**DARPA ASKE DCC – Milestone 7, 2019**

Authors: Akrotirianakis Ioannis, Amar Viswanathan Kannan, Fradkin Dmitriy, Roy Aditi, Tugba Kulahcioglu, Canedo Arquimedes, Mohammad Al Faruque

**Contents**

[1. Introduction 3](#_Toc20754796)

[2. Summary and general architecture of DCC 3](#_Toc20754797)

[3. Knowledge Graphs for machine assisted curation 4](#_Toc20754798)

[3.1 Overview 4](#_Toc20754799)

[3.2 Knowledge graph population 5](#_Toc20754800)

[3.3 Triple store and RDF graphs 5](#_Toc20754801)

[3.4 Featured queries 5](#_Toc20754802)

[3.5 Future improvements 6](#_Toc20754803)

[4. Text2Graph 7](#_Toc20754804)

[4.1 Overview 7](#_Toc20754805)

[4.2 Combining our annotations with other sources 8](#_Toc20754806)

[4.3 Computer Science Ontology analysis 9](#_Toc20754807)

[4.4 Combining human annotations with CSO 11](#_Toc20754808)

[4.5 Automatic annotation 12](#_Toc20754809)

[4.6 Integration within the text2graph pipeline 12](#_Toc20754810)

[4.7 Future improvements 13](#_Toc20754811)

[5. Image2Graph 13](#_Toc20754812)

[5.1 Overview 13](#_Toc20754813)

[5.2 Flow detection in the absence of arrows in DL diagrams: 14](#_Toc20754814)

[5.3 Component Extraction using Region-based Convolutional Neural Networks (R-CNN) 16](#_Toc20754815)

[5.4 Improving flow detection in DL diagrams: 16](#_Toc20754816)

[6. Code2Graph 17](#_Toc20754817)

[6.1 Overview 17](#_Toc20754818)

[6.2 Expanding the Dataset that code2graph can Tackle 18](#_Toc20754819)

[6.3 Formatting the RDFs to Match Turtle Format 18](#_Toc20754820)

[6.4 Future Work 20](#_Toc20754821)

# Introduction

The purpose of this report is to provide an overview of the work that has been done during **August 1st** to **September 30th, 2019**. We present details for our multimodal (text, images, and code) knowledge graph extraction process and discuss the various updates in our three major modules: text2graph, image2graph, and code2graph. Our effort focused around improving the learning models by increasing the datasets associated with text, image and source code. Our models can now handle larger datasets and detect more accurate relations within the knowledge graph. Details of our progress and results are described in the following sections of this report.

# Summary and general architecture of DCC

We start by briefly reviewing the general architecture of our framework which is shown in Figure 1. The text2graph module generate a Knowledge Graph (KG) from text, where nodes are entities describing deep learning methods or tasks, and the edges represent the relations between those entities. For the KG generated from images (*image2graph*), the entities are derived by the shapes used to describe the deep learning architecture and the text within them, whereas the relations are derived by the arrows connecting those shapes. Finally, the *code2graph* module generates a KG using the information extracted from the source code, where the nodes are actual functions used by TensorFlow, and the edges represent the way these functions are related to form the executable code. It should be noted that in order to construct each KG for the whole corpus, we first apply our models to individual papers and then integrate the entities and relations for all the documents and all modalities. We only focus on papers that use Python as the programming language and TensorFlow/Keras as the Deep Learning framework.

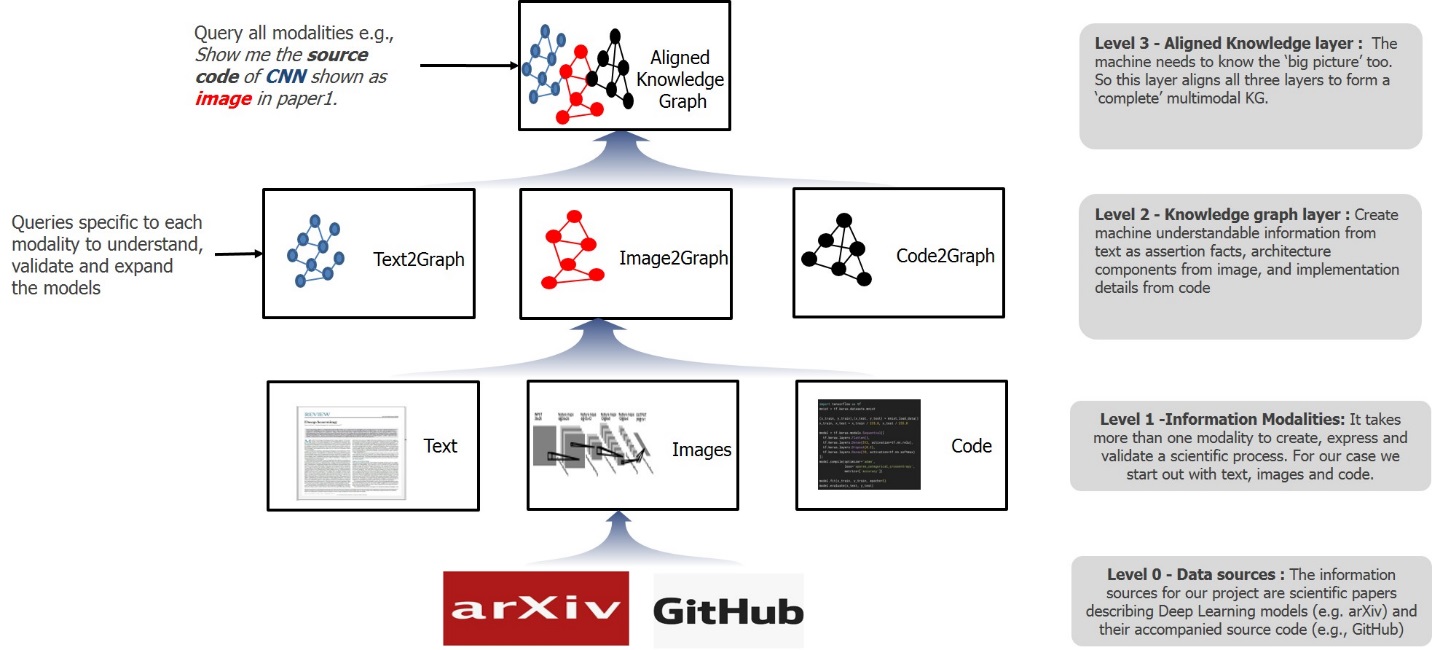


Figure 1: DCC architecture within the ASKE framework

# Knowledge Graphs for machine assisted curation

## Overview

During Milestone 7, we focused on the following aspects:

1. Improving accuracy of extraction of all modalities
2. Increasing the number of datasets for extraction
3. Creating domain specific schemas that allow for better retrieval of machine curated deep learning models

In the last milestone we created an initial version of our ontology schema (DeepSciKG). Our goal was to represent three diverse modalities in one single representation. In knowledge graphs this necessitated the creation of our schema. The initial version consisted of a set of 189 Classes, 12 Object Properties, 10 Data Properties and 10 annotation properties. Designed with extensibility in mind, this schema was able to represent knowledge from text, images and source code as abstract syntax trees.

Since the last milestone, based on our extraction capabilities, we have improved the same ontology to include 277 Object classes, 16 Object Properties, 13 Data Properties and 2 annotation properties. The object classes were increased to enable more fine tuned and precise querying over the knowledge graph. Most of the changes to the classes came from adding more TensorFlow modules based on our extraction and more text entities and image components. The ontology schema can be quickly summarized as follows:

1. The top level classes are CodeEntity, FigureComponent, Function, ImageComponent, Modality, Publication, PublicationAuthor, Repository, SourceCodeFile,, TextEntity and tf
2. CodeEntity basically consists of two kinds of classes – TensorFlowDefined and UserDefined. TensorFlowDefined
3. FigureComponent basically categorizes all the image2graph extractions as their own classes and they are – ActivationBlock, ConcatBlock, ConvBlock, DeconvBlock, DenseBlock, DropoutBlock, EmbedBlock, FlattenBlock, InputBlock, LossBlock, LSTMBlock, LSTMSeqBlock, NormBlock, OutputBlock, PoolingBlock, RNNBlock, RNNSeqBlock, UnpoolingBlock
4. Function and ImageComponent are used to categorize instances and map them to the right classes.
5. Modality consists of Code, Figure and Text, where Text is subdivided into AbstractText, BodyText, CaptionText and TitleText
6. Publication and PublicationAuthor focuses on representing just the publication metadata and author metadata
7. Repository and SourceCodeFiles are used for more fine grained representations of source code data
8. TextEntities represent the set of classes that are extracted from the text2graph task. We currently extract – GenericTerm, Material, Method, Metrick, OtherScientificTerm and Task
9. Finally, tf represents the hierarchy of modules that are present in the tensorflow documentation.

We extract instances of text entities, image blocks and code segments and map them to the right classes in the ontology. This allows us to query all the three modalities simultaneously. The metadata also allows us to see the relations between publications and authors that will give an overall view of where deep learning-based development happens over time.

In phase two we have worked on constructing a multi modal linked repository for papers and source code related to DL algorithms and methods. This will enhance and enable knowledge aggregation, discovery, analysis and serve as a test base for generating rapid prototypes of state-of-the-art research. This was a two-step *process i) Domain Ontology creation and ii) Knowledge graph population*

## Knowledge graph population

As stated above knowledge graph population focused on populating the schema with instances extracted from the different modalities. For this, we consolidated the papers from papers with code and organized them by conference type, year and code framework. From the initial extraction of the dataset we were able to consolidate 1848 papers. Out of these 1848 papers we narrowed down on 739 papers which contain code repositories published in either pytorch or tensorflow. The datasets for both of these can be found in the repository development branches i.e. <https://github.com/deepcurator/DCC/tree/development/src>. The file pwc.csv shows the entire 1848 papers, while the file pwc\_edited\_tensorflow\_pytorch.csv is the file that shows all the papers that we can process. The statistics for the knowledge graph are shown below:

|  |  |
| --- | --- |
| **Modality** | **Graphs generated** |
| Text | 739 |
| Images | 151 |
| Code | 444 tensorflow graphs |

## Triple store and RDF graphs

The RDF graphs for each of the following modalities are published in our Open Science Foundation repository at <https://osf.io/jdhw8/>. Now this repository contains individual graphs of all the 3 modalities for every paper. To load these graphs, you would need an RDF enabled Triple Store. We used Virtuoso Open Source Edition version 7.2.5. This is available at <https://github.com/openlink/virtuoso-opensource/releases>

To load the graphs we followed the Virtuoso Wiki instructions at <http://vos.openlinksw.com/owiki/wiki/VOS/VirtBulkRDFLoader>

## Featured queries

RDF graphs are queried using a query language called SPARQL. This is a graph pattern query language which uses a conjunction of patterns to query the graph and deliver results. These graph pattern queries are very expressive and a sample of the featured queries which we think are interesting is shown below in plain English (will actually be in SPARQL in the demo):

1. Show all ICML papers using conv blocks and the tensorflow implementations used for the same with default arguments
   1. This query now will show all the default arguments used, which will give scientists an idea of implicit default values used in a particular domain or subdomain (in this case convolutional neural networks in ICML)
2. Show all the activation blocks used in tensorflow by NeurIPS, CVPR, ICML, ECCV and ICCV along with the default argument parameters
   1. This will give an idea of the most commonly used activation functions
   2. The same query can be filtered even more with relu, tanh and other kind of activations as well.
3. Show all GRUCells definition used by papers since 2013 to 2018 by all conferences.
   1. This will show all GRUCells, which is a tensorflow defined function used by all published code of conferences.

## Future improvements

We are currently focusing our work on the following two main directions

**Graph Based:**

Before the next milestone, we will be working on the following for the multimodal knowledge graph:

1. Automate the alignment process with more fine grained details, particularly :
   1. Utilizing entity properties as features for alignment - Currently we are using textual features to see alignment. However, a convolutional block would consist of a block of entities from other modalities. Our aim is to investigate such scenarios.
   2. Merge entity metadata with DBpedia and other scientific graph labels - DBpedia and other public graphs can have well defined entity schemas and we intend to utilize them for seamless integration with the Linked Open Data community and for better searchable graphs.

**Code2Graph Based:**

We have started work on translating PyTorch code to graphs as well. We have currently developed a system that extracts Abstract Syntax Trees from both PyTorch and Tensorflow codes. In the next milestone, the PyTorch code will also be included in the knowledge graph, which will lead to more fine-grained class definitions and a wider coverage of deep learning frameworks. This will also allow scientists to query for cross domain code for the same deep learning model and look at how default arguments vary across platforms.

# Text2Graph

## Overview

We have enhanced the general architecture of the text2graph module focusing on the generation of larger datasets that will be used to train our statistical models for Named Entity Recognition (NER) and Relation Extraction (RE). Training robust and accurate NER and RE models requires high quality of labeled data, namely, text that have been annotated with entities and relations that are more relevant and have a unique meaning in the context of Deep Learning. This is a time-consuming process if it is done manually, as it requires the availability of highly skilled domain experts reading and annotating a large number of Deep Learning papers. In order to increase our labeled data (annotated text) we have developed a process that can *automatically annotate DL papers*, thereby significantly reducing the manual effort required for this vital task. The updated general pipeline of our text2graph module is shown in Figure 2 where the additional annotation components can be seen.

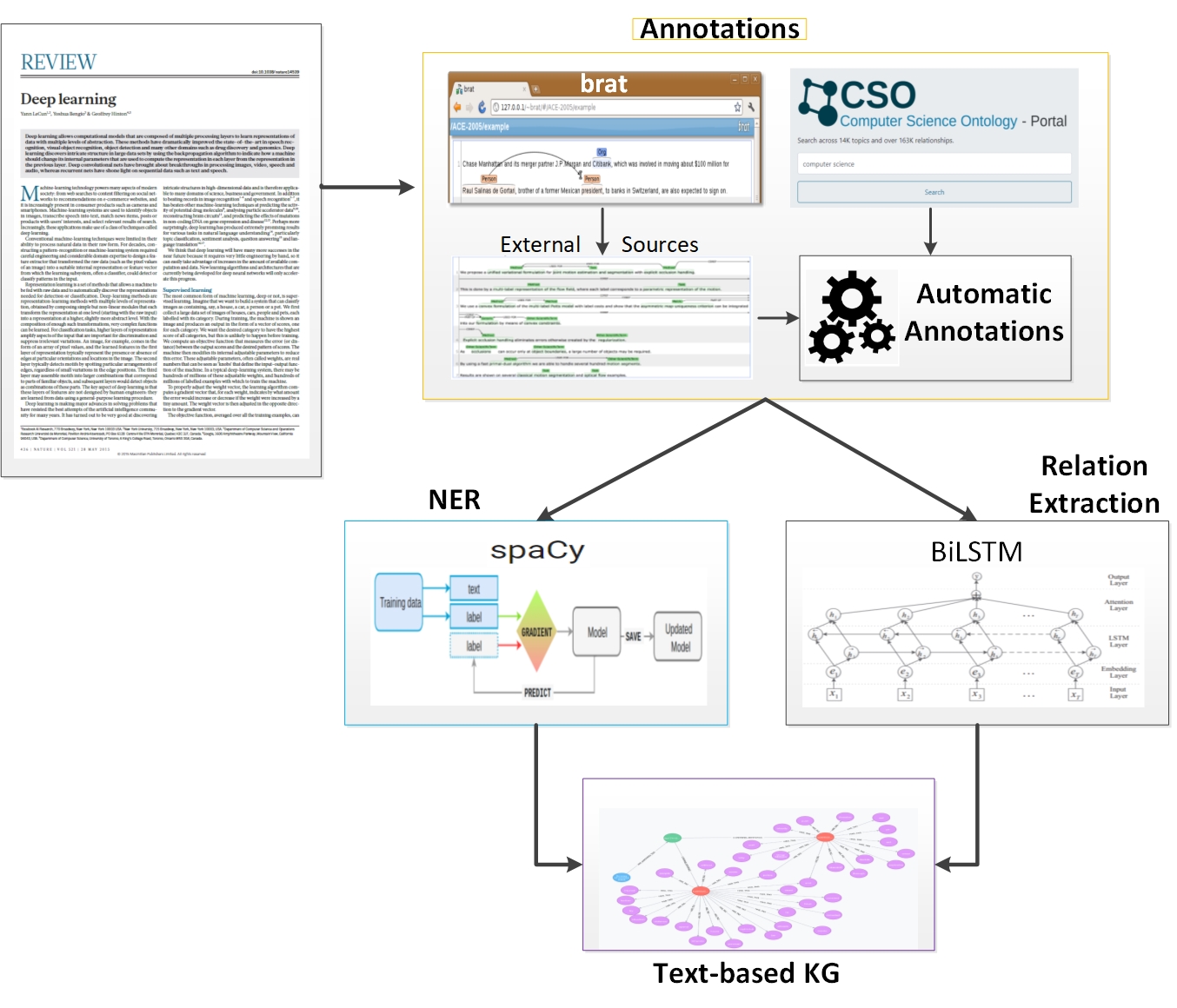


Figure 2: The text2graph pipeline

In addition, building a Knowledge Graph for a complex scientific domain, such as Deep Learning, it is easier when an initial set of entities and relations are available, so that information extracted from new documents can be added to it, rather than building it up completely from scratch. To accomplish the above tasks, we needed

1. a vocabulary of strings mapping to entity types of interest,
2. a mapping from pairs of strings to relations of interest.

In the following subsections we will describe how we constructed such an initial *ontology*, and an *automated annotation tool*, by combining three data sources:

* Our own original annotations of named entities and relations, produced during Phase 1 using the web-based annotation tool Brat[[1]](#footnote-2) on 100 Deep Learning papers taken from [PapersWithCode.com](https://paperswithcode.com/),
* External annotations derived from the sciIE project[[2]](#footnote-3) of the University of Washington,
* The Computer Science Ontology (CSO)[[3]](#footnote-4), which is a large-scale ontology of various research areas in CS and consists of about 16 million publications.

## Combining our annotations with other sources

The first source of annotations at our disposal is the one we produced during Phase 1 on 100 abstracts, using Brat. We collect all this data with some automated clean-up, and manual review. This results in:

* 1,121 terms mapped to entities
* 698 mappings from term pairs to relations[[4]](#footnote-5)

The second source is the annotations from the sciIE project from the University of Washington. These annotations have several differences with ours. More specifically:

* Entities:
  + we consider the following 6 entities: Method, Generic, Task, Material, Eval, Other
  + sciIE considers the following 6 entities: Method, Generic, Task, Material, Metric, OtherScientificTerms
* Relations:
  + we consider the following 7 relations: Compare, Conjunction, Feature-of, Part-of, Used-for, IsA, sameAs
  + sciIE considers the following 8 relations: used-for, feature-of, coref, conjunction, hyponym-of, evaluate-for, part-of, compare

These differences can be resolved by mapping the sciIE entities and relations to ours as follows:

* Metric 🡪 Eval
* OtherScientificTerms 🡪 Other
* Hyponym 🡪 isA
* Also, we remove all COREF relations because we currently do not handle them in the same way
* All other entities and relations stay the same (up to case change)

The annotations obtained from sciIE contain 5,885 distinct entity strings and 6,035 relation strings. Initially, we convert everything to lower-case and remove duplicates – this results in 5,813 entities and 6,008 relations.  After removing 'Coref', only 4,659 relations are left. Next, we remove *conflicts* which arise when the same string has two different entity types, or the same pair has two relation types. This results in:  5,539 entities and 4,647 relations[[5]](#footnote-6).

Furthermore, we combine our annotations and those from the sciIE project, before merging with the CSO (which will be discussed in the next section). We use a similar cleaning approach to the one described previously, that is, (i.) we concatenate data frames of entities and relations from our annotation and SciIE, (ii.) remove duplicate entries and (iii.) remove conflicts (of which there are relatively few). This leaves us with 6,512 entities and 5,342 relations.

## Computer Science Ontology analysis

The Computer Science Ontology (CSO) is a large-scale ontology of research areas in Computer Science, containing multiple entities and instances of relations. We aim to integrate CSO into our Knowledge Graph and will consider occurrences of entities and relations from CSO in text as *valid annotations*. The ‘.owl’ file downloaded from CSO[[6]](#footnote-7) contains 162K triples. Some of our observation are:

* There are multiple entities, some pointing to external resources such as Wikipedia, DBPedia, Microsoft academia,
* There are 8 unique relations:
  + type –

rdflib.term.URIRef('<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>'),

* + label –

rdflib.term.URIRef('<http://www.w3.org/2000/01/rdf-schema#label>'),

* + contributesTo - rdflib.term.URIRef('<http://cso.kmi.open.ac.uk/schema/cso#contributesTo>'), -- this is similar to our own relation type of 'usedFor’
  + relatedEquivalent –

rdflib.term.URIRef( '<http://cso.kmi.open.ac.uk/schema/cso#relatedEquivalent>'), this is equivalent to our own relation type of 'sameAs'

* + preferentialEquivalent –

rdflib.term.URIRef(' <http://cso.kmi.open.ac.uk/schema/cso#preferentialEquivalent>'), this is equivalent to our own relation type of 'sameAs'. Moreover, the entry pointed to is the one usually containing all the relevant links and relations

* + sameAs –

rdflib.term.URIRef('<http://www.w3.org/2002/07/owl#sameAs>'), refers only to external URIs (DBPedia, Wikipedia)

* + superTopicOf –

rdflib.term.URIRef('<http://cso.kmi.open.ac.uk/schema/cso#superTopicOf>'), this is inverse of our own relation of 'isA'

* + relatedLink –

rdflib.term.URIRef('<http://schema.org/relatedLink>')

Further analysis and attempts to build NER & RE models using this ontology revealed the following *challenges*:

* Topic names (after decoding from URI to utf8) often include terms in parenthesis:
  + abbreviations (or reverse) in inconsistent styles/formats. For example:
    - 'xml (extensible\_markup\_language)' – abbreviation is outside of parenthesis, while the full term is inside
    - 'support\_vector\_machine (svms)' – mismatch in plural vs singular
    - 'peer to peer (p2p) network' – abbreviation is in middle of phrase, use of '2' to represent '2'
    - 'orthogonal frequency division multiple access (ofd' –  incomplete parenthesis
  + certain hard-to-detect alternatives. For example: 'two-dimensional (2d)', 'wireless (wifi) communication'
  + contextual terms within parentheses. For example: 'android (operating system)', 'prolog (programming language)' – the context within the parentheses needs to be removed to identify entities properly
  + complex terms. For example: '(min ,max ,+) functions', '(i ,j) conditions'; here the values in parenthesis are part of topic name and can't be removed. Thus, this and previous case cannot be separated.
* The only entity type in CSO is 'topic' and the set of relations is limited and applies to topics rather than concepts. For example, 'neural networks' may be superTopicOf 'backpropagation', but clearly 'backpropagation’ is not a type of 'neural network'. Thus the superTopicOf is not a useful relation.

Despite of the above challenges, we can still attempt to expand our vocabulary and set of relations by combining CSO and human annotations. The following approach is taken to integrate CSO:

* *Processing topics:* 
  + we focus on nodes with URI starting with ''<https://cso.kmi.open.ac.uk/topics/>'
  + in the URI suffix (after topic), we replace '\_' with spaces
  + check for parenthesis and attempt to verify whether inner part of text is abbreviation of outer (or vice-versa).
    - Check the shorter part against common stopwords, to avoid having abbreviations like 'of' causing problems later
    - If the shorter part is not a stopword , we separate the values, and add 'sameAs' relation between them
    - If it is, we ignore it - it won't be very helpful in entity detection anyway
* *Processing relations:* 
  + In the 1st pass we create a map for each URI to '[preferentialEquivalent](http://cso.kmi.open.ac.uk/schema/cso" \l "preferentialEquivalent)'. If a URI does not have a '[preferentialEquivalent](http://cso.kmi.open.ac.uk/schema/cso" \l "preferentialEquivalent)', it maps to itself.
  + , for each entity string, we create a map for the string to the full URI of the preferential equivalent
  + We keep a map from pairs of strings to relations of interest: [cso#contributesTo](http://cso.kmi.open.ac.uk/schema/cso" \l "contributesTo), [superTopicOf](http://cso.kmi.open.ac.uk/schema/cso" \l "superTopicOf)

## Combining human annotations with CSO

The process of merging human annotations with CSO is as follows:

1. Extract topics from CSO as strings, and then map them to URIs
2. Extract some useful relations from CSO corresponding to: usedFor and sameAs. The first two steps are performed by a script called collect\_cso.py
3. Find which of CSO terms occur in our annotation data, and then generate sets of terms both from CSO and annotations that we can thus map to the same URI.
4. Create new URIs for annotation terms that didn't match any in CSO
5. For all the URIs from steps 3 and 4, we create a mapping to our entity types (Method, Data, Generic) based on annotations
6. Merge relations from CSO and from our annotations
7. Thus, we end up with:
   * mapping from strings to URIs, with multiple strings mapping to same URI: 20102 strings  (though many may not be relevant for deep learning)
   * mapping from URIs to Entity Types: 6,455 URIs
   * mapping from pairs of URIs to Relations: 49,918 pairs

Steps 3-7 are performed by merge\_cso\_brat.py

Note that the above process may result in *conflicts* between CSO and annotations, or between annotations made by different users in different contexts. Some examples are below:

* "short-time fourier transform": <https://cso.kmi.open.ac.uk/topics/stft> vs <https://cso.kmi.open.ac.uk/topics/short_time_fourier_transforms>
  + Due to CSO treating these as different topics. Fixing would require changing CSO
* "ir":<https://cso.kmi.open.ac.uk/topics/information_retrieval> vs <https://cso.kmi.open.ac.uk/topics/impulse_radio>
  + Handled by excluding 'ir' from abbreviations extracted from CSO
* "named entities": Task vs Generic
  + Happens because 'named entity' and 'named entity recognition' are equivalent topics in CSO, but are mapped to different entities by annotators. The reasons for other conflicts of same type (below) are similar: 'language model' and 'language modelling'
* "state space models": Method vs Generic.  Fixed by adjusting brat annotation to be Method
* <https://siemens/quantum%20graphical%20models> - "https://cso.kmi.open.ac.uk/topics/graphical\_model": Compare vs isA
* "https://cso.kmi.open.ac.uk/topics/chinese\_word\_segmentation"-"https://cso.kmi.open.ac.uk/topics/word\_segmentation": isA vs Feature-of
  + The two above were manually corrected in brat\_relations\_clean.txt

Additional conflicts appeared when we merged with sciIE annotations, for example:

* "natural language processing": Method vs Task
* “question answering systems": Task vs Method
* "machine learning algorithms": Task vs Method
* "semantic annotation": Other vs Method
* "https://cso.kmi.open.ac.uk/topics/automatic\_evaluation"-"https://cso.kmi.open.ac.uk/topics/machine\_translations": isA vs Evaluate-for
* "https://cso.kmi.open.ac.uk/topics/constraint\_propagation"-"https://cso.kmi.open.ac.uk/topics/constraint\_programming": isA vs Part-of

In the following months, we will work on improved conflict avoidance and resolution. At present time, these conflicts are either resolved manually (by editing annotations) or randomly – by keeping the first relation or entity that a string or pair maps to.

## Automatic annotation

We can now use the data structures created in the previous section to annotate new pieces of text. Specifically, given a regular text taken from a PDF file describing a DL paper, we can:

1. find occurrences of all strings in our vocabulary in the text, with corresponding URIs,
2. select those for which we have URI-to-entity mapping: we now have performed entity recognition

* We can check for *overlapping annotations*, namely those covering the same text, and take the one corresponding to the longest text. For example, if text contains ‘loss function’, and we have entities ‘loss function’ and ‘loss’, only the annotation corresponding to the ‘loss function’ will be retained.

1. For all pairs of strings matching entities we check for relations in the URI pairs-to-relations map: this step provides initial relation extraction

This workflow is implemented in the script auto\_annotate.py

## Integration within the text2graph pipeline

The overall vision that we are working towards is that of a dynamic, continuously expanding ontology for Deep Learning publications. More specifically:

* When a new publication becomes available, we use text2graph pipeline (trained using Auto-Annotated data) to identify entities or relations between pairs of entities not yet present in the ontology and add them to the ontology.
* After processing a number of new publications we can retrain our models using auto-annotation approach on the expanded ontology
* We can also use auto-annotation to annotate new publications – this doesn’t fit with the workflow, but can be a simpler alternative?
* We can try to leverage SNORKEL[[7]](#footnote-8) to improve auto-annotate and derive better labels.

## Future improvements

There are several directions for improving the quality of the data collected:

* Improved resolution of conflicts in manual annotations - possibly with a human review of conflicts, which are relatively rare
* Improved resolution of conflicts when merging manual annotations with CSO, also possibly with human review
* Improved relation extraction by creating an explicit graph from the 'ground truth' data and performing reasoning about relations. For example, if ‘A Used-For B’ and ‘B isA C’, then we should be able to infer that ‘A Used-For C’ without this being explicitly in the relation map.
* Expanding the vocabulary and relations by bringing in other ontologies, such as DBPedia and Yago. It is necessary to keep in mind, however, that these will not have the same entity types, and many of the relations we are looking for, and that only a fraction of such ontologies will be relevant for our domains. Thus, the anticipated effort will need to be balanced by expected benefits.

# Image2Graph

## Overview

The main architecture of the image2graph module is shown in Figure 3 and has not changed from that developed in Phase 1. Given the PDF of a deep learning research paper, the image2graph module consists of four major components: (1) extraction of all the figures from a research paper, (2) identification of those figures showing DL model diagram, (3) analysis of the diagrams depicting DL models, and (4) construction of a graph representing the information extracted from the DL diagram. During the last two months we have focused on improving the detection of the information flow, as well as interpreting different colors in nodes and arrows that are used in Deep Learning architectures. The results are summarized in the following sections.

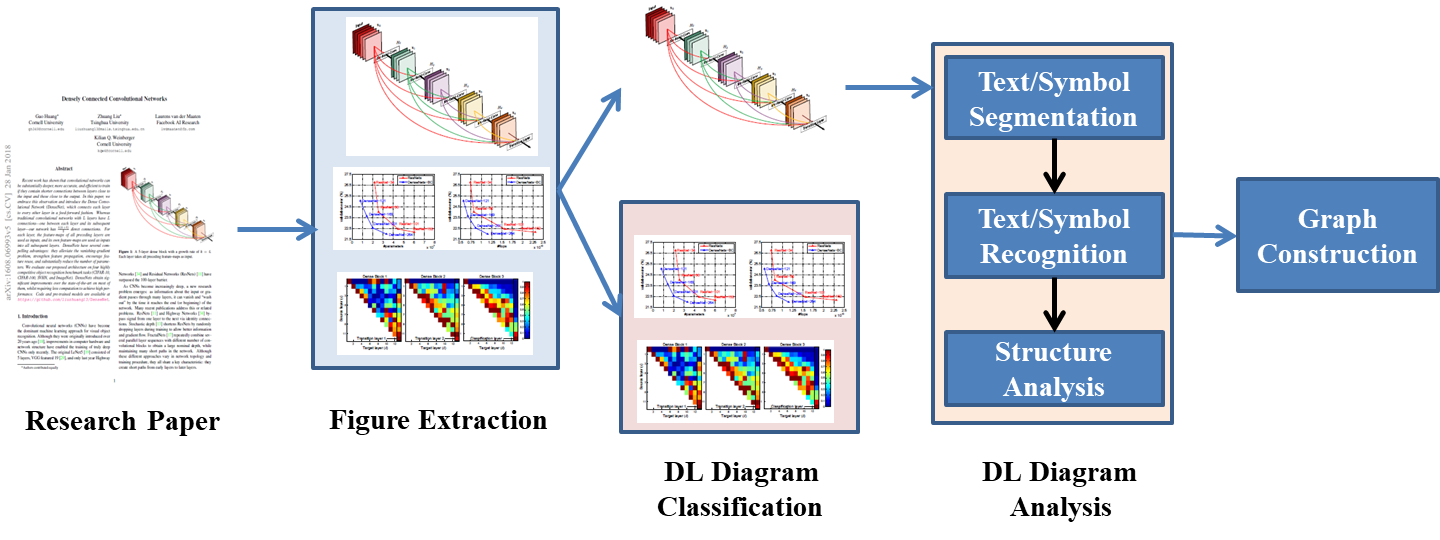
****

Figure 3 The image2graph pipeline

## Flow detection in the absence of arrows in DL diagrams:

The image2graph graph prediction takes as input an image from appear illustrating a DL diagram, and returns as output a set of relationships denoted as

<subject-predicate-object>.

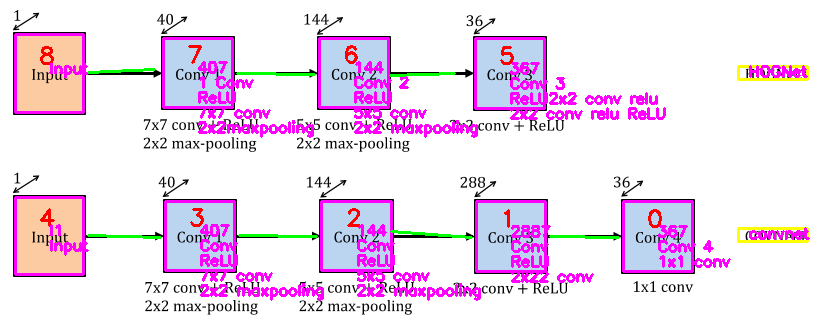
For example, <component1 – isType - activation> and <component1 – hasDescription – [‘leaky’, ‘relu’]>.

Currently we identified 10 predicates/relations to represent a diagram as follows:

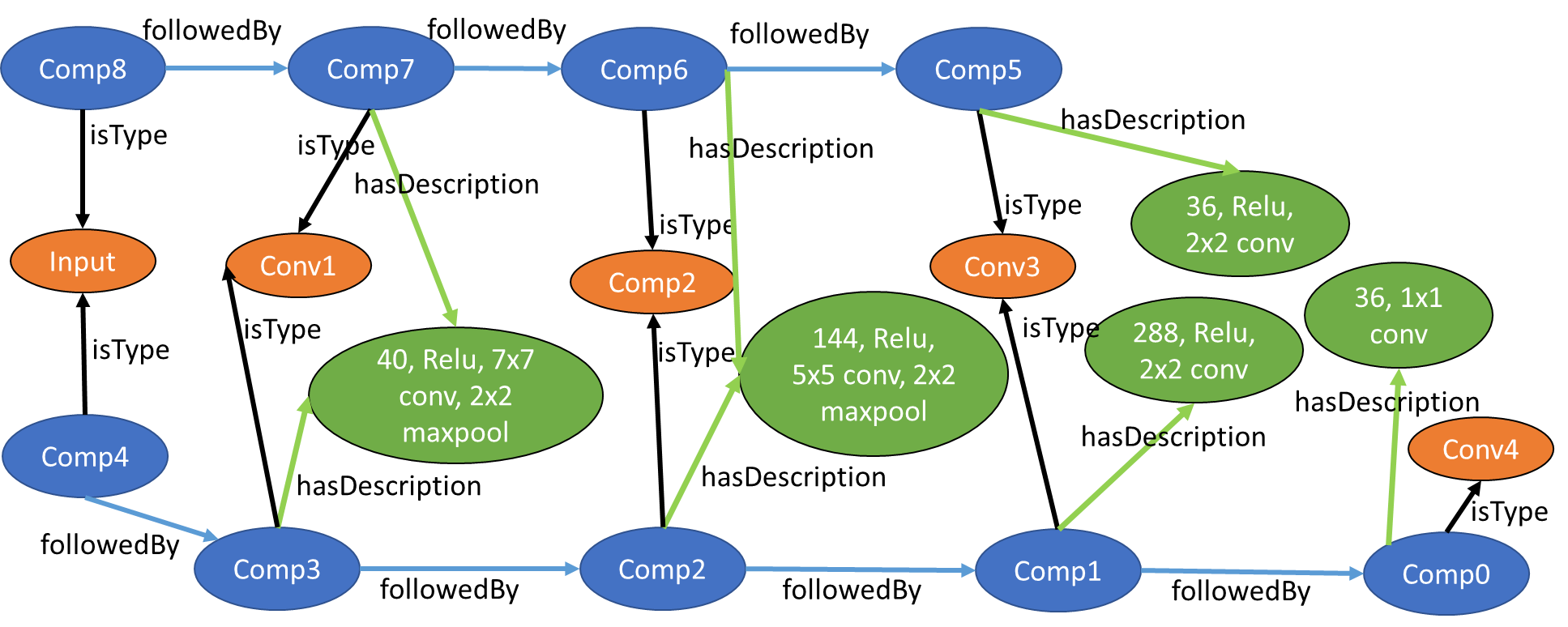
1. “isA”: <FigureID - isA - Figure>
2. “foundIn”: <FigureID - foundIn – “paper title”>
3. “hasCaption”: <FigureID - hasCaption – “figure caption”>
4. “partOf”: <ComponentID/TextTD - partOf - FigureID>
5. “hasPos”: <ComponentID/TextID - hasPos - (x, y, h, w)>
6. “isType”: <ComponentID/TextID - isType – “layer name”>
7. “hasDescription”: <ComponentID/TextID - hasDescription - [list of words]>
8. “hasFlow”: <FigureID - hasFlow – “flow direction”>
9. “followedBy”: <ComponentID/TextI - followedBy - ComponentID/TextID>

Each diagram figure is represented by a unique FigureID which is a combination of paper file name and figure number within that paper. The “foundIn” predicate/relation represents the title of the paper from where the current diagram is extracted. “hasCaption” represents the caption associated with the current diagram figure. Next, from each diagram multiple text components or node components are extracted each of which is represented by a unique ID. “partOf” relationship establishes presence of such unique components. Next each component is described with its position using the relationship “hasPos”. The type the component in terms of different layer name is captured with “isType” predicate. The layer names used for “isType” predicate are: "conv, "deconv”, "dense”, "flatten”, "dropout", "pooling", "unpooling", "concat", "RNN", "RNNseq", "LSTM", "LSTMseq", "norm", "embed", "activation", "loss". All other description found in the text box or node (like kernel size, etc) is put under “hadDescription” predicate.

Finally, to establish links among multiple components found in a diagram, flow direction is an important information. “hasFlow” captures overall flow direction information in terms of either ”left-to-right” or “right-to-left” or “top-to-bottom” or “bottom-to-top”. Individual connectivity between two components is captured using “followedBy” predicate. Figure 4-b shows one such graph with only “isType”, “hasDescription” and “followedBy” predicates extracted from a diagram shown in Figure 4-a.



(a)



(b)

Figure 4: (a) All the nodes extracted from an example DL diagram, (b) image2graph extended form of (a)

## Component Extraction using Region-based Convolutional Neural Networks (R-CNN)

To automatically detect the objects/symbols in an image we attempted to use the deep learning based technique named *faster R-CNN[[8]](#footnote-9)*.

However, in the absence of any labeled dataset from this domain of DL diagram, we started with creation of our own dataset for training a R-CNN. We created a prototype database from 100 diagram figures that contain approximately 2,200 nodes and 1,000 arrows. We plan to increase the dataset size with an addition of another 1,000 images and then explore the approach with attention-guided learning to remove bias in training data and re-focus the network’s attention on the right and consistent patterns encountered in DL models. In addition, this approach will provide robustness and consistency in detecting the right building blocks of DL architectures and the information they convey (e.g., there are different ways to represent a CNN architecture that are used in DL papers). Top-down attention maps[[9]](#footnote-10) can be seen as one form of interpretation of neural networks.

## Improving flow detection in DL diagrams:

As discussed in the previous report (Milestone 6), extraction of overall flow direction is required in absence of arrows or when arrow detection is erroneous. The first step to detect such flow direction is to determine whether the is *horizontal* or *vertical*. We apply a combination of component projection and pixel-wise value projection along the vertical and horizonal directions. The one with maximum number of components and pixel values are assumed to have dominant flow. For example, in Figure 5, the projection along horizontal direction clearly shows more weight than the vertical one. Thus, the flow direction in this case is detected as horizontal.

The diagrams having horizontal flow direction could be either *left-to-right* or *right-to-left*. Similarly, for vertical flow, flow could be either *top-to-bottom* or *bottom-to-top*. The default flow direction is assumed to be left-to-right for horizontal and top-to-bottom for vertical flow. In the absence of any arrow, we try to detect the presence of an “input” component which will mark the starting point of the flow. Alternatively, the components with names “output” or “FC”[[10]](#footnote-11) or “loss” are searched in a diagram, in order to identify the ending component. If anyone of these components is found, then the flow direction is updated accordingly.

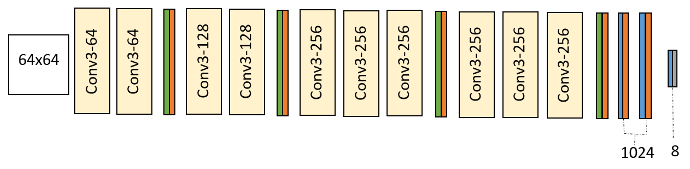


Figure 5: DL diagram without arrows showing direction

# Code2Graph

## Overview

The focus of the code2graph module has been on the development of methodologies to extract the Resource Description Framework (RDF) representations from the source code included in Deep Learning (DL) publications. During DCC Phase 1, two main approaches have been studied: (i.) the Computational Graph-based Approach and (ii.) the Lightweight Approach. In the Computational Graph-based Approach, we created a pipeline to extract and simplify the RDF graphs by executing the code (shown in green-colored boxes). On the other hand, the Lightweight Approach extracts the graphs by analyzing the abstract syntactic structure of the code (shown in blue-colored boxes). The following sections describe the status of code2graph in Milestone 7. The summaries of both extracting approaches were included in the Milestone 5 report. The updated pipeline architecture was explained in Milestone 6.

In Milestone 7, the main tasks associated with the code2graph module are

1. Expanding the dataset that code2graph can tackle,
2. Formatting the RDFs generated by the code2graph module to improve compatibility with the other two modalities (i.e., image2graph and text2graph).

The current overall pipeline is shown in Figure 6.

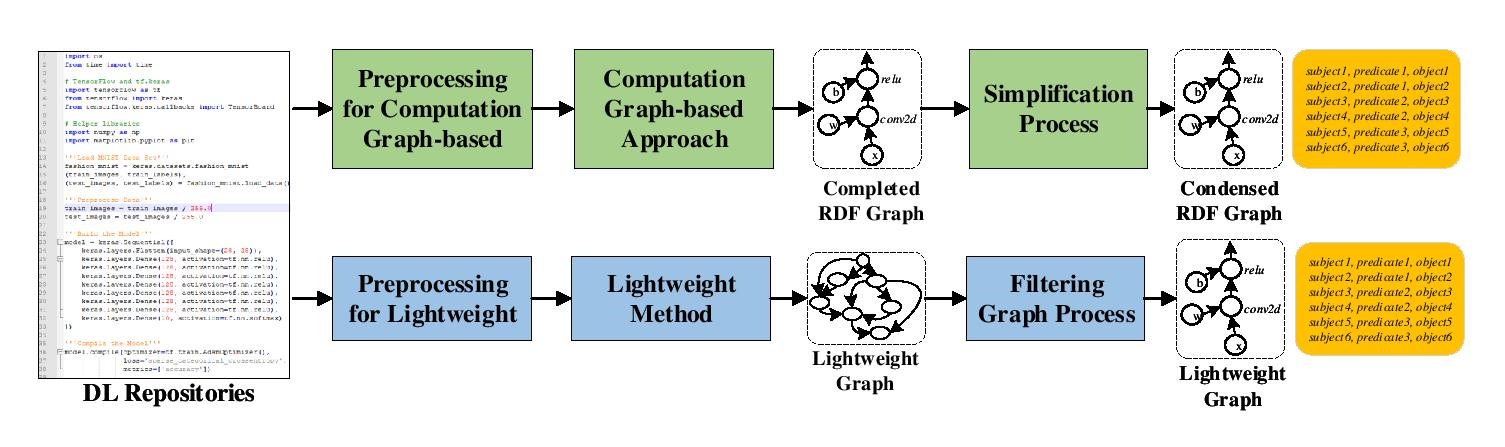


Figure 6: The current code2graph pipeline

## Expanding the Dataset that code2graph can Tackle

The original dataset we acquired from Siemens contains 100 papers, 16 of which had repositories that adopt TensorFlow or Keras as the primary machine learning framework. However, compared to known knowledge-base datasets, the size of the data is not enough to perform meaningful Knowledge Graph Embedding (KGE) tasks. Therefore, in Milestone 6, we developed the PWC-Scraper in order to continuously scrape the website [PapersWithCode.com](https://paperswithcode.com/) and obtain information related to DL papers including: ‘title”, ‘abstract, ‘paper link’, ‘code link’, ‘tags’, and the compressed source code of the corresponding master branch.

For the current milestone (Milestone 7), we have further enhanced the PWC-Scraper so that it will now automate the process of scraping paper items, extracting the corresponding lightweight RDF graph from the saved source code, and reporting the overall statistics via email. To date, we have already collected 956 scientific papers and their accompanied repositories. Among those papers, 156 papers adopt TensorFlow as the primary programming framework and 134 of the TensorFlow papers can successfully be processed by the Lightweight approach.

## Formatting the RDFs to Match Turtle Format

During the Milestone 7, we have focused on adjusting the format of RDFs generated from code2graph. We have already adopted and updated the template ontology definition provided by Siemens Group. Each RDF generated from the lightweight method will be in *turtle format (.ttl)*. With the base namespace “'<https://github.com/deepcurator/DCC/>'”, we built the Unified Resource Identifiers (URI) of entities and relations using this base namespace as prefix. The hierarchical snapshots of the ontology are shown in the Figure 7. To specify the hierarchical relationship between types, we used the predicate defined by the W3 community: “<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>”. The root of the type information is *owl:Thing* and 8 upper classes which are *Entity, Function, ImageComponent Modality, Publication, Repository, SourceCodeFile,* and *tf* as shown in Figure 7.A.

Under *Function*, we have two subclasses: *TensorFlowDefined* and *UserDefined*. The subclass information under *TensorFlowDefined* is acquired by parsing all the available API function calls from the official TensorFlow website. The connections are constructed using the relations defined in *object properties* (shown in Figure 7.B) and *datatype properties* (shown in Figure 7.C). The difference between an object property and a datatype property lies in the fact that an object property defines a relation between two entities while a datatype property defines a relation between an entity and a literal. For example, entities that are user-defined functions can be connected by the relation *calls*, meaning that one of them calls the other.

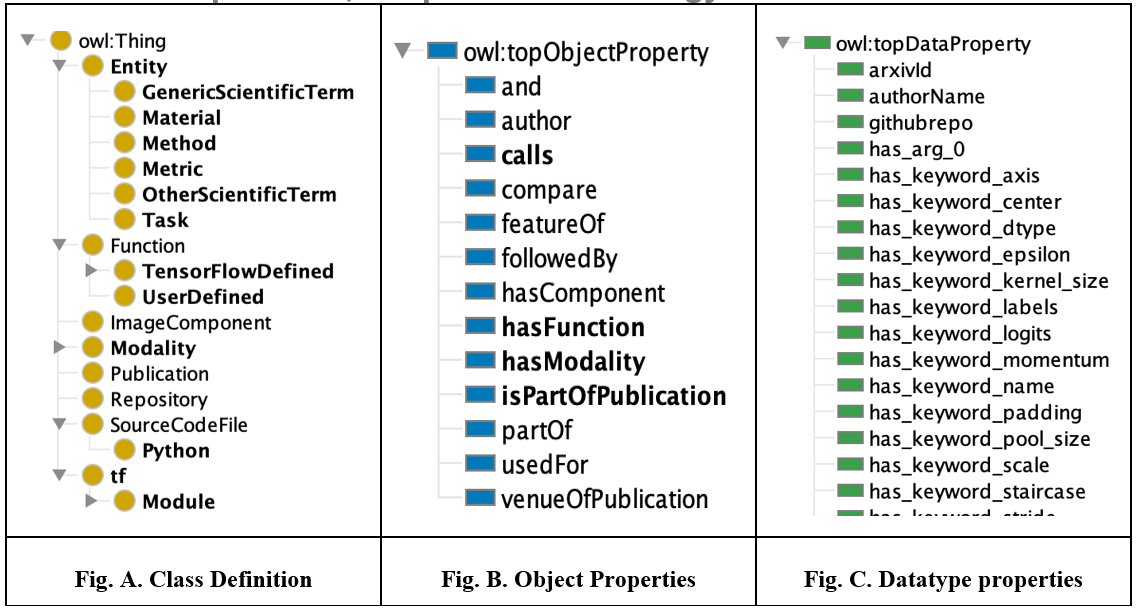
****

Figure 7: Hierarchical snapshots of code2graph ontology

In the pipeline of the Lightweight approach, RDF graph generation will start right after the call tree is created. The algorithm will determine each of the function calls in the call tree to be either *UserDefined* or *TensorFlowDefined* by performing the function call mapping procedure that we have described in the report of Milestone 5. According to the call tree structure, the corresponding relations (such as *calls* and *followedBy*) between functions will then be added. Once the process is finished, all the entities can be viewed in the format shown in Figure D. For all the RDF graphs that have been successfully extracted using the lightweight method, we have updated their format so that they can be processed and parsed by the Ontology viewer Protege[[11]](#footnote-12).

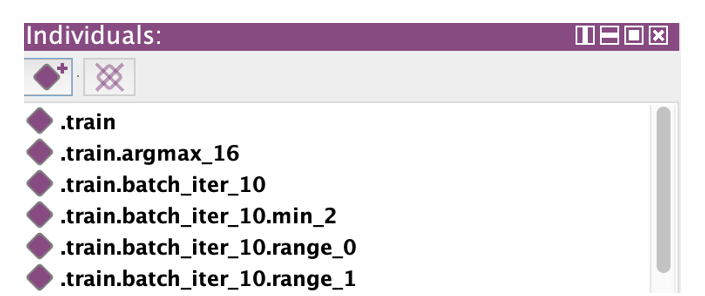


Figure 8: Format of the entities

## Future Work

We currently focus our effort on the following two tasks

**1) Expand and Filter the RDF Graphs**

In order to start training inference algorithms for RDF *graph completion*, we still need to extract more RDFs. Besides, we will also cooperate with the Siemens group on aligning the lightweight RDF graphs to the global ontology definition. Mapping to the ontology will be crucial for creating the knowledge base and effectively aligning it with RDF graphs extracted from the other modalities (i.e. text2graph and image2graph).

**2) Fine-Tune the Hyperparameters and Apply More KGE Methods**

Since Milestone 6, we have already built a pipeline of KGE methods that can process the RDF graphs generated from both approaches and subsequently train the embeddings for all entities and relations in the graphs, performing missing entity prediction in the level of triples. The next step will be utilizing the pipeline to fine-tune the results by either finding the golden set of hyperparameters or improving the quality of our datasets. We will also implement different Knowledge Graph Embedding (KGE) methods and compare the results between them. In addition, we will be studying more KGE methods that consider multi-hop relation paths or entity types so that the performance of predicting the missing entities on the RDF graphs can be further improved.

One of the approaches we developed involves the use of *temporal knowledge graphs* to capture context information of the triples. Instead of naming the subject and object nodes with sequential and hierarchical information, we can encode that information as *“time”* in the temporal graphs. In this approach, time will be appended to the original triples: *subject, predicate, object, time*.

Another approach is to capture sequential and hierarchical information in the edges (relations) of the graph. Currently, the number of entities is much greater than the number of relations. We hope to improve the performance of KGE methods by increasing the number of unique relations. In addition, we will explore the *auto-encoder architecture* for completing the head or tail entities when either is missing, predict relations given the two entities in the RDF triple, and create a sequential model to capture the RDF structure beyond first-order logic. This embedding will eventually aid in improving the code knowledge graph inferred by the team.

1. http://brat.nlplab.org [↑](#footnote-ref-2)
2. http://nlp.cs.washington.edu/sciIE/ [↑](#footnote-ref-3)
3. https://cso.kmi.open.ac.uk/home [↑](#footnote-ref-4)
4. This information is actually collected from our annotated (.ann) files by the script collect\_brat\_vocabulary.py [↑](#footnote-ref-5)
5. This information is collected by a script collect\_uwa\_vocabulary.py. [↑](#footnote-ref-6)
6. CSO.3.1.owl [↑](#footnote-ref-7)
7. https://www.snorkel.org/ [↑](#footnote-ref-8)
8. <https://arxiv.org/abs/1506.01497> [↑](#footnote-ref-9)
9. <https://arxiv.org/abs/1608.00507> [↑](#footnote-ref-10)
10. Which is the abbreviation for Fully Connected layers, and are usually used at the end of many deep learning architectures [↑](#footnote-ref-11)
11. <https://protege.stanford.edu/> [↑](#footnote-ref-12)