Dutta, Biswanath. and Das, Puranjani. (2023). Semantic Annotator for Knowledge Graph Exploration: Pattern-Based NLP Technique. Journal of Information and Knowledge (Formerly SRELS Journal of Information Management) (accepted)

Semantic Annotator for knowledge Graph Exploration: pattern-based NLP technique

ABSTRACT

Semantic Annotator for knowledge Graph Exploration, abbreviated as SAGE is a “Thing” annotation system. Here, “Thing” refers to any concept, named individuals (aka entities), entity relations, and attributes. The system is primarily built based on the idea of “string to thing” where the “string” is any given text (e.g., abstract of an article) as input by the user. For annotation, the system utilises the knowledge graph(s). SAGE can be used by anyone for annotating Things and for their exploitation on the Web. The annotation of things is done through exact and partial matches. For the exact matches, the system makes explicit the name of the knowledge graphs it is sourced from. It also shows the type hierarchies for the matched named entities. In the current work, we describe the SAGE annotation system, designed on pattern-based NLP techniques, along with its features and various usage, and the experimental results.

KEYWORDS

Semantic annotation, thing annotation, entity annotation, knowledge graph exploration, thing spotting, NLP, semantic annotation platform, automated annotation, application

1. INTRODUCTION

A knowledge graph (KG) refers to a representation of an intelligent web of data that is informed by an ontology [1]. It forms a knowledge base storing the interlinked data descriptions of entities (aka objects, both abstract and physical) and concepts (aka entity types) following the graph-structured data model. KGs can be viewed as data that can be queried, as a graph that can be analyzed, and as a knowledge base from which new facts can be derived [2, 21]. There have been many KGs built across different domains both in academia and industry, for instance, CODO KG [3], DTO [4], BAO [5], DRON [6], the Google Knowledge Graph [7], DBpedia KG [8], Wikidata [9]. KG is used in information search and retrieval, recommendation, chatbot, knowledge management, intelligent system design, data governance, etc. Several real-world applications are built centering KGs. For instance, question-answer systems [10], like WolframAlpha [11], semantic search and retrieval (e.g., Google search engine’s results ‘infobox’ that appears next to the search results), information integration (Gupta et al., 2012), data visualisation and exploration (e.g., CovidGraph [13]), and automatic annotation (e.g., DBpedia Spotlight).

The current work focuses on automatic annotation. It designs and develops an automated semantic annotation system, namely SAGE, a Semantic Annotator for Exploration of “things” in a Knowledge Graph(s). The “things” refers to entities, entity relationships, attributes, and entity types (aka concepts). Here, annotation refers to a process of spotting, linking, and extracting information about the “things” in the input text from the KG. This work is primarily inspired by DBpedia Spotlight [14] and other general-purpose text annotation systems, such as BRAT [15], Doccano [16], etc. DBpedia Spotlight is a tool used for semantic annotation based on DBpedia KG. It is limited to named entity recognition and does not annotate concepts, entity relations, or attributes. FICLONE is another tool that improves upon DBpedia Spotlight, but its performance and accuracy are dependent on the annotations from DBpedia Spotlight [17]. The other most recent annotation systems are, for example, MTab4D [18] and LinkingPark [19]. These studies are limited to the annotation of tabular data. MTab4D is a semantic annotation system for tabular data matching table elements with KGs, such as DBpedia KG. It combines multiple matching signals from table elements, such as table columns to entity types, columns to properties, and cells to entities. The study addresses schema heterogeneity, data ambiguity, and noisiness. LinkingPark is an automatic semantic table interpretation system. It matches tabular elements to KGs. LinkingPark is a stand-alone system designed as a modular framework with the following modules: an entity linking module, a property linking module, a type inference module, and a knowledge graph module.

SAGE is a standalone desktop application for semantic annotation. It follows the idea of "strings to things." The system annotates the KG things that exist in the input text as strings. In other words, it identifies and labels strings in an input text with things (the elements that are defined with URIs) found in the KG. In a KG, each thing has a URI. SAGE binds the URIs of the things in the KG to the phrases present in the text. So, the thing from the user-given text is annotated, linked, negotiated, and explored on the Web. The advantage of SAGE over the other existing annotations systems are many. They are: it is not restricted to any specific KG, users can choose their KG as per the domain needs and annotation tasks; the annotation is supported for entities, relations, attributes, and concepts; allows the annotation based on multiple KGs at a time; provides user-friendly GUI to add input text and to upload/ delete KGs for the annotation tasks; support the upload of input text in multiple formats, such as .txt, .doc, .pdf, .rtf; supports the information retrieval about annotated things from the KG without writing a query; annotations are made following the exact match and partial match approach; spots the missing terms (including the entities) from the KG but available in the input text. Hence, SAGE can be referred to as a single platform for the exploration of KG in multiple ways.

SAGE has a wide range of uses. For instance: (1) it can be utilized as a library tool to enhance the users' reading experiences. For instance, it can be plugged into digital library systems, abstracting databases, etc. This will enable readers to unearth things and their related facts in the text from the KG while reading the documents; (2) in a similar fashion, for example, the doctors and laboratory assistants can run their notes and transcription through SAGE to explore things (e.g., medical terms, diseases, drugs, viruses) that they are not familiar with; (3) the ontologists can use SAGE as a platform to identify the gaps in their ontology against the domain literature; (4) can be utilized for comparing the coverage of multiple KGs.

The current work is built using pattern-based NLP techniques. Pattern-based NLP, a subfield of NLP (Natural Language Processing), approaches are based on linguistic or lexicographic knowledge, as well as existing human knowledge regarding the contents of the text that is to be processed. The knowledge is mined by using predefined or discovered patterns [20]. The process involves using various techniques such as regular expression and string manipulations to identify specific patterns in the text data. The extracted patterns can then be used for various NLP tasks, such as text classification, named entity recognition, sentiment analysis, and more. The goal of pattern-based NLP is to automate the extraction of useful information from unstructured text data and make it easier to analyze and understand. The evaluation of SAGE reveals impressive results.

The main contributions of this work are: (1) discusses the design and development of a desktop application called SAGE for "thing" annotation from KGs; (2) the application provides GUI for uploading knowledge graphs, input text for annotation, and exploration of things from the KGs with ease; (3) the application reduces the technological complexity of exploration of KGs which usually demand the user's technological expertise.

The rest of the paper is organized as follows: section 2 discusses the SAGE features followed by the SAGE architecture in section 3 and SAGE design approaches in section 4. Section 5 describes the results and evaluation of the system. Section 6 concludes the paper by mentioning the limitations and future directions of the current work.

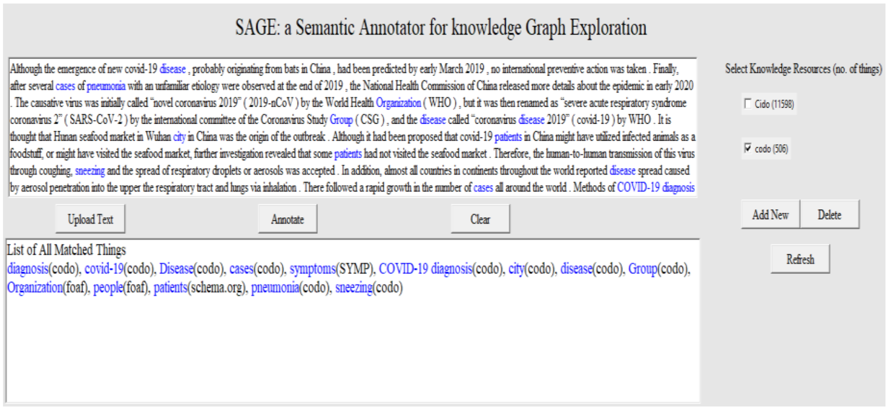
2. SAGE FEATURES

We briefly discuss here the following primary features of the SAGE semantic annotation system: (1) exploration of KGs without running a query (2) identification of missing things in a KG (3) estimation of coverage of KGs (4) KG statistics. The SAGE features are detailed in [21].

Exploration of KG

SAGE enables its users (e.g., an ontologist, a domain expert, a KG explorer, a reader) to search and analyze KGs in relation to literature (e.g., scholarly literature). By simply copying and pasting a text excerpt, or by uploading a text document from the user's field of interest, the tool will annotate any term in the text associated with a thing in the KG and link it to its URI, allowing the user to easily explore them from their web source. With SAGE, users don't need to have any prior knowledge of query building or coding, making it accessible to anyone. Additionally, SAGE provides a user-friendly interface for uploading one's own KG in RDF/XML format, enabling users to search the graph of their choice. Multiple KGs can also be selected for the exploration of literature. As mentioned above, SAGE supports two kinds of annotations based on the level of matching. These are based on the exact matches and partial matches as described below.

a) *Exact match*- SAGE identifies and annotates the exact matching terms in a text excerpt, linking them to their URIs. Upon being clicked the description of the term on the Web will open up. For example, terms such as *pneumonia* and *diseases* found in a paper [22] were annotated against CODO [27], a COVID-19 ontology. The exact matches are presented in two ways (in-text annotation and annotated list of terms) to provide a comprehensive view of the annotations as described below-
1. The text may not have enough space to display the source vocabularies (aka ontology) for the IRIs that the terms are annotated with. So, all terms with exact matches in the selected vocabularies by the user are presented in an annotated list with the source vocabularies indicated next to them in circular parenthesis “()”, as seen in Figure 1.
2. It is not possible to show all the exact matches within the text itself. Multiword terms, such as "Covid-19 diagnosis," are given priority in the in-text annotations, while single words are listed in the annotated list of exact matches. For example, "Covid-19 diagnosis" would be annotated within the text, while "covid-19" and "diagnosis" would appear in the annotated list as seen in Figure 1.
3. When a term has multiple exact matches in various KGs (when more than one KG is selected for annotation), it becomes complex to annotate the term within the text using multiple IRIs. To overcome this, one of the IRIs is annotated in the text while both of them can be found in the annotated list of terms. This allows for a clear distinction of multiple matches for the same term from different KGs. If a term, like "covid-19", is found in two Covid-19 related KGs, only one IRI will be annotated in the text, and then both will be listed in the annotated terms list.


SAGE: a Semantic Annotator for knowledge Graph Exploration
Select Knowledge Resources (no. of things)
Upload Text
Annotate
Clear
Add New
Delete
Refresh
Although the emergence of new covid-19 disease , probably originating from bats in China , had been predicted by early March 2019 , no international preventive action was taken . Finally, after several cases of pneumonia with an unfamiliar etiology were observed at the end of 2019 , the National Health Commission of China released more details about the epidemic in early 2020 . The causative virus was initially called "novel coronavirus 2019" (2019-nCoV) by the World Health Organization (WHO) , but it was then renamed as "severe acute respiratory syndrome coronavirus 2" (SARS-CoV-2) by the international committee of the Coronavirus Study Group (CSG) , and the disease called "coronavirus disease 2019" (covid-19) by WHO . It is thought that Hunan seafood market in Wuhan city in China was the origin of the outbreak . Although it had been proposed that covid-19 patients in China might have utilized infected animals as a foodstuff, or might have visited the seafood market, further investigation revealed that some patients had not visited the seafood market . Therefore, the human-to-human transmission of this virus through coughing, sneezing and the spread of respiratory droplets or aerosols was accepted . In addition, almost all countries in continents throughout the world reported disease spread caused by aerosol penetration into the upper the respiratory tract and lungs via inhalation . There followed a rapid growth in the number of cases all around the world . Methods of COVID-19 diagnosis

[ ] Cido (11598)
[✓] codo (506)

List of All Matched Things
diagnosis(codo), covid-19(codo), Disease(codo), cases(codo), symptoms(SYMP), COVID-19 diagnosis(codo), city(codo), disease(codo), Group(codo), Organization(foaf), people(foaf), patients(schema.org), pneumonia(codo), sneezing(codo)

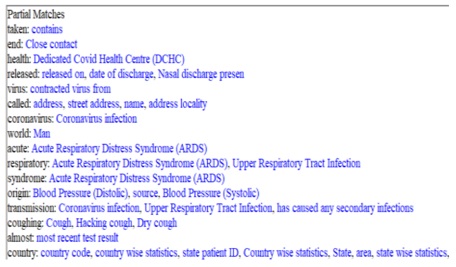
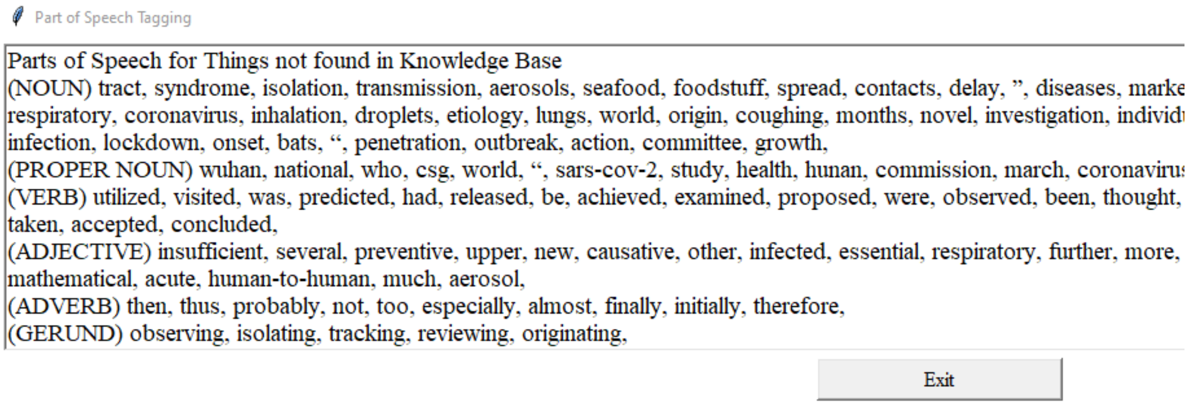

Partial Matches
taken: contains
end: Close contact
health: Dedicated Covid Health Centre (DCHC)
released: released on, date of discharge, Nasal discharge presen
virus: contracted virus from
called: address, street address, name, address locality
coronavirus: Coronavirus infection
world: Man
acute: Acute Respiratory Distress Syndrome (ARDS)
respiratory: Acute Respiratory Distress Syndrome (ARDS), Upper Respiratory Tract Infection
syndrome: Acute Respiratory Distress Syndrome (ARDS)
origin: Blood Pressure (Distolic), source, Blood Pressure (Systolic)
transmission: Coronavirus infection, Upper Respiratory Tract Infection, has caused any secondary infections
coughing: Cough, Hacking cough, Dry cough
almost: most recent test result
country: country code, country wise statistics, state patient ID, Country wise statistics, State, area, state wise statistics,

Figure 1. Main window, with exact matches **Figure 2. Partial matches**

b) *Partial match*- SAGE also presents a list of partially matched terms, offering users contextually relevant resources. These terms do not exist independently in a KG, but as parts of other terms. For example, we can see words like ‘transmission’, ‘released’, etc. in Figure 1, which are not being highlighted or annotated. Upon asking SAGE to show the Partial Matches found from CODO ontology, Figure 2 opens up. In the figure, the words ‘transmission’ and ‘released’ extracts the things like ‘coronavirus infection’ and ‘released on’ respectively, which are annotated. This shows that the words like ‘transmission’ and ‘released’ exist as a part of the words mentioned above. It can be noticed that the word ‘transmission’ does not exist in ‘coronavirus infection’, and neither does the word ‘released’ occur in ‘date of discharge’. This is because 'infection' is a synonym for 'transmission' and 'discharge' is a synonym for 'released'. The approach is elaborated in section 4, algorithm 4.

Identification of missing things in a KG

As can be seen from Figure 1, the things like *epidemic*, *etiology* remain unannotated. This means that the terms "epidemic" and "etiology" were not found in CODO KG. These terms may still be relevant to the subject and could be valuable additions to the KG/ ontology. SAGE spots such missing terms and lists them along with their parts of speech (POS). This feature is useful for maintaining ontologies/ KGs and ensuring that all relevant terms are included in them. For implementing this feature in SAGE, the Stanford POS tagger is used from the NLTK library of Python. Originally, Stanford POS tagger categorised the terms in the given string into 36 POS categories, for example, CC, VBG, VBZ, NNP, NNPS, TO, etc. [23]. However, SAGE drops some of the categories, especially those that are not useful from the KG perspective, for example, DT, TO, WRB, and CC. It further analyses and reorganizes the rest of the categories (e.g., NN, NNP, VBG, JJ, RB, etc.) into six: Nouns, Proper Nouns, Verbs, Adjectives, Gerunds, and Adverbs. For example, NN (Noun, singular or mass) and NNS (Noun, plural) are categorised as Noun, NNP (Proper Noun, singular) and, NNPS (Proper Noun, plural) are categorised as Proper Noun, and so forth. SAGE displays the results as a dictionary structure where the POS is the key, and the words exhibiting the mentioned POS are listed as the value of the key as shown in Figure 3.



Parts of Speech for Things not found in Knowledge Base
(NOUN) tract, syndrome, isolation, transmission, aerosols, seafood, foodstuff, spread, contacts, delay, ", diseases, marke
respiratory, coronavirus, inhalation, droplets, etiology, lungs, world, origin, coughing, months, novel, investigation, individ
infection, lockdown, onset, bats, ", penetration, outbreak, action, committee, growth,
(PROPER NOUN) wuhan, national, who, csg, world, ", sars-cov-2, study, health, hunan, commission, march, coronaviru
(VERB) utilized, visited, was, predicted, had, released, be, achieved, examined, proposed, were, observed, been, thought,
taken, accepted, concluded,
(ADJECTIVE) insufficient, several, preventive, upper, new, causative, other, infected, essential, respiratory, further, more,
mathematical, acute, human-to-human, much, aerosol,
(ADVERB) then, thus, probably, not, too, especially, almost, finally, initially, therefore,
(GERUND) observing, isolating, tracking, reviewing, originating,

Figure 3. Depicts unmatched terms with their POS

Tree view of entity types

SAGE shows the type information for the exactly matched entities in a tree structure from the KGs. For example, *COVID-19* is an entity (a named entity) that was found in the exact match from CODO as shown in Figure 1. Figure 4(a) shows the type information of "COVID-19" in the hierarchy indicating that it is a type of "Coronavirus infection", which falls under the supper class "Disease" in CODO. Similarly, Figure 4(b) shows the type information of "COVID-19" and the class hierarchy when the CIDO is considered. This feature of SAGE lets the user explore more about the entities found in the input text, such as their class type and their entire hierarchy in an ontology/ KG. This also helps in comparing and analysing the granularity and specificity of ontologies at their hierarchy level.

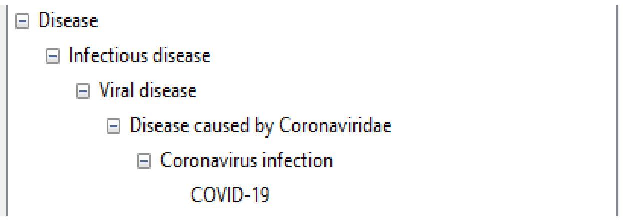


Figure 4a. Exact matched entity types in CODO

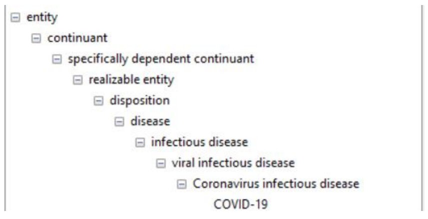
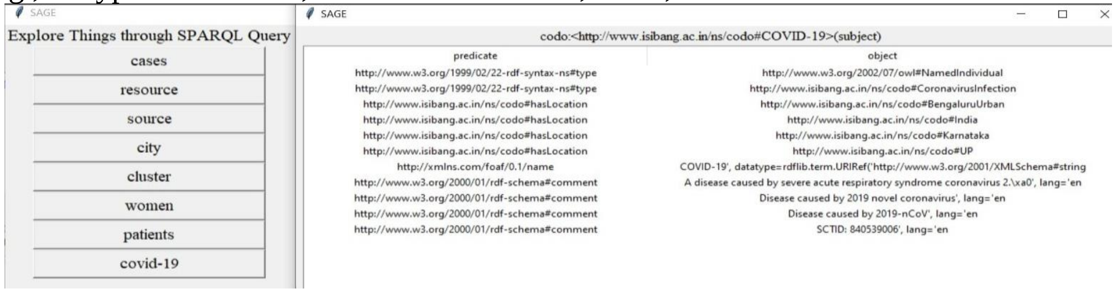


Figure 4b. Exact matched entity types in CIDO

Predefined query

SPARQL SELECT queries are run on exact matches, which returns all the tuples with the exact matched things. The query syntax is programmed and gets executed upon clicking the matched terms. The user does not have to write the query. Figure 5 shows the query results from the KG against an exactly matched thing "COVID-19." As can be seen from the figure, the system retrieves its related data (as triples) available in CODO, e.g., its type information, location information, name, etc.



codo:<http://www.isibang.ac.in/ns/codo#COVID-19>(subject)

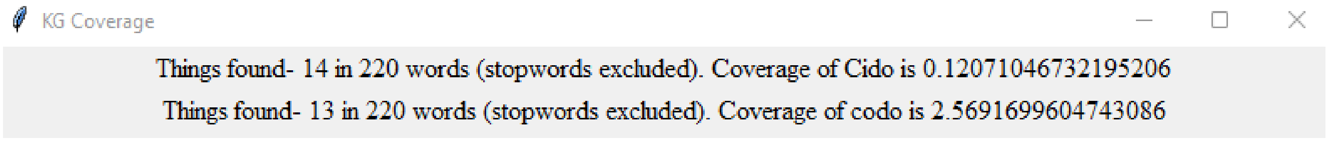
predicate	object
http://www.w3.org/1999/02/22-rdf-syntax-ns#type	http://www.w3.org/2002/07/owl#NamedIndividual
http://www.w3.org/1999/02/22-rdf-syntax-ns#type	http://www.isibang.ac.in/ns/codo#CoronavirusInfection
http://www.isibang.ac.in/ns/codo#hasLocation	http://www.isibang.ac.in/ns/codo#BengaluruUrban
http://www.isibang.ac.in/ns/codo#hasLocation	http://www.isibang.ac.in/ns/codo#India
http://www.isibang.ac.in/ns/codo#hasLocation	http://www.isibang.ac.in/ns/codo#Karnataka
http://www.isibang.ac.in/ns/codo#hasLocation	http://www.isibang.ac.in/ns/codo#UP
http://xmlns.com/foaf/0.1/name	COVID-19', datatype=rdflib.term.URIRef('http://www.w3.org/2001/XMLSchema#string
http://www.w3.org/2000/01/rdf-schema#comment	A disease caused by severe acute respiratory syndrome coronavirus 2.\xa0', lang='en
http://www.w3.org/2000/01/rdf-schema#comment	Disease caused by 2019 novel coronavirus', lang='en
http://www.w3.org/2000/01/rdf-schema#comment	Disease caused by 2019-nCoV', lang='en
http://www.w3.org/2000/01/rdf-schema#comment	SCTID: 840539006', lang='en

Figure 5. The list of exact matches and a snippet of the query results

Estimation of coverage of KG

The coverage calculation formula in SAGE is used to determine the percentage of exact matches found in the uploaded text against a selected KG. This feature enables ontology engineers to visualize and compare the coverage of multiple ontologies, KGs, etc. within a domain. The coverage of each KG is represented in terms of the exact matches found from the text after dropping stopwords. For example, as indicated in Figure 6, the exact match things found from the text in Figure 1, against CODO is 2.57% and CIDO is 0.12%. Equation 1 provides the coverage calculation formula.

$$coverage = \frac{no.of\ exactly\ matched\ things\ in\ text\ from\ KG}{total\ number\ of\ things\ in\ KG} \times 100 \ldots\ldots\ldots\ldots\ldots\ldots \text{(Equation 1)}$$



Things found- 14 in 220 words (stopwords excluded). Coverage of Cido is 0.12071046732195206

Things found- 13 in 220 words (stopwords excluded). Coverage of codo is 2.5691699604743086

Figure 6. Coverages of two KGs against a piece of text, with 220 words after excluding stopwords

KG statistics

As mentioned above, the user can upload the KG of their choice using the GUI provided by SAGE. The interface allows the user to choose the most relevant KGs for their annotation needs. The KGs uploaded are displayed in the right-hand panel of the main window in SAGE (see Figure 1). This panel shows the names of the uploaded KGs along with the number of things in each KG. This information helps the user to check the overall coverage of various KGs and compare them, which is important in evaluating the usefulness of the KGs for a particular task. For example, as depicted in Figure 1, CIDO [24] and CODO ontologies consist of 11598 and 856 things, respectively.

3. SAGE ARCHITECTURE

We describe the architecture of SAGE annotation system and its various features.

SAGE is designed following a 3-layer architecture as shown in Figure 7. As can be seen from the figure, the system takes two types of inputs: (1) text to be annotated and (2) KGs to be explored. The input text is a string from which the things will be identified and annotated based on the input KGs. Data Layer holds the knowledge base(s) in JSON format that is by extracting things and related information from the input KGs given in RDF/XML format. The business logic layer is responsible for receiving, processing, and responding to annotation tasks and other service requests (e.g., facts query, entity type, and type hierarchy, KG coverage). It finds exactly-matched things in the input text with things in the KG, retrieves URIs and type information, and runs SELECT SPARQL queries on them. It also handles partial matches (the contextual search [25]) and categorizes unmatched terms according to their parts of speech (POS). The system allows users to learn more about things by linking them to their URIs and descriptions on the Web. The presentation layer provides GUI to send information service requests and receive responses. It displays the results of text annotation, query results, and type information to the user. The SAGE software is freely available to download and use from https://tinyurl.com/yc8p5nm3. Note: SAGE is an open-source project, the source code will soon be released through the GitHub platform https://github.com/.

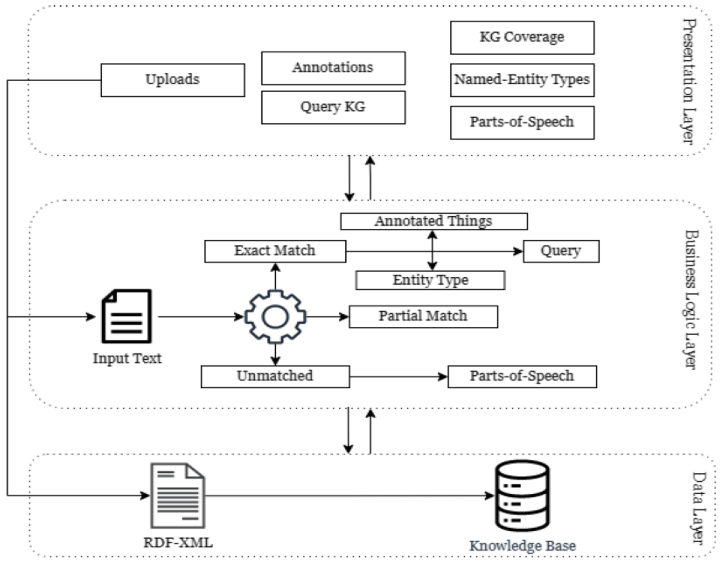


Figure 7. Software Architecture of SAGE

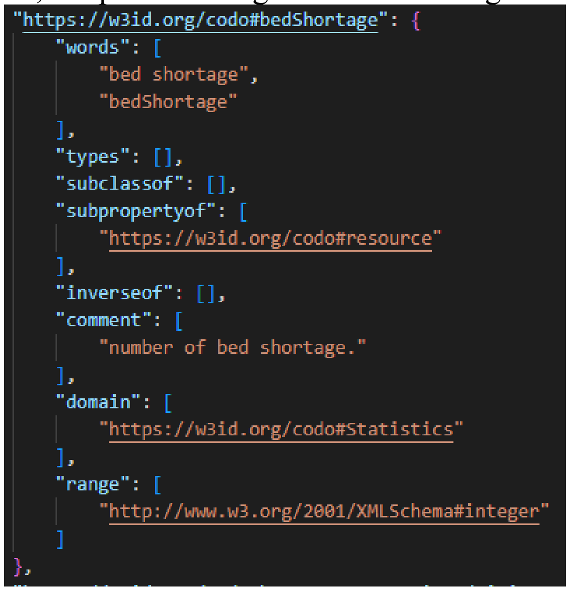
4. SAGE DESIGN APPROACH

We describe the primary algorithms that have been designed and developed to build the core functions of SAGE, such as exact match, partial match, and the creation of the class hierarchy described above. To explore a KG, SAGE first creates a knowledge base (KB) from the user-provided KG. The approach to KB creation is detailed below. We define a function called *string_found*, a string-matching function that is used throughout the SAGE pipeline is also described.

Algorithm 1: Knowledge Base creation

SAGE presently accepts the KGs in RDF/XML. SAGE's GUI allows users to upload or delete KGs directly from the interface (as depicted in Figure 1). Once the user uploads a KG, the system produces the KB as a JSON dictionary. For KB creation, we define a function called create_dictionary for base creation (see Algorithm 1). The KB is structured and formed with the following elements: *words, types, subclassof, subpropertyof, inverseof, comment, domain, and range*. Here, *Words* is a set consisting of synonymous terms used to describe a thing in a KG. The field *types* is used to contain the class of which an individual is an instance. *Subclassof* contains the parent class of classes in the KG. *Subpropertyof* contains the parent property of an object property or data property. *Inverseof* contains the inverse of a property described in the KG. *Comment* provides a description of the thing. *Domain* and *Range* are applicable for properties in the KG. A property links subject and object. The subjects and objects linked by a property are members of classes (named-entities). The class(es) in which the subject values of the property falls is the *Domain* class(es) and the class(es) in which the object values of the property falls is the *Range* class(es) [26].

The KB is created by extracting the data from a KG against each of the above-mentioned elements. The element *words* captures the data from various fields of a KG. Presently, SAGE supports the following data fields of a KG: rdfs:label, skos:altlabel, skos:prefLabel, skos:hiddenLabel NDF-RT:Synonym, (http://radlex.org/RID/) RID:Synonym, RID:Preferred_name, NDF-RT:MeSH_Name, NDF-RT:Display_Name (S8 of Algorithm 1). Here, "rdfs:," "skos:", "NDF-RT:" etc. represents the namespace prefixes for the URIs, "http://www.w3.org/2000/01/rdf-schema#", "http://www.w3.org/2004/02/skos/core#", and "http://radlex.org/RID/", respectively. The element "subClassOf" stores the data from rdfs:subClassOf a KG (S9 of Algorithm 1). The same is followed for capturing and storing data for other elements of a KB, as shown in S10-S15 of Algorithm 1. Figure 8 provides a glimpse of a KB. As can be seen from figure 8, in the KB, each thing is stored with its URI as its key. For example, in the figure, "https://w3id.org/codo#bedShortage" is a thing and it is stored as a key.



```
"https://w3id.org/codo#bedShortage": {
    "words": [
        "bed shortage",
        "bedShortage"
    ],
    "types": [],
    "subclassof": [],
    "subpropertyof": [
        "https://w3id.org/codo#resource"
    ],
    "inverseof": [],
    "comment": [
        "number of bed shortage."
    ],
    "domain": [
        "https://w3id.org/codo#Statistics"
    ],
    "range": [
        "http://www.w3.org/2001/XMLSchema#integer"
    ]
},
```

Figure 8. Part of a JSON file created from a knowledge graph

Algorithm 1
Require: A knowledge graph in RDF/XML
Ensure: A JSON file containing the data in the knowledge graph

```
S1: DEFINE FUNCTION create_dictionary(base_name):
S2:     parse the file into 'xml'
S3:     SET Prefixes= [Predefined Prefixes listed on the top]
S4:     SET template= {"words":[], "types"=[] "subclassof": [], "subpropertyof": [],
        "inverseof": [], "comment": [], "domain":[], "range":[]}
S5:     SET Things=[Make a set of all 'things' through tags (these tags contain 'ObjectProperty',
        'DatatypeProperty', 'Class', 'NamedIndividual', 'Description' )]
S6:     SET base={}
S7:     For things in Things do
S8:         find rdfs: label, skos: altlabel, NDF-RT:Synonym, NDF-RT MeSH_Name, NDF-RT:
            Display_Name, RID:Synonym, RID:Preferred_name, skos:PrefLabel, skos:hiddenLabel and store the
            values in template['words']
S9:         find rdfs: subClassOf and store the values in template['subclassof']
S10:        find rdfs: subPropertyOf and store the values in template['subpropertyof']
S11:        find owl: inverseOf and store the values in template['inverseof']
S12:        find rdfs: comment or skos:definition and store the values in template['comment']
S13:        find rdfs: domain and store the values in template['domain']
S14:        find rdfs: range and store the values in template['range']
S15:        find rdfs: type and store the values in template['type']
S16:        base[Thing URI]= template
```

```
S17:       make template empty
S18:    base["PREFIXES"]=Prefixes
S19:    Save base as JSON file
```

Algorithm 2: string_found function

The *string_found* function is created for the use of string matching throughout the SAGE pipeline. In the string_found function (algorithm 2), we take 2 input strings, namely string1 and, string2. string1 consists of the text we need to find in string2. This algorithm checks if the singular or plural version of the text in string1 is present in string2 (see steps S6 and S8). If the function finds the singular or plural version of the text in string1 present in string2, it returns the singular or plural of the text in string1 as the output, respectively. This can be seen in S7 and S9. If the function fails to find any match, it returns False as seen in S10.

Algorithm 2 string_found
Require: string1 and string2
Ensure: matched version of string1 in string2 or False

```
S1: DEFINE string_found(string1, string2)
S2:     if string1.lower() in string2.lower() then
S3:             l=string2.lower().index(string1.lower())
S4:             token=string2[l: l+len(string1)]
S5:             return token
S6:     else if plural(string1.lower()) in string2.lower() then
S7:             return plural(string1)
S8:     else if singular(string1.lower()) in string2.lower() then
S9:             return singular(string1)
S10:    return False
```

Algorithm 3: Exact match

Algorithm 3 describes the process of finding exact matches against a text corpus. For this, we match the terms in the set of words for each thing in the KB against the text corpus using the string_found function defined in Algorithm 2. Once a match is found, we retrieve the URI of the thing from the KB and bind it with the matched term in the input text. As mentioned above (section 2), in SAGE two types of annotations happen: in-text and as a list of annotated things. The things used for both are stored separately in two dictionaries with their related necessary information. This function also returns the remaining strings from the input text after stripping off the terms which found an exact match in the KB.

Algorithm 3 exact_matches
Require: input_string and base (json file(s))
Ensure: 2 dictionaries for matches Annotation_Matches, All_matches, and unmatched part of the input_string

```
S1: DEFINE exact_matches(string, base)
S2:     Vector=After dropping punctuation and stopwords from the string
S3:     SET Annotation_Matches={}
S4:     for j in base.keys() do
S5:             SET terms=base[j][1]
S6:             SET types=base[j][0]
```

```
S7:         for term in terms do
S8:             if STRING FOUND(term, Vector) then
S9:                 SET key=STRING FOUND(term, Vector)
S10:                Annotation_Matches[key]=(types,j)
S11:            SET All_Matches={}
S12:            for i, j in Annotation_Matches.keys() do
S13:                if i!=j then
S14:                    if STRING FOUND(i,j) then
S15:                        SET All_Matches[i]=Annotation_Matches[i]
S16:        if keys in Annotation_Matches.keys() and All_Matches.keys() then
S17:            DELETE Annotation_Matches[i]
S18.    for keys in All_Matches.keys() do
S19:            if keys in Vector then
S20:                Vector.drop(keys)
S21.    return Annotation_Matches, All_Matches, Vector
```

It is worth noting that our search process goes in the following way- terms in the KG(s) are searched for in the text. Searching could have been done from the other direction as well, i.e., terms in input text are searched in KG. We have opted for the former one, i.e., terms in the KG are searched for in the text because the terms in the ontology often happen to be several words in length (phrases). So, to proceed with the search from text to ontology, one has to create phrases of varying lengths and then perform the search. This increases the complexity of the program as explained below.
Say, there are *n* things in the given KG, and an *m* words-long text corpus is given to be annotated. Let us assume that these things do not have synonyms.

When we go from KG to text the search is executed *n* x *m* times. Now, when we go from the text to KG, first we create phrases. So, if there are *m* words, then we have *(m-1)* 2-word phrases, *(m-2)* 3-word phrases, *(m-3)* 4-word phrases, *(m-4)* 5-word phrases, *(m-5)* 6-word phrases, till we have 1 m-word phrase. Therefore, the number of terms will be,

$$m\times(m-1)\times(m-2)\times(m-3)\times(m-4)\times(m-5)\dots 1 = m!$$

Hence, the search will be conducted *n* x *m!* times.

$$n\times m \le n\times m! \qquad (1)$$

Thereby, to achieve a reduced complexity we perform the search from the KG(s) to the text corpus.
If the things in the KG(s) had *x* number of synonyms on average, the equation (1) would change to,

$$n\times x\times m \le n\times x\times m! \qquad (2)$$

Even in this case, the complexity is lesser if we go from KG to text for searching.

Algorithm 4: Partial match
Algorithm 4 describes the process of finding partial matches. Here, we find the existence of unmatched terms as part of ‘things’ in the knowledge base. The algorithm takes the Vector (a string of terms that do not find any exact match from the knowledge base) returned from Algorithm 3, as the input. We first split the input string into a list of words. For each word in the unmatched part of the input string, we extract its synonyms from WordNet [28] and create a list of terms to be matched against the KB (see S6 of Algorithm 4). We use synonyms of the unmatched words, along with the existing unmatched words, to increase the recall of the system. Sometimes, knowledge resources fail to accommodate all the terms associated with a ‘thing’, and the system misses out on spotting contextually relevant ‘things’. Adapting this approach of creating a list of synonyms for an unmatched term, has significantly increased the recall, as can be seen in Evaluation section 3. We reuse the string_found function in the partial_matches function too (see step S11).

Algorithm 4 partial_matches
Require: unmatched string and base (json file(s))
Ensure: *Partial* in the form of a dictionary

```
S1: DEFINE partial_matches(Vector, base)                    *Vector from Algorithm 3
S2:     SET Partial ={}
S3:     Vector=drop stopwords and punctuations
S4:     Vector= Vector.split()
S5:     for i in Vector do
S6:         list_of_synonyms=get_synonyms_from wordnet(i)
S7:         for k in list_of_synonyms:
S8:             for j IN base.keys() do
S9:                 SET terms=base[j][1]
S10:                for term in terms do
S11:                    if STRING FOUND(i, term) then
S12:                        if i in Partial.keys() then
S13:                            Partial [i].append([term, j])
S14:                        else
S15:                            Partial [i]=[[term, j]]
S16:                        break
S17:    return Partial
```

Algorithm 5: Entity tree view

For the named-entities found in the text from the KB, SAGE presents its type information. The system expands the class hierarchy tree of the type of the named-entity as can be seen in Figure 4. The Treeview widget of Tkinter is used for this purpose. Algorithm 5 describes the basic algorithm required to extract the hierarchy tree for a particular class. The defined function create_hierarchy, a recursive in nature (see step S3). The function is called again within itself in step S8 of algorithm 5. The function looks for the type information of the named-entity, which gives the URI of a class. The URI of the class is used to fetch its parent class's URI (listed under 'subclassof' in the KB). The function then again calls itself to fetch the parent class's URI of the previously obtained class. The dictionary hierarchy stores the classes and their various levels, where "1" is the named-entity, "2" type of named-entity (which is a class), "3" parent class of the class found in level 2, "3" parent class of the class found in level 3, and henceforth. The recursion continues till the 'subclassof' field of the class found in the previous step is found to be empty.

Algorithm 5 Create Hierarchy
Require: knowledge_base (JSON file), URI of the named_entity type whose class hierarchy is to be found
Ensure: *hierarchy* in the form of a dictionary which looks like this {1: named_entity 2: Type class 3: Parent of type class ……}

```
S1: hierarchy={1:named_entity, 2: named_entity_type}
S2: num=3
S3: DEFINE create_hierarchy(knowledge_base, named_entity_type_URI, num):
S4:     while knowledge_base[named_entity_type_URI][subclassof]!=[]:
S5:             parent_class_URI=knowledge_base[named_entity_type_URI][subclassof]
```

```
S6:                     hierarchy[num]=knowledge_base[parent_class_URI][words][0]
S7:             num=num+1
S8:             create_hierarchy(knowledge_base, parent_class_URI, num)
S9:     return hierarchy
```

Predefined query

The best way to explore a KG is through a SPARQL query. SAGE provides the user the facility to run SELECT queries on the exact matches found in the text. This feature is independent of the user's ability to write SPARQL queries and lets the non-experts explore the KG as well. The RDFLIB library of Python is used to write and run a query on the given graphs. When the exact matches are found, the system automatically retrieves the results, whether the "thing" is a class, property, or instance along with the information about the source KG. Table 1 provides the pre-defined query syntax that the system runs. Here, s stands for the subject, p for the predicate, and o for the object. When a class is found, the query in row 1 column 3 of table 1 is run, substituting the URI of the class in the given position. Properties (Data and Object) and instances also follow similarly, as shown in rows 2 and 3 of Table 1, respectively.

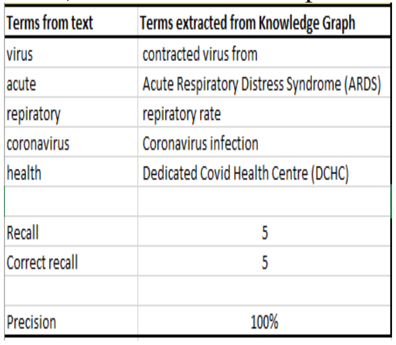
Sl. No.	Thing Category	Query Built
1	Class	SELECT DISTINCT ?s ?p WHERE {?s ?p <URI of thing>}
2	Property	SELECT DISTINCT ?s ?o WHERE {?s <URI of thing> ?o}
3	Instance	SELECT DISTINCT ?p ?o WHERE {<URI of thing> ?p ?o}

Table 1: Query syntax for things search

5. RESULTS AND EVALUATION

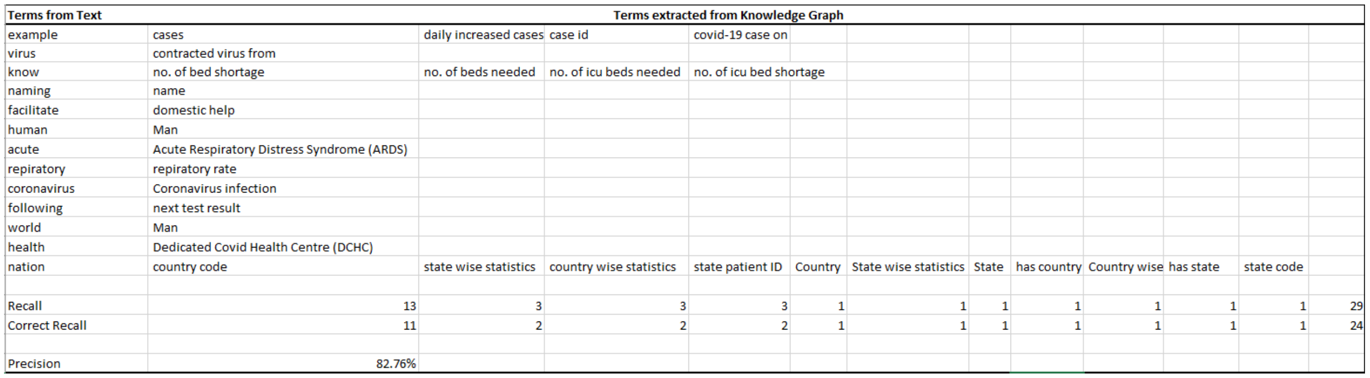
Here, we discuss the experimental results and the system evaluation.

A. *Precision and Recall of Partial Matches*: SAGE has compared the precision and recall of things found in partial matches with and without using WordNet as a resource. In figures 9 and 10, CODO ontology is used as the knowledge base and the text is taken from a COVID-19-related webpage of WHO. In Figure 9, SAGE found the things in bold as partial matches for the terms in the column on the left. 5 things were found; hence the recall is 5. When checked manually, all 5 of them were deemed contextually relevant. Therefore, the precision is calculated to be 100%. In Fig. 10, the recall rose to 29, with 24 contextually relevant recalls. Hence, the precision became 82.76%.



Terms from text	Terms extracted from Knowledge Graph
virus	contracted virus from
acute	Acute Respiratory Distress Syndrome (ARDS)
repiratory	repiratory rate
coronavirus	Coronavirus infection
health	Dedicated Covid Health Centre (DCHC)
Recall	5
Correct recall	5
Precision	100%

Figure 9. Partial Match results without using Wordnet



Terms from Text	Terms extracted from Knowledge Graph											
example	cases	daily increased cases	case id	covid-19 case on								
virus	contracted virus from											
know	no. of bed shortage	no. of beds needed	no. of icu beds needed	no. of icu bed shortage								
naming	name											
facilitate	domestic help											
human	Man											
acute	Acute Respiratory Distress Syndrome (ARDS)											
repiratory	repiratory rate											
coronavirus	Coronavirus infection											
following	next test result											
world	Man											
health	Dedicated Covid Health Centre (DCHC)											
nation	country code	state wise statistics	country wise statistics	state patient ID	Country	State wise statistics	State	has country	Country wise	has state	state code	
Recall	13	3	3	3	1	1	1	1	1	1	1	29
Correct Recall	11	2	2	2	1	1	1	1	1	1	1	24
Precision	82.76%											

Figure 10. Partial Match results after using WordNet

After trying this experiment with multiple text samples on the same KB, we found the average precision of SAGE's partial match (after using WordNet as an additional resource) is **83.18%**, whereas the recall has increased significantly in all cases (more than double).

B. *Feature comparison*: Table 2 compares the features of DBpedia Spotlight with SAGE, and highlights the advantages of SAGE over DBpedia Spotlight. It shows that SAGE is more customizable and domain-friendly compared to DBpedia Spotlight, which is more of a general-purpose annotator. The KB in SAGE can be customized, making it more suitable for domain-specific applications. SAGE also supports a wider range of features, such as the categorization of retrieved entities as object Properties and datatype Properties, which are not supported by DBpedia Spotlight. Additionally, SAGE enables ontology maintenance by suggesting possible missing terms, which is not possible with DBpedia Spotlight.

Features	**DBpedia Spotlight**	**SAGE**
Annotation (of exact things)	✓	✓
Retrieval (of similar 'things', not complete matches)	**X**	✓
Things Retrieved		
Object Properties	**X**	✓
Datatype Properties	**X**	✓
Classes	✓	✓
Named Individuals	✓	✓
Other Features		
Type information	✓	✓
Spotting Missing Terms	**X**	✓
Upload Text Corpus	✓	✓
Customized KB supported	**X**	✓
Predefined Query	**X**	✓
Coverage of KB	**X**	✓

Table 2: Comparison of features of DBpedia Spotlight with SAGE

C. *Response Time for primary annotation*: Table 3 displays the results of testing SAGE's performance with text excerpts of various lengths and KGs of various lengths. The results show that the response time of SAGE for exact match search and retrieval is directly proportional to the length of the text input. As the length of the text increases, the response time also increases. For example, in Table 3, we see this relationship demonstrated using the CIDO ontology and different lengths of text excerpts.

Ontology Name	**Total Things**	**Length of text before removing stopwords**	**Annotation Time (seconds)**
COVID-19 (CIDO)	11598	200	10.448144674301147
		500	17.92897057533264
		1000	29.03405261039734
		2000	51.816343784332275

Table 3. Relationship between the time taken and length of input text taken from [29]

6. CONCLUSION

SAGE has been designed and developed to utilise KG for "thing" annotation. It is capable of annotating text with KG resources and its user-friendly GUI makes exploring things in KGs easy. The tool allows the addition of multiple user-selected KGs to its KB. SAGE not only annotates named entities, but also concepts, entity relations, and attributes. The tool offers two types of annotations: complete match and partial match. A partial match is useful as it annotates contextually relevant resources. The tool also identifies entities mentioned in the text but not defined as a resource in its knowledge base and displays them through parsing with a POS system, which is helpful for further enriching the KG. The tool's unmatched terms pane lists missing resources in the KG and in the future, a GUI will be developed to enable knowledge engineers to add missing things to the KG directly from the SAGE's unmatched terms result interface. At present, SAGE supports only the English language. We aim to support other languages including Indic languages like Hindi and Bengali. Currently, SAGE is a standalone desktop software and, in the future, we plan on providing SAGE as a web-based service.

ACKNOWLEDGMENT

This work is executed under the research project entitled "Integrated and Unified Data Model for Publication and Sharing of prolonged pandemic data as FAIR Semantic Data: COVID-19 as a case study", funded by Indian Statistical Institute Kolkata.

REFERENCES

1. Idehen, K.U. (2020). Linked Data, Ontologies, and Knowledge Graphs. Retrieved December 13, 2022, from https://www.linkedin.com/pulse/linked-data-ontologies-knowledge-graphs-kingsley-uyi-idehen
2. Blumaumer, A., & Kiryakov, A. (n.d.). Knowledge Graphs: 5 Use Cases and 10 Steps to Get There - Ontotext. Retrieved December 13, 2022, from https://www.ontotext.com/knowledgehub/webinars/knowledge-graphs-5-use-cases-and-10-steps-to-get-there/
3. DeBellis, M., & Dutta, B. (2021). The Covid-19 CODO Development Process: an Agile Approach to Knowledge Graph Development. Communications in Computer and Information Science, 1459 CCIS:153–168. https://doi.org/10.1007/978-3-030-91305-2_12/COVER
4. Lin, Y., Mehta, S., Küçük-McGinty, H., Turner, J. P., Vidovic, D., Forlin, M., Koleti, A., Nguyen, D. T., Jensen, L. J., Guha, R., Mathias, S. L., Ursu, O., Stathias, V., Duan, J., Nabizadeh, N., Chung, C., Mader, C., Visser, U., Yang, J. J., … Schürer, S. C. (2017). Drug target ontology to classify and integrate drug discovery data. Journal of Biomedical Semantics, 8(1). https://doi.org/10.1186/S13326-017-0161-X
5. BioAssay Ontology. (n.d.). Retrieved December 13, 2022, from https://bioportal.bioontology.org/ontologies/BAO
6. Hogan, W. R., Hanna, J., Hicks, A., Amirova, S., Bramblett, B., Diller, M., Enderez, R., Modzelewski, T., Vasconcelos, M., & Delcher, C. (2017). Therapeutic indications and other use-case-driven updates in the drug ontology: Anti-malarials, anti-hypertensives, opioid analgesics, and a large term request. Journal of Biomedical Semantics, 8(1). https://doi.org/10.1186/S13326-017-0121-5
7. Google Knowledge Graph. (n.d.). Retrieved December 13, 2022, from https://developers.google.com/knowledge-graph
8. Lehmann, J., Isele, R., Jakob, M., Jentzsch, A., Kontokostas, D., Mendes, P. N., ... & Bizer, C. (2015). Dbpedia–a large-scale, multilingual knowledge base extracted from wikipedia. Semantic web, 6(2): 167-195.
9. Vrandečić, D., & Krötzsch, M. (2014). Wikidata: a free collaborative knowledgebase. Communications of the ACM, 57(10) : 78-85.

10. Huang, X., Zhang, J., Xu, Z. & Ou, L (2021). A knowledge graph-based question-answering method for medical domain. PeerJ Computer Science, 7. https://peerj.com/articles/cs-667/
11. Wolfram|Alpha. (n.d.). Retrieved December 13, 2022, from https://www.wolframalpha.com/
12. Gupta S., Szekely P., Knoblock C.A., Goel A., Taheriyan M. & Muslea M. Karma (2012). A system for mapping structured sources into the Semantic Web. In Extended Semantic Web Conference, Springer, 2012, pp. 430-434.
13. CovidGraph. (n.d.). Retrieved December 13, 2022, from https://healthecco.org/covidgraph/
14. Daiber, J., Jakob, M., Hokamp, C., & Mendes, P.N. : Improving efficiency and accuracy in multilingual entity extraction. In Proceedings of the 9th International Conference on Semantic Systems (I-SEMANTICS '13). Association for Computing Machinery, New York, NY, USA, 2013, pp.121–124. https://doi.org/10.1145/2506182.2506198
15. brat rapid annotation tool. (n.d.). Retrieved December 13, 2022, from https://brat.nlplab.org/
16. doccano, GitHub. (n.d.). Retrieved December 13, 2022, from https://github.com/doccano
17. Chabchoub, M., Gagnon, M. & Web, A. Z (2018). FICLONE: improving DBpedia spotlight using named entity recognition and collective disambiguation. Open Journal Semantic Web, 5(1): 12–28.
18. Nguyen, P., Kertkeidkachorn,N., Ichise, R., & Takeda, H. (2022) MTab4D: Semantic Annotation of Tabular Data with DBpedia. Semantic Web.
19. Shuang Chen, Alperen Karaoglu, Carina Negreanu, Tingting Ma, Jin-Ge Yao, Jack Williams, Feng Jiang, Andy Gordon, Chin-Yew Lin, (2022). LinkingPark: An automatic semantic table interpretation system. Journal of Web Semantics, 74. https://doi.org/10.1016/j.websem.2022.100733.
20. Hogenboom, F., Frasincar, F., & Kaymak, U. (2010). An overview of approaches to extract information from natural language corpora. Information Foraging Lab 69.
21. Dutta, B. & Das, P. SAGE: A Semantic Annotator for knowledge Graph Exploration. In ASIS&T Mid-Year Conference "Expanding Horizons of Information Science and Technology and Beyond", April 11-13, 2023, virtual, https://doi.org/10.5281/zenodo.7597207
22. Lotfi, M., Hamblin, M. & Acta, N. R. (2020). COVID-19: Transmission, prevention, and potential therapeutic opportunities. Clinica Chimica Acta, 508: 254–266. https://doi.org/10.1016/j.cca.2020.05.044.
23. Penn part-of-speech tags (n.d.) Retrieved December 13, 2022, from https://cs.nyu.edu/~grishman/jet/guide/PennPOS.html
24. He, Y., Yu, H., Ong, E., Wang, Y. & Liu, Y (2020). CIDO, a community-based ontology for coronavirus disease knowledge and data integration, sharing, and analysis. Scientific Data, 7(181). https://doi.org/10.1038/s41597-020-0523-6
25. Giunchiglia, F., Maltese, V., & Dutta, B. (2012). Domains and context: first steps towards managing diversity in knowledge. *Journal of Web Semantics: science, Services and Agents on the World Wide Web*, *12-13*, 53-63. http://dx.doi.org/10.1016/j.websem.2011.11.007
26. Object Property Description, Protégé 5 Documentation, GitHub (n.d.). Retrieved December 13, 2022, from http://protegeproject.github.io/protege/views/object-property-description/
27. Dutta, B., & DeBellis, M. (2020). CODO: An Ontology for Collection and Analysis of Covid-19 Data. In D. Aveiro, J. Dietz, & J. Filipe (Eds.), Proc. of the 12th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management - KEOD (pp. 76-85). SciTePress. http://dx.doi.org/10.5220/0010112500760085
28. George A. Miller (1995). WordNet: a lexical database for English. Communications of ACM 38(11): 39–41. https://doi.org/10.1145/219717.219748
29. Marco Ciotti, Massimo Ciccozzi, Alessandro Terrinoni, Wen-Can Jiang, Cheng-Bin Wang & Sergio Bernardini (2020). The COVID-19 pandemic. Critical Reviews in Clinical Laboratory Sciences, 57(6): 365-388. https://doi.org/10.1080/10408363.2020.1783198