Capstone Project: Understanding 'Things' using Semantic Graph Classification

Rahul Parundekar

Machine Learning Nanodegree, Udacity
rparundekar@gmail.com
https://github.com/rparundekar

Abstract. The abstract should summarize the contents of the paper and should contain at least 70 and at most 150 words. It should be written using the *abstract* environment.

Keywords: Ontology, Semantic Web, Graph Kernels, Graph Classification, Deep Learning

1 Definition

1.1 Project Overview

The world around us contains different types of things (e.g. people, places, objects, ideas, etc.). Predominantly, these things are defined by their attributes like shape, color, etc. These things are also defined by their relationships with other things. For example, Washington D.C. is a place and U.S.A is a country. But they have a relationship of Washington D.C. being the capital of U.S.A., which adds extra meaning to Washington D.C. This same role is played by Paris for France.

The Semantic Web is an extention of the Knowledge Representation and Reasoning topic within Artificial Intelligence that aims at representing such things and their attributes and relationship using symbols at web scale and enabling the agent to reason about them. It is defined as "an extension of the current Web in which information is given well-defined meaning, better enabling computers and people to work in cooperation"[2]. A common data structure used for the representation of data in the Semantic Web is graphs. In these semantic graphs the nodes, properties, and edges of graphs are very well suited to describe the things, their attributes, and their relationships of things in the domain. A few example domais where such graphs are used are:

- Linked Data - The web-scale semantic data graph that is part of the Semantic Web[5].
- Spoken systems - the output of Natural Language Processing is a parse tree [14].
- Social networks are graphs [1].

- Scene recognition - High level semantic information in images are graphs of arrangements of things can be a graph [14].
- Virtual & Augmented Reality environments can be represented as a semantic graphs [10].

Project Motivation - Type Identification in a Semantic Graph: The project is motivated by our assumption that if an Agent is able to classify things by understanding its attributes and relationships into semantic types, we could in the future generalize it to an Agent that can act on the meaning of the things. Some examples applications for domains above are:

- Linked Data - Agents can understand and automatically assist users at web scale [2].
- Spoken systems - Understanding user intent by Virtual Assistants like Siri, Alexa, etc. for home automation [16].
- Predicting and Recommending links in Social Networks [1].
- Scene recognition - Urban scene understanding and its possible outdoor applications like understanding traffic, etc. [3].
- Virtural Assistants like Mara [13] in Virtual & Augmented Reality environments.

Dataset: We use DBpedia[1] as an exemplary dataset as a starting point to study Semantic Graph Classification. DBpedia is a large-scale knowledge base extracted from Wikipedia[9]. It contains structured information extracted out of Wikipedia (e.g. page links, categories, infoboxes, etc.)[8]. The semantic data in DBpedia can be represented as a graph of nodes and edges. In this case, the nodes are *things* (i.e. entities) and the edges are links/relationships between the *things*. Each *thing* has one or more **types** & **categories** associated with it. The user community creating DBpedia maintains an Ontology[2] that specifies these types of each of the *things*.

TODO: Show some graphs for dbpedia

We use the subset of DBpedia[3], which was generated from the March/April 2016 dump of Wikipedia. In this dataset, the *things*, their attributes and their relationships are extracted from the info-boxes (`infobox_properties_en.ttl`) using and the DBpedia Ontology (`dbpedia_2016-04.owl`). We also use the types associated with the *things*(`instance_types_en.ttl` & `yago_types.ttl`) and the categories (`article_categories_en.ttl`) that they belong to.

Our goal is to try and estimate the types and categories of the *things* from their attributes and relationships. For example, if you look at examples of categories in DBpedia, Achilies has been put into the categories - demigods, people

[1] http://wiki.dbpedia.org/

[2] An Ontology is defined as a formal specification of the types, properties, and relationships of the entities that exist for a particular domain. In other words, it is the schema definition of the semantic data.

[3] http://wiki.dbpedia.org/downloads-2016-04

of trojan war, characters in Illead, etc. What makes him part of those categories? Can we learn the definitions of these based on the attributes and relationships of Achilies?

Other Approaches: TODO: Explain the approaches briefly

1.2 Problem Statement

Traditionally, type inferencing in semantic data has been done with inference engines (e.g. OWL-DL uses SHIQ description logics [6] for inferences). However, as pointed out in Paulheim et. al [12], the actual Semantic Web data in the wild contains a lot of noise. Even a single noisy instance along with the inference rules can break the entailements in the types, or can add new entailments that may be incorrect. We thus need a robust way for identifying the types of the *things* in the Semantic Web.

In addition to the noise, the Semantic Web also makes an *Open World Assumption* [4]. In the *Open World Asumption* the Agent cannot assume that some proposition (i.e. some fact in the world) is **NOT TRUE** because it is `false`. Since the Agent may not have full visibility into the data, it may think the proposition to be `false` when it actually might be `true`. In classic Machine Learning on the other hand, typically the feature is known to be `true` or `false` depending on whether it is **actually `true` or `false`** (i.e. ground truth is known). Our type classification should also consider this assumption.

Problem Definition To achieve robustnessand to overcome the *Open World Assumption*, we need to train a classifier using existing data so that we can create a generic model that can be used for classifying unseen data. A Multi-class classifier in Machine Learning is used to create a model to classify data into one of multiple classes. However, in DBpedia, one *thing* can belong to more than one classes. The Canadian Institute for Advanced Research's `CIFAR-100` dataset is another example where the data can have more than one classes associated with it[7]. To perform such classification, we need to create a Multi-label classifier[17].

"Given the Semantic Graph for *things* in DBpedia (containing their attributes and relationships with other *things*), create a Multi-label classifier using Machine Learning techniques that can provide a robust type inferencing mechanism in the presence of noisy data & the inherant *Open World Assumption* made by the Semantic Web."

Proposed Approach We classify the *things* in DBpedia and identify their types and categories based on the semantic graph of their attributes and their relationships. While the dataset is a Semantic Graph, the classic algorithms in Machine Learning deal with feature vectors (e.g. the numerical features used to represent the object, etc.) and are aimed at discriminating between different inputs to those features to identify the target type/s. So, in order to perform

machine learning, we need to extract features for the things in the graph & also create the target multi-label vectors to be used for training & testing.

We extract features for each thing in DBpedia (hereafter called as an *individuals*) using a Random Walk approach [18]. For each random walk, we start at the individual in the Semantic Graph. We can choose a step randomly among different kind of steps - note the attribute / relationship / incoming relationship presence and stay on same node OR note the relationship / incoming relationship name and traverse to adjacent node. The sequence of steps noted down for the walk together form the extracted feature.

By varying the length, the number of walks, and the type of steps taken, we can extract different kind of features. More number of walks give more number of features, which means more data to classify with. Longer length of walks result in longer sequence of steps, which means very specialized descriptions of the individual. And finally, different step types adds to the complexity of descriptions.

Meanwhile, each individual has one or more types associated with it. We also extract these target classes. Our target classes for each individual can come from three sources - types from the DBpedia Ontology, DBpedia categories and types from the Yago [15] Ontology (all three downloaded from the source mentioned in Section 1.1 / Dataset). For each of these sources, we create multi-label target vectors dataset such that the value for the label for a type t in the vector for each individual is `1` if the individual is an instance of type t, or else is `0`.

Once we extract the features and the target types, we can then perform the Multi-label classification. For each of the three sources, we vary the number of walks, length of the walks and the type of the steps taken to generate the datasets and then use Deep Learning to perform the Multi-label classification.

We compare the effect of varying the random walk parameters with out chosen metrics (see below) and investigate some different Fully Connected Deep Neural Network architectures. We also compare our results with a baseline using simple Logistic Regression, and also with SDtype[12] and SLCN[11].

Expected Result: By comparing simple Logistic Regression and multiple Deep Learning architectures should confirm the soundness of our approach that Deep Learning approach gives more accurate results. It should also help us pick one Deep Learning architecture for the next comparison.

SDtype[12] and SLCN[11] both only use incoming relationships. We compare our classification performance with these two for both DBpedia Ontology types as well as Yago Ontology types. Our hope is that the random walk features will have comparable performane to these two approaches. At the least, we expect that using the same available data as the above two approaches, our Deep Learning architecture produces better results.

Metrics: The F_1-`score` metric is the harmonic mean of the `precision` and the `recall` of a model's classification[4].

[4] https://en.wikipedia.org/wiki/F1_score

$$F_1 = \frac{2 \cdot precision \cdot recall}{precision + recall}$$

where, `precision` is the ratio of textittrue positive classifications to the *predicted positive classifications* (i.e. *true positives* + *false positive* classifications) and `recall` is the ratio of *true positive classifications* to the *possible positive classifications* (i.e. *true positives* + *false negative* classifications). Compared to the `accuracy` metric, which is the ratio of *correct classifications* to *total classifications*, the F_1-score is a more robust mettric for model performance since it penalizes the model more heavily for both *false positive* as well as *false negative* classifications.

We have two main reasons for choosing F_1-score as our metric. First, since we have multiple-labels and a large dataset, the presence of the labels is sparse. Because of this, the number of *true negatives* wil overpower the calculation of the `accuracy` score, as they will dwarf the number of `false positives` and *false negatives*. Second, both SDtype[12] and SLCN[11] use the F_1-score metric, the performance of our model could be easily compared.

Since we are performing Multi-label classification, our output values will be a vector with length equal to the number of classes and having binary values with a value of 1 in position i when the individual belongs to class C_i. We need to specialize our definition of the F_1 score for such data. Like in the SLCN[11] paper, we we re-define `precision`, `recall`, and F_1`-score` as hP, hR and hF as follows:

$$hP = \frac{\sum_{i=1}^{|C|} tp_i}{\sum_{i=1}^{|C|} tp_i + fp_i}$$

$$hR = \frac{\sum_{i=1}^{|C|} tp_i}{\sum_{i=1}^{|C|} tp_i + fn_i}$$

$$hF = \frac{2 \cdot hP \cdot hR}{hP + hR}$$

where tp are the *true positive*, fp are the *false positive* and fn are the *false positive* classifications.

2 Generating Training & Testing Dataset from the Semantic Graph

As meentioned in Section 1.2, we need to first convert our Semantic Graph data and the types associated with the individuals in the graph into feature vectors and target vectors that can be used for Multi-label classification.

2.1 Creating Multi-label Target Vectors:

Our target classes for each individual can come from three sources - types from the DBpedia Ontology, DBpedia categories and types from the Yago [15] Ontology. To extract the multi-label target vectors, we first take the inner join of each of those three sources with the individuals available in the graph. We then create a multi-label target vector for each individual such that the value for the label for a type t in the vector is `1` if the individual is an instance of type t, or else is `0`. Once we identify the inner-join and the multi-label vectors, we remove the individuals from the Semantic graph that do not have type information. While this step reduces the possible relationships we may encounter with individuals for which there is no type information available, it also removes spurious individuals (e.g. things generated from Wikipedia articles that have no discernable types, things in external datasets for which we do not have type data, etc.). See the detailed algorithm in Figure 1.

2.2 Feature Extraction from the Semantic Graph:

We use a Random Walk approach to extract the features for each individual [18] in the reduced semantic graph. We start with the individual for which the features are to be extracted. We then create a list of possible steps that we can take from that node from one or more different types of steps available. With the DBpedia semantic graph, the type of steps available are - presence of an attribute, presence of an outgoing relationship, presence of an incoming relationship (where for all these three, after taking the step we will land on the same node), and a step on an outgoing relationship (where we will land on a different node with whom the node has a relationship). We then select one step from this list of available steps to land on the next node. We can then take the next step. By taking l such steps we form a path. This random walk of length l then becomes one feature for our classification. By performing n such random walks starting at one instance, we can extract upto n unique features for that instance. We repeat this for each instance, to extract features for the entire dataset. See the detailed algorithm in Figure 2.

We take some additional precautions in preparing the data. We make sure that the walks are not empty. This can happen if there are no attributes or relationships extracted for the individual from Wikipedia. Individuals where no walks were found are also removed. Also, since the next node after taking the the attribute, relationship and incoming relationship steps is the same node, it

```
def createTargetVectors(semanticGraph, typeFile):
    types = {}
    typeIndex = {}

    for individual, type in typeFile:
        # We need to ensure that top level types are removed,
        # since they will be true for all individuals.
        # Note: 'owl:Thing' and 'rdfs:Resource' are
        # standard compact URLs for
        # top level Semantic Web classes.
        if (type == 'owl:Thing' || type=='rdfs:Resource'):
            continue

        # Perform left-join by checking if individual
        # is present in semantic graph
        if (individual in semanticGraph):
            types[individual].append(type)
            if(type not in typeIndex):
              # By adding to type index here, we ensure that
              # all individuals at least have one type
               typeIndex[type]=len(types)-1

    targetVectors = []
    for individual in semanticGraph:
        # Perform right-join by checking if
        # individual has a type associated with it
        if (individual not in types):
            # Remove individual from graph
            # to complete inner-join
            del semanticGraph[individual]
        else:
            # Create the multi-label target vector such that
            # targetVector[type] = 1, if individual is
            # an instance of that type; and
            # targetVector[type] = 0, otherwise
            targetVector = [0] * len(types)
            for type in types[individual]:
                targetVector[typeIndex[type]] = 1
            targetVectors[individual].append(targetVector)

    return targetVectors

#Create the multi-label target vectors
targetVectors = createTargetVectors(semanticGraph, typeFile)
```

Fig. 1. Algorithm for Creating the Multi-label Target Vectors

```
def extractFeatures(semanticGraph, n, maxLength, stepTypes):
    walkIndex = {}
    walks = {}
    for individual in semanticGraph:
        # For n number of walks
        for i in range(n):
          # To choose the length, we compare 2 strategies:
          # Strategy 1: Fixed Length
          #   Here, the length is fixed to maxLength
          # Strategy 2: Variable length from 1 upto maxLength
          #   Here, the length is chosen with probability
          #       (maxLength-l+1) / (1+2+...+maxLength)
          #   This allows for shorter walks to be more dominant,
          #   since longer walks lead to sparser features.
          #   For example, chooseLength(2)
          #     returns l=1 with probability 2/3
          #     returns l=2 with probability 1/3
           l = chooseLength(maxLength)
           walk = []
           currentNode=individual
           for step in range(l):
              availableSteps = []
              nextNodes = []
              # Create list of available steps from stepTypes
              # (Assume standard graph helper functions below)
              # 1. Attribute presence
              if ('attribute' in stepTypes):
                 for attr,value in currentNode.attributes():
                    availableSteps.append('hasAttr_' + attr)
                    nextNode.append(currentNode)
              # 2. Relationship presence
              if ('relationship' in stepTypes):
                 for rel, node in currentNode.links():
                    availableSteps.append('hasRel_' + rel)
                    nextNode.append(currentNode)
              # 3. Incoming relationship presence
              if ('incoming' in stepTypes):
                 for rel, node in currentNode.incomingLinks():
                    availableSteps.append('hasInRel_' + rel)
                    nextNode.append(currentNode)
              # 4. Relationship step
              # If l=1, then this is same as 2 \& 3,
              # and so we add these only if l>1
              if ('step' in stepTypes && l>1):
                 for rel, node in currentNode.links():
                    availableSteps.append(rel + '->')
                    nextNode.append(node)
                 for rel, node in currentNode.incomingLinks():
                    availableSteps.append (rel + '<-')
                    nextNode.append(node)

             # Take a step randomly
             stepIndex = randInt(0,len(availableSteps)-1)
             step = availableSteps[stepIndex]
             nextNode = nextNodes[stepIndex]
             walk.append(step) #Append to the walk
             currentNode = nextNode #Move to next node
    return walks

#Extract features
extractedFeatures = extractFeatures(semanticGraph, n,
           maxLength, stepTypes)
```

Fig. 2. Algorithm for Extracting Features Using Random Walks

may introduce multiple features with the same steps but different orders. To eliminate this issue, we sort the steps taken at the same node in a lexicographic order before adding their sequence to the walk.

Important Note on Open vs. Closed World Assumption It is important to note that the meaning of the feature values, which are either `0` or `1`, here is different than classic Machine Learning. In classic Machine Learning all the feature values are known to be `0` or `1` before hand. That is, the value of the feature is `1` if the feature is `present`, and else it is `0`. With the Random Walk method, however, the meaning changes. The feature is `1` if the random walk was `detected`. But the feature may be `0`, even if the feature is `present` but `not detected`. We feel this nuance, which is similar to making an *Open World Assumption*, is an important distinction to classic Machine Learning, which makes a *Closed World Assumption*. Fortunately, this is not too bad, since by default the Semantic Web makes an Open World Assumption.

Another thing to note here is that even if we ask our algorithm to return `n` number of walks, if some of the walks are traversed multiple times, then the total number of features extracted may not be `n`.

3 Analysis

After extracting the feature vectors and the target vectors , we performed analysis to explore the data, identify the algorithms to create the classifier, and highlight the benchmark/s against which we will compare the performance of our classifier.

3.1 Data Exploration

Data Source As mentioned earlier, the source of our data is a subset of DBpedia[5], which was generated from the March/April 2016 dump of Wikipedia. While the data is available for multiple languages (English, French, German, etc.) and multiple RDF formats (Turtle, N-Triples, etc.)[6], we restrict ourselves to the English dumps in N-Triple format. The subset of data we use can be described as two parts:

- **The properties file:** The main semantic data (attributes and relationships) that we use for our Semantic Graph are the facts extracted from Infoboxes in Wikipedia. We use the `infobox_properties_en.ttl` file as the source for this data.
- **The type files:** Our target classes for each individual can come from three sources - types from the DBpedia Ontology, DBpedia categories and types from the Yago Ontology. At the end of our paper, we present the performance of our algorithm for each of these sources. The source files used for

[5] http://wiki.dbpedia.org/downloads-2016-04
[6] https://en.wikipedia.org/wiki/Resource_Description_Framework#Serialization_formats

these sources are `instance_types_en.ttl`, `article_categories_en.ttl` & `yago_types.ttl`, respectively. We also use the DBpedia ontology (`dbpedia_2016-04.owl`) to infer the parent types for those in `instance_types_en.ttl`.

The original number of facts in each of these sources are shown in Table 1.

Table 1. Number of facts

Source file	Number of facts (#triples)
`infobox_properties_en.ttl`	30,024,094
`instance_types_en.ttl`	5,214,242
`article_categories_en.ttl`	22,583,312
`yago_types.ttl`	57,879,000

.

Reduced Semantic Graphs After Creating Target Vectors As explained in the algorithm in Section 2.1, we take the inner-join of the type data and the semantic graph to reduce the number of instances and remove unnecessary individuals without types. For the three type sources and the properties file, we have three reduced Semantic Graphs - `DBpedia-OntologyTypes`, `DBpedia-Categories`, `DBpedia-YagoTypes`. The statistics about these are described in Table 2 and Table 3.

Table 2. Total Number of Individuals and Average Number of Attributes, Relationships & Incoming Relationships

.

Semantic Graph	Total # individuals	Average # attributes	Average #relationships	Average # incoming relationships
`DBpedia-OntologyTypes`	3.18M	5.78	3.58	2.74
`DBpedia-Categories`	1	1	1	1
`DBpedia-YagoTypes`	1	1	1	1

Traing & Testing Dataset from Extracted Features and Target Vectors From the above reduced Semantic Graphs, we then use the algorithm described

Table 3. Average Number of Distinct Attributes, Relationships & Incoming Relationships

Semantic Graph	Average # attributes	Average #relationships	Average # incoming relationships
DBpedia-OntologyTypes	4.85	2.30	0.425
DBpedia-Categories	1	1	1
DBpedia-YagoTypes	1	1	1

in Section 2.2 to extract features for classification. We create various training and testing datasets to explore the capability of our approach. We do not describe these datasets here, and instead choose to describe them in the next section and present their statistics as we come across them in the following sections.

3.2 Algorithms & Techniques

3.3 Exploratory Visualizations

3.4 Benchmarks

References

1. Backstrom, L., Leskovec, J.: Supervised random walks: predicting and recommending links in social networks. In: Proceedings of the fourth ACM international conference on Web search and data mining. pp. 635–644. ACM (2011)
2. Berners-Lee, T., Hendler, J., Lassila, O., et al.: The semantic web. Scientific american 284(5), 28–37 (2001)
3. Cordts, M., Omran, M., Ramos, S., Scharwächter, T., Enzweiler, M., Benenson, R., Franke, U., Roth, S., Schiele, B.: The cityscapes dataset. In: CVPR Workshop on the Future of Datasets in Vision. vol. 1, p. 3 (2015)
4. Drummond, N., Shearer, R.: The open world assumption. In: eSI Workshop: The Closed World of Databases meets the Open World of the Semantic Web. vol. 15 (2006)
5. Heath, T., Bizer, C.: Linked data: Evolving the web into a global data space. Synthesis lectures on the semantic web: theory and technology 1(1), 1–136 (2011)
6. Horrocks, I., Patel-Schneider, P.F., Van Harmelen, F.: From shiq and rdf to owl: The making of a web ontology language. Web semantics: science, services and agents on the World Wide Web 1(1), 7–26 (2003)
7. Krizhevsky, A., Hinton, G.: Learning multiple layers of features from tiny images (2009)
8. Lehmann, J., Isele, R., Jakob, M., Jentzsch, A., Kontokostas, D., Mendes, P.N., Hellmann, S., Morsey, M., van Kleef, P., Auer, S., Bizer, C.: DBpedia - a large-scale, multilingual knowledge base extracted from wikipedia. Semantic Web Journal 6(2), 167–195 (2015), http://jens-lehmann.org/files/2014/swj_dbpedia.pdf

9. Lehmann, J., Isele, R., Jakob, M., Jentzsch, A., Kontokostas, D., Mendes, P.N., Hellmann, S., Morsey, M., Van Kleef, P., Auer, S., et al.: Dbpedia–a large-scale, multilingual knowledge base extracted from wikipedia. Semantic Web 6(2), 167–195 (2015)
10. Lugrin, J.L., Cavazza, M.: Making sense of virtual environments: action representation, grounding and common sense. In: Proceedings of the 12th international conference on Intelligent user interfaces. pp. 225–234. ACM (2007)
11. Melo, A., Paulheim, H., Völker, J.: Type prediction in rdf knowledge bases using hierarchical multilabel classification (2016)
12. Paulheim, H., Bizer, C.: Type inference on noisy rdf data. In: International Semantic Web Conference. pp. 510–525. Springer (2013)
13. Schmeil, A., Broll, W.: Mara-a mobile augmented reality-based virtual assistant. In: Virtual Reality Conference, 2007. VR'07. IEEE. pp. 267–270. IEEE (2007)
14. Socher, R., Lin, C.C., Manning, C., Ng, A.Y.: Parsing natural scenes and natural language with recursive neural networks. In: Proceedings of the 28th international conference on machine learning (ICML-11). pp. 129–136 (2011)
15. Suchanek, F.M., Kasneci, G., Weikum, G.: Yago: a core of semantic knowledge. In: Proceedings of the 16th international conference on World Wide Web. pp. 697–706. ACM (2007)
16. Tang, B.: The emergence of artificial intelligence in the home: Products, services, and broader developments of consumer oriented ai (2017)
17. Tsoumakas, G., Katakis, I.: Multi-label classification: An overview. International Journal of Data Warehousing and Mining 3(3) (2006)
18. Tsuda, K., Saigo, H.: Graph classification. Managing and mining graph data pp. 337–363 (2010)