

UNIVERSITÀ DEGLI STUDI DI SALERNO

Dipartimento di Ingegneria dell'Informazione ed
Elettrica e Matematica Applicata



Progetto Tecnologie Semantiche

Vincenzo Vigliotti

Matr: 0622701206

Sommario

1. Executive Summary	3
1.1 Descrizione preliminare del progetto	3
1.2 Problemi affrontati	3
2. Progettazione ed implementazione	4
2.1 Testo analizzato	4
2.2 Pre-processing del testo	4
2.3 Coreference resolution	4
2.4 Identificazione e rimozione delle stopwords	4
2.5 Estrazione delle triple dal testo	5
2.6 Lemmatizzazione	6
2.7 Salvataggio delle triple	7
2.8 Creazione training, test e validation sets	8
2.9 Scelta del modello e fine tuning degli iperparametri	9
2.10 Creazione e predizione di triple <i>unseen</i>	11
2.11 SPARQL <i>queries</i> su DBpedia	12
2.12 Creazione di un grafo Node4J ed inserimento delle risorse	15
3. Analisi sperimentale del sistema	16
3.1 Analisi quantitative dei risultati	16
3.2 Analisi qualitative dei risultati	16
Indice delle figure	19

1. Executive Summary

1.1 Descrizione preliminare del progetto

In questo progetto di Tecnologie Semantiche sono state utilizzate tecniche di estrazione semantica per ricavare rilevanti informazioni semantiche in forma di triple (soggetto, predicato, complemento) da un testo in linguaggio naturale, per poi addestrare una rete neurale con il framework Ampligraph allo scopo di fare inferenza di nuove triple mai viste dalla rete, per poi riportare l'insieme delle triple estratte e predette all'interno di un grafo di Neo4J, in cui ogni risorsa, laddove possibile, è rappresentato da un uri ottenuto tramite query a Dbpedia o, in caso contrario, da un uri arbitrario.

Il codice del progetto è disponibile su GitHub al seguente indirizzo:
<https://github.com/ItsaiNk/SemanticTechnologiesProject>

1.2 Problemi affrontati

- Selezione del testo su cui lavorare
- Effettuare pre-processing del testo
- Effettuare *coreference resolution* sull'intero testo, allo scopo di sostituire eventuali pronomi con gli appropriati soggetti/oggetti
- Identificazione e rimozione delle cosiddette *stopwords*
- *Tokenization, POS tagging, lemmatization*
- Estrazione delle triple dal testo
- Scelta della migliore rete neurale da addestrare con Ampligraph, *fine tuning* degli *hyperparameters*
- Addestramento del modello
- Creazione e predizione di triple *unseen*
- Associazione, laddove possibile, degli elementi della tripla con uri estratti da Dbpedia tramite query
- Creazione di un grafo Node4J ed inserimento delle risorse e *link* all'interno.

2. Progettazione ed implementazione

2.1 Testo analizzato

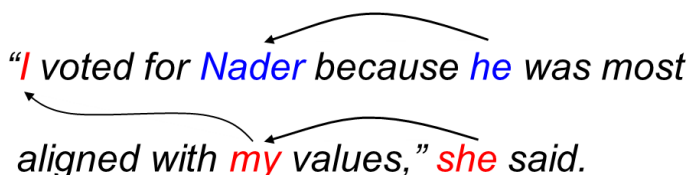
Ho scelto di incentrare il progetto sull'analisi di un'opera letteraria, in particolare la base di partenza sono stati i riassunti dei sette libri della saga di Harry Potter, i cui *plot* in lingua inglese sono stati copiati da wikipedia manualmente, dalle singole pagine relative ai libri, ed inseriti in file di testo denominati *hpi*, con *i=1,2,...,7*. I file di testo sono inseriti all'interno della cartella *plots*.

2.2 Pre-processing del testo

Allo scopo di semplificare le successive operazioni di analisi testuale, ho definito una funzione `preprocess_text(text)` definita all'interno di `nltk_utils.py` (righe 98-120). La funzione prende in input un testo, all'interno del quale ricerca e sostituisce eventuali forme contratte. Ad esempio, sostituisce la forma contratta *'nt* con l'equivalente forma *not*, oppure *'ve* con *have*. La scelta si è rivelata particolarmente efficace nelle successive operazioni di *coreference resolution*, rimozione delle *stopwords*, oltre al principale obiettivo di estrazione delle triple dal testo.

2.3 Coreference resolution

Con *coreference resolution* si intende il ricercare all'interno di un testo tutte le espressioni che si riferiscono alla stessa entità. Spesso, questo *task* non è così banale da eseguire, poiché oltre ad analizzare o capire il significato di una parola o di una frase son richieste anche informazioni sul contesto, conoscenze del mondo reale come luoghi, eventi o persone famose, tendenze di associare determinate parole in uno specifico contesto, riconoscimento del genere grammaticale ed altre proprietà. Il task è stato implementato nella funzione `coref_resolution(text)` all'interno del file `coreference.py` attraverso due librerie¹:



The diagram shows two sentences. The first sentence is "I voted for Nader because he was most". The second sentence is "aligned with my values," she said. Arrows indicate coreference: an arrow from "I" to "she", an arrow from "he" to "Nader", and an arrow from "my" to "values".

Figura 1 - Esempio di coreference resolution

- SpaCy: una libreria open source per l'elaborazione del linguaggio naturale basata su *deep learning*. In particolare, mette a disposizione modelli per le operazioni di *clustering* e *tokenization* per diverse lingue (nel caso in esame la lingua è quella inglese)
- Neuralcoref: una libreria anche essa basata su *deep learning* che sfrutta il *parser* di SpaCy per l'operazione vera e propria di coreference resolution.

2.4 Identificazione e rimozione delle stopwords

Le *stopwords* sono parole che sono poco significative nell'analisi di un testo poiché ricorrono frequentemente e dunque rappresentano una minore informazione. Tuttavia, ho potuto notare che la rimozione di suddette parole prima della *coreference resolution* risultasse solamente dannosa al processo, ottenendo risultati peggiori. Per questo motivo ho scelto di effettuare questo task solamente dopo quelli precedentemente

¹ <https://towardsdatascience.com/from-text-to-knowledge-the-information-extraction-pipeline-b65e7e30273e>

descritti. Le stopwords variano a seconda della lingua e non esiste una lista definitiva, su cui tutti concordano, di queste parole. Nel progetto ho scelto di utilizzare quelle presenti nella libreria *nltk*, di cui ho potuto ottenere un elenco completo tramite la funzione `print_stopwords()` (righe 74-76) che ho definito all'interno del file `nltk_utils.py`. A partire da questo elenco ne ho ottenuto uno *custom*, in cui ho rimosso tutte le stopwords che corrispondessero o contenessero verbi. L'elenco modificato è stato inserito all'interno di un file di testo chiamato `stopwords`, per poi caricarlo all'interno di un oggetto di tipo lista e salvarlo in un file oggetto nel percorso `./obj/custom_stopwords.obj` tramite la libreria *pickle* (l'implementazione di quanto descritto si trova nella funzione `create_stopwords_custom_object(filename)` nel file `nltk_utils.py`, righe 79-86). È possibile caricare in qualunque momento questo oggetto tramite la funzione `load_stopwords_custom_object()` (righe 89-91). L'operazione di rimozione delle stopwords dal testo è implementata nella funzione `remove_stopwords(text, custom_stopwords)` definita nello stesso file (righe 10-20).

```
10 def remove_stopwords(text, custom_stopwords):
11     if custom_stopwords:
12         stopwords_list = load_stopwords_custom_object()
13     else:
14         stopwords_list = stopwords.words('english')
15     text_tokens = word_tokenize(text)
16     tokens = [word for word in text_tokens if word.lower() not in stopwords_list]
17     text = ""
18     for token in tokens:
19         text = text + str(token) + " "
20     return text
```

Figura 2 - funzione `nltk_utils.py/remove_stopwords`

`Custom_stopwords` è una variabile booleana che permette di selezionare il caricamento della lista custom o quella di default. La funzione divide il testo in *token*, ovvero parti elementari di testo, attraverso la funzione built-in della libreria *nltk* e per ognuno di essi controlla se appartiene all'elenco delle stopwords non ammesse. Il testo risultante in uscita dalla funzione sarà la concatenazione di tutti i token di parole ammesse.

2.5 Estrazione delle triple dal testo

Il fulcro della prima parte del progetto è stata l'estrazione dal testo delle triple, ovvero di relazioni in forma soggetto-predicato-complemento. L'università di Stanford ha realizzato una libreria chiamata *StanfordOpenIE* disponibile a tutti, con le migliori prestazioni tra i tool open source. Originariamente in linguaggio Java, esiste un *wrapper* per il linguaggio Python. All'interno del file `triplets.py` è possibile trovare tutta l'implementazione necessaria al task. Ho creato una classe `OpenIEClient`, il cui costruttore definisce il valore della proprietà `openie.affinity_probability_cap`, ovvero il limite inferiore di certezza che deve avere il modello per prendere come valida una tripla (il valore di default è pari a 1/3, ho trovato un giusto compromesso con un valore pari a 0.8), per poi assegnare alla variabile `client` una istanza della classe *StanfordOpenIE*. All'interno della classe ho definito la funzione `extract_triplets(self, text)`, che prende in input il testo da esaminare e ritorna l'elenco delle relazioni individuate attraverso la funzione `annotate` del client. Quest'ultima integra diverse operazioni sul testo come *clustering*, *tokenization* e *pos tagging*, in maniera totalmente trasparente all'utente. Ho scelto di creare una classe in questo caso e non una singola funzione per migliorare le prestazioni: tramite la classe ho potuto istanziare il client una singola volta, diminuendo notevolmente i tempi di analisi di ogni singolo testo.

2.6 Lemmatizzazione

La lemmatizzazione è il processo, in questo caso automatizzato, di riduzione di una forma flessa di una parola alla sua forma canonica detta lemma, ovvero la forma di citazione di una parola in un dizionario². Inizialmente ho implementato all'intero del file `nltk_utils.py` una funzione `lemmatize_triplets(triplets)` (righe 36-47), la quale esegue il task di lemmatizzazione su ogni parola della relazione di ogni tripla in input. Ho notato tuttavia come i risultati non fossero soddisfacenti, poiché molte parole son risultate corrotte. Ho scelto dunque di effettuare l'operazione soltanto sui verbi presenti all'interno del predicato della tripla, attraverso la funzione `lemmatize_triplets_only_verbs(triplets)` (righe 50-67).

```
50 def lemmatize_triplets_only_verbs(triplets):
51     lemmatizer = WordNetLemmatizer()
52     for triplet in triplets:
53         relation = triplet["relation"]
54         tagged_tokenized_relation = pos_tag(word_tokenize(str(relation)))
55         new_relation = ""
56         for token in tagged_tokenized_relation:
57             if new_relation == "":
58                 if get_wordnet_pos(token[1]) is wordnet.VERB:
59                     new_relation = new_relation + str(lemmatizer.lemmatize(token[0], wordnet.VERB))
60                 else:
61                     new_relation = new_relation + str(token[0])
62             else:
63                 if get_wordnet_pos(token[1]) is wordnet.VERB:
64                     new_relation = new_relation + " " + str(lemmatizer.lemmatize(token[0], wordnet.VERB))
65                 else:
66                     new_relation = new_relation + " " + str(token[0])
67         triplet["relation"] = new_relation
```

Figura 3 - funzione `nltk_utils.py/lemmatize_triplets_only_verbs`

Allo scopo, ho scelto l'implementazione con WordNet presente nella libreria nltk. Per ogni tripla in input:

- Ricavo il termine centrale, ovvero il predicato
- Effettuo il pos tag di ogni token tramite la funzione integrata in nltk
- Per ogni token ricavato ricavo l'equivalente del tag in WordNet tramite la funzione `get_wordnet_pos(treebank_tag)` (righe 23-33)
- Se corrisponde ad un verbo, ricavo il lemma del token e lo sostituisco; altrimenti, il token viene lasciato inalterato.

```
23 def get_wordnet_pos(treebank_tag):
24     if treebank_tag.startswith('J'):
25         return wordnet.ADJ
26     elif treebank_tag.startswith('V'):
27         return wordnet.VERB
28     elif treebank_tag.startswith('N'):
29         return wordnet.NOUN
30     elif treebank_tag.startswith('R'):
31         return wordnet.ADV
32     else:
33         return "n"
```

Figura 4 - funzione `nltk_utils.py/get_wordnet_pos`

² <https://it.wikipedia.org/wiki/Lemmatizzazione>
[https://it.wikipedia.org/wiki/Lemma \(linguistica\)](https://it.wikipedia.org/wiki/Lemma_(linguistica))

2.7 Salvataggio delle triple

Al termine delle elaborazioni precedenti, le triple vengono salvate all'interno di un file csv tramite la funzione `triplets_to_csv(triplets, filename)` presente in `main.py` (righe 26-31), in cui ogni riga corrisponde ad una tripla i cui termini (soggetto, predicato, complemento) sono separati da una virgola.

```
26 def triplets_to_csv(triplets, filename):
27     with open(filename, "w", newline='') as f:
28         writer = csv.writer(f)
29         for triplet in triplets:
30             data = [triplet["subject"], triplet["relation"], triplet["object"]]
31             writer.writerow(data)
```

Figura 5 - funzione `main.py/triplets_to_csv`

Le operazioni presentate nei punti 2.2 – 2.7 sono eseguiti in sequenza attraverso la funzione `create_csvs()` (righe 34-63 in `main.py`)

```
34 def create_csvs():
35     client = OpenIEClient()
36     for i in range(start, stop + 1):
37         text = readfile("./plots/hp" + str(i))
38         print("\n*****TEXT*****\n")
39         print(text)
40         print("\n*****\n")
41         text = preprocess_text(text)
42         if coreference:
43             text = coref_resolution(text)
44         if stopwords:
45             text = remove_stopwords(text, custom_stopwords)
46
47         triplets = client.extract_triplets(text)
48
49         if lemmatize:
50             if only_verbs:
51                 lemmatize_triplets_only_verbs(triplets)
52             else:
53                 lemmatize_triplets(triplets)
54
55         triplets_to_csv(triplets, csv_folder + "triplets_hp" + str(i) + ".csv")
56
57     # how to manipulate: is a dict with 3 keys:
58     # subject => triplet["subject"] return the subject of the triplet
59     # relation => triplet["relation"] return the relation (predicate) of the triplet
60     # object => triplet["object"] return the object of the triplet
61     #
62     # each triplet is of class 'dict'
63     # var 'triplets' is of class 'list'
```

Figura 6 - funzione `main.py/create_csvs`

2.8 Creazione training, test e validation sets

Per effettuare il fine tuning degli *hyperparameters* e per addestrare successivamente il modello con Ampligraph, l'elenco delle triple ottenuto deve essere opportunamente diviso in training, test e validation set. Ampligraph mette a disposizione una funzione, *train_test_split_no_unseen*, che permette di dividere in due set l'elenco di triple secondo la percentuale indicata (entrambi variabili in ingresso alla funzione) e garantendo al contempo che il secondo set presenti solo entità e relazioni che hanno anche una occorrenza nel primo. La funzione è chiamata due volte all'interno di *_create_sets()* (righe 25-38 in *ampligraph_training.py*):

- Dall'elenco di tutte le triple ricavo inizialmente il *training set* e un secondo set, con una proporzione di 80%/20%
- La funzione viene chiamata una seconda volta sul set ottenuto precedentemente, con una proporzione di 83%/17%, ottenendo così rispettivamente il *test set* ed il *validation set*.

I tre set sono salvati all'interno di tre file all'interno della cartella *training sets* e possono essere richiamati in ogni momento tramite la funzione *_load_sets()* (righe 12-22 in *ampligraph_training.py*)

```
12 def _load_sets():
13     training_set = test_set = validation_set = None
14     if os.path.exists("./training_set/train") and os.path.exists("./training_set/test") and os.path.exists(
15         "./training_set/valid"):
16         with open("./training_set/train", "rb") as f:
17             training_set = pickle.load(f)
18         with open("./training_set/test", "rb") as f:
19             test_set = pickle.load(f)
20         with open("./training_set/valid", "rb") as f:
21             validation_set = pickle.load(f)
22     return training_set, test_set, validation_set
23
24
25 def _create_sets():
26     triplets_hp = load_from_csv(csv_folder, "triplets_hp_merged.csv", sep=",")
27     print("Number of relations: " + str(len(triplets_hp)))
28     test_val_split_size = int((len(triplets_hp) / 100) * 20)
29     training_set, test_set = train_test_split_no_unseen(triplets_hp, test_size=test_val_split_size)
30     val_split_size = int((test_val_split_size / 100) * 17)
31     test_set, validation_set = train_test_split_no_unseen(test_set, test_size=val_split_size)
32     with open("./training_set/train", "wb") as f:
33         pickle.dump(training_set, f)
34     with open("./training_set/test", "wb") as f:
35         pickle.dump(test_set, f)
36     with open("./training_set/valid", "wb") as f:
37         pickle.dump(validation_set, f)
38     return training_set, test_set, validation_set
```

Figura 7 - funzioni *ampligraph_training.py/_load_sets* e *ampligraph_training.py/_create_sets*

2.9 Scelta del modello e fine tuning degli iperparametri

Ampligraph mette a disposizione diversi modelli *Knowledge Graph Embedding* da addestrare³. Per scegliere quale di essi utilizzare ho fatto riferimento alle performance ottenute da ognuno sui più comuni datasets usati in letteratura, in particolare i valori di MRR e Hits@1,3,10 ottenuti⁴. In tutti i casi, il modello che ha ottenuto prestazioni migliori è *CompLex*, su cui è ricaduta la mia scelta. Ho effettuato quindi il fine tuning degli iperparametri, eseguendo due diverse grid search:

- La prima presenta un elevato numero di parametri diversi, con lo scopo di trovare alcuni parametri importanti come funzione di *loss*, *regularizer* e *optimizer*. È possibile trovare la configurazione all'interno del file `param_grid_1.py`.

```
1  param_grid_1 = {
2      "batches_count": [50, 100],
3      "seed": 0,
4      "epochs": [100],
5      "k": [150, 100, 50],
6      "eta": [5, 10, 15, 20, 25, 30, 35, 40, 45, 50],
7      "loss": ["pairwise", "multiclass_nll", "self_adversarial"],
8      # We take care of mapping the params to corresponding classes
9      "loss_params": {
10         # margin corresponding to both pairwise and adversarial loss
11         "margin": [0.5, 20],
12         # alpha corresponding to adversarial loss
13         "alpha": [0.5]
14     },
15     "embedding_model_params": {
16         # generate corruption using all entities during training
17         "negative_corruption_entities": "all"
18     },
19     "regularizer": [None, "LP"],
20     "regularizer_params": {
21         "p": [2],
22         "lambda": [1e-3, 1e-4, 1e-5]
23     },
24     "optimizer": ["adam"],
25     "optimizer_params": {
26         "lr": [0.01, 0.001, 0.0001]
27     },
28     "verbose": True
29 }
```

Figura 8 - Prima param grid utilizzata per fine tuning degli iperparametri

³ Elenco dei modelli disponibili: https://docs.ampligraph.org/en/1.4.0/ampligraph.latent_features.html

⁴ I dati sono disponibili al seguente indirizzo: <https://docs.ampligraph.org/en/1.4.0/experiments.html>

Con questa grid search ho fissato la scelta di diversi parametri:

- batches_count: 50
- k: 150
- loss: multiclass_nll
- regularizer: LP
- lambda: 1e-4
- lr: 0.001

L'algoritmo è stato eseguito sul mio pc con accelerazione GPU, impiegando un totale di 27 ore.

- Ho successivamente effettuato una seconda grid search, provando ad aumentare il numero di epoche da 100 a 200, e provando alcune alternative ai parametri trovati precedentemente (ad esempio rieseguire la grid search per $\lambda = [1e-4, 1e-5]$ o $lr = [1e-3, 1e-4]$). È possibile trovare la configurazione all'interno del file `param_grid_2.py`.

```
1  param_grid_2 = {
2      "batches_count": [50],
3      "seed": 0,
4      "epochs": [200],
5      "k": [150],
6      "eta": [5, 10, 15, 20],
7      "loss": ["multiclass_nll"],
8      "embedding_model_params": {
9          # generate corruption using all entities during training
10         "negative_corruption_entities": "all"
11     },
12     "regularizer": ["LP"],
13     "regularizer_params": {
14         "p": [2, 3],
15         "lambda": [1e-4, 1e-5]
16     },
17     "optimizer": ["adam"],
18     "optimizer_params": {
19         "lr": [1e-3, 1e-4]
20     },
21     "verbose": True
22 }
```

Figura 9 - Seconda param grid utilizzata per fine tuning degli iperparametri

Al termine della seconda esecuzione, che ha impiegato un totale di 1h e 30 minuti circa ad essere completata, ho ottenuto i parametri definitivi del mio modello:

- batches_count = 50
- epochs = 200
- k = 150
- eta = 10 (indica il numero di campioni negativi da generare per ogni campione positivo del training set)
- loss = multiclass_nll
- regularizer = LP
- optimizer = adam

- `embedding_model_params = {'negative_corruption_entities': 'all'}` (valore di default per ComplEx)
- `regularizer_params = {'p': 2, 'lambda': 0.0001}`
- `optimizer_params = {'lr': 0.001}`

Per entrambe le esecuzioni, ho utilizzato la funzione `grid_search_hyperparams()` che ho definito all'interno del file `ampligraph_training.py` (righe 73-109).

2.10 Creazione e predizione di triple *unseen*

I modelli addestrati con Ampligraph permettono di predire con quale probabilità una tripla non appartenente al training set sia vera. Le triple cosiddette *unseen* devono essere formate, tuttavia, da soggetti, predicati e complementi già noti al modello. Per la creazione delle triple da predire ho definito una funzione all'interno del file `ampligraph_predict.py` chiamata `create_unseen()` (righe 12-32). La generazione delle triple è randomica, ovvero dal csv contenente tutte le triple estratte precedentemente ricavo l'elenco dei soggetti, dei predicati e dei complementi, estraggo in maniera randomica da ogni lista un elemento formando una tripla mai vista (verificando che la nuova tripla non appartenga alla lista di quelle presenti nel training set). In totale sono state generate 1.000.000 di triple divisi in 10 file denominati *unseen_i*, con *i=1,2,...,9*. I file di testo sono inseriti all'interno della cartella *csv*.

```

12 def create_unseen():
13     triplets_hp = load_from_csv(csv_folder, "triplets_hp_merged.csv", sep=",")
14     random.seed(42)
15     subjects = np.unique(triplets_hp[:, 0]).tolist()
16     predicates = np.unique(triplets_hp[:, 1]).tolist()
17     objects = np.unique(triplets_hp[:, 2]).tolist()
18     triplets_hp = triplets_hp.tolist()
19     for i in range(num_gen_repetitions):
20         with open(csv_folder + "unseen" + str(i) + ".csv", "w", newline="") as f:
21             writer = csv.writer(f)
22             for _ in tqdm(range(num_gen_unseen)):
23                 added = False
24                 while not added:
25                     s = random.choice(subjects)
26                     p = random.choice(predicates)
27                     o = random.choice(objects)
28                     if s != o:
29                         triple = [s, p, o]
30                         if triple not in triplets_hp:
31                             writer.writerow(triple)
32                             added = True

```

Figura 10 - funzione `ampligraph_predict.py/create_unseen()`

Per la predizione, nello stesso file ho definito la funzione `predict_unseen()` (righe 35-46). Per ogni file precedentemente creato, per ogni tripla all'interno del file il modello assegna uno *score*, il quale può essere successivamente convertito in una probabilità compresa tra 0 e 1. Maggiore lo score, maggiore è la probabilità per cui la tripla esaminata potrebbe essere, secondo il modello, verosimile. Ho scelto di salvare

all'interno di diversi file tutte le triple con una probabilità maggiore del 98%. In totale sono stati creati 10 file denominati *predicted*i**, con *i=1,2,...,9*, con all'interno le triple *unseen* contenute nel corrispondente file *unseen*i** ritenute verosimili. I file di testo sono inseriti all'interno della cartella *csv*.

```
35 def predict_unseen():
36     os.environ['CUDA_VISIBLE_DEVICES'] = '-1'
37     model = restore_model('./training_set/model.pkl')
38     for i in range(num_gen_repetitions):
39         with open(csv_folder + "predicted"+str(i)+".csv", "w", newline="") as f_out:
40             writer = csv.writer(f_out)
41             triplets_unseen = load_from_csv(csv_folder, "unseen"+str(i)+".csv", sep=",")
42             scores = model.predict(triplets_unseen)
43             probs = expit(scores)
44             for j in range(len(probs)):
45                 if probs[j] >= 0.98:
46                     writer.writerow(triplets_unseen[j])
```

Figura 11 - funzione *ampligraph_predict.py/predict_unseen()*

2.11 SPARQL queries su DBpedia

Un URI (Uniform Resource Identifier) è una sequenza di caratteri che identifica in modo univoco una risorsa. Nel caso in esame, ho cercato laddove possibile di associare ad ogni soggetto, predicato e complemento di ogni tripla (sia quelle del training set sia quelle predette) l'URI della corrispondente risorsa su DBpedia, un progetto nato con lo scopo di estrarre informazioni strutturate da Wikipedia e trasporle come *linked data* in formato *RDF*. Per ricercare una risorsa all'interno del database, si utilizza un linguaggio di query simile a *SQL* chiamato *SPARQL*. Tutte le funzioni utili allo scopo sono contenute all'interno del file *SPARQL_query.py*. La funzione principale è chiamata *query(element)* (righe 22-43). Essa riceve l'elemento da ricercare in input in formato stringa, ed effettua le seguenti query in ordine, fermandosi alla prima che restituisce un risultato valido:

- "SELECT ?s WHERE {{?s rdfs:label *element*@en ; a owl:Thing .}}", con cui si ricercano tutte le risorse la cui *label* è uguale ad *element* (si noti il tag @en per ricercare risorse in lingua inglese) e il cui *rdf:type* sia pari a *owl:Thing*
- "SELECT ?s WHERE {{?altName rdfs:label *element*@en ; dbo:wikiPageRedirects ?s .}}", con cui si ricercano le risorse il cui nome alternativo sia pari ad *element* e che reindirizza alla risorsa cercata (ad esempio ricercando Harry Potter, digitato in modo non corretto con una t in meno, si viene reindirizzati sulla pagina della risorsa Harry Potter)
- "SELECT ?s WHERE {{?s rdfs:label *element*@en ; a skos:Concept .}}", simile alla prima query ma cercando risorse il cui tipo sia *skos:Concept*.

```

8  def _execute_query(query_string):
9      sparql = SPARQLWrapper("https://dbpedia.org/sparql")
10     sparql.setQuery(query_string)
11     sparql.setReturnFormat(XML)
12     try:
13         qres = sparql.query().convert()
14     except:
15         return None
16     results = qres.getElementsByTagName("uri")
17     if len(results) > 0:
18         return str(results[0].firstChild.nodeValue)
19     return None
20
21
22 def query(element):
23     query_string = "SELECT ?s WHERE {{?s rdfs:label \"\" + element + "\"@en ;a owl:Thing .}}"
24     result = _execute_query(query_string)
25     if result is not None:
26         return _check_result(element, result)
27     else:
28         query_string = "SELECT ?s WHERE {{?altName rdfs:label \"\" + element + "\"@en ;dbo:wikiPageRedirects ?s .}}"
29         result = _execute_query(query_string)
30         if result is not None:
31             return _check_result(element, result)
32         else:
33             query_string = "SELECT ?s WHERE {{?s rdfs:label \"\" + element + "\"@en ;a skos:Concept .}}"
34             result = _execute_query(query_string)
35             if result is not None:
36                 return _check_result(element, result)
37             else:
38                 uri = _search_page(element)
39                 if uri is not None:
40                     return _check_result(element, str(uri))
41                 else:
42                     return None
43     return None

```

Figura 12 - funzioni SPARQL_query.py/query e SPARQL_query.py/_execute_query

Laddove nessuna query restituisca un risultato, la causa potrebbe risiedere all'interno di *element*. Infatti, alcune risorse presentano un suffisso in caso di disambiguazione (ad esempio ricercando "Hedwig", la civetta di Harry, non si ottiene nessun risultato poiché il nome della risorsa è "Hedwig_(Harry_Potter)"). Per aumentare il numero di risorse individuate, ho pensato di ricorrere a un secondo sistema diverso dalle query: all'interno della pagina https://dbpedia.org/page/Magical_creatures_in_Harry_Potter è possibile trovare un elenco completo delle creature ed entità presenti in Harry Potter, sotto forma di tag HTML anchor (<a>). Tramite la funzione `_search_page(element)` (righe 46-62) ricerco all'interno della pagina un tag anchor la cui label contiene la stringa *element* data in input. In caso positivo, mi restituisce l'attributo *href* che corrisponde all'URI desiderato.

```

46 def _search_page(element):
47     headers = {
48         'Access-Control-Allow-Origin': '*',
49         'Access-Control-Allow-Methods': 'GET',
50         'Access-Control-Allow-Headers': 'Content-Type',
51         'Access-Control-Max-Age': '3600',
52         'User-Agent': 'Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:52.0) Gecko/20100101 Firefox/52.0'
53     }
54     url = "https://dbpedia.org/page/Magical_creatures_in_Harry_Potter"
55     req = requests.get(url, headers)
56     soup = BeautifulSoup(req.content, 'lxml')
57     elements = soup.find_all('a')
58     exp = ".*:" + element + ".*"
59     for element in elements:
60         if element.find(string=re.compile(exp)):
61             return element['href']
62     return None

```

Figura 13 - funzione SPARQL_query.py/_search_page

Non tutti gli elementi possiedono una propria pagina su DBpedia, ad esempio ricercando “Aragog” (il ragno gigante custodito da Hagrid) si viene ricondotti alla pagina di Rubeus Hagrid. Ho dunque implementato una funzione chiamata `_check_result(element, result)`, che riceve in ingresso sia l’elemento ricercato che il risultato (l’URI) ottenuto precedentemente dalla query o dalla ricerca del tag anchor. Confronto la parte terminale dell’uri con la stringa `element` utilizzando `SequenceMatcher` della libreria `difflib`: se esse risultano simili con un tasso maggiore del 50%, l’URI resta inalterato, altrimenti viene aggiunta la stringa `#element` (ad esempio nel caso precedente di Aragog si ottiene l’uri `https://dbpedia.org/page/Rubeus_Hagrid#Aragog`, mentre per l’elemento “Voldemort” si ottiene `https://dbpedia.org/page/Lord_Voldemort` poiché essendo la somiglianza maggiore del 50% non viene aggiunta nessuna stringa al termine dell’uri).

```

65 def _similar(a, b):
66     return SequenceMatcher(None, a, b).ratio()
67
68
69 def _check_result(element, result):
70     sub_result = result.replace("http://dbpedia.org/resource/", "")
71     sub_result = sub_result.replace("_ (Harry_Potter)", "")
72     if _similar(element, sub_result) >= 0.5:
73         return result
74     else:
75         return result + "#" + str(element).replace(" ", "_")

```

Figura 14 - funzioni SPARQL_query.py/_check_result e SPARQL_query.py/_similar

2.12 Creazione di un grafo Node4J ed inserimento delle risorse

Neo4J è un sistema di gestione di *graph database*, ovvero di database che usano la struttura a grafi per queries semantiche con nodi, archi e proprietà. Ho dunque popolato un grafo con le triple ottenute in precedenza, grazie ad una libreria che mette in interazione *RDFLib* con un database Neo4j tramite il plugin *n10s*⁵. Le funzioni necessarie al task sono presenti nel file `neo4j_utils.py`. Ho definito innanzitutto la funzione

`create_graph()` (righe 6-11) per la creazione dell'oggetto di classe *Graph*: tramite la libreria sopra citata è possibile connettersi ad un database di Neo4j (nel caso in esame ho creato un database in locale) indicando uri, nome del database e le credenziali, username e password, necessarie

```
6 def create_graph():
7     g = Graph(store="Neo4j")
8     config = {'uri': uri, 'database': database_name,
9             'auth': {'user': auth_user, 'pwd': auth_pwd}}
10    g.open(config, create=False)
11    return g
```

Figura 15 - funzione `neo4j_utils.py/create_graph()`

al collegamento. La funzione restituisce dunque il grafo configurato. Successivamente utilizzo la funzione `add_triple(graph, s, p, o, dict_elements={})` (righe 26-27) per popolarlo. Essa riceve in input diversi parametri:

- L'oggetto *Graph* a cui aggiungere le risorse
- Soggetto (*s*), predicato (*p*) e complemento (*o*) della tripla da aggiungere
- `dict_elements`, un dizionario il cui utilizzo approfondirò in seguito (di default è pari ad un dizionario vuoto)

Le risorse di un grafo devono essere necessariamente identificate da un URI tramite l'oggetto *URIRef* di *RDFLib*. Ho dunque implementato la funzione `_create_uri(string, dict_elements={})` che permette di ottenere un URI valido per *s*, *p* ed *o* nel seguente modo:

- effettua una query SPARQL a DBpedia tramite le funzioni descritte al paragrafo precedente cercando la risorsa e URI corrispondente
- in caso di insuccesso, crea un URI del tipo `http://example.com/HP#element`, sostituendo eventuali spazi in *element* con il carattere `_`.

Per aumentare il numero di soggetti individuati correttamente, ho creato manualmente un dizionario all'interno del file `references.py`, che mappa alcuni elementi con la stringa corretta da ricercare (ad

```
14 def _create_uri(string, dict_elements={}):
15     base_uri = "http://example.com/HP#"
16     e1 = str(string).lower().title()
17     if e1 in dict_elements.keys():
18         e1 = dict_elements[e1]
19     res = query(e1)
20     if res is not None:
21         return URIRef(str(res))
22     else:
23         return URIRef(base_uri + str(string).lower().title().replace(" ", "_"))
```

Figura 16 - funzione `neo4j_utils.py/_create_uri`

esempio all'elemento "Harry" è associata la stringa "Harry Potter" o "W. Weasley" con "Arthur Weasley"). In questo modo ho esteso il numero di elementi correttamente individuati tramite query a DBpedia. Questo dizionario è passato alla funzione e corrisponde alla variabile `dict_elements` nominata precedentemente. Tramite queste funzioni ho popolato il grafo con le triple appartenenti sia al training set che quelle predette precedentemente.

⁵ [neo4j-labs/rdflib-neo4j](https://github.com/neo4j-labs/rdflib-neo4j): RDFLib Store backed by neo4j + n10s (github.com)

3. Analisi sperimentale del sistema

3.1 Analisi quantitative dei risultati

Con i task descritti nei paragrafi 2.1-2.7 sono state ricavate **1385** relazioni (presenti nel file `triplets_hp_merged.csv`). Di queste, **1346** sono ritenute valide ed uniche dalla funzione `train_test_split_no_unseen` di Ampligraph, e solo **39** sono state scartate. Con le percentuali descritte precedentemente, i campioni sono stati così divisi:

- **1077** relazioni nel **training set**
- **224** relazioni nel **test set**
- **45** relazioni nel **validation set**.

Il modello *ComplEx* ha ottenuto i seguenti risultati:

- MRR: 0.27
- Hits@10: 0.32
- Hits@3: 0.27
- Hits@1: 0.24

I valori sono in linea con quanto ci si aspettasse di ottenere con un dataset estremamente variegato, ottenuto da riassunti di libri di narrativa che non presentano informazioni di sfondo (ad esempio dei personaggi o dell'ambientazione) e in maniera puramente algoritmica senza una rifinitura da parte umana.

Tra il milione di triple *unseen* generate in maniera casuale, solo **102** (ovvero lo 0.0102%) hanno ottenuto una probabilità pari o superiore al 98% di essere vere, per cui il modello è stato molto selettivo ed è indice di un buon apprendimento del contesto.

Di seguito invece le statistiche sui riscontri degli elementi tramite query a DBpedia:

- 136 su 417 soggetti (32.61%)
- 121 su 421 predicati (28.74%)
- 221 su 899 complementi (24.58%)

Buona parte dei complementi è composto da più parole o parti di frase; dunque, era lecito aspettarsi la percentuale più bassa di *match*. Buon riscontro invece tra i soggetti ed i predicati.

3.2 Analisi qualitative dei risultati

Buona parte delle triple estratte presentano soggetti, predicati e complementi composti da più parole o intere parti di frase. Questo problema, tuttavia, è tipico nelle soluzioni open source o free to use esistenti. È stato in ogni modo mitigato quanto possibile con tutte le pre-elaborazioni del testo descritte nel capitolo precedente. Per migliorare ulteriormente il risultato, oltre ad intervenire manualmente, si potrebbe addestrare specificatamente il modello NLP usato sul contesto dell'opera analizzata oppure effettuare una analisi di triple con elementi simili per estrarre i termini chiave.

298	Hermione	secretly brews	Polyjuice Potion
299	Hermione	brews	Polyjuice Potion allowing
300	Hermione	secretly brews	Polyjuice Potion allowing

Figura 17 - esempio di triple con elementi in comune

Nell'esempio riportato nell'immagine sopra, potremmo estrarre dai soggetti, dai predicati e dai complementi le parole che si ripetono ottenendo dunque "*Hermione, brews, Polyjuice Potion*". Tuttavia queste analisi sono

al di fuori dello scopo del progetto, dove si è cercato di ideare e mettere alla prova un sistema completamente automatico senza ulteriori interventi.

Di seguito è riportata una panoramica del grafo Node4j ottenuto:

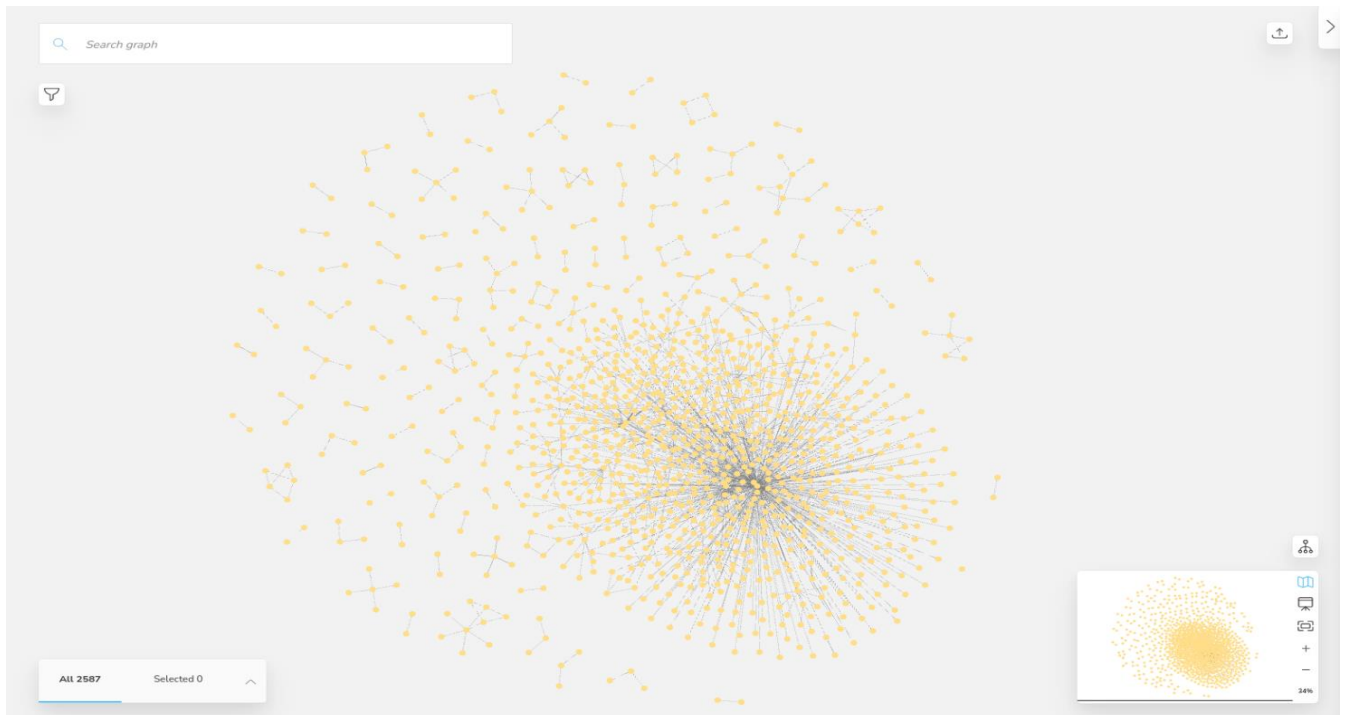


Figura 18 - panoramica del grafo Node4j

è possibile notare la presenza di un grande cluster principale di elementi, circondati da un numero minore di relazioni isolate. Dalle due immagini a seguire è possibile notare come al centro del cluster ci sia il protagonista del libro, Harry Potter, e il suo principale antagonista, Lord Voldemort, come era lecito aspettarsi.

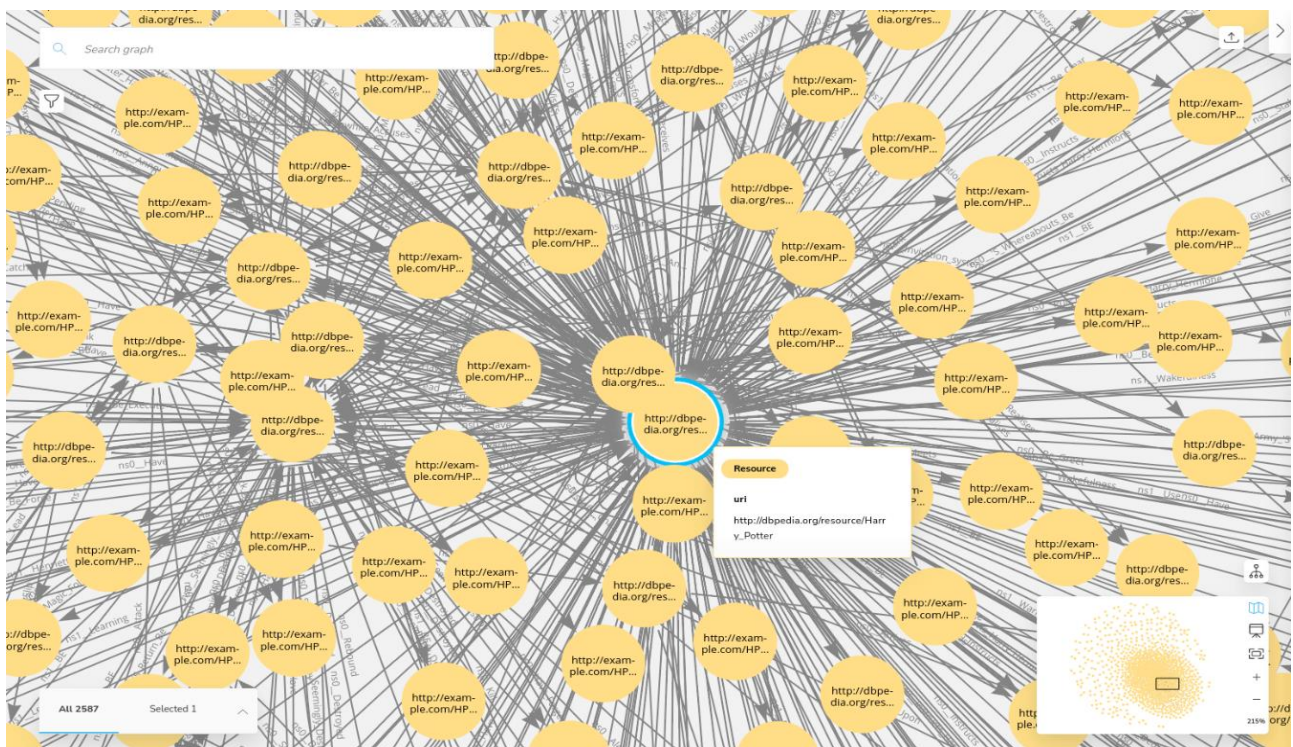


Figura 19 - dettaglio del grafo Node4j (1)

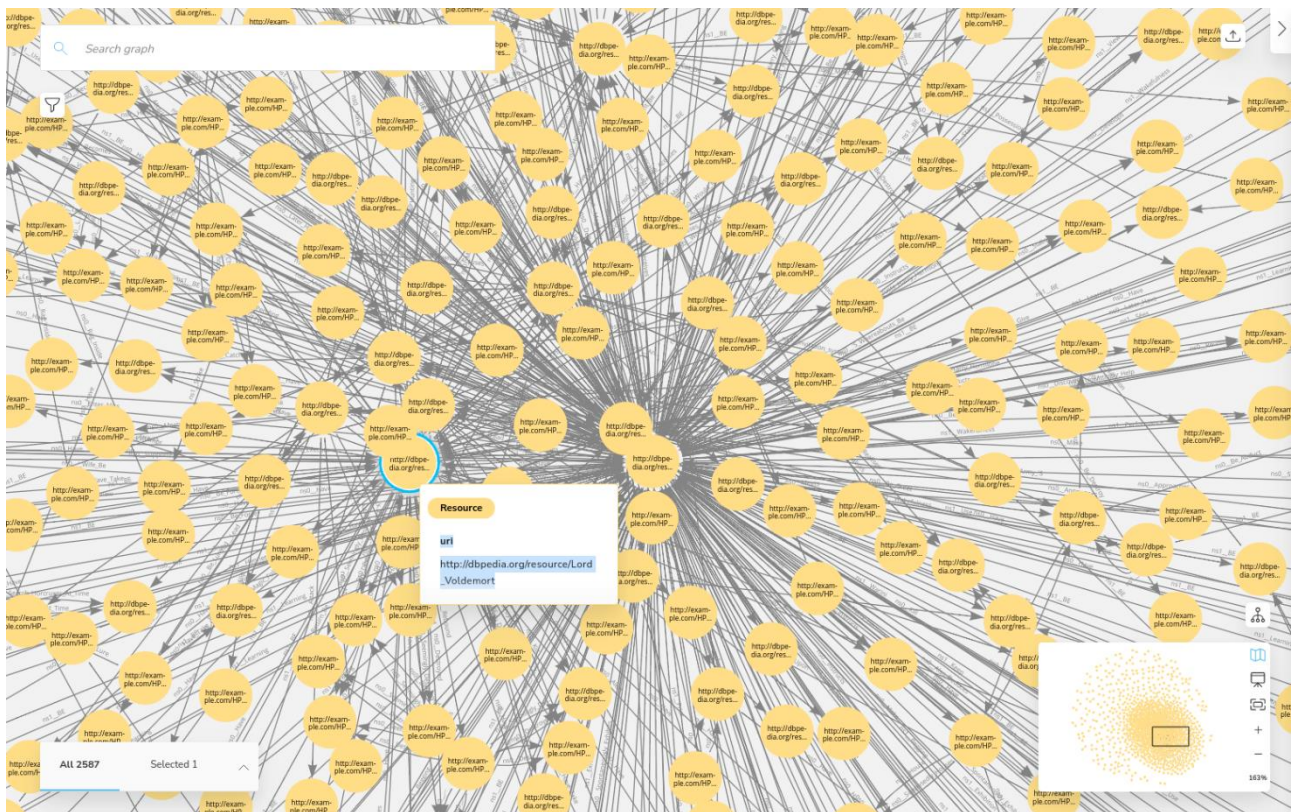


Figura 20 - dettaglio del grafo Node4j (2)

Indice delle figure

FIGURA 1 - ESEMPIO DI COREFERENCE RESOLUTION.....	4
FIGURA 2 - FUNZIONE NLTK_UTILS.PY/REMOVE_STOPWORDS.....	5
FIGURA 3 - FUNZIONE NLTK_UTILS.PY/LEMMATIZE_TRIPLETS_ONLY_VERBS	6
FIGURA 4 - FUNZIONE NLTK_UTILS.PY/GET_WORDNET_POS.....	6
FIGURA 5 - FUNZIONE MAIN.PY/TRIPLETS_TO_CSV.....	7
FIGURA 6 - FUNZIONE MAIN.PY/CREATE_CSVS	7
FIGURA 7 - FUNZIONI AMPLIGRAPH_TRAINING.PY/_LOAD_SETS E AMPLIGRAPH_TRAINING.PY/_CREATE_SETS	8
FIGURA 8 - PRIMA PARAM GRID UTILIZZATA PER FINE TUNING DEGLI IPERPARAMETRI	9
FIGURA 9 - SECONDA PARAM GRID UTILIZZATA PER FINE TUNING DEGLI IPERPARAMETRI	10
FIGURA 10 - FUNZIONE AMPLIGRAPH_PREDICT.PY/CREATE_UNSEEN().....	11
FIGURA 11 - FUNZIONE AMPLIGRAPH_PREDICT.PY/PREDICT_UNSEEN().....	12
FIGURA 12 - FUNZIONI SPARQL_QUERY.PY/QUERY E SPARQL_QUERY.PY/_EXECUTE_QUERY	13
FIGURA 13 - FUNZIONE SPARQL_QUERY.PY/_SEARCH_PAGE.....	14
FIGURA 14 - FUNZIONI SPARQL_QUERY.PY/_CHECK_RESULT E SPARQL_QUERY.PY/_SIMILAR	14
FIGURA 15 - FUNZIONE NEO4J_UTILS.PY/CREATE_GRAPH()	15
FIGURA 16 - FUNZIONE NEO4J_UTILS.PY/_CREATE_URI	15
FIGURA 17 - ESEMPIO DI TRIPLE CON ELEMENTI IN COMUNE	16
FIGURA 18 - PANORAMICA DEL GRAFO NODE4J.....	17
FIGURA 19 - DETTAGLIO DEL GRAFO NODE4J (1)	17
FIGURA 20 - DETTAGLIO DEL GRAFO NODE4J (2)	18