Graph based kNN (XXYYZZ)

1 Concrete algorithm

Given a set of labels $\mathcal{L}$ and the dataset $S = \{(s^i, y^i) \mid s^i \in \mathbb{R}^n, y^i \in \mathcal{L}\}$ and the input sample $x \in \mathbb{R}^n$, 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 minkowski norm to select the k closest samples. The relation is defined as follow:

Definition 1.1 (Relation $\preccurlyeq$): Given the points $x, s_i, s_j \in \mathbb{R}^n$, "$s_i \preccurlyeq_x s_j$" means s_i is closer to the input than s_i w.r.t. the minkowski norm that is $\| x - s^i \|_p \leq \| x - s^j \|_p$. The subscript x will be omitted when is clear from the context.

XXYYZZ first build a "precedence" graph $\mathbb{G}$ in which nodes are 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

The precedence graph $\mathbb{G}_x = (V, E)$ is a graphical representation of the totally ordered set $(S, \preccurlyeq_x)$:

- $V = S$
- $E = \{(v_i, v_j) \mid \| x - s^i \|_p \leq \| x - s^j \|_p\}$

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 each vertex v_i also has a set attribute, named `predecessors`, that contains all the vertices v_j such that there is an incoming edge from v_j but not an outgoing edge that is

$$(v_j, v_i) \in E \text{ and } (v_i, v_j) \notin E$$

This means that

$$\forall v_j \in v_i.\texttt{predecessors} \;\; v_j \preccurlyeq v_i \;\; \text{and } v_i \not\preccurlyeq v_j$$

This information will be used for the generation of valid paths based on how close the samples are to the input sample. In the remainder of the document the vertex associated with sample s_i is denoted with $\mathbb{G}_x[s_i]$ while the sample associated with the vertex v_j is denoted simply with s_j.

Algorithm 1 shows how the graph is created. Given the sample dataset S and the input sample x, it first create the graph $\mathbb{G}$ with only the vertices with their `predecessors` and `adjacent` attributes set to the empty set (line `1`). Then for each unordered pair of samples (s_i, s_j) we check which sample between s_i and s_j is closer to the input sample x w.r t. the minkowski norm and update both vertex attributes accordingly (line $2-12$). If s_i and s_j are equidistant than only the adjacent lists are updated since $s_j \preccurlyeq s_i$ and $s_i \preccurlyeq s_j$ (line $3-5$). 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 $\mathbb{G}_x$: precedence graph

$\mathbb{G}_x \leftarrow$**initialize_graph**(S)
for (s_i, s_j) **in** $\{(s_i, s_j) |\ s_i, s_j \in S, s_i \neq s_j\}$ **do**
  if $s_i \preccurlyeq s_j$ **and** $s_j \preccurlyeq s_i$ **then**
    add $\mathbb{G}_x[s_j]$ **to** $\mathbb{G}_x[s_i]$.`adjacent`
    add $\mathbb{G}_x[s_i]$ **to** $\mathbb{G}_x[s_j]$.`adjacent`
  else if $s_i \preccurlyeq s_j$ **then**
    add $\mathbb{G}[s_j]$ **to** $\mathbb{G}_x[s_i]$.`adjacent`
    add $\mathbb{G}[s_i]$ **to** $\mathbb{G}_x[s_j]$.`predecessors`
  else
    add $\mathbb{G}_x[s_i]$ **to** $\mathbb{G}_x[s_j]$.`adjacent`
    add $\mathbb{G}_x[s_j]$ **to** $\mathbb{G}_x[s_i]$.`predecessors`
  end
end
return $\mathbb{G}_x$

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:

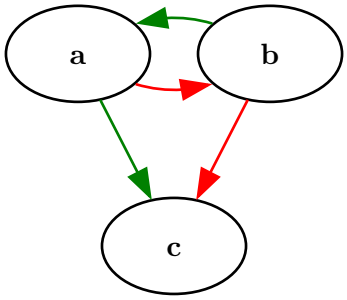


Figure 1: Example of precedence graph.

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 $v_i \preccurlyeq v_j$ but $v_j \not\preccurlyeq v_i$ then every path in which v_j occurs must also contain v_i and it must precede v_j. This means that not all paths starting from the input 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 $\forall v_i \in \mathcal{P}$:

$$\forall v_j \in v_i.\texttt{predecessors} \quad v_j \text{ is a predecessor of } v_i \text{ in } \mathcal{P}$$

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

- [***b***, ***a***,***c***]
- [***a***, ***b***,***c***]

Prop 1.1: Every valid path starts with a vertex v_i such that the sample s_i is one of the closest sample to the input x

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

Prop 1.2: If a vertex v do not occur in a valid path $\mathcal{P}$ and every vertex in v.`predecessors` is in $\mathcal{P}$ then the path $\mathcal{P} + [v]$ is still valid.

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

Definition 1.2.2 (Safe vertex): A vertex satisfying the conditions of Prop 1.2 for a path $\mathcal{P}$ is called a *safe* vertex for $\mathcal{P}$.

Prop 1.3: Valid paths of length $k-1$ starting from vertices v_i with empty v_i.`predecessors` contains the k closest samples from the input.

Proof: Notice that if the set `predecessors` of a vertex v_i is empty it means that the associated sample s_i is one of the closest sample to the input x since $\nexists s_j \in S$ s.t. $s_j \preccurlyeq s_i$ and $s_i \not\preccurlyeq s_j$. Then the proposition 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 path are generated given the the precedence graph $\mathbb{G}_x$ 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 having an empty set as `predecessors` (lines $1-2$) which means the traversal starts from the vertex associated with samples closest to the input 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 **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 among the adjacent of the path's last vertex 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 $\mathbb{G}_x$ abd $n \in \mathbb{N}$ Algorithm 2 returns all the valid paths of length[2] n within $\mathbb{G}_x$.

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:

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

Algorithm 2: `generate_paths` method

Input G_x: precedence graph, **n**: length of the path

Output Set of valid paths

```
1  closest_vertices ← {[v_i] | v_i ∈ 𝔾_x, v_i.precedence = ∅}
2  queue ← create_queue(closest_vertices)
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
```

- $(k = 1)$: In the first iteration of the loop the queue contains all the paths composed vertices with empty `predecessors` which are of length 0 and surely are valid by definition.
- $(k = h + 1)$: At iteration $h + 1$, the paths inside the queue at 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 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 x the algorithm first build the precedence graph $\mathbb{G}$ (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.

Algorithm 3: XXYYZZ algoritm

Input x: the input sample S: samples dataset
k: number of neighbours

Output Set of possible classification of x

```
1 𝔾_x ← create_precedence_graph(S, x)
2 paths ← generate_paths(𝔾_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
```

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

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 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

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)
```

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 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

Algorithm 5: `split_dataset` method

Input S: samples dataset

Output S_0: the left side dataset S_1: the right side dataset
π: 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** π
10 $d_{s'} \leftarrow$ distance between s' **and** π
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** π
17 $d_{s'} \leftarrow$ distance between s' **and** π
18 $\pi \leftarrow$ **translate**$\left(\pi,\ \frac{d_{s'}}{2},\ \overrightarrow{p_{\min}p_{\max}}\right)$
19 **return** $(S_{\max}, S_{\min}, \pi)$

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 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

```
1  queue ← create_queue({BSP-T.root})
2  partitions ← ∅
3  while queue is not empty do
4      current_node ← queue.pop()
5      if current_node is Leaf then
6          partitions ← partitions ∪ {current_node.dataset()}
7      else
8          new_nodes ← next_nodes(current_node, x)
9          queue.append(new_nodes)
10     end
11 end
12 return partitions
```

3 Naive abstract classifier

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

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

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 x which means the returned set of labels must contains all the output of the concrete classifier on each point of region that is

$$\text{XXYYZZ}^A(R) \supseteq \bigcup_{x' \in R} \text{XXYYZZ}(x') \tag{1}$$

Computing $\text{XXYYZZ}^A(P^\varepsilon(x))$ by applying the concrete classifier to 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 by applying XXYYZZ with a precedence graph (i.e., the one created in line 1 in Algorithm 3) that contains all the valid paths of each graph constructed by the concrete classifiers in Eq. 1, the output of XXYYZZ^A can be computed because the number of valid paths to explore is finite. This leads to Algorithm 7 which shows the abstract classifier to be the same as Algorithm 3 with the only difference being the creation of the precedence graph from which extracts the valid paths.

3.1 Abstract precedence graph

Let $\mathbb{G}_x^A$ be the (abstract) precedence graph build by XXYYZZ^A when called with the perturbation $P^\varepsilon(x)$ and suppose for example there are two samples $s_1, s_2 \in S$ and points $x_1, x_2 \in P^\varepsilon(x)$ such that

$$\begin{cases} s_1 \preccurlyeq_{\text{x1}} s_2 \text{ and } s_2 \not\preccurlyeq_{\text{x1}} s_1 \\ s_2 \preccurlyeq_{\text{x2}} s_1 \text{ and } s_1 \not\preccurlyeq_{\text{x2}} s_2 \end{cases}$$

Algorithm 7: XXYYZZ^A algoritm

Input x: the input sample ε: the degree of perturbation
S: samples dataset k: number of neighbours

Output Set of possible classification of x

1 $\mathbb{G}_x^A \leftarrow$ **create_abstract_precedence_graph**$(S,\ x,\ \varepsilon)$
2 `paths` $\leftarrow$ **generate_paths**$\big(\mathbb{G}_x^A,\ k-1\big)$
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`

that is s_1 is strictly closer to x_1 than s_2 while s_2 is strictly closer to x_2 than s_1. In this case paths in which s_1 is a predecessor of s_2 and those in which s_2 is a predecessor of s_1 are valid paths in $\mathbb{G}_{x_1}$ and $\mathbb{G}_{x_2}$ respectively and so they need to be both valid paths in the abstract precedence graph $\mathbb{G}_x^A$ as well. This leads to the definition of the following relation between samples:

Definition 3.1.1 ($\preccurlyeq^A_{(x,\varepsilon)}$ relation): Given $x \in \mathbb{R}^n, \varepsilon \in \mathbb{N}$ and $s_1 s_2 \in S$

$$s_1 \preccurlyeq^A_{(x,\varepsilon)} s_2 \iff \exists x_i \in P^\varepsilon(x)\ s_1 \preccurlyeq_{x_i} s_2$$

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

By this definition $s_1 \preccurlyeq^A_x s_2$ and $s_2 \preccurlyeq^A_x s_1$ and so there should be a edge between $\mathbb{G}^A_x[s_1]$ and $\mathbb{G}^A_x[s_2]$ in both direction. Consequently just as the concrete precedence graph is the graphical representation of the total order $(S, \preccurlyeq_x)$, the abstract precedence graph is the graphical representation of the total order $\left(S, \preccurlyeq^A_x\right)$. So the procedure `create_abstract_precedence_graph` is the same as Algorithm 1 with the only difference that the order relation used 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:

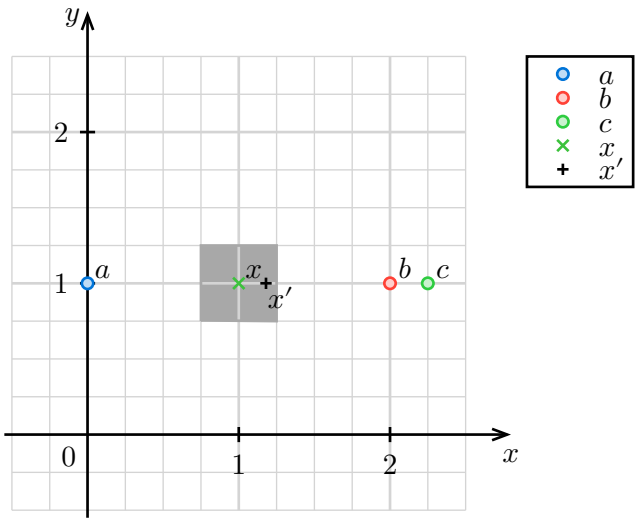
$$[\boldsymbol{b}, \boldsymbol{c}, \boldsymbol{a}]$$

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.

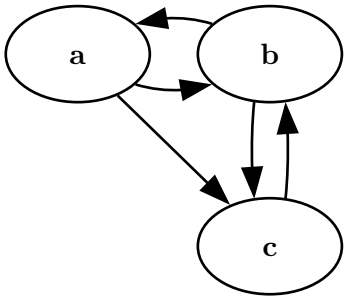
Prop 3.1: Given a sample $s \in S$

$$\mathbb{G}^A_x[s].\texttt{predecessors} = \bigcap_{x' \in P^\varepsilon(x)} \mathbb{G}_{x_i}[s].\texttt{predecessors}$$

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



a) Plot of dataset and perturbation of Example 1.1



b) Abstract precedence graph

3.2 Soundness of the abstract classifier

Given the perturbation $P^{\varepsilon}(x)$, 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

$$\bigcup_{x' \in P^{\varepsilon}(x)} \text{XXYYZZ}(x') \subseteq \text{XXYYZZ}^{A}(P^{\varepsilon}(x))$$

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

$$\bigcup_{x' \in P^{\varepsilon}(x)} \text{XXYYZZ}(x') \subseteq \text{XXYYZZ}^{A}(P^{\varepsilon}(x))$$

Proof: Is suffice the show that the abstract precedence graph $\mathbb{G}_x^A$ satisfy the condition

$$\bigcup_{x' \in P^{\varepsilon}(x)} \texttt{valid\_path}\big(\mathbb{G}_{x_i}\big) \subseteq \texttt{valid\_path}\big(\mathbb{G}_x^A\big) \tag{2}$$

where $\texttt{valid\_path}(G)$ denotes the set of valid paths in a graph G. Without loss of generality consider a valid path $p = [v_0, v_1, ..., v_{k-1}]$ in $\mathbb{G}_{x_i}$ for some $x_i \in P^{\varepsilon}(x)$. By definition of path in graph and $\preccurlyeq_{x_i}$ we have that

$$s_0 \preccurlyeq_x^A s_1 \preccurlyeq_x^A ... \preccurlyeq_x^A s_{k-1}$$

This means that p is also a path in $\mathbb{G}_x^A$. Moreover since p is a valid path and, by Prop 1.4, $\forall v_i \in p$

$$\mathbb{G}_x^A[s_i].\texttt{predecessors} \subset \mathbb{G}_{x_i}[s_i].\texttt{predecessors}$$

p satisfy the conditions of being valid also for $\mathbb{G}_x^A$ and so $p \in \texttt{valid\_path}\big(\mathbb{G}_x^A\big)$ □

3.2.1 Incompleteness

In general the abstract classifier is sound but not *complete* (or *exact*). For example consider the set of samples:

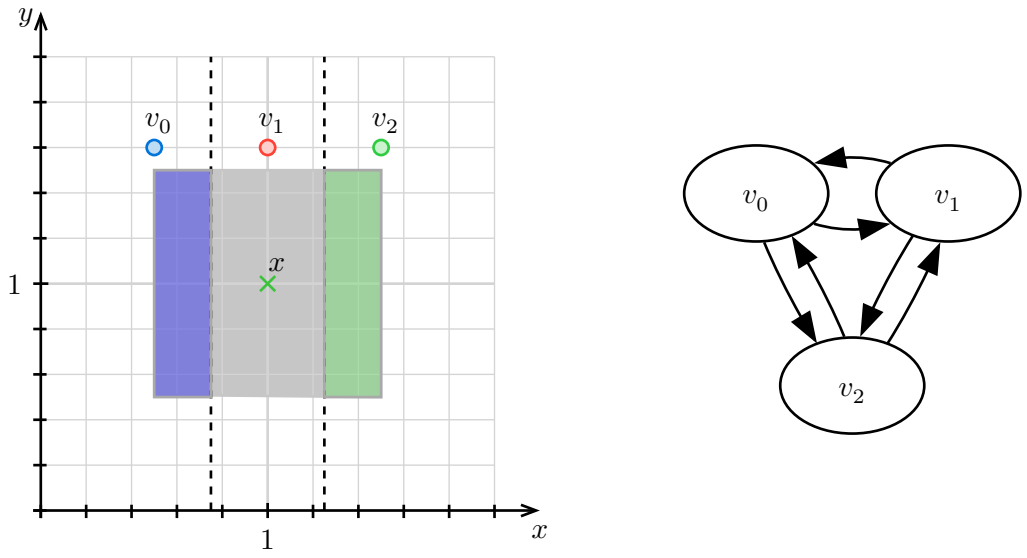
- s_0: (0.75, 1.3)
- s_1: (1.0, 1.3)
- s_2: (1.25, 1.3)

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. It easy to see that in this case the valid paths in the abstract precedence graph are the permutations of the sequence of vertices $[\boldsymbol{v_0}, \boldsymbol{v_1}, \boldsymbol{v_2}]$ in particular:

1. $[\boldsymbol{v_0}, \boldsymbol{v_1}, \boldsymbol{v_2}]$
2. $[\boldsymbol{v_0}, \boldsymbol{v_2}, \boldsymbol{v_1}]$

but there is no $x' \in P^{\varepsilon}(x)$ such that second path is a valid path in $\mathbb{G}_{x'}$. This is because the regions of space containing the points closer the sample s_0 (i.e. the blue region) and the one with point closer to sample s_2 than s_1 (i.e. the green region) do not intersect.

The only case in which the abstract classifier is complete is when $k = 1$ because in that case the vertex v_i that can occur in a valid path is the one for which there is point $x_i \in P^{\varepsilon}(x)$ such s_i is the closest sample to x_i.



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

Figure 3: Example of incompleteness

3.3 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 $A = \{s_0, s_1, ..., s_{k-1}\}$ to the input x and let s_k be the furthest sample from x with distance d. Now consider a generic point $x' \in P^{\varepsilon}(x)$ and the hypershpere $H_{x'}$ centered in x' and radius the distance between x' and $s_{x'} = \underset{s_i \in A}{\operatorname{argmax}} \|x' - s_i\|_p$. Surely the set of samples in $H_{x'}$ is $A \cup B$ where B is the set (possibly empty) of samples closer to x' than any sample in A. So the enclosing hypershpere H of all the hyperspheres $H_{x'}$ for every $x' \in P^{\varepsilon}(x)$ is the one that surely contains all the k closest sample of each point in the perturbation.

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

$$\|x' - y\|_p \leq \|x - x'\|_p + \|x - y\|_p$$

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

$$\|x' - s_{x'}\|_p \leq \|x - x'\|_p + \|x - s_{x'}\|_p \leq \sqrt{n} \cdot \varepsilon + d$$

where $s_{x'} = \underset{s_i \in A}{\operatorname{argmax}} \|x' - s_i\|_p$.

So the hypershpere of interest H is centered in the input x and has radius $2\varepsilon\sqrt{N} + d$.

4 A better abstract classifier

One issue with the naive abstract classifier is that there are cases in which is not efficient. For example consider the case in which there are n samples equidistant from the perturbation $P^\varepsilon(x)$ then in this case the abstract classifier 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. The problem with the naive abstract classifier approach is that it explicitly enumerate all the possible paths before classifying the input but is not necessary do so. Notice that the classification of the input does not depend on the order of the k closest samples but only on the number of occurrences of each label. This means that paths can be also be represented as set of vertices rather than a sequence. Moreover samples with the same labels are equivalent from the classification point of view. For example suppose in the previous example $n = 10$, $k = 7$ and among the n samples

- 6 samples have the label l_1;
- 3 samples have the label l_2;
- 1 samples have the label l_3

then is easy to see that in any valid paths made by the equidistant samples at least 3 samples have label l_1 and since at most 3 samples can have label l_2 the possible classifications of the input are the labels l_1 and l_2.

This observations suggests a more efficient representation for a set of valid paths in $\mathbb{G}_x^A$. For example consider an abstract precedence graph $\mathbb{G}_x^A = (\{v_0, v_1, v_2, v_3\}, E)$ such that:

- $V_\emptyset = \{v_0, v_1\}$
- $V_{\{v_0\}} = \{v_2, v_3\}$

where $V_B = \{v \in V \mid v.\texttt{predecessors} = B\}$. With this setup the set of valid paths of length 2 contains the following paths:

- $P_1 = \{v_0, v_2, v_1\}$
- $P_2 = \{v_0, v_1, v_3\}$
- $P_3 = \{v_0, v_2, v_3\}$

Now suppose the tuple $(B, R, n) \in \mathcal{P}(V) \times \mathcal{P}(V) \times \mathbb{N}^+$ where $R \subseteq B$ and $|R| \leq n \leq |B|$ denotes the set of subsets of B of size n containing the set R then the set

$$A_{p_{12}} = \left\{ A_0 \cup A_1 \mid (A_0, A_1) \in (V_\emptyset, \{v_0\}, 2) \times \left(V_{\{v_0\}}, \emptyset, 1\right) \right\}$$

contains exactly the path P_1 and P_2 since:

- $(V_\emptyset, \{v_0\}, 2) = \{\{v_0, v_1\}\}$
- $\left(V_{\{v_0\}}, \emptyset, 1\right) = \{\{v_2\}, \{v_3\}\}$

Similarly the set

$$A_{p_3} = \left\{A_0 \cup A_1 \mid (A_0, A_1) \in (V_\emptyset, \{v_0\}, 1) \times \left(V_{\{v_0\}}, \emptyset, 2\right)\right\}$$

contains exactly the path P_3 as:

- $(V_\emptyset, \{v_0\}, 1) = \{\{v_0\}\}$
- $\left(V_{\{v_0\}}, \emptyset, 2\right) = \{\{v_2, v_3\}\}$

This example illustrates a way of representing a set of paths more efficiently without enumerating all of them. Essentially the idea is create sets like $A = \{(B_i, R_i, n_i)\}_{i\in\{1,...,n\}}$ containing n tuples $(B_i, R_i, n_i) \in \mathcal{P}(V) \times \mathcal{P}(V) \times \mathbb{N}^+$ such that $\sum_{i=1}^{i=n} = k$. More formally

Definition 4.1 (Abstract vertex): Given the abstract precedence graph $\mathbb{G}_x^A = (V, E)$ the tuple $v^A = (B, R, n) \in (\mathcal{P}(V) \setminus \emptyset) \times \mathcal{P}(V) \times \mathbb{N}^+$ such that $R \subseteq B$ and $|R| \leq n \leq |B|$ is called an *abstract vertex* and represents a set of set of vertices, denoted with $\langle v^A \rangle$, which is defined as

$$\langle v^A \rangle = \{\{v_0, v_1, ..., v_{n-1}\} \subseteq B \mid R \subseteq \{v_0, v_1, ...v_{n-1}\}\}$$

Basically $\langle A, R, n \rangle$ denotes the set of subsets of A of size n containing the set R.

Definition 4.2: Given an abstract vertex $v^A = (B, R, n)$:

- $\texttt{set}(v^A)$ is the underlying set of vertices of the abstract vertex that is $\texttt{set}(v^A) = B$.
- $\texttt{req}(v^A)$ are the required vertices in every set of vertex in $\langle v^A \rangle$ which means $\texttt{req}(v^A) = R$.
- $\texttt{len}(v^A)$ is the size of the sets in $\langle v^A \rangle$ that is $\texttt{len}(v^A) = n$

Definition 4.3 (Abstract Path): Given the abstract precedence graph $\mathbb{G}_x^A = (V, E)$ the set $\mathsf{P}^A = \left\{v_i^A\right\}_{i\in\{1,...,n\}} \in \mathcal{P}((\mathcal{P}(V) \setminus \emptyset) \times \mathcal{P}(V) \times \mathbb{N}^+)$ containing n abstract vertices is called an *abstract path* and represents a set of paths in $\mathbb{G}_x^A$, denoted with $\langle\!\langle \mathsf{P}^A \rangle\!\rangle$, which is defined as

$$\langle\!\langle \mathsf{P}^A \rangle\!\rangle = \left\{ \bigcup_{1\leq i\leq n} V_i \mid (V_1, V_2, ..., V_n) \in \langle v_1^A \rangle \times \langle v_2^A \rangle \times ... \times \langle v_n^A \rangle \right\}$$

Essentially $\langle\!\langle \mathsf{P}^A \rangle\!\rangle$ contains the sets of vertices obtained by taking the union of the sets of vertices in the cartesian product $\langle v_1^A \rangle \times \langle v_2^A \rangle \times ... \times \langle v_n^A \rangle$.

Definition 4.4: Given an abstract path $\mathsf{P}^A = \left\{v_i^A\right\}_{i\in\{1,...,n\}}$

- $\texttt{set}^\star(\mathsf{P}^A)$ is the set containing the underlying set of each abstract vertex in p^A that is $\texttt{set}^\star(p^A) = \left\{\texttt{set}(v^A) \mid v^A \in p^A\right\}$
- $\texttt{req}^\star(\mathsf{P}^A)$ is the set comprising the required vertices of each abstract vertex in p^A that is $\texttt{req}^\star(p^A) = \left\{\texttt{req}(v^A) \mid v^A \in p^A\right\}$
- $\texttt{len}^\star(\mathsf{P}^A)$ is the length of the abstract path and represent the length the paths in $\langle\!\langle \mathsf{P}^A \rangle\!\rangle$ that is $\texttt{len}^\star(\mathsf{P}^A) = \sum_{i=0}^{i=n} \texttt{len}(v_i^A)$.

So following this definitions:

$A_{p_{12}} = \langle\!\langle \{(V_\emptyset, \{v_0\}, 2), (V_{\{v_0\}}, \emptyset, 1)\} \rangle\!\rangle$ and $A_{p_3} = \langle\!\langle \{(V_\emptyset, \{v_0\}, 1), (V_{\{v_0\}}, \emptyset, 2)\} \rangle\!\rangle$

4.1 Abstract paths generation

For a given abstract precedence graph $\mathbb{G}_x^A = (V, E)$ not all abstract paths contains exclusively paths that are also valid. To ensure that this is the case, the set of vertices is partitioned according to the value of the `predecessors` attribute. So if, given a subset $B \subseteq V$ of size i,

$$V_B^i = \{v \in V \mid V.\texttt{predecessors} = B\}$$

then partition is equal to the following set

$$VS = \left\{V_{B_{01}}^0, V_{B_{11}}^1, V_{B_{12}}^1, ..., V_{B_{1i_1}}^1, V_{B_{21}}^2, ..., V_{B_{2i_2}}^2, ..., V_{B_{m1}}^m, ..., V_{B_{mi_m}}^m\right\}$$

Afterwards the abstract paths are constructed using abstract vertices $v^A = (A, R, n)$ such that $A \in VS$ leading to the addition of the following utility function:

- $\texttt{pred}(v^A)$ denotes the set of predecessors of the vertices in $\texttt{set}(v^A)$ that is if $\texttt{set}(v^A) = V_B^i$ then $\texttt{pred}(v^A) = B$

With this setup a valid abstract paths is defined as:

Definition 4.1.1 (Valid abstract path): Given the abstract precedence graph $\mathbb{G}_x^A = (V, E)$ an abstract path $P^A = \{v_i^A\}_{i \in \{1,...,n\}} \in \mathcal{P}(VS \times \mathcal{P}(V) \times \mathbb{N}^+)$ is *valid* if every path in $\langle\!\langle A \rangle\!\rangle$ is also valid which occurs when for every $v_0^A \in P^A$ the following conditions are satisfied

1. $\texttt{pred}(v_0^A) \subseteq \bigcup_{v_i^A \in P^A} \texttt{set}(v_i^A)$
2. $\forall v^A \in P^A \quad \texttt{pred}(v_0^A) \cap \texttt{set}(v^A) \subseteq \texttt{req}(v^A)$

The condition above essentially means that for every vertex v in a path $p \in \langle\!\langle P^A \rangle\!\rangle$ the predecessors of v are also present in p.

Moreover, similarly to the concrete case, not every abstract vertex v^A is *safe* for a valid abstract path P^A in the sense that it could happen that by adding v^A to P^A the abstract path is no more valid. This lead to the following definition

Definition 4.1.2 (Safe abstract vertex): Let $P^A = \{v_i^A\}_{i \in \{1,...,n\}}$ be a valid abstract path for a given abstract precedence graph $\mathbb{G}_x^A = (V, E)$. The abstract vertex v^A is safe for P^A if the following conditions are satisfied:

- $\underline{\texttt{set}(v^A) \in \texttt{set}^\star(P^A)}$: In this case let $v_0^A \in P^A$ such that $\texttt{set}(v^A) = \texttt{set}(v_0^A)$ then v^A is safe for P^A if and only if $\texttt{req}(v^A) = \emptyset$ and $\texttt{len}(v^A) + \texttt{len}(v_0^A) \leq \texttt{set}(v^A)$.
- $\underline{\texttt{set}(v^A) \notin \texttt{set}^\star(P^A)}$: In this cases v^A is safe for P^A if and only if the following conditions are satisfied:

  1. $\texttt{pred}(v^A) \subseteq \bigcup_{v_i^A \in P^A} \texttt{set}(v_i^A)$
  2. $\forall v_0^A \in P^A \quad \texttt{len}(v_0^A) \geq |(\texttt{req}(v_0^A) \cup \texttt{pred}(v^A)) \cap \texttt{set}(v_0^A)|$

  Essentially together the two conditions above states it must exists a set in $\langle\!\langle P^A \rangle\!\rangle$ that contains $\texttt{pred}(v^A)$.

Algorithm 8: extend_abs_path method

Input P^A: An abstract path, v^A: An abstract vertex

Output P^A extend with v^A satisfying the conditions of Definition 4.1.1

```
1  new_path ← P^A
2  if not safe(P^A, v^A) then
3  |  return new_path
4  end
5  if set(v^A) ∈ set*(new_path) then
6  |  v_0^A ← v ∈ P^A such that set(v) = set(v^A)
7  |  len(v_0^A) ← len(v_0^A) + len(v^A)
8  else
9  |  for v_i^A in new_path do
10 |  |  req(v_i^A) ← req(v_i^A) ∪ (pred(v^A) ∩ set(v_i^A))
11 |  end
12 |  new_path ← new_path ∪ {v^A}
13 end
14 return new_path
```

Definition 4.1.2 only states the conditions that must be satisfied so that an abstract vertex can be added to an abstract path but after the addition the existing elements of the abstract path needs to be changed in order to satisfy the conditions of the Definition 4.1.1. Algorithm 8 shows how an abstract path P^A is extend with an abstract vertex v^A. It stars by first creating a copy, the variable `new_path`, of the abstract path P^A (line 1) and then check whether v^A is a safe vertex for P^A. If it is not safe then simply return the copied path unaltered (line $2-3$) otherwise if exists an abstract vertex $v \in$ `new_path` such that $\texttt{set}(v) = \texttt{set}(v^A)$ then increase the size of the set of vertices in $\langle v \rangle$ by $\texttt{len}(v^A)$ (line $4-6$). On the other hand if no abstract vertex in `new_path` has the same underlying set as v^A then for each abstract vertex $v_i^A \in$ `new_path` adds to $\texttt{req}(v_i^A)$ the vertices in $\texttt{pred}(v^A)$ that also belong to $\texttt{set}(v_i^A)$ (line $7-10$). Finally extends `new_path` with the new abstract vertex v^A before returning it (line $11-12$).

Prop 4.1: For every valid abstract path P^A and abstract vertex v^A, $\texttt{extend\_path}(P^A, v^A)$ is always a valid abstract path.

Proof: Let $P_1^A = \texttt{extend\_path}(P^A, v^A)$. By definition if v^A is not a safe for P^A then P_1^A is simply an unaltered copy of P^A which is valid by hypothesis otherwise for every abstract vertex $v_i^A \in P_1^A$:

$$\left(\texttt{pred}(v^A) \cap \texttt{set}(v_i^A)\right) \subseteq \texttt{req}(v_i^A)$$

Since for every $v_i^A \in P_1^A$ $\texttt{set}(v_i^A)$ is not modified and $\texttt{req}(v_i^A)$ is only augmented this means that the conditions of Definition 4.1.1 are satisfied and so in both cases P_1^A is a valid abstract path. □

Prop 4.2: Let P^A and v^A be a valid abstract path and abstract vertex respectively and let $P_1^A = \texttt{extend\_path}(P^A, v^A)$. If v^A is not safe for P^A then $\texttt{len}^\star(P_1^A) = \texttt{len}^\star(P^A)$ otherwise $\texttt{len}^\star(P_1^A) = \texttt{len}^\star(P^A) + \texttt{len}(v^A)$.

Proof: by definition of `extend_path` if v^A is not safe for P^A then $P_1^A = P^A$ and so $\texttt{len}^\star(P_1^A) = \texttt{len}^\star(P^A)$. If v^A is safe then $P_1^A = P^A \cup \{v^A\}$ and so

$$\texttt{len}^\star(P_1^A) = \texttt{len}^\star(P^A) + \texttt{len}(v^A)$$

by definition of the $\texttt{len}^\star$ function. □

The method `extend_path` is used for the generation of abstract paths of length k. Algorithm 9 shows how, given an abstract precedence graph $G_x^A = (V, E)$ and the length k, the abstract paths of length k are generated. It make use of three quantities:

- The variable n, which denotes the length of the abstract paths being created.
- The variable `abs_vertices` which is a set containing the abstract vertices that will be used to extend the abstract paths.
- The variable `abs_paths` which is a set containing the abstract paths being generated.

The algoritm starts by initializing the variable n with 0, `abs_vertices` with the set containing all the abstract vertices v^A such that

- $\texttt{set}(v^A) = V_B \neq \emptyset$ where $V_B = \{v \in V |\ v.(\texttt{predecessors}) = \texttt{B}\}$ for some $B \subseteq V$
- $\texttt{req}(v^A) = \emptyset$
- $\texttt{len}(v^A) = 1$

and `abs_paths` with a set containing the valid abstract path of length 1 (lines 1-3). Then repeat indefinitely the following (lines 4-17):

1. Increase n by 1 and check wether the paths in `abs_paths` are of the desired length k and if this is the case then simply return the set `abs_paths` (lines 5-8).

2. Construct a new set of abstract paths by extending each abstract path in `abs_paths` with the safe abstract vertices in `abs_vertices` through the method `extend_abs_path` and afterwards assign it to `abs_paths` (lines 9-16).

Prop 4.3: Algorithm 9 generates all the valid abstract paths of the desired length k.

Proof: The proposition can be proven by showing that in the i-th iteration of the while loop, at line 5 (i.e. before the check on the length) the `abs_paths` contains all the valid abstract path of length i. We use induction on i:

- $\underline{(i = 1)}$: In the first iteration `abs_paths` contains only the abstract path $\{(V_\emptyset^0, \emptyset, 1)\}$ which is valid by definition. Moreover it is easy to see that is also the only valid path of length 1.

- $\underline{(i = h + 1)}$: Let H be the set `abs_paths` in the h-th iteration. So in the $h + 1$-th iteration the abstract paths in `abs_paths` are obtained by extending each paths in H with all the possible safe abstract vertices v^A with length 1 and $\texttt{req}(v^A) = \emptyset$. By induction hypothesis every abstract path in H are valid with length h and so, by Prop 4.1 and Prop 4.2 each path in `abs_paths` is also valid and has length $h + 1$. This means that `abs_paths` contains all the valid abstract path of length $h + 1$

So at iteration k of the while the algorithm returns all the valid abstract paths of the desired length k. □

Algorithm 9: generate_abstract_paths method

Input G_x^A: An abstract precedence graph, k: Required length of the abstract path

Output Set of abstract paths of length k

```
1  n ← 0
2  abs_vertices ← {(V_B, ∅, 1) | B ∈ V} \ {(∅, ∅, 1)}
3  abs_paths ← {{(V_∅, ∅, 1)}}
4  while true do
5      n ← n + 1
6      if n = k
7          return abs_paths
8      end
9      new_abs_paths ← ∅
10     for abs_vertex in abs_vertices do
11         for abs_path in abs_paths do
12             new_abs_path ← extend_path(abs_path, abs_vertex)
13             if new_abs_paths ≠ abs_path
14                 new_abs_paths ← new_abs_paths ∪ {new_abs_path}
15             end
16         end
17     end
18     abs_paths ← new_abs_paths
19     end
20 end
```

Prop 4.4: Given an abstract precedence graph $G_x^A = (V, E)$ and $k \in \mathbb{N}^+$ let AP_k be the set of valid abstract paths of length k (i.e. $AP_k = \texttt{generate\_abs\_paths}(G_x^A, k)$). Then the set

$$CP_k = \bigcup_{P_i^A \in AP_k} \langle\!\langle P_i^A \rangle\!\rangle$$

contains all the possible valid paths of length $k-1$ in G_x^A.

Proof: The proposition can be proved using induction on the length k of the abstract paths in AP_k:

- $\underline{(k = 1)}$: In this case $AP_k = \{\{(V_\emptyset, \emptyset, 1)\}\}$ and by definition of the $\langle\!\langle \cdot \rangle\!\rangle$ operator

  $$CP_k = \langle\!\langle \{(V_\emptyset, \emptyset, 1)\} \rangle\!\rangle = \{\{v\} \mid v \in V_\emptyset\} = \{\{v\} \mid v \in V \wedge v.\texttt{predecessors} = \emptyset\}$$

  So CP_k contains all the paths made of a single vertex $v \in V$ such that $v.\texttt{predecessors} = \emptyset$ which by definition are valid and with length $k-1$.

- $\underline{(k = h + 1)}$: In this case, by definition of the method `generate_abs_paths` and because $AP_k = \texttt{generate\_abs\_paths}(G_x^A, k)$, the abstract paths in AP_k are obtained by extend-

ing each path in $AP_h = \texttt{generate\_abs\_paths}\big(G_x^A, h\big)$, where $h \in \mathbb{N}^+$, with all the possible safe abstract vertices in the set

$$AV = \{(V_B, \emptyset, 1) \mid B \in V\} \setminus \{(\emptyset, \emptyset, 1)\}$$

This means that, following the definition of $\langle\!\langle \cdot \rangle\!\rangle$ operator and safe abstract vertex

$$CP_k = \big\{P \cup \{v\} \mid P \in CP_h \wedge \exists\, v^A \in AV \text{ s.t. } \texttt{pred}\big(v^A\big) \subseteq P \wedge v \in \langle v^A \rangle\big\}$$

In other words CP_k is obtained by adding every safe vertex $v \in V$ to each path of CP_h. By induction hypothesis CP_h contains all the valid paths of length $h-1$ so it follows that CP_k will contains all the valid paths of length $h = k-1$

□

4.2 Classification

For the classification of the input once all the abstract path of length k are generated, it is necessary to compute all the possible occurrences of labels in the paths represented by the abstract paths. To do so let $\texttt{labels} : (\mathcal{P}(V) \setminus \emptyset) \times \mathcal{P}(V) \times \mathbb{N}^+ \to \mathcal{P}(\mathcal{P}(V \times \mathbb{N}^+))$ be a function that given an abstract vertex v^A yields a set whose elements are multisets (or bag) [4] quantifying the occurrences of each label found in some vertex sets within $\langle v^A \rangle$. To understand how the function $\texttt{labels}$ is defined notice that an element A of $\langle v^A \rangle$ is the union of the set $\texttt{req}\big(v^A\big)$ and a subset B of $\texttt{set}\big(v^A\big) \setminus \texttt{req}\big(v^A\big)$ with size $\texttt{len}\big(v^A\big) - |\texttt{req}\big(v^A\big)|$. So if R_L and B_L are the multisets containing the labels in $\texttt{req}\big(v^A\big)$ and $\texttt{set}\big(v^A\big) \setminus \texttt{req}\big(v^A\big)$ respectively then the function $\texttt{labels}$ is defined as

$$\texttt{labels}\big(v^A\big) = \{R_L \oplus B \mid B \subseteq B_L\}$$

where $\oplus$ is the sum operation between multisets.

With the function $\texttt{labels}$ its easy to define the function $\texttt{labels}^\star$ that given an abstract path P^A return a set of multisets each expressing the occurrences of the labels in a path within $\langle\!\langle P^A \rangle\!\rangle$. So given the abstract path $P^A = \big\{v_i^A\big\}_{i \in \{1,\dots,n\}}$ the function $\texttt{labels}^\star : \mathcal{P}(\mathcal{P}((\mathcal{P}(V) \setminus \emptyset) \times \mathcal{P}(V) \times \mathbb{N}^+)) \to \mathcal{P}(\mathcal{P}(V \times \mathbb{N}^+))$ is defined as

$$\texttt{labels}^\star\big(P^A\big) = \big\{A_0 \oplus A_1 \oplus \dots \oplus A_n \mid A_i \in \texttt{labels}\big(v_i^A\big), i \in \{1, \dots, n\}\big\}$$

Once all the multisets are generated then all the possible classifications of the input are the most frequent labels in each multisets.

4.3 Abstract classifier

Algorithm 10 illustrate how the abstract classifier works which is not too much different from the naive abstract classifier. Given as input the set of samples S, the input x to be classified, the degree of perturbation ε applied to the input and the number k of neighbours to consider for the classification, the algorithm starts by creating the abstract precedence graph $\mathbb{G}_x^A$ (line 1) which is then used to generate all the possible valid abstract paths of length k (line 2). Afterwards the set containing all the possible classification of the input x is constructed which is then return at the end (line 3-11). To do so, for each abstract path generated before, the multisets expressing all the possible occurrences of each label is computed and then the most frequent labels are collected from each multiset (line 5-10).

Algorithm 10: XXYYZZ2^A algorithm

Input x: the input sample ε: the degree of perturbation
S: samples dataset k: number of neighbours

Output Set of possible classifications of the input

```
G_x^A ← create_abstract_precedence_graph(S, x, ε)
abs_paths ← generate_abstract_paths(G_x^A, k)
classifications ← ∅
for abs_path in abs_paths do
    abs_path_labels ← labels*(abs_path)
    for labels in abs_path_labels do
        most_frequent ← most frequent labels in labels
        classifications ← classifications ∪ {most_frequent}
    end
end
return classifications
```

Theorem 4.1: Given the dataset $S = \{(s^i, y^i) \mid s^i \in \mathbb{R}^n, y^i \in \mathcal{L}\}$ and the perturbation $P^\varepsilon(x)$ of the input sample x, XXYYZZ^A and XXYYZZ2^A are equivalent that is for every $k \in \mathbb{N}^+$ and $\varepsilon \in \mathbb{R}$

$$\text{XXYYZZ}^A(x, \varepsilon, S, k) = \text{XXYYZZ2}^A(x, \varepsilon, S, k)$$

Proof: The equivalence derive from the fact that both XXYYZZ^A and XXYYZZ2^A classify the input by computing the most frequent labels in every valid path of length $k-1$. This is valid for XXYYZZ^A because it is an explicit part of its definition while XXYYZZ2^A indirectly accomplishes this due to Prop 4.4, since the set of all abstract valid path of length k is representative of all valid paths of length $k-1$, therefore XXYYZZ2^A inherently considers the full spectrum of label frequencies found within any valid path of length $k-1$. □

4.4 Comparison with NAVe

Some comparisons with Nave tool with perturbation $\varepsilon = 0.01$ and interval domain.

Dataset	Runtime (mm:ss)			Stability		Robustness	
	NAVe	XXYYZZ2^A		NAVe	XXYYZZ2^A	NAVe	XXYYZZ2^A
Fourclass	00:01	00:09	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

Dataset	Runtime (mm:ss)			Stability		Robustness	
	NAVe	XXYYZZ2^A		**NAVe**	XXYYZZ2^A	**NAVe**	XXYYZZ2^A
Pendigits	10:16	09:54	k=1	96.7	97.7	95.6	96.4
			k=2	94.0	95.3	93.3	94.5
			k=3	96.1	97.8	95.3	96.6
			k=5	95.7	97.7	94.9	96.4
			k=7	95.0	97.6	94.1	96.2
Letter	37:44	30:33	k=1	82.7	88.6	82.2	87.8
			k=2	70.4	78.4	70.4	78.3
			k=3	75.1	85.9	75.0	85.5
			k=5	69.5	83.5	69.4	83.0
			k=7	65.2	82.1	65.1	81.7

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.

[4] L. da F. Costa, "An Introduction to Multisets," 2021.