Understanding 'Things' using Semantic Graph Classification

Rahul Parundekar

***Abstract*—The world around us contains different types of things (e.g. people, places, objects, ideas, etc.) that are defined by their attributes and relationship. To act automatically on such data, any software Agent needs to be able to infer the meaning of these things that form a Semantic Graph. Using DBpedia as an exemplary dataset, we create a robust type inferencing system in the presence of noisy data and the Open World Assumption on the Semantic Web. Our approach extracts features from the Semantic Graph using multiple Random Walks and then performs multi-label classification using Deep Neural Networks. This report presents our exploration, experimentation, and results of identifying DBpedia ontology types, categories and Yago ontology types for individuals in DBpedia. Our method consistently performs better than state-of-the-art type inferencing systems, like SDtype and SLCN, from which we conclude that Random Walk based feature extraction and multi-label classification is a promising approach in understanding things and contexts in domains that represent information as a Semantic Graph.**

I. Definition

A. Project Overview

1) Problem Domain: 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. They 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 the city.

Our domain for the project is the Semantic Web where information is represented with well-defined meaning, thus better enabling computers (Agents) and people to work in co-operation [2]. It is an extension 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 Agents to reason about them. Information in the Semantic Web is represented as Semantic Graphs, since the nodes, properties, and edges of graphs are very well suited to describe the things, their attributes, and their relationships with other things in the domain. Semantic Graphs are also popularly used in various other domains like Linked Data [8], Spoken Dialog Systems [23], Social Networks [1], Scene Understanding [23], Virtual & Augmented Reality [14], etc.

2) Project Origin: In such a setting, an Agent on the Semantic Web needs to be able to understand the information that it comes across. By inferring the type of the thing, it may be able to perform automated actions. 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 and context of the things. For example, such an Agent can be a virtual assistant [25] [22], an autonomous system [4] or other software[1].

3) Related Datasets: We use DBpedia[1] as an exemplary dataset as a starting point to study type inferencing in a Semantic Graph. DBpedia is an encyclopedic Semantic Graph of structured information extracted from Wikipedia (e.g. page links, categories, infoboxes, etc.) [13] [12]. For example, in the subset of DBpedia shown in Figure 1, we can see the individual for United_States (on the left) connected to Washington D.C. using the *capital* relationship.

Each *thing* in DBpedia has one or more **types** and **categories** associated with it. The user community creating DBpedia maintains an Ontology[2] that specifies these types of the *things*. We use the subset of DBpedia[3], which was generated from the March/April 2016 dump of Wikipedia. In this dataset, we use the *things*, their attributes and their relationships extracted from Wikipedia infoboxes (`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` and `yago_types.ttl`) and the categories (`article_categories_en.ttl`) they belong to. The types and categories associated with United States and Washington D.C. can be seen in Figure 4.

Our goal is to try and estimate the types and categories of such *things* from their attributes and relationships.

B. Problem Statement

Traditionally, type inferencing in the Semantic Web has been done with inference engines (e.g. OWL-DL uses SHIQ description logics [9] for inferences). However, as pointed out in Paulheim et. al [20], 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 entailments in the types or can add new entailments that may be incorrect. Thus, we need a robust way of identifying the types of the things in the Semantic Web. In addition to the noise, the Semantic Web also makes an *Open World Assumption* [6]. In the *Open World Assumption*, the Agent cannot assume that some proposition (i.e. some fact in the world) is **NOT TRUE** because it is `absent`.

1) Problem Definition: Given the Semantic Graph for *things* in DBpedia (containing their attributes and relationships with other *things*), we would like to create a classifier using

[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.

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

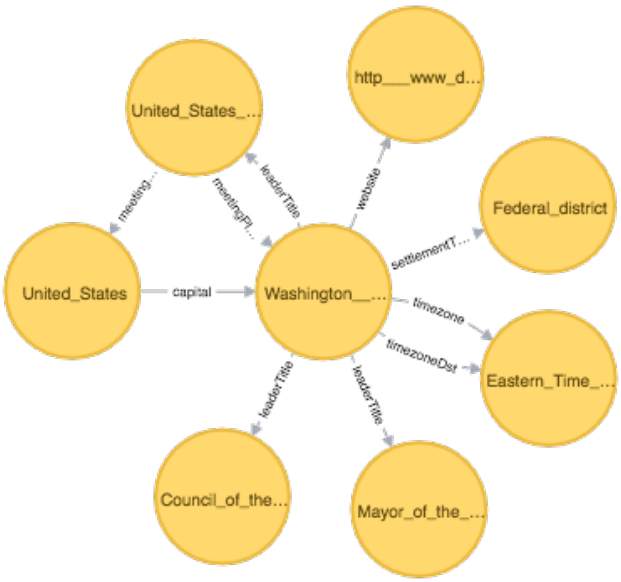

Fig. 1: Example neighborhood of Washington D.C. in the DBpedia Semantic Graph. The United_States individual is connected to it via the *capital* relationship.

.

Machine Learning techniques that can provide a type inferencing mechanism in the presence of noisy data and the inherent *Open World Assumption* made by the Semantic Web. Multi-class classifiers are used in Machine Learning to create models to classify data into ONE of the multiple classes. However, in DBpedia, one *thing* can belong to MORE THAN ONE class. To perform such classification, we need to create a Multi-label classifier [26][4]. There has been some work in combining random walks on graphs with Deep Learning. Perozzi et. al [21], for example, use a social network graph for link prediction and recommendation. It is not, however, a semantic graph.

2) Outline of Tasks: We use a four-phased approach to solve the problem.

- **Feature Extraction Phase:** We extract multi-label target vectors from the type and category files and also extract feature vectors from the Semantic Graph to make it Machine Learning-ready using Random Walks [27]. These Random Walks extract features by taking a number of walks of varying length on the Semantic Graph and noting down each attribute or relationship that we step across.
- **Feature Exploration Phase:** Next, we explore the type of steps available and their contribution to the classes using Logistic Regression and Fully-connected Neural Networks.
- **Model Experimentation Phase:** We then experiment with different walk lengths and types of data for our Random Walks, using one Fully-connected Neural Network structure that we select based on the previous phase. Once we understand the best performing step-types, lengths and number of steps for Random Walks, we proceed with the final phase.
- **Type Inferencing Phase:** In this last phase, after refining the model, we perform the final classification by inferring the types and categories of DBpedia using multi-label classification.

Our expected result is that the model is able to perform multi-label classification and outperform benchmarks SDtype[20] and SLCN[16] (discussed later) for type and category inference in DBpedia.

3) Metrics: The F_1-`score` is the harmonic mean of the `precision` and the `recall` of a model's classifications[5]. We have two main reasons for choosing F_1-score as our metric. First, since we have a large dataset with relatively few multi-labels class types, the presence of the labels is sparse. Because of this, the number of *true negatives* will overpower the calculation of the `accuracy` score, as they will dwarf the number of *false positives* and *false negatives*. Second, both SDtype [20] and SLCN [16] use the F_1-score metric, the performance of our model could be easily compared. Since we are performing Multi-label classification, we specialize our definition of the `precision` and `recall` score as follows:

$$precision = \frac{\sum_{i=1}^{|C|} tp_i}{\sum_{i=1}^{|C|} tp_i + fp_i}, recall = \frac{\sum_{i=1}^{|C|} tp_i}{\sum_{i=1}^{|C|} tp_i + fn_i}$$

where C is the number of target classes, tp are the *true positive*, fp are the *false positive* and fn are the *false positive* classifications.

II. Feature Extraction Phase

We 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.

A. 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 [24] 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

[4]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 [11]

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

```
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' or 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
```

Fig. 2: Algorithm for Creating the Multi-label Target Vectors

have no discernable types, things in external datasets for which we do not have type data, etc.). See the algorithm in Figure 2 for more detail.

B. Feature Extraction from the Semantic Graph:

We use a Random Walk approach to extract features for each individual [27] in the reduced semantic graph. We start our walk on 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 four type of steps available are - presence of an attribute, presence of an outgoing relationship, presence of an incoming relationship (for all these three, after taking the step we will land on the same node), and a step on an incoming or outgoing relationship (here, 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 up to n unique features for that instance. We repeat this for each instance, to extract features for the entire dataset. See the algorithm in Figure 3 for more detail.

```
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 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():
                        if(('hasAttr_' + attr) not in availableSteps):
                            availableSteps.append('hasAttr_'+attr)
                            nextNode.append(currentNode)
               #2. Relationship presence
               if ('relationship' in stepTypes):
                    for rel, node in currentNode.links():
                        if(('hasRel_'+rel) not in availableSteps):
                            availableSteps.append('hasRel_' + rel)
                            nextNode.append(currentNode)
               # 3. Incoming relationship presence
               if ('incoming' in stepTypes):
                    for rel,node in currentNode.incomingLinks():
                        if(('hasInRel_' + rel) not in availableSteps):
                            availableSteps.append('hasInRel_'+rel)
                            nextNode.append(currentNode)
               #4. Relationship step
               #If l=1, then this is same as 2 and 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
          #Add to the walks
          if walk not in walks:
              walks.append(walk)
    return walks
```

Fig. 3: Algorithm for Extracting Features Using Random Walks

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 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.

TABLE I: 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

.

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, the feature value is 0 if the feature is absent or 1 if it is present. 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.

III. Analysis

A. Data Exploration

1) Data Source: As mentioned earlier, the source of our data is a subset of DBpedia[6], 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.)[7], we restricted ourselves to the English dumps in the N-Triple format. The subset of data we used 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 these sources are instance_types_en.ttl, article_categories_en.ttl and 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 I.

2) Reduced Semantic Graphs After Creating Target Vectors: As explained in the algorithm in Section II-A, 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 II[8].

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

[7]https://en.wikipedia.org/wiki/Resource_Description_Framework#Serialization_formats

```
Examples Types for United_States:
                Settlement
                Country
                PopulatedPlace

Examples for Washington__D_C_:
                Place
                Settlement
                Location
                http___schema_org_Place

Examples for Aristotle:
                Philosopher
                Person
                http___xmlns_com_foaf_0_1_Person
```

Fig. 4: Examples of Types in the DBpedia-OntologyType Dataset

TABLE II: Total Number of Individuals and Average Number of Attributes, Relationships and Incoming Relationships

Semantic Graph	# individuals	Avg. # attributes	Avg. inc. #relationships	Avg. # relationships	# Types / Categories
DBpedia-OntologyTypes	3.18M	5.78	3.58	2.74	526
DBpedia-Categories	2.26M	6.25	2.89	2.11	10999
DBpedia-YagoTypes	2.78M	5.97	4.21	3.08	2083

Figure 4 shows the example types for select individuals in the DBpedia-OntologyTypes dataset.

The datasets were split into batches of upto 5000 individuals in each (lesser, if unnecessary individuals were removed). The batches were split into 80% training - 20% testing split[9].

B. Algorithms and Techniques

After the Feature Extraction Phase, we proceed with the Feature Exploration, Model Experimentation, and Type Inference phases.

1) Feature Exploration Phase: In performing our random walk, we have four types of steps we can take from any individual:

- **Attribute presence:** where we pick randomly among the attributes present, note the attribute name, and stay on the same node (see Figure 5 for examples).
- **Relationship presence:** where we pick randomly among the relationships present, note the relationship name, and stay on the same node (see Figure 6 for examples).
- **Incoming Relationship presence:** where we pick randomly among the incoming relationships present, note the incoming relationship name, and stay on the same node (see Figure 7 for examples).
- **Relationship Step:** where we pick randomly among the incoming and outgoing relationships and move to the node that the relationship connects our individual to.

[8]We limited classes to only those with a support of at least 200 individuals for DBpedia-Categories and Dbpedia-YagoTypes as explained in Section IV-C2

[9]Since this is an exploratory paper and first of its kind, the real world applications are not yet clear for these datasets and comparisons are not available. Accordingly, use of separate validation and final test datasets was found

```
Examples for United_States:
        Features: (upto 5 of 15)
                has_imageFlag
                has_ethnicGroups
                has_populationEstimate
                has_commonName
                has_nationalAnthem

Examples for Washington__D_C_:
        Features: (upto 5 of 17)
                has_elevationMinFt
                has_populationRank
                has_areaCode
                has_populationMetro
                has_areaWaterSqMi

Example for Aristotle:
        Features: (upto 5 of 4)
                has_name
                has_mainInterests
                has_birthDate
                has_deathDate
```

Fig. 5: Examples of Features of Attribute Presence Step-Type

```
Examples for United_States:
        Features: (upto 5 of 4)
                hasRel_demonym
                hasRel_capital
                hasRel_largestCity
                hasRel_leaderName

Examples for Washington__D_C_:
        Features: (upto 5 of 1)
                hasRel_leaderTitle

Examples for Aristotle:
        Features: (upto 5 of 2)
                hasRel_influences
                hasRel_deathDate
```

Fig. 6: Examples of Features of Relationship Presence Step-Type

For this exploration, we use only the `DBpedia-OntologyTypes` dataset. By understanding the effect of the features and their effect on different Fully-connected Neural Network structures, we can select one network structure and select the parameters of random walks in the next phase.

We use three Fully-connected Neural Network structures for exploration. Neural networks are connectionist systems that are loosely analogous with 'neurons' in the brain. A fully-connected neural network contains different layers of such connected 'neurons', each layer providing some specific purpose in mapping the inputs to the outputs:

- Input, hidden, and output layers: Each layer consists of multiple neurons that are fully connected to the neurons previous layer. These neurons act like individual logistic regressors. The feature vectors are the inputs to the first layer, while the number of neurons in the output layer is equal to the number of target vectors. The number of hidden layers determines the complex features that can be extracted, while the number of neurons in each layer defines the number of dimensions that one layer's inputs need to be converted into, which are also the number of features for the next layer. Generally, the deeper the neural network, the more complex features can be extracted. However, after a certain depth, the network may start overfitting as it learns to model the noise. We explore different depths of neural networks.
- Regularization Layers: To avoid overfitting, we can use the Dropout layer. While training, the Dropout layer between each hidden layer suppresses the inputs to the next layer. As a result, the model has to learn from redundant information but also learns to ignore complex relationships that lead to overfitting. We use a dropout layer after each layer with a dropout rate of 0.2 (i.e. 20% of neurons are suppressed).
- Activation Functions: Each neuron performs some linear mathematical function on its inputs and typically has a sigmoid function on its output. Despite increasing depth, the combination of linear functions can be proven to still be linear. To introduce non-linearity we need to modify its activation function. A ReLU has a simple non-linear function $r(x) = max(0, a)$, where a is the output of multiplying the weights by the input and adding bias. This allows the neural network to model higher order functions, thus increasing the capability of learning the function from the input to the output.
- Learning algorithm: Using each batch input data, we calculate the outputs of the neural network using its different layers. This output may be completely wrong. So we use a loss function (like binary cross entropy) to determine the loss. We take a small step in the differential in the loss function to reach the minima of the loss function, which in turn maximizes prediction accuracy. We propagate this loss to all the hidden layers and back to the input layers to update the weights so that they reduce their losses. This batch-wise computation of the loss and taking a step down the gradient of the loss funtion is called Stochastic Gradient Descent. By varying the length of the step in the direction of the gradient (i.e. adaptive gradient) neural networks have demonstrated to reach more optimal solutions with faster convergence.
- Normalization Layer: To accelerate the learning, we use a normalization technique at the output of each layer, before the activation function. By normalizing the outputs, we prevent the numbers from being too large or too small so that the values are more comparable on the next layer.
- Other hyper-parameters: The above can be considered as hyper-parameters of the neural network, which we can vary to try different network structures and determine what type of network structures can best model

```
Examples for United_States:
        Features: (upto 5 of 153)
                hasInRel_almaMater
                hasInRel_channel
                hasInRel_hometown
                hasInRel_based
                hasInRel_label

Examples for Washington__D_C_:
        Features: (upto 5 of 64)
                hasInRel_regions
                hasInRel_unit
                hasInRel_origin
                hasInRel_restingPlace
                hasInRel_recorded

Examples for Aristotle:
        Features: (upto 5 of 3)
                hasInRel_influenced
                hasInRel_influences
                hasInRel_mainInterests
```

Fig. 7: Examples of Features of Incoming Relationship Presence Step-Type

the function from inputs to the outputs. Batch-size is one other hyper-parameter that we tune. The number of input examples used in the mini-natch determines the gradient and the length of the step we take. If we take too large a step, we may overshoot and lead to a less optimal solution, while if we take really small batches, then our step may not be in the right direction of the gradient to minimize. Weight initialization is another hyper-parameter, where the initial weights define faster or slower convergence - if all the weights are same (say, 1) then the gradient step will affect all of them the same. With a right distribution (e.g. normal distribution) of the weights, each neuron adjusts its weight differently, leading to better learning.

While the above is not an exhaustive hyper-parameter list, it is sufficient for our description below and the Refinement Section IV-C2.

For all three neural networks, the number of inputs in the input layer equals the number of features and the number of neurons in the output layer equal the target variable. Upon experimentation, the fully connected layers were initialized using Xavier initialization (i.e. Glorot Normal initialization [7]), as it lets the signal reach deeper into the network since the weights are not too small or too large to start in dead or saturated regions. Based on some preliminary exploration we trained each for 5 epochs as the F_1-score plateaus after that, or in the case of the 8 layer network, starts overfitting. For learning, we use Adam optimizer with the Nestrov accelerated gradient approach [5] (Nadam) since it is an adaptive gradient method and our data is sparse. We use 'binary cross-entropy' loss function for learning.

Except for the Simple Logistic Regression classifier, we have the following additional structure features. We use both Nadam optimizer and Batch Normalization allows for accelerated learning. To avoid overfitting, especially since the data is sparse and the network may learn very specialized definitions, we use a Dropout layer. The dropout rate was selected as 0.2 after multiple experimentations showed that rate of 0.4 was too slow, and given the large data (about 3 Million samples). Other hyper-parameters (e.g. learning rate) were kept to default values in Keras [3].

- **Logistic Regression:** The simple logistic regression network has only one fully-connected layer. This will help us establish a baseline with respect to the other two structures and provide us a sanity check. Figure 8 shows the simple network architecture.

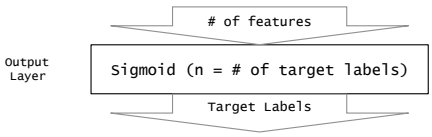

Fig. 8: Simple Logistic Regression Network Structure

.

- **4 Layer Fully Connected Neural Network:** The first deep fully-connected neural network contains 4 layers including the input and output. The shape and sizes of the layers can be seen in Figure 9.

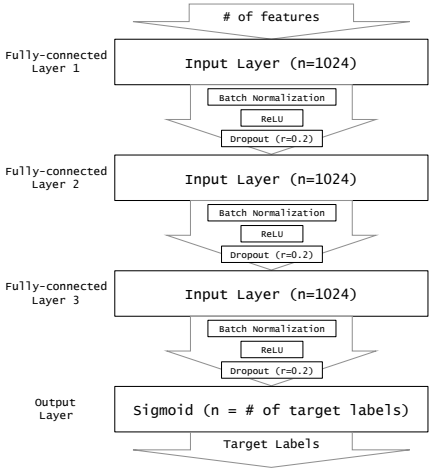

Fig. 9: 4 Layer Deep Neural Network Structure

.

- **8 Layer Fully Connected Neural Network** The second deep fully-connected neural network contains 8 layers including the input and output. Compared to the 4 layer network, we chose to explore another structure with fewer neurons per layer but deeper strucutre. The shape and sizes of the layers can be seen in Figure 10.

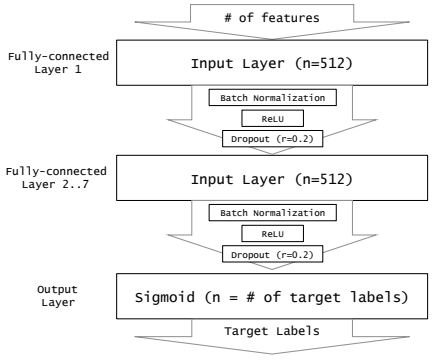

Fig. 10: 8 Layer Deep Neural Network Structure

.

Our exploration of the feature space available is described in detail in Section III-C. Based on the conclusion of these exploration experiments, we select one network structure for the next phase.

2) Model Experimentation Phase: We choose the Fully-connected Neural Network architecture with 4 Layers as our starting point for this phase, based on the conclusion of the previous phase (see Section III-C). We tried out diffferent improvements before settling down on the structure shown in Figure 11.

- We increased the depth as much as we could, but noticed that after 6 layers the network starts overfitting the data. We concluded this based on the divergence between the training and testing F_1-score after initial simultaneous increase.
- We tried layers with 2048 neurons as well. However, we noted that as we start random walks of length longer than one step, our number of features grows very rapidly

and even goes into millions. With feature vectors of that length, given the memory of the GPU (4GB), we restricted to 1024 neurons to not run into memory overflow.
- We also tried running our experiments for 10 epochs. However, we noticed little to no improvement for most of the experiments, and the training time for the larger datasets was too prohibitive to run for 10 epochs. Based on this, we restrcted to using 5 epochs.

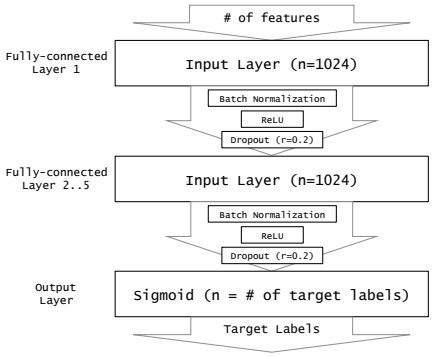

Fig. 11: 6 Layer Deep Neural Network Structure

.

After having selected a fully-connected Neural Network structure, we vary the different parameters to the `extractFeatures` function with different lengths, different number of walks and different types of steps. By comparing the classification performance of the different experiments, we can select the most appropriate parameters. For these experiments, we use only the `DBpedia-OntologyTypes` dataset. Here, we group the types of steps into two:

- **Step but stay on same node:** Steps of type attribute presence, relationship presence and incoming relationship presence.
- **Step and move across the relationship:** Step over any incoming and outgoing relationship and move to the next node.

The delails of the experiments and results are presented in the Section IV.

3) Type Inferencing Phase: Finally, after having selected the random walk parameters and the network structure, we perform the final classification for the `DBpedia-OntologyTypes`, `DBpedia-Categories` and `DBpedia-YagoTypes` datasets with some refinements to the network structure, pre-processing, etc. We describe this in Section V.

C. Exploratory Visualizations

Our exploratory visualizations come from the feature extraction phase. We use the `extractFeatures` function to extract features of four type of steps. We select incrementing lengths of 5, 10, and 25 for our random walks and explore the ability of the 3 structures described in Section III-B1 to learn to classify the `DBpedia-OntologyTypes` dataset. This dataset has a total of 3.18M individuals grouped into 638 batches. There are 529 target classes.

1) Attribute Presence: We select the attribute randomly from all the attributes that the individual has, and note the name of the attribute (and ignore the value) for the step. Table III shows the F_1 scores of the different structures at different lengths and different number of walks. We note that performance increases with more random walks and the structure with 4 layers performs the best. The maximum F_1-score we reach here is 0.8767.

TABLE III: Performance of the 3 structures using only the *attribute presence* step

# of random walks	# disitnct features	Logistic Regression	4 Layers fully connected	8 Layers fully connected
10	1993	0.8224	0.8596	0.8590
15	2006	0.8330	0.8687	0.8683
20	2014	0.8374	0.8733	0.8727
All	2020	0.8421	0.8765	0.8767

2) Relationship Presence: Similar to the attribute presence step-type, we also explore the classification ability of the relationship presence step-type. Table IV shows the relevant F_1 scores. Here too, we note that the structure with 4 layers performs the best. The maximum F_1-score we reach here is 0.8058.

TABLE IV: Performance of the 3 structures using only the *relationship presence* step

# of random walks	# disitnct features	Logistic Regression	4 Layers fully connected	8 Layers fully connected
10	1066	0.7720	0.7956	0.7995
15	1072	0.7751	0.8001	0.7981
20	1071	0.7749	0.8058	0.8007
All	1075	0.7743	0.8025	0.7989

3) Incoming Relationship Presence: Similar to the attribute presence type of step, we also explore the classification ability of the incoming relationship presence step. Table V shows the relevant F_1-scores. Here too, we note that the structure with 4 layers performs the best. The maximum F_1-score we reach here is 0.7619.

TABLE V: Performance of the 3 structures using only the *incoming relationship presence* step

# of random walks	# disitnct features	Logistic Regression	4 Layers fully connected	8 Layers fully connected
10	1043	0.6557	0.7619	0.7612
15	1072	0.6596	0.7597	0.7586
20	1071	0.6603	0.7563	0.7629
All	1075	0.6693	0.7476	0.7596

4) All 3: We also explore the classification ability of the all above three step-types. Table VI shows the relevant F_1 scores. The maximum F_1-score we reach here is 0.9235.

Figure 12 below shows the comparison of the learning of the four step-types for all possible features and the three structures (we do not show the ones with fewer random walks, due to lack of space).

TABLE VI: Performance of the 3 structures using all 3 step-types

# of random walks	# disitnct features	Logistic Regression	4 Layers fully connected	8 Layers fully connected
10	4014	0.8256	0.8987	0.8960
15	4092	0.8428	0.9097	0.9083
20	4110	0.8484	0.9158	0.9144
All	4169	0.8613	0.9235	0.9219

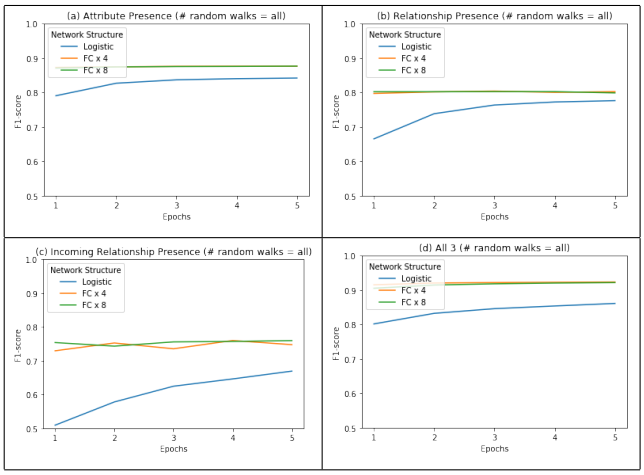

Fig. 12: Performance of the 3 structures for all features for all step-types

5) Conclusion: From the tables and Figure 12, we note that among the three types of steps, the *attribute presence* step by itself has the most information to identify the classes. Meanwhile, the maximum F_1-score we get is 0.9235 when we use all three step-types. The *incoming relationship presence* step scores the least at 0.7597. Also, among the three network structures, the Fully-connected Neural Network with 4 layers performs the best. And so we use that structure for the model experimentation phase.

D. Benchmarks

There has been considerable prior art in inferring types of semantic data on the Semantic Web [19][18].Most relevant of these are the SDtype[20] and SLCN[16][17] papers, which also try to perform type inferencing from Semantic Web data. For `DBpedia-OntologyTypes` SDtype achieves an F_1-score of 0.765 and SLCN[16] achieves a score of 0.847. While for the `DBpedia-YagoTypes` dataset, it is 0.666 and 0.702. It should be noted that both the SDtype and SLCN papers work on the 2014 DBpedia dump, unlike ours which is the 2016 dump.

1) Note on using attributes and relationship presence steps for classifying: As noted by both these papers, in original generation of DBpedia from Wikipedia, the types of the individuals were assigned using the infobox properties and the attributes and relationships were generated using a combination of that. Based on this, the authors do not use attribute presence and relationship presence for their classification as they claim the classification to be straight forward. And so, to compare with the benchmarks, we use only our algorithm with the relationship step only, having random walk length greater than 1.

```
Examples for United_States:
        Features: (upto 5 of 124)
                hasInRel_production, hasInRel_regionServed
                hasInRel_citizenship, hasInRel_label
                hasInRel_twin, has_percentWater
                hasInRel_production, hasInRel_recorded
                hasInRel_origin, hasInRel_sourceLocation

Examples for Washington__D_C_:
        Features: (upto 5 of 122)
                hasInRel_regionServed, has_areaTotalSqMi
                has_name, has_populationTotal
                hasInRel_beltwayCity, hasInRel_venue
                hasInRel_based, hasInRel_homeTown
                hasInRel_billed, hasInRel_terminusB

Examples for Aristotle:
        Features: (upto 5 of 42)
                hasInRel_influenced, has_mainInterests
                has_deathDate, has_mainInterests
                hasInRel_mainInterests, hasRel_deathDate
                hasRel_deathDate, has_mainInterests
                hasRel_influences
```

Fig. 13: Examples of Features of `Stay` Step Category of Walks of Length 2

For an overall accuracy, we consider a third benchmark set by using a combination of all 3 steps of attribute, relationship, and incoming relationship presence. To generate the third benchmark, we trained the Fully-connected neural network structure with 6 layers on the dataset containing 4169 features extracted with all three step types - attribute, relationship and incoming relationship presence. With 15 epochs, to achieve maximum F_1 score, we set this benchmark to be 0.9254. Since the dataset is a combination of all 3 step types its performance is better than all three individually.

IV. Methodology

In the Model Experimentation phase, we try to identify the right features to be extracted. We vary the length and number of walks, and choose from either step-types that stay on the same node or the relationship and incoming relationship step-type that moves to the adjacent node or both. We also use two strategies for selecting the length - fixed length of exactly 2 or 3, AND variable length of upto 2 or 3 with higher probability for lower lengths (as described in the `extractFeatures` function in Figure 3.

So in total, we have 18 variations:

- *Length of walks* - 2 or 3
- *Number of walks* - 25, 50 or 100
- *Category of steps* -
  - `Stay`: Step but stay on same node (Attribute, relationship and incoming relationship presence) (see Figure 13 for examples), OR
  - `Move`: Step on incoming or outgoing relationship but move to adjacent node (relationship step) (see Figure 14 for examples), OR
  - `Both`: All four types of steps can be performed - attribute, relationship, and incoming relationship presence or step on an outgoing or incoming relationship. (see Figure 15 for examples), OR
- *Length strategy* - Fixed or Variable length

```
Examples for United_States:
        Features: (upto 5 of 105)
                office<-birthPlace->
                citizenship<-placeOfDeath->
                leaderName->predecessor<-
                regionServed<-leaderName->
                mouthCountry<-mouthLocation->

Examples for Washington__D_C_:
        Features: (upto 5 of 83)
                leaderTitle->order<-
                stadium<-previous<-
                placeOfBirth<-director<-
                locale<-operator->
                nearestCity<-location->

Examples for Aristotle:
        Features: (upto 5 of 20)
                influences<-author<-
                mainInterests<-placeOfBirth->
                influences<-influences->
                deathDate->birthPlace<-
                influences<-influenced<-
```

Fig. 14: Examples of Features of `Move` Step Category of Walks of Length 2

```
Examples for United_States:
        Features: (upto 5 of 124)
                hasInRel_institution , nationality<-
                hasInRel_production , hasInRel_restingPlace
                hasInRel_firstAired , hasInRel_releaseDate
                hasInRel_ideology , region<-
                hasInRel_manufacturer , hasInRel_subdivisionName

Examples for Washington__D_C_:
        Features: (upto 5 of 122)
                hasInRel_governingBody , campus<-
                placeOfBirth<-birthPlace->
                hasInRel_residence , areaServed<-
                hasInRel_beltwayCity , hasInRel_city
                hasInRel_currentowner , birthDate<-

Examples for Aristotle:
        Features: (upto 5 of 74)
                influences->has_name
                has_deathDate
                hasRel_influences , has_mainInterests
                has_deathDate , deathDate->
                hasInRel_influences , hasRel_deathDate
```

Fig. 15: Examples of Features of `Both` Step Category of Walks of Length 2

A. Preprocessing

Since the feature extraction and target multi-label vector generation steps take care of data cleanup, and making sure that the inputs are binary, not much pre-processing was needed. However we soon realized we quickly ran out of memory space when the magnitude of the sparsity when we take random walks. The number of features can go in the millions. Even with a modest 1024 neurons in the first layer, we quickly run out of memory. So, we encode our inputs into something reasonable. Since the dataset is sparse and at max, we may have 100 features in each row, we decided to superimpose features. We select an input feature size of 8384 neurons (1024x8). With this limited space, our encoding function is:

$$X[i \mod 8384] = \lfloor i \div 8384 \rfloor + 1$$

We take additional steps to distribute the values equally in positive and negative numbers to help the network discriminate better. The absence of any random walk detected at *i*th location is encoded by 0. Since the encoding function is applied consistently to all values, we do not need any additional normalization or scaling.

B. Implementation

1) Feature Extraction Phase: Our feature extraction and target vector generation code was implemented in Java. Since the dataset was made available in `Turtle` RDF syntax, we used the Apache Jena Java library[10] to parse it. Initially, we loaded the data in Neo4J Graph database[11], since it seems the appropriate data structure. However, one complication we faced was that the creation of each random walk dataset in all phases was taking about 8-10 hours on a Macbook Pro with 16GB RAM. So we decided to optimize by holding the semantic graph in memory and performing the random walks. With this improvement, we achieved considerable speed up - we were able to create all datasets for Phase 1 and 2 in 16 hours. The target vector extraction code took about 1.5 hours on the same machine.

2) Feature Exploration, Model Experimentation, Type Inferencing Phases: We use Python for the implementation of all these three phases. The Deep Learning framework used was Keras[12] (v1.2.2) on top of TensorFlow[13] (v1.0.0). We developed our code in Python notebooks (one for each experiment, with common code). We created a Sequential Model for each of the network structures described before and compile them using Keras as shown in Figure 16. For each epoch, we iterated through the training batches and train the model using the `train_on_batch` function. After training, we test the model using the test batches. To calculate the F_1-score, we use our own function based on the now removed F_1-score metric from the Keras source code[14].

Another complication we faced was in the `load` function is the time taken to load the dataset from CSV files. Originally, the output of the feature extraction and target vector generation was a pair of CSV files, one for batch. However, with the sparsity of data and a large number of features, these quickly ballooned to in order of 100s of GBs. To avoid this disk space use as well as make the loading faster we created a sparse-CSV representation, where each row would only contain the column numbers for which its value is 1.

3) Infrastructure: For the feature extraction and the target vector generation Java code, we used an Apple Macbook Pro with 16GB RAM. For the Deep Learning in the other three phases we used an AWS `g2.2xlarge` instance that has one graphics card with 4GB RAM using NVIDIA GRID GPUs[15]. Training and testing the model took about 45 minutes to 6 hours, depending on the number of features extracted in the extraction step.

C. Refinement

We first present the results of the model exploration phase. And then, we discuss the improvements we make to the structure for learning in the final type inferencing phase.

[10]https://jena.apache.org/
[11]https://neo4j.com/
[12]https://keras.io/
[13]https://www.tensorflow.org/
[14]https://github.com/fchollet/keras
[15]https://aws.amazon.com/blogs/aws/new-g2-instance-type-with-4x-more-gpu-power/

```
def count_predictions(y_true, y_pred):
    true_pos = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)))
    pred_pos= K.sum(K.round(K.clip(y_pred, 0, 1)))
    possible_pos = K.sum(K.round(K.clip(y_true, 0, 1)))
    return true_pos, pred_pos, possible_pos

def f1score(y_true, y_pred):
    true_pos, pred_pos, possible_pos
            = count_predictions(y_true, y_pred)
    precision = true_pos / (pred_pos + K.epsilon())
    recall = true_pos / (poss_pos + K.epsilon())
    f1score = 2.0 * precision * recall /
                        (precision+recall+ K.epsilon())
    return f1score

#Create instance for the sequential model
deepModel = Sequential(name='Deep Model (6 Dense Layers)')

#Add input layer
deepModel.add(Dense(1024, input_dim=dataX.shape[1],
          init='glorot_normal'))
deepModel.add(BatchNormalization())
deepModel.add(Activation('relu'))
deepModel.add(Dropout(0.2))

#Layer 2
deepModel.add(Dense(1024, init='glorot_normal'))
deepModel.add(BatchNormalization())
deepModel.add(Activation('relu'))
deepModel.add(Dropout(0.2))

#Layer 3
deepModel.add(Dense(1024, init='glorot_normal'))
deepModel.add(BatchNormalization())
deepModel.add(Activation('relu'))
deepModel.add(Dropout(0.2))

#Layer 4
deepModel.add(Dense(1024, init='glorot_normal'))
deepModel.add(BatchNormalization())
deepModel.add(Activation('relu'))
deepModel.add(Dropout(0.2))

#Layer 5
deepModel.add(Dense(1024, init='glorot_normal'))
deepModel.add(BatchNormalization())
deepModel.add(Activation('relu'))
deepModel.add(Dropout(0.2))

#Output layer
deepModel.add(Dense(dataY.shape[1], activation='sigmoid',
       init='glorot_normal'))

#Model compilation
deepModel.compile(loss='binary_crossentropy',
       optimizer='nadam', metrics=[f1score])
```

Fig. 16: Example Sequential Model Creation for Fully-connected Strucutre with 6 Layers

```
#Create the train-test split
numberOfFiles = 638
testSplit=0.2
listOfFiles=list(range(1,numberOfFiles+1))
random.shuffle(listOfFiles)
splitIndex=int((1-testSplit)*numberOfFiles)

for epoch in range(0,5):
   #Train
   for index in range(0,splitIndex):
      dataX, dataY = load(xFiles[index], yFile[index])
      loss, f1score=deepModel.train_on_batch(dataX, dataY)

   #Test
   total_true_pos=0
   total_pred_pos=0
   total_poss_pos=0
   for index in range(splitIndex+1,numberOfFiles+1):
      dataX, dataY = load(xFiles[index], yFile[index])
      predY=model.predict_on_batch(dataX)
      true_pos, pred_pos, possible_pos
            = count_predictions(y_true, y_pred)
      total_true_pos+=true_pos
      total_pred_pos+=pred_pos
      total_poss_pos+=poss_pos

   #Calculate Precision, Recall and F1-score
   precision = total_true_pos /(total_pred_pos + K.epsilon())
   recall = total_true_pos / (total_poss_pos + K.epsilon())
   f1score = 2.0 * precision * recall /
                        (precision+recall+ K.epsilon())
  print("Epoch {}: {}".format(epoch, f1score))
```

Fig. 17: Simplified algorithm used for training and testing

TABLE VII: Results of Model Experimentation Phase

#	Length of walk	Step Category	Strategy for Length	F_1-score for *n* steps		
				n=25	n=50	n=100
1	2	Stay	Fixed	0.9144	0.9182	0.9190
2	2	Stay	Variable	0.9122	0.9168	0.9177
3	2	Move	Fixed	0.8401	0.8435	0.8456
4	2	Move	Variable	0.8260	0.8328	0.8393
5	2	Both	Fixed	0.9126	0.9211	0.9260
6	2	Both	Variable	0.9083	0.9137	0.9166
7	3	Stay	Fixed	0.8945	0.9037	0.9102
8	3	Stay	Variable	0.9015	0.9078	0.9091
9	3	Move	Fixed	0.8317	0.8427	0.8509
10	3	Move	Variable	0.8283	0.8215	0.8299
11	3	Both	Fixed	0.7474	0.8045	0.8669
12	3	Both	Variable	0.8742	0.8785	0.8757

1) Model Experimentation Phase Results: As described at the start of this section, we vary the length of the walks, the number of walks, the type of steps and the length strategy to conduct our experiments. Each of the following results was recorded at the end of 5 epochs (where the performance of almost all the learning plateaued). Table VII shows the results of the experiments.

We can draw the following conclusions from the results in Table VII:

- **As the number of walks increases, the F_1-score also increases:** This is consistent with Phase 1. This is because the number of features extracted also increases.
- **Between fixed and variable length strategies, the fixed length works better:** When we increase the length of the walks we create more complex features. Between fixed and variable length strategies, the fixed length set is made up entirely of these complex features, while the variable length is made of simple features as well. So, given an equal number of walks the fixed strategy has more complex features and performs better.
- **Between step categories, the one with both performs better:** With 2 steps, moving across a relationship creates new features which encode the type information of the adjacent node. So essentially, we are trying to identify the type of the node based on the type of the adjacent node. While the strategy to use attributes, relationship and incoming relationship presence works better than the strategy to move across an outgoing or incoming relationship, their combination works better than either of them.
- **Between features with lengths 2 and 3, a feature**

with length 2 performs better: We conclude that this is because, the type of instance depends on its attributes, relationships and incoming relationships and those of its neighbors. Individuals two steps away have a lesser effect on the type of the current individual.

2) Improvements: Having performed optimizations for the Deep Learning network structures earlier, we do not change this itself. We had already maxed out the number of nodes we can hold in memory and depth due to overfitting concerns. We tried to optimize our network by using LeakyReLUs[15], but the learning was worse.

Upon investigating the features, we found that the step type to move across relationships had scope for improvement. We noted that sometimes, the walk returned to the original individual after traversing to another individual. We ensured that the walk did not return to the original individual. A second optimization we made was select only distinct attributes and relationships in the `stay` step category and to move only across distinct relationships in the `move` step category. By doing this, we ensure that our random walk is able to extract more diverse features about the different types of neighbors. The number of walks was also increased to 125.

We also decided to taper the width of the network as we go deeper. Intuitively, by doing this we are forcing the network to learn feature representations in another dimension space (with fewer dimensions). One of the advantages of this in deep learning is to learn more complex representations. Figure 18 shows the final network. Since this may require more learning, we choose 8 epochs for the training.

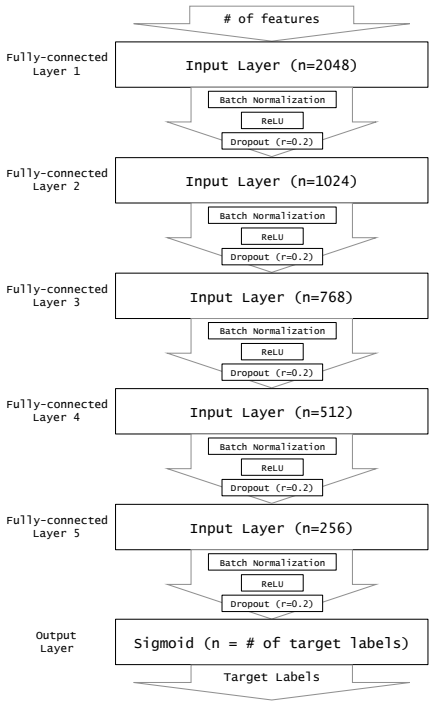

Fig. 18: Final Fully-connected Network Structure

.

Our dataset was split into batches of upto 5000 individuals. While the theory behind it is not completely developved, Keskar et. al [10] suggest based on empirical observations that a small batch size, without going too small, leads to better performance due to smaller gradients. And so, instead of using the `train_on_batch` function in Keras, we switch to the usual `fit` function with an inner batch of length 256 from our original batch.

We put one additional restriction on the `DBpedia-Categories` & `DBpedia-YagoTypes` datasets. We only consider types and categories with at least 200 instance support. This was done since the long-tail of classes with fewer than 200 instance support becomes unmanageable for our classifier as the target vector would cause an out of memory error. As such, our comparison with the DBpedia-Yago benchmarks should be taken in this changed context. The final number of classes in `DBpedia-YagoTypes` was 2083 and that in `DBpedia-Categories` was 10999.

Finally, we also instead of the original train-test split, we split the dataset into training (70%), validation (20%) and test (10%). We then proceed with the final Type Inferencing phase.

V. Results

A. Model Evaluation and Validation

For each of the three datasets - `DBpedia-OntologyTypes`, `DBpedia-Categories` and `DBpedia-YagoTypes`, we create the training, validation and test dataset based on the refinements. The number of random walks chosen was 125. We chose the length of the walk to 2 with the fixed-length strategy. We use different step-categories as well - one where we only use the `move` step category and one using both `stay` and `move` step categories. Table VIII shows the results of the experiments.

TABLE VIII: Results of Type Inferencing Phase

#	Dataset	Step Cat.	# distinct features	F_1-score	
				Validation	Test
13	DBpedia-OntologyTypes	Move	159766	0.8668	0.8668
14	DBpedia-OntologyTypes	Both	233,419	0.9203	0.9200
15	DBpedia-YagoTypes	Move	167,357	0.8576	0.8594
16	DBpedia-YagoTypes	Both	729,468	0.8550	0.8549
17	DBpedia-Categories	Move	73,442	0.2909	0.2895
18	DBpedia-Categories	Both	390,373	0.3388	0.3392

.

1) Robust-ness: As described earlier, we randomly split our data into *training*, *validation* and *test* datasets - data for about 3.1M individual split approximately into datasets of size 70%,20% and 10% respectively. We keep the test data away from the training and validation cycles in the Type Inferencing phase. Decisions on hyper-parameters and refinements that we try here are based on the ability of the model trained on the training data to classify the validation dataset. We want our model to generalize as much as possible to unseen data. Once we perform our refinements and after learning, we exposed our test dataset as the unseen data. Since the F_1-score of the validation and the test is almost the same, we can say that the model performs identically with unseen data. If we were to deploy our model to the real world with similar data profile, we can be confident that it will perform similarly well as a result of these identical numbers and given the large dataset

size (about 300K for test data). We can thus say our model is robust.

Based on the results we our final model and the random walk method combined is a robust approach to performing type inferencing in noisy semantic data that also has an Open World Assumption and performs better as compared to current best benchmarks.

B. Justification

For `DBpedia-OntologyTypes`, SDtype achieves an F_1-score of 0.765 and SLCN[16] achieves a score of 0.847. In comparison, our model with random walks of fixed length 2 and moving to adjacent individual (i.e. using `move` step category) achieves an F_1 score of 0.8668. This model beaths the benchmarks. The model trained on both `move` as well as `stay` step categories achieves an F_1 score of 0.9200, which is on-par with our third benchmark F_1-score of 0.9254 set using all step-types with all available attributes, relationships and incoming relationships. This shows that features extracted with random walks of length 2 perform at least as equal to individual features.

For the `DBpedia-YagoTypes` dataset, SDtype achieves an F_1-score of 0.666 and SLCN[16] achieves a score of 0.702. In comparison, our model with random walks of fixed length 2 and moving to adjacent individual achieves (i.e. using `move` step category) an F_1 score of 0.8594. The model trained on both `move` as well as `stay` step categories achieves an F_1 score of 0.8549. Again, our models beat the benchmarks. However, since we remove the types with a support of less than 200 individuals, this needs further confirmation.

Our F_1 scores for `DBpedia-Categories` dataset models, trained on both `move` as well as `both` step categories achieves an F_1 score of 0.2895 and 0.3392 respectively. Since we do not have any benchmarks to compare these with, we only present these initial results.

Thus we can see that our method to extract features from Semantic Graph using random walks and multi-label classification using deep learning better than existing best solutions that rely on incoming relationships only. It is able to perform classifications in noisy data as well as the Open World Assumption and hence is robust. Since our approach also works on an unseen test set with the same performance, we can reasonably say that the final parameters are appropriate and the model generalizes well.

VI. Conclusion

A. Free-form Visualization

As shown in Figure 19, our Random Walk approach to extract features from Semantic Graphs is able to generate a neighborhood-context by encoding the contexts of the neighbors surrounding the individual we would like to classify. With a random walk of length 2, each walk encodes a signal for two types of information - the first step encodes the information of the type of the current individual based on the kind of relationships it participates in. The second information is the recursive context information about the neighboring individuals based on their neighbours. When we take multiple walks within this

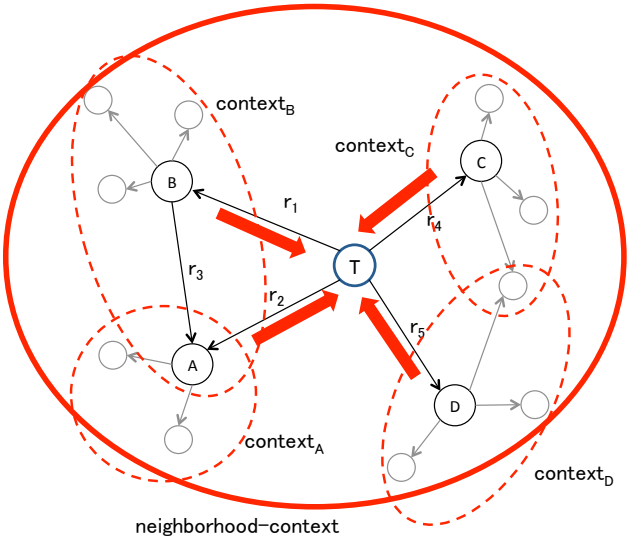

Fig. 19: Context encoding with Random Walks of Length 2

.

neighborhood, the context of the neighborhood starts adding up towards our current individual, thus helping us identify its type.

Using Deep Learning, we can create a robust classifier that uses these extracted features to identify the types in the presence of noise in the data and the Open World Assumption. Additionally, by adding the label de-noising autoencoder, we also are able to overcome noisy labels.

B. Reflection

At the onset, we set out to investigate type inferencing for individuals in a Semantic Graph by using its attributes and relationships. To do this we investigated the random walk method to extract features from the individual's neighborhood. We gradually investigated the capability of our technique to classify DBpedia individuals into multiple types in the DBpedia Ontology using the following phases:

- **Feature Exploration:** We investigated different step types and their individual contribution to the classes. We also investigated different Fully-connected Neural Network structures and selected one model for further analysis.
- **Model Experimentation:** By varying type of steps, number of walks, the length of walks, and strategy for the lengths of those walks we understood how these parameters
- **Type Inferencing:** Finally, after having selected one combination of parameters, we classified individuals in the `DBpedia-OntologyTypes`, `DBpedia-Categories`, and `DBpedia-YagoTypes` datasets.

Our results showed that this approach for classifying *things* in the Semantic Graph outperform state-of-the-art inferencing techniques in the presence of noisy data and the Open World Assumption.

This end-to-end approach is both interesting in its capability as well as its novelty. While creating a sound methodology for achieving the above results was challenging and took many experiments (those that we do not report here), our results show that the random walk approach to extracting features from individuals in a Semantic Graph is a promising step to understanding the context and meaning of *things*. We hope that in the future we would be able to use our approach for semantic understanding in various domains other than the Semantic Web.

C. Improvements

There are two main improvements that we feel are important to investigate in the future.

1) Use attribute values: In the current work, we completely drop the attribute values since the effective strategy to incorporate values together with the random walk strategy was not clear. By using the attributes, we feel that our approach can become better and achieve higher F_1-score.

2) Noise in the labels: We mentioned earlier that the RDF data on the Semantic Web is noisy. For this work, we assume, however, that the labels are correct. If these labels were to have noise in them as well, we would need to investigate some label de-noising techniques such as Wang et. al [28].

3) Exploring other domains: Finally, we would like to try our approach on other domains that use a semantic graph for inferencing like NLP, Scene Understanding, VR, and AR, etc.

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] Chollet, F.: Keras (2015)

[4] 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)

[5] Dozat, T.: Incorporating nesterov momentum into adam. Tech. rep., Stanford University, Tech. Rep., 2015.[Online]. Available: http://cs229. stanford. edu/proj2015/054 report. pdf (2015)

[6] 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)

[7] Glorot, X., Bengio, Y.: Understanding the difficulty of training deep feedforward neural networks. In: Aistats. vol. 9, pp. 249–256 (2010)

[8] 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)

[9] 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)

[10] Keskar, N.S., Mudigere, D., Nocedal, J., Smelyanskiy, M., Tang, P.T.P.: On large-batch training for deep learning: Generalization gap and sharp minima. arXiv preprint arXiv:1609.04836 (2016)

[11] Krizhevsky, A., Hinton, G.: Learning multiple layers of features from tiny images (2009)

[12] 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

[13] 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)

[14] 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)

[15] Maas, A.L., Hannun, A.Y., Ng, A.Y.: Rectifier nonlinearities improve neural network acoustic models. In: Proc. ICML. vol. 30 (2013)

[16] Melo, A., Paulheim, H., Völker, J.: Type prediction in rdf knowledge bases using hierarchical multilabel classification (2016)

[17] Melo, A., Völker, J., Paulheim, H.: Type prediction in noisy rdf knowledge bases using hierarchical multilabel classification with graph and latent features. International Journal on Artificial Intelligence Tools 26(02), 1760011 (2017)

[18] Miao, Q., Fang, R., Song, S., Zheng, Z., Fang, L., Meng, Y., Sun, J.: Automatic identifying entity type in linked data. PACLIC 30 p. 383 (2016)

[19] Paulheim, H.: Knowledge graph refinement: A survey of approaches and evaluation methods. Semantic web 8(3), 489–508 (2017)

[20] Paulheim, H., Bizer, C.: Type inference on noisy rdf data. In: International Semantic Web Conference. pp. 510–525. Springer (2013)

[21] Perozzi, B., Al-Rfou, R., Skiena, S.: Deepwalk: Online learning of social representations. In: Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining. pp. 701–710. ACM (2014)

[22] 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)

[23] 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)

[24] 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)

[25] Tang, B.: The emergence of artificial intelligence in the home: Products, services, and broader developments of consumer oriented ai (2017)

[26] Tsoumakas, G., Katakis, I.: Multi-label classification: An overview. International Journal of Data Warehousing and Mining 3(3) (2006)

[27] Tsuda, K., Saigo, H.: Graph classification. Managing and mining graph data pp. 337–363 (2010)

[28] Wang, D., Tan, X.: Label-denoising auto-encoder for classification with inaccurate supervision information. In: Pattern Recognition (ICPR), 2014 22nd International Conference on. pp. 3648–3653. IEEE (2014)