Appendix A. Extraction of Subclass Relations

A.1 Extraction of Noun Phrases

Supposing the input is a set of texts $D = \{d_1, d_2, \dots, d_{|D|}\}$, where each text consists of sentences denoted as $d_i = \{s_1, s_2, \dots, s_{|d_i|}\}$. Extract nouns and noun phrases from a set of texts D by Algorithm A.1, which mainly consists of the following three steps:

1. Transform each sentence in D into the parse tree (representing the syntactic structure of the sentence derived from a context-free grammar, where the leaf nodes are the words of the sentence and the other nodes are non-terminal terms defined by the grammar, e.g., *nltk.tree.Tree* data structure in NLTK).
2. For all subtrees in the parse tree, check if a subtree matches the modifier pattern (by Algorithm A.2 that inputs a subtree and the noun phrase pattern and then outputs a noun phrase), add the matched noun/noun phrase in form of a truple (*pre-modifier*, *head noun*, *post-modifier*) to a set S_{np}. The identification of part-of-speech of words is implemented by NLTK (https://www.nltk.org/).
3. Remove the following two types of nouns and noun phrases that are unsuitable for representing a class (by the RemoveNonClassNP function): (1) its head noun is neither a common noun nor a plural noun, and (2) it belongs to one of the types of TABLE 1.

```
Algorithm A.1: ExtractNounPhrase
Input:    $D = \{d_1, d_2, \dots, d_{|D|}\}$, each $d_i = \{s_1, s_2, \dots, s_{|d_i|}\}$, $NounPhrasePattern$
Output:   $S_{np}$ which is the set of the extracted nouns/noun phrases
1   $S_{np} \leftarrow \emptyset$
2   for $d_i \in D$ do
3     for $s_j \in d_i$ do
4       $parsetree = \boldsymbol{GetParseTree}(s_j)$
5       for $subtree \in \boldsymbol{TraverseTree}(parsetree)$ do
6         if $\boldsymbol{Match}(subtree, NounPhrasePattern)$ is not $None$ then
7           $(pre, head, post) \leftarrow \boldsymbol{Match}(subtree, NounPhrasePattern)$
8           $S_{np} \leftarrow S_{np} \cup \{(pre, head, post)\}$
9   $S_{np} \leftarrow \boldsymbol{RemoveNonClassNP}(S_{np})$
10  return $S_{np}$
```

```
Algorithm A.2: Match
Input:    $subtree$ and $NounPhrasePattern$
Output:   $(pre, head, post)$ or $None$
1   $pre, head, post \leftarrow \emptyset, \emptyset, \emptyset$
2   $flag_{pre}, flag_{head}, flag_{post} \leftarrow 0, 0, 0$
3   for $i$ in $\boldsymbol{Range}(\boldsymbol{Len}(subtree.child))$ do
      // supposing the leaf of $subtree.child[i]$ is the head noun
4     if $subtree.child[i].label$ is a $Noun$ then
5       $flag_{head} \leftarrow 1$
6       $head \leftarrow subtree.child[i].leaves$
      // determine whether the parse trees before $subtree.child[i]$ satisfy the <pre-modifier> of $NounPhrasePattern$
7       for $tree \in subtree.child[:i-1]$ do
8         if $tree.label$ matches the elements in <pre-modifier> then
9           $flag_{pre} \leftarrow 1$
10          $pre \leftarrow pre \cup \{tree.leaves\}$
11        else
12          $flag_{pre} \leftarrow 0$
13          break
        // determine whether the parse trees after $subtree.child[i]$ satisfy the <post-modifier> of $NounPhrasePattern$
```

```
14        if $subtree.child[i+1:].label$ matches <post-modifier> then
15            $flag_{post} \leftarrow 1$
16            $post \leftarrow subtree.child[i+1:].leaves$
17    if $flag_{pre}$ and $flag_{head}$ and $flag_{post}$ then
18        return $(pre, head, post)$
19 return $None$
```

A.2 Identification of Subclass Relations

Identify the subclass relations between any two nouns and noun phrases within S_{np} by comparing whether the heads of them are the same or not (line 1-2) and whether the set of the modifiers of one is a subset of that of the other (line 3-9) by Algorithm A.3, where np_1, $np_2 \in S_{np}$.

```
Algorithm A.3: ExtractSubclassRelation
Input:     $np_1 = (pre_1, head_1, post_1)$ and $np_2 = (pre_2, head_2, post_2)$
Output:    $SubclassRelation$ or $None$
1  if not $\boldsymbol{Equal}(head_1, head_2)$ then
2      return $None$
3  $modifier_1 \leftarrow pre_1 \cup \{post_1\}$
4  $modifier_2 \leftarrow pre_2 \cup \{post_2\}$
5  if $modifier_1 \subset modifier_2$ then
6      return $np_2$ is the subclass of $np_1$
7  if $modifier_2 \subset modifier_1$ then
8      return $np_1$ is the subclass of $np_2$
9  return $None$
```

Appendix B. Split a Class Tree to Avoid Polysemy of Class Name

B.1 Use Common Words of the Texts of Class

Polysemy may appear among the names of classes, e.g., Figure B.1 illustrates the class tree of "bank" constructed from DUC2002 dataset (with 567 news articles), where the "bank" in "bank of the river" and that in "US investment bank" has different meanings. A way to solve the problem is to split the class tree with polysemy into different trees such that each tree has no polysemy.

As common words of a set of texts represent a commonality and the texts of the same class share more common words than the texts of different classes, the common words of classes can be used to solve the problem of polysemy. Algorithm B.1 uses the common words of the subclasses of the root to re-classify the subclasses into several clusters that belong to new trees because fewer common words of more texts can better represent a class than more common words of fewer texts. The clustering function is implemented by using an agglomerative method [38], which views each subclass as a cluster and then merges the two most similar clusters according to the similarity between two clusters C_i and C_j measured by $|Commonwords(C_i) \cap Commonwords(C_j)|/|Commonwords(C_i) \cup Commonwords(C_j)|$ where $Commonwords(C_i) = \cap_{n \in C_i} Commonwords(n) - \{n_0\}$ as the root is the polysemy that appears in the common words of the texts in each subclass. The clustering process is terminated when any two clusters do not share common words.

Algorithm B.1 can also be used to detect whether a class tree has polysemy: if the number of the trees of the clustering in Algorithm B.1 is 1, the class tree does not have polysemy, otherwise the class tree has polysemy.

B.2 Verification

To verify the performance of Algorithm B.1, we conducted experiments on the "bank" class tree extracted from the DUC2002 dataset. The results are evaluated by the precision of the classification of the classes represented by nodes.

We adopt two approaches to observe the effect: (1) the approach based on common words of the subclasses of the root, which generates three new trees as shown in Figure B.1. The result shows that the precision of the classification of the nodes is 89.47%; and, (2) the approach based on the common words of the leaf nodes of the class tree, and the precision of the classification of the leaf nodes is 78.95%. An interpretation is that some of the leaf nodes contain only one text which reduces the representativeness of common words in representing a class.

```
Algorithm B.1: SplitTree (T)
Input:      A class tree $T = \langle V, SR \rangle$, where $V = \{n_0, n_1, \dots, n_{|V|}\}$ is the set of classes represented by nouns/noun phrases and $n_0$ is the root.
            The nodes are extracted from the texts by the proposed approach. Each class (denoted as $n$) contains a set of texts denoted as $R(n)$
            and a set of common words of the texts obtained by the function $Commonwords(n)$. $SR = \{(n_i, n_j) | n_i, n_j \in V, n_j$ is the subclass of $n_i\}$
            is the set of subclass relations.
External    $\boldsymbol{GetSubclass}(n)$ // input class $n$ and then return the set of its subclasses
Functions:  $\boldsymbol{Commonwords}(n)$ // input a class $n$ and then return a set of common words of R(n).
            $\boldsymbol{Clustering}(\{Commonwords(n_1), \dots, Commonwords(n_{|N|})\})$ // input a set of common words of the classes, and then return the
            clusters of the classes.
Output:     The new trees $T$ // each class with no superclass indentify a root of a new tree
1   $N \leftarrow \boldsymbol{GetSubclass}(n_0)$                    // $N = \{n_1, n_2, \dots n_{|N|}\}$
2   $Clusters \leftarrow \boldsymbol{Clustering}(\{Commonwords(n_1), \dots, Commonwords(n_{|N|})\})$
3   $V \leftarrow V - \{n_0\}$
4   for $n \in N$ do
5       $SR \leftarrow SR - \{(n_0, n)\}$
6   for $C \in Clusters$ do
7       $Commonwords(root_C) \leftarrow \bigcap_{n \in C} CommonWords(n)$ // $root_C$ is a noun or noun phrases, representing the new root of classes in
    $C$.
8       $V \leftarrow V \cup \{root_C\}$
9       for $n \in C$ do
10          $SR \leftarrow SR \cup (root_C, n)$
11  return $T = \langle V, SR \rangle$
```

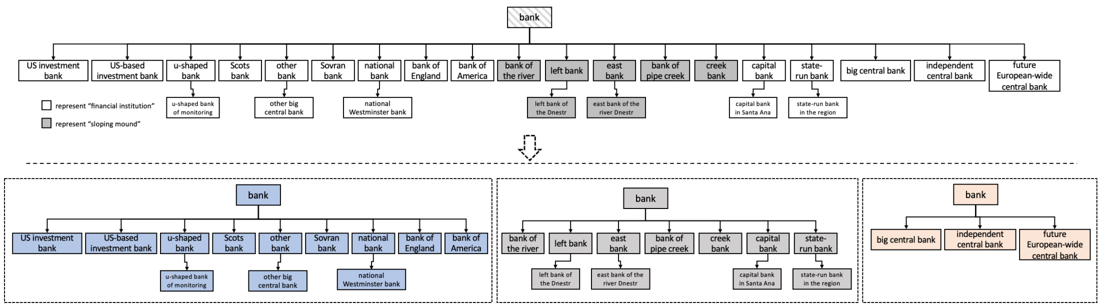

Figure B.1. Three new trees generated from the "bank" tree.

Appendix C. Comparison between Coverages of Three Approaches

The coverage of the class trees generated from the Summ dataset and AI dataset by three different approaches are shown in TABLE C.1 and C.2 respectively.

On the Summ dataset, the average coverages of the ten class trees extracted by the modifier-based approach, the Hearst Pattern approach and the ISA-Pattern approach are 96.10%, 4.00% and 1.10% and the average coverages of the class trees extracted by the integrated approach is the same as the modifier-based approach.

On the AI dataset, the average coverages of the ten class trees extracted by the modifier-based approach, the Hearst Pattern approach and the ISA-Pattern approach are 97.66%, 4.30% and 11.10% and the average coverages of the class trees extracted by integrated approach is 97.88%, which is 0.22% higher than the results of the modifier-based approach.

This comparison shows that the modifier-based approach proposed by this paper is suitable for extracting common abstraction relation. The cause is that the modifier-based approach is based on noun and noun phrases, which are widely used by various texts. But the other two approaches are based on specific words that are not widely used.

TABLE C.1. COVERAGE OF THE CLASS TREES EXTRACTED FROM SUMM DATASET BY DIFFERENT APPROACHES.

Top *k*	Roots of 10 Trees	Coverage of the Class Trees Generated by Modifier-Based Approach	Coverage of the Class Trees Generated by Hearst Pattern Approach	Coverage of the Class Trees Generated by ISA Pattern Approach	Coverage of the Class Trees Generated by Integrated Approach
1	*summarization*	99.00%	5.00%	2.00%	99.00%
2	*system*	97.00%	6.00%	1.00%	97.00%
3	*summary*	97.00%	2.00%	3.00%	97.00%
4	*set*	97.00%	1.00%	0.00%	97.00%
5	*number*	96.00%	3.00%	0.00%	96.00%
6	*result*	96.00%	0.00%	3.00%	96.00%
7	*information*	95.00%	4.00%	0.00%	95.00%
8	*document*	95.00%	7.00%	1.00%	95.00%
9	*method*	95.00%	12.00%	0.00%	95.00%
10	*evaluation*	94.00%	0.00%	1.00%	94.00%
-	Average	96.10%	4.00%	1.10%	96.10%

TABLE C.2. COVERAGE OF THE CLASS TREES EXTRACTED FROM AI DATASET BY DIFFERENT APPROACHES.

Top k	Roots of 10 Trees	Coverage of the Class Trees Generated by Modifier-Based Approach	Coverage of the Class Trees Generated by Hearst Pattern Approach	Coverage of the Class Trees Generated by ISA Pattern Approach	Coverage of the Class Trees Generated by Integrated Approach
1	*result*	100.00%	2.42%	8.06%	100.00%
2	*set*	99.46%	0.81%	31.45%	99.46%
3	*formula*	98.39%	12.90%	30.65%	98.92% ↑
4	*case*	98.39%	8.60%	0.00%	98.39%
5	*example*	97.85%	1.88%	4.84%	97.85%
6	*problem*	97.31%	15.59%	8.87%	97.85% ↑
7	*number*	96.77%	0.00%	26.88%	97.85% ↑
8	*section*	96.77%	0.00%	0.00%	96.77%
9	*order*	95.97%	0.54%	0.27%	95.97%
10	*way*	95.70%	0.27%	0.00%	95.70%
-	Average	97.66%	4.30%	11.10%	97.88% ↑

Appendix D. Construct Abstraction Trees of Verbs with WordNet

In addition to nouns and noun phrases, verbs can also be used to construct class tree in which the subclass relation can be constructed by using the WordNet that defines the synonymy and troponymy of verbs. Since there are rarely syntactic patterns to discover subclass relations of verbs, the approach uses the modifier pattern and the WordNet to construct class trees.

There are several patterns to construct verb phrase, e.g., the verb phrase "go slowly" can be considered as the subclass of "go", while the verb phrase "care for" should not be considered as the subclass of "care". Among the verb phrase patterns, the pattern <verb> <adverb> can be used as the basis for the modifier-based approach to identifying the subclass relation. Summ dataset is used to conduct the experiment to construct class trees of verbs. TABEL D.1 shows the details of the verb-based class trees with the coverage equal to or larger than 90%.

Figure D.1 shows a part of structure of the class trees of verbs. Compared to the class trees constructed by nouns/noun phrases, the class trees constructed by verbs/verb phrases have the following characteristic: (1) they have fewer nodes and subclass relations but have more synonymy relations; (2) the class trees of verbs are flatter than the class trees of nouns because verbs have fewer modifiers than nouns and noun phrases; (3) the average proportion of the subclass relations identified by the modifier-based approach based on verbs is 13.49%, while the approach based on noun and noun phrases is 83.79%, which illustrates that the construction of the class trees on verbs is mainly dependent on the WordNet; and, (4) the coverage of class trees on verbs is wider than the class trees on nouns and noun phrases, for example, the top-5 high-coverage class trees on verbs can commonly manage 100 texts while the top-5 high-coverage class trees noun can commonly manage 89 texts. The class trees on verbs/verb phrases provide an action dimension for accessing texts if the WordNet is available but the verbs render more general meaning than nouns.

TABLE D.1. COVERAGE OF THE CLASS TREES EXTRACTED FROM SUMM DATASET BY DIFFERENT APPROACHES.

Top-*k*	**Root of the Tree**	**Number of Nodes**	**Number of Subclass Relations**	**Number of Synonymy Relations**	**Number of Texts Represented by Top-*k* Trees**	**Number of Texts Commonly Represented by Top-*k* Trees**
1	*use*	32	27	11	100	100
2	*be*	121	114	79	100	100
3	*show*	49	35	47	100	100
4	*base*	5	1	7	100	100
5	*have*	54	42	59	100	100
6	*do*	57	45	34	98	98
7	*include*	8	7	1	97	96
8	*follow*	39	27	14	96	94
9	*describe*	32	20	32	95	91
10	*give*	66	50	75	93	85
11	*take*	134	95	153	92	81
12	*contain*	25	14	27	92	77
13	*make*	114	96	122	92	72
14	*consider*	40	31	24	91	66
15	*compare*	4	2	1	91	63
16	*perform*	29	26	11	90	61

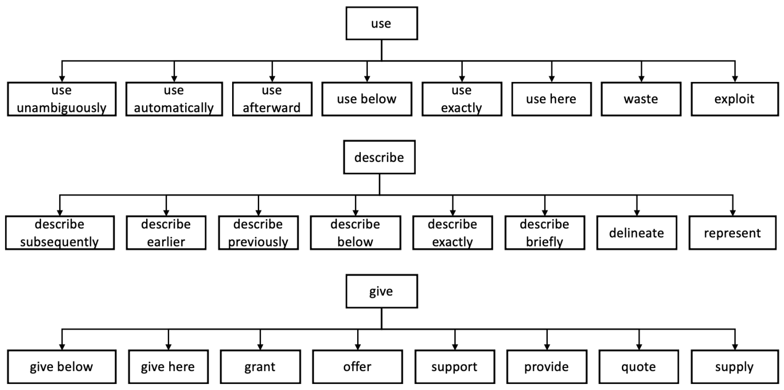

Figure D.1. A part of the class trees on verbs.

Appendix E. Management Operations

Retrieve resources, add resources, delete resources and merge spaces are four basic management operations of the resource space.

E.1 Establish Relations Between Resource Space and its Resources

Assume that the set of resources for generating its resource space is stored in a directory on a file system without losing generality. A resource is retrieved by its name and the path of the directory, denoted as *path*/*name*. The resources belonging to a class *n* (i.e., $\{path_1/name_1, \ldots, path_k/name_k\} \in n$) is obtained by the function $B(d, n)$. The efficiency of determining the classes of the set of resources is $O(1)$.

E.2 Retrieve Operations

The retrieval operation enables users or application systems to use the following SQL-like language to accurately obtain the interested resources from a resource space $RS = \{X_1, X_2, \ldots, X_N\}$ where $X_i = <V_i, SR_i>$ represents a class tree and $i \in [1, N]$. For example, the 2-dimensional resource space extracted from the GitHub dataset (1000 README text files) is represented as *RS* = <*network*, *model*>, where the *network* is a class tree consisting of 1428 classes and 1885 subclass relations between the classes <{*network*, *neural network*, *attention network*, *deep network*, …}, {(*network*, *neural network*), (*network*, *attention network*), (*network*, *deep network*), …}>, and the *model* is a class tree consisting of 1431 classes and 1722 subclass relations between the classes <{*model*, *classification model*, *ImageNet model*, *language model*, …}, {(*model*, *classification model*), (*model*, *ImageNet model*), (*model*, *language model*), …}>.

1. SELECT RESOURCE FROM *RS* WHERE $X_i = m$. It searches the dimension $X_i = <V_i, SR_i>$ in a resource space $RS = \{X_1, X_2, \ldots, X_N\}$ according to the given representation *m* (noun or noun phrase) of the expected class (i.e., find class *n* of the dimension according to the given *m*) and then returns $R(n)$, where $n \in V_i$, $n=m$. For example, the statement for retrieving projects from *RS* = <*network*, *model*> is represented as SELECT RESOURCE FROM *RS* WHERE *network* = "neural network". It returns 400 projects from 1000 projects of the resource space, each of which has a README file containing the noun phrase "neural network". As a project can be accessed from a class and all its super-classes, the number of classes is larger than the number of projects. As the search is on the class tree of a dimension, so it can quickly reduce the search space.
2. SELECT RESOURCE FROM *RS* WHERE $X_i = (m_1, m_2, \ldots, m_p)$. It searches the dimension $X_i = <V_i, SR_i>$ in a resource space $RS = \{X_1, X_2, \ldots, X_N\}$ according to a set of given representations $m_1, \ldots, m_P$ of the expected classes (i.e., find the classes $n_1, \ldots, n_P$ of the dimension according to the given $m_1, \ldots, m_P$) and then returns the set of texts accessed by these classes, i.e., $\bigcup_{j=1}^{P} R(n_j)$, where $n_j \in V_i$, $n_j = m_j$ and $j \in [1, P]$. For example, the following statement is for retrieving texts from *RS* = <*network*, *model*>: SELECT RESOURCE FROM *RS* WHERE *network* = ("recurrent neural network", "adversarial network"). It returns 106 projects consisting of the class "recurrent neural network" that contains 58 projects and "adversarial network" that contains 48 projects. As the *RS* is a 2-NF resource space, a project can only belong to one classes of a dimension. This statement supports users or application systems to retrieve resources from multiple classes of a dimension at one time.
3. SELECT RESOURCE FROM *RS* WHERE $X_1 = m_1$ AND $X_2 = m_2$ AND … AND $X_N = m_N$. It searches dimensions $X_1 = <V_1, SR_1>$, $X_2 = <V_2, SR_2>$, …, and $X_N = <V_N, SR_N>$ in a resource space $RS = \{X_1, X_2, \ldots, X_N\}$ according to a set of given representations $m_1, \ldots, m_N$ of the expected classes (i.e., find the classes $n_1, \ldots, n_N$ of the multiple dimensions according to the given $m_1, \ldots, m_N$ and n_1 is a class of X_1, n_2 is a class of X_2, …, n_N is a class of X_N)

and then returns $\bigcap_{i=1}^{N} R(n_i)$, where $n_i \in V_i$, $n_i = m_i$ and $i \in [1, N]$. For example, the statement for retrieving the *RS* = <*network*, *model*> is represented as: SELECT RESOURCE FROM *RS* WHERE *network* = "neural network" AND *model* = "segmentation model", which returns 4 common projects of the class "neural network" of *network* dimension that contains 400 projects and the class "segmentation model" of the *model* dimension that contains 14 projects.

4. SHOW COORDINATE OF *d* ON *RS* AT $\{X_1, X_2, \ldots, X_N\}$. It shows the coordinate $(p[X_1], p[X_2], \ldots, p[X_N])$ of a resource *d* (the representation of *d* includes the name and the path of the directory of the resource, i.e., *path*/*name*) in the resource set *D* accessed by resource space $RS = \{X_1, X_2, \ldots, X_N\}$, i.e., $d \in D = \bigcup_{i=1}^{N} R(X_i) = \{d_1, d_2, \ldots, d_{|D|}\}$, $R(X_i)$ is the set of resources that can be accessed from the dimension X_i. The $p[X_i]$ is a path started from the root of X_i (denoted as $<n_0, n_1, \ldots, n_{|p[Xi]|}>$) representing the projection of a resource *d* on X_i, and $d \in R(n)$ if and only if $n \in p[X_i]$. For the *RS* = <*network*, *model*> extracted from the 1000 GitHub README texts $D = \{d_1, d_2, \ldots, d_{1000}\}$, the statement for showing the coordinate of a text such as $d_{423} \in D$ in *RS* = <*network*, *model*> is represented as: SHOW COORDINATE OF d_{423} ON *RS* AT {*network*, *model*} (d_{423} is the *path*/*name* of the README text file of the project "Scene Representation Networks" at https://github.com/vsitzmann/scene-representation-networks). It returns the coordinate of d_{423} in *RS*: (<"network", "scene representation network">, <"model", "core model", "core SRNs model">) as the paths framed by the red dotted line shown in Figure E.1. It shows that d_{423} can be accessed from both the *network* dimension and the *model* dimension. This statement supports users or application systems to easily obtain the coordinate of a resource in the resource space to clearly shows the classes and subclass relations on multiple dimensions for retrieving resources and analyzing resources efficiently.

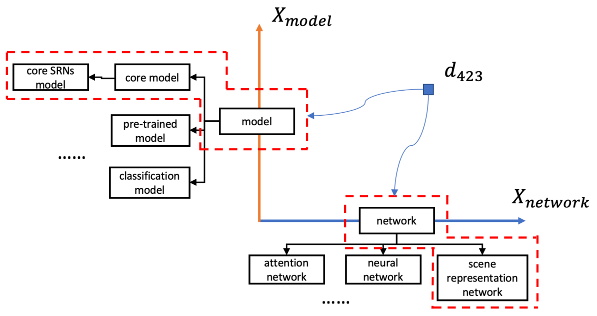

Figure E.1. Projections of resource d_{423} onto two dimensions.

As the resources in the resource space are classified with the process of extracting class trees, retrieving a resource of a class carries out by searching the class on the class trees with an average time complexity of $O(\log n)$ and then getting the resource with a time complexity of $O(1)$.

E.3 Add Resources

The add operation enables users or application systems to use the following SQL-like language to add a new text resource *d* (*d* is the *path*/*name* of the resource) to a resource space $RS = \{X_1, X_2, \ldots, X_N\}$, represented as: INSERT *d* INTO *RS*.

For the new text *d* and the resource space *RS*, and the add operation proceeds as follows, for each $X_i = <V_i, SR_i>$ and *d*:

1. Extract the class tree with the root of X_i from the content of *d*, denoted as $T' = <V', SR'>$. $T' = <\emptyset, \emptyset>$ if the root node of X_i doesn't appear in the content of *d*.
2. For each class *n* in V' but not in V_i, add it to V_i (i.e., $V_i = V_i \cup \{n\}$) and add the corresponding subclass relations to SR_i. If there exists a subclass relation (n_i, n_j) in SR_i so that *n* is the subclass of n_i and n_j is the subclass of *n*, then delete the subclass relation (n_i, n_j) from SR_i, add the subclass relations (n_i, n) and (n, n_j) to SR_i. Otherwise, for each n' in V_i, if *n* is the subclass of n', then add the subclass relation (n', n) to SR_i.
3. For each class *n* in V_i, calculate the $B(d, n)$, get the projection of the resource *d* on X_i, denoted as $p[X_i] = <n_0, n_1, \ldots, n_{|p[Xi]|}>$, where $n_{i+1} = \underset{n, n \in GetSubclass(n_i)}{\arg\max} B(d, n)$, $i \in [0, |p[Xi]|\text{-}1]$ and $GetSubclass(n_i)$ returns the set of subclasses of the class n_i.
4. For each class *n* in $p[X_i]$, add *d* to the set of the texts represented by class *n*, i.e., $R(n) = R(n) \cup \{d\}$.

For example, the statement for adding a resource to a 2-dimensional resource space *RS* = <*network*, *model*> is represented as INSERT d_{1001} INTO *RS* (d_{1001} is the *path*/*name* of the README text file of the project "Cedgan Interpolation" at https://github.com/dizhu-gis/cedgan-interpolation), where d_{1001} is a text that has not been managed by *RS*. A part of the contents of d_{1001} is shown in Figure E.2 (a). A part structure of *RS* = {*network*, *model*} is shown in Figure E.2 (b). Figure E.2 (c) shows the abstraction trees extracted from d_{1001} for both the *network* and the *model* dimensions of *RS*. The *RS* after the add operation is shown in Figure E.2 (d) (i.e., a new class "conditional generative adversarial neural network" is added to the *network* dimension, and no new class is added to *model* dimension). The part framed by the red dotted line is the projection of d_{1001} on two dimensions.

The add operation can automatically add new classes and subclass relations according to the added text and extract the classes to which the added resource belongs on each dimension and their abstractions.

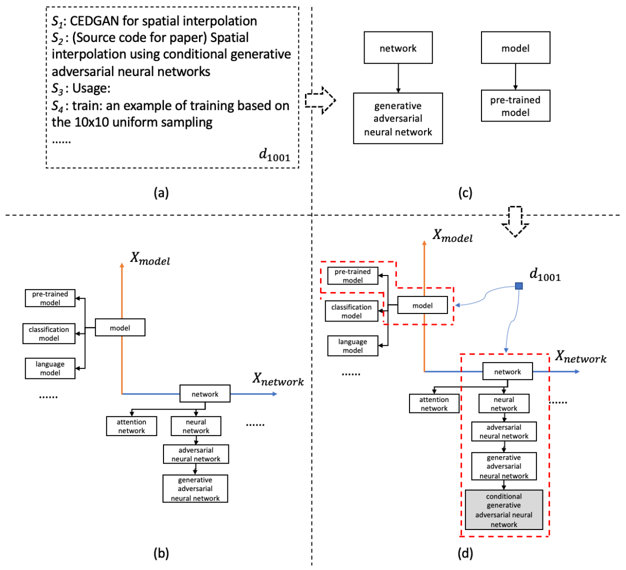
S_1: CEDGAN for spatial interpolation
S_2: (Source code for paper) Spatial interpolation using conditional generative adversarial neural networks
S_3: Usage:
S_4: train: an example of training based on the 10x10 uniform sampling
......
d_{1001}
(a)
network
generative adversarial neural network
model
pre-trained model
(c)
X_{model}
pre-trained model
classification model
language model
......
model
network
$X_{network}$
attention network
neural network
adversarial neural network
generative adversarial neural network
......
(b)
X_{model}
d_{1001}
pre-trained model
classification model
language model
......
model
network
$X_{network}$
attention network
neural network
adversarial neural network
generative adversarial neural network
conditional generative adversarial neural network
......
(d)

Figure E.2. Adding resource d_{1001} to *RS*.

E.4 Delete Resources

The delete operation enables users or application systems to use the following SQL-like language to delete a resource d (the *path*/*name* of the resource) from the resource space $RS = \{X_1, X_2, \ldots, X_N\}$, represented as DELETE d FROM *RS*.

For a text d (including path/name) and the resource space $RS = \{X_1, X_2, \ldots, X_N\}$, the operation for deleting d takes the following steps:

1. Show the coordinate $(p[X_1], p[X_2], \ldots, p[X_N])$ of resource d in *RS*, where $p[X_i]$ is a path started from the root of X_i (denoted as $<n_0, n_1, \ldots, n_{|p[X_i]|}>$).
2. For each dimension X_i and each class n in the $p[X_i]$, delete d from $R(n)$, i.e., $R(n) = R(n) - \{d\}$.
3. Delete d from the file system.

For example, the statement for deleting a resource from the 2-dimensional resource space $RS = <$*network*, *model*$>$ is represented as DELETE d_{467} FROM *RS* (the project "FFN" at https://github.com/google/ffn). It deletes a text d_{467} from the $RS = \{$*network*, *model*$\}$, the process is depicted in Figure E.3.

Users or application systems can also use the following statement to delete the resources from the resource space $RS = \{X_1, X_2, \ldots, X_N\}$ by specifying the classes of the dimensions.

1. DELETE RESOURCE FROM *RS* WHERE $X_i = m$. It deletes the resources accessed by the expected class n of the given representation m from the dimension $X_i = <V_i, SR_i>$ of the *RS*, i.e., deletes the resources in $R(n)$ where $n \in V_i$, $n = m$.
2. DELETE RESOURCE FROM *RS* WHERE $X_i = (m_1, m_2, \ldots, m_p)$. It deletes the resources in classes $n_1, \ldots, n_P$ according to the given representations $m_1, \ldots, m_P$ from the dimension $X_i = <V_i, SR_i>$ of the *RS*, i.e., delete the resources in $\bigcup_{j=1}^{P} R(n_j)$ where $n_j \in V_i$, $n_j = m_j$ and $j \in [1, P]$.
3. DELETE RESOURCE FROM *RS* WHERE $X_1 = m_1$ AND $X_2 = m_2$ AND ... AND $X_N = m_N$. It deletes the resources accessed by the expected classes $n_1, \ldots, n_N$ according to the given representations $m_1, \ldots, m_P$ from the dimensions $X_1 = <V_1, SR_1>, \ldots, X_N = <V_N, SR_N>$ of the *RS*, i.e., delete the resources in $\bigcap_{i=1}^{N} R(n_i)$ where $n_i \in V_i$, $n_i = m_i$ and $i \in [1, N]$.

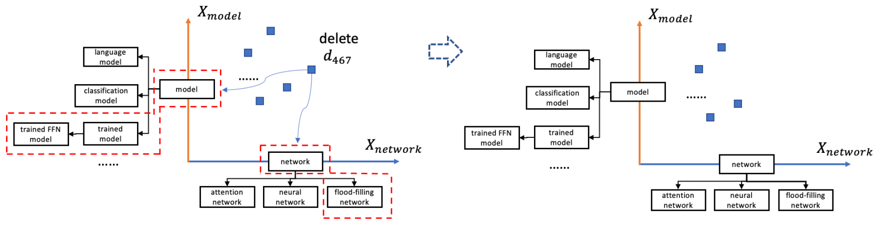

Figure E.3. Delete resource d_{467} from *RS*.

E.5 Merge Space

The merge operation enables users or application systems to use the following statement to merge two spaces $RS = \{X_1, X_2, \ldots, X_N\}$ and $RS' = \{X'_1, X'_2, \ldots, X'_N\}$ by merging their dimensions representing the same class (i.e., with the same root nodes), where $RS^* = \{X_1 \cup X'_1, X_2 \cup X'_2, \ldots, X_N \cup X'_N\}$, X_i and X'_i represent the same class, and $X_i \cup X'_i$ represents the merge of the dimensions, $i \in [1, N]$: MERGE *RS* AND *RS'* INTO RS^*.

Two dimensions $X_i = <V_i, SR_i>$ and $X'_i = <V'_i, SR'_i>$ of two resource spaces are merged into $X^*_i = <V^*_i, SR^*_i>$ by two steps: (1) Merge node sets (i.e., $V_i \cup V'_i$) by combining the same nodes (i.e., nouns/noun phrases), denoted as V^*_i; and (2) Merge subclass relations (i.e., $SR_i \cup SR'_i$) by removing the redundant subclass relations and duplicating the child tree with two or more parent nodes to construct trees, denoted as SR^*_i.

To verify the merge operation in the GitHub application, the README dataset *D* (1000 texts) is split into two sub-datasets *D'* (468 texts) and *D"* (532 texts). Two dimensions (*network* and *model*) can be extracted from *D'* and *D"* respectively, denoted as *RS'*= {*network'*, *model'*} and *RS"* = {*network"*, *model"*}, where the *network'* is a class tree consisting of 821 classes and 1088 subclass relations between the classes, the *model'* is a class tree consisting of 875 classes and 1080 subclass relations, the *network"* is a class tree consisting of 712 classes and 922 subclass relations, and the *model"* is a class tree consisting of 670 classes and 822 subclass relations.

The statements for merging the two spaces *RS'* and *RS"* can be represented as MERGE RS' AND RS'' INTO RS^*= {*network*, model**}, where *network** is the class tree consisting of 1428 classes and 1885 subclass relations obtained by merging dimensions *network'* and *network"*, and *model** is the class tree consisting of 1431 classes and 1722 subclass relations obtained by merging dimension *model'* and dimension *model"*.

The number of nodes and subclass relations before and after the merge operations are shown in Table E.1. We can see that *network* and *network** have the same 1428 classes and 1885 subclass relations, and *model* and *model** have the same 1431 classes and 1722 subclass relations. This verifies the correctness of the merge operation. The dimensions in the resource spaces discovered by different datasets may have duplicate classes and subclass relations. The merge operation is to merge the same classes and subclass relations to generate a concise resource space without losing semantics.

The merge operation provides the basis for accessing multiple resource spaces and distributed construction of resource spaces on large-scale dataset.

TABLE E.1. THE NUMBER OF NODES AND SUBCLASS RELATIONS FOR THE ORIGINAL AND MERGED DIMENSIONS.

Resource Space	**Dimensions**	**Number of Nodes**	**Number of Subclass Relations**
RS'	*network'*	821	1088
	model'	875	1080
RS"	*network"*	712	922
	model"	670	822
*RS**	*network**	1428	1885
	*model**	1431	1722
RS	*network*	1428	1885
	model	1431	1722