Ricerca di nodi vicini

Knowledge Engineering

Sofia Galante
sofia.galante@stud.unifi.it

Agata Parietti
agata.parietti@stud.unifi.it

Indice

1 Introduzione

Questo progetto ha come scopo quello di implementare una ricerca di nodi vicini in SPARQL data una Knowledge Base a propria scelta.
Per poter raggiungere questo scopo si deve:

1. eseguire l'embedding dei nodi della KB;
2. estendere SPARQL con una funzione esterna per poter calcolare la distanza tra gli embeddings dei nodi.

Visto che l'embedding svolto mappa ogni nodo in un vettore in uno spazio n-dimensionale, si è deciso di utilizzare la **cosine distance** come metrica. La formula è la seguente:

$$d(v_1, v_2) = 1 - cos(\theta) = 1 - \frac{v_1 \cdot v_2}{||v_1||||v_2||} \tag{1}$$

dove θ è l'angolo tra i vettori v_1 e v_2.
Si noti che la distanza è quindi un numero reale che sta tra 0 (nodi identici) e 2 (nodi opposti).
Gli embeddings vengono svolti con dei programmi python, mentre l'estensione di SPARQL è stata fatta con delle classi Java. L'estensione è stata fatta per la piattaforma Fuseki.

2 Embeddings

Per decidere l'embedding giusto da usare si sono letti diversi articoli (presenti in bibliografia). Durante la ricerca, si è notato che esistono due diverse scuole di pensiero per quanto riguarda gli embeddings di una KB:

1. la prima vede la KB come un Knowledge Graph e cerca di creare un embedding che rispetti le regole di transizione nel grafo;

2. la seconda trasforma le triple della KB in frasi e svolge un embedding classico dell'NLP, basato sui language models.

Per questo motivo, si è deciso di implementare un metodo per entrambe le scuole di pensiero, lasciando poi all'utente la scelta di quale utilizzare durante la sua ricerca.

2.1 Creazione del dataframe dalla KB

Per poter effettuare l'embedding sulla KB, si deve prima trasformarla in un dataframe di triple.
Per compiere questo passaggio, si è utilizzata la libreria *rdflib* che permette di leggere un grafo di una KB e di estrarre da essa tutte le triple presenti. Il risultato è poi stato salvato con *pandas*.

Nel nostro progetto, è possibile sia creare un embedding di un grafo salvato in locale (in questo caso il grafo dovrà essere inserito nella cartella *dataset*) oppure di utilizzare un grafo direttamente da un rdfstore.
La seconda modalità è stata resa possibile dall'utilizzo della libreria *sparqlwrapper*. In caso si volesse eseguire l'embedding di un grafo direttamente da un rdfstore, i passaggi sono i seguenti:

1. si deve modificare il file *sparql.txt*, contente l'rdfstore di interesse;

2. si deve modificare il file *query.txt*, contente una query scritta in linguaggio SPARQL; si noti che questa query deve essere di tipo CONSTRUCT, in quanto rdflib deve ricevere in ingresso un grafo serializzato nel formato RDF.

Un esempio di file *sparql.txt* e di *query.txt* è il seguente:

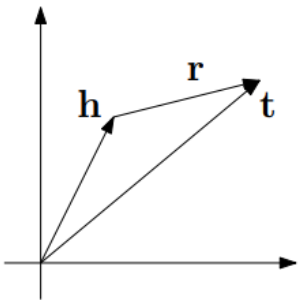
1. **sparql.txt**: *https://query.wikidata.org/sparql*

2. **query.txt**:
   CONSTRUCT {*?s ?p ?o*}
   WHERE {
   ?s ?p ?o
   }
   LIMIT 100000}

In questo caso, il grafo costruito è composto dalle prime 100000 triple presenti su *wikidata*.

2.2 TransE

L'embedding di tipo TransE è un embedding di tipo *transizionale*.
Con l'algoritmo TransE, ogni entità e ogni relazione di una KB viene mappata in uno spazio vettoriale facendo in modo che si mantenga la relazione tra le triple.
Data una tripla *(h, r, t)* vengono fatti degli embedding per ogni elemento (**h**, **r**, **t**) tali che:



$$\mathbf{h} + \mathbf{r} \simeq \mathbf{t}$$

Per imparare gli embedding si minimizza un margin-based ranking criterion

$$L = \Sigma_{(h,r,t)\in S}\Sigma_{(h',r,t')\in S_{(h,r,t)}}[\gamma + d(\mathbf{h}+\mathbf{r},\mathbf{t}) - d(\mathbf{h}'+\mathbf{r},\mathbf{t}')]_+ \qquad (2)$$

dove:

1. S è l'insieme di tutte le triple della KB;
2. $(\mathbf{h}', \mathbf{r}, \mathbf{t}')$ è una tripla corrotta (cioè ottenuta prendendo una tripla esistente in S e modificando uno dei due nodi) e $S_{(h,r,t)}$ è l'insieme di triple corrotte ottenute a partire dalla tripla (h, r, t);
3. $d(\mathbf{h}+\mathbf{r},\mathbf{t})$ è una misura di dissimilarità (nel nostro caso la $L_2 - norm$);
4. γ è un margin hyperparameter;
5. $[x]_+$ indica la parte intera di x.

2.2.1 Implementazione

L'algoritmo che implementa TransE si trova nel file *te_embedding.py*.
Il modello TransE si trova nella libreria *torchkge*, un'estensione di *PyTorch* che permette di fare vari embeddings di tipo transizionale (e non) su una KB.

Gli iperparametri che l'utente può scegliere sono:

1. $emb\_dim \rightarrow$ la dimensione dei vettori dell'embedding;
2. $lr \rightarrow$ learning rate per il training;
3. $n\_epochs \rightarrow$ numero di epoche per il training;
4. $b\_size \rightarrow$ dimensione del batch;
5. $margin \rightarrow$ margin hyperparameter.

Dei valori di default sono stati scelti per la KB di prova fornita da fuseki (**Cheese.ttl**) ma si consiglia di modificare i parametri a seconda delle dimensioni della KB utilizzata.
Si noti che la KB viene prima trasformata in un Dataframe pandas per poter essere data in input al modello. La trasformazione in Dataframe pandas è stata resa possibile dal parser presente nella libreria *rdflib* e l'implementazione di questa trasformazione si trova nel file *dataset_preparation.py*.

2.3 RDF2Vec

RDF2Vec è un'interpretazione per KB dell'embedding NLP Word2Vector.
In questo caso le triple della KB vengono reinterpretate come delle frasi della forma $nodo_1 - relazione_1 - nodo_2 - relazione_2 - nodo_3$ e così via (si noti che la lunghezza della catena $nodo - relazione$ è un iperparametro).

L'algoritmo RDF2Vec compie quindi i seguenti passi:

1. per ogni nodo, si creano n "frasi" di lunghezza d (n e d sono a scelta dell'utente);

2. su queste frasi si applica il Word2Vector $\rightarrow$ gli embedding vengono aggiornati per imparare un *language model* che permette di prevedere la "parola" (i.e. il nodo o la relazione) che si trova al posto t di una frase date le parole che ha prima, dopo, o intorno.

Nel nostro caso, si è deciso di utilizzare il *Continuous Bag-of-Words Model*, in cui la parola predetta dal modello è la parola centrale in una finestra di grandezza predefinita.
Si vuole, quindi, massimizzare la *averange log-probability*:

$$\frac{1}{T}\sum_{t=1}^{T} logP(w_t|w_{t-c}, w_{t-c+1}, ..., w_{t+c-1}, w_{t+c}) \tag{3}$$

dove T è il numero di parole totali.
La probabilità sopra riportata è calcolata con una *softmax*.

2.3.1 Implementazione

L'algoritmo che implementa RDF2Vec si trova nel file *w2v_embedding.py*.
La libreria utilizzata per compiere l'embedding è *gensim*.

Gli iperparametri che l'utente può scegliere sono:

1. $n\_phrases \rightarrow$ numero di "frasi" costruite per nodo;

2. $dim\_phrases \rightarrow$ numero di "parole" presenti in una "frase";

3. $workers \rightarrow$ numero di thread della CPU usati per fare l'embedding;

4. $emb\_dim \rightarrow$ la dimensione dei vettori dell'embedding;

5. $min\_count \rightarrow$ il numero minimo di volte con cui una "parola" deve apparire nelle "frasi" per essere considerata;

6. $sg \rightarrow$ un parametro che indica il metodo con cui creare il *language model* (0 = Continuous Bag-of-Words Model, 1 = Skip-Gram).

Dei valori di default sono stati scelti per la KB di prova fornita da fuseki (**Cheese.ttl**) ma si consiglia di modificare i parametri a seconda delle dimensioni della KB utilizzata.
Si noti che la KB viene prima trasformata in un Dataframe pandas per poter essere data in input al modello. La trasformazione in Dataframe pandas è stata resa possibile dal parser presente nella libreria *rdflib* e l'implementazione di questa trasformazione si trova nel file *dataset_preparation.py*.

3 Estensione di SPARQL

L'estensione di SPARQL è stata fatta attraverso 3 diverse classi java: $Embeddings.class$, $W2V.class$ e $TE.class$.
La classe $Embeddings.class$ contiene tutte le funzioni per caricare gli embeddings dei nodi e calcolare la distanza tra di essi, mentre le classi $W2V.class$ e $TE.class$ sono estensioni della classe $FunctionBase2$ di $fuseki$ e implementano la funzione *exec(NodeValue1, NodeValue2)* che prende in ingresso due nodi della KB e restituisce un terzo nodo in uscita (nel nostro caso, il nodo in uscita è un numero *double* equivalente alla distanza tra i nodi in ingresso).
Per una gestione ottimale della memoria, si noti che gli embeddings vengono caricati solo al primo utilizzo della funzione e poi salvati in una HashMap persistente in memoria.

Per funzionare, fuseki deve accedere unicamente ai file .class e .jar delle classi implementate. Si è comunque deciso di condividere il file sorgente .java per poter vedere più nel dettaglio il codice.

Nella sezione successiva sono presenti diversi esempi su come si possono utilizzare le classi.

4 Esempi di query SPARQL con le classi implementate

Le classi TE e $W2V$ possono essere richiamate all'interno di normali query SPARQL.

Come mostrato in figura, la funzione restituisce la distanza tra due nodi.

I risultati ottenibili autonomamente dalla funzione sono alquanto limitati, ma si possono estendere utilizzando le clausole SPARQL e avere risultati molto più completi.
Ad esempio, si può ottenere l'elenco dei nodi vicini ad un nodo ordinati in base a quanto sono effettivamente vicini:

```
PREFIX f: <java:>

SELECT DISTINCT ?s1 ?dist WHERE {
  ?s1 ?p1 ?o1
  BIND(f:TE(<http://data.kasabi.com/dataset/cheese/halloumi>, ?s1) as ?dist)
  FILTER(?s1 != <http://data.kasabi.com/dataset/cheese/halloumi>)
} ORDER BY ?dist
```

Table | Response | 691 results in 17.507 seconds | Simple view | Ellipse | Filter query results | Page size: 50

	s1	dist
1	<http://ec.europa.eu/eurostat/ramon/rdfdata/nuts2008/ITC4A>	"0.14925667247030660"^^<http://www.w3.org/2001/XMLSchema#double>
2	<http://ec.europa.eu/eurostat/ramon/rdfdata/nuts2008/ITE42>	"0.19483171001167310"^^<http://www.w3.org/2001/XMLSchema#double>
3	<http://data.kasabi.com/dataset/cheese/formaggella-del-luinese>	"0.20044471778829642e0"^^<http://www.w3.org/2001/XMLSchema#double>
4	<http://data.kasabi.com/dataset/italy/pesaro>	"0.22127851360444872e0"^^<http://www.w3.org/2001/XMLSchema#double>
5	<http://data.kasabi.com/dataset/italy/pradleves>	"0.22195252132309340"^^<http://www.w3.org/2001/XMLSchema#double>
6	<http://ec.europa.eu/eurostat/ramon/rdfdata/nuts2008/ITE2>	"0.25230508622893744e0"^^<http://www.w3.org/2001/XMLSchema#double>
7	<http://data.kasabi.com/dataset/italy/massa>	"0.26496289835877420"^^<http://www.w3.org/2001/XMLSchema#double>
8	<http://ec.europa.eu/eurostat/ramon/rdfdata/nuts2008/ITD5>	"0.28641897538910100"^^<http://www.w3.org/2001/XMLSchema#double>
9	<http://data.kasabi.com/dataset/italy/paulilatino>	"0.28714477591737597e0"^^<http://www.w3.org/2001/XMLSchema#double>
10	<http://data.kasabi.com/dataset/italy/barletta-andria-trani>	"0.29114243878297630"^^<http://www.w3.org/2001/XMLSchema#double>
11	<http://ec.europa.eu/eurostat/ramon/rdfdata/nuts2008/ITD3>	"0.29912183902101497e0"^^<http://www.w3.org/2001/XMLSchema#double>

Oppure si possono calcolare le distanze tra tutti i nodi nella KB e osservare quali sono i nodi più vicini tra loro:

```
PREFIX f: <java:>
SELECT ?s1 ?s2 ?dist WHERE {
  {
    SELECT ?s1 ?s2
    WHERE {
        ?s1 ?p1 ?o1.
        ?s2 ?p2 ?o2
      FILTER(?s1 != ?s2)
    } GROUP BY ?s1 ?s2
    ORDER BY ?s1
  }
  BIND(f:W2V(?s1, ?s2) as ?dist)
}
LIMIT 25
```

Table | Response | 25 results in 32.097 seconds | Simple view | Ellipse | Filter query results | Page size: 50

	s1	s2	dist
1	<http://data.kasabi.com/dataset/cheese/abondance>	<http://data.kasabi.com/dataset/cheese/afuegal-pitu>	"0.10265697236641502e0"^^<http://www.w3.org/2001/XMLSchema#doubl...
2	<http://data.kasabi.com/dataset/cheese/abondance>	<http://data.kasabi.com/dataset/cheese/agri-di-valtorta>	"0.02696675213607857 6e0"^^<http://www.w3.org/2001/XMLSchema#dou...
3	<http://data.kasabi.com/dataset/cheese/abondance>	<http://data.kasabi.com/dataset/cheese/allgauer-bergkase>	"3.731351638780467E-8"^^<http://www.w3.org/2001/XMLSchema#double>
4	<http://data.kasabi.com/dataset/cheese/abondance>	<http://data.kasabi.com/dataset/cheese/allgauer-emmentaler>	"0.00835770831278193e0"^^<http://www.w3.org/2001/XMLSchema#double>
5	<http://data.kasabi.com/dataset/cheese/abondance>	<http://data.kasabi.com/dataset/cheese/allgauer-sennalpkase>	"0.04326828867389309 6e0"^^<http://www.w3.org/2001/XMLSchema#do...
6	<http://data.kasabi.com/dataset/cheese/abondance>	<http://data.kasabi.com/dataset/cheese/almkase->	"0.00581789221438844 3e0"^^<http://www.w3.org/2001/XMLSchema#dou...
7	<http://data.kasabi.com/dataset/cheese/abondance>	<http://data.kasabi.com/dataset/cheese/altenburger-ziegenkase>	"3.675549204948947E-4"^^<http://www.w3.org/2001/XMLSchema#double>
8	<http://data.kasabi.com/dataset/cheese/abondance>	<http://data.kasabi.com/dataset/cheese/anevato>	"0.00264236917231852 6e0"^^<http://www.w3.org/2001/XMLSchema#dou...
9	<http://data.kasabi.com/dataset/cheese/abondance>	<http://data.kasabi.com/dataset/cheese/arzua-ulloa>	"0.05775541015736352e0"^^<http://www.w3.org/2001/XMLSchema#double>
10	<http://data.kasabi.com/dataset/cheese/abondance>	<http://data.kasabi.com/dataset/cheese/asiago>	"6.136143540659278E-4"^^<http://www.w3.org/2001/XMLSchema#double>
11	<http://data.kasabi.com/dataset/cheese/abondance>	<http://data.kasabi.com/dataset/cheese/asiago-dallevo>	"0.07569278082016029e0"^^<http://www.w3.org/2001/XMLSchema#doub...

Infine, si noti che per passare da una funzione all'altra basta modificare il nome della funzione chiamata all'interno del BIND.

5 Future implementazioni

5.1 Estendere a nuovi tipi di Embeddings

Il progetto da noi presentato è estendibile sia nei programmi python, sia nelle classi java per essere utilizzato con altri tipi di embeddings diversi da *RDF2Vec* e *TransE*.
In particolare, la libreria *torchkge* utilizzata per *TransE* contiene al suo interno altri modelli di embeddings, rendendo il tutto, quindi, ancora più facile da estendere.

Per quanto riguarda le classi java, l'unica cosa da fare è creare una classe java che estende la classe *FunctionBase2* di Fuseki e che nella sua funzione *exec()* richiami la funzione omonima della classe *Embeddings*.

5.2 Ottimizzazione dell'uso della memoria

Nel caso di utilizzo di una KB salvata in locale, i file python attuali caricano l'intera KB per crearne l'embedding. Questo può essere un problema per KB molto grandi, in quanto occupano molta memoria in RAM.
Una possibile implementazione futura può essere, quindi, quella di ottimizzare la lettura della KB in memoria.

Riferimenti bibliografici

[1] Apache-Jena Fuseki
https://jena.apache.org/documentation/fuseki2/index.html

[2] Petar Ristoski, Heiko Paulheim. RDF2Vec: RDF Graph Embeddings for Data Mining
https://madoc.bib.uni-mannheim.de/41307/1/Ristoski_RDF2Vec.pdf

[3] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran. Translating Embeddings for Modeling Multi-relational Data
https://proceedings.neurips.cc/paper/2013/file/1cecc7a77928ca8133fa24680a88d2f9-Paper.pdf

[4] TorchKGE
https://torchkge.readthedocs.io/en/latest/

[5] Gensim
https://radimrehurek.com/gensim/

[6] RDFLib
https://rdflib.readthedocs.io/en/stable/

[7] SparqlWrapper
https://pypi.org/project/SPARQLWrapper/