Graph based kNN (XXYYZZ)

Contents

1 Concrete algorithm

Given a set of labels $\mathcal{L}$ and the dataset $S = \left\{(\boldsymbol{s}^i, \boldsymbol{y}^i) \mid \boldsymbol{s}^i \in \mathbb{R}^n, \boldsymbol{y}^i \in \mathcal{L}\right\}$ and the input sample $\boldsymbol{x} \in \mathbb{R}^n$, XXYYZZ : $\mathbb{R}^n \to \mathcal{P}(\mathcal{L})$ is function that classify the input with the most frequent label within the k closest samples to the input like the kNN algorithm but differs from it in the way those samples are found. Rather than sort the samples in oder of their proximity to the input and then select the first k samples, XXYYZZ use the relation of "being closer to the input" between pair of samples w.r.t. the euclidean norm to select the k closest samples. The relation is defined as follow:

Definition 1.1 (Relation $\preccurlyeq_{\boldsymbol{x}}$): Given the points $\boldsymbol{x}, \boldsymbol{s}_i, \boldsymbol{s}_j \in \mathbb{R}^n$,

$$\boldsymbol{s}_i \preccurlyeq_{\boldsymbol{x}} \boldsymbol{s}_j \iff \|\boldsymbol{x} - \boldsymbol{s}_i\| \leq \|\boldsymbol{x} - \boldsymbol{s}_j\|$$

The subscript $\boldsymbol{x}$ will be omitted when is clear from the context.

XXYYZZ first build a "precedence" graph $\mathbb{G}$ in which nodes represents the samples in S and edges model the relation $\preccurlyeq_x$ (i.e. the tail is closer to the input than the head) and then classify the input with the most frequent labels within the vertices composing the valid paths[1] of length $k - 1$ starting from the samples closest to the input.

1.1 Precedence graph

Given the set of samples S the precedence graph $\mathbb{G}_{\boldsymbol{x}} = (V, E)$ is a graphical representation of the totally ordered set $(S, \preccurlyeq_x)$>

- $V = S$
- $E = \left\{(\boldsymbol{s}_i, \boldsymbol{s}_j) \mid \boldsymbol{s}_i \preccurlyeq \boldsymbol{s}_j \text{ is true}\right\}$

where $\|\cdot\|_p$ is the minkowski norm.

The graph is implemented using adjacent lists so each vertex object has a list of adjacent vertices. Moreover given a vertex $\boldsymbol{s}$:

- `pred`$(\boldsymbol{s})$ denotes the set of vertices $\boldsymbol{s}_i$ such that there is an edge from $\boldsymbol{s}_i$ to $\boldsymbol{s}$ but not the other way around that is

$$(\boldsymbol{s}_i, \boldsymbol{s}) \in E \wedge (\boldsymbol{s}, \boldsymbol{s}_i) \notin E$$

- `same_dist`$(\boldsymbol{s})$ represents the set of vertices $\boldsymbol{s}_i$ such that there is a bidirectional edge between $\boldsymbol{s}_i$ and $\boldsymbol{s}$ that is

$$(\boldsymbol{s}_i, \boldsymbol{s}) \in E \wedge (\boldsymbol{s}, \boldsymbol{s}_i) \in E$$

This means that

$$\forall \boldsymbol{s}_j \in \texttt{pred}(\boldsymbol{s}) \ \boldsymbol{s}_j \preccurlyeq \boldsymbol{s} \wedge \boldsymbol{s} \not\preccurlyeq \boldsymbol{s}_j$$

This observation will be used for the generation of valid paths based on how close the samples are to the input sample.

Algorithm 1 shows how the precendence graph is created. Given the sample dataset S and the input sample $\boldsymbol{x}$, it first create the graph $\mathbb{G}$ with only the vertices habing no adjacent vertex. attributes set to the (line `1`). Then for each unordered pair of samples $(\boldsymbol{s}_i, \boldsymbol{s}_j)$ the closer sample to the input $\boldsymbol{x}$ w.r t. the euclidean norm is determined and update the adjacent lists of both vertices accordingly (line `2-12`). Finally the created graph is returned (line `15`).

[1]The definition of a valid is given later in Section 1.2

Algorithm 1: `create_precedence_graph` method

```
Input S: samples dataset x: the input sample
Output $\mathbb{G}_{\boldsymbol{x}}$: precedence graph
1  $\mathbb{G}_{\boldsymbol{x}}$ ← initialize_graph(S)
2  for $(s_i, s_j)$ in $\{(s_i, s_j) |\ s_i, s_j \in \mathtt{S}, s_i \neq s_j\}$ do
3      if $s_i \preccurlyeq s_j$ and $s_j \preccurlyeq s_i$ then
4          add $s_j$ to adjacent of $s_i$
5          add $s_i$ to adjacent of $s_j$
6      else if $s_i \preccurlyeq s_j$ then
7          add $s_j$ to adjacent of of $s_i$
8          else
9              add $s_i$ to adjacent of of $s_j$
10             end
11         end
12         return $\mathbb{G}_{\boldsymbol{x}}$
```

Example 1.1: Consider a dataset $S \subset \mathbb{R}^2 \times \mathcal{L}$ composed by the following samples

- $\boldsymbol{a} = ((0, 1), 1)$
- $\boldsymbol{b} = ((2, 0), -1)$
- $\boldsymbol{c} = ((2.25, 0), 1)$

and the input sample $\boldsymbol{x} = (1, 1)$. In this setup the samples $\boldsymbol{a}$ and $\boldsymbol{b}$ are equidistant from the input $\boldsymbol{x}$ while $\boldsymbol{c}$ is the furthest one so we have the following precedence graph:

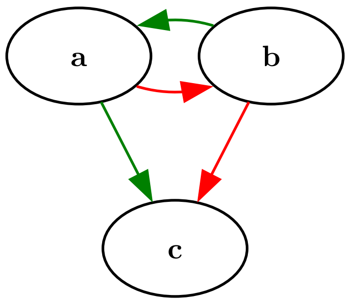


Figure 1: Example of precedence graph.

1.2 Paths generation

In order to have a sound classification the paths within the precedence graph used to classify the input must satisfy all the proximity relation among the samples that is if $\boldsymbol{s}_i \preccurlyeq \boldsymbol{s}_j$ but $\boldsymbol{s}_j \not\preccurlyeq \boldsymbol{s}_i$ then every path in which $\boldsymbol{s}_j$ occurs must also contain $\boldsymbol{s}_i$ and it must precede $\boldsymbol{s}_j$. This means that not all paths starting from the closest sample can be used to classify the input. For example in the graph of Example 1.1 the path $[\boldsymbol{b}, \boldsymbol{c}]$ is not a valid because $\boldsymbol{a} \preccurlyeq \boldsymbol{c}$ but $\boldsymbol{c} \not\preccurlyeq \boldsymbol{a}$ so this means that in every valid path $\boldsymbol{c}$ must be preceded by $\boldsymbol{a}$. This observation leads to the following definition fo a valid path:

Definition 1.2.1 (Valid path): A path $\mathcal{P}$ in a precedence graph is *valid* if and only if $\forall \boldsymbol{s}_i \in \mathcal{P}$:

$$\forall \boldsymbol{s}_j \in \texttt{pred}(\boldsymbol{s}_i)\ \boldsymbol{s}_j \text{ is a predecessor of } \boldsymbol{s}_i \text{ in } \mathcal{P}$$

By this definition in the graph of Example 1.1 only the edges highlighted with the same color form valid paths which are:

- $[\boldsymbol{b}, \boldsymbol{a}, \boldsymbol{c}]$
- $[\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}]$

Prop 1.1: Every valid path starts with a vertex v_i such that the sample $\boldsymbol{s}_i$ is one of the closest sample to the input $\boldsymbol{x}$

Proof: Follows directly from the definition of valid path. □

Prop 1.2: If a vertex $\boldsymbol{s}$ do not occur in a valid path $\mathcal{P}$ and every vertex in $\texttt{pred}(\boldsymbol{s})$ is in $\mathcal{P}$ then the path $\mathcal{P} + [\boldsymbol{s}]$ is still valid.

Proof: Follows directly from the definition of valid path. □

Definition 1.2.2 (safe vertex): Given the valid path $\mathcal{P} = [\boldsymbol{s}_0, \boldsymbol{s}_1, ..., \boldsymbol{s}_m]$ the vertex $\boldsymbol{s}$ is *safe* for $\mathcal{P}$ if the path $\mathcal{P} + [\boldsymbol{s}]$ is a valid path.

Prop 1.3: Valid paths of length $k - 1$ starting from vertices $\boldsymbol{s}_i$ such that $\texttt{pred}(\boldsymbol{s}) = \emptyset$ contains the k closest samples from the input.

Proof: Follows directly from the definitions of a path in a graph and valid paths. □

The generation of paths is done by traversing the graph in the same fashion of the BFS algorithm while respecting the condition of a valid path. Algorithm 2 shows how paths are generated given the the precedence graph $\mathbb{G}_{\boldsymbol{x}}$ and the desired length n of the path. The algorithm make use of a FIFO queue (the variable `queue`) to maintain the list of all valid paths of length less than $k < n$. First it initialize the queue with the vertices $\boldsymbol{s}$ such that $\texttt{pred}(\boldsymbol{s}) = \emptyset$ (lines `1-2`) which means the traversal starts from the samples closest to the input $\boldsymbol{x}$. A counter k which denotes the length of the paths in the queue is initialized with 0 (line `3`). Then until the queue is not empty extracts all the paths present in the queue and check whether their length is equal to desired length (i.e the input $\mathbf{n}$) and if this is the case the it simply returns the extracted paths (lines `6-7`) otherwise iteratively extend each path with every safe vertex within the adjacent of the path's last sample and add the extended paths to the queue for the next iteration of the loop (lines `8-15`). Before the next loop the counter k is also incremented to reflect the length of the paths in the queue (line `16`).

Prop 1.4: Given as input the the precedence graph $\mathbb{G}_{\boldsymbol{x}}$ and $n \in \mathbb{N}$ Algorithm 2 returns all the valid paths of length[2] n within $\mathbb{G}_{\boldsymbol{x}}$.

Proof: The proposition can be proved by showing, that at the k-th iteration of the while loop at line 5 (ie. before the check on the length of the paths on the queue), the queue contains all the valid path of length $k - 1$. We use induction on k to prove this:

- ($k = 1$): In the first iteration of the loop the queue contains all the paths composed vertices $\boldsymbol{s}$ such that $\texttt{pred}(\boldsymbol{s}) = \emptyset$ which are of length 0 and surely are valid by definition.

[2]The convention used for path of length 0 is that it contains only the starting vertex.

Algorithm 2: `generate_paths` method

```
Input $\mathtt{G_x}$: precedence graph, var(n): length of the path
Output Set of valid paths
1  closest_samples ← $\{[s_i] \mid s_i \in \mathtt{G_x} \wedge \mathtt{pred}(s_i) = \emptyset\}$
2  queue ← create_queue(closest_samples)
3  k ← 0
4  while queue not empty do
5      current_paths ← queue.popAll()
6      if k = n then
7          return current_paths
8      for current_path in current_paths do
9          last_vertex ← current_path.last()
10         for adj in last_vertex.adjacent do
11             if adj is safe for current_path then
12                 queue.append(current_path + [adj])
13             end
14         end
15     end
16     k ← k + 1
17 end
```

- $(k = h + 1)$: At iteration $h + 1$, the paths inside the queue at line 5 are obtained by extending all the paths in the queue in the h-th iteration with every safe vertex among the adjacent of the path's last vertex. By induction hypothesis the queue in the h-th iteration contains all the valid paths of length $h - 1$ and so in the $h + 1$ iteration, by definition of safe vertex, the queue contains all the valid paths of length h.

This means when the method returns, at iteration n, the queue contains all the possible valid path of length $n - 1$. □

1.3 XXYYZZ algorithm

Algorithm 3 shows the pseudocode of the algorithm. Given a set of samples S and the input sample $\boldsymbol{x}$ the algorithm first build the precedence graph $\mathbb{G}_{\boldsymbol{x}}$ (line 1). Then the `generate_paths` method is used to find the valid paths of length length $k - 1$ within the graph (line 2). Afterwards the set containing the most frequent labels within the samples composing each path is computed and returned as the possible classifications of the input sample (line 3-8).

Since there could be multiple valid paths in the graph due to samples in S equidistant to the input, the latter could be classified with different labels hence the need to return a set of labels rather than a single value.

Theorem 1.1: Given a dataset $S = \{(\boldsymbol{s}_i, \boldsymbol{y}_i) \mid \boldsymbol{s}_i \in \mathbb{R}^n, \boldsymbol{y}_i \in \mathbb{R}\}$ and the input sample $\boldsymbol{x}$, XXYYZZ returns all the possible classifications of the input $\boldsymbol{x}$.

Algorithm 3: XXYYZZ algoritm

```
    Input  x: the input sample        S: samples dataset
           k: number of neighbours
    Output Set of possible classification of $x$
1   $\mathbb{G}_{\boldsymbol{x}} \leftarrow$ create_precedence_graph($S$, $x$)
2   paths $\leftarrow$ generate_paths($\mathbb{G}_{\boldsymbol{x}}$, $k-1$)
3   classification $\leftarrow \{\}$
4   for path in paths do
5       labels $\leftarrow$ most frequent labels in path
6       classification $\leftarrow$ labels $\cup$ classification
7   end
8   return classification
```

Proof: XXYYZZ compute the set of possible classification by finding the most frequent labels in the paths returned by the generate_paths method which, by Prop 1.3 and Prop 1.4, returns all the possible k closest samples to the input. Therefore XXYYZZ surely returns all the possible classification of the input $\boldsymbol{x}$. □

Example 1.2: Consider the setup of Example 1.1. Running XXYYZZ with $k = 3$ will return the singleton $\{1\}$ because the most frequent label in all the valid path is exactly 1 meanwhile with $k = 2$ the result is the set $\{1, -1\}$ because there is a ties in both paths.

2 Graph construction optimization

One problem with XXYYZZ is that it not really efficient because it requires the iteration over all the pairs of samples for the construction of the graph which in the worst case can be fully connected. So the overall complexity of the algorithm is $O(n^2)$ where n is number of samples in the dataset but hides an important constant which is the dimension of the samples.

One way to optimize the construction of the graph is reduce the number of samples used to create the graph since not all are needed for the classification. One method to reduce the number of samples is the following:

1. Partition the the dataset so that each partition contain at most a m of samples where $k \leq m \ll n$.
2. Find the partition P containing the input x.
3. Compute the distance r between x and the k-th closest sample to x in P.
4. Find the partitions PS intersecting the hypersphere centered in x and radius r.
5. Build the graph with the points in partitions PS that are inside the hypershpere.

2.1 Dataset partitioning

The dataset is partitioned with a binary space partition tree (BSP-Tree) [1] using random projection [2] like is done in the ANNOY tool [3]. The main idea is to split the hyperspace along a random hyperplane which will in turn split the dataset in two and then recursively split again the two half spaces in the same manner until the subspace can't be further halved because it contains the minimum number of samples. With this procedure a BSP-Tree is built in which leafs are the partitions of the space while the internal nodes split the space according to the random hyperplane. Algorithm 4 shows how the BSP-Tree is constructed. Given the dataset S and the minimum size of a partition m, it first check whether the dataset has size less than m

Algorithm 4: `build_bsp_tree` method

```
Input S: samples dataset, m: the minimum size of a partition
Output BSP-Tree
1 if |S| ≤ m then
2     return Leaf(S)
3 end
4 left_dataset, right_dataset, hyperplane ← split_dataset(S)
5 left_tree ← build_bsp_tree(left_dataset, m)
6 right_tree ← build_bsp_tree(right_dataset, m)
7 return Node(hyperplane, left_tree, right_tree)
```

in which is case just return a tree made of a single leaf initialized with the dataset (line `1-3`) otherwise split the dataset with `split_dataset` method which in addition to the two datasets returns also the splitting hyperplane (line `4`). Then builds the left and right BSP-Tree by calling itself on the "left" and "right" dataset (line `5-6`) to then return the node initialized with the splitting hyperplane and the subtrees constructed before (line `7`).

2.1.1 Dataset split

One way to split dataset is for example to simply pick two random sample and split the dataset using the perpendicular bisector of the two samples. Another method is to use k-Means with $k = 2$ which will split the dataset in two and the splitting hyperplane would be the perpendicular bisector of the two cluster centers. The problem with both strategies is that they don't guarantee a balanced tree and partitions with at least k samples.

One method that ensures both requirements are satisfied is to first split the dataset according to the perpendicular bisector of two random points p_1 and p_2 and then move the splitting hyperplane along the line joining the two points in the direction of the point having the most closer samples until one half space contains at most one sample more than the other. The method `split_dataset` illustrated by Algorithm 5 does exactly this. Given the dataset S it starts by sampling two random points p_1 and p_2 and then split dataset according to their perpendicular bisector thus creating the datasets S_0 and S_1 (line `1-4`). Suppose now *max* and *min* are the indices of dataset with most samples and the one with fewer samples respectively. Then the algorithm calculate the number of samples, denoted by `to_move`, that need to be moved from S_{max} to S_{min} so that the difference between their sizes is almost 1 (line `5-6`). Afterwards moves `to_move` samples from S_{max} to S_{min} and each time translate the splitting hyperplane along line joining the points p_1 and p_2 in the direction of p_{max} by the amount $d_{s'}$ which is the distance between the samples being moved and the current hyperplane (line `7-15`). Finally moves the the splitting hyperplane again in the same direction as before by $\frac{d_{s'}}{2}$ where $d_{s'}$ is the distance between the sample in S_{max} closest to the splitting hyperplane and then return the the two dataset and the hyperplane (line `16-19`). The last translation of the hyperplane is done so that so it split the samples as evenly as possible.

2.2 Finding partitions by point or hypersphere

Finding the partition to which a point belongs or the ones that intersect a hypersphere can be done by traversing the BSP-Tree from top to bottom until leaf nodes are not reached like in any binary search tree. Algorithm 6 shows how this search is executed. Given a BSP-Tree and the input query which can be a point or a hypersphere, it first initialize a queue, which contains the nodes that need to be traversed, with the root of the tree (line `1`). Then until the queue

Algorithm 5: `split_dataset` method

```
Input $S$: samples dataset
Output $S_0$: the left side dataset    $S_1$: the right side dataset
       $\pi$: the splitting hyperplane
1  $\mathtt{p}_0, \mathtt{p}_1 \leftarrow$ **random_points**$(2)$
2  $\pi \leftarrow$ perpendicular bisector of $\mathtt{s}_1$ **and** $\mathtt{s}_2$
3  $S_0 \leftarrow \{s \mid s \in S,\ \pi(s) \geq 0\}$
4  $S_1 \leftarrow \{s \mid s \in S,\ \pi(s) < 0\}$
5  $\max, \min \leftarrow \underset{i \in \{0,1\}}{\operatorname{argmax}} |S_i|, \underset{i \in \{0,1\}}{\operatorname{argmin}} |S_i|$
6  $\mathtt{to\_move} \leftarrow \left\lfloor \frac{|S_0| - |S_1|}{2} \right\rceil$
7  $\mathtt{moved} \leftarrow 0$
8  **while** $\mathtt{moved} \neq \mathtt{to\_move}$ **do**
9      $s' \leftarrow$ sample **in** $S_{\max}$ closest **to** $\pi$
10     $d_{s'} \leftarrow$ distance between $s'$ **and** $\pi$
11     $S_{\min} \leftarrow S_{\min} \cup \{s'\}$
12     $S_{\max} \leftarrow S_{\max} \setminus \{s'\}$
13     $\pi \leftarrow$ **translate**$(\pi,\ d_{s'},\ \overrightarrow{p_{\min}p_{\max}})$
14     $\mathtt{moved} \leftarrow \mathtt{moved} + 1$
15 **end**
16 $s' \leftarrow$ sample **in** $S_{\max}$ closest **to** $\pi$
17 $d_{s'} \leftarrow$ distance between $s'$ **and** $\pi$
18 $\pi \leftarrow$ **translate**$\left(\pi,\ \frac{d_{s'}}{2},\ \overrightarrow{p_{\min}p_{\max}}\right)$
19 **return** $(S_{\max}, S_{\min}, \pi)$
```

is not empty extracts a node from the queue and check whether is a leaf in which case collect the dataset associated with the leaf (line `5-6`) otherwise compute the next nodes to traverse by calling the `next_nodes` method and add this nodes to the queue (line `7-9`). If the input is a point then `next_nodes` returns the subtree associated with the half space in which the input resides by checking on which side of the splitting hyperplane associated with the node the input is located (if the point is exactly on the hyperplane then returns both subtree). On the other hand if the input is a hypersphere then first checks if the splitting hyperplane intersect the hypersphere in which case returns both subtree otherwise return the subtree associated with the half space where the hypersphere resides. At the end returns the collected partitions which in the case of a point is the partition in which the point is located while for a hypershpere are the partitions that intersect with it (line `12`).

Algorithm 6: `query_partition` method

```
Input BSP-T: the BSP-Tree, x: the input query(point or hypershpere)
Output Set of partitions
queue ← create_queue({BSP-T.root})
partitions ← ∅
while queue is not empty do
    current_node ← queue.pop()
    if current_node is Leaf then
        partitions ← partitions ∪ {current_node.dataset()}
    else
        new_nodes ← next_nodes(current_node, x)
        queue.append(new_nodes)
    end
end
return partitions
```

3 Abstract classifier

In the abstract case there is the same datasets $S = \{(\boldsymbol{s}_i, \boldsymbol{y}_i) \mid \boldsymbol{s}_i \in \mathbb{R}^n, \boldsymbol{y}_i \in \mathbb{R}\}$ but the input query is not a single point $\boldsymbol{x} \in \mathbb{R}^n$ but instead is region of space around the point x. This region of space, denoted with $P^\varepsilon(\boldsymbol{x})$, represents a (small) perturbation of the point $\boldsymbol{x}$ and is defined as the ℓ_∞ ball centered in $\boldsymbol{x}$ and radius ε:

$$P^\varepsilon(\boldsymbol{x}) = \{\boldsymbol{x}' \mid \boldsymbol{x}' \in \mathbb{R}^n \wedge \|\boldsymbol{x}' - \boldsymbol{x}\|_\infty \leq \varepsilon\} \tag{1}$$

In this case the abstract classifier $\text{XXYYZZ}^A : \mathcal{P}(\mathbb{R}^n) \to \mathcal{P}(\mathcal{L})$ is a function that takes as input a region of space (i.e, $P^\varepsilon(x)$) and outputs a set of labels. The condition that the abstract classifier must satisfy is that it has to be a *sound abstraction* of the concrete classifier over the perturbation of the input $\boldsymbol{x}$ which means the returned set of labels must contains all the output of the concrete classifier on each point of region that is

$$\text{XXYYZZ}^A(R) \supseteq \bigcup_{\boldsymbol{x}' \in R} \text{XXYYZZ}(x') \tag{2}$$

Computing $\text{XXYYZZ}^A(P^\varepsilon(\boldsymbol{x}))$ by applying the concrete classifier on each point of the perturbation is obviously unfeasible since it contains an infinite number of points. But notice that the output of the concrete classifier depends manly on the valid paths within the precedence graph it builds. So following this observation $\text{XXYYZZ}^A(R)$ can be computed as follow:

1. Create an (abstract) precedence graph $\mathbb{G}^A_{\boldsymbol{x}}$ such that it contains all the valid paths of any concrete precedence graph $\mathbb{G}_{\boldsymbol{x}'}$ where $\boldsymbol{x}' \in P^\varepsilon(\boldsymbol{x})$;
2. Collect every path in $\mathbb{G}^A_{\boldsymbol{x}}$ which is a valid path in some concrete precedence graph $\mathbb{G}_{\boldsymbol{x}'}$
3. Classify the perturbation with most frequent labels in each path collected in the previous step.

With the above procedure the output of XXYYZZ^A can be computed since the number of valid paths to explore is finite.

3.1 Abstract precedence graph

To understand how to construct the abstract precedence graph $\mathbb{G}^A_{\boldsymbol{x}}$ suppose for example there are two samples $\boldsymbol{s}_1, \boldsymbol{s}_2 \in S$ and points $x_1, x_2 \in P^\varepsilon(\boldsymbol{x})$ such that

$$\begin{cases} \boldsymbol{s}_1 \preccurlyeq_{\boldsymbol{x}_1} \boldsymbol{s}_2 \text{ and } \boldsymbol{s}_2 \not\preccurlyeq_{\boldsymbol{x}_1} \boldsymbol{s}_1 \\ \boldsymbol{s}_2 \preccurlyeq_{\boldsymbol{x}_2} \boldsymbol{s}_1 \text{ and } \boldsymbol{s}_1 \not\preccurlyeq_{\boldsymbol{x}_2} \boldsymbol{s}_2 \end{cases}$$

that is $\boldsymbol{s}_1$ is strictly closer to $\boldsymbol{x}_1$ than $\boldsymbol{s}_2$ while $\boldsymbol{s}_2$ is strictly closer to $\boldsymbol{x}_2$ than $\boldsymbol{s}_1$. In this case paths in which $\boldsymbol{s}_1$ is a predecessor of $\boldsymbol{s}_2$ and those in which $\boldsymbol{s}_2$ is a predecessor of $\boldsymbol{s}_1$ are valid paths in $\mathbb{G}_{\boldsymbol{x}_1}$ and $\mathbb{G}_{\boldsymbol{x}_2}$ respectively and so they need to be both valid paths in the abstract precedence graph $\mathbb{G}^A_{\boldsymbol{x}}$ as well. This leads to the definition of the following relation between samples:

Definition 3.1.1 ($\preccurlyeq^A_{(\boldsymbol{x},\varepsilon)}$ relation): Given $\boldsymbol{x} \in \mathbb{R}^n, \varepsilon \in \mathbb{N}$ and $\boldsymbol{s}_1, \boldsymbol{s}_2 \in S$

$$\boldsymbol{s}_1 \preccurlyeq^A_{(\boldsymbol{x},\varepsilon)} \boldsymbol{s}_2 \iff \exists \boldsymbol{x}_i \in P^\varepsilon(\boldsymbol{x})\ \boldsymbol{s}_1 \preccurlyeq_{\boldsymbol{x}_i} \boldsymbol{s}_2$$

In the following, for ease of the notation, the subscript ε will be dropped since it is constant.

By this definition $\boldsymbol{s}_1 \preccurlyeq^A_x \boldsymbol{s}_2$ and $\boldsymbol{s}_2 \preccurlyeq^A_x \boldsymbol{s}_1$ and so there should be a edge between $\mathbb{G}^A_{\boldsymbol{x}}[\boldsymbol{s}_1]$ and $\mathbb{G}^A_{\boldsymbol{x}}[\boldsymbol{s}_2]$ in both direction. If instead $\boldsymbol{s}_1 \preccurlyeq^A_x \boldsymbol{s}_2$ but $\boldsymbol{s}_2 \not\preccurlyeq^A_x \boldsymbol{s}_1$ then there is only an edge from $\mathbb{G}^A_{\boldsymbol{x}}[\boldsymbol{s}_1]$ to $\mathbb{G}^A_{\boldsymbol{x}}[\boldsymbol{s}_2]$ but not the other way around.

To determine if the relation $\preccurlyeq^A_{\boldsymbol{x}}$ exists between two samples $\boldsymbol{s}_1, \boldsymbol{s}_2$, one would need to verify for each point $\boldsymbol{x}_1 \in P^\varepsilon(\boldsymbol{x})$ whether $\boldsymbol{s}_1 \preccurlyeq_{\boldsymbol{x}_1} \boldsymbol{s}_2$ holds. However, this approach is impractical due to the infinite number of points in $P^\varepsilon(\boldsymbol{x})$. A more feasible method is to check whether the perpendicular bisector of $\boldsymbol{s}_1$ and $\boldsymbol{s}_2$ intersect with $P^\varepsilon(\boldsymbol{x})$. The following result shows a sufficient condition that can be used to check this intersection efficiently:

Definition 3.1.2 (*pos_neg* function): Given $x \in \mathbb{R}$ let $\textit{pos\_neg} : \mathbb{R}^n \to \{-1, 1\}$ defined as

$$\textit{pos\_neg}(x) = \mathbb{H}(x) - 1 \cdot (1 - \mathbb{H}(x)) = \begin{cases} 1 \text{ if } x \geq 0 \\ -1 \text{ otherwise} \end{cases}$$

where $\mathbb{H}$ is the heaveside step function with the convention that $\mathbb{H}(0) = 1$.

Definition 3.1.3 (Hyperplane): Given $\boldsymbol{n}, \boldsymbol{v} \in \mathbb{R}^n, b \in \mathbb{R}$ let $\pi \overset{\text{def}}{:=} \boldsymbol{n} \cdot \boldsymbol{v} - b = 0$ be an equation of an hyperplane then given $\boldsymbol{w} \in \mathbb{R}^n$:

- $\pi(\boldsymbol{w})$ denotes the value $\boldsymbol{n} \cdot \boldsymbol{w} - b$
- $\textit{poly}(\pi)$ is the polynomial expression of the hyperplane equation that is $\textit{poly}(\pi) \overset{\text{def}}{:=} \sum_{i=1}^{i=n} n_i v_i - b$

Prop 3.1: Given $\boldsymbol{n}, \boldsymbol{v} \in \mathbb{R}^n, b \in \mathbb{R}$ let $\pi_{\boldsymbol{n},b} \overset{\text{def}}{:=} \boldsymbol{n} \cdot \boldsymbol{v} - b = 0$ be an hyperplane and $P^\varepsilon(\boldsymbol{x})$ a perturbation of a point $\boldsymbol{x} \in \mathbb{R}^n$ defined as Eq. 1 then

$$\pi_{\boldsymbol{n},b} \text{ interesect } P^\varepsilon(x) \iff \boldsymbol{x} \in \pi \text{ or } \textit{sign}\big(\pi_{\boldsymbol{n},b}(\boldsymbol{x})\big) \neq \textit{sign}\big(\pi_{\boldsymbol{n},b}(\boldsymbol{x}')\big)$$

where $\boldsymbol{x}' = \boldsymbol{x} - \varepsilon \cdot \textit{sign}\big(\pi_{\boldsymbol{n},b}(\boldsymbol{x})\big) \cdot \textit{pos\_neg}^\star(\boldsymbol{n})$ and $\textit{pos\_neg}^\star$ is the component-wise *pos_neg* operation over vectors in $\mathbb{R}^n$. Essentially the hyperplane π interesect the perturbation of $\boldsymbol{x}$ if and only if $\boldsymbol{x}$ and the point $\boldsymbol{x}'$, which is the vertex of the hypercube $P^\varepsilon(x)$ in the direction of π from $\boldsymbol{x}$, are on the oppisite side of π.

Proof: The proposition can be proved by demonstrating each directions of the implication separetly:

- ($\Longrightarrow$): Suppose $\pi_{\boldsymbol{n},b}$ interesect $P^\varepsilon(x)$ and $\boldsymbol{x} \notin \pi$. Since $\pi_{\boldsymbol{n},b}$ is the perpendicular bisector between s_1 and s_2 it means there exists a point $\boldsymbol{x}'' \in P^\varepsilon(\boldsymbol{x})$ such that it is equidistant to both samples (i.e. $\boldsymbol{x}'' \in \pi_{\boldsymbol{n},b}$). Because $\boldsymbol{x}'' \in P^\varepsilon(\boldsymbol{x})$ it can be defined as

$$\boldsymbol{x}'' = \boldsymbol{x} - \textit{sign}\big(\pi_{\boldsymbol{n},b}(\boldsymbol{x})\big) \cdot \boldsymbol{\varepsilon}'' \odot \textit{pos\_neg}^\star(\boldsymbol{n})$$

where $\odot$ denotes the *Hadamard* product and $\boldsymbol{\varepsilon}'' \in \mathbb{R}^n$ is a positive vector such that $\|\boldsymbol{\varepsilon}''\|_\infty \leq \varepsilon$. $\boldsymbol{x}'$ can also be defined in terms of $\boldsymbol{x}''$ as

$$\begin{aligned} \boldsymbol{x}' &= \boldsymbol{x}'' - \textit{sign}\big(\pi_{\boldsymbol{n},b}(\boldsymbol{x})\big) \cdot \boldsymbol{\varepsilon}' \odot \textit{pos\_neg}^\star(\boldsymbol{n}) \\ &= \boldsymbol{x} - \textit{sign}\big(\pi_{\boldsymbol{n},b}(\boldsymbol{x})\big) \cdot (\boldsymbol{\varepsilon}'' + \boldsymbol{\varepsilon}') \odot \textit{pos\_neg}^\star(\boldsymbol{n}) \end{aligned}$$

for some $\boldsymbol{\varepsilon}' \in \mathbb{R}^n$ positive vector such that $|\varepsilon''_i + \varepsilon'_i| = \varepsilon$. With this setup $\textit{sign}\big(\pi_{\boldsymbol{n},b}(\boldsymbol{x}')\big)$ is as follow:

$$\begin{aligned} \textit{sign}\big(\pi_{\boldsymbol{n},b}(\boldsymbol{x}')\big) &= \textit{sign}\left(\boldsymbol{n} \cdot \left[\boldsymbol{x} - \overbrace{\textit{sign}\big(\pi_{\boldsymbol{n},b}(\boldsymbol{x})\big)}^{s} \cdot (\boldsymbol{\varepsilon}'' + \boldsymbol{\varepsilon}') \odot \overbrace{\textit{pos\_neg}^\star(\boldsymbol{n})}^{\mathbf{dir}}\right] + b\right) \\ &= \textit{sign}(\boldsymbol{n} \cdot [\boldsymbol{x} - s \cdot (\boldsymbol{\varepsilon}'' \odot \mathbf{dir} + \boldsymbol{\varepsilon}' \odot \mathbf{dir})] + b) \\ &= \textit{sign}(\boldsymbol{n} \cdot [(\boldsymbol{x} - s \cdot \boldsymbol{\varepsilon}'' \odot \mathbf{dir}) - s \cdot \boldsymbol{\varepsilon}' \odot \mathbf{dir}] + b) \end{aligned}$$

$$\begin{aligned}
&= sign(\boldsymbol{n} \cdot (\boldsymbol{x} - s \cdot \boldsymbol{\varepsilon}'' \odot \mathbf{dir}) + b - s \cdot \boldsymbol{n} \cdot \boldsymbol{\varepsilon}' \odot \mathbf{dir}) \\
&= sign(\boldsymbol{n} \cdot \boldsymbol{x}'' + b - s \cdot \boldsymbol{n} \cdot \boldsymbol{\varepsilon}' \odot \mathbf{dir}) \\
&= sign\left(\overbrace{\boldsymbol{n} \cdot \boldsymbol{x}'' + b}^{=0} - s \cdot \boldsymbol{n} \cdot \boldsymbol{\varepsilon}' \odot \mathbf{dir}\right) \\
&= sign\left(-\,sign(\pi_{\boldsymbol{n},b}(\boldsymbol{x})) \cdot \overbrace{\boldsymbol{n} \cdot \boldsymbol{\varepsilon}' \odot pos\_neg^\star(\boldsymbol{n})}^{\text{positive}}\right) \qquad (3) \\
&= -\,sign(\pi_{\boldsymbol{n},b}(\boldsymbol{x})) \neq sign(\pi_{\boldsymbol{n},b}(\boldsymbol{x}))
\end{aligned}$$

In Eq. 3 $\forall i \in \{0, ..., n-1\}$ if $n_i \neq 0$ then the sign of $\varepsilon'_i \cdot pos\_neg(n_i)$ is the same as n_i and so the sign of $n_i \cdot \varepsilon'_i \cdot pos\_neg\ (n_i)$ is always postive hence $\boldsymbol{n} \cdot \boldsymbol{\varepsilon}' \odot pos\_neg^\star(\boldsymbol{n})$ is a positive value.

In the case $\boldsymbol{x} \in \pi_{\boldsymbol{n},b}$ then the implication holds vacuously.

- $(\Longleftarrow)$: if $\boldsymbol{x} \in \pi_{\boldsymbol{n},b}$ or $sign(\pi_{\boldsymbol{n},b}(\boldsymbol{x})) \neq sign(\pi_{\boldsymbol{n},b}(\boldsymbol{x}'))$ then it means $\pi_{\boldsymbol{n},b}$ surely interesect $P^\varepsilon(x)$.

$\square$

The procedure to create the abstract precedence graph is the same as Algorithm 1 with the only difference being the order relation used which is $\preccurlyeq^A_x$.

Example 3.1: Consider the setup of Example 1.1 and perturbation $P^\varepsilon(x)$ where $\varepsilon = 0.25$ shown in the plot in Figure 2 a). In this case:

- $a \preccurlyeq^A_x b \wedge b \preccurlyeq^A_x a$
- $a \preccurlyeq^A_x c \wedge c \not\preccurlyeq^A_x a$
- $b \preccurlyeq^A_x c \wedge b \preccurlyeq^A_x c$

and so the abstract precedence is the one shown in Figure 2 b). The abstract precedence graph contains an additional valid path w.r.t the concrete precedence graph which is:

$$[\boldsymbol{b}, \boldsymbol{c}, \boldsymbol{a}]$$

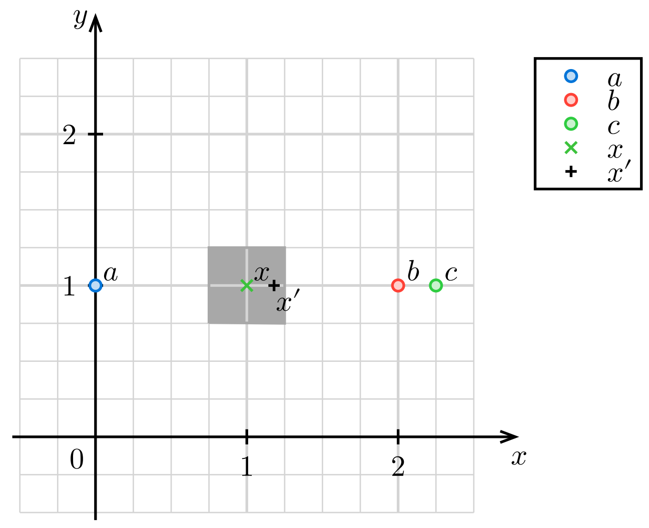
because there are points (e.g., the point x') in the perturbation such that $c \preccurlyeq a \wedge a \not\preccurlyeq c$ and so in the abstract case there are 3 valid paths.

Prop 3.2: Given a sample $\boldsymbol{s} \in S$ and an abstract Precedence graph $\mathbb{G}^A_{\boldsymbol{x}}$

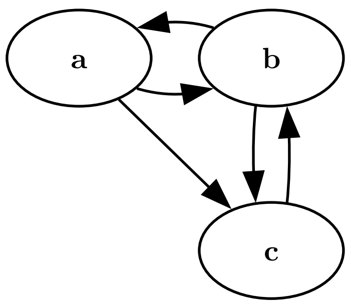
$$\mathtt{pred}^A(\boldsymbol{s}) = \bigcap_{\boldsymbol{x}' \in P^\varepsilon(\boldsymbol{x})} \mathtt{pred}_{\boldsymbol{x}'}(\boldsymbol{s})$$

where $\mathtt{pred}^A(\boldsymbol{s})$ and $\mathtt{pred}_{\boldsymbol{x}'}(\boldsymbol{s})$ are the value of $\mathtt{pred}(\boldsymbol{s})$ in the graph $\mathbb{G}^A_{\boldsymbol{x}}$ ahd $\mathbb{G}_{\boldsymbol{x}'}$ respectively.

Proof: Follows from the definition of$\preccurlyeq^A_x$ $\square$



a) Plot of dataset and perturbation of Example 1.1



b) Abstract precedence graph

3.2 Path generation

Another difference between the concrete and abstract case is the definition of a valid path. To see why consider for example the set of samples:

- $\boldsymbol{s}_0$: (0.75, 1.3)
- $\boldsymbol{s}_1$: (1.0, 1.3)
- $\boldsymbol{s}_2$: (1.25, 1.3)

and the input perturbation $P^{\varepsilon}((1,1))$ with $\varepsilon = 0.05$. Figure 3 a) shows the plot of the samples and the perturbation region while Figure 3 b) illustrate the abstract graph. According to Definition 1.2.1 the valid paths in the abstract precedence graph are the permutations of the sequence of vertices $[\boldsymbol{s_0}, \boldsymbol{s_1}, \boldsymbol{s_2}]$ in particular:

1. $[\boldsymbol{s_0}, \boldsymbol{s_1}, \boldsymbol{s_2}]$
2. $[\boldsymbol{s_0}, \boldsymbol{s_2}, \boldsymbol{s_1}]$

but notice that there is no $\boldsymbol{x}' \in P^{\varepsilon}(\boldsymbol{x})$ such that second path is a valid order of precedence in $\mathbb{G}_{\boldsymbol{x}'}$. This is because the regions of space containing the points closer the sample $\boldsymbol{s}_0$ (i.e. the blue region) and the one with points closer to sample $\boldsymbol{s}_2$ than $\boldsymbol{s}_1$ (i. e. the green region) do not intersect.

3.3 Valid Path

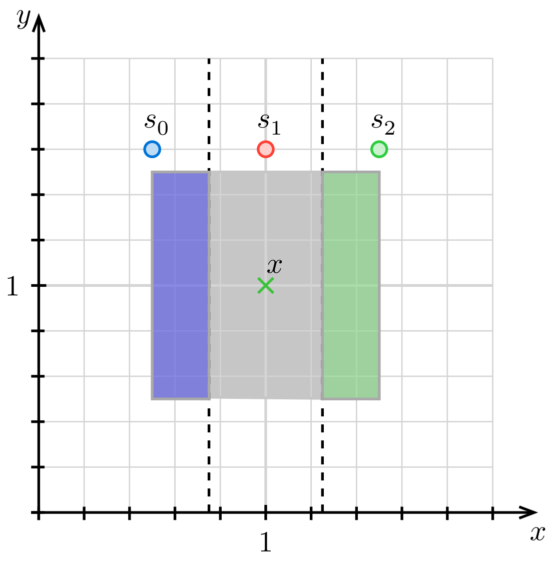
The previous example illustrate that when adding a vertex v to a path p checking only the predecessors of the vertex do not suffice and the existence of a region of the perturbation containing points such that the samples satisfy the precedence order in p must also be taken into consideration. This leads to the following definition for a valid path in the abstract case:

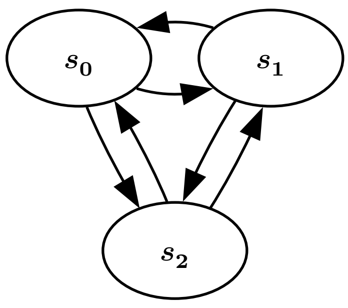
Definition 3.3.1 (Valid path): Given an abstract precedence graph $\mathbb{G}^{A}_{\boldsymbol{x}'}$ let $\mathcal{P} = [\boldsymbol{v_0}, \boldsymbol{v_1}, ..., \boldsymbol{v_m}]$ be a path in $\mathbb{G}^{A}_{x}$. The path $\mathcal{P}$ is *valid* if and only if

- $\forall \boldsymbol{s}_i \in \mathcal{P}.\ \forall \boldsymbol{x}_j \in \texttt{pred}(\boldsymbol{x}_i).\ \boldsymbol{x}_j$ is a predecessor of $\boldsymbol{s}_i$ in $\mathcal{P}$
- $\exists R \subseteq P^{\varepsilon}(\boldsymbol{x})$ s.t. $R \neq \emptyset \wedge \forall \boldsymbol{x}' \in R.\ \mathcal{P}$ is a valid path in $\mathbb{G}_{\boldsymbol{x}}$

The definition of a safe vertex reamins unchanged.

To see how the sub region R can be calculated consider the samples of the previous example. The path $[\boldsymbol{s_1}]$ is valid if there is a region of points $R_1 \in P^{\varepsilon}(\boldsymbol{x})$ such that the sample $\boldsymbol{s_1}$ is





a) Plot of dataset and perturbation b) Abstract precedence graph

Figure 3: Example of invalid path

the closest to them. This means that points in R_1 must satisfy the following linear system of inequalities:

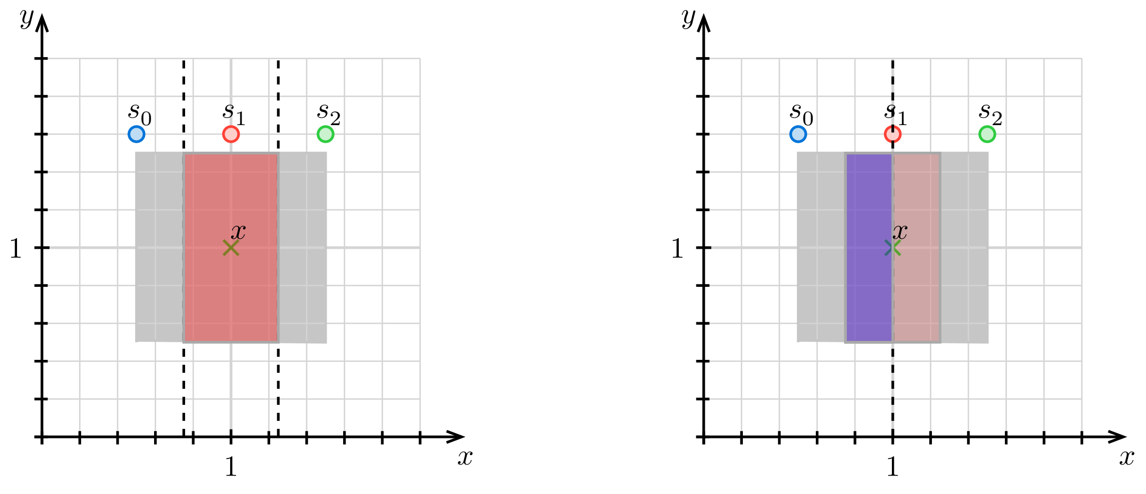
$$\begin{cases} x_1 - 1.125 \leq 0 \\ -x_1 + 0.75 \leq 0 \\ 0.75 \leq x_1 \leq 1.75 \\ 0.75 \leq x_2 \leq 1.75 \end{cases}$$

where $x_1 - 1.25 = 0$ and $x_1 - 0.75 = 0$ are the perpendicular bisector between the samples $\boldsymbol{s_1}$ and $\boldsymbol{s_0}$ and samples $\boldsymbol{s_1}$ and $\boldsymbol{s_2}$ respectively. This region is shown in Figure 4 a). Consider now the path $[\boldsymbol{s_1}, \boldsymbol{s_0}]$. This path is valid if it exists a region of points $R_2 \in P^{\varepsilon}(\boldsymbol{x})$ such that the closest samples is $\boldsymbol{s_1}$ and the second closest is $\boldsymbol{s_0}$. So points in R_2 must satisfy the following linear system of inequalities:

$$\begin{cases} x_1 - 1.25 \leq 0 \\ -x_1 + 0.75 \leq 0 \\ x_1 - 1 \leq 0 \\ 0.75 \leq x_1 \leq 1.75 \\ 0.75 \leq x_2 \leq 1.75 \end{cases} \tag{4}$$

where the first two inequalities define the R_1 region while the third inequality exclude from R_1 those points that are strictly closer to the sample $\boldsymbol{s_2}$ than $\boldsymbol{s_0}$ resulting in the blue region illustrated in Figure 4 b). Finally the same logic applies to path $[\boldsymbol{s_1}, \boldsymbol{s_0}, \boldsymbol{s_2}]$ which defines the same the linear system as Eq. 4.

On the other hand the linear system of inequalities associated with path $[\boldsymbol{s_0}, \boldsymbol{s_2}]$ is



a) Region of points satisfying the path $[\boldsymbol{v_1}]$

b) Region of points satisfying the path $[\boldsymbol{v_1}, \boldsymbol{v_2}]$

Figure 4: Example of region of points denoted by a path

$$\begin{cases} x_1 - 1 \leq 0 \\ x_1 - 0.875 \leq 0 \\ -x_1 + 1.25 \leq 0 \\ 0.75 \leq x_1 \leq 1.75 \\ 0.75 \leq x_2 \leq 1.75 \end{cases}$$

and is shown in Figure 3 a). As can be seen from the plot the two regions (i.e. the blue one containing points for which the sample $\boldsymbol{s_0}$ is the closest one and the green containing points for which the sample $\boldsymbol{s_2}$ is closer than $\boldsymbol{s_1}$) do not intersect hence the linear system has no solutions. So the path $[\boldsymbol{s_0}, \boldsymbol{s_2}]$ and consequently $[\boldsymbol{s_0}, \boldsymbol{s_2}, \boldsymbol{s_1}]$ are not valid paths.

To summarize, given a path $\mathcal{P}$, the idea is to identify a polytope inside the perturbation region using a linear system of inequalities which contiains only those points for which the sequence of samples defined by $\mathcal{P}$ is ordered according to distance to this points. Then if this polytope exists (i.e the linear system has solutions) it means that there is at least one concrete precedence graphs $\mathbb{G}_{\boldsymbol{x}'}$ for some $\boldsymbol{x}' \in P^{\varepsilon}(x)$ such that $\mathcal{P}$ is valid in $\mathbb{G}_{\boldsymbol{x}'}$. Conversely, if the linear system has no solutions, it means that no point in the perturbation region can produce the sequence of samples as defined by the path.

Algorithm 7 shows hoe to construct a polytope given a path and the bounds of the perturbation as a set, where each element is of the form $min_1 \leq x_i \leq max_i$. It builds the linear system of inequalities using the `bisector` function, which, given two samples $\boldsymbol{s}_i$ and $\boldsymbol{s}_j$, returns the perpendicular bisector β between them such that $\beta(\boldsymbol{s}_j) \geq 0$. The algorithm begins by creating the linear system of inequalities from the bounds provided in the input (line 1). Then it adds inequalities to ensure that the sample $\boldsymbol{s}_0$ (i.e. the path first sample) is the closest to the perturbation (lines 2-5). This is achieved by constructing the perpendicular bisector between $\boldsymbol{s}_0$ and the samples in `same_dist`$(\boldsymbol{s}_0)$ using the `bisector` function, and imposing that the perturbation region lies in the half-space containing $\boldsymbol{s}_0$ (lines $2-4$). Subsequently, for each next sample $\boldsymbol{s}_i$ in the path, inequalities are added to ensure that the sample $\boldsymbol{s}_i$ is the closest to the perturbation region after the sample $\boldsymbol{s}_j$ $0 \leq j < i$ (lines $7-17$). This is done by first constraining the sample $\boldsymbol{s}_i$ to be closer to the perturbation than each sample in `same_dist`$(\boldsymbol{s}_i)$ (lines 8-11), and then ensuring that samples $\boldsymbol{s}_j$ for $0 \leq j < i$ are closer than sample $\boldsymbol{s}_i$ (lines 12-17). Finally, the created linear system is returned (line 18).

Algorithm 7: `build_polyhedron` method

Input `path`: A sequence of vertices, `bounds`: Bounds of the perturbation given as a set whose elements are of the form $min_1 \leq x_i \leq max_i$

Output A linear system of inequalities

```
1  LSI ← bounds
2  for vertex in same_dist(path[1]) do
3      $\beta$ ← bisector(path[1], vertex)
4      LSI ← LSI $\cup$ {$poly(\beta) \leq 0$}
5  end
6  n ← length(path)
7  for i from 2 to n do
8      for vertex in same_dist(path[$i$]) do
9          $\beta$ ← bisector(path[$i$], vertex)
10         LSI ← LSI $\cup$ {$poly(\beta) \leq 0$}
11     end
12     for j from 1 to i − 1 do
13         if path[$j$] $\notin$ same_dist(path[$i$]) $\wedge$ path[$j$] $\notin$ pred(path[$i$]) then
14             $\beta$ ← bisector(path[$j$], path[$i$])
15             LSI ← LSI $\cup$ {$poly(\beta) \leq 0$}
16         end
17     end
18 end
19 return LSI
```

3.4 Valid path generation

Algorithm 8 shows how the valid paths on length n are generated given the abstract precedence graph $\mathbb{G}^A_{\boldsymbol{x}}$. Essentially the algorithm is same as the Algorithm 2 with the only difference being that before starting to construct the paths of the desired length it checks whether the initial paths (i.e. those containing a single vertex) are valid or not (line 2-7).

Prop 3.3: Given as input the the abstract precedence graph $\mathbb{G}^A_{\boldsymbol{x}}$ and the length $n \in \mathbb{N}$ Algorithm 8 returns all the valid paths of length n within $\mathbb{G}^A_{\boldsymbol{x}}$.

Proof: The proposition can be proved by showing, using induction on k, that at the k-th iteration of the while loop, at line 11 the queue contians all the valid paths of length $k - 1$:

- ($k = 1$):In the first iteration of the loop the queue contains all the path made of a single sample $\boldsymbol{s}_i$ such that:
    - `pred`$(\boldsymbol{s})_i = \emptyset$ since $\boldsymbol{s}_i \in$ `closest_vertices` by defintion (line 1);
    - $\exists R \in P^\varepsilon(x)$ s.t. $\forall \boldsymbol{x}' \in R$. sample $\boldsymbol{s_i}$ is the closest sample by defintoin (lines 3-7)

  This means that the path $[\boldsymbol{s}_i]$ is a valid path by definition

Algorithm 8: `abstract_generate_paths` method

```
   Input $G_x$: precedence graph, n: length of the path
   Output Set of valid paths of length n
1  closest_vertices ← $\{v_i \mid v_i \in \mathbb{G}_x,\ \texttt{pred}(v_i) = \emptyset\}$
2  init_paths ← {}
3  for vertex in closest_vertices do
4      if vertex is safe for empty_path then
5          init_paths ← init_paths ∪ {[vertex]}
6      end
7  end
8  queue ← create_queue(init_paths)
9  k ← 0
10 while queue not empty do
11     current_paths ← queue.popAll()
12     if k = n then
13         return current_paths
14     for current_path in current_paths do
15         last_vertex ← current_path.last()
16         for adj in last_vertex.adjacent do
17             if adj is safe for current_path then
18                 queue.append(current_path + [adj])
19             end
20         end
21     end
22     k ← k + 1
23 end
```

- ($k = h + 1$): At iteration $h + 1$, the paths inside the queue at line 11 are obtained by extending all the paths in the queue in the h-th iteration with every safe vertex among the adjacent of the path's last vertex. By induction hypothesis the queue in the h-th iteration contains all the valid paths of length h and since they are extended with every safe vertices it means that in the $h + 1$ iteration the queue contains only and all the valid paths of length $h = k - 1$.

□

3.5 XXYYZZ^A Algorithm

Algorithm 9 shows how the abstract classifier works. Given the samples dataset $\boldsymbol{S}$, the input sample $\boldsymbol{x}$ and the number k of neighbours to consider fot the classification, tt stars by constructing the abstract precedence graph $\mathbb{G}_{\boldsymbol{x}}^A$ as described in Section 3.1 (line (1)). Afterwards extracts all the valid paths within $\mathbb{G}_{\boldsymbol{x}}^A$ (line (2)) to then collect the most frequent labels in each path (line 3-7). Finally the collected labels are return as the classification of the input (line 8).

Algorithm 9: XXYYZZ^A algoritm

```
Input  x: the input sample        S: samples dataset
       k: number of neighbours
Output Set of possible classification of $x$
1 $\mathbb{G}^A_{\boldsymbol{x}}$ ← create_abstract_precedence_graph(S, x)
2 paths ← abstract_generate_paths($\mathbb{G}^A_{\boldsymbol{x}}$, $k-1$)
3 classification ← {}
4 for path in paths do
5     labels ← most frequent labels in path
6     classification ← labels ∪ classification
7 end
8 return classification
```

3.5.1 Completeness

As mentioned in the beginning of this chapter, given the perturbation $P^\varepsilon(\boldsymbol{x})$, the abstract classifier is said to be *sound* if the computed set of labels contains all the output of the concrete classifier on each point of the perturbation that is when

$$\bigcup_{\boldsymbol{x}' \in P^\varepsilon(\boldsymbol{x})} \text{XXYYZZ}(\boldsymbol{x}', S, k) \subseteq \text{XXYYZZ}^A(P^\varepsilon(\boldsymbol{x}), S, k) \tag{5}$$

Actually the abstract classifier satisfy the stronger condition of being *exact* (or *complete*) where the equality sign in Eq. 5 holds. In fact, as the following result shows, the abstract classifier output contains all and only the outputs of the concrete classifier for each point within the perturbation.

Theorem 3.1: Given a dataset $S = \{(\boldsymbol{s}_i, \boldsymbol{y}_i) \mid \boldsymbol{s}_i \in \mathbb{R}^n, \boldsymbol{y}_i \in \mathcal{L}\}$ and the perturbation $P^\varepsilon(\boldsymbol{x})$ of the input sample $\boldsymbol{x}$, then

$$\bigcup_{\boldsymbol{x}' \in P^\varepsilon(\boldsymbol{x})} \text{XXYYZZ}(\boldsymbol{x}', S, k) = \text{XXYYZZ}^A(P^\varepsilon(\boldsymbol{x}), S, k)$$

Proof: The main difference, other than the precedence relation, between the concrete and abstract classifier is the way they collect the valid paths. Since Prop 1.4 and Prop 3.3 ensures that both algorithms consider all and only the valid paths of length k for the classification the preposition can be proven by separately demonstrating that:

1. $\bigcup_{\boldsymbol{x}' \in P^\varepsilon(\boldsymbol{x})} \texttt{valid\_path}\left(\mathbb{G}_{\boldsymbol{x}_i}\right) \subseteq \texttt{valid\_path}\left(\mathbb{G}^A_{\boldsymbol{x}}\right)$
2. $\bigcup_{\boldsymbol{x}' \in P^\varepsilon(\boldsymbol{x})} \texttt{valid\_path}\left(\mathbb{G}_{\boldsymbol{x}_i}\right) \supseteq \texttt{valid\_path}\left(\mathbb{G}^A_{\boldsymbol{x}}\right)$

where `valid_path`(G) denotes the set of valid paths in a graph G.

1) Without loss of generality consider a valid path $\mathcal{P} = [\boldsymbol{s}_0, \boldsymbol{s}_1, ..., \boldsymbol{s}_{k-1}]$ of length k in $\mathbb{G}_{\boldsymbol{x}_i}$ for some $\boldsymbol{x}_i \in P^\varepsilon(\boldsymbol{x})$. By definition of path in graph and $\preccurlyeq_{\boldsymbol{x}_i}$ we have that

$$s_0 \preccurlyeq^A_{\boldsymbol{x}_i} s_1 \preccurlyeq^A_{\boldsymbol{x}_i} \cdots \preccurlyeq^A_{\boldsymbol{x}_i} s_{k-1}$$

This means that $\mathcal{P}$ is also a path in $\mathbb{G}^A_{\boldsymbol{x}}$. Moreover since $\mathcal{P}$ is a valid path and, by Prop 3.2, $\forall v_i \in \mathcal{P}$

$$\texttt{pred}\big(\mathbb{G}^A_{\boldsymbol{x}}[s_i]\big) \subseteq \texttt{pred}\Big(\mathbb{G}_{\boldsymbol{x}_i}[s_i]\Big)$$

it means that p satisfy both the condition of Definition 3.3.1 hence $p \in \texttt{valid\_path}\big(\mathbb{G}^A_{\boldsymbol{x}}\big)$.

2) Suppose p is a valid path in $\mathbb{G}^A_{\boldsymbol{x}}$. Then, by definition of valid path, it means there exists $\boldsymbol{x'} \in P^\varepsilon(\boldsymbol{x})$ such that the path represents a valid order of precedence according to the distance between the samples and $\boldsymbol{x'}$. This means that $p \in \texttt{valid\_path}\ (\mathbb{G}_{\boldsymbol{x'}})$

□

3.6 Abstract precedence graph construction optimization

Since in the abstract case the query is the perturbation of the input rather the a single point, the algorithm to reduce the set of sample used for the precedence graph construction needs to be changed. The only difference with the concrete counterpart is the step 4) because, in order to ensure the soundness of the abstract classifier, the hypershpere must contain all the k closest sample of each point in the perturbation.

In order to find the minimum radius r of the hypershpere consider the k closest sample $A = \{\boldsymbol{s}_1, \boldsymbol{s}_2, ..., \boldsymbol{s}_k\}$ to the input $\boldsymbol{x}$ and let s_k be the furthest sample from $\boldsymbol{x}$ with distance d. Now consider a generic point $\boldsymbol{x'} \in P^\varepsilon(\boldsymbol{x})$ and the hypershpere $H_{\boldsymbol{x'}}$ centered in $\boldsymbol{x'}$ and radius the distance between $\boldsymbol{x'}$ and $\boldsymbol{s}_{\boldsymbol{x'}} = \operatorname*{argmax}_{\boldsymbol{s}_i \in A} \|\boldsymbol{x'} - \boldsymbol{s}_i\|$. Surely the set of samples in $H_{\boldsymbol{x'}}$ is $A \cup B$ where B is the set (possibly empty) of samples closer to $\boldsymbol{x'}$ than any sample in A. So the enclosing hypershpere H of all the hyperspheres $H_{\boldsymbol{x'}}$ for every $\boldsymbol{x'} \in P^\varepsilon(\boldsymbol{x})$ is the one that surely contains all the k closest sample of each point in the perturbation.

Notice that given a points $\boldsymbol{y} \in \mathbb{R}^n$ and $\boldsymbol{x'} \in P^\varepsilon(\boldsymbol{x})$

$$\|\boldsymbol{x'} - \boldsymbol{y}\| \leq \|\boldsymbol{x} - \boldsymbol{x'}\| + \|\boldsymbol{x} - \boldsymbol{y}\|$$

due to the triangular inequality property of norms. So this means that for every $\boldsymbol{x'} \in P^\varepsilon(\boldsymbol{x})$

$$\|\boldsymbol{x'} - \boldsymbol{s}_{\boldsymbol{x'}}\| \leq \|\boldsymbol{x} - \boldsymbol{x'}\| + \|\boldsymbol{x} - \boldsymbol{s}_{\boldsymbol{x'}}\| \leq \sqrt{N} \cdot \varepsilon + d$$

Since the maximum radius of $H_{x'}$ for any $x' \in P^\varepsilon(x)$ is $\sqrt{N} \cdot \varepsilon + d$ it means the hypershpere of interest H is centered in the input x and has radius $2\varepsilon\sqrt{N} + d$.

4 Optimizations

One problem with the Algorithm 9 is that is quite inefficient. For example consider the case in which there are n samples equidistant from the perturbation $P^\varepsilon(\boldsymbol{x})$ then in this case the abstract precedence graph could contains at least

$$\frac{n!}{(n-k)!}$$

valid paths which for just $n = 10$ and $k = 7$ is already over 10^6 making the naive abstract classifier quite inefficient because it explicitly enumerate all the possible paths of length k before classifying the input. To overcame this problem there are several optimizations that can be used to acceletare the classification process:

Opt. 1) Suppose that in a valid path $\mathcal{P}$ of length $m < k$ the two most common labels ℓ_1 and ℓ_2 have t_1 and t_2 occurrences respectively then if $t_1 - t_2 \geq k - m$ it means that ℓ_1 will be among the dominant labels regardless of the last $k - m$ samples of the path. So there is no need to further extend the path and the label ℓ_1 can be added among the possible classifications of the input.

Opt. 2) One special case of **Opt. 1)** is when the first $\lceil \frac{k}{2} \rceil$ samples of a valid path $\mathcal{P}$ have the same label ℓ then again there is need to extend the path further and ℓ can be added among the possible classifications of the input.

Opt. 3) For the input to be classified with label ℓ is sufficient to find a single valid path within the abstract precendence graph having ℓ among the most common labels, This means once such path is found there is no need to search for other paths with ℓ as the most occurring label.

Following the optimisations listed above the abstract classifier become as shown in Algorithm 10. It sarts by constructing the abstract precedence graph and the collecting then distinct labels among the samples of the graph (line `1-2`). Then for each label ℓ construct the set `paths` of valid paths made only of samples with ℓ label (line `5`) and then check whether the **Opt. 2** optimisation can be applied to any such constructed path. If this is the case then add label ℓ among the possibile classification of the input and continue to the next label (line `6-9`). Otherwise following the observation of **Opt 3.** search for a path that satisfy the condition of **Opt 1.** (line `10-19`). To do so initialize with the set `paths` a priority queue where the priority is the tuple $(\text{n. occurrence of } \ell, \text{path length})$ (line `10`). Then until the queue is not empy extract the path with the highest priority and extend it with all the possibile safe samples among the adjacent of the path last sample such that the label ℓ remains one of the most occurring label and check whether the **Opt. 1** can be applied to any such generated paths and if so add label ℓ to the possibile classifications of the input and continue to the next label (line `11-17`). Otherwise add the generated paths to priority queue (line `18`). At the end return the possibile classifications of the input (line `21`).

Algorithm 10: Optimised XXYYZZ^A algoritm

```
Input  x: the input sample        S: samples dataset
       k: number of neighbours
Output Set of possible classification of $x$
1  $\mathbb{G}_{\boldsymbol{x}}^A \leftarrow$ create_abstract_precedence_graph(S, x)
2  possible_labels $\leftarrow$ distinct labels among the samples in $\mathbb{G}_{\boldsymbol{x}}^A$
3  classification $\leftarrow \{\}$
4  for label in possible_labels do
5      paths $\leftarrow$ generate_init_valid_paths($\mathbb{G}_{\boldsymbol{x}}^A$, label)
6      if Opt. 2 can be applied to any path in paths then
7          classification $\leftarrow$ {label} $\cup$ classification
8          continue
9      end
10     queue $\leftarrow$ init_priority_queue(paths)
11     while queue not empty do
12         path $\leftarrow$ queue.pop()
13         new_paths $\leftarrow$ extend_path(path)
14         if Opt. 1 can be applied to any path in new_paths then
15             classification $\leftarrow$ {label} $\cup$ classification
16             break
17         end
18         queue.add(new_paths)
19     end
20 end
21 return classification
```

5 Comparison with NAVe

Some comparisons with Nave tool using the euclidean norm and interval domain.

Dataset	ε	**Runtime (mm:ss)**			**Stability**		**Robustness**	
		NAVe	XXYYZZ^A		**NAVe**	XXYYZZ^A	**NAVe**	XXYYZZ^A
Fourclass	0.01	00:01	00:01	k=1	99.2	99.6	99.2	99.6
				k=2	98.4	98.4	98.4	98.4
				k=3	99.6	99.6	99.6	99.6
				k=5	98.4	98.8	98.4	98.8
				k=7	97.7	98.4	97.7	98.4
	0.05	00:01	00:04	k=1	45.0	71.3	45.0	71.3
				k=2	39.9	65.1	39.9	65.1
				k=3	42.6	71.7	42.6	71.7
				k=5	40.3	72.8	40.3	72.8
				k=7	37.6	74.4	37.6	74.4
Pendigits	0.01	11:39	00:37	k=1	96.7	97.8	95.6	96.5
				k=2	94.0	96.2	93.3	95.2
				k=3	96.1	98.1	95.3	96.8
				k=5	95.7	98.3	94.9	96.6
				k=7	95.0	98.1	94.1	96.2
	0.05	11:53	07:14	k=1	59.0	81.7	59.0	81.5
				k=2	52.6	79.7	52.6	79.5
				k=3	60.6	85.4	60.6	85.2
				k=5	59.8	86.2	59.7	85.9
				k=7	58.9	86.6	58.9	86.2
Letter	0.01	29:45	06:55	k=1	82.7	88.7	82.2	87.9
				k=2	70.4	79.9	70.4	79.7
				k=3	75.1	87.4	75.0	86.8
				k=5	69.5	86.6	69.4	85.8
				k=7	65.2	86.0	65.1	85.1
	0.02	30:30	10:47	k=1	54.6	73.0	54.6	72.9
				k=2	42.1	61.4	42.1	61.4
				k=3	45.4	70.6	45.4	70.6
				k=5	40.3	70.8	40.3	70.7
				k=7	37.0	69.6	37.0	69.5

Bibliography

[1] de Berg M.; van Kreveld M.; Overmars M.; Schwarzkopf O., *Computational Geometry.* Springer-Verlag, 2000.

[2] M. S. Charikar, “Similarity estimation techniques from rounding algorithms,” in *Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing*, in STOC '02. Montreal, Quebec, Canada: Association for Computing Machinery, 2002, pp. 380–388. doi: 10.1145/509907.509965.

[3] E. Bernhardsson, “Annoy.” GitHub, 2024.