Information Fusion 101 (2024) 101979

Contents lists available at ScienceDirect

Information Fusion

journal homepage: www.elsevier.com/locate/inffus

Full Length Article

SWINN: Efficient nearest neighbor search in sliding windows using graphs

Saulo Martiello Mastelini [a,*], Bruno Veloso [b,d], Max Halford [c], André Carlos Ponce de Leon Ferreira de Carvalho [a], João Gama [b,d]

[a] Institute of Mathematics and Computer Science - University of São Paulo, São Carlos, Brazil
[b] INESC TEC, Porto, Portugal
[c] Carbonfact, Paris, France
[d] Faculty of Economics - University of Porto, Porto, Portugal

ARTICLE INFO

Keywords:
Nearest neighbor search
Sliding window
FIFO
Proximity query
Graph

ABSTRACT

Nearest neighbor search (NNS) is one of the main concerns in data stream applications since similarity queries can be used in multiple scenarios. Online NNS is usually performed on a sliding window by lazily scanning every element currently stored in the window. This paper proposes Sliding Window-based Incremental Nearest Neighbors (SWINN), a graph-based online search index algorithm for speeding up NNS in potentially never-ending and dynamic data stream tasks. Our proposal broadens the application of online NNS-based solutions, as even moderately large data buffers become impractical to handle when a naive NNS strategy is selected. SWINN enables efficient handling of large data buffers by using an incremental strategy to build and update a search graph supporting any distance metric. Vertices can be added and removed from the search graph. To keep the graph reliable for search queries, lightweight graph maintenance routines are run. According to experimental results, SWINN is significantly faster than performing a naive complete scan of the data buffer while keeping competitive search recall values. We also apply SWINN to online classification and regression tasks and show that our proposal is effective against popular online machine learning algorithms.

1. Introduction

Nearest Neighbor Search (NNS) is essential to many computer science applications, including image recognition, data compression, anomaly detection, robotics, databases, and computational linguistics [1–3]. The notion of proximity can be extended to multiple interpretations and dealt with differently by distinct research communities. Proximity is also correlated to the concept of learning, and it is fundamental for multiple machine learning (ML) algorithms. Recent works have confirmed the importance of NNS to ML, such as [4], which suggested a link between Deep Neural Networks and NNS.

Nonetheless, using a naive approach for dealing with NNS can rapidly become impractical, even with moderately sized datasets, e.g., a brute-force strategy or a complete linear scan (LS) of the data. Therefore, in many applications, more efficient NNS approaches are needed. ML research leverages discoveries from other research areas to enhance NNS, thus enabling large-scale computing [3,5–8]. The most usual strategy is to devise auxiliary data structures, referred to as search indices, for which performing NNS is faster than relying on an LS of the data. Such search indices may provide either exact or approximate search results. A reference tool for NNS benchmarking was introduced in [1], where multiple NNS techniques are compared.

In online learning scenarios, data arrives continuously and might change its generative distribution over time. Typically, online k-NN algorithms keep a buffer of the most recent instances where they perform the NNS. This data buffer is updated using a First in, First out (FIFO) strategy, i.e., a sliding window. To the best of our knowledge, search via LS is the prevailing strategy in streaming applications, with the user controlling the data buffer length. Hence, considering the buffer length L, the cost to perform a single query is $O(L)$. On the other hand, search queries are exact. Reducing the cost of an LS could be helpful even if it reaches approximate results.

In this study, we want to stress the relevance of k-NN-based algorithms in streaming applications [9]. Although k-NN is a straightforward lazy algorithm, it poses a strong baseline or even the best ML algorithm for multiple tasks. Moreover, k-NN is also a robust learning algorithm, as it supports user-defined distance measures and has few hyperparameters for adjustment. This last aspect is especially relevant in online ML due to the dynamic nature of the learning tasks and the

* Corresponding author.
E-mail addresses: saulomastelini@gmail.com (S.M. Mastelini), bveloso@fep.up.pt (B. Veloso), maxhalford25@gmail.com (M. Halford), andre@icmc.usp.br (A.C.P.d.L.F. de Carvalho), jgama@fep.up.pt (J. Gama).

https://doi.org/10.1016/j.inffus.2023.101979
Received 25 November 2022; Received in revised form 31 July 2023; Accepted 18 August 2023
Available online 24 August 2023

inherent difficulty of adjusting hyperparameters online. Some online k-NN variants implement long-term memory schemes and sophisticated mechanisms for dealing with concept drift [10,11]. Still, these solutions do not employ efficient strategies for speeding up NNS.

Some effort has been made to create incremental search indices and, thus, speed up NNS time. Examples include kd-Trees [2,12], Product Quantization and related techniques [13,14]. Nonetheless, such solutions are limited to specific distance measures and might offer limited support for removing data from the search index.

Graph-based NNS has risen as the state-of-the-art in recent years [1,3]. Besides working with generic distance or similarity metrics, graphs can be naturally expanded with new nodes, an appealing feature to online ML applications. Nonetheless, to the best of our knowledge, frequent element removal and its impact on the search graph structure are still to be explored. This paper proposes Sliding Window-based Incremental Nearest Neighbors (SWINN) to handle approximate NNS in sliding windows. Our proposal is inspired by NN-Descent [15], one of the most popular graph-based approximate NNS strategies for batch data. SWINN is significantly faster than an LS of the sliding window when L is sufficiently large while keeping competitive search recall.

Our main contributions can be summarized as follows:

- We propose Sliding Window-based Incremental Nearest Neighbors (SWINN) for handling NNS in sliding windows;
- We compare SWINN against the current state-of-the-art online k-NN models using a comprehensive and extensive synthetic setup. We show that our proposal can deliver comparative search recall results while being significantly faster than an LS of the data buffer;
- We study how each hyperparameter of SWINN impacts its running time, memory footprint, and search recall. We also suggest a set of default hyperparameter values to use in general tasks and provide guidelines to select good hyperparameter value combinations if tuning is required;
- We study in which situations SWINN might be preferable to a LS and in which situations the opposite holds;
- We perform case studies comparing SWINN against popular online ML models in classification and regression tasks, showing more evidence to support our proposal's effectiveness.

The remaining of this manuscript is organized as follows. Section 2 presents previous works related to this research and the theoretical foundations needed to support our proposal and facilitate its understanding. We introduce SWINN and its main aspects in Section 3. We present our synthetic setup to compare SWINN against the current prevalent solution, i.e., an LS of the sliding window, in Section 4. Next, we discuss the obtained results using the controlled data in Section 5 and provide guidelines for selecting hyperparameter values. Complementary results and examples are available in Appendix. We perform case studies comparing SWINN and LS in classification and regression tasks against popular online ML algorithms in Section 6. Finally, we present our final considerations and future work directions in Section 7.

2. Background

This section presents the related work closest to our proposal and the theoretical foundations needed to understand better how SWINN works.

2.1. Related work

Multiple search index algorithms were proposed in the search for more efficient NNS strategies, primarily for batch applications. Some of these search indices were also adapted to allow incremental point addition and, in rare cases, element removal. The usage of partition trees is a frequent trend. In particular, the kd-Tree [16] and Ball-tree [17] models are widely employed to create query structures for NNS. Ball-trees can deal with high-dimensional data, whereas kd-trees are better suited for datasets with a small or moderate number of dimensions. A few attempts to adapt kd-Tree to incremental environments were proposed in the last years [2,12]. These adaptations can deal with data insertion and element removal. Nonetheless, frequent insertion/removal of elements (as in a sliding window regimen) makes these tree-based solutions impractical to our application setup due to the frequent need to re-balance the tree structures. Besides, kd-Trees are not well-suited to high-dimensional scenarios. Ball-trees could be used instead. However, their construction can become costly, even in batch-based applications. Another limitation of existing partition tree solutions is that they cannot work with arbitrary distance metrics.

A popular alternative trend in batch-based NNS is Locality Sensitive Hashing (LSH) [5,7]. Unlike traditional hashing techniques, LSH aims to map similar inputs to the same hash table position. Multiple families of mapping functions suited to different distance metrics were proposed throughout the years. Effectively, LSH acts as a discretization technique: query points will be mapped to the same positions as their nearest neighbors with high probability. LSH has been effectively used in multiple real-world applications [7]. Although LSH's projection scheme is naturally incremental, LSH techniques cannot be easily set up online due to the high dependence on the correct choice of hyperparameter values. Non-stationary scenarios might make this problem even more evident. Besides, data projection might be too costly to perform constantly. Lastly, LSH solutions also cannot work with arbitrary distance metrics.

Random Partition Trees (RPT) were effectively applied to real-world NNS problems by combining aspects from the two previous NNS solutions. One of the most popular versions of RPTs is implemented in the Spotify-backed library, Annoy [1]. Annoy creates ensembles of oblique trees whose partitions are hyperplanes equidistant to two randomly sampled points. Therefore, the RPTs use a partition scheme similar to many LSH solutions rather than applying axis-aligned splits like kd-Trees. RPTs can be independently created because the partitions do not depend on data-driven statistics. Moreover, tree construction is comparatively cheaper than in the vanilla kd-Tree algorithm. On the other hand, RPTs are also limited to a restricted group of distance metrics and are static by design. Once the RPT forest is created, it should be used to perform all search queries. Element addition and removal are not trivial, as we deal effectively with multiple search indices, i.e., the individual trees, rather than a single search index. Recently, a partition scheme based on random forests was proposed to perform incremental NNS [18]. In this case, however, the focus on a regression task rather than NNS. The partitions are determined by split merits used to build incremental trees, instead of a completely random mechanism as in Annoy's trees. In the mentioned work, the splits are axis-aligned, as in kd-Trees.

Vector Quantization (VQ) is another strategy used for NNS [6,8]. This family of solutions can efficiently deal with high-dimensional data but can be comparatively less accurate than the other presented strategies. The efficiency and performance trade-off is dependent on the correct hyperparameter choice. VQ-based solutions are also limited to specific distance metrics. The most popular VQ strategy is Product Quantization (PQ), which relies on batch clustering to create data partitions and encode the original data using the created partitions' information. Variations of VQ, including Product Quantization, have been proposed for dealing with online learning scenarios [13,14]. The online PQ strategy can be applied to a sliding window learning regimen but might be too restrictive regarding distance measures.

Graph-based methods have become a synonym for NNS effectiveness and efficiency in the last few years [3,19]. Among the existing solutions, small-world graph-based methods [3,20] have received special attention from the research community. Nonetheless, the cost of graph construction usually renders this kind of solution impractical

in online learning scenarios. Recently, a graph-based solution was proposed to handle incremental insertion and delete options [21] when using the Euclidean distance. The solution has different strategies to cope with edge deletion and can accurately do so. Nonetheless, this proposal is designed to work in a batch-incremental fashion, i.e., vertex addition and removal are performed after predefined intervals. Moreover, the graph-based NNS caters to processing high amounts of entries and handling a high number of vertices. On the other hand, our application setting is a pure online setting with constant element addition and deletion, i.e., a sliding window.

Early baselines for graph-based NNS have interesting properties that enable their adaptation to online NNS. In particular, the pivotal NN-Descent algorithm [15] is especially appealing because its construction is inherently incremental. NN-Descent also works with arbitrary distance metrics and could be adapted to handle frequent element addition and removal. Our proposal, SWINN, is inspired by the NN-Descent search index-building algorithm. Nonetheless, we extend the original algorithm to enable performing local changes in the search index. Such changes are necessary to add and remove new elements and to keep the graph reliable for search queries.

2.2. Preliminaries

Next, we present definitions used throughout the paper that help to describe the functioning of SWINN. We start by formalizing the concept of a graph.

Definition 1. A weighted graph, G, is defined by the tuple $G = (V, E)$, where V represents the set of vertices and $E = \{(o, d, w) \mid o \in V, d \in V, w \in \mathbb{R}, \text{and } o \neq d\}$ is the edge set. In the cases where the weight information is irrelevant, we will use (o, d) as a shorthand for (o, d, w).

If the edges in E are directed, then $(o, d) \neq (d, o)$. If the ordering of the edges is irrelevant, the graph is undirected. SWINN uses directed graphs weighted by the distance between the instances in the sliding window. For such, we need to define the concept of the direct neighborhood.

Definition 2. The direct neighborhood, $N_v \subset V$, of a vertex $v \in V$ is defined as $N_v = \{n \in V \setminus \{v\} \mid (v, n) \in E\}$, i.e., the nodes for which v have direct edges.

In our setting, we consider a first-in, first-out (FIFO) data buffer with limited size, i.e., a sliding window. We denote by L the sliding window length, which is a user-defined parameter.

3. Sliding window-based incremental nearest neighbors

We propose **S**liding **W**indow-based **I**ncremental **N**earest **N**eighbor (SWINN), which is an adapted version of the NN-Descent [15] algorithm suited for incremental learning scenarios. SWINN is based upon the assumption that "the neighbor of my neighbors might as well be my neighbor". Each instance in the sliding window data buffer is mapped to a vertex in the neighborhood graph. New vertices are added as new instances arrive, and the oldest instances (and their corresponding vertices) are dropped from the graph using the FIFO strategy. SWINN uses an iterative process to create and keep a NN graph. The high-level basic steps for graph construction are given as follows:

1. Start with a random neighborhood graph;
2. For each node in the search graph:
    (a) Refine the current neighborhood by checking if there are better neighborhood options among the neighbors of the current neighbors;
3. If the total number of neighborhood changes is smaller than a given stopping criterion, then stop.

Although simple, the above-presented general lines for NNS graph construction are powerful in multiple aspects. First, there are no limitations in the type of distance measure used to build the search index, which is not the case when considering LSH, trees, or quantization-based search indices [6,8]. Second, the graph-building process is incremental by design. Moreover, unlike tree-based indices, e.g., kd-trees [2, 12], we can perform searches from any starting vertex, making index update procedures easier.

We can improve the above graph construction strategy by directly joining the neighbors of a given node. Instead of performing two graph hops, i.e., checking whether the neighbors' neighbors are better candidates than the current ones, we make all the neighbors of a given node consider each other as possible new neighbors. Hence, we perform a single traversal hop.

We work with directed graphs; each vertex must have at most K direct neighbors. We use a capital letter to differentiate the number of neighbors used for graph building (K) and the number of neighbors used during search queries (k). The former value is fixed for SWINN's graphs, whereas the second may vary for each query. Although the number of direct neighbors, K, is fixed, we also need to consider the reverse neighborhood, as defined next.

Definition 3. The reverse neighborhood, $N'_v \subset V$, of a vertex $v \in V$ is defined as $N'_v = \{n_r \in V \setminus \{v\} \mid (n_r, v) \in E\}$, i.e., the vertices with direct edges to v.

Considering the reverse neighborhood, the number of edges might increase considerably to a value higher than K. As we work in a sliding window regimen, the maximum number of vertices the search graph will have is limited by the length of the sliding window (L), i.e., $L = |V|$. A vertex's maximum number of reverse neighbors is bounded by $L-1$. The input data define reverse neighborhood characteristics. Some works discussed different phenomena related to the placement of points in the search space and how these factors influence graph-based search indices [22,23]. For instance, vertices with a high number of direct and reverse neighbors, i.e., the so-called hubs, might increase the chance of a search reaching local minima. Given the typically reduced number of instances considered in the sliding window learning regimen, discussing the mentioned influencing factors is out of the scope of the current work and will be addressed in future research.

Next, we define the total neighborhood as follows:

Definition 4. The total neighborhood, T_v, of a vertex $v \in V$ is defined as $T_v = N_v \cup N'_v$, i.e., the union between the direct and reverse neighbors of v.

When presenting the high-level description of the graph-building mechanism, we assumed that a set of instances was previously available. In online ML scenarios, learning occurs as soon as new instances arrive. SWINN has a `warm-up` period when instances are buffered, and no graph is built. During this stage, searches are performed using the LS for all the data. After the warm-up period, SWINN builds an initial graph and keeps updating the search index by adding and removing instances. From this point onward, only the NN graph is used to perform search queries.

We will detail in the following subsections each aspect of SWINN, including graph refinement, how to perform search queries using the NN graph, how to remove vertices from the graph, and possible ways to reduce the memory footprint of the search index. In the end, we will combine the description of each component to describe how the whole SWINN algorithm works.

3.1. Refinement

The refinement procedure sequentially exchanges edges to make the search index converge to the NN graph. For such, SWINN performs neighborhood joins considering combinations of reverse neighbors and direct neighbors related to a reference vertex, i.e., select all

combinations among their reverse and direct neighbors for a given vertex.

Since the number of reverse neighbors a vertex can have is unbounded, the number of possible combinations may become too high. Limiting the number of neighbors joined at each refinement step can decrease costs. This strategy was proposed for NN-Descent, and we adopted it for SWINN. Specifically, SWINN takes a sample of the total neighborhood from a reference vertex when its total number surpasses a user-given threshold, $\texttt{max}_c$.

Even by limiting the number of vertices involved in neighborhood joins, NN-Descent can still perform redundant neighborhood change checks. This may happen when a previously attempted edge connection occurs during neighborhood joins. To reduce this problem and save processing time, NN-Descent keeps track of edges already tried by keeping binary flags for each vertex. Thus, each time a new direct neighbor is added to a vertex, a binary flag corresponding to the new neighbor is set to `true`. If this neighbor is used in a neighborhood join, its binary flag is set to `false`.

Neighborhood joins are performed by simultaneously exploring the following combinations: (1) both vertices involved in the join have their flags set to `true`; (2) the origin of the edge has a `true` flag, whereas the destination has a `false` flag.

These two constraints ensure that the left side of the join, i.e., the origin of the directed edge, was not previously involved in a neighborhood join. Note that reverse neighbors also have a binary flag corresponding to the reference vertex. Although the binary flags add a slight memory overhead, they avoid trying to link previously evaluated vertices. Such as NN-Descent, SWINN applies an incremental version of this optimization. We present in Appendix A.1.1 an example of the graph refinement procedure. SWINN applies two tests to stop performing the refinement process:

1. The maximum number of iterations is reached ($\texttt{max}_{\text{iter}}$);
2. The total number of edge changes during an iteration of refinement is smaller than δKL, where δ is a convergence tolerance parameter.

The computational complexity for refining the graph is dominated by matching vertex pairs. A naive strategy would combine, on the limit, every possible vertex pair. This action would result in cubic costs, as all vertices need to be visited as the focus of the refinement process. Instead, we limit the number of expanded neighbors for every explored vertex. Hence, the total cost of the refinement procedure is $O(\texttt{max}_c^2 L)$. We present the neighborhood refinement procedure in Algorithm 1. Next, we explain how the search is performed in SWINN.

Algorithm 1 Graph refinement procedure.

Require:
$V_r \subseteq V$: a list of vertices to perform the graph refinement procedure;
$\texttt{max}_c$: the maximum number of candidates to consider in local neighborhood joins;
δ: convergence criterion;
$\texttt{max}_{\text{iter}}$: the maximum number of iterations allowed for graph refinement.

procedure SWINN-REFINEMENT(V_r, $\texttt{max}_c$, δ, $\texttt{max}_{\text{iter}}$)
  for $i \in \{1, ..., \texttt{max}_{\text{iter}}\}$ **do**
    for $v \in V_r$ **do**
      Retrieve T_v and its corresponding binary flags B_v ▷ total neighborhood
      Take a random sample, $S(T_v)$, of size $\texttt{max}_c$ out of T_v
      for every $v' \in S(T_v) | B_v(v') = \texttt{true}$ **do**
        Attempt to create edges between v' and vertices whose binary flags are `true`
        Attempt to create edges between v' and vertices whose binary flags are `false`
      Finish the procedure if the total number of edge changes is smaller than $\delta K |V_r|$

3.2. Search

SWINN relies on a simple greedy search strategy to look for the NNs of a given query point. We start by explaining how search works in the 1-NN case. Afterward, we expand our discussion to include the cases where $k > 1$. Although we select a specific K value to build the index, the number of returned NNs (k) is only bound by L during the search.

The search starts from a randomly chosen vertex, v. The distance from this vertex to the query, dubbed d_b, is saved as the current best solution. SWINN expands the total neighborhood of the starting vertex, T_v. The search traverses to the first element $T_{v,1}$, in the unordered list of total neighbors. If the distance from $T_{v,1}$ to the query is smaller than d_b, $T_{v,1}$ becomes the new best solution and its neighbors are marked for further exploration. SWINN also updates the value of d_b, accordingly. Otherwise, $T_{v,1}$ is discarded from the search procedure and a new neighbor of v, $T_{v,2}$ is selected to continue the exploration. In this setting, the search uses a standard queue structure to keep the vertices that are yet to be explored. The visited vertex with the smallest distance to the query is the final answer. The drawback of this strategy is that no vertex whose distance to the query is smaller than the current value of d_b will be visited. Hence, d_b bounds the minimum distance to the query point a vertex must have in order to be visited during the search.

Alternatively, we can set a $\epsilon \geq 0$ parameter to extend the distance bound, i.e., during the search, the distance bound will be defined as $(1+\epsilon)d_b$. Hence, vertices can still be explored, even though their distance to the query is (slightly) higher than the current best solution. Setting $\epsilon > 0$ usually increases the search accuracy at the cost of increased running time. To find an adequate trade-off, we experiment with different values of ϵ in our experimental setup. We also illustrate the 1-NN search in SWINN with a toy example presented in Appendix A.1.2.

We can generalize the search for the k-NN case using two binary heap data structures. The first heap, H_{min}, is a min-heap and keeps a pool of vertices whose neighborhood is yet to be explored. This heap first explores the vertices with the smallest distances to the query element. The second heap, H_{max}, is a max-heap and stores the search results. The head of H_{max} carries the farthest among the k-NNs.

During the search, elements are removed from the head of H_{min} and have their total neighborhood explored, as in the 1-NN case. Each vertex in the total neighborhood of the head of H_{min} is a candidate to be added to both H_{min} and H_{max}. The distance bound, previously described, is calculated using the head element of H_{max}. A new vertex is added to H_{max} only if its distance to the query is smaller than the distance bound. In this case, the head element of H_{max} is removed, and the new candidate is added to both H_{min} and H_{max}. Note that H_{max} always carries at most k elements, whereas H_{min} is not bounded in length. The distance bound is also updated with the new head element in H_{max}. The search concludes when there are no remaining elements to explore in H_{min}. At this point, H_{max} has k elements partially sorted in the reverse order. These items can then be quickly sorted in increasing order according to their distance to the query and returned as the search result.

The cost of the greedy search performed by SWINN is mainly dominated by the branching factor and the number of hops from the origin until the query. The branching factor can be at most T_v, whereas the number of graph hops, h, is data and starting point dependent. Therefore, the total search in a naive greedy search is $O(T_v^h)$. The literature on graph-based NNS reports small practical h values, which turn the search cost sublinear in the number of instances in the data buffer. Notwithstanding, we also add mechanisms to avoid visiting vertices more than once and rely on the distance bound to limit vertex exploration. For that reason, the actual branching factor is much smaller than T_v. Lastly, we also propose a graph pruning strategy that

can possibly reduce the branching factor even further. Thus, effectively reducing the search cost to $O(\max_c^h)$ in the worst case. Note that we are not considering the cost of maintaining the heap structures used internally, which overall is $O(k \log k)$. As k is fixed during the search, we consider this extra cost constant for a given search query.

3.3. Vertex addition

We already have tools to refine the neighborhood of a random graph, and we can also perform search queries once the k-NN graph is created. Hence, we can add new elements to the search graph using a simple strategy. When a new data point arrives, SWINN uses the existing graph to find the instances' K neighbors. SWINN then creates a new vertex to accommodate the new instance and adds direct edges to the K found vertices. Our proposal also updates the reverse neighborhood of the selected neighbors to account for the newly added vertex. The cost to add a new element is asymptotically equivalent to the search cost of a single query with $k = K$.

3.4. Element removal

In SWINN, instances are supposed to be constantly removed from and added to the graph in a sliding window. Hence, we need to ensure that the search index remains reliable, even though some of its nodes and edges will constantly change. As previously discussed, new elements can be added by using the current search graph to find the direct neighborhood of the new vertex. Vertex removal is more challenging than element addition.

A possible option for removal is to apply the needed changes in the index, i.e., remove nodes/edges, and then run the refinement procedure (Section 3.1) using the whole graph. However, this action has a cost of $O(\max_c^2 L)$, as previously discussed. Following this strategy is more costly than performing a linear scan (LS) in the data buffer. Another option is to perform graph refinements at predefined intervals. Even so, there are no guarantees that the graph will be reliable for NNS during the intervals between refinements. We need to balance search index efficiency and reliability. For such, let us first define search reliability in our context.

Definition 5. A graph-based search index is reliable for NNS when there is only a single connected component, i.e., every node in the graph can be reached from any given starting point.

If this is not the case, the search is deemed to fail with a wrong choice of starting point. Some vertices might become inaccessible even if a seed vertex close to the query is selected. In SWINN, we perform local adjustments to the graph after each element removal. Therefore, only the local neighborhood of the removed vertex is updated.

We get the list of reverse and direct neighbors of the vertex to be removed, i.e., the oldest element in the sliding window. The oldest element is removed from the graph together with its edges. We then perform two filtering procedures in the previously retrieved neighborhood lists.

1. From the list of reverse vertices, only keep those with no direct neighbors;
2. From the list of direct neighbors, only keep those with no reverse neighbors.

On the one hand, we have a list of reverse neighbors without direct neighbors. On the other hand, we have vertices without reverse neighbors. Although none of the situations is inherently problematic, as long as there are alternative paths to reach one group starting from the other, we cannot directly guarantee that both groups are not separate, i.e., the two lists might contain border vertices in two sub-graphs. We could verify if that is the case using a Minimum Spanning Tree algorithm (MST), such as Kruskal's [24], and verify whether a single MST or a forest thereof is obtained. Notwithstanding, Kruskal's algorithm has a cost of $O(E \log V)$, which in our case is $O(KL \log L)$, due to the nature of the search index. Running an MST building algorithm after almost every vertex removal is too costly. We propose an alternative solution to keep the graph reliable and computationally efficient.

For such, we first check if there is any intersection between the two filtered lists. If so, these vertices are isolated in the search graph. We fix this situation by re-adding them to the index. Next, for each node in the list of filtered reverse neighbors, we add new K neighbors using the graph search procedure. However, instead of randomly selecting a random search seed vertex among all those available, we randomly select a vertex from the second filtered list, i.e., the list of direct neighbors of the removed vertex. As a result, we might create connections between two separate sub-graphs. In case the second list is empty, we still add new neighbors to the members of the first list using random search seed vertices. After creating the new connections, SWINN applies the graph refinement procedure to the vertices from the two filtered lists. The total cost for removing a vertex v from the search graph is $O(T_v + \max_c^2 T_v)$. The right-hand side of the complexity expression comes from running the refinement procedure only considering the neighborhood of the removed vertex.

3.5. Edge pruning

As vertices can have mutual edges and intersecting (undirected) neighborhoods, the resulting search index might contain multiple and, potentially, redundant paths between two given vertices. Even though some edges, at first glance, are worse than others, they still can be helpful during index searches, e.g., a given vertex's worst edge, w.r.t. distance, might provide the most direct path toward the search query. On the other hand, if paths are redundant, they will increase the search time and the memory footprint of the search index.

As discussed in Section 3.2, SWINN relies on greedy search, as many of the preceding batch-based graph solutions. For this reason, at each step of the search, the total neighborhood of the current node must be expanded to select the potential best path to follow. At this point, the impact of having multiple paths between two vertices is twofold: (1) they can boost search accuracy by enabling faster hops towards the target vertex; (2) redundant paths increase the neighborhood exploration time with no direct benefits to the search accuracy.

The search graph can be pruned to reduce the number of edges and speed up the search. However, care must be taken when removing edges to avoid creating multiple connected components, i.e., subgraphs. Due to the incremental nature of the underlying search process, we should avoid calculating distances between vertices to save computational resources. Hence, we propose a simple edge pruning procedure that accounts for calculated distances between vertices. We illustrate the graph pruning procedure in Appendix A.1.3.

The main idea is akin to the refinement procedure employed to create the search graph. This time though, we want to "remove connections to neighbors of my neighbors if they are closer to my neighbor than they are from myself". SWINN creates a min-heap during the pruning procedure to track the direct and reverse neighbors of a focus vertex v, ordered by their distance. The best neighbor is retrieved from the min-heap and added to a set of selected neighbors S. For each remaining candidate c in the neighborhood heap, SWINN first checks if an edge exists between c and one of the selected neighbors $s \in S$. When it is not the case, c is added to S. If an edge exists between c and an element $s \in S$, the farthest among the undirected edges (v, s) and (s, c) is discarded.

Therefore, if we imagine a triangle formed by vertices v, s, and c, SWINN effectively removes the polygon's longest side. Following this strategy, v, s, and c are still mutually reachable even after removing the longest edge among them. Pruning can be applied to all vertices in the search index, but in SWINN, we only apply it after performing (local) graph refinements. Even in these cases, a vertex v

is only subjected to edge pruning if $T_v > \texttt{max}_c$, i.e., the size of its total neighborhood is higher than the maximum number of allowed candidates in neighborhood joins.

Even though the presented procedure guarantees that vertices are accessible and the index is not split into multiple connected components, care must be taken to remove edges. Due to the usage of a greedy search algorithm, long edges might help to avoid local minima. Hence, neither keeping all the edges nor removing all possible redundant paths might be optimal depending on the considered performance measure. For this reason, we add the parameter $\texttt{prune}_{\text{prob}}$ to control the probability of removing a redundant edge. Therefore, the pruning procedure works as previously described, with the difference that each time a redundant edge is found, there is a probability, $1 - \texttt{prune}_{\text{prob}}$, of not removing it from the graph. The value of $\texttt{prune}_{\text{prob}}$ is a user-given parameter. When $\texttt{prune}_{\text{prob}} = 0$, no edges are removed, whereas when $\texttt{prune}_{\text{prob}} = 1$, all possibly redundant edges are going to be removed. The cost of the pruning operation for a single vertex is $O(T_v)$.

3.6. The complete SWINN functioning and its complexity analysis

In this section, we show how each previously presented element of SWINN works in combination with the others. The complete SWINN updating procedure is presented in Algorithm 2.

Algorithm 2 The complete SWINN update function.

Require:
D: the data stream;
K: the number of direct neighbors each vertex must have;
$\texttt{max}_c$: the maximum number of candidates to consider in local neighborhood joins;
$\texttt{prune}_{\text{prob}}$: the probability of removing a long edge;

function SWINN-UPDATE(D, K, $\texttt{max}_c$, $\texttt{prune}_{\text{prob}}$)
  Let Q be a queue with at most L elements and G be an empty graph
  while D has instances **do**
    $\texttt{item} \leftarrow \texttt{next}(D)$
    Let v_{new} be a new vertex containing item and no edges
    if G is empty and $|Q| < L$ **then**
      Add v_{new} to Q
      if $|Q|$ is equal to the warm-up period **then**
        Create a random graph $G = (V, E)$ using all elements in Q
        Refine G using all the vertices (Section 3.1)
        Skip to the next instance
    if $|Q| = L$ **then**
      Remove the oldest vertex, v_{old}, from the graph
      Fix the previous neighborhood, $T_{v_{\text{old}}}$, of v_{old} (Section 3.4)
      Apply the graph refinement procedure to the vertices in $T_{v_{\text{old}}}$
      for $v \in T_{v_{\text{old}}}$ **do**
        if $|T_v| > \texttt{max}_c$ **then**
          Prune redundant edges of v with probability $\texttt{prune}_{\text{prob}}$
    Find the K nearest neighbors of v_{new} using G (Section 3.2)
    Add v_{new} along with edges to its K nearest neighbors to G
    Add v_{new} to Q
  return G

We already discussed the computational costs to perform search queries given a refined search graph. The total cost to update the search graph with a new instance and discard the oldest one comes directly from the SWINN's components costs previously presented. Removing a single node has a cost of $O(T_v + \texttt{max}_c^2 T_v)$. The removal cost encompasses the local graph refinement performed to keep the search index reliable for further search queries. The missing part is the cost of finding the K closest neighbors of the new vertex before its addition, which is $O(\texttt{max}_c^h)$. Hence, the total for updating SWINN's graph with a new instance is $O(T_v(\texttt{max}_c^2 + 1) + \texttt{max}_c^h)$. It is clear that the $\texttt{max}_c$ parameter plays an important role in SWINN's cost. The values of T_v and h are also highly impacting and come directly from the choice of K and from the characteristics of the problem. The former aspect is controlled by the user, whereas the latter is inherent to each task. In practice, if the main goal is speed, K must be as small as possible while ensuring a single connected graph component. Note, however, that fewer connections, i.e., smaller K values, might imply a higher number of graph hops needed during search queries. This observation not only impacts the total running time negatively but also can increase the chance of the search becoming stuck in local minima.

4. Experimental setup on simulated data

In this section, we describe the experimental setup used in our experiments, including the data, the evaluation measures, the baseline, and the settings used in SWINN. SWINN was implemented in Python and will be incorporated into River [25] for ease of public use and access. All the reported experiments were performed sequentially in a CentOS machine with 2 Intel Xeon E5-2667v4 processors with eight cores running at 3.2 GHz and 512 GB of DDR3 RAM.

4.1. Baseline, data and evaluation metrics

For the baseline, we compared the performance of SWINN to a linear scan (LS) of the data buffer. LS is also sometimes referred to as an exhaustive search. First, we compared our proposal against the baseline by using synthetic data. Our goal was to understand better how the hyperparameters involved in the design of SWINN affect its predictive performance.

In our experiments, we generated 10 uniformly sampled features and 50 000 instances for each case study. This experimental setup did not consider concept drift, as sliding window-based learning models are naturally adaptive by keeping only the most recent data. The evaluation strategy followed a test-than-train approach [26], i.e., the popular prequential evaluation approach used in data stream learning [27]. In this strategy, for every incoming instance, we first retrieve the position of its k-NN in the sliding window and then add the new example to the buffer.

The query results of the LS baseline are used as the ground-truth values to calculate the Search Recall (SR) values, which vary between $[0, 1]$, where 1 is the best possible value. Thus, SR represents the mean percent of the true NNs that SWINN can retrieve. As LS checks every possible example in the data buffer, its SR is always 1. Ideally, we want to obtain values as close to 1 as possible with SWINN while being faster than LS performing search queries. Hence, we measure the running time of both LS and SWINN in seconds, considering element insertion, search query, and the total run time. We also measure the memory footprint of the compared algorithms for NNS.

4.2. Graph configurations

Table 1 presents the list of hyperparameters assessed for SWINN. We separate the hyperparameters into two categories: index building and search. We left the hyperparameters values of the two convergence criteria fixed during our experiments, following guidelines defined in the literature [15,23]. Moreover, we set SWINN to use the first 100 instances in the stream to warm-up. We discuss the impact of changing the values of the remaining hyperparameters in the experiments' discussion section. The values of L (window length) were applied to both SWINN and the LS baseline.

Table 1
List of SWINN hyperparameters considered in our experimental setup. The choice of L and k also applies to the Linear Scan baseline.

Stage	Name	Description	Value(s)
Build	δ	Convergence criterion	0.001
	$\texttt{max}_{\text{iter}}$	Maximum number of iterations for graph refinement	10
	L	Sliding window length	{100, 250, 500, 1000, 5000}
	K	Number of neighbors during graph construction	{5, 10, 20, 30}
	$\texttt{max}_{\text{c}}$	Maximum number of candidates to consider during local neighborhood joins	{20, 30, 50, 100}
	$\texttt{prune}_{\text{prob}}$	Probability of pruning a long edge	{0.0, 0.3, 0.5, 0.7, 1.0}
Search	ϵ	Tolerance hyperparameter for the distance bound	{0.0, 0.1, 0.5, 1.0}
	k	Number of neighbors to retrieve during search queries	{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

5. Results and discussion

In this section, we evaluate the performance of SWINN in the synthetic data described in Section 4.1. We start by analyzing how each hyperparameter affects SWINN's performance and later compare SWINN against the LS of the data buffer. In Section 6, we apply our findings in classification and regression case studies using synthetic and real-world data.

Multiple aspects have an impact on the performance of the search index, e.g., it is widely known that distance-based methods suffer the so-called "curse of dimensionality". The NN-Descent authors mention in the original paper that their method works better for data with less than 20 features [15]. Besides, the efficacy of a search index is closely related to the characteristics of the data. In this section, we limit our analysis to a controlled experimental setup to better understand each aspect of SWINN and its impact on predictive performance.

External factors, such as data dimension, outliers and hubness [15,22,23,28], remain open questions and should be addressed in the future. However, as we operate in a sliding window environment, possible sources of search instability are transient. For example, it is known that nodes with an increased number of total neighbors affect NN-Descent in a negative way [22,23]. In SWINN, as more data are observed, these nodes are eventually removed from the graph. Therefore, some challenges faced by batch applications might not affect SWINN to the same extent.

We organize our discussion by formulating the following research questions, which will be addressed in the next subsections:

- **Q1:** Is SWINN effective regardless of the sliding window size?
- **Q2:** Does the number of requested nearest neighbors impact the search recall during graph search?
- **Q3:** How ϵ value impacts search recall and running time during search queries?
- **Q4:** What is the impact of the chosen number of neighbors used to build SWINN's graph on search recall and running time?
- **Q5:** What is the impact of neighborhood sampling when performing local joins during graph refinement?
- **Q6:** What is the impact of edge pruning on the overall search recall and memory and time costs?

5.1. Sliding window size

We ran every hyperparameter combination described in Section 4.2 and checked the effectiveness of SWINN accordingly to the running time and SR. We report our findings in Table 2 accounting for hyperparameter value combinations that were faster than a LS while also delivering SRs larger than 0.9. When considering only time, no gains were observed for $L = 100$. In fact, in our experiments, we set the warm-start period to 100 instances, as reported in Section 4.2. As shown in the table, gains in processing time are only noticeable starting at $L = 250$. SWINN time efficiency becomes more apparent as the window length increases, and our proposal was always faster than an LS when $L = 5000$.

Still, considering $L = 250$ and SR, no hyperparameter value combination delivers SR values higher than 0.9 while also being faster than a LS. We only start to observe hyperparameter value combinations able to be faster than LS and with SR > 0.9 when using $L = 500$. Nonetheless, the amount of hyperparameter value combinations able to deliver the combined gains are minimal (13.72%). Therefore, answering the research question **Q1**, we conclude that SWINN is better suited for windows with more than 500 instances. Hence, by setting the warm-up period to 500, one can achieve a good compromise in performance. In the following sections, we investigate the effect of the other hyperparameters of our technique and check how they impact SWINN's performance. We will, from here onward, limit our analysis to $L = 1000$ and $L = 5000$.

Table 2
Overall performance of SWINN compared to a linear scan of the data window, considering time and search recall.

Window length	100	250	500	1000	5000
$\text{Time}_{\text{SWINN}} < \text{Time}_{\text{LS}}$	0.00%	21.31%	48.72%	77.56%	100.00%
$\text{SR}_{\text{SWINN}} > 0.9$	100.00%	65.62%	61.87%	55.37%	46.78%
Both	0.00%	0.00%	13.72%	32.97%	46.78%

5.2. The impact of ϵ on graph search

The greedy search used in SWINN might become stuck in local minima. To deal with this limitation, we include the ϵ hyperparameter, which allows SWINN to explore vertices slightly worse than the available nearest vertex at each search step. Nonetheless, overextending the number of explored vertices during the search has a toll on the running time. Hence, a balance must be achieved by setting a proper ϵ value.

We analyze how the ϵ hyperparameter influences the running time and SR of SWINN in general. For such, we select the most unconstrained version of SWINN among our experimental setup, employing $K = 30$ neighbors to build the search index, using $\texttt{max}_{\text{c}} = 100$, and disabling edge-pruning. We select this combination to minimize other parameters' effects on the SR. In particular, the choice of the highest evaluated K value guarantees a single connected component in the search index and multiple redundant paths between vertices.

In Fig. 1, we present our analysis considering queries for the 1-st until the 10-th nearest neighbor. When analyzing the left side of the figure (first column), the choice of $\epsilon = 0$ has a negative impact on SR, especially when $k = 1$. SWINN produces SR values close to 0 when searching the 1-NN, regardless of the window length. There is a significant jump in SR for every selection of $\epsilon > 0$. Still, we observe that the higher the value of k, the higher the SR, a finding that answers the research question **Q2**. This behavior is related to how the search is performed in the graph. The distance bound, described in Section 3.2, is defined by the worst neighbor among the current best candidates. Thus, when $k = 10$, the 10-th nearest neighbor at any given point of the search will define the threshold to either explore or ignore the neighborhood of a candidate node. When $k = 1$, only a candidate will be kept as a solution, and the inclusion criterion will become more restrictive. In these cases, increasing the ϵ value is mandatory.

Overall, $\epsilon = 0.1$ provides the best balance between SR and running time, especially as the number of queried neighbors increases. Nonetheless, other values of ϵ might be selected depending on the primary goal.

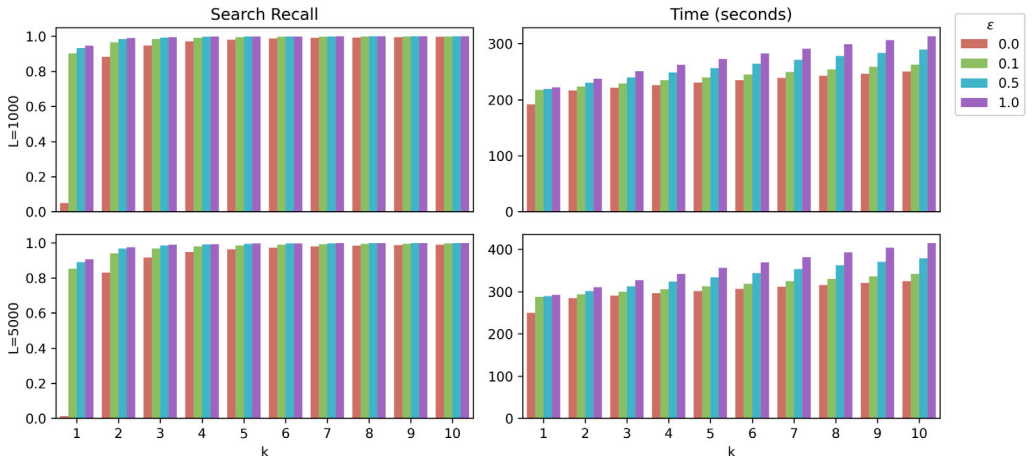

Fig. 1. The impact of ϵ on the total search recall (left) and search time (right). From top to bottom: window lengths of 500, 1000, and 5000 instances. Each color represents a ϵ value, and each group of bars represents the number of neighbors searched in each query.

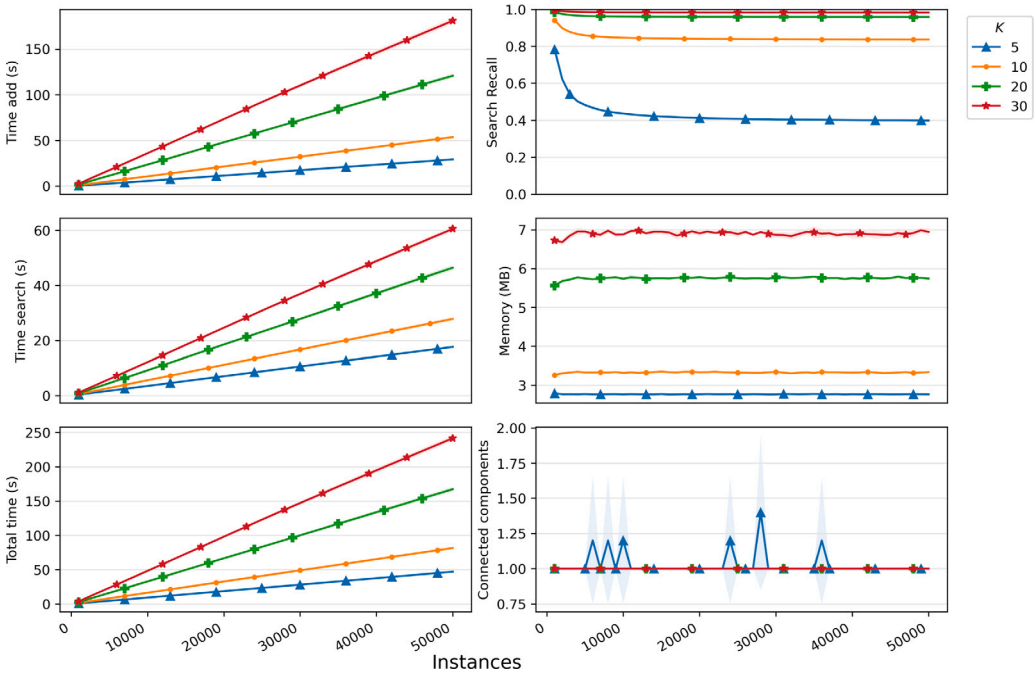

Fig. 2. The effect of increasing the value of K in SWINN, when $L = 1000$.

When searching for an increased number of neighbors, the choice of ϵ has little impact on SR, although it saves computation time. These observations answer the research question **Q3**. From this point onward, we select $\epsilon = 0.1$ to analyze the impact of K, $\texttt{max}_c$, and $\texttt{prune}_{\text{prob}}$ in SWINN.

5.3. The impact of the value of K on the search index

So far, we have only analyzed factors external to the search index building. These factors directly impact the search graph's performance but do not dictate how it is built. We now analyze how the choice of the value of K impacts the SR, the running time, the memory footprint, and the number of connected components in the search graph. We present our results in Fig. 2 considering $L = 1000$. Similar results were observed with $L = 5000$ and can be checked in Appendix.

To answer the research question **Q4**, we start by considering the running time measurements, depicted in the first column of Fig. 2. As expected, the running time to add elements to the graph and perform search queries increases as the value of K increases. Nonetheless, we also must consider how the choice of the value of K impacts the other performance measurements. Looking at the top-right chart of Fig. 2, we perceive that $K = 5$ yields significantly smaller values of SR in comparison with the other values of K. The bottom-right chart gives us the possible reason. Every value of K, except for $K = 5$, results in a graph with a single connected component. On the other hand, $K = 5$ generates more than one sub-graph on multiple occasions. Hence, SR results are expected to degrade as not every vertex will be reachable. For $K > 5$, a single connected component is always obtained. To answer the research question **Q4**, in general, the higher the K, the more accurate SWINN becomes, at the cost of increased computational resource usage.

Our goal is to find the best balance between time, SR, and search reliability. We believe that $K = 20$ represents a good compromise between all the metrics and will be used from this point onward. The downside of this choice, depicted in the middle chart on the right side of Fig. 2, is the increased use of memory to store the graph. As expected, the memory footprint of SWINN using $K = 20$ is smaller than the version with $K = 30$. Still, it is more costly than performing

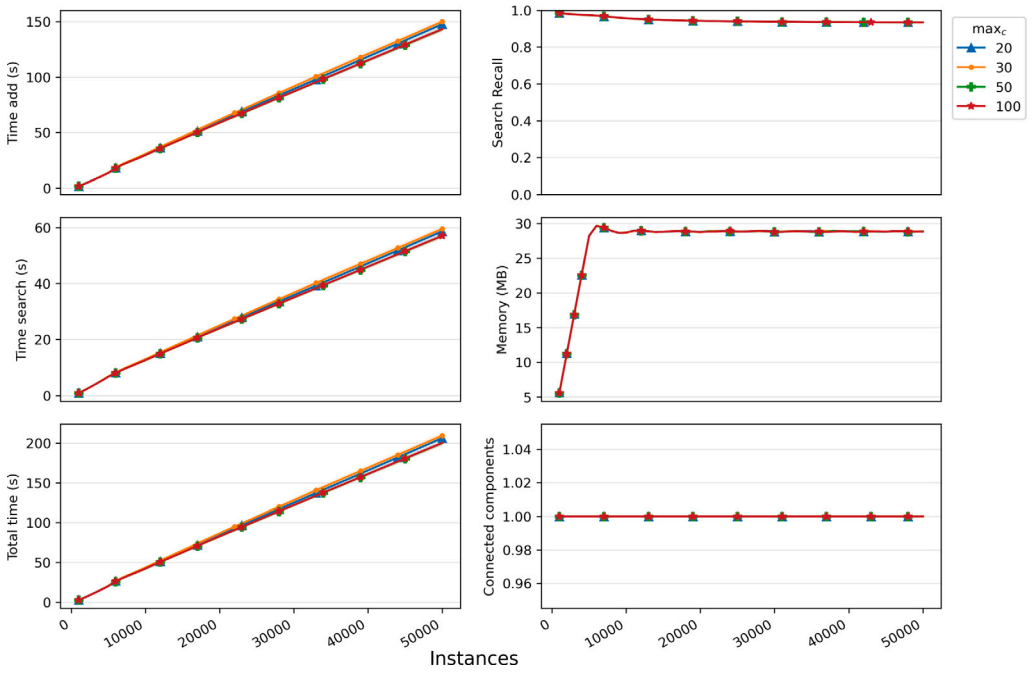

Fig. 3. The effect of increasing the value of $\texttt{max}_c$ in SWINN, when $L = 5000$.

an LS in the data buffer. SWINN spends additional memory resources to keep all the vertices and edges in addition to all the instances in the buffer. However, our primary focus is to improve the running time performance and keep competitive SR values.

5.4. The impact of neighborhood sampling during local joins

We also evaluate the impact of $\texttt{max}_c$ in the overall performance of SWINN. We used $K = 20$, as previously mentioned, and disabled edge-pruning. We present the results for $L = 5000$ in Fig. 3, while the results for $L = 1000$ are included in Appendix.

As it can be observed, the running time differences between different values of $\texttt{max}_c$ are negligible. In fact, versions that bound the number of neighbors in local joins to 20 and 30 are slightly slower than the less restrictive versions of SWINN. This is due to the cost of performing vertex sampling during the local joins. These differences are also hard to notice when using $L = 1000$.

To answer the research question **Q5**, we believe that limiting the number of vertices participating in local joins is useful when the total neighborhood is high, i.e., when the value of K is high, and vertices have an increased number of reverse neighbors. The user controls the first factor, while the second is data-dependent. As a compromise, we select $\texttt{max}_c = 50$ to have a possible balanced solution, regardless of the window length.

5.5. The impact of edge pruning

Finally, we evaluate the impact of edge pruning in SWINN, illustrating the effect of increasing the chance of removing possibly redundant edges in Fig. 4. For such, we selected $L = 5000$, but similar observations can be made to $L = 1000$.

It is difficult to observe changes in the running time and SR in this figure. The number of connected components remains unchanged, as desired. We cannot perceive significant variations in the memory footprint of SWINN. Unlike the batch graph-based search index, the edges in SWINN have a transient characteristic. Edge pruning could be helpful in cases where the value of K is higher than the values we evaluated in our experimental setup. Even in these cases, every node is eventually removed, even the so-called hubs in related literature [22, 23]. To answer the research question **Q6**, the benefits and drawbacks of edge pruning are shadowed by the naturally evolving characteristics of sliding windows-based nearest neighbor search. Therefore, we adopt $\texttt{prune}_{\text{prob}} = 0$ in the follow-up experiments, i.e., no edge pruning.

5.6. Comparing SWINN against a linear scan of the data

We also compare the performance of SWINN against a complete LS of the data buffer using the hyperparameter values combination obtained in the last section. In other words, we use SWINN with $K = 20$, $\texttt{max}_c = 50$, and $\texttt{prune}_{\text{prob}} = 0$. We present the obtained results in Table 3, considering windows of 1000 and 5000 instances. It is no surprise that the memory usage of SWINN is higher than LS'. This occurs because SWINN uses additional memory resources to store the search index, whereas LS only stores the instances per se. Notwithstanding, our primary focus is the running time, and even for large data windows, the amount of memory used by SWINN is easily manageable in traditional data stream setups.

Regarding SR, SWINN cannot match LS, regardless of the L value. This comes from relying on a greedy search strategy to perform queries in the search graph. Hence, the search is prone to be stuck in local minima, as it is primarily noticed when searching and using $k = 1$. As discussed in Section 5.2, ϵ can be used to enhance the greedy search and significantly improve the SR when the value of k is small. Still, when $k \leq 2$, the choice of $\epsilon = 0.1$ is not enough to make the SR values approach those obtained when $k > 2$, usually over 0.97. As an alternative, the user could increase ϵ further when searching for a small number of nearest neighbors. Nonetheless, in the future, we intend to search for alternatives to the greedy search currently used in SWINN and its batch counterpart.

Running time was the characteristic where SWINN truly shined. In Table 3, we highlight the number of times SWINN was faster than LS inside parenthesis. When $L = 1000$, SWINN was always faster than LS, though sometimes the speedup of our proposal was at most 20%. However, when we move to windows of 5000 instances, SWINN is generally at least 6 times faster than the LS. We believe SWINN is a solid choice to perform k-NN for larger window sizes, which would be impractical using LS.

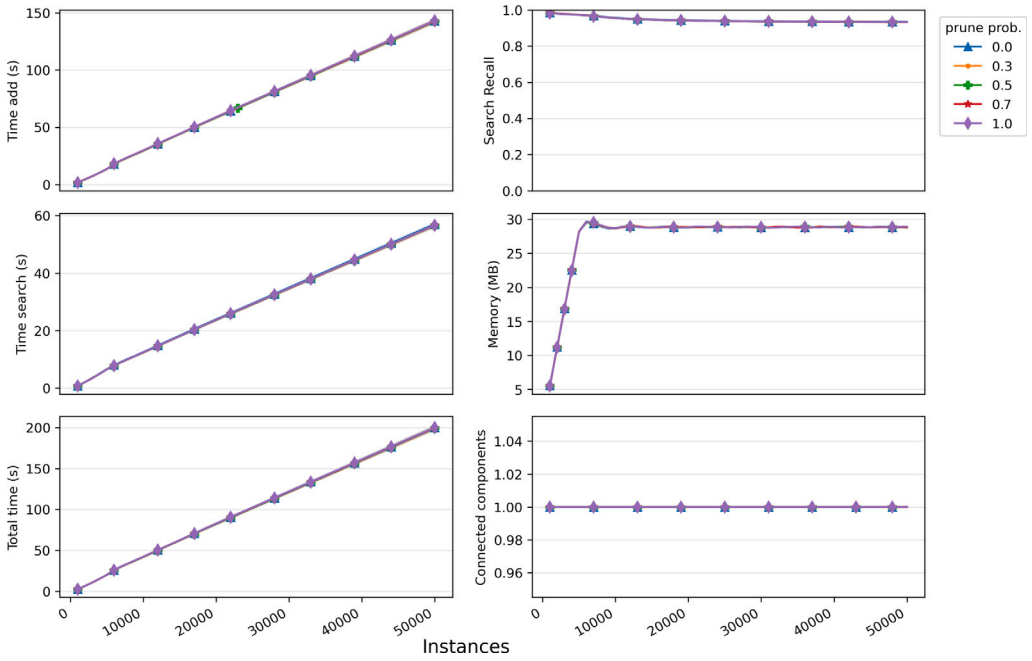

Fig. 4. The effect of increasing the value of $\texttt{prune}_{\text{prob}}$ in SWINN when $L = 5000$.

Table 3
Mean Search Recall, memory usage, and running time of LS and SWINN when considering $L = 1000$ and $L = 5000$. We indicate the mean memory usage of LS and SWINN alongside the window length. The number around parenthesis in the time measurements indicates how much faster SWINN performed in comparison with LS.

k	$L = 1000$ (LS: 1.22 MB, SWINN: 5.76 MB)				$L = 5000$ (LS: 6.11 MB, SWINN: 28.86 MB)			
	SR		Time		SR		Time	
	LS	SWINN	LS	SWINN	LS	SWINN	LS	SWINN
1	**1.00 ± 0.00**	0.81 ± 0.00	218.57 ± 13.87	**146.63 ± 3.45** (×1.49)	**1.00 ± 0.00**	0.75 ± 0.00	1370.13 ± 18.67	**179.00 ± 3.72** (×7.65)
2	**1.00 ± 0.00**	0.92 ± 0.00	218.77 ± 13.98	**150.86 ± 3.48** (×1.45)	**1.00 ± 0.00**	0.87 ± 0.00	1369.02 ± 20.62	**184.05 ± 3.31** (×7.44)
3	**1.00 ± 0.00**	0.95 ± 0.00	218.17 ± 13.85	**155.14 ± 3.59** (×1.41)	**1.00 ± 0.00**	0.92 ± 0.00	1367.35 ± 19.23	**188.87 ± 3.19** (×7.24)
4	**1.00 ± 0.00**	0.97 ± 0.00	218.52 ± 13.78	**159.22 ± 3.69** (×1.37)	**1.00 ± 0.00**	0.95 ± 0.00	1348.04 ± 19.71	**193.50 ± 3.19** (×6.97)
5	**1.00 ± 0.00**	0.98 ± 0.00	218.99 ± 13.30	**163.22 ± 3.82** (×1.34)	**1.00 ± 0.00**	0.96 ± 0.00	1359.93 ± 19.66	**198.09 ± 3.22** (×6.87)
6	**1.00 ± 0.00**	0.98 ± 0.00	218.55 ± 14.49	**167.07 ± 3.88** (×1.31)	**1.00 ± 0.00**	0.97 ± 0.00	1350.87 ± 18.44	**202.61 ± 3.24** (×6.67)
7	**1.00 ± 0.00**	0.99 ± 0.00	220.07 ± 13.89	**170.83 ± 3.97** (×1.29)	**1.00 ± 0.00**	0.97 ± 0.00	1368.97 ± 21.69	**207.02 ± 3.29** (×6.61)
8	**1.00 ± 0.00**	0.99 ± 0.00	218.33 ± 13.90	**174.36 ± 4.05** (×1.25)	**1.00 ± 0.00**	0.98 ± 0.00	1352.17 ± 19.21	**211.29 ± 3.40** (×6.40)
9	**1.00 ± 0.00**	0.99 ± 0.00	219.44 ± 14.60	**177.91 ± 4.08** (×1.23)	**1.00 ± 0.00**	0.98 ± 0.00	1366.96 ± 20.72	**215.59 ± 3.37** (×6.34)
10	**1.00 ± 0.00**	0.99 ± 0.00	218.58 ± 13.58	**181.19 ± 4.24** (×1.21)	**1.00 ± 0.00**	0.98 ± 0.00	1355.69 ± 20.67	**219.78 ± 3.47** (×6.17)

6. Case studies

We investigated three case studies to assess how SWINN performs when applied to supervised ML tasks. Unsupervised learning is also viable, e.g., anomaly detection, but we wanted straightforward ways of comparing predictive performance.

The first case study concerns a real-world problem, i.e., the hourly prediction of bike availability in a bike station in the city of Toulouse — France. We use 3801 hourly observations taken from pl. Etienne Esquirol's bike station. This specific task concerns a nowcasting problem. In other words, we are interested in predicting the observation that will come immediately next to the last available one. Differently from the other case studies that we will discuss later, there are explicit and strong temporal dependencies among consecutive observations. Such cases fit perfectly with the data ingestion policy used by SWINN when considering short-term memory.

In the last two studies, we used data generators for classification and regression tasks. Our idea is to define a limited time to allow different classification and regression algorithms to run and verify their effectiveness in predictive performance and data processing throughput. Data generators can produce synthetic instances indefinitely and allow the user to control the data characteristics, such as concept drift. These were desired characteristics to perform our evaluation case study. Thus, we relied on synthetic data, rather than real-world datasets, as means of better evaluating the behavior of our proposal and its contenders in the predictive tasks where they were applied. We allowed each compared algorithm to run for one hour in the same machine,[1] in which each datum from the data generators was processed sequentially, using the prequential strategy. We also used similar classification and regression algorithms in both cases. All the algorithms are available in River [25] and were instantiated with their default hyperparameter settings, as implemented in River. For the k-NN models, we set $k = 5$ in all the cases, which is also the default in River. As for SWINN, we selected the same set of hyperparameters reported in Section 5. Table 4 presents a list of the compared algorithms used in the classification and regression tasks.

Next, we provide more details about each experiment carried out and the results obtained for each case study.

6.1. Real application: nowcasting bike sharing supply

An advantage of k-NN models is their ability to aggregate data dynamically. For instance, in a time series nowcasting problem, the

[1] We used the same computer described in our experimental setup.

Table 4
Compared **C**lassification and **R**egression algorithms and their acronyms.

Acronym	Meaning	Reference
LR	Linear Regression (R) or Logistic Regression (C)	–
HAT	Hoeffding Adaptive Tree	[29]
ARF	Adaptive Random Forest	[30,31]
LS(L)	k-NN using LS and a window with L instances	–
SWINN(L)	k-NN using SWINN and a window with L instances	–

Table 5
Results for the bike sharing case study. Linear Regression (LR) was the fastest contender in all the cases, processing all the data in less than one second. The k-NN models yielded competitive MAE results even without using augmented features.

Algorithm	MAE	Memory (MB)	Time (s)
LS(1000)	4.6485	0.7008	12
SWINN(1000)	4.9727	5.2700	9
LR	6.1111	**0.0052**	**<1**
ARF	4.6398	11.2700	11
HAT	5.1989	0.3340	1
LR$^+$	**4.4143**	0.0957	**<1**
ARF$^+$	4.8750	11.3700	13
HAT$^+$	4.5679	0.8984	1

standard approach is to retrieve a window of past values at the same day and/or hour, average the values in the window, and use these lagged averages as features. We consider an application of this setting that concerns the bike availability prediction for the next hour. Intuitively, knowing the average supply of bikes at a bike-sharing station at any given hour gives a good idea of how many bikes might be available in the future at the same hour.

The downside of this approach is that such augmented features need to be specified explicitly. The way a k-NN model operates using online data provides a smart alternative. When queried, the k-NN naturally retrieves samples that are most similar. Indeed, the aggregation window does not have to be explicitly specified. Moreover, the k-NN can also take into consideration exogenous information when searching for similar points in the past, such as the temperature.

We validate this property of k-NN by running a simple experiment. We use historicaldata collected for several bike stations in the city of Toulouse, France. Each data point shows the current number of available bikes in a bike-sharing station, as well as the current weather. We turn this into a nowcasting problem with temporal resampling on the hour. Next, we feed the data into several online models, asking them to forecast the number of bikes available in the next hour. We also benchmarked LR$^+$, HAT$^+$, and ARF$^+$, wherein we augment each model with lagged averages of past values aggregated by the hour of the day. The results are summarized in Table 5.

Concerning MAE, with the exception of ARF, the LS and SWINN models generally outperform the contenders that do not employ extra, handcrafted, lagged aggregations. This fact happens because the remaining models have no ability to aggregate past data, and thus only rely on the current data point, which might not be very informative. ARF is able to achieve competitive performance despite not being able to explicitly aggregate past data. This efficacy comes from its ensemble-based nature, complemented by the data partitioning scheme inherent to decision trees, and the usage of model leaves. In particular, ARF's tree leaves employ LR models to deliver accurate predictions [32].

The model versions augmented with additional features are able to reduce prediction errors compared to their original counterparts. Those gains, however, imply increased computational costs that, although barely noticeable in our case study, might become prohibitive in the long run in a real-world setting. LR, as expected, is the fastest model, regardless of the usage of augmented features. HAT is the second fastest model. ARF is the most memory and time-intensive model, surpassing the computational costs of SWINN. Our proposal is able to deliver similar predictive performance when compared to LS while reducing the running time costs.

In nowcasting problems, it might make sense to deploy one model instance per sensor, e.g., a bike-sharing station. In such cases, the optimal feature engineering and hyperparameter choice is prone to vary from one sensor to another. Not having to create different sets of augmented features is a desired property that can be fulfilled by the usage of k-NN models. SWINN can boost the data processing throughput in such applications when compared with LS.

6.2. Synthetic data: regression

We start by discussing a regression task. For such, we have chosen the Friedman Drift data generator [33], which is available in the River library. We selected the variant with Local and Expanding Abrupt (LEA) concept drifts, denoted by Friedman(LEA). This variant of the data generator implements three concept drifts affecting the feature space locally. The affected portions expand after each drift. Hence, the last concept drift is the most pronounced. We set the drifts to occur after 250 000, 450 000, and 1 500 000 instances.

The selected performance metrics were the Root Mean Squared Error (RMSE) and the coefficient of determination (R^2), due to their popularity and complementary nature. For the former metric, the smaller its value, the better, whereas the opposite happens with the latter metric. The best possible value of R^2 is 1, when the regression model perfectly captures the underlying regression patterns. When R^2 equals 0, there is no correlation between the regressor output and the ground truth.

We added an incremental standard scaling procedure at the beginning of the processing pipeline of each regressor. The default behavior of Hoeffding Tree-based regressors in River is to build model trees, i.e., decision trees whose leaves carry LR models to provide predictions. That is the case with ARF and HAT regressors. The LR and NNS models work better when all the features are on the same scale. Therefore, all regressors had the input data scaled before processing each datum.

We present the results of the regression case study in Fig. 5. We adopted the logarithmic scale for the x-axis due to the large gap in the number of processed instances by the different algorithms. LR and HAT were the fastest algorithms due to their simplicity and efficiency. These were the only two regressors that reached the last concept drift point. Indeed, we can perceive an increase in the predictive error of LR and HAT past 10^6 instances due to concept drift. However, only HAT could start reacting to the drift and gradually reduce the predictive error. LR is not equipped to deal with concept drifts, and its RMSE presents a pronounced increase. The remaining regressors did not reach the last concept drift point.

Table 6 details the differences between the compared regression models. In the table, we can see the large gap between the number of processed instances by LR and the remaining regressors. On the other hand, LR yielded the worst values of RMSE and R^2 among the compared regressors. HAT was around nine times faster than the fastest NNS-based regressor, i.e., SWINN(1000). Tree-based models had the edge in predictive performance, being ARF the most accurate algorithm. LR, HAT, LS(1000), and all the SWINN variants processed more instances than ARF during the allowed time.

By increasing the L parameter, SWINN-based k-NN regressors could surpass LS(1000) in predictive performance and still process a considerable number of instances. Among the k-NN models, LS(5000) was the most accurate, although also the slowest model. In all the cases, SWINN was over four times faster than LS(5000). The SWINN-based k-NN regressors obtained RMSE and R^2 values close to those obtained by the LS variants. By tweaking the values of ϵ and K, the predictive performance gap between LS and SWINN could be reduced, while the speed differences could be increased even further. Notwithstanding, we acknowledge that more efficient and effective graph query strategies must be applied to the search index. The greedy nature of the graph search, allied with a simple strategy to select the query starting point, might cause the search to be stuck in local minima. Still, SWINN has shown potential to be an efficient strategy for performing NNS in regression tasks.

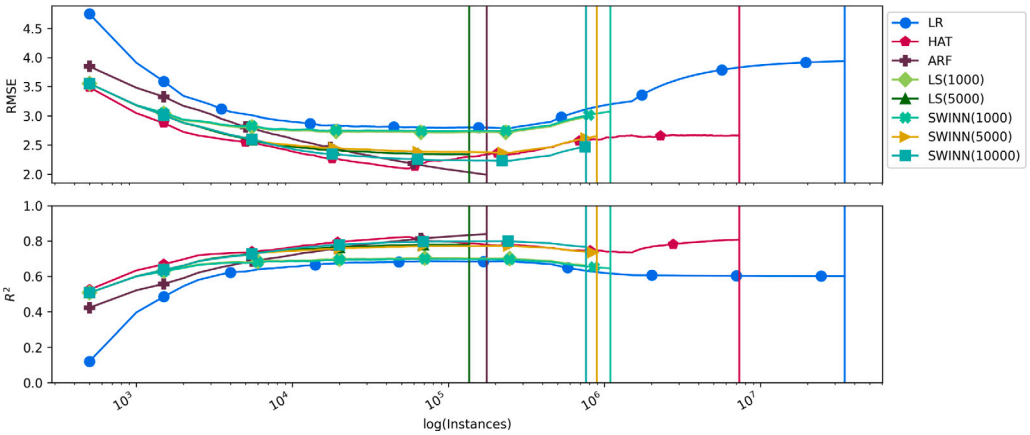

Fig. 5. Results of different regressors when processing an instance of the Friedman(LEA) data generator for one hour. The x-axis is in a logarithmic scale due to the large differences between the compared models. Vertical lines denote the maximum number of instances each model could process during the allowed time.

Table 6
Regression case study using the Friedman(LEA) dataset. The reported RMSE and R^2 values correspond to the measurements after processing incoming data for one hour.

Algorithm	Instances processed	LS(5000) speed up	RMSE	R^2
LR	45 957 500	258.92×	3.94	0.60
HAT	9 762 500	55.00×	2.67	0.81
ARF	231 500	1.30×	1.99	0.84
LS(1000)	949 500	5.35×	2.98	0.66
LS(5000)	177 500	1.00×	2.34	0.78
SWINN(1000)	1 186 500	6.68×	3.07	0.65
SWINN(5000)	871 000	4.91×	2.65	0.73
SWINN(10000)	764 000	4.30×	2.47	0.77

Table 7
The classification case study's results using an instance of the Random RBF with gradual drifts dataset. The reported accuracy and F1 values correspond to the measurements after processing incoming data for one hour.

Algorithm	Instances processed	LS(5000) speed up	Accuracy	F1
LR	4 090 500	134.11×	0.79	0.83
HAT	1 198 000	39.28×	0.89	0.91
ARF	368 000	12.07×	0.92	0.93
LS(1000)	132 500	4.34×	0.97	0.98
LS(5000)	30 500	1.00×	0.97	0.98
SWINN(1000)	317 000	10.39×	0.89	0.91
SWINN(5000)	180 000	5.90×	0.88	0.90
SWINN(10000)	130 500	4.28×	0.89	0.91

6.3. Synthetic data: classification

In the classification case, we relied on the data generator, denoted Random Radial Basis Function (RBF), with gradual concept drifts (RBF-GD). We set up RBF-GD to create 20 features, 2 classes, and 50 micro-clusters from which 10 slowly shift (the change speed was set to 0.01).

Feature scale does not impact the classification version of HAT and ARF. Hence, we only applied feature scaling to LR and the NNS-based classifiers. We know that keeping an incremental feature standardization pipeline component and scaling all the features incurs extra running time costs. Nonetheless, that is the nature of linear and NNS models: feature scale matters. As we aimed for a realistic comparison of the algorithms, we opted to evaluate the classifiers in the same way they would be applied in real-world scenarios.

We present the evolving performance curves for Accuracy and the F1 score in Fig. 6. Once again, LR and HAT were the fastest approaches, as expected. The ARF classifier, differently from its regression counterpart, was able to surpass the k-NN models in terms of the number of processed instances. Hoeffding Trees for classification store counters as split-enabling statistics and rely on highly efficient approximation techniques [34] to evaluate split candidates. Regression trees keep variance estimators as split-enabling statistics and rely on more costly split evaluation procedures due to the continuous nature of the labels. For that reason, the classification version of ARF is faster than the ARF regressor. The same holds when comparing the HAT classifier and regressor.

Looking at the final results in detail, as reported in Table 7, we realize that the performance differences between SWINN and LS were more apparent in classification tasks. We argue that the number of selected features impacted the predictive performance. Also, due to the discrete nature of the labels, the approximation nature of SWINN's NNS might have a higher impact on the final performance. This observation comes from the fact that the most common label among the returned neighbors is the final answer. In regression, the average label value is the output. Hence, the influence of false positives in a search query might be more pronounced in classification tasks.

With the increase in dimensions, hubness [23] might also impact the graph structure. The curse of dimensionality might also influence the greedy search applied in SWINN. The influence of the number of features on performance has been addressed by NN-Descent authors [15] and subsequent works [22,23]. It remains an open issue to further study in online ML scenarios. Increasing the value of ϵ may increase SWINN's recall in these cases at the cost of slightly higher running time. Even in such settings, SWINN ought to be faster than performing an LS of the data buffer when the value of L is high, as the final results reported in Table 7 hint at. Even when using $L = 10000$, our proposal processed almost the same number of instances as the LS(1000) The differences in predictive performance between LS and SWINN give us more pieces of evidence to support focusing on graph search as the next step. In cases where the high dimensionality is an impeding factor to perform incremental NNS, a possible solution is to apply dimensionality reduction techniques [19,35] as a data processing step prior to the NNS graph construction.

7. Final considerations

In this paper, we proposed Sliding Window-based Incremental Nearest Neighbors (SWINN), an online and graph-based nearest neighbor search (NNS) algorithm. SWINN is primarily meant to work in a sliding window regimen, where new instances arrive, and the old ones are discarded. Nonetheless, any vertex of SWINN's search graph can be removed, and new vertices can be added. Therefore, other types of data ingestion can also be explored.

Our experiments show that SWINN effectively deals with online NNS when the sliding window size is sufficiently large ($L \geq 1000$). SWINN is significantly faster than a complete linear scan (LS) of the

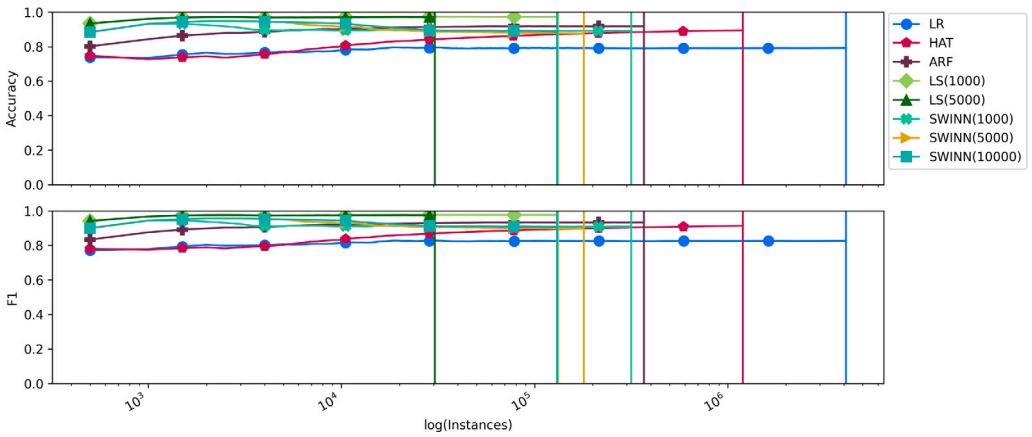

Fig. 6. Results of different classifiers when processing an instance of the RBF-GD for one hour. The x-axis is a logarithmic scale due to the large differences between the compared models. Vertical lines denote the maximum number of instances each model was able to process during the allowed time.

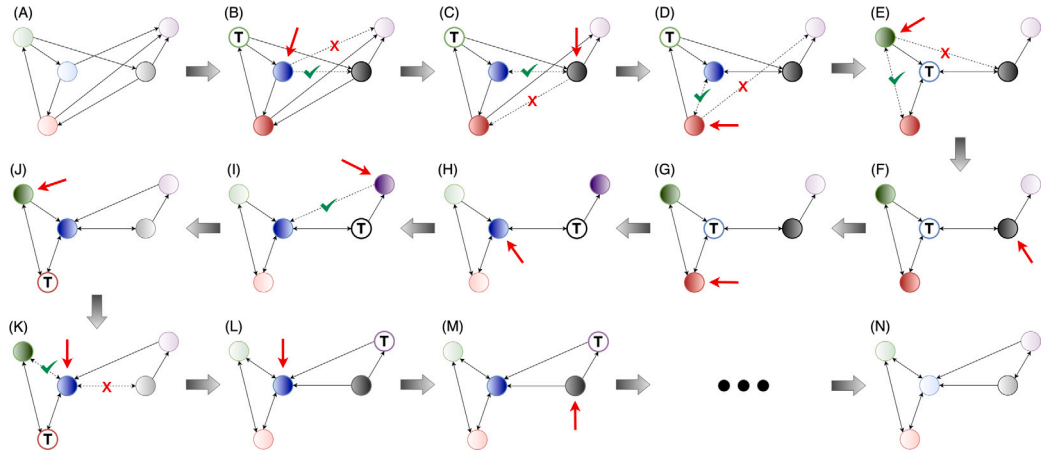

Fig. A.1. An example of the SWINN refinement procedure. From steps A to M, one iteration of neighborhood refinement is illustrated. Step N shows the search index after the neighborhood refinement converges. Vertices with a bold **T** are the target during the refinement steps. Vertices with a darker color shade are the neighbors of the target vertex, i.e., the vertices involved in the neighborhood join. Arrows indicate the vertex whose neighbors are updated in each step.

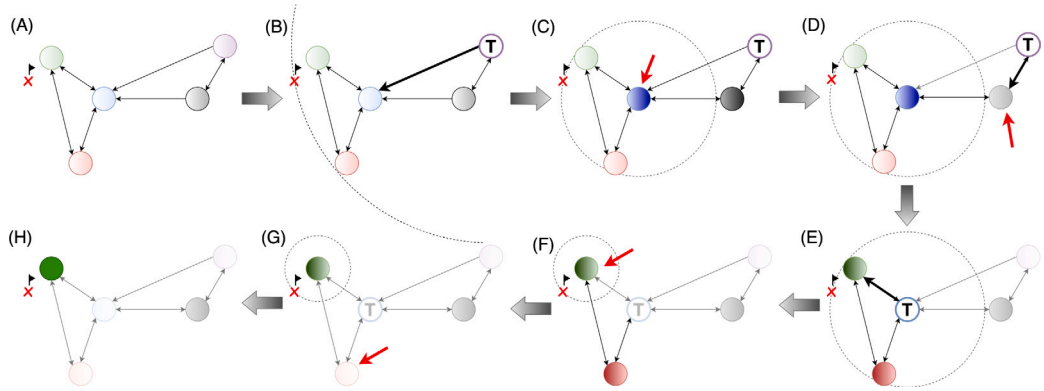

Fig. A.2. Illustration of the search procedure on an already constructed search graph (step A). The search procedure starts on a random seed vertex (step B). The dashed circle represents the distance bound given by the current best solution. Candidates whose distance to the query point (represented by x) is larger than the distance bound are ignored.

sliding window while keeping competitive search recall to the LS approach. Furthermore, SWINN can deal with arbitrary distance metrics and copes with concept drift similarly to the LS strategy. Our proposal is also effective when applied to online ML tasks, from which we give classification and regression examples. We compare SWINN-based k-NN models against popular online classification and regression algorithms.

In future work, we intend to further explore search strategies in SWINN. SWINN relies on a simple bounded greedy search to perform graph searches. Moreover, search initialization is performed by selecting a single random vertex as the starting point. Both the search strategy and the search initialization can be further improved. An example would be using a multi-vertex search start and exploring additional search heuristics to perform graph traversal.

Additionally, applying SWINN as the building block of more complex k-NN-based solutions could be an exciting venue to explore. For instance, our proposal could be used in multi-memory solutions, where

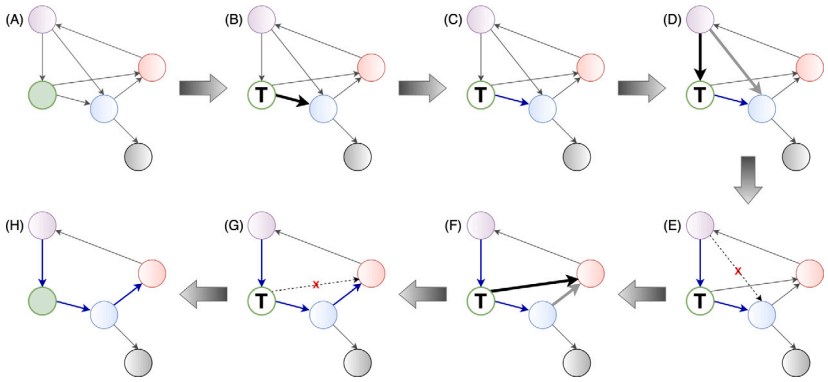

Fig. A.3. Illustration of the edge pruning capabilities starting from the green vertex. Starting from the top left to the bottom right: each edge of green is selected, and the smallest are kept. Edges (green, red) and (purple, blue) are removed by the procedure. Note that two hops will be needed to reach red starting from green after removing the edges. Similar cases happen for the other affected vertices.

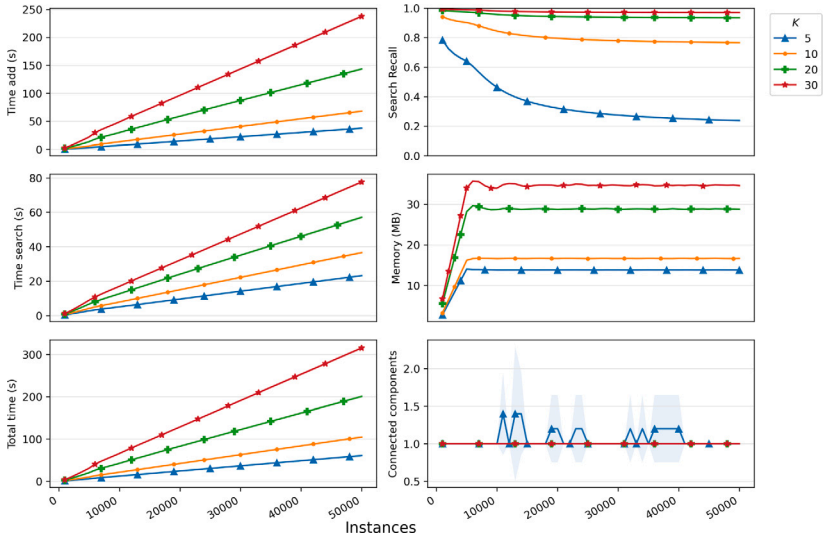

Fig. A.4. Results obtained when varying K and using sliding windows of length 5000.

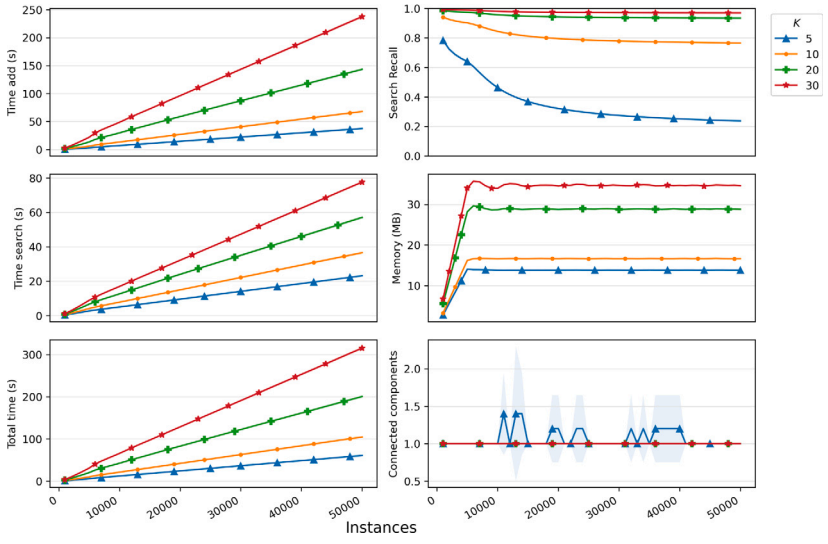

Fig. A.5. Results obtained when varying $\texttt{max}_c$ and using sliding windows of length 1000.

two levels of data buffering are applied, namely, short and long-term memories. The short-term memory holds the most recent data items in a sliding window fashion, while a larger window, possibly populated via reservoir sampling, carries examples of old concepts. Decisions are taken by performing NNS in both windows and combining the answers found in each memory level. Last, we intend to explore other

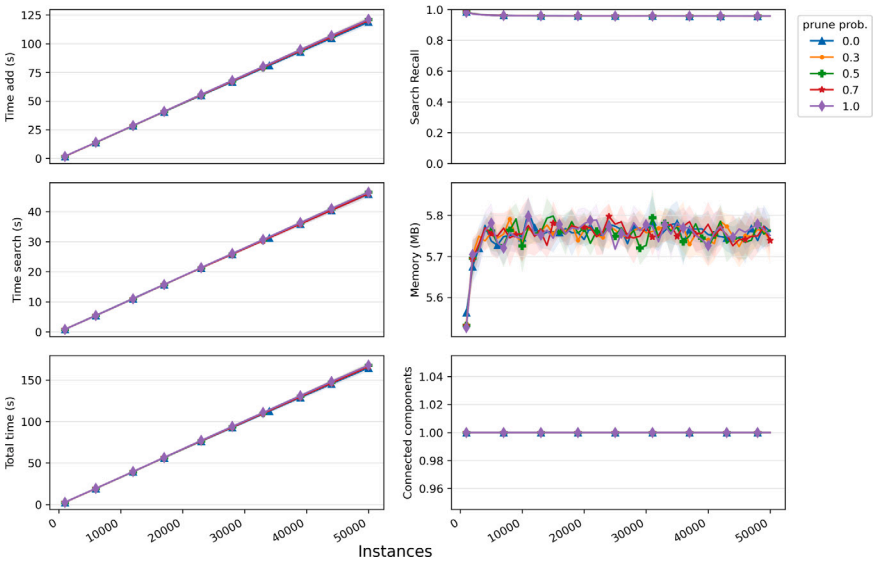

Fig. A.6. Results obtained when varying $\texttt{prune}_{\text{prob}}$ and using sliding windows of length 1000.

distance metrics and investigate how the choice of the metric impacts the performance of online ML tasks.

CRediT authorship contribution statement

Saulo Martiello Mastelini: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Data curation, Writing – original draft, Visualization. **Bruno Veloso:** Conceptualization, Validation, Writing – review & editing. **Max Halford:** Conceptualization, Software, Writing – review & editing. **André Carlos Ponce de Leon Ferreira de Carvalho:** Methodology, Validation, Resources, Supervision, Writing – review & editing. **João Gama:** Conceptualization, Methodology, Validation, Resources, Supervision, Writing – review & editing, Project administration.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

All the data used is publicly available and the code will be made publicly available in the River library.

Acknowledgments

This research was supported by the São Paulo Research Foundation, grant #2021/10488-7. The experiments were carried out using the computational resources from the Center for Mathematical Sciences Applied to Industry (CeMEAI) funded by FAPESP (grant 2013/07375-0).

Appendix

A.1. SWINN application examples

In this section we present illustrative toy examples for some of SWINN's main operations. Namely, graph refinement and search, and edge pruning.

A.1.1. Graph refinement

Fig. A.1 presents an example illustrating how the refinement procedure works in SWINN. We start our example in step A, where an initial graph built with $K = 2$ is presented. In this state, the vertices are not necessarily linked to their NNs. In step B, the vertex green is the target vertex, and its neighbors (direct and reverse) are selected for the neighborhood join. In steps B, C, and D, the edges of vertices blue, gray, and red, respectively, are updated. In all the cases, better neighbors are selected by computing the distances from the vertex in focus (signaled by the red arrow) to the other vertices involved in the neighborhood join. SWINN deletes previous edges to make room for the new ones in case the new options are closer in the distance.

The refinement proceeds by shifting the target vertex to blue in step E. Similarly, the neighborhood of vertex green is improved in step E. However, in steps F and G, the existing connections do not change, as gray and red are already connected to their best neighboring vertices. The same happens to blue when gray becomes the target vertex (step H). At this point, purple has only one reverse neighbor (gray) and no direct neighbors. Thus, in step I, purple adds a new edge to blue, as the latter is the only available option.

During step J, red becomes the new target. At this stage, the neighborhood does not change for green since blue is already its direct neighbor. Next, blue adds an edge to green in step K. SWINN does not limit reciprocal edges between vertices. The benefits of applying such a constraint to the size of the edge set are surpassed by the additional actions needed to keep the NN connections as more data are monitored.

Notwithstanding, if it were not by the edge (purple, blue), in step K, two sub-graphs would be generated by removing the connection (blue, gray). This situation is problematic and might happen when K is not sufficiently large. The search can fail if there is more than a single connected component, as not every vertex is accessible from any given starting point in the search index. We will discuss the implications of the choice of K in the experimental results.

Steps L and M also do not imply changes in the search index, as the existing connections are already the best options compared to the joined vertices. At the end of step M, all the existing vertices acted as the reference vertex. Therefore, one iteration of the refinement process was performed. The process is repeated until the convergence criteria are met. Step N shows the example graph after convergence.

A.1.2. Graph search

We illustrate a 1-NN search query in Fig. A.2, using $\epsilon = 0$. In step A, we take the same graph obtained after the refinement performed in the

last illustrative example. The query instance is signaled by x on the left side of the graphs. The search starts in step B by randomly choosing a starting seed vertex (the purple vertex). To reduce the search space, SWINN defines a search bound, given by the distance from the current best solution to the query. Step B represents the search bound by a circle centered in purple. The circle's radius corresponds to the distance from purple to the query point. We can relax the definition of the search bound to decrease the chance of falling into local minima during the search. To do so, one must select $\epsilon > 0$.

The search procedure proceeds by selecting the best option among the current neighbors of the starting vertex. In our example, blue is the closest available vertex. In step C, we shift our focus to blue and update the distance bound. The vertex gray was not yet explored (step D), as it is also a neighbor of the seed vertex. However, the distance between gray and the query is larger than the updated distance bound. Therefore, gray is not further explored.

Next, in step E, blue becomes the target vertex, the current best solution. The search continues by exploring the neighbors of blue, which were not yet visited (green and red). The first selected neighbor is green. SWINN's vertices keep an unordered list of neighbors, so, in our example, green is first selected for illustration purposes. The red vertex could also come first if it were the first element in the list of total neighbors. As green is closer than blue to the query point, the distance bound is updated in step F. There is only one remaining vertex to explore (step G), i.e., red. Nevertheless, the search ignores this vertex because the distance of red to the query is larger than the current distance bound.

The search process stops in step H when green is selected as the query result. It must be observed that, in this toy example, all the nodes were visited during the search. Thus, the search cost was asymptotically equivalent to a LS. However, in practice, the number of available vertices is much higher, and the combination of the greedy search strategy and the distance bound significantly reduces the search space and speedup graph traverse.

In the illustrated search process, the distance bound is the distance from the best vertex found so far to the query item. Nonetheless, by relying on this approach, the search process might end up in local minima. For example, suppose there was an even closer vertex to the query only accessible via red. The search would not reach this vertex by using the strict distance bound. The ϵ parameter is defined to avoid such pitfalls and decrease the change of stopping the search at local minima.

A.1.3. Edge pruning

We illustrate how the pruning procedure works with Fig. A.3. The figure shows a hypothetical search graph where edge pruning is applied to the green vertex, which we refer to as the focus vertex. Step A shows the starting search index. At the beginning of the procedure, we create an empty list of selected neighbors. All the existing neighbors of the focus vertex are candidates for either removal from or addition to the list of selected neighbors. Only the selected neighbors will remain at the end of the pruning procedure. We compare each neighbor of green against the list of selected neighbors accordingly to their distance. If a candidate vertex is also a neighbor of one vertex in the list of selected vertices, only the best edge among two compared vertices is kept.

In step B of Fig. A.3, the best available edge, blue, is selected for analysis (highlighted in **bold**). As there are no selected neighbors yet, blue is added to the list of selected neighbors (step C). Next, in step D, the second best edge (purple, green) is analyzed, and the list of selected nodes is checked for possible redundant paths. In fact, purple is also a neighbor of blue, which was previously selected. As the edge between purple and green is shorter than that between purple and blue, the latter edge is removed (step E). In step F, the third and last neighbor of green is selected (red) for analysis. Indeed, red is also a neighbor of blue. Again, the shortest edge among the two compared vertices is kept, leading to the removal of red as a neighbor of green (step G). The procedure stops at step H, as there are no more edges to explore from the focus vertex.

A.2. Additional experimental results

This section presents additional results obtained during our empirical evaluation of SWINN.

A.2.1. Impact of K in the search graph for windows of 5000 instances

We present the results obtained when varying the values of K and using $L = 5000$ in Fig. A.4.

A.2.2. Impact of max_c in the search graph for windows of 1000 instances

Fig. A.5 presents the results obtained when varying the values of $\texttt{max}_c$ and using $L = 1000$.

A.2.3. The impact of $prune_{prob}$ in the search graph for windows of 1000instances

Fig. A.6 presents the results obtained when varying the values of $\texttt{prune}_{\text{prob}}$ and using $L = 1000$.

References

[1] Martin Aumüller, Erik Bernhardsson, Alexander Faithfull, Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms, Inf. Syst. 87 (2020) 101374.

[2] Yixi Cai, Wei Xu, Fu Zhang, Ikd-tree: An incremental kd tree for robotic applications, 2021, arXiv preprint arXiv:2102.10808.

[3] Larissa C. Shimomura, Rafael Seidi Oyamada, Marcos R. Vieira, Daniel S. Kaster, A survey on graph-based methods for similarity searches in metric spaces, Inf. Syst. 95 (2021) 101507.

[4] Pedro Domingos, Every model learned by gradient descent is approximately a kernel machine, 2020, arXiv preprint arXiv:2012.00152.

[5] Mayur Datar, Nicole Immorlica, Piotr Indyk, Vahab S. Mirrokni, Locality-sensitive hashing scheme based on p-stable distributions, in: Proceedings of the Twentieth Annual Symposium on Computational Geometry, 2004, pp. 253–262.

[6] Herve Jegou, Matthijs Douze, Cordelia Schmid, Product quantization for nearest neighbor search, IEEE Trans. Pattern Anal. Mach. Intell. 33 (1) (2010) 117–128.

[7] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya Razenshteyn, Ludwig Schmidt, Practical and optimal lsh for angular distance, 2015, arXiv preprint arXiv:1509.02897.

[8] Yusuke Matsui, Yusuke Uchida, Hervé Jégou, Shin'ichi Satoh, A survey of product quantization, ITE Trans. Media Technol. Appl. 6 (1) (2018) 2–10.

[9] Roberto Souto Maior de Barros, Silas Garrido Teixeira de Carvalho Santos, Jean Paul Barddal, Evaluating k-nn in the classification of data streams with concept drift, 2022, arXiv preprint arXiv:2210.03119.

[10] Viktor Losing, Barbara Hammer, Heiko Wersing, Tackling heterogeneous concept drift with the self-adjusting memory (sam), Knowl. Inf. Syst. 54 (1) (2018) 171–201.

[11] Viktor Losing, Barbara Hammer, Heiko Wersing, Albert Bifet, Randomizing the self-adjusting memory for enhanced handling of concept drift, in: 2020 International Joint Conference on Neural Networks, IJCNN, IEEE, 2020, pp. 1–8.

[12] Jaemin Jo, Jinwook Seo, Jean-Daniel Fekete, Panene: A progressive algorithm for indexing and querying approximate k-nearest neighbors, IEEE Trans. Vis. Comput. Graphics 26 (2) (2018) 1347–1360.

[13] Donna Xu, Ivor W. Tsang, Ying Zhang, Online product quantization, IEEE Trans. Knowl. Data Eng. 30 (11) (2018) 2185–2198.

[14] Qi Liu, Jin Zhang, Defu Lian, Yong Ge, Jianhui Ma, Enhong Chen, Online additive quantization, in: Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining, 2021, pp. 1098–1108.

[15] Wei Dong, Charikar Moses, Kai Li, Efficient k-nearest neighbor graph construction for generic similarity measures, in: Proceedings of the 20th International Conference on World Wide Web, 2011, pp. 577–586.

[16] Jerome H. Friedman, Jon Louis Bentley, Raphael Ari Finkel, An algorithm for finding best matches in logarithmic expected time, ACM Trans. Math. Softw. 3 (3) (1977) 209–226.

[17] Stephen M. Omohundro, Five Balltree Construction Algorithms, International Computer Science Institute, Berkeley, 1989.

[18] Yibin Sun, Bernhard Pfahringer, Heitor Murilo Gomes, Albert Bifet, Soknl: A novel way of integrating k-nearest neighbours with adaptive random forest regression for data streams, Data Min. Knowl. Discov. 36 (5) (2022) 2006–2032.

[19] Saquib Sarfraz, Marios Koulakis, Constantin Seibold, Rainer Stiefelhagen, Hierarchical nearest neighbor graph embedding for efficient dimensionality reduction, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2022, pp. 336–345.

[20] Yu A. Malkov, Dmitry A. Yashunin, Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs, IEEE Trans. Pattern Anal. Mach. Intell. 42 (4) (2018) 824–836.

[21] Zhaozhuo Xu, Weijie Zhao, Shulong Tan, Zhixin Zhou, Ping Li, Proximity graph maintenance for fast online nearest neighbor search, 2022, arXiv preprint arXiv:2206.10839.

[22] Brankica Bratić, Michael E. Houle, Vladimir Kurbalija, Vincent Oria, Miloš Radovanović, Nn-descent on high-dimensional data, in: Proceedings of the 8th International Conference on Web Intelligence, Mining and Semantics, 2018, pp. 1–8.

[23] Brankica Bratić, Michael E. Houle, Vladimir Kurbalija, Vincent Oria, Miloš Radovanović, The influence of hubness on nn-descent, Int. J. Artif. Intell. Tools 28 (06) (2019) 1960002.

[24] Joseph B. Kruskal, On the shortest spanning subtree of a graph and the traveling salesman problem, Proc. Amer. Math. Soc. 7 (1) (1956) 48–50.

[25] Jacob Montiel, Max Halford, Saulo Martiello Mastelini, Geoffrey Bolmier, Raphael Sourty, Robin Vaysse, Adil Zouitine, Heitor Murilo Gomes, Jesse Read, Talel Abdessalem, et al., River: machine learning for streaming data in python, J. Mach. Learn. Res. 22 (110) (2021) 1–8.

[26] Avrim Blum, Adam Kalai, John Langford, Beating the hold-out: Bounds for k-fold and progressive cross-validation, in: Proceedings of the Twelfth Annual Conference on Computational Learning Theory, 1999, pp. 203–208.

[27] Joao Gama, Raquel Sebastiao, Pedro Pereira Rodrigues, Issues in evaluation of stream learning algorithms, in: Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2009, pp. 329–338.

[28] Gautam Bhattacharya, Koushik Ghosh, Ananda S. Chowdhury, Outlier detection using neighborhood rank difference, Pattern Recognit. Lett. 60 (2015) 24–31.

[29] Albert Bifet, Ricard Gavalda, Adaptive learning from evolving data streams, in: International Symposium on Intelligent Data Analysis, Springer, 2009, pp. 249–260.

[30] Heitor M. Gomes, Albert Bifet, Jesse Read, Jean Paul Barddal, Fabrício Enembreck, Bernhard Pfharinger, Geoff Holmes, Talel Abdessalem, Adaptive random forests for evolving data stream classification, Mach. Learn. 106 (9) (2017) 1469–1495.

[31] Heitor Murilo Gomes, Jean Paul Barddal, Luis Eduardo Boiko Ferreira, Albert Bifet, Adaptive random forests for data stream regression, in: ESANN, 2018.

[32] Saulo Martiello Mastelini, Felipe Kenji Nakano, Celine Vens, André Carlos Ponce de Leon Ferreira, et al., Online extra trees regressor, IEEE Trans. Neural Netw. Learn. Syst. (2022).

[33] Elena Ikonomovska, Joao Gama, Sašo Džeroski, Learning model trees from evolving data streams, Data Min. Knowl. Discov. 23 (1) (2011) 128–168.

[34] Bernhard Pfahringer, Geoffrey Holmes, Richard Kirkby, Handling numeric attributes in hoeffding trees, in: Pacific-Asia Conference on Knowledge Discovery and Data Mining, Springer, 2008, pp. 296–307.

[35] Leland McInnes, John Healy, James Melville, Umap: Uniform manifold approximation and projection for dimension reduction, 2018, arXiv preprint arXiv:1802.03426.