StratAlign: Uncovering Tactical Patterns through Large-Scale Event Sequence Matching

Ahmed El-Roby[1], Abdelrahman Hefny[2], Alireza Choubineh[3]
Carleton University

Abstract

The burgeoning field of sports analytics has led to the proliferation of tools that deliver real-time insights to coaching staff, aiding them in strategic decision-making during games. However, existing systems focus solely on in-game data, thereby overlooking the benefits of incorporating historical data for deeper, contextual insights. In this paper, we introduce StratAlign, a novel system designed to mine large-scale historical events data to identify similar ball movement patterns, or "trajectories," in football games. We propose a dynamic time warping (DTW)-based distance function that offers robust trajectory comparison, and employ a clustering mechanism to efficiently prune the search space. Furthermore, StratAlign is built with scalability in mind, using a disk-based hash index to maintain the dataset and several memory-eviction strategies to operate within limited resource constraints. We also address the challenge of aligning event timestamps with actual video footage through an innovative computer vision approach. Our experimental evaluations confirm the system's ability to retrieve relevant ball trajectories in significantly less time compared to an in-memory baseline solution, making StratAlign an efficient and cost-effective tool for real-time strategic analysis in football games.

!

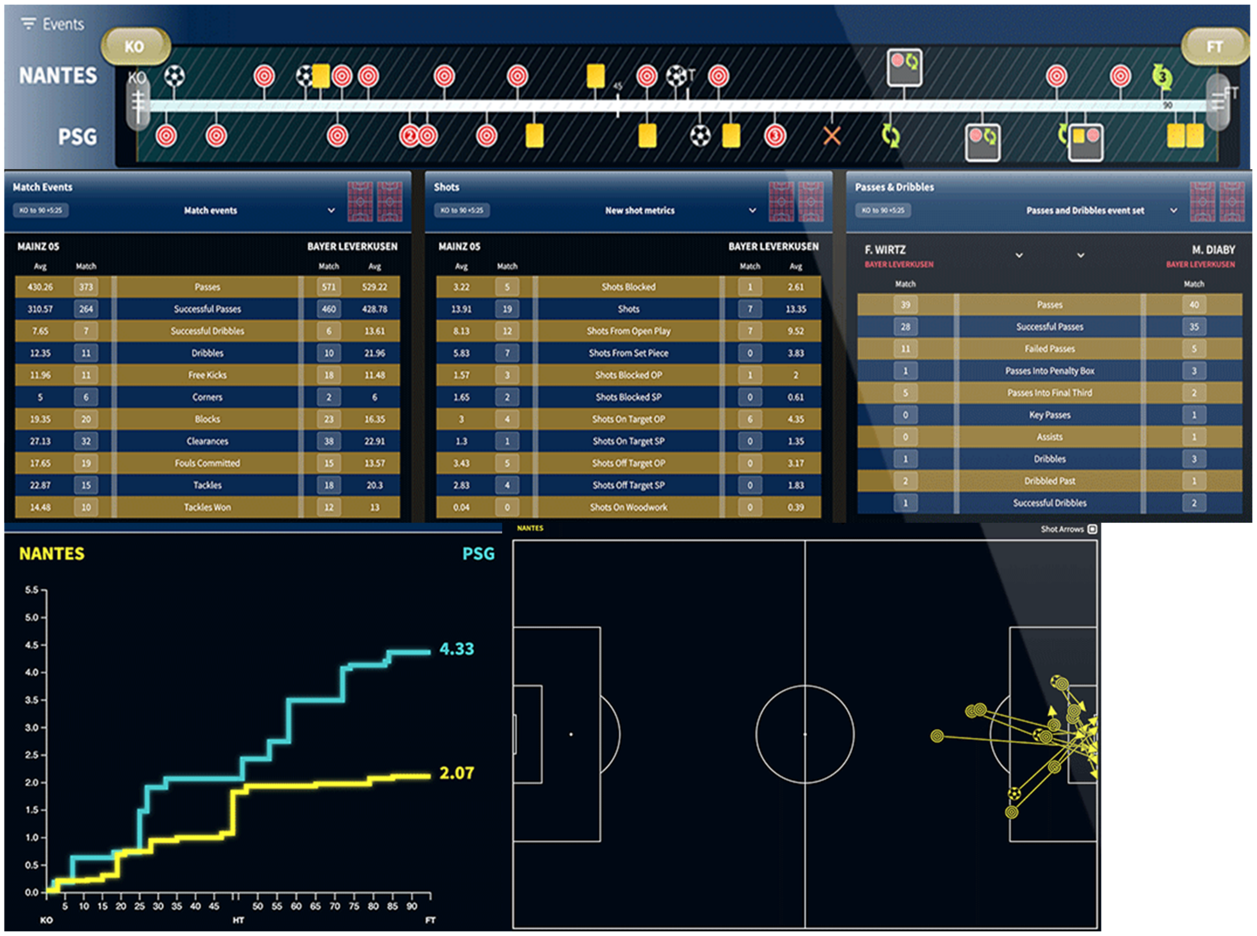
1. Introduction

Recent years have witnessed a great interest in providing teams and analysts with real-time insights during games of different sports [1, 2]. In football, these insights include but are not limited to: a chronological account of match events, compiled statistics (including passes, dribbles, and shots), and graphical presentations of pertinent data through charts and tables as demonstrated in Figure 1, which shows an example of one tool [1] that provides such insights. Such tools play an important role in supporting decision-making for the coaching staff to adapt their in-game strategies according to the changes happening during the game with respect to the collected data. Nevertheless, current solutions predominantly focus on the assimilation and presentation of in-game data, neglecting the potential value that could be derived from the integration of historical data to facilitate a more profound understanding of the ongoing game dynamics. This brings to light an opportunity for the development of tools capable of collocating live events with historical data to offer richer insights. To illustrate the potential utility of such an approach, we present the following example:

[1] ahmed.elroby@carleton.ca

[2] abdelrahman.em@gmail.com

[3] alirezachoubineh@cmail.carleton.ca



Match Events

KO to 90 +5:25 | Match events

MAINZ 05 Avg	MAINZ 05 Match		BAYER LEVERKUSEN Match	BAYER LEVERKUSEN Avg
430.26	373	Passes	571	529.22
310.57	264	Successful Passes	460	428.78
7.65	7	Successful Dribbles	6	13.61
12.35	11	Dribbles	10	21.96
11.96	11	Free Kicks	18	11.48
5	6	Corners	2	6
19.35	20	Blocks	23	16.35
27.13	32	Clearances	38	22.91
17.65	19	Fouls Committed	15	13.57
22.87	15	Tackles	18	20.3
14.48	10	Tackles Won	12	13

Shots

KO to 90 +5:25 | New shot metrics

MAINZ 05 Avg	MAINZ 05 Match		BAYER LEVERKUSEN Match	BAYER LEVERKUSEN Avg
3.22	5	Shots Blocked	1	2.61
13.91	19	Shots	7	13.35
8.13	12	Shots From Open Play	7	9.52
5.83	7	Shots From Set Piece	0	3.83
1.57	3	Shots Blocked OP	1	2
1.65	2	Shots Blocked SP	0	0.61
3	4	Shots On Target OP	6	4.35
1.3	1	Shots On Target SP	0	1.35
3.43	5	Shots Off Target OP	0	3.17
2.83	4	Shots Off Target SP	0	1.83
0.04	0	Shots On Woodwork	0	0.39

Passes & Dribbles

KO to 90 +5:25 | Passes and Dribbles event set

F. WIRTZ BAYER LEVERKUSEN Match		M. DIABY BAYER LEVERKUSEN Match
39	Passes	40
28	Successful Passes	35
11	Failed Passes	5
1	Passes Into Penalty Box	3
5	Passes Into Final Third	2
0	Key Passes	1
0	Assists	1
1	Dribbles	3
2	Dribbled Past	1
1	Successful Dribbles	2

Figure 1: Example of live analytics shown during games from Statsbomb's IQ Live [1]. On top, a visualization of the important events of the game is shown as a timeline. In the middle, important statistics about the match events, shots, dribbles and passes are shown in tables. On the bottom, plots of the race charts of the two teams and a shot map of one of the teams are shown.

Example 1: During a match between teams A and B, team A effectively orchestrates a build-up strategy initiating from a goal kick and progressing into team B's half. Team B's coaching staff find it challenging to counter this strategy effectively. Unbeknownst to them, a similar strategy was previously deployed by a third team, C, in a different competition and season. However, it was successfully thwarted by team D employing a specific pressure scheme. Regrettably, due to the constraints in the existing analytical tools and the lack of cross-referential data from disparate seasons and competitions, team B's staff remains unaware of this potentially effective counterstrategy. A solution leveraging a tool that can swiftly identify and highlight such historical patterns could significantly enhance the strategic depth and responsiveness of coaching teams during live matches. This initiative underscores the necessity for a tool proficient in automating the recognition of analogous play patterns across different databases, thereby presenting them to coaching staff in real-time to refine decision-making processes during a game.

Building such a tool is a challenging task due to the scale of the mined historical data. The data that can be used to help with such a task is called the *events data*, which can be viewed as logs of all the events happening during any game. Due to the exhaustive nature of the data collected in these logs, a large number of events for each game are recorded (typically over 3000 events per game). Collecting data over multiple seasons in different competitions can result in storing tens of millions of events that correspond to tens of gigabytes of data [3, 4]. Identifying sequences of events analogous to a particular series of interest (for instance, the strategy employed by team A in Example 1) from this vast repository necessitates sifting through millions of historical events, a process that would consume an impractical amount of time (on the order of tens of minutes), thereby failing to meet the urgent demands of real-time game analysis. This inadequacy in the baseline solution arises fundamentally from two issues: 1. The enormity of the historical data entails that the majority of it is stored on disk during the retrieval process, which historically remains the slowest storage medium in computing environments, considerably elongating the search time [5, 6, 7]. 2. The target sequence bears resemblance to only a fraction of the total search space, implying that a large proportion of computational cycles dedicated to finding matches will be expended on non-congruent sequences.

In this paper, we introduce StratAlign, a system that addresses the aforementioned challenges. Following is the outline of StratAlign:
Ball Trajectories: First, we define what sequences of events can be useful in identifying playing patterns in our dataset. Predominantly, these encompass sequences wherein the ball is actively maneuvered on the pitch (e.g., we would not be interested in events that log players receiving cards or substitutions). These sequences of events can be represented as trajectories of the movement of the ball in one team's possession. Despite filtering, the dataset remains extensive. For example, in our dataset (discussed in Section 7), from over 14 million events, we were able to extract over 445 thousand trajectories.
Trajectories Distance Function: Second, we introduce a distance function that can be used to compare any pair of trajectories. While the Euclidean distance could be a customary choice given the pitch's Euclidean space consideration, we argue that the Euclidean distance is not suitable in comparing pairs of trajectories for two reasons: 1. The trajectories can be of arbitrary lengths. 2. The Euclidean distance neglects the temporal aspects inherent to the trajectories. To address this, we adopt a dynamic time warping (DTW) methodology to calculate the distance between any pair of trajectories [8]. This approach can stretch or compress sections of the trajectory to align similar patterns that occur at different rates and is resilient to noise and outlying points in the trajectories. Figure 2 shows four different examples of an input trajectory (orange) and its most similar trajectory (red) from a dataset of 3715 matches. The figure shows how much similar the two trajectories are, revealing similar patterns in progressing the ball down the pitch towards the opponent's half. Although the DTW approach demands higher computational resources compared to the Euclidean metric, StratAlign mitigates this by eliminating most of the search space of trajectories to be compared with an input trajectory.
Dividing and Indexing the Search Space: In the framework of StratAlign, we leverage strategic division and indexing of the search space to enhance efficiency by avoiding needless comparisons between an input trajectory and others that hold a low likelihood of matching. To demonstrate the need for such feature, we ran an experiment where we store trajectories of different number of matches in memory, and for an input trajectory, we simply scan all the trajectories to retrieve the most similar one using the DTW distance function. We repeat this experiment for different input trajectory lengths and data coming from different number of matches. For any input trajectory length, we repeat the experiment 20

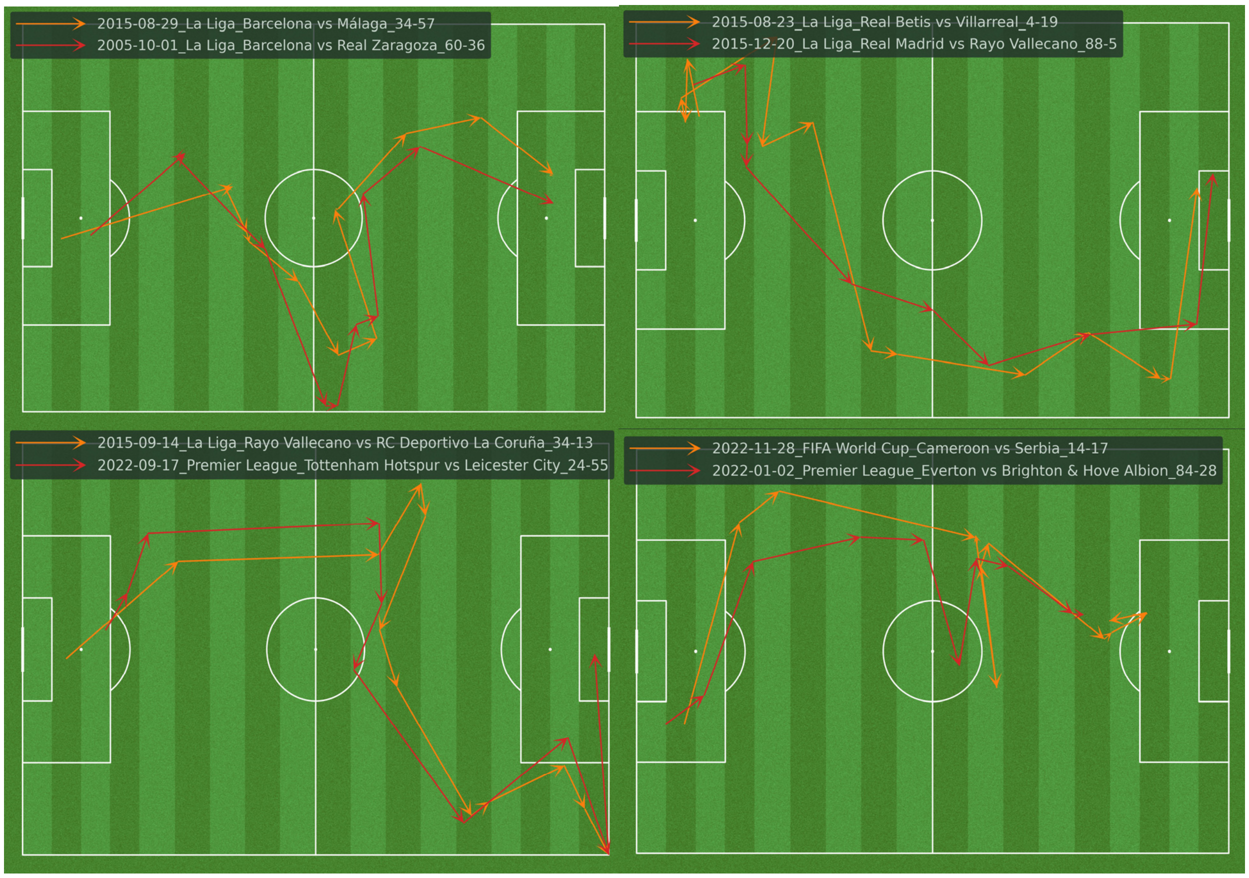


Figure 2: Four representative trajectories, shown in red, are identified as the most similar to the input trajectories, depicted in orange, based on the Dynamic Time Warping (DTW) distance metric.

times and report the average execution time. The results of this experiment are shown in Figure 3. The experiment incorporates five distinct trajectory datasets: (1) Trajectories from Premier League matches spanning the 2021-2022 and 2022-2023 seasons, denoted in yellow; (2) Dataset (1) extended with Statsbomb open data as of July 20, 2023[4], represented in orange; (3) Dataset (1) extended with Statsbomb open data up to August 19, 2023[5], marked in blue; (4) A replication of Dataset (3), indicated in green; and (5) Dataset (3) amplified threefold, shown in purple. The figure demonstrates that the execution time is influenced by both the length of the input trajectory and the size of the dataset under analysis. From the experimental results, it becomes evident that scanning the entirety of large datasets to identify the trajectory most similar to a given input trajectory is impractical for real-time analytics, where

[4] https://github.com/statsbomb/open-data/tree/4f8773dd63606e8d248b77ddab51df8b09150177

[5] https://github.com/statsbomb/open-data/tree/0067cae166a56aa80b2ef18f61e16158d6a7359a

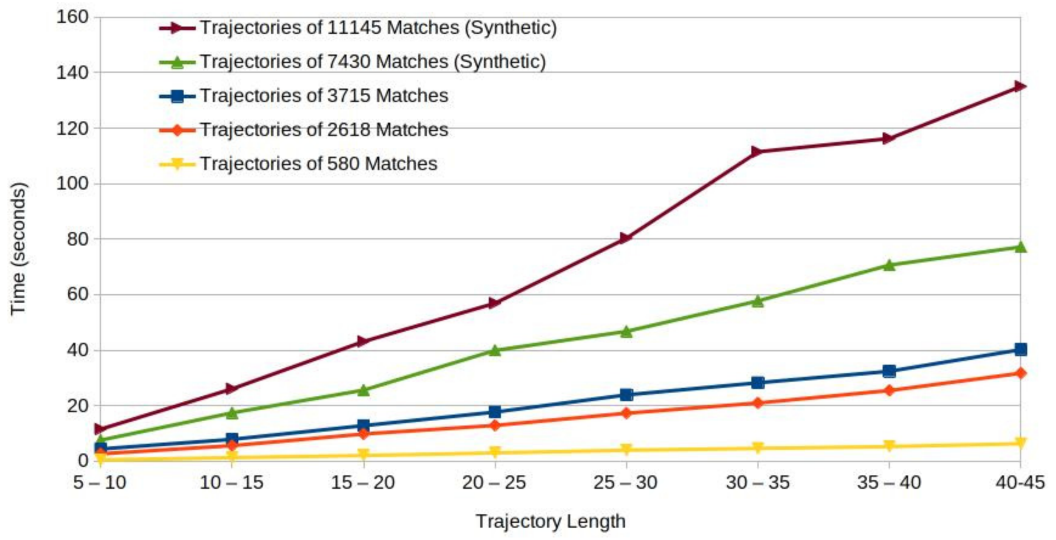


Figure 3: The execution time of scanning all the trajectories to find the most similar trajectory to an input trajectory. The x-axis shows the input trajectory length, and the y-axis shows the average execution time of 20 runs.

minimal query latency is required. In contrast, for datasets of moderate size, the observed execution times extend to several seconds or even minutes.

In StratAlign, we employ a pre-processing step involving the clustering of trajectories to streamline the search process during query time. This entails segregating the dataset into clusters so that, while processing a query, we can focus solely on the clusters most resembling the input trajectory, effectively reducing the search space and negating the need to scan all available trajectories. Given the computational and memory demands of pre-calculating distances between each trajectory pair using the DTW distance function, StratAlign adopts a strategy of compressing trajectories into a format that retains essential geometric features. This facilitates efficient clustering while ensuring trajectories with high similarities are grouped together or in adjacent clusters. Upon receipt of an input trajectory, StratAlign identifies the most congruent clusters by assessing the similarity between the input and each cluster's representative elements (e.g., medoids) through the application of the DTW function. Consequently, the ensuing comparison process is restricted to the trajectories within the selected clusters, a strategy that significantly narrows the search space and promotes computational efficiency.

Ensuring Scalability: StratAlign has been conceptualized with a paramount focus on scalability, accommodating user-configured resource limitations to adeptly manage large datasets characteristic of real-world historical data. Recognizing the impracticability of hosting all trajectories in-memory due to the substantial size of authentic historical datasets and the steep costs associated with high resource consumption in contemporary shared cloud frameworks [9, 10, 11], StratAlign navigates this challenge through a meticulous allocation of available resources. We take advantage of our partitioning of the search space to store in memory only a hash index of the clusters that are stored on disk. We utilize the available memory budget to host the index as well as a selected number of clusters according to multi-

ple memory-eviction strategies. This architectural decision assures commendable efficiency for StratAlign even when operating within a constrained memory environment, therefore offering a viable solution for real-time analysis without necessitating extensive resources.
Visualization: In StratAlign, we have elected to present our output through video snippets that delineate the trajectory sequences, offering coaching personnel a vivid depiction that encompasses players' movements and positioning across the field during the pertinent sequences. Despite its inherent advantages, this approach entails substantial complexities owing to the discrepancies between the game videos' offset times and the corresponding clock times documented in the event logs, denoted in minutes and seconds. Given the necessity to maintain a video repository corresponding to the matches the data is collected from, establishing the offset between the event logs' clock time and the initiation point in the video file represents a formidable challenge, exacerbated by the magnitude of the endeavor and the prohibitive demands of manual annotations for individual video files. Addressing this, StratAlign introduces an innovative solution leveraging machine learning and computer vision frameworks, engineered specifically to ascertain temporal offsets from video materials. This technique orchestrates the strengths of object detection and optical character recognition (OCR) technology, underpinned by heuristic principles. At its core, our methodology is focused on the systematic extraction of timestamps embedded within scoreboards across a predefined subset of frames. These temporal timestamps are subsequently used to compute precise temporal offsets, effectively measuring the temporal difference between match time and corresponding frame time. This derived offset enables the precise localization of the starting point for a video segment that aligns with the trajectory sequence of specific interest.

Following is the summary of our contributions:

- To the best of our knowledge, we are the first to develop a scalable solution to address the problem of large-scale ball trajectory matching that finds the most similar trajectories to an input trajectory in less than 1 second for long trajectories and in less than half a second for most trajectories.
- We eliminate most of the search space for any input trajectory by partitioning the search space of historical trajectories into a large number of smaller clusters such that the number of comparisons for an input trajectory is reduced by 91% in the worst case and 98% in the best case without sacrificing the quality of the output.
- We develop a visualization solution that is based on automatic alignment between the clock time in the event logs and when the events occur in the video such that it is possible to retrieve the video snippet of any event of interest.
- We experimentally evaluate our approach and show that with sufficient memory budget, it is 11 times more efficient on average than an in-memory baseline that stores only the ball trajectories in memory. We also show that with extremely conservative memory budget StratAlign is 3 times more efficient on average than the in-memory baseline despite using only 10% of the memory size used in the baseline.

The rest of the paper is organized as follows: Section 2 presents the overall architecture of StratAlign. Section 3 introduces important definitions. Section 4 discusses the trajectory similarity distance function. Section 5 presents how we organize the search space and how this organization serves our purpose of conserving compute and memory resources. Section 6 presents our approach to automatically identify each video's offset such that it aligns with the event logs. Section 7 discusses the various experiments that demonstrate the efficiency and scalability of StratAlign. Finally, Section 8 concludes the paper.

2. StratAlign Architecture

Figure 4 shows the architecture of StratAlign, which does pre-processing in the offline phase, and serves the user's queries in the online phase. In the offline phase, StratAlign starts by scanning all the raw event log files to extract all ball trajectories (will be formally defined in Section 3). In order to facilitate efficient clustering of these trajectories, StratAlign summarizes each trajectory by storing only a small number of features that represent the trajectories. The trajectories are organized into clusters, markedly reducing the total count of groups compared to individual trajectories. Depending on the user's allocated memory budget, these clusters are stored either in memory or on the disk. A representative trajectory is kept in memory for each cluster (e.g., the medoid), and a pointer to where its cluster is stored is also kept (either in memory or on disk). This pair of information (i.e., the representative trajectory and the pointer) make up the trajectory index. This organization of the search space is discussed in Section 5. In the online phase, an input trajectory is given by the user, StratAlign iterates over the index entries to find the representative trajectories that are most similar to the input trajectory, then it retrieves their corresponding clusters. The input trajectory is then compared with all the trajectories in the clusters in order to find the top-k similar trajectories. Finally, the videos of the most similar trajectories are retrieved from the videos database (Section 6 discusses how such videos are managed).

3. Problem Definition

In this section, we introduce the definitions needed for the discussions in the next sections.

Definition 1 (Attribute): Let A be a set of all possible attribute keys and V be the set of all possible attribute values. An attribute $a \in A$ is a function $a: A \rightarrow V$ mapping an attribute key to its value.

Definition 2 (Event): Let E be the set of all possible events. An event $e \in E$ is defined as a subset of A, i.e., $e \subseteq A$.

Definition 3 (Document): Let D be the set of all possible documents. A document $d \in D$ is defined as an ordered sequence of events, i.e., $d = (e_1, e_2, \ldots, e_\mathrm{n})$ for some $n \geq 0$, where each $e_\mathrm{i} \in E$ for $1 \leq i \leq n$.

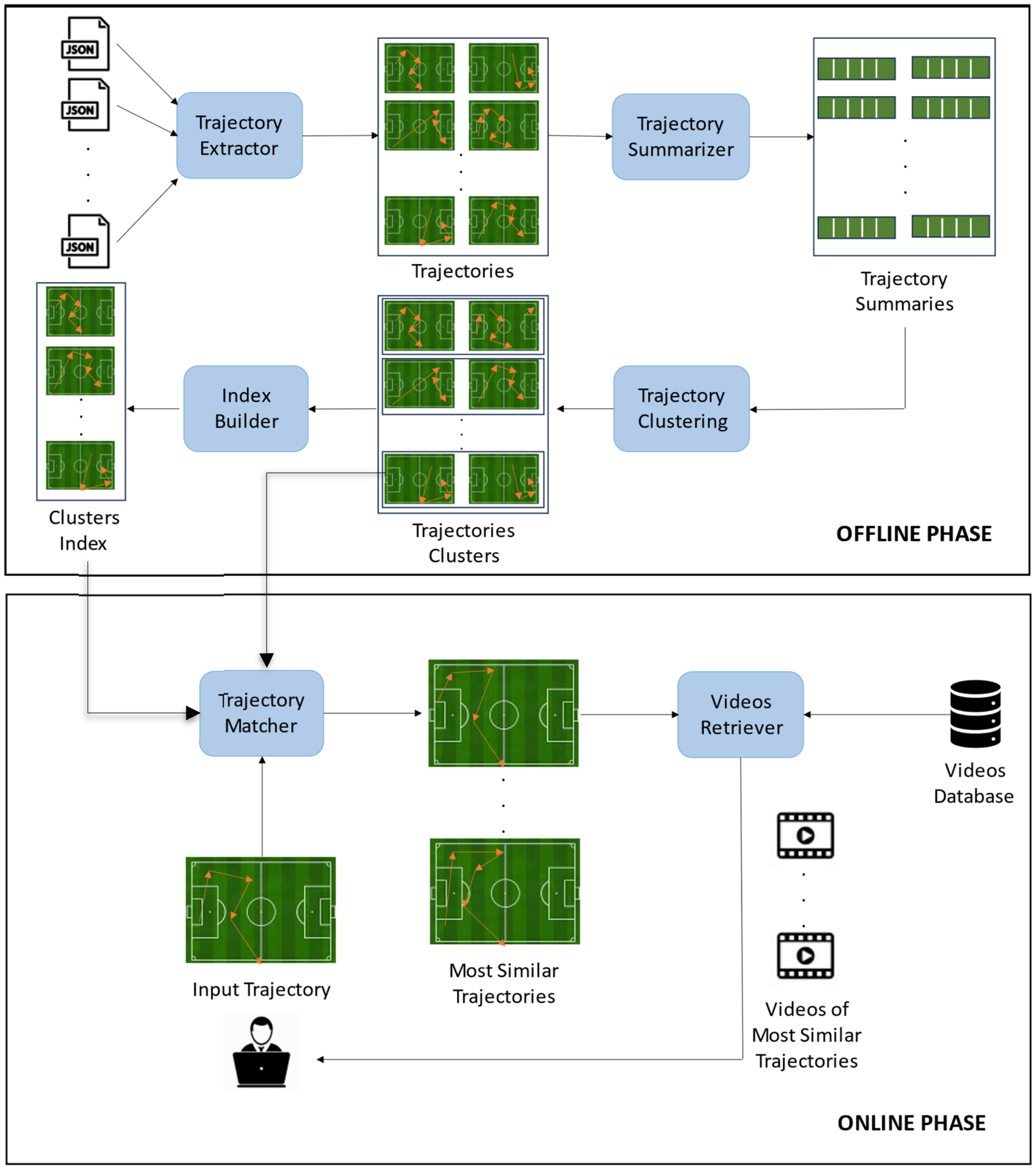


Figure 4: The architecture of StratAlign.

Definition 4 (Dataset): Let S be a dataset. A dataset S is defined as a set of documents, i.e., $S \subseteq D$.

Definition 5 (Trajectory): Let us denote an attribute key of interest as $k \in A$ and a particular value of this key as $v \in V$. A trajectory T is a maximal ordered sequence of consecutive events $(e_1, e_2, \dots, e_m)$ from any document $d \in S$, where $m \geq 0$, satisfying the following conditions:

i. Each event e_i, for $1 \leq i \leq m$, contains an attribute $a \in e_i$ such that $a(k) = v$.

ii. The trajectory T starts with an event e_1 in d that either is the first event in d or follows an event e_0 where $a(k) \neq v$ in e_0.

iii. The trajectory T ends with an event e_m in d that either is the last event in d or precedes an event e_{m+1} where $a(k) \neq v$ in e_{m+1}.

In practice, the attribute used to define the trajectory is the attribute "possession_team". This definition entails that the trajectory is the sequence of events where the ball is in one team's possession. Based on the previous definitions, we formalize the problem of finding the most similar top-k trajectories for a given input trajectory.

Definition 6 (Top-k Trajectory Similarity Search):

Input:

1. A trajectory T_{input} which is an ordered sequence of events, defined as $T_{input} = (e_1, e_2, \ldots, e_p)$ for some $p \geq 0$, where each event $e_i \in E$ for $1 \leq i \leq p$.
2. A dataset S which is defined as a set of documents, where each document $d \in D$ can produce one or multiple trajectories as defined earlier.
3. A positive integer k representing the number of desired top similar trajectories.
4. A distance function $dist: T \times T \rightarrow R^+$ which measures the similarity between two trajectories. The distance function yields non-negative real numbers, with smaller values indicating higher similarity.

Output:

A set R containing the top-k trajectories from S most similar to T_{input}, i.e., $R = \{T_1, T_2, \ldots, T_k\}$ such that:

1. Each trajectory T_i is derived from some document $d \in S$.
2. The trajectories in R are ordered by increasing distance from T_{input} according to $dist$. Specifically, for all $i < j \leq k$, $dist(T_{input}, T_i) \leq dist(T_{input}, T_j)$.

In the following section, we discuss the distance function used in StratAlign.

4. Distance Between Trajectories

While utilizing Euclidean distance is a prevalent practice for determining distances within Euclidean spaces, we maintain that for the specific objective of distinguishing the most analogous trajectories to a designated input trajectory, this approach is not applicable. This stems from the fact that Euclidean distance necessitates equal sequence lengths, establishing the distance based on corresponding sequence points, yet it remains susceptible to alterations in amplitude and phase shifts. Contrastingly, Dynamic Time Warping (DTW) [12, 13, 8] operates without the precondition of sequence alignment, endeavoring instead to identify the most optimal alignment across sequences of diverse lengths. This technique "warps" the temporal dimension of the sequences to facilitate the most favorable alignment, thus offering a strategic solution for managing sequences exhibiting disparities in length or those undergoing phase disparities owing to time shifts. DTW has proven efficacious in applications like speech and gesture recognition, as well as in time series analysis, areas where phase deviations and time expansions are customary phenomena. Following, we discuss the details of calculating the DTW distance between a pair of ball trajectories.

```
Algorithm 1: Dynamic Time Warping Distance
Data: Two trajectories $T_a = \{l_a(1), l_a(2), \ldots, l_a(n)\}$ and $T_b = \{l_b(1), l_b(2), \ldots, l_b(m)\}$
Result: DTW distance between $T_a$ and $T_b$
Function DTW_DISTANCE($T_a, T_b$);
    Declare matrix $M[1 \ldots n][1 \ldots m]$;
    $M[1][1] \leftarrow$ EUCLIDEAN_DISTANCE($l_a(1), l_b(1)$);
    for $i \leftarrow 2$ to $n$ do
        $M[i][1] \leftarrow M[i-1][1]+$ EUCLIDEAN_DISTANCE($l_a(i), l_b(1)$);
    for $j \leftarrow 2$ to $m$ do
        $M[1][j] \leftarrow M[1][j-1]+$ EUCLIDEAN_DISTANCE($l_a(1), l_b(j)$);
    for $i \leftarrow 2$ to $n$ do
        for $j \leftarrow 2$ to $m$ do
            $cost \leftarrow$ EUCLIDEAN_DISTANCE($l_a(i), l_b(j)$);
            $M[i][j] \leftarrow cost + \min\{M[i-1][j], M[i][j-1], M[i-1][j-1]\}$;
    return $M[n][m]$;
Function EUCLIDEAN_DISTANCE($p_1, p_2$);
    return $\sqrt{(p_1.x - p_2.x)^2 + (p_1.y - p_2.y)^2}$;
```

Initially, we will integrate the notion of location into our definition of trajectory. Following this, we will explore the application of DTW in calculating the distance between various trajectories, guided by the locations pertinent to their respective events. For each event e_i in a trajectory T, associate a location $l(e_i)$ from the set L of all possible locations. A location can be represented as a point in a two-dimensional space (i.e., $l(e_i) = (x, y)$). Now, the trajectory T can be seen as a sequence of locations: $T = [l(e_1), l(e_2), \ldots, l(e_p)]$. Dynamic Time Warping (DTW) is a technique used for measuring the similarity between two temporal sequences that may vary in speed [12, 13]. For trajectories, this translates to sequences of locations. Given two trajectories T_a and T_b with lengths n and m respectively, the DTW distance is calculated using a matrix X of size $n \times m$. Let $dist\left(l(e_i^a), l(e_j^b)\right)$ be the Euclidean distance between the locations of events e_i^a from T_a and e_j^b from T_b. Then, each element $M(i, j)$ of the matrix is defined recursively as:

$$M(i,j) = dist\left(l(e_i^a), l(e_j^b)\right) + min\{M(i-1,j), M(i-1,j-1), M(i,j-1)\}$$

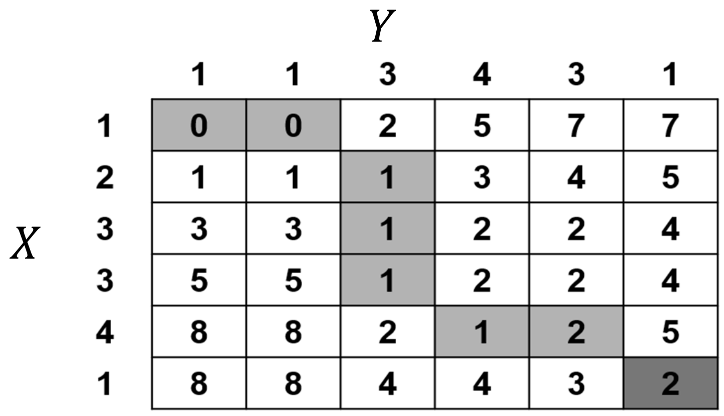
With initial conditions:

$M(1,j) = \sum_{k=1}^{j} dist\left(l(e_1^a), l(e_k^b)\right)$,

$M(i,1) = \sum_{k=1}^{i} dist\left(l(e_k^a), l(e_1^b)\right)$,

$M(1,1) = dist\left(l(e_1^a), l(e_1^b)\right)$

The value $D(n, m)$ gives the DTW distance between the two trajectories T_a and T_b. Algorithm 1 shows the steps of calculating the DTW distance between the two input trajectories. The time complexity of this algorithm is $O(n \times m)$, where n is the length of the first trajectory and m is the length of the second trajectory.



		Y					
		1	1	3	4	3	1
X	1	0	0	2	5	7	7
	2	1	1	1	3	4	5
	3	3	3	1	2	2	4
	3	5	5	1	2	2	4
	4	8	8	2	1	2	5
	1	8	8	4	4	3	2

Figure 5: An example of how the DTW distance is calculated for two 1-dimensional series.

Figure 5 illustrates an application of DTW to determine the distance between two 1-dimensional series, denoted as X and Y. The value situated at the bottom right of the depicted matrix indicates the distance between the two series, while the shaded elements delineate the warping path, highlighting which points in one series are being compared to the points in the other series. This method is adaptable to n-dimensional series, where the Euclidean distance between corresponding points in each series informs the individual matrix element computations.

5. Organizing the Search Space

Given the scale of historical events data, comparing the input trajectory with all the trajectories extracted from the events data according to *Definition 5* would be inefficient and contradicts with the time-critical nature of the problem. Moreover, to carry out such comparisons, the historical trajectories need to be stored in memory to avoid inefficient disk access. Again, given the scale of the problem, this will be either impractical, or expensive [9, 10, 11]. Motivated by our objective of finding a small number of trajectories that match our input trajectory the most, we conclude that most of computations used to scan all the trajectories are unnecessary. Thus, our focus pivots towards the structuring of the historical trajectory search space in a manner that substantially reduces the necessary computational actions for each input scenario. In this section, we discuss the strategy to achieve this through the summarization of trajectories, a process which allows for efficient clustering while preserving essential geometric characteristics. Following this, we elaborate on our methodology for indexing the clusters to ensure resource-efficient access.

5.1. Summarizing and Clustering Trajectories

The aim behind clustering trajectories in the StratAlign system is not fundamentally to pinpoint meaningful clusters but to efficiently discard trajectories that are considerably divergent from the input trajectory. Hence, our preferred clustering strategy is one that (1) enables quick computations and (2) arranges the search space into spherical groups, mirroring the Euclidean nature of the trajectories. Considering employing the Dynamic Time Warping (DTW) distance metric for clustering could logically lead to the adoption of k-medoids clustering [14, 15]. Nonetheless, as discussed in Section 4, the time complexity of DTW is $O(n \times m)$. For the sake of simplicity of discussion, we assume that the average trajectory length is L. Therefore, the time complexity of DTW can be considered $O(L^2)$. Therefore, the time complexity of k-medoids clustering using DTW as the distance function of choice would be $O(N^2.L^2 + N^2.k)$ if we precompute a pairwise distance matrix between the trajectories, or $O(N.k.L^2)$ if we use an online approach, where N is the number of trajectories, k is the number of clusters, and L is the average trajectory size (see the Appendix for details on the analysis of these time complexities). To enhance efficiency while maintaining the potential to seamlessly integrate future trajectories with limited additional burden, we opt for the k-means clustering method, which is acknowledged for its heightened efficiency [14, 15]. Facilitating this pathway requires the transformation of trajectories into fixed-length feature vectors that retain their geometric attributes.

As discussed in Section 4, the trajectory can be defined as a sequence of locations $T = [l(e_1), l(e_2), \ldots, l(e_p)]$, where each location is represented as a point in a two-dimensional space (i.e., $l(e_i) = (x, y)$). We summarize each trajectory as a fixed-size feature vector with 5 elements: the mean x and y coordinates, the direction vector's x and y components, and the total length of the trajectory (number of events). Following is an explanation of each component:

1. Mean Point: The mean point of the trajectory is calculated by averaging the x and y coordinates across all the points in the sequence:
   $Mean_x = \frac{1}{p}\sum_{i=1}^{p} x_i,$
   $Mean_y = \frac{1}{p}\sum_{i=1}^{p} y_i,$
   where p is the total number of points in the trajectory.
2. Direction Vector: The overall direction of the trajectory is represented by a vector from the first to the last point, computed as:
   $Direction_x = x_p - x_1,$
   $Direction_y = y_p - y_1.$
3. Trajectory Length: The total length of the trajectory in terms of the number of events (p).

This representation provides a concise summary that captures some geometric aspects of the trajectory, although it might lose some details related to the internal structure, such as curves or repeated patterns within the trajectory. In Section 7, we experimentally verify the quality of this method in summarizing trajectories.

Given two trajectories, T_a and T_b, defined as sequences of 2-dimensional points:
$T_a = [l(e_1^a), l(e_2^a), \ldots, l(e_{p_a}^a)],$

$T_b = [l(e_1^b), l(e_2^b), \ldots, l(e_{p_b}^b)],$

where $(l(e_i^j) = (x_i^j, y_i^j))$, the comparison between the two trajectories is performed using a combined distance measure composed of three distinct components.

1. Mean Points Distance: The mean points of the two trajectories are first calculated:
   $\text{Mean}_{x_a} = \frac{1}{p_a}\sum_{i=1}^{p_a} x_i^a,$
   $\text{Mean}_{y_a} = \frac{1}{p_a}\sum_{i=1}^{p_a} y_i^a,$
   $\text{Mean}_{x_b} = \frac{1}{p_b}\sum_{i=1}^{p_b} x_i^b,$
   $\text{Mean}_{y_b} = \frac{1}{p_b}\sum_{i=1}^{p_b} y_i^b,$

   and their Euclidean distance is computed:
   $d_{\text{mean}} = \sqrt{\left(\text{Mean}_{x_a} - \text{Mean}_{x_b}\right)^2 + \left(\text{Mean}_{y_a} - \text{Mean}_{y_b}\right)^2}.$

2. Direction Vectors Distance: The direction vectors for both trajectories are:
   $v_a = \left(x_{p_a}^a - x_1^a, y_{p_a}^a - y_1^a\right)$, and $v_b = \left(x_{p_b}^b - x_1^b, y_{p_b}^b - y_1^b\right)$, and their cosine similarity is computed and converted to distance:
   $\text{sim}_{\text{direction}} = v_a \cdot v_b,$
   $d_{\text{direction}} = 1 - \text{sim}_{\text{direction}}.$

3. Trajectory Length Distance: The lengths of the trajectories, defined as the number of event sequences, are compared using the absolute difference:
   $d_{\text{length}} = |p_a - p_b|.$

The three distance components are normalized using min-max normalization and the total distance between the trajectories is computed as the weighted sum of the three components:
$d(T_a, T_b) = (d_{\text{normalized}} + d_{\text{direction\_normalized}} + d_{\text{length\_normalized}}) \,/\, 3.$
This combined distance measure takes into account the mean position, overall direction, and length of the trajectories, providing a comprehensive comparison that reflects their key geometric characteristics.

Given a set of trajectories $\mathcal{T} = \{\mathrm{T}_1, \mathrm{T}_2, \dots, \mathrm{T}_n\}$, the objective is to partition them into k clusters, such that the trajectories within each cluster are more similar to each other in terms of their geometric characteristics. The similarity between any pair of trajectories T_i and T_j is determined by the distance function $d(T_i, T_j)$ as defined previously. The k-means clustering with the customized distance function is executed as follows:

1. Initialization: We randomly choose k initial centroids from $\mathcal{T}$. Let $(\mathrm{C}_1, \mathrm{C}_2, \dots, \mathrm{C}_k)$ represent the centroids.
2. Assignment: We assign each trajectory T_i to the nearest centroid C_j based on the distance function $\mathrm{d}(\mathrm{T_i}, \mathrm{C_j})$. We form k clusters $(\mathcal{C}_1, \mathcal{C}_2, \dots, \mathcal{C}_k)$, where $\mathcal{C}_j = \{\mathrm{T_i} \mid \mathrm{d}(\mathrm{T_i}, \mathrm{C_j}) = \min_{l=1}^{k} \mathrm{d}\,(\mathrm{T_i}, \mathrm{C_l})\}$.

```
Algorithm 2: K-means Clustering for Trajectories
   Input: A set of trajectories $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$, number of clusters $k$
   Output: Clusters $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_k$
 1 Initialization: Choose $k$ initial centroids;
 2 repeat
 3     Assignment:;
 4     foreach $T_i \in \mathcal{T}$ do
 5         Assign $T_i$ to the nearest centroid $C_j$ based on $d(T_i, C_j)$;
 6     Update:;
 7     foreach cluster $\mathcal{C}_j$ do
 8         $Mean_{x_j} = \frac{1}{|\mathcal{C}_j|} \sum_{T_i \in \mathcal{C}_j} Mean_{x_i}$;
 9         $Mean_{y_j} = \frac{1}{|\mathcal{C}_j|} \sum_{T_i \in \mathcal{C}_j} Mean_{y_i}$;
10         $v_{jx} = \frac{1}{|\mathcal{C}_j|} \sum_{T_i \in \mathcal{C}_j} v_{ix}$;
11         $v_{jy} = \frac{1}{|\mathcal{C}_j|} \sum_{T_i \in \mathcal{C}_j} v_{iy}$;
12         $p_j = \frac{1}{|\mathcal{C}_j|} \sum_{T_i \in \mathcal{C}_j} p_i$;
13 until centroids no longer change;
14 return $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_k$
```

3. Update: We compute the new centroid for each cluster $\mathcal{C}_j$ by taking the mean of the geometric characteristics of the trajectories within the cluster:
   $\text{Mean}_{x_j} = \frac{1}{|\mathcal{C}_j|} \sum_{T_i \in \mathcal{C}_j} \text{Mean}_{x_i}$,
   $\text{Mean}_{y_j} = \frac{1}{|\mathcal{C}_j|} \sum_{T_i \in \mathcal{C}_j} \text{Mean}_{y_i}$,
   $v_{jx} = \frac{1}{|\mathcal{C}_j|} \sum_{T_i \in \mathcal{C}_j} v_{ix}$,
   $v_{jy} = \frac{1}{|\mathcal{C}_j|} \sum_{T_i \in \mathcal{C}_j} v_{iy}$,
   $p_j = \frac{1}{|\mathcal{C}_j|} \sum_{T_i \in \mathcal{C}_j} p_i$,
   where Mean_{x_i}, Mean_{y_i}, v_j, and p_i are the mean points, direction vector, and length of trajectory T_i, respectively.

4. Convergence Check: Steps 2 and 3 are repeated until the centroids no longer change, or some predefined criterion (e.g., a maximum number of iterations or a threshold on the change in within-cluster sum of squares) is met.

The final clusters $(\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_k)$ represent the partitioning of the trajectories into k groups that are geometrically similar according to the defined distance function. Algorithm 2 shows how the clusters are created.

```
Algorithm 3: Finding Top-k Similar Trajectories
   Data: Clusters $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_k$, Medoids $C_1, C_2, \ldots, C_k$, Input Trajectory $T$
   Result: Top-k similar trajectories to $T$
 1 for $i = 1$ to $k$ do
 2     Identify memory address of $\mathcal{C}_i$ and associate with $C_i$;
 3 for $i = 1$ to $k$ do
 4     Calculate DTW distance: $d(T, C_i)$;
 5 Retrieve clusters with the smallest DTW distances to $T$;
 6 for each retrieved cluster do
 7     Compute DTW distance between $T$ and each trajectory;
 8     Sort trajectories by DTW distance;
 9     Select top-$k$ similar trajectories;
10 return Top-$k$ similar trajectories;
```

5.2. Indexing Clusters and Searching for Most Similar Trajectories

StratAlign is allocated a specific memory budget for operation. It is presupposed that this memory budget is less than the cumulative size of the trajectories within our dataset. For the sake of subsequent discussion, we shall momentarily postulate that there exists a sufficient memory budget to accommodate all the trajectories (partitioned into clusters as delineated in the previous section). Subsequent to this assumption, we will explore the scenario wherein the memory budget is insufficient to cover the total size of the trajectories.

Given the clusters obtained through k-means clustering, $(\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_k)$, we select the medoids of each cluster $(\mathrm{C}_1, \mathrm{C}_2, \ldots, \mathrm{C}_\mathrm{k})$. We build an associative mapping to facilitate the retrieval of clusters similar to a given input trajectory, followed by the identification of the top-k similar trajectories within the retrieved cluster. Our solution proceeds as follows (discussed in Algorithm 3):

1. Memory Address Association: For each cluster $\mathcal{C}_i$, identify the memory address where the cluster's data is stored.
2. Medoid-Address Mapping: Create an associative structure (e.g., hash map) to store the relationship between the medoids and the corresponding memory addresses. For each cluster $\mathcal{C}_i$, the medoid C_i is used as the key, and the memory address of $\mathcal{C}_i$ is used as the value.
3. Similarity Comparison to Medoids: Given an input trajectory T, we compare it to the k medoids of the clusters using the Dynamic Time Warping (DTW) distance function. We calculate the DTW distances $(\{\mathrm{d}(\mathrm{T}, \mathrm{C}_1), \mathrm{d}(\mathrm{T}, \mathrm{C}_2), \ldots, \mathrm{d}(\mathrm{T}, \mathrm{C}_\mathrm{k})\})$.
4. Most Similar Cluster Retrieval: We identify the clusters with the smallest DTW distances to T. Then, we retrieve the memory addresses of the most similar clusters.
5. Comparison to Trajectories Within Clusters: We access the retrieved clusters and compute the DTW distance between T and each trajectory within the clusters. We sort the trajectories by the DTW distance, and select the top-k similar trajectories.

This multi-level approach significantly reduces the search space by eliminating most of the historical trajectories from the comparison with the input trajectory using the DTW distance. Indeed, we eliminate at least 91% of the search space on average using this approach and 98% in the best case. In Section 7, we show that this does not come at the expense of the quality of the output.

The previous discussion was based on the assumption that all the trajectories fit in memory. However, as discussed earlier, this is not a practical assumption either due to scalability requirements or monetary cost. Given a constrained memory budget denoted by "Mem," the necessity to manage large clusters $(\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_{k})$ motivates the employment of a caching strategy. Upon a request of a cluster as explained earlier, it is loaded from the disk to the cache if the cache is not full. If the cache is full, at least one cluster that currently resides in the cache must be evicted before loading the requested cluster into the cache. Given the immutable nature of the clusters during query time, this means that this cluster's space is simply freed. If the requested cluster is requested again in the future, assuming it was not evicted from the cache, it will be accessed from the cache without accessing disk. In StratAlign, we use the least recently used policy (LRU), where we maintain a record of the access sequence of clusters. Whenever a cluster is accessed (either read from the cache or loaded into the cache), we update its position in the queue. We then evict the cluster that was accessed least recently when space is required to be freed in the cache. The LRU caching strategy is particularly suited to this scenario due to its simplicity and effectiveness in handling access patterns where recently used clusters are likely to be accessed again in the near future. By prioritizing the retention of recently accessed clusters and evicting those accessed least recently, this approach minimizes the likelihood of cache misses, providing efficient access to the clusters. This strategic use of an LRU-based cache offers a robust solution to the challenge of managing large clusters within the constraint of the memory budget "Mem." By carefully calibrating the cache size and efficiently implementing the LRU policy, this approach ensures responsive retrieval of clusters, enabling effective similarity-based analysis of trajectories without exceeding the available memory resources.

6. Overcoming Arbitrary Video Offsets

We introduce a pipeline that leverages machine learning-driven computer vision techniques to automate the process of determining the temporal offset between timestamps of the events in the events dataset and the corresponding recorded footage. An example of such difference is shown in Figure 6. Our proposed approach is divided into an offline phase and an online phase. In the offline phase, we train a scoreboard detection model and preprocess our video database to associate each video file with a JSON file that stores the offset(s) between the video file and the clock time of randomly selected snapshots from the video file. In the online phase, given a timestamp of an event of interest, we retrieve the accurate video offset of the queried event by employing an automated iterative seeking approach to overcome the scenarios where there are disparities in the offset for the same video file and eventually return a video snippet that starts at the same time as the input event.

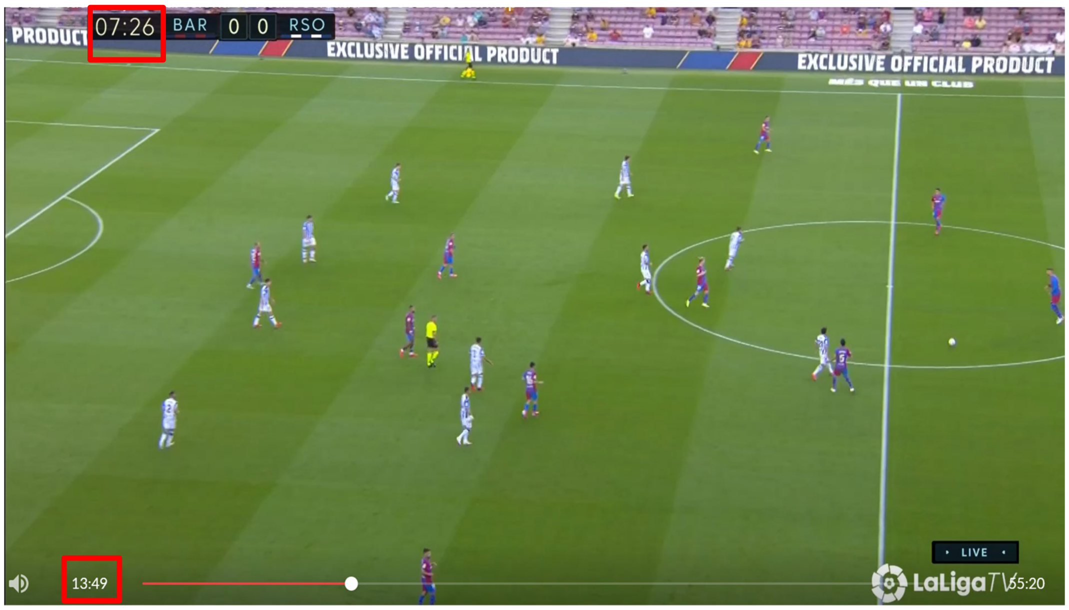


Figure 6: Example of difference in between the timestamp of events (top left) and the offset of the event starting in the video file (bottom left).

6.1. Offline Phase

Figure 7 shows the steps of the offline phase. Initially, a comprehensive iteration of the video repository is undertaken, and each video undergoes a series of operations, resulting in the creation of dedicated JSON files for each video instance. Within these JSON files, one or more entries are encoded, containing the timestamps extracted from the video frames (specifically, the scoreboard) as the key, paired with the corresponding calculated offset values. The aforementioned operations can be encapsulated in three fundamental components, each of which are explained in the subsequent sections.

Scoreboard Detection

To locate the temporal markers corresponding to football matches within video frames, the initial phase entails the recognition of the existing scoreboards in each frame. This task can be effectively accomplished through the utilization of an object detection model. The underlying concept of an object detection model is centered around its ability to learn an encoding from input images and subsequently map them to annotated ground truth during the training phase. These annotations consist of coordinates of bounding boxes specifying the location of the target object within the image, along with the corresponding class label denoting the object's category. In the deployment phase, this acquired encoding and mapping knowledge can be leveraged to predict the coordinates for bounding boxes surrounding each instance of our desired objects, should they be present within the frame. In the contemporary landscape of computer vision, many architectures have emerged to address object detection tasks. One of the most notable among these is the YOLOv8 [16] architecture. Our methodology includes the fine-tuning of the pre-trained YOLOv8 model (trained using the COCO dataset [17]) using a bespoke dataset specifically developed for this application. A principal obstacle encountered in our work was the scarcity of training data compatible with our problem scope. To address this, we manually assembled and annotated a dataset, designed to meet the unique requirements of our task.

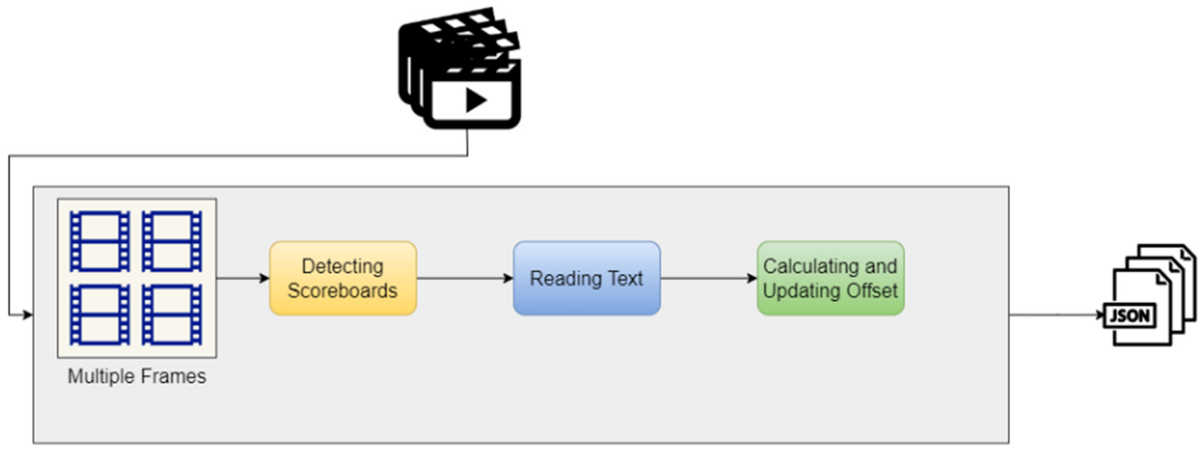


Figure 7: The pipeline of the offline phase.

The successful implementation of our method is critically dependent on the accurate identification of scoreboards. Thus, the dataset compilation required meticulous attention to various factors. One such factor is the heterogeneity in scoreboard designs employed globally, characterized by variations in color schemes, aspect ratios, typographies, clock positions, and the representation of added time. To accommodate these divergent designs, our dataset incorporates images of scoreboards from multiple football leagues, tournaments, and international championships. In our work, we annotated data from 32 different scoreboard layouts. This diverse inclusion aims to capture a comprehensive array of scoreboard designs, mitigating the risk of model overfitting. The finalized dataset comprises annotated images segregated into two scoreboard categories. The first category is designated for scoreboards that are predominantly located at one of the upper corners of the video frames. The second category is conceived for scoreboards that consume a larger frame area, providing additional information. Figure 8 shows examples of different snapshots that we annotated to fine-tune YOLOv8. The figure shows examples from both classes of scoreboards (smaller scoreboards located in one of the corners of the screen, or bigger scoreboards in the bottom center). To enhance the diversity and robustness of our image dataset, we employed a sequence of augmentation techniques. Initially, we introduce a random hue adjustment spanning a range of -25° to $+25^{\circ}$. This is followed by a random saturation modification ranging from -50% to $+50\%$. Subsequently, we applied a random brightness alteration, ranging from -25% to $+25\%$. Lastly, we implemented a random exposure adjustment, varying from -10% to $+10\%$. This approach allows us to enhance the dataset's variety and ensure its effectiveness across different phases of model development and evaluation.

The scoreboard detection step is crucial for finding the timestamp and significantly impacts the performance and accuracy of the pipeline. Given the resource-intensive nature of OCR models, precise identifi-

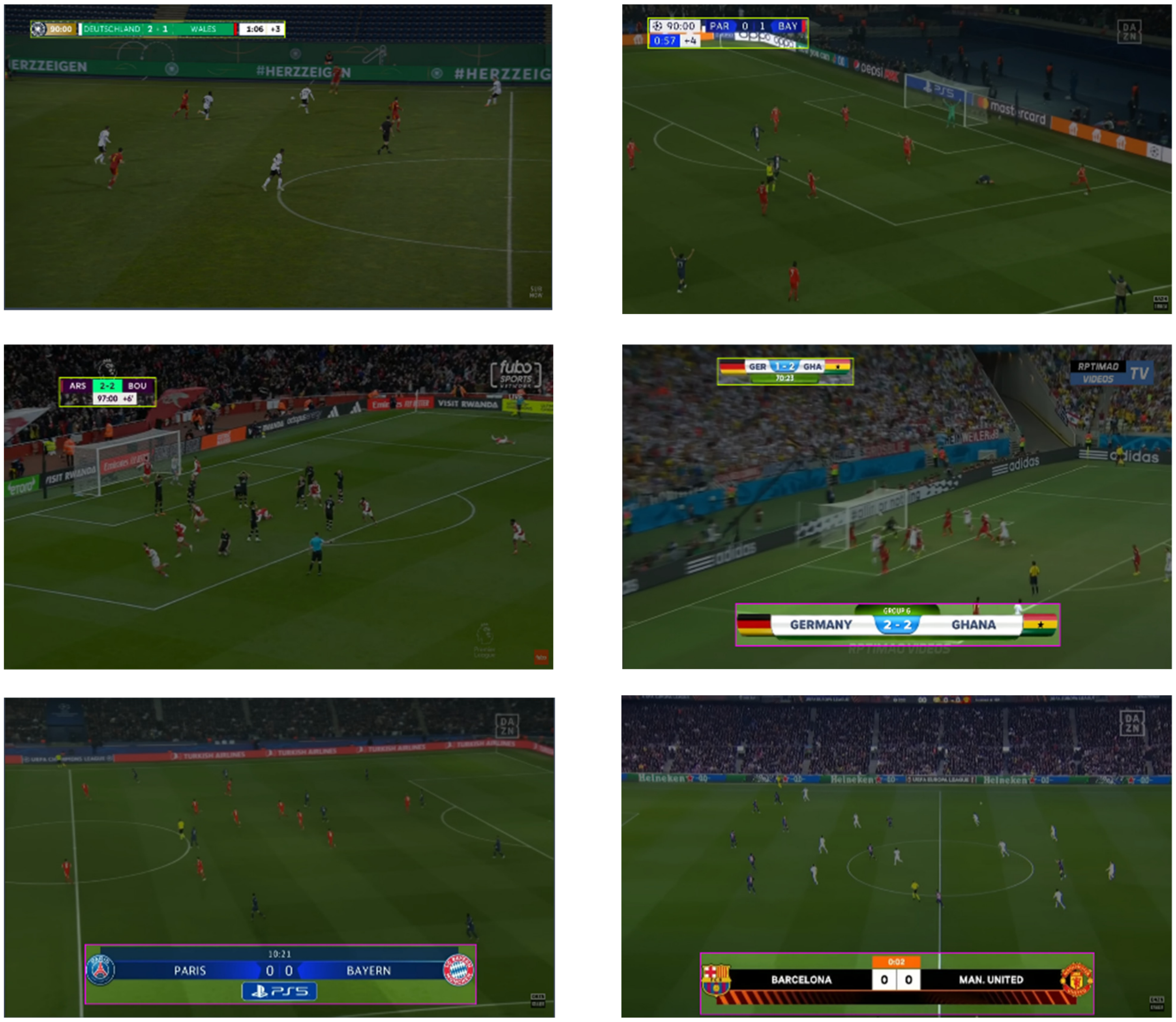


Figure 8: Examples of how we annotate our data to fine-tune YOLOv8 to detect scoreboards.

cation and isolation of scoreboards significantly reduces the input size of the OCR model [18]. This reduction in the input size contributes to the marked enhancement in performance, emphasizing the pivotal role of this detection phase in the overall framework. Upon the successful identification of a scoreboard within a frame, this localized section of the image is cropped and fed into the subsequent layer.

Reading text from the scoreboard

Subsequently, the following stage entails the extraction of textual content from the previously identified scoreboard region. For this purpose, an Optical Character Recognition (OCR) model is required. Within the realm of OCR, numerous methodologies exist, and deep learning techniques excel in terms of accuracy, performance, and efficiency. These methods typically entail the integration of an object detection model, responsible for locating text within an image, followed by a text recognition model, which encodes the output from the text detection model and translates it into its textual equivalent. Various architectures have been proposed for OCR, but in this project, we have chosen to utilize EasyOCR [18]. This choice is motivated by EasyOCR's remarkable accuracy and efficiency, its adaptability in accommodating different architectures as its submodules, and its compatibility with the other components of our pipeline. As the default configuration, EasyOCR employs the CRAFT [19] text detector for the purpose of identifying text snippets within an image. CRAFT is particularly renowned for its proficiency in effectively detecting text, even when it appears on curved or irregular surfaces. The output generated by the text detector submodule is subsequently channeled into the recognition module. This module, also set as the default configuration, employs a ResNet [20] feature extractor, followed by several layers of bidirectional Long Short-Term Memory (LSTM) networks. These LSTM layers facilitate the learning of patterns within character sequences. Moreover, the recognition module employs the CTC (Connectionist Temporal Classification) loss function, which is specifically designed to handle situations where the alignments between sequences are not known in advance. This feature is particularly advantageous for recognizing text in images with varying layouts and orientations.

Therefore, we leverage EasyOCR to effectively interpret text from the scoreboard. This tool provides us with the extracted textual fragments, accompanied by the bounding box coordinates of the text and a confidence score. The confidence score serves as a valuable gauge of the result reliability, influencing our decision-making process concerning the onward transmission or the omission of the output to subsequent layers. It is noteworthy that while EasyOCR showcases commendable performance, it does not attain flawless accuracy. Occasional inaccuracies in character detection, particularly for specific fonts, have been observed. However, the consistency of these inaccuracies across the majority of predictions facilitates the formulation of heuristic strategies to mitigate their impact.

Heuristic rules

The concluding layer within the offline phase entails the extraction of timestamps from all textual fragments obtained through the preceding OCR model subprocess, followed by the calculation of temporal offsets. To accomplish this, the extracted text fragments go through a specified regular expression pattern designed to distinguish timestamps and convert to milliseconds such that they have the same unit as the timestamp in the events files. In order to mitigate potential inaccuracies introduced by the OCR model, complementary heuristic rules are incorporated into the original regular expressions. A common instance of inaccuracy is relevant to the OCR model's challenge in identifying the colon character separating minutes and seconds on the clock. Frequently, the OCR model misrecognizes the colon character as a semicolon, dot, pipe, or even the letter "l". Given the certainty that these alternative characters do not feature in the clock's representation, their presence signals an erroneous prediction of the colon. Consequently, these alternative characters are systematically replaced with the appropriate colon character. Another notable inaccuracy derives from the OCR model's occasional detection of excessive empty spaces between the digits on the clock. Since such gaps have no place within the clock's composition, the identification of empty spaces signifies a misstep in the OCR model's prediction. These extra spaces are therefore removed from the textual data. Although our experimental investigations have yet

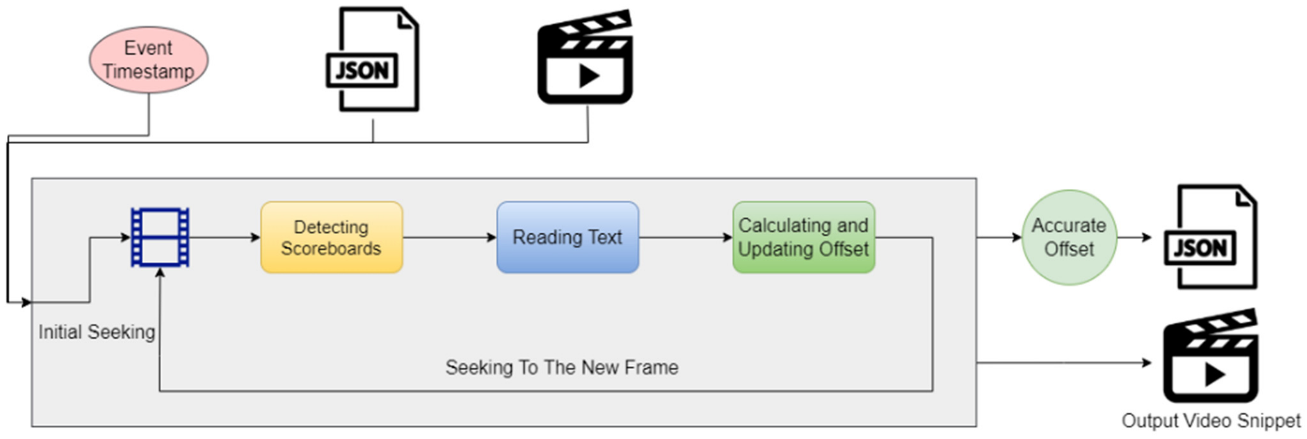


Figure 9: The pipeline of the online phase.

to reveal additional inaccuracies, the proposed approach remains adaptable to novel inaccuracies. Should such disparities occur, modifying the regular expression pattern through the inclusion of additional heuristic rules is straightforward to implement.

6.2. Online Phase

The aforementioned offset should theoretically remain constant throughout the video's entirety. However, empirical observations reveal the potential emergence of discontinuities, attributable to factors such as commercial advertisements, variances in TV production, etc. These circumstances introduce the potential for deviations in the temporal offset across the footage's duration. One way to address this challenge is by exhaustively increasing the number of snapshots processed during the offline phase. However, this will result in a significantly slower execution time and excessive use of computing resources to accurately detect what is usually an uncommon disparity. Therefore, we opt to address this issue in the online phase while StratAlign is answering a user's query through the execution of a supplementary operation. The pipeline of this operation is shown in Figure 9. During this supplementary phase, the initial offset, determined during the offline phase, serves as the base temporal anchor. Subsequently, an accurate offset is determined via an iterative process, explained by the algorithm denoted as Algorithm 4. When the offset is first calculated, a sanitary step is applied to check the clock at the calculated offset. If the clock in the snapshot matches the input event's timestamp, StratAlign trims the input video to output the video snippet starting with the input event. If there exists a discrepancy between the clock and the event's timestamp, a fresh offset is determined, coupled with the incorporation of a new record in the JSON file aligned with the input video. This iteration continues with successive verifications until a match is identified. Empirical evaluations confirm the efficiency of this approach in approximating the accurate temporal offset. Once the refined offset corresponding to the desired timestamp is calculated, it is added as an entry into the relevant JSON file, associated with the respective video file, thereby establishing a reference for subsequent utilization. The trimmed video snippet starting at the input event is also output.

```
Algorithm 4: Iterative Offset Update
   Input: Timestamp $T$, Initial Offset $O_0$
   Output: Final Offset $O$
 1 Initialize $O \leftarrow O_0$;
 2 repeat
 3     Seek the video to $T + O$;
 4     Detect scoreboard;
 5     Read text;
 6     Calculate new offset $O_{new}$;
 7     $O \leftarrow O_{new}$;
 8 until $O$ no longer changes;
 9 Add the new offset as an entry to the JSON file;
10 return $O$;
```

7. Experimental Evaluation

7.1. Dataset

In this paper, we use the Statsbomb open data version of August 19, 2023[6]. We also use the 2021-2022 and the 2022-2023 seasons data from the Premier League, which was provided by Statsbomb. In total, our dataset includes a total of over 14 million events, from which we extracted over 445 thousand trajectories.

7.2. Compute Environment

We run our experiments on a 32-core Intel(R) Xeon(R) CPU @ 2.20GHz computer with 32 GB in RAM and 1 TB of disk space running Debian GNU/Linux 11. We implemented StratAlign in RUST and Python. It is worth noting that all of the experiments in this section are multi-threaded to best utilize the available resources in our computing environment.
In the following experiments, unless otherwise stated, the default set of parameters in StratAlign is as follows: We cluster the search space into 500 clusters. We retrieve the 30 clusters whose medoids are most similar to the input trajectory for scanning. We assign a memory budget that is sufficient to store all the trajectories.

7.3. Efficiency

In this section, we compare StratAlign to the in-memory scanning approach in terms of execution time. We randomly-sample 10% of different lengths trajectories of for testing. We report the average execution time. Figure 10 shows the results of this experiment. The figure illustrates that StratAlign demonstrates substantial improvements in execution time compared to the Scanning approach; notably, the execution time for the Scanning approach exceeds that of StratAlign by an order of magnitude for trajetories comprising more than 10 events. The error bars in the figure represent the standard deviation

[6] https://github.com/statsbomb/open-data/tree/0067cae166a56aa80b2ef18f61e16158d6a7359a

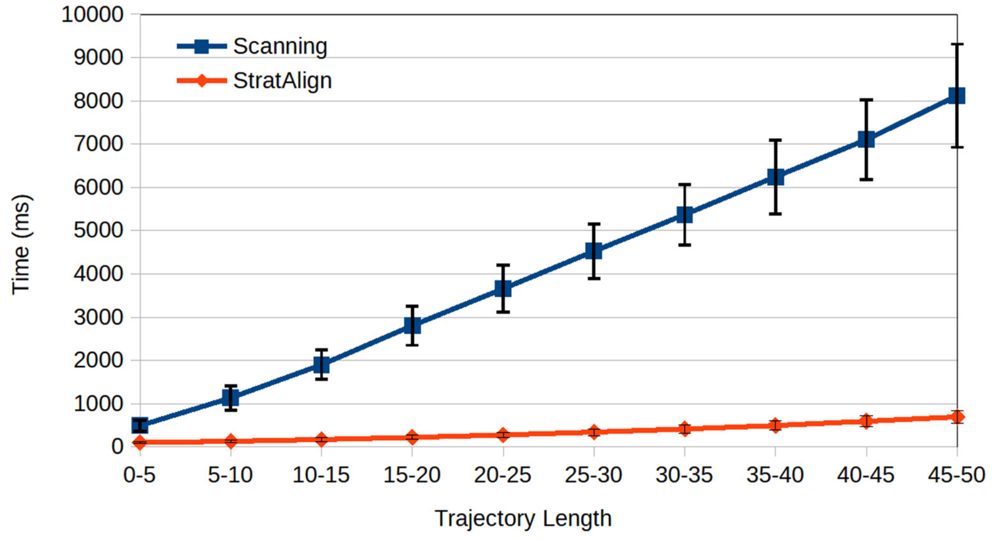


Figure 10: The execution time to retrieve the most similar trajectory to a given input trajectory for StratAlign and the in-memory Scanning approach.

values across multiple experiments conducted for trajectories of identical length. These results provide empirical evidence that StratAlign offers more consistent and predictable performance.

In order to assess the efficacy of our indexing methodology within resource-constrained settings, we subject StratAlign to an evaluation operating on a restricted memory allocation. Specifically, this memory constraint allows only a fraction of the total trajectories to be accommodated in the system's memory. The parameters in this experiment are set to their default values. We sample 10% of the trajectories of the median length (25-30) for testing. Figure 11 depicts the outcome of this experimental setup, in which StratAlign is evaluated using a range of memory budgets, starting as low as 10% of the total size required for storing all trajectories, and incrementing by 10% up to full capacity (100%). The results substantiate that even under stringent resource limitations—such as a 10% memory allocation—StratAlign maintains a latency that comfortably resides within real-time responsiveness. As anticipated, an increment in cache size correlates almost linearly with a decrease in the average execution time.

7.4. Quality of Output Trajectories

To comprehensively assess the quality of our algorithm for finding the most similar trajectory, we use the following two metrics: Relative Error, and Top-k Accuracy.

Given an input trajectory T, let T_{true} be the true most similar trajectory as determined by Dynamic Time Warping (DTW), and let T' represent the output most similar trajectory found by StratAlign. The Relative Error (RERE) is then defined as: $\text{Relative Error(RE)} = \left|\frac{\text{DTW}(T,T') - \text{DTW}(T,T_{\text{true}})}{\text{DTW}(T,T_{\text{true}})}\right|$. Here, $DTW(T,T')$ denotes the DTW distance between the input trajectory T and the approximated trajectory T'. This metric serves to normalize the absolute approximation error in terms of the actual DTW distance to T_{true}. A

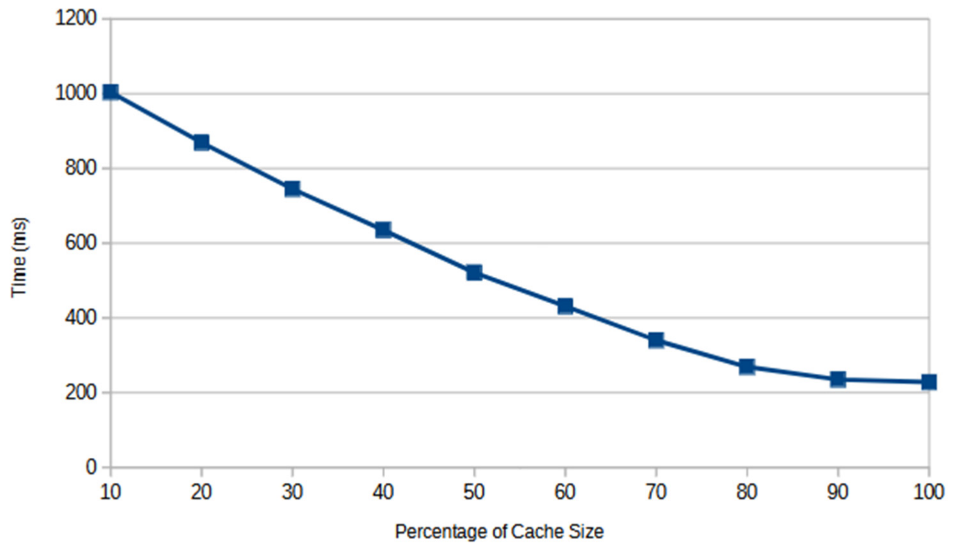


Figure 11: The effect of changing the cache size to fit only a percentage of the total size of trajectories.

Relative Error of 0 would signify a perfect approximation, while values greater than 0 represent varying levels of approximation inaccuracies. This metric provides a nuanced evaluation of the approximation error by scaling it with the true DTW distance, thereby yielding a more contextual understanding of the algorithm's efficacy in approximating the most similar trajectory.

Top-k accuracy is employed to quantify how often the trajectory identified by StratAlign appears within the top-k most similar trajectories according to the ground truth, which is established using Dynamic Time Warping (DTW) as the distance function. Given an input trajectory T, the ground truth method computes the DTW distances between T and all other trajectories in the dataset. These distances are then sorted in ascending order to identify the true top-k most similar trajectories to T. Our approach is also run on the same input trajectory T to find a trajectory T', which is purported to be the most similar to T. The aim is to investigate whether T' lies within the top-k list of the ground truth. For each input trajectory, a Boolean variable is assigned a value of 1 if the trajectory T' obtained from our approach is present within the top-k list obtained from the ground truth method. Otherwise, the variable is assigned a value of 0. The Top-k Accuracy is then computed as follows: $\text{Top-k Accuracy} = \frac{\text{Sum of Boolean variables}}{n} \times 100$. Here, n is the total number of input trajectories for which the experiments are run.

Table 1 enumerates the quality metrics for StratAlign across varying trajectory lengths. The results indicate that the relative error remains bounded by 0.05, signifying that the DTW distance between the input and output trajectories deviates by no more than 5% when compared to the distance between the input trajectory and its true most similar counterpart from historical data. The table further reveals that, in 88% to 98% of the test cases, the output trajectory is among the true top-5 most similar trajectories relative to the input. In 96% to almost 100% of the test cases, the output trajectory resides within the true top-10 most similar trajectories. These findings substantiate that our approach, while designed for efficiency, does not compromise the quality of the output at the expense of enhanced performance.

Table 1: The quality scores for StratAlign for different trajectory lengths.

Trajectory Length	Relative Error	Top 5% Accuracy	Top 10% Accuracy
5-10	0.05	97.3	99.19
10-15	0.03	97.67	99.66
15-20	0.03	97.16	99.77
20-25	0.04	95.34	99.51
25-30	0.04	94.04	99.01
30-35	0.04	91.24	98.40
35-40	0.04	89.64	97.57
40-45	0.04	89.28	97.93
45-50	0.04	87.88	97.74

7.5. Ablation Study

In this section, we evaluate the effects of changing two parameters in StratAlign: (1) the number of clusters that we group trajectories into, and (2) the number of clusters that are most similar to an input trajectory, which are retrieved for scanning to find the top-k similar trajectories.

In this experiment, we focus on ascertaining the impact of varying the number of clusters utilized to partition the trajectory search space. Operating under the default parameters mentioned in 7.2, while changing only the number of clusters created. We also run the experiment using the median trajectory length (25-30). Figure 12 presents the implications of modifying the cluster count on both the average time taken to identify the most analogous trajectory to a given input and on the top 5/10% accuracy metrics. The data reveals an inverse relationship between the number of clusters and the execution time; a lower count of clusters results in an elongated execution time. This extension in time can be attributed to the constant number of clusters retrieved for scanning, leading to a larger subset of trajectories to be scanned. However, this enlarged scope for comparison enhances the likelihood of uncovering higher quality matches, as evidenced by the improved accuracy scores. Conversely, escalating the number of clusters engenders a reduction in both the computational time and the quality metrics.

Next, we evaluate the effect of changing the number of retrieved clusters for scanning. Recall that the default setting retrieves 30 clusters. Figure 13 shows the efficiency results of this experiment. As anticipated, increasing the number of retrieved clusters results in a larger set of trajectories subjected to comparison with the input trajectories via Dynamic Time Warping (DTW) distance, thereby leading to an extended execution time. Nonetheless, the execution time for the longest considered trajectory length remains below 1.2 seconds, a duration still within the realm of real-time responsiveness.

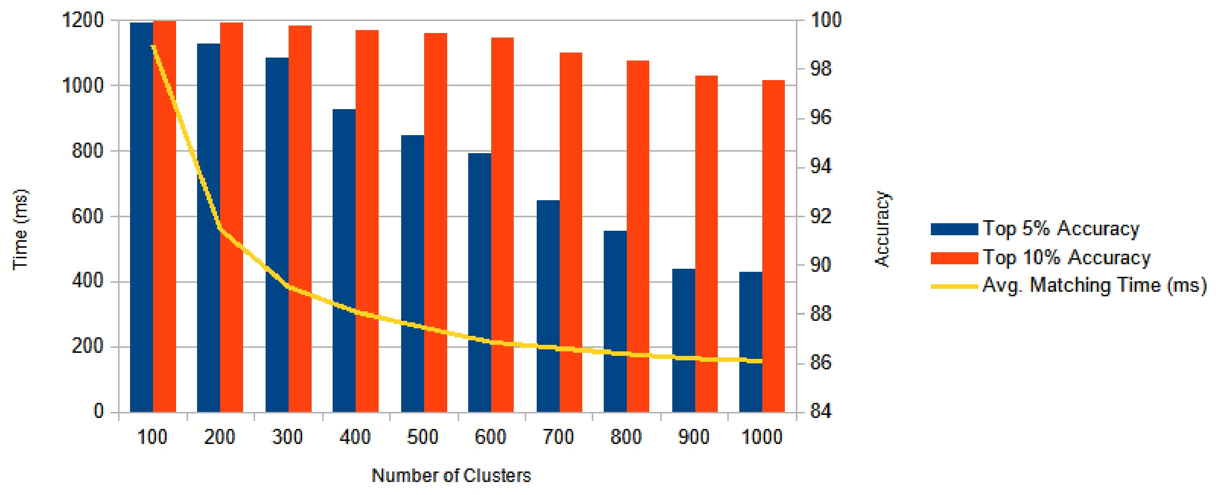


Figure 12: The trade-off imposed by different choices of number of clusters to partition the search space.

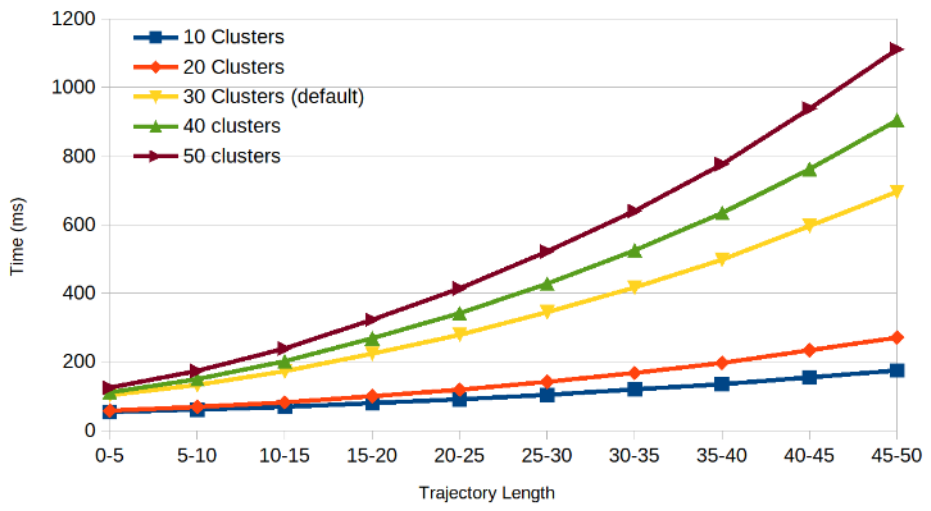


Figure 13: The execution time of StratAlign using different number of retrieved clusters to perform scanning on.

To contextualize the findings of the preceding experiment, we further examine the quality metrics of the output as a function of the number of retrieved clusters. Figure 14 shows the average relative error associated with different quantities of retrieved clusters. As anticipated, there exists an inverse relationship between the number of clusters retrieved and the relative error: fewer clusters result in elevated relative error, while more clusters correspond to diminished relative error. Notably, the decrement in

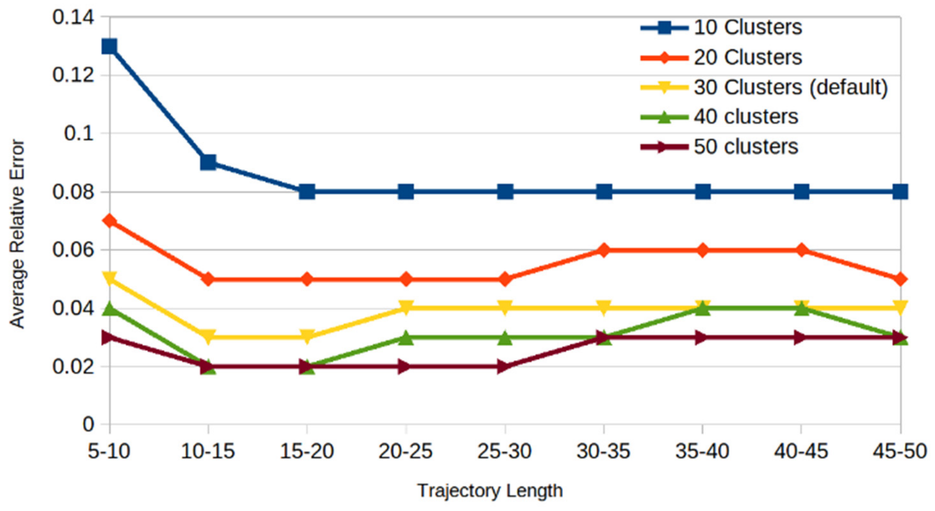


Figure 14: Average relative error of the output of StratAlign for varying trajectory lengths using different numbers of retrieved clusters.

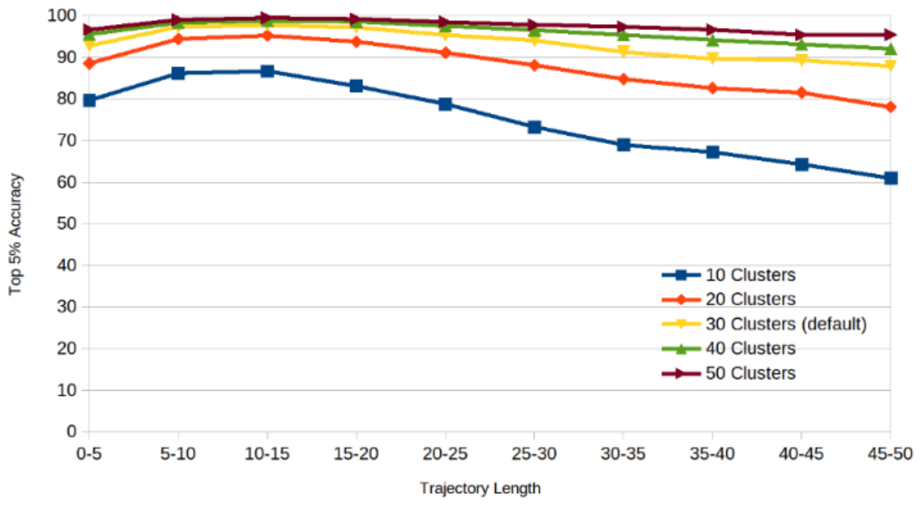


Figure 15: The Top 5% accuracy of the output of StratAlign for varying trajectory lengths using different numbers of retrieved clusters.

relative error when transitioning from 30 to 40 and 50 clusters is less pronounced compared to the transition from 10 to 20 and 30 clusters. This observation validates our selection of 30 clusters for retrieval, as it represents a reasonable trade-off between computational efficiency and reduced relative error. Corroborating observations are evident in Figure 15 and Figure 16, which delineate the Top 5% and Top 10% accuracy metrics, respectively. A direct relationship is observed between the number of retrieved clusters and the ensuing accuracy: increasing the number of clusters correspondingly elevates the accuracy. However, the difference in accuracy between 30, 40, and 50 clusters is notably less substantial than the discrepancy observed when comparing 10 and 20 clusters.

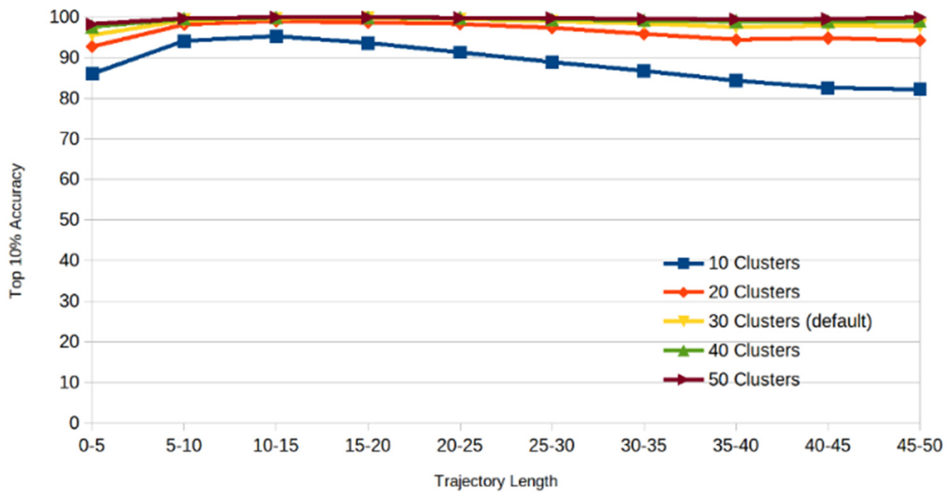


Figure 16: The Top 10% accuracy of the output of StratAlign for varying trajectory lengths using different numbers of retrieved clusters.

7.6. Evaluating Video Offset Detection

As explained in Section 6.1, we manually annotate our training dataset using two classes: One class refers to the scoreboard commonly found at one of the two top corners of the frames. This variant of scoreboard is annotated as “scoreboard” class in our dataset. The other class is intended for the scoreboards that occupy a larger portion of the frame, convey more information, and are typically presented at the beginning of each half, or after goals are scored. We referred to this type of scoreboard as “scoreboard_big” class in our dataset. After applying the augmentation techniques discussed in Section 6.1, our dataset expanded to comprise a total of 1,291 images. These images were then partitioned into distinct sets for training, validation, and testing, with 1,128 images designated for training, 106 for validation, and 57 for testing, respectively.

As previously highlighted, the scoreboard detection subprocess represents the core foundation of the entire pipeline. While we leveraged an off-the-shelf OCR model and successfully addressed its limitations through the introduction of heuristic rules, it is vital that the input data fed into the OCR model maintains a high degree of accuracy. Consequently, we placed significant emphasis on upholding the quality of the object detection model and conducted an exhaustive assessment of its performance. Following, we describe the metrics employed to evaluate the quality of the object detection subprocess.

Intersection over Union (IoU) is a commonly employed metric for the evaluation of object detection and instance segmentation algorithms. Given a ground-truth bounding box G and a predicted bounding box P, IoU is defined as the ratio of the area of their intersection to the area of their union. Mathematically, the IoU can be expressed as follows: $IoU(G, P) = \frac{|G \cap P|}{|G \cup P|}$. Here, $|G \cap P|$ denotes the area of intersection between G and P, and $|G \cup P|$ represents the area of union between the two. The intersection area is obtained by calculating the overlapping region between G and P, while the union area is computed as the sum of the areas of G and P minus their intersection: $|G \cup P| = |G| + |P| - |G \cap P|$. The IoU metric ranges from 0 to 1, where a value of 0 indicates no overlap and 1 signifies a perfect match between the predicted and ground-truth bounding boxes. In many applications, an IoU threshold is specified to determine whether a prediction is considered a true positive (TP), false positive (FP), or false negative (FN). In our experiments, IoU greater than 0.5 is used as a criterion for successful detection.

Precision stands as the ratio of true positive predictions over the combined total of true positive and false positive predictions. An elevated precision metric signifies that the model produces fewer erroneous positive predictions. Notably, it is important to recognize that increasing the Intersection over Union (IoU) threshold concurrently decreases the occurrence of false positive predictions, thereby resulting in a higher precision score. Precision is calculated as follows: $Precision = \frac{TP}{TP+FP}$, where TN stands for true negative. Recall on the other hand, is the ratio of true positive predictions over the summation of true positive and false negative predictions. A higher recall metric signifies that the model produces fewer erroneous negative predictions. It is essential to acknowledge that increasing the IoU threshold can lead to a greater number of false positive predictions and subsequently results in a lower recall score. Recall is calculated as follows: $Recall = \frac{TP}{TP+FN}$.

Average Precision (AP) incorporates the trade-off between precision and recall and considers both false positives and false negatives. Average precision can be conceptualized as the area under the precision-recall curve. It is important to note that average precision is computed independently for each of the classes (scoreboard and scoreboard_big) within the dataset. The average precision is calculated as follows: $\mathrm{AP} = \int_0^1 p(r)\ dr$. However, in a discrete setting, this continuous integral is approximated by a summation over specific recall values at which the precision changes: $\mathrm{AP} = \sum_{r \in R} \Delta r \times p(r)$, where $p(r)$ represents precision at a particular recall level r, and Δr indicates the change in recall. There are two commonly utilized variants of average precision: (1) Average precision at IoU 0.5 (AP50), which calculates the average precision with a specific IoU threshold of 0.5. It assesses the model's ability to make predictions with a moderate level of overlap with ground truth bounding boxes, and (2) average precision at IoU ranging from 0.5 to 0.95 (AP50-95), where the mean of average precisions across a range of IoU thresholds, typically from 0.5 to 0.95 is calculated (in intervals of 0.05). AP50-95 serves as a more precise evaluation metric, as it indicates the model's capability to predict objects with a high degree of overlap (IoU) with the ground truth bounding boxes, across a spectrum of IoU values. A high AP50-95 value indicates that the model consistently predicts objects with a high IoU score, implying greater accuracy in localization and prediction.

Table 2 presents the performance metrics of our scoreboard detection methodology. The first row shows the results of detection of the small scoreboard that is usually on one of the corners of the screen. The second row shows the results of detection of the big scoreboard that is usually shown at the beginning of each period of the game or when goals are scored. The third row shows the average results across the two classes. The results indicate exemplary quality in identifying smaller scoreboards, as evidenced by near-optimal values for precision, recall, and AP50. Furthermore, the AP50-95 metric substantiates the robustness of our approach, demonstrating commendable performance even under more stringent IoU thresholds. For the scoreboard_big class, our analysis reveals an elevated precision metric, albeit at the expense of reduced recall. This observation is principally attributed to the class's underrepresentation in the training dataset. Despite this limitation, we consider the obtained scores to be acceptable. The infrequent occurrence of large scoreboards in practical applications, combined with our iterative seek-and-update strategy during the online phase, significantly mitigates the risks associated with occasional non-detection of the scoreboard.

Table 2: The quality results of the scoreboard detection model.

Class	Precision	Recall	AP50	AP50-95
Scoreboard	0.954	1	0.977	0.831
Scoreboard_big	0.993	0.75	0.756	0.618
Both Classes	0.973	0.875	0.866	0.724

Regarding efficiency during the online phase, we run StratAlign using 250 queries. The average latency to calculate the offset for an input event is 1.5 seconds (standard deviation is 0.8 seconds). On average, it takes StratAlign approximately 400 milliseconds to store the video snippet on disk.

8. Conclusion

This paper presented StratAlign, an innovative system aimed at filling a critical gap in the field of sports analytics by incorporating historical events data into real-time strategic analysis for football games. Through the use of a dynamic time warping (DTW)-based distance function, StratAlign offers a more nuanced and robust mechanism for trajectory comparison. The system's clustering approach effectively prunes the search space, optimizing computational performance. Additionally, StratAlign has been designed with scalability as a core consideration. Utilizing a disk-based hash index and memory-eviction strategies, the system efficiently operates within restricted computational resources. We have also addressed the non-trivial problem of timestamp alignment with actual video footage, employing computer vision techniques to resolve this challenge. Our experimental evaluations substantiate StratAlign's efficiency, demonstrating that it is capable of retrieving relevant ball trajectories in a substantially shorter time frame as compared to existing baseline solution. This makes StratAlign not only an effective tool but also a cost-efficient option for real-time strategic decision-making in football games. In the future, we are exploring the possibility of incorporating the players situational variables to provide even more contextually enriched insights for strategic planning. Moreover, we are also investigating how we can extend StratAlign's capabilities to other sports.

References

[1] Statsbomb, "IQLive," [Online]. Available: https://statsbomb.com/what-we-do/iq-live/.

[2] Stats Perform, "Proportal," [Online]. Available: https://www.statsperform.com/team-performance/football-performance/match-analysis/proportal/.

[3] L. Pappalardo, P. Cintia, A. Rossi, E. Massucco, P. Ferragina, D. Pedreschi and F. Giannotti, "A public data set of spatio-temporal match events in soccer competitions," *Scientific data,* vol. 6, no. 1, 2019.

[4] Statsbomb, "StatsBomb Open Data," [Online]. Available: https://github.com/statsbomb/open-data.

[5] J. Gray and F. Putzolu, "The 5 minute rule for trading memory for disc accesses and the 10 byte rule for trading memory for CPU time," in *Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, 1987.

[6] J. Gray and G. Graefe, "The five-minute rule ten years later, and other computer storage rules of thumb," *ACM Sigmod Record,* vol. 26, no. 4, 1997.

[7] G. Graefe, "The five-minute rule twenty years later, and how flash memory changes the rules," in *Proceedings of the 3rd international workshop on Data management on new hardware*, 2007.

[8] H. Sakoe and S. Chiba, "Dynamic programming algorithm optimization for spoken word recognition," *IEEE transactions on acoustics, speech, and signal processing,* vol. 26, no. 1, 1987.

[9] Amazon, "AWS Pricing," [Online]. Available: https://aws.amazon.com/pricing/.

[10] Google, "Google Cloud Pricing," [Online]. Available: https://cloud.google.com/pricing/.

[11] Microsoft, "Azure Pricing," [Online]. Available: https://azure.microsoft.com/en-us/pricing/.

[12] D. J. Berndt and J. Clifford, "Using dynamic time warping to find patterns in time series," in *Proceedings of the 3rd international conference on knowledge discovery and data mining*, 1994.

[13] E. Keogh and C. A. Ratanamahatana, "Exact indexing of dynamic time warping," *Knowledge and information systems,* vol. 7, 2005.

[14] A. K. Jain, M. N. Murty and P. J. Flynn, "Data clustering: a review," *ACM computing surveys (CSUR,* vol. 31, no. 3, 1999.

[15] R. Xu and D. Wunsch, "Survey of clustering algorithms," *IEEE Transactions on neural networks,* vol. 16, no. 3, 2005.

[16] G. Jocher, A. Chaurasia and J. Qiu, "Yltralytics YOLOv8," 2023.

[17] T.-Y. Lin, M. Mair, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollar and C. L. Zitnick, "Microsoft coco: Common objects in context," 2014.

[18] Jaided AI, "EASYOCR," [Online]. Available: https://github.com/JaidedAI/EasyOCR.

[19] Y. Baek, B. Lee, D. han, S. Yun and H. Lee, "Character region awareness for text detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2019.

[20] K. He, X. Zhang, S. Ren and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016.

Appendix

The Time Complexities of the k-medoids Approaches

With Pairwise Distance Matrix:

1. Initialization: Select k data points as initial medoids.
2. Calculate Pairwise Distance Matrix:

Compute the distance between all pairs of trajectories, resulting in an $N \times N$ matrix.
Time complexity: $O(N^2 . L^2)$, where N is the number of trajectories, and L is the average length of the trajectories.

3. Cluster Assignment:

Assign each trajectory to the nearest medoid using the precomputed distance matrix.
Time complexity: $O(N . k)$.

4. Medoid Update:

For each cluster, test each data point to see if it would be a better medoid.
Compute the total distance to all other points in the cluster using the distance matrix.
Replace the medoid if a better one is found.
Time complexity: $O(N^2 . k)$.

5. Convergence Check: Repeat steps 3 and 4 until the medoids do not change, or the change is below a threshold.

Overall Complexity: $O(N^2 . L^2 + N^2 . k)$ for one iteration.

Without Pairwise Distance Matrix (Online Approach):

1. Initialization: Select k data points randomly or using some heuristic as initial medoids.
2. Sequential Processing:
    a. Calculate Distance to Medoids: Compute the distance from the current trajectory to each medoid (using, e.g., DTW). Time complexity: $O(k . L^2)$.
    b. Assign Cluster: Assign the current trajectory to the nearest medoid.
    c. Check for Medoid Update: Test if the current trajectory is a better medoid for its cluster by maintaining a running sum of distances.
    d. Update Medoid if Necessary: Replace the current cluster's medoid if a better one is found.
    e. Repeat: Continue with the next trajectory.

Overall time complexity for this step: $O(N . k . L^2)$.

3. Convergence Check: You may run through the dataset multiple times or include logic to detect if changes to medoids are below a threshold.

Overall Complexity: $O(N . k . L^2)$ for each pass through the data.

The precomputed matrix approach may offer some efficiency in subsequent iterations of the algorithm since the distances are already computed for smaller datasets. However, for larger datasets, this advantage is often outweighed by the initial computation and storage costs. In fact, using our dataset, we extract 445 thousand trajectories, approximately. Assuming the distance between any pair of trajectories is stored in a 32-bit float, this would require $(445K)^2 \times 32$ bytes of storage space, or 6 TB.