finally, there is scope to provide formal results for approximation guarantees.

References

[1] Pankaj K Agarwal, Kyle Fox, Kamesh Munagala, Abhinandan Nath, Jiangwei Pan, and Erin Taylor. Subtrajectory clustering: Models and algorithms. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 75–87, 2018.

[18] Further details on the UCR archive can be found here

[2] Saeed Aghabozorgi, Ali Seyed Shirkhorshidi, and Teh Ying Wah. Time-series clustering–a decade review. *Information Systems*, 53:16–38, 2015.

[3] Rakesh Agrawal, Christos Faloutsos, and Arun Swami. Efficient similarity search in sequence databases. In *International conference on foundations of data organization and algorithms*, pages 69–84. Springer, 1993.

[4] M Ankerst, M Breunig, HP Kriegel, R Ng, and J Sander. Ordering points to identify the clustering structure. In *Proc. ACM SIGMOD*, volume 99, 2008.

[5] David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. Technical report, Stanford, 2006.

[6] Davide Azzalini, Fabio Azzalini, Mirjana Mazuran, and Letizia Tanca. Evolution of financial time series clusters. In *SEBD*, 2019.

[7] Ignacio Benítez, José-Luis Díez, Alfredo Quijano, and Ignacio Delgado. Dynamic clustering of residential electricity consumption time series data based on hausdorff distance. *Electric Power Systems Research*, 140:517–526, 2016.

[8] Bela Bollobas, Gautam Das, Dimitrios Gunopulos, and Heikki Mannila. Time-series similarity problems and well-separated geometric sets. In *Proceedings of the thirteenth annual symposium on Computational geometry*, pages 454–456, 1997.

[9] Kevin Buchin, Maike Buchin, Marc Van Kreveld, and Jun Luo. Finding long and similar parts of trajectories. In *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 296–305, 2009.

[10] Alessandro Casini and Pierre Perron. Structural breaks in time series. *arXiv preprint arXiv:1805.03807*, 2018.

[11] Kelvin Kam Wing Chu and Man Hon Wong. Fast time-series searching with scaling and shifting. In *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 237–248, 1999.

[12] José G Dias, Jeroen K Vermunt, and Sofia Ramos. Clustering financial time series: New insights from an extended hidden markov model. *European Journal of Operational Research*, 243(3):852–864, 2015.

[13] Anja F Ernst, Marieke E Timmerman, Bertus F Jeronimus, and Casper J Albers. Insight into individual differences in emotion dynamics with clustering. *Assessment*, 28(4):1186–1206, 2021.

[14] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *kdd*, volume 96, pages 226–231, 1996.

[15] Eugene F Fama. Market efficiency, long-term returns, and behavioral finance. *Journal of financial economics*, 49(3):283–306, 1998.

[16] Philip Hans Franses and Thomas Wiemann. Intertemporal similarity of economic time series: an application of dynamic time warping. *Computational Economics*, 56(1):59–75, 2020.

[17] Martin Gavrilov, Dragomir Anguelov, Piotr Indyk, and Rajeev Motwani. Mining the stock market (extended abstract) which measure is best? In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 487–496, 2000.

[18] Tomasz Górecki. Using derivatives in a longest common subsequence dissimilarity measure for time series classification. *Pattern Recognition Letters*, 45:99–105, 2014.

[19] Joachim Gudmundsson and Marc Van Kreveld. Computing longest duration flocks in trajectory data. In *Proceedings of the 14th annual ACM international symposium on Advances in geographic information systems*, pages 35–42, 2006.

[20] Kartikay Gupta and Niladri Chatterjee. Financial time series clustering. In *International Conference on Information and Communication Technology for Intelligent Systems*, pages 146–156. Springer, 2017.

[21] Sariel Har-Peled. *Geometric approximation algorithms*. Number 173. American Mathematical Soc., 2011.

[22] Sariel Har-Peled and Soham Mazumdar. On coresets for k-means and k-median clustering. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 291–300, 2004.

[23] Lingxiao Huang, K Sudhir, and Nisheeth Vishnoi. Coresets for time series clustering. *Advances in Neural Information Processing Systems*, 34, 2021.

[24] Ali Javed, Byung Suk Lee, and Donna M Rizzo. A benchmark study on time series clustering. *Machine Learning with Applications*, 1:100001, 2020.

[25] Hoyoung Jeung, Heng Tao Shen, and Xiaofang Zhou. Convoy queries in spatio-temporal databases. In *2008 IEEE 24th International Conference on Data Engineering*, pages 1457–1459. IEEE, 2008.

[26] Hoyoung Jeung, Man Lung Yiu, Xiaofang Zhou, Christian S Jensen, and Heng Tao Shen. Discovery of convoys in trajectory databases. *arXiv preprint arXiv:1002.0963*, 2010.

[27] Panos Kalnis, Nikos Mamoulis, and Spiridon Bakiras. On discovering moving clusters in spatio-temporal data. In *International symposium on spatial and temporal databases*, pages 364–381. Springer, 2005.

[28] Jong Hee Kang, William Welbourne, Benjamin Stewart, and Gaetano Borriello. Extracting places from traces of locations. *ACM SIGMOBILE Mobile Computing and Communications Review*, 9(3):58–68, 2005.

[29] Eamonn Keogh, Kaushik Chakrabarti, Michael Pazzani, and Sharad Mehrotra. Dimensionality reduction for fast similarity search in large time series databases. *Knowledge and information Systems*, 3(3):263–286, 2001.

[30] Teuvo Kohonen. The self-organizing map. *Proceedings of the IEEE*, 78(9):1464–1480, 1990.

[31] Peter Langfelder, Bin Zhang, and Steve Horvath. Defining clusters from a hierarchical cluster tree: the dynamic tree cut package for r. *Bioinformatics*, 24(5):719–720, 2008.

[32] Jae-Gil Lee, Jiawei Han, and Kyu-Young Whang. Trajectory clustering: a partition-and-group framework. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 593–604, 2007.

[33] Yang Li, Qixing Huang, Michael Kerber, Lin Zhang, and Leonidas Guibas. Large-scale joint map matching of gps traces. In *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 214–223, 2013.

[34] Yang Li, Yangyan Li, Dimitrios Gunopulos, and Leonidas Guibas. Knowledge-based trajectory completion from sparse gps samples. In *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 1–10, 2016.

[35] T Warren Liao. Clustering of time series data—a survey. *Pattern recognition*, 38(11):1857–1874, 2005.

[36] James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. Oakland, CA, USA, 1967.

[37] Sepideh Mahabadi and Ali Vakilian. Individual fairness for k-clustering. In *International Conference on Machine Learning*, pages 6586–6596. PMLR, 2020.

[38] Meena Mahajan, Prajakta Nimbhorkar, and Kasturi Varadarajan. The planar k-means problem is np-hard. In *International workshop on algorithms and computation*, pages 274–285. Springer, 2009.

[39] What Is Data Mining. Data mining: Concepts and techniques. *Morgan Kaufinann*, 10:559–569, 2006.

[40] Ismail Bin Mohamad and Dauda Usman. Standardization and its effects on k-means clustering algorithm. *Research Journal of Applied Sciences, Engineering and Technology*, 6(17):3299–3303, 2013.

[41] Kei Nakagawa, Mitsuyoshi Imamura, and Kenichi Yoshida. Stock price prediction using k-medoids clustering with indexing dynamic time warping. *Electronics and Communications in Japan*, 102(2):3–8, 2019.

[42] Tim Oates, Laura Firoiu, and Paul R Cohen. Clustering time series with hidden markov models and dynamic time warping. In *Proceedings of the IJCAI-99 workshop on neural, symbolic and reinforcement learning methods for sequence learning*, pages 17–21. Citeseer, 1999.

[43] Dong Hwan Oh and Andrew J Patton. Dynamic factor copula models with estimated cluster assignments. 2021.

[44] Andrey Tietbohl Palma, Vania Bogorny, Bart Kuijpers, and Luis Otavio Alvares. A clustering-based approach for discovering interesting places in trajectories. In *Proceedings of the 2008 ACM symposium on Applied computing*, pages 863–868, 2008.

[45] John Paparrizos, Chunwei Liu, Aaron J Elmore, and Michael J Franklin. Debunking four long-standing misconceptions of time-series distance measures. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1887–1905, 2020.

[46] Maurice Roux. A comparative study of divisive and agglomerative hierarchical clustering algorithms. *Journal of Classification*, 35(2):345–366, 2018.

[47] Artur Starczewski and Adam Krzyżak. Performance evaluation of the silhouette index. In *International conference on artificial intelligence and soft computing*, pages 49–58. Springer, 2015.

[48] Sheng Wang, Zhifeng Bao, J Shane Culpepper, and Gao Cong. A survey on trajectory data management, analytics, and learning. *ACM Computing Surveys (CSUR)*, 54(2):1–36, 2021.

[49] Wei Wang, Jiong Yang, Richard Muntz, et al. Sting: A statistical information grid approach to spatial data mining. In *Vldb*, volume 97, pages 186–195. Citeseer, 1997.

[50] Seyedjamal Zolhavarieh, Saeed Aghabozorgi, and Ying Wah Teh. A review of subsequence time series clustering. *The Scientific World Journal*, 2014, 2014.