GrCluster: A score function to model hierarchy in knowledge graph embeddings

Anonymous Author(s)

ABSTRACT

Low-dimensional embeddings for knowledge graph entities and relations help preserve their latent semantics while enabling computation efficiency. These embeddings are often used to perform tasks such as machine translation, sentiment analysis, knowledge graph completion, and information extraction. Knowledge graph embedding methods aid in the representation of entities and relationships of a knowledge graph in continuous vector spaces. However, most existing techniques ignore the inherent hierarchical structure of entities of the knowledge graph, defined by ontological relationships between entity types. This paper introduces a novel score function called GrCluster that helps fill that gap. GrCluster is a simple, intuitive and efficient scoring function that considers the hierarchy of entities of a knowledge graph. The effectiveness of GrCluster is demonstrated by integrating it into several well known embedding models. The experimental results show consistent improvements across metrics and embedding models for the tasks of entity prediction and triplet classification.

CCS CONCEPTS

• **Computing methodologies → Ontology engineering**; *Continuous space search*; Statistical relational learning;

KEYWORDS

knowledge representation, knowledge graph embeddings, representation learning, relational learning, hierarchy, wordnet

ACM Reference Format:

Anonymous Author(s). 2020. GrCluster: A score function to model hierarchy in knowledge graph embeddings. In *Proceedings of ACM SAC Conference (SAC'20).* ACM, New York, NY, USA, Article 4, 4 pages. https://doi.org/xx.xxx/xxx_x

1 SUPPLEMENTARY INFORMATION

In this section, information is provided for the reproducibility of the experiments conducted in the paper. The codes that are used to perform the experiments are publicly available at tinyurl.com/grcluster-supplement.

1.1 Dependencies

1.1.1 Hardware dependencies. Multiple systems were used to train and test various models. All models were trained in a distributed manner across 6 systems. 4 systems had the following hardware specifications:

- CPU : 8th Generation Intel i7 Core Processor
- RAM : 16GB
- Hard disk space : 1TB

The remaining 2 systems had the following hardware specification:

- CPU : Intel Xeon Processor E5
- GPU : 4 x Nvidia GeForce GTX 1080
- RAM : 64GB
- Hard disk space : 1TB

1.1.2 Software dependencies. Table 1 provides a list of software dependencies that are required to run the required codes. To run the codes available at tinyurl.com/grcluster-supplement, please make sure the packages and softwares are installed on the system.

Package / Software	Version
Ubuntu	16.04
Python3	3.5
Cuda (To run on GPU)	9.0
Cudnn (To run on GPU)	7.4
Tensorflow	1.12
Tensorflow-gpu (To run on GPU)	1.12
Numpy	1.16
Tqdm	4.22
Pickle	(Available with Python3)
Argparse	(Available with Python3)
Math	(Available with Python3)
Itertools	(Available with Python3)
Sys	(Available with Python3)
Os	(Available with Python3)

Table 1: Software dependencies for running provided codes.

1.2 Executing code

To execute the main script 'main.py', use the 'python3' command to invoke the Python3 interpreter, and send the argument 'main.py' to execute statements from that script. Additional command line arguments must be provided to set hyperparameters and models that need to be trained. Table 2 gives a list of short and long arguments. All generated embeddings and results are available in 'Results.zip' compressed file in the given link. On extracting the zipped file, a 'Results' directory will be generated. The directory structure for experiments involving the WN18 dataset follow this pattern: Results/<Method>/<Sampling Type>/<Model>/<Test Setting>/><XXX.pickle>. The directory structure for experiments involving the WNNH dataset follow this pattern: Results/WNNH/<Model>/<XXX.pickle>. To view any pickle file, please use the pickle package available on python3 to load and display contents

SAC'20, March 30-April 3, 2020, Brno, Czech Republic

ACM ISBN 978-1-4503-6866-7/20/03…$15.00
https://doi.org/xx.xxx/xxx_x

Flag	Long Argument	Description	Type	Default
	–no-train	Do not train embeddings	bool	False
	–no-test	Do not test embeddings	bool	False
	–no-link-prediction	Do not test embeddings for entity prediction	bool	False
	–no-triplet-classification	Do not test embeddings on triplet classification	bool	False
-s	–embedding-size	Embedding size of each vector	int	100
-b	–batch-size	Batch size while training	int	1024
-m	–margin	Margin of error allowed in the loss	float	1
-r	–learning-rate	Learning rate for the optimizer	float	0.001
-e	–epochs	Number of epochs to train embeddings	int	500
-t	–infinitely-train	Train infinitely with patience	bool	False
-p	–patience	Patience while training the embedding model for validation loss to improve	int	50
-o	–output-file	Pickle file name for the trained model to save	str	None
-i	–input-file	Pickle file name for the trained model to load	str	None
-d	–dataset	Dataset to be used ['WN18', 'WN_HIERARCHY']	str	'WN18'
-a	–embedding-model	Embedding Model to be used ['TransE', 'TransH', 'TransR', 'TransD', 'DistMult', 'ComplEx', 'HolE']	str	'TransE'
-n	–original	Train using original embedding model	bool	False
-q	–sampling-type	Method used to sample data ['uniform', 'bernoulli']	str	'uniform'
-f	–discount-factor	Discounting factor used for distance	float	None
-g	–test-setting	Sampling setting while testing ['raw', 'filter']	str	'raw'
-c	–triplet-classification-times	Number of times triplet classification must be performed	int	25

Table 2: Flags and long arguments that can be used to run the code.

of the file. Alternatively Results/Results.xlsx contains all the results consolidated in a spreadsheet.

1.3 Hyperparameters used

Table 3 presents the list of hyperparameters that can be adjusted and the value of each hyperparameter that remained constant throughout all experiments for a fair comparison of all models.

Hyperparameter	Value(s)
Margin	1
Learning Rate	0.001
Batch Size	1024
Embedding Dimension	100
Discount Factor (For GrCluster)	{0.1,0.25,0.5, 0.75,0.9}
Regularization (For DistMult and ComplEx)	0.0001
Optimizer	Adam

Table 3: Hyperparameters used for various experiments

2 ALGORITHMS

Algorithm 1 creates the forest of hierarchy trees from the WN18 dataset. The first step finds the immediate children for each entity from the dataset. For each permutation of a parent-child entity pair, a depth-first search is performed for the child entity starting at the parent entity. If the child entity is found, its tree is added as a subtree to the parent entity's tree. Finally, if the child is found in a parent tree, it is not a root entity, and therefore added to the *not_root_entities* set.

Algorithm 2 computes the distances between two entities in the forest of hierarchical trees. For a given triplet, each entity in the set of *root_entities* is used as a starting point to perform a depth-first search to the head entity. A similar process is done for the tail entity for that given triplet. The two entity sequences are concatenated, and the common ancestors are removed. The number of remaining entities signifies the number of edges between the two entities in the hierarchy forest.

To decrease the amount of computation, the structure of the WNNH dataset was leveraged. The head and the tail entities of the triplets in the WNNH dataset are always in the same hierarchy subtree. Therefore, entities across hierarchy trees need not be considered. Algorithm 3 offers a dynamic programming solution to arrive at the distance between two entities given a training example. The first step involves recognizing the immediate children of an entity. The second step involves performing a depth-first search for all the descendants of an entity, while updating a global table that stores previously computed distances. If the descendants of an entity have already been searched for, the global table can be looked up for the required information.

The advantage leveraged in Algorithm 3 is not possible while computing the distances between entities in the WN18 dataset because entities in the WN18 dataset are connected by relations other

Algorithm 1: To create the hierarchy forest from the training examples of WN18 dataset

```
Function create_hierarchy_forest train_data
    global descendants = Empty Dictionary
    entities = Empty Set
    foreach head_entity, relation, tail_entity in train_data do
        Add head_entity and tail_entity to entities
    end
    foreach entity in entities do
        descendants[entity] = Empty Dictionary
    end
    foreach head_entity, relation, tail_entity in train_data do
        if relation == 'hypernym' then
            descendants[tail_entity][head_entity] = Empty
              Dictionary
        end
        if relation == 'hyponym' then
            descendants[head_entity][tail_entity] = Empty
              Dictionary
        end
    end
    not_root_entities = Empty Set
    foreach child_entity in entities do
        foreach parent_entity in entities do
            if parent_entity != child_entity then
                if insert(parent_entity, child_entity) then
                    Add child_entity to not_root_entities
                end
            end
        end
    end
    root_entities = entities - not_root_entities
    return root_entities, descendants
end
Function insert parent_entity, search_child_entity
    if length(descendants[parent_entity]) == 0 then
        return False
    end
    foreach child_entity in descendants[parent_entity] do
        if child_entity == search_child_entity then
            descendants[parent_entity][child_entity] =
              descendants[child_entity]
            return True
        end
        if insert(child_entity, search_child_entity) then
            return True
        end
    end
    return False
end
```

Algorithm 2: To calculate the distance between two entities in the training examples of WN18 dataset

```
Function get_entity_distances train_data
    distances = Empty List foreach head_entity, relation,
      tail_entity in train_data do
        chain_to_head = get_chain(head_entity)
        chain_to_tail = get_chain(tail_entity)
        union = chain_to_head ∪ chain_to_tail
        intersection = chain_to_head ∩ chain_to_tail
        difference = union - intersection
        Add length(difference) to distances
    end
    return distances
end
Function get_chain entity, parent_entity = None,
  parent_dictionary = None
    if parent_entity == None then
        foreach parent_entity in root_entities do
            chain = get_chain(entity, parent_entity,
              descendants[parent_entity])
            if length(chain) != 0 then
                return chain
            end
        end
    end
    if entity == parent_entity then
        chain = Empty List
        Add entity to chain
        return chain
    end
    foreach child_entity in parent_dictionary do
        chain = get_chain(entity, child_entity,
          parent_dictionary[child_entity])
        if length(chain) != 0 then
            Add parent_entity to chain
            return chain
        end
    end
    return False
end
```

than the parent-child relation. Inter hierarchy tree connections disallow ignoring chunks of the forest, which is implicit in Algorithm 3.

3 EVALUATION METHODOLOGY

The performance of the trained models were evaluated on the tasks of entity prediction and triplet classification.

3.1 Entity Prediction

The procedure of entity prediction, as used in [1], is task of completing a triplet by predicting the missing entity. The model predicts h given (r, t) or t given (h, r). To implement this task, for each triplet

Algorithm 3: To calculate the distance between two entities in the training examples of WNNH dataset

```
Function get_entity_distances train_data
    global immediate_children = Empty Dictionary
    entities = Empty Set
    foreach head_entity, relation, tail_entity in train_data do
        Add head_entity and tail_entity to entities
    end
    foreach entity in entities do
        immediate_children[entity] = Empty List
    end
    foreach child_entity, relation, parent_entity in train_data
      do
        Add child_entity to
          immediate_children[parent_entity]
    end
    global all_distances = Empty Dictionary
    foreach entity in entities do
        all_distances[entity] =
          get_descendants(immediate_children[entity])
    end
    distances = Empty List
    foreach head_entity, relation, tail_entity in train_data do
        Add all_distances[tail_entity][head_entity] to
          distances
    end
    return distances
end
Function get_descendants children
    descendant_distances = Empty Dictionary
    foreach child in children do
        if child in distances then
            Copy all_distances[child] to
              descendant_distances_copy
            foreach child_copy in descendant_distances_copy
              do
                descendant_distances[child_copy] =
                  descendant_distances_copy[child_copy] + 1
            end
            continue
        end
        all_distances[child] =
          get_descendants(immediate_children[child])
        Copy all_distances[child] to
          descendant_distances_copy
        foreach child_copy in descendant_distances_copy do
            descendant_distances[child_copy] =
              descendant_distances_copy[child_copy] + 1
        end
    end
    return descendant_distances
end
```

in the test set, the head entity (or tail entity) is replaced by all entities to generate new triplets. These triplets are passed to the embedding model to obtain a score. On obtaining the scores for all the new triplets, the scores are sorted in ascending order. Ranking the scores in ascending order helps obtain the rank of the original correct triplet. Since replacing the entities of a triplet may cause the creation of positive triplets that may exist in the knowledge graph, the objective function may assign a lower score for that triplet, which in turn would assign a higher rank for the triplet in testing. Therefore, triplets that exist in the knowledge graph are removed from the triplets generated to test a particular sample. The former test setting is called raw and the latter test setting is called filtered. Three metrics are used while comparing the results of the entity prediction task: Mean Reciprocal Rank (MRR), Hits@3 (H3) and Hits@10 (H10).

3.2 Triplet Classification

This task performs binary classification on a triplet, whether a given triplet (h, r, t) is correct or not. This test was introduced by Socher et al. [2]. A similar methodology was followed while generating the negative samples for the WN18 dataset. The Uniform sampling method was used generate negative samples for the triplets in the test split. For each model, the triplet classification task was done 25 times, and the result of the average accuracy has been displayed.

REFERENCES

[1] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. 2013. Translating embeddings for modeling multi-relational data. In *Advances in neural information processing systems*. 2787–2795.

[2] Richard Socher, Danqi Chen, Christopher D Manning, and Andrew Ng. 2013. Reasoning with neural tensor networks for knowledge base completion. In *Advances in neural information processing systems*. 926–934.