Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

PARLEN: uno strumento modulare per l'analisi di articoli e il riconoscimento di entità

Relatore:
Dott.
ANGELO DI IORIO

Presentata da:
GABRIELE DE LUCA

Prima Sessione
2015/2016

a chi ha saputo amarmi
incondizionatamente...

Introduzione

Una grande parte dei contenuti del web del 2016 è testuale: attraverso articoli piuttosto che ricerche o pubblicazioni, da anni, tra blog e giornali vanno diffondendosi dati ed informazioni. Per quanto però un essere umano leggendo un un testo è in grado di coglierne il significato, dal punto di vista di una macchina le cose cambiano: il dato nudo e crudo che rappresenta il testo ha contenuto semantico ma quest'ultimo non è esplicitato e formalizzato in modo che un agente software sia in grado di processarlo. Tim Berners-Lee ha risolto brillantemente il problema ideando il *Semantic Web*[1], una evoluzione del World Wide Web in un ambiente in cui i documenti pubblicati sono associati a dati ed informazioni che ne descrivono il contenuto semantico. In questo ambiente, i dati sono strutturati e possono essere meglio interrogati dai motori di ricerca o interpretati da elaborazioni automatiche.

Le potenzialità che derivano dalla rappresentazione semantica delle informazioni sono innumerevoli. Un esempio lampante è quella funzione offerta dai siti web più moderni che ci propongono contenuti simili a quello che stiamo già visualizzando, a dimostrazione del fatto che anche un elaboratore è in grado di riconoscere gli argomenti piuttosto che le persone o, più in generale, le entità citate all'interno di un testo.

Un contesto in cui risulterebbe molto interessante operare un'analisi semantica è quello dei giornali: oramai tutti quanti pubblicano online i propri articoli ed i quotidiani più noti rendono il loro archivio storico accessibile in rete, talvolta gratuitamente. Per esempio, sul sito dell'archivio storico di Repubblica[2], è possibile consultare tutti gli articoli pubblicati sia sul sito stesso che sulla versione cartacea del giornale, a partire dal 1984. Se provassimo ad analizzare anche solo una minima parte di questa mole di dati, potremmo ottenere una serie di informazioni supplementari che in un certo senso erano

già presenti nei testi, ma che magari non erano immediatamente percettibili.

Pensiamo ad esempio alla tematica dell'*inflazione*, argomento tutt'altro che moderno, sul quale si è parlato, si parla e si parlerà tanto. Partendo da tutti gli articoli di Repubblica che parlano di inflazione e con l'aiuto di un algoritmo, si potrebbero ricercare nei testi persone, organizzazioni, luoghi ed entità in generale, dando origine ad un dataset che potremmo in seguito interrogare per scoprire, ad esempio, quante volte è stato citato *Mario Draghi* in articoli pubblicati da Repubblica in cui si parla di inflazione. Se poi si pensa che l'analisi può essere operata su una qualsiasi sorgente (anche diversa da Repubblica) e che le ricerche possono essere ulteriormente filtrate (ad esempio per data) ed intrecciate tra più giornali, è facilmente intuibile che le possibilità d'analisi diventano innumerevoli.

Inoltre, le entità riconosciute potrebbero essere collegate ad altre risorse, come ad esempio la pagina *Wikipedia* o il profilo *Facebook* di un dato personaggio o di una data società. Questo grazie a *Linked Open Data*[3], un progetto di W3C che si propone di estendere il web con diversi dataset RDF che aggiungono informazioni semantiche ai contenuti, che a loro volta risultano essere collegati tra loro in maniera molto più sofisticata ed espressiva rispetto al semplice collegamento ipertestuale.

Il lettore potrebbe quindi effettuare una lettura interattiva con entità correlate a risorse esterne e con dati supplementari che completano ed estendono l'informazione iniziale.

In questo documento, vado a spiegare come ho implementato *PARLEN* (Pipeline for Automatic Recognition and Linking of Entities in Newspapers), un'applicazione modulare che permette appunto l'analisi di articoli di giornale (o di testi in generale), il riconoscimento e l'estrazione di entità dagli stessi e la produzione di una ontologia che descrive in maniera formale il contenuto dei testi analizzati, consentendo la possibilità di ulteriori approfondimenti ed interrogazioni. Nello specifico, PARLEN è in grado di analizzare articoli di giornale provenienti dagli archivi di Repubblica e Sole 24 Ore[4], in quanto sarà utilizzato in primissima battuta da alcuni ricercatori dell'Università di Bologna che necessitano di condurre alcuni approfondimenti su articoli pubblicati da questi due quotidiani.

Ad ogni modo, la modularità di PARLEN permette una facile integrazione con altre sorgenti di dati, altri giornali o altri testi in generale.

Inoltre, PARLEN permette anche la visualizzazione interattiva degli articoli (o dei testi) che analizza, generandone una versione interattiva in cui le entità riconosciute sono cliccabili e collegate a alla loro pagina DBpedia.

L'ontologia prodotta è in formato RDF: interrogandola è possibile condurre un'analisi estremamente approfondita, soprattutto se i dati vengono confrontati ed estesi con quelli offerti da Linked Open Data.

Si deve considerare che il processo di analisi ed estrazione di contenuti si suddivide in fasi ed ogni fase ha i suoi punti critici che possono contribuire ad ottenere un output indesiderato. A partire dalla sorgente di dati, che può presentare errori strutturali o ortografici fino ad arrivare ad alcune ambiguità che possono verificarsi nel momento in cui vengono estratte le entità (Es. "Presidente del Consiglio", cambia a seconda dell'anno di pubblicazione dell'articolo).

Tuttavia vedremo in cosa PARLEN eccelle ed anche in cosa non eccelle; pregi e difetti, problemi già risolti e possibili migliorie, possibili espansioni, vista l'architettura modulare.

Nel primo capitolo approfondiremo il concetto di Semantic Web, vedendo come si possono rappresentare i dati in RDF. Analizzeremo inoltre pregi e difetti dei più importanti tool che si occupano di analisi del linguaggio naturale e di riconoscimento ed estrazione di entità.

Nel secondo capitolo vedremo quali sono i moduli che compongono PARLEN, come comunicano tra di loro, input, output e vedremo anche come si fa ad utilizzare l'applicazione.

Nel terzo capitolo scenderemo nei dettagli dell'implementazione di ogni modulo, vedremo le tecnologie con cui PARLEN è sviluppato e le librerie più importanti utilizzate.

Nel quarto, faremo alcuni test approfonditi e ne analizzeremo e valuteremo risultati.

Infine, nella conclusione vedremo anche possibili sviluppi futuri dell'applicazione.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Lo stato dell'arte

1.1 Semantic Web e Linked Open Data

Possiamo pensare al Semantic Web come ad un ambiente in cui le informazioni possono essere descritte in un formato comprensibile da una macchina. Il primo linguaggio con cui esse furono rappresentate è stato l'RDF[5] (Resource Description Framework), una particolare applicazione XML. Esso è stato poi esteso da OWL[6], una variante che ne estende le potenzialità.

La rappresentazione dell'informazione è operata ispirandosi a principi della logica del primordine secondo cui l'informazione può essere espressa ricorrendo all'utilizzo di asserzioni, triple costituite da soggetto, predicato e valore. Gli elementi che compongono la tripla sono indicati da URI che identificano univocamente la risorsa e possono essere scelti arbitrariamente.

Vediamo un esempio concreto considerando la seguente asserzione:

Barack Obama vive in America

- Soggetto: **Barack Obama** (http://it.wikipedia.org/wiki/Barack_Obama)
- Predicato: **vive in** (http://it.wiktionary.org/wiki/vivere)
- Valore: **America** (http://it.wikipedia.org/wiki/America)

Dalla nascita del Semantic Web sono nate numerose ontologie in RDF che permettono di descrivere un'ampia quantità di informazioni. Una tra le più note è FOAF[7], un'ontologia che serve a descrivere le persone e le relazioni che hanno con altre persone, oggetti ed attività. L'accesso all'ontologia è libero in quanto Open Source.

FOAF è inoltre parte del progetto Linked Open Data. Quest'ultimo, come accennato, è un progetto di W3C Semantic Web Education e si propone di estendere il web pubblicando e collegando tra di loro dataset RDF Open Source. Un altro dei dataset facenti parte del progetto è quello di DBpedia, che da solo contiene 3.4 milioni di concetti, espressi in 1 bilione di triple, in 11 lingue differenti. I numeri dell'intero progetto, poi, sono interessantissimi: nel 2011 si contavano 31 bilioni di triple RDF[3]. Numeri che oggi non possono essere che saliti.

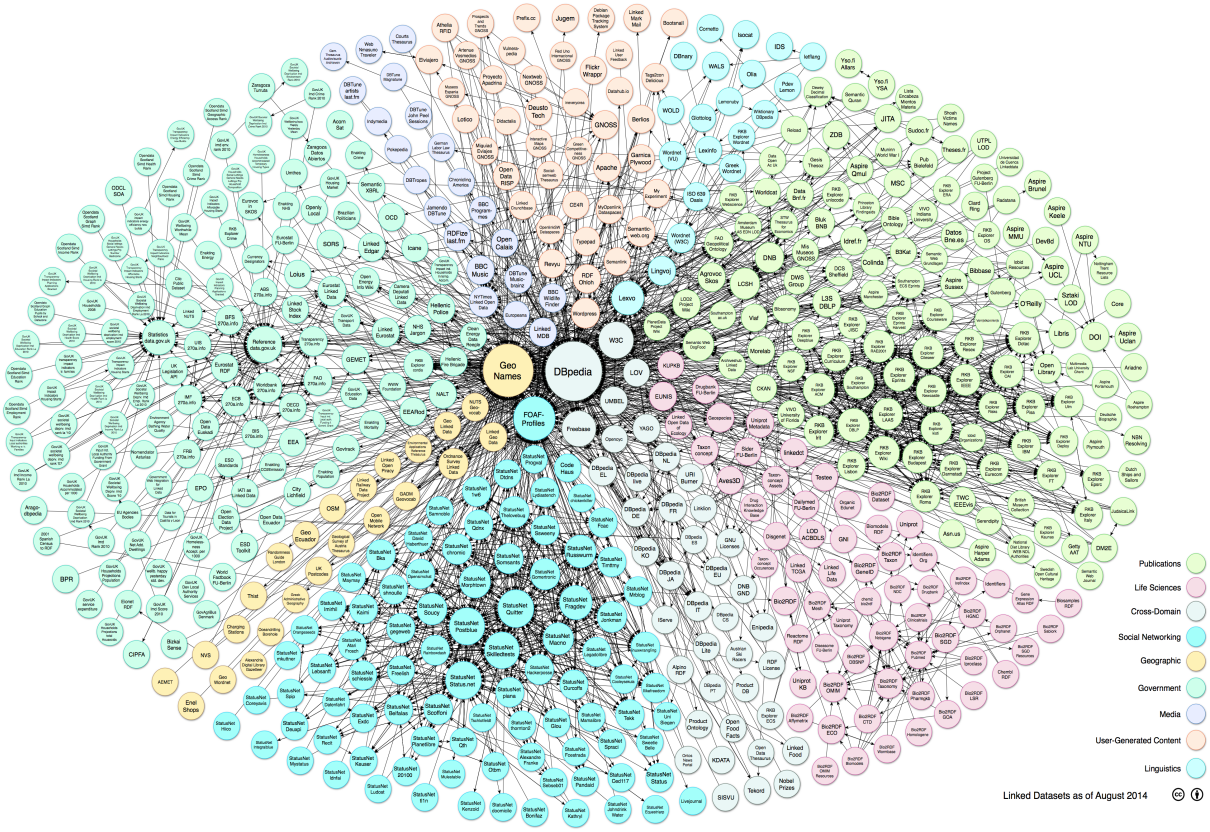


Figura 1.1: Il diagramma mostra i dataset del progetto LOD e come sono connessi. Agosto 2014

Davanti a queste cifre è facile comprendere quante possibilità si aprono. Aziende, privati e ricercatori possono utilizzare questa mole di dati per scopi diversi; inoltre, tool per l'analisi del linguaggio naturale possono riconoscere un enorme quantitativo di entità utilizzando i datasets di Linked Open Data come sorgente di dati.

1.2 Il Semantic Web nei giornali online

I siti web dei quotidiani costituiscono il contesto ideale per l'implementazione di alcune funzionalità intelligenti che sfruttano le potenzialità offerte dal Semantic Web.

Infatti, un articolo di giornale è tipicamente un testo in cui compaiono riferimenti a fatti, persone, luoghi, organizzazioni, eventi o, come abbiamo definito più genericamente, entità. Attraverso il riconoscimento di tali entità e l'utilizzo di database interni ma anche dei Linked Open Data, i siti web dei quotidiani costruiscono un sistema di contenuti correlati, esortando il lettore alla lettura di contenuti simili a quelli visualizzati di solito.

Inoltre, sul sito di Repubblica[8], ma anche su quello del Sole 24 Ore[9] o in quello del Corriere della Sera, troviamo riferimenti sul numero di volte che entità riconosciute come tali vengono citate, sia in tutto il giornale che quando si parla di un argomento specifico. Tuttavia, questo dato risulta essere impreciso in alcuni casi. Consideriamo ad esempio il seguente screenshot.

Figura 1.2: Sezione "entità correlate" su repubblica.it, 2016

A giudicare da esso, pare che *Silvio Berlusconi* sia stato citato 56 volte da quando il giornale è nato. Il dato è sicuramente errato e non è comprensibile a cosa faccia riferimento il numero 56.

Sul sito del Sole 24 Ore, invece, la stessa analisi sembra essere eseguita meglio. Inoltre pare che il l'algoritmo riesca a riconoscere più entità rispetto a quello di Repubblica, fornendo un elevato numero di tag e suddividendo i risultati in maniera più intuitiva ed organizzata.

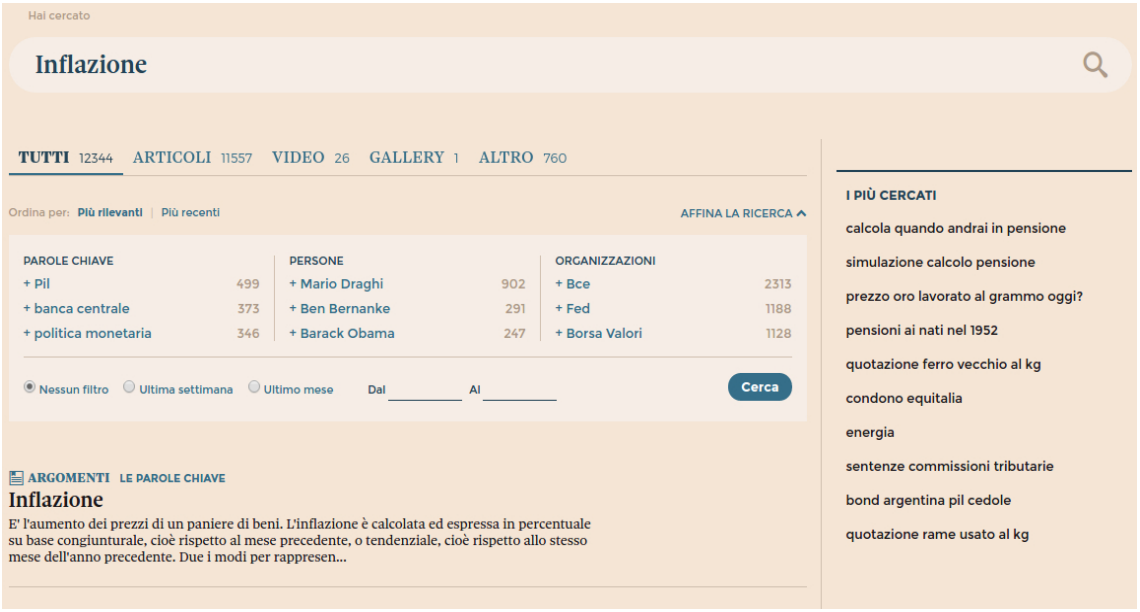


Figura 1.3: Sezione "entità correlate" su sole24ore.it, 2016

Ad ogni modo, con soli questi dati, relativi alla sola ricerca in corso, non si può pensare di condurre un'analisi approfondita, anche se, per carità, non è questo il compito dei giornali. Le informazioni supplementari ed i collegamenti veloci a persone o enti servono solo ad aiutare la navigazione del lettore e non sono sicuramente sufficienti per condurre delle analisi specifiche supplementari.

Per la conduzione di un'analisi approfondita, abbiamo bisogno di un software specializzato.

1.3 Strumenti per l'analisi dei testi ed il riconoscimento di entità

PARLEN non si occupa direttamente di riconoscere e collegare entità da un testo; quest'operazione infatti è pensata per essere eseguita da un servizio esterno fornito da uno dei tanti tool specializzati in questo tipo di analisi. La scelta è piuttosto vasta ed a seconda delle esigenze si può preferire un tool piuttosto che un altro.

Io ho considerato tool che come requisito fondamentale fossero Open Source, anche se esistono dei software proprietari in grado si svolgere un lavoro eccellente. Inoltre, un'architettura modulare è un altro requisito di cui ho tenuto particolarmente conto: la maggior parte dei software che gestiscono contenuto semantico forniscono diversi servizi specializzati. C'è il modulo che analizza il contenuto da un punto di vista morfologico-grammaticale, il modulo che si occupa di estrarre entità, quello che le collega a risorse esterne; poi, in alcuni casi, ci sono moduli che permettono di salvare e gestire, su una base di dati persistente, contenuti semantici precedentemente elaborati o forniti in input dall'utente; ed infine, moduli per la visualizzazione facilitata dei risultati dell'elaborazione.

Queste sono solo alcune delle caratteristiche e dei servizi forniti dai tool più completi. Per lo scopo di PARLEN è molto comodo poterne utilizzarne solo alcuni. In particolare, ci interessa affidarci ad uno o più servizi esterni che possano prendere in input del testo e ci possano dare in output le entità riconosciute, con collegamento a risorsa esterna (es. DBpedia).

Ultimo requisito, quello più importante: supporto alla lingua italiana.

Vediamo quali sono i principali software che possiamo utilizzare in questo momento.

1.3.1 Apache Stanbol e CELI

Si tratta di un insieme di componenti software accessibili attraverso interfacce RESTful e che forniscono dei servizi semantici[10].

Esso è pensato per grosse applicazioni che gestiscono contenuti testuali ma l'architettura modulare ne permette l'utilizzo anche per analisi su set di dati ridotti ed isolati.

Supporta la lingua inglese, italiana, francese, spagnola e tedesca. I suoi moduli principali sono:

- **Content Enhancement**: servizio per aggiungere informazioni semantiche a contenuti, accessibili via SPARQL, che ne sono privi.
- **Reasoning**: servizio in grado di ottenere informazioni semantiche aggiuntive relative ai contenuti ottenuti precedentemente dal servizio di Content Enhancement.
- **Knowledge Models**: servizio che permette la manipolazione dei modelli di dati usati per immagazzinare informazioni semantiche.
- **Persistence**: servizio che permette il salvataggio e la ricerca di informazioni semantiche.

Il primo, forse anche il secondo servizio possono tornare sicuramente molto utili a PARLEN Tuttavia, il supporto alla lingua italiana è solo parziale. Infatti, per poter utilizzare una parte fondamentale del primo modulo, quella che opera la NER (Named entity Recognition), è obbligatorio affidarsi ad un servizio esterno a Stanbol, il Celi Ner Engine. Celi[11] è una società che realizza tecnologie semantiche. Fornisce, gratuitamente per scopi di ricerca, un prodotto (Linguagrid[12]) compatibile con l'architettura di Stanbol.

In prospettiva, pensando a PARLEN e Stanbol, l'architettura definitiva pare essere particolarmente complessa. Inoltre, non essendoci una demo online e non potendo utilizzare il modulo con la lingua italiana, non ho avuto modo di installarlo e provarlo personalmente.

1.3.2 Dandelion API e SpazioDati

Dandelion API[13] è uno dei migliori servizi per analisi testuale e ricerca di contenuti semantici attualmente in circolazione ed è un orgoglio tutto italiano, sviluppato da SpazioDati[14], una S.r.l. con sede a Trento.

Supporta l'inglese, l'italiano, il francese, il portoghese ed il tedesco. Non è Open Source. Tuttavia, l'utilizzo delle API è gratuito fino a circa 800 query al giorno[15]. Per questo motivo non ho potuto integrarlo con PARLEN

Le sue potenzialità sono subito chiare: ogni componente è infatti testabile utilizzando una fantastica web demo online[16]. L'ho voluto testare a fondo, soprattutto nel suo servizio che, a mio avviso, funziona meglio: l'Entity Extraction.

Analizziamo le potenzialità e gli aspetti più importanti dei servizi offerti.

Entity Extraction

Prende in input un testo e ne estrae le entità, collegandole a DBpedia. Sembra il solito componente comune a tutti i tool in circolazione ma ha delle piccole funzioni che lo rendono unico.

Si può stabilire un parametro che va da 1 a 10 e che indica la *confidence*. Attraverso esso, si riescono a gestire in maniera ottimale i falsi positivi ed i falsi negativi. In pratica, più il parametro è vicino al 10, più la computazione sarà "pignola": in questo caso il numero di entità riconosciute diminuisce ma la probabilità che le entità estratte siano dei falsi positivi sono veramente basse. In caso contrario, quando il parametro tende ad 1, il numero di entità estratte sale drasticamente, così però come il numero di falsi negativi. Dopo alcuni test mi sono accorto che un valore tra il 7 e l' 8 è il compromesso migliore, almeno con la lingua italiana.

Un'altra caratteristica interessante riguarda la *disambiguazione*: l'Extractor di Dandelion opera sicuramente una fase preliminare in cui distingue i vari lemmi dal punto di vista lessicale, grammaticale e semantico (quando possibile) ed attraverso cui costruisce un contesto per disambiguare entità ambigue. Se l'input è: *"Roberto Baggio giocava a calcio"*, Dandelion riconoscerà "Roberto Baggio" come persona ed il "calcio" come sport. Se l'input però diventa: *"Il latte contiene calcio"*, l'entità "calcio" sarà riconosciuta come elemento chimico.

In questa tipologia di tool è fondamentale che esista una procedura per la disambiguazione e che funzioni bene; quella di Dandelion è in assoluto la migliore.

In output è inoltre fornito un dato particolare: l'*accuracy*, un numero che va da 0 ad 1 associato ad ogni entità estratta: quando tende ad uno, indica una totale fiducia da parte del software della positività dell'istanza.

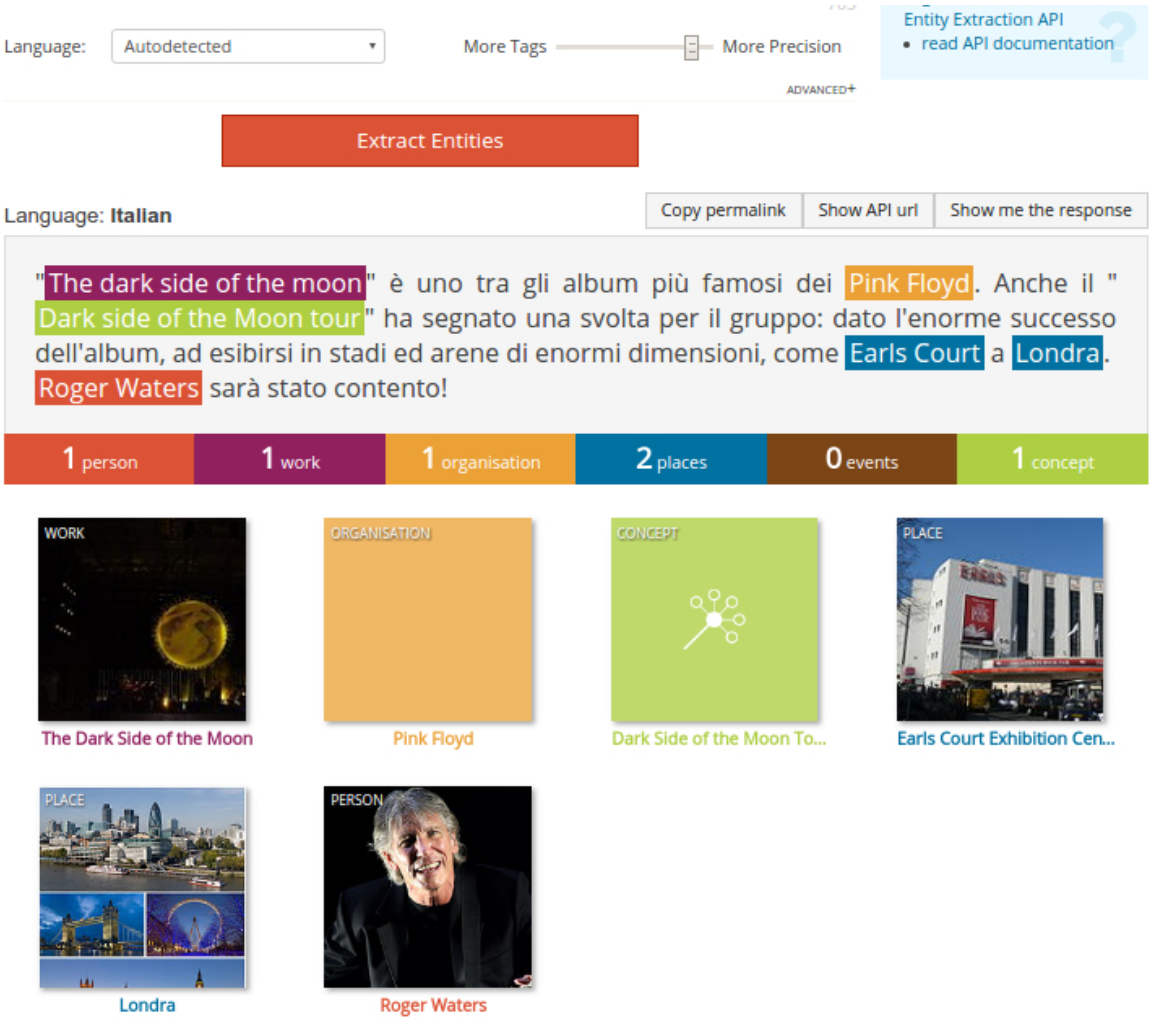


Figura 1.4: Web demo del modulo "Entity Extraction" di Dandelion API. Mostra le entità riconosciute a partire dal testo in input

Infine, interessante anche il numero di tipologie diverse di entità riconosciute: ben 7. Si distinguono Persone, Opere, Organizzazioni, Luoghi, Eventi e Concetti.

Text Similarity

Si tratta di un componente attualmente in versione beta che prende in input due testi, li confronta ed esprime un valore in percentuale che indica le somiglianze sintattiche ed

uno ne che indica quelle semantiche. Per il tipo di analisi che PARLEN conduce, un modulo con queste funzionalità non ritorna particolarmente utile. Tuttavia l'ho testato: funziona bene, anche se intuitivamente l'algoritmo utilizzato non dev'essere di particolare complessità.

Text Categorization

Un altro componente dalle caratteristiche estremamente interessanti: permette di classificare un testo come facente parte di una o più categorie, liberamente definibili. In pratica permette di capire quali sono i topic principali di cui tratta un testo, ed è in grado di farlo senza alcun tipo di training ma semplicemente utilizzando un algoritmo che sfrutta le informazioni semantiche che il modulo riesce ad estrarre. Purtroppo la demo online di questo componente permette il testing solo con la lingua inglese. Ad ogni modo funziona perfettamente: nonostante numerosi tentativi, solo in una occasione i risultati ottenuti non combaciavano perfettamente col reale contenuto del testo. La demo inoltre visualizza i risultati in un grafico intuitivo attraverso cui si evince come l'appartenenza del testo ad una categoria viene espressa con il solito valore all'interno di un range.

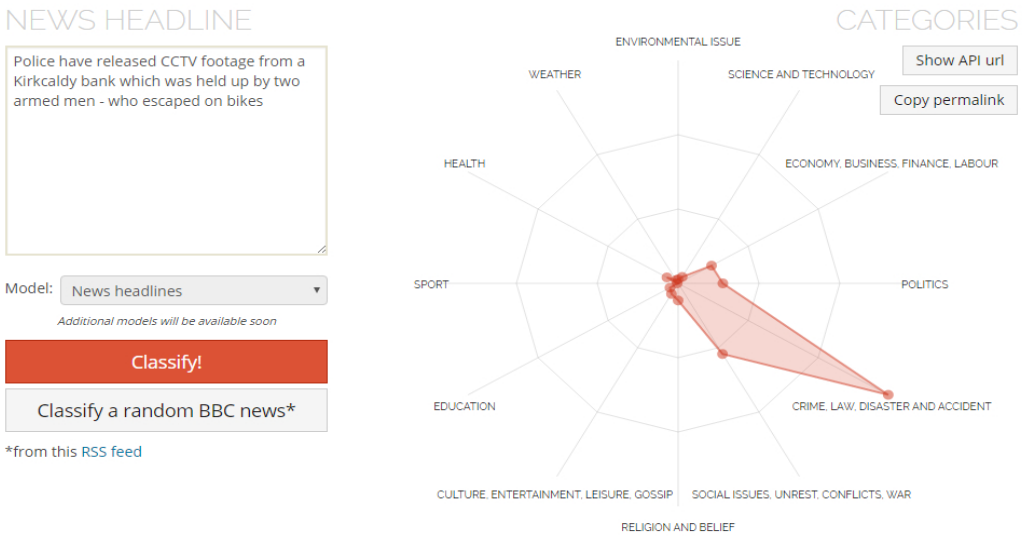


Figura 1.5: Web demo del modulo "Text Categorization" di Dandelion API. Mostra i topic trattati nel testo in input.

Sentiment Analysis

Questo modulo prova a stabilire se l'opinione espressa in un piccolo testo è positiva, negativa o neutra. Purtroppo esso analizza solo testi di piccole dimensioni. In genere, gli algoritmi che vengono utilizzati per questo tipo di analisi assegnano ad ogni lemma del testo un valore numerico ne che esprime la positività/neutralità/negatività. Non è escluso che anche informazioni semantiche vengano in aiuto durante la computazione. Anche questo modulo funziona discretamente bene, fatta eccezione per qualche caso, uno dei quali è visibile nell'immagine che segue.

Figura 1.6: Web demo del modulo "Sentiment Analysis" di Dandelion API. Mostra l'eventuale positività rilevata nella frase analizzata. In rosso un caso di malfunzionamento.

Dandelion API si dimostra quindi uno dei migliori servizi per analisi semantica in circolazione; è completo ed ogni suo componente funziona estremamente bene. I parametri che si possono specificare e la tendenza ad esprimere i risultati anche utilizzando mezze misure lo rendono unico nel suo genere e permettono a chi lo utilizza di gestire con precisione ogni modulo. Chi si approccia ad utilizzarlo non si trova davanti alcun vincolo, quanto meno tecnicamente parlando. Come si intuisce dal nome infatti, il servizio funziona esclusivamente con interfaccia RESTful, attraverso delle API. Gli sviluppatori hanno inoltre pensato bene di fornire una vasta scelta di librerie attraverso cui contattare le api col proprio linguaggio preferito. Tra gli altri linguaggi, supportati anche *Nodejs*, *Python* e *Ruby*. Notevole anche la scelta di poter integrare il servizio con *Google Drive*, *Wordpress* o addirittura *Stanbol*.

1.3.3 Polyglot

Polyglot è un software Open Source sviluppato in Python. Nella documentazione ufficiale[17] viene presentato come una *pipeline*. Questo perché i moduli che lo compongono, lavorano appunto a cascata e l'output di uno diventa l'input del successivo. Questa tipologia d'approccio è estremamente diffusa in software che analizzano il linguaggio naturale in quanto quasi sempre gli algoritmi si basano sulla suddivisione del testo in parti più piccole e sull'aggiunta di nuove informazioni che torneranno utili per l'esecuzione del modulo seguente.

Polyglot supporta in assoluto il maggior numero di lingue, 196 per l'esattezza. Purtroppo però non include una funzione fondamentale: il linking delle entità riconosciute a risorse esterne come DBpedia. Una grave mancanza, soprattutto se si pensa che il database attraverso cui riconosce le entità è proprio quello di *Wikipedia*. Ad ogni modo svolge un buon lavoro, sopratutto con quei moduli che si occupano delle fasi preliminari dell'analisi. Può risultare quindi molto utile per coloro i quali intendono svolgere analisi morfo-sintattiche o per coloro che non necessitano di collegamenti a risorse esterne ma hanno bisogno di individuare precise entità o tipologie di entità all'interno di un set di dati. Anche le performance non sono niente male: in circa 15/20 secondi riesce ad operare un'analisi completa su un testo di 5000 caratteri, in italiano.

Vediamo le caratteristiche dei moduli che lo compongono.

Language Detection

Si occupa semplicemente di individuare la lingua del testo in input.

Tokenization

Attraverso questo modulo, Polyglot suddivide il testo in input distinguendo le singole parole e le singole frasi. Può tornare utile in qualche situazione ed è soprattutto indispensabile per la corretta esecuzione degli altri moduli facenti parte della pipeline.

POS (part of speech) Tagging

Questo modulo assegna ad ogni parola una categoria che ne identifica il ruolo sintattico. Polyglot distingue 17 categorie tra cui verbi, avverbi, nomi, congiunzioni, pronomi etc.

Named Entity Extraction

Questo è il modulo più importante: quello che si occupa dell'estrazione delle entità. Esso supporta 40 tra le lingue più diffuse; è in grado di distinguere 3 diversi tipi di entità: persone, luoghi ed organizzazioni. Come già detto, fa riferimento a Wikipedia e purtroppo nè questo nè altri moduli della pipeline si occupano del linking della risorsa. I risultati sono quindi vuoti di contenuto semantico e le ambiguità sono quasi una costante. Tutto sommato, riconosce un elevato numero di entità, cosa che comunque non aiuta la riduzione dei falsi negativi.

Morphological Analysis

Questo è uno dei moduli di Polyglot che funziona meglio. Il suo scopo, inoltre è molto interessante: riconosce la composizione morfologica delle parole in input, restituendo in output eventuali altri lemmi di senso compiuto che vanno a comporre la parola di partenza. Nessuno degli altri tool online include un modulo con questa funzionalità. Se utilizzato bene, può essere utile per il riconoscimento ad esempio di parole di origine

latina. Anche in questo caso però, siamo davanti ad un'analisi che di semantico non ha nulla.

```
words = ["preprocessing", "processor", "invaluable", "thankful", "crossed"]
for w in words:
  w = Word(w, language="en")
  print("{:<20}{}".format(w, w.morphemes))
```

```
preprocessing       ['pre', 'process', 'ing']
processor           ['process', 'or']
invaluable          ['in', 'valuable']
thankful            ['thank', 'ful']
crossed             ['cross', 'ed']
```

Figura 1.7: Esempio di utilizzo e relativo output del modulo "Morphological Analysis" di Polyglot.

Transliteration

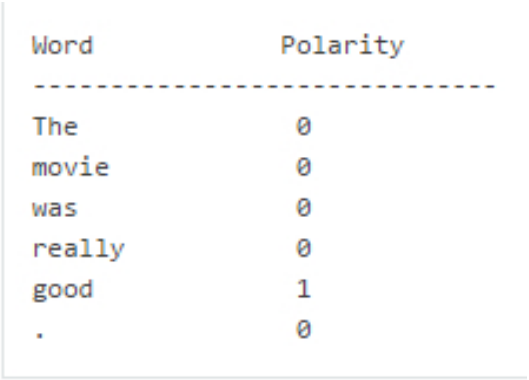
Tecnicamente con traslitterazione si indica la trasposizione di grafemi da un sistema di scrittura ad un altro. E questo è quello che fa questo modulo. Supporta 69 lingue e può lavorare su un testo lungo arbitrariamente. Conoscendo solo il sistema di scrittura occidentale, non sono in grado di valutare la qualità di questo componente in prima persona.

Sentiment

Questo è il classico modulo che cerca un sentimento positivo o negativo in una frase. L'algoritmo assegna un punteggio ad ogni parola del testo in input: -1 nel caso in cui la parola venga considerata negativa (brutto, cattivo, non, no, etc), 0 in caso di parola neutra, 1 in caso di parola positiva (bene, bello, bravo, etc). In seguito viene calcolata una media tra -1 e 1 che indica il sentimento individuato. Purtroppo, non operando dal punto di vista semantico, non possiamo avere la certezza di individuare correttamente il sentimento di una frase. Tuttavia, anche l'approccio dei soli punteggi risulta vincente in

molti casi, soprattutto con frasi di una lunghezza compresa tra i 100 ed i 1000 caratteri.

Polyglot è tutto sommato un buon software, ben documentato e facile da utilizzare.



Word	Polarity
The	0
movie	0
was	0
really	0
good	1
.	0

Figura 1.8: Esempio di output del modulo "Sentiment" di Polyglot.

Può essere usato come libreria in Python ma anche da riga di comando attraverso una CLI.

Il fatto che sia Open Source gli garantisce di eccellere soprattutto in quello per cui è stato pensato: supporto massimo al multilingua. Non è uno strumento adatto per chi pensa di utilizzare strumenti come LOD e DBpedia, come nel caso di PARLEN Resta comunque un buon applicativo per chi ha esigenze d'analisi più che altro legate all'aspetto sintattico, morfologico e grammaticale o per chi necessita di lavorare con entità mirate.

1.3.4 The Opener Project

Si tratta di un progetto Open Source molto grande, lo si intuisce già dalla pagina Github[18] omonima: ben 50 repository attivi che vanno a comporre quest'insieme di software specializzati nell'analisi del linguaggio naturale. Nella homepage[19] del progetto, una frase estremamente riassuntiva ci permette di capire le funzionalità principali del pacchetto software: "Opinioni, Entità e Sentimenti in sei lingue ed attraverso vari Domini". Tutti termini già utilizzati in questo capitolo. Effettivamente, sulla carta, The Opener Project non sembra avere niente in meno rispetto a Dandelion API, anzi: Opener è intanto Open Source; inoltre, se le API di Dandelion ci permettevano di fare sostanzialmente 4 tipi di analisi, con Opener possiamo farne molte di più. L'architettura

di quest'ultimo è estremamente modulare; ogni modulo è altamente specializzato in un compito preciso e, come nel caso di Polyglot, l'utilizzo è orientato alla costruzione di una pipeline.

Sostanzialmente The Opener Project sembra prendere il meglio dai tool descritti fin'ora:

come Polyglot ha una pratica architettura, un sottoinsieme di moduli dedicati all'analisi sintattico/grammaticale, una natura OpenSource;

come Dandelion, ci permette di estrarre entità collegate a risorse esterne, effettuare analisi su opinioni e sentimenti.

Oltre alle funzionalità principali di Opener, che andremo ad approfondire in seguito, esistono alcuni moduli che offrono dei servizi che nessuno dei tool fin'ora descritti offriva. Vale la pena citarli.

Coreference

Volendo semplificare, questo modulo prende in input un XML che descrive entità e cerca altre entità correlate in qualche modo a quelle di partenza. Di un famoso personaggio possiamo ad esempio conoscere le relazioni famigliari o le cose per cui è noto.

Polarity tagger

Uno dei moduli dedicati all'analisi del sentimento. Offre la possibilità di guidare l'algoritmo indicando quali sono le parole che più frequentemente possono esprimere positività o negatività; esiste anche la possibilità di specificare un dominio all'interno del quale operare l'analisi, per avere risultati più veritieri.

NER/NED

Questi sono i due moduli principali per si occupano di riconoscere e collegare entità a DBpedia. Il secondo in particolare, Named entity Disambiguation, interroga una istanza demo di DBpedia Spotlight[20]. Si tratta di un altro progetto Open Source che annota automaticamente in un testo, citazioni a risorse di DBpedia.

Presumibilmente, i moduli di Opener utilizzano poi ulteriori algoritmi sui risultati di DBpedia Spotlight per disambiguare le entità.

Tuttavia, la disambiguazione di Dandelion funziona leggermente meglio: Opener, almeno con la lingua italiana, fà talvolta fatica e riconosce, per tornare all'esempio fatto in precedenza, il "calcio" sempre e comunque come sport, anche quando si parla di elementi chimici. Tuttavia, soprattutto con le persone funziona bene: Mr. Obama o Barak Obama vengono entrambi riconosciuti e correttamente ricollegati con l'attuale Presidente degli Stati Uniti.

The Opener Project raggiunge il miglior compromesso tra i potenziali tool che possano offrire servizi a PARLEN Ho deciso quindi di utilizzarlo. Inoltre presenta una caratteristica tecnica molto utile: tutti i suoi componenti sono eseguibili da riga di comando oppure come server web. Ciò ha permesso maggior libertà durante la progettazione architetturale di PARLEN

1.4 La lingua italiana e l'analisi del linguaggio naturale

Come abbiamo visto, i tool che abbiamo descritto supportano lingue diverse e non sempre l'italiano è tra esse. Questo accade perché gli algoritmi utilizzati per l'analisi del linguaggio naturale e per il riconoscimento delle entità, fanno spesso riferimento a regole sintattico-grammaticali o dataset RDF strettamente correlati con la lingua del testo in analisi. Tutto sommato, un dato molto interessante riguarda l'Italia: sono molte

le attività e le organizzazioni che dedicano il proprio lavoro allo sviluppo di strumenti per l'analisi semantica di contenuti. Abbiamo analizzato le grandi potenzialità di Dandelion

API ma SpazioDati, la S.r.l che lo ha sviluppato, non ha solo questo come prodotto. L'azienda offre anche altri tipi di servizi enterprise[21]; per citarne alcuni:

- realizzazione di mashup tra i dati interni e sorgenti esterne per abilitare nuove dimensioni di analisi;

- miglioramenti dei processi di gestione dei dati di oganizzazioni;
- implementazione di strategie e portali Open Data;
- progettazione, integrazione ed applicazione ontologie.

Abbiamo poi citato Celi, altra azienda che crea prodotti per "estrarre conoscenza e creare valore dai dati linguistici", come si legge sul loro sito. I loro prodotti principali sono 5[22], tra di essi c'è anche un motore di ricerca semantico ed un software per tecnologia vocale avanzata. Propongono inoltre un' applicazione chiamata *Blogmeter* che riconosce ed analizza ciò che viene detto online su un tema, un'azienda, un marchio o un personaggio pubblico. Software come questi al giorno d'oggi sono sempre più richiesti e sono sempre di più le aziende che prendono importanti scelte aziendali anche seguendo opinioni e feedback online.

Infine in Italia abbiamo anche un movimento che segue e valuta i risultati di tool per il processo del linguaggio naturale in italiano. Parliamo di Evalita[23], una campagna di valutazione che organizza eventi che come scopo principale hanno quello di promuovere lo sviluppo di tecnologie per la lingua e per il parlato, offrendo un framework condiviso che utilizza diversi sistemi ed approcci tecnici. Questa organizzazione è attiva dal 2007 e continua ogni anno a promuovere lo sviluppo di questa tipologia di tool, con attenzione massima al supporto della lingua italiana.

Capitolo 2

PARLEN: Una pipeline per scraping, riconoscimento di entità e produzione di ontologie

Vedremo ora nel dettaglio cosa fa e come funziona PARLEN Lo faremo analizzando i ruoli, gli input, gli output e anche gli eventuali limiti di ogni modulo che lo compongono.

PARLEN è costituito sostanzialmente di 4 moduli, più un quinto che funge da CLI (command line interface). Attraverso quest'ultimo è possibile eseguire da riga di comando tutti i metodi che l'applicazione espone.

In fase preliminare avevo pensato di implementare PARLEN come un'applicazione web in cloud, utilizzabile con il browser da qualsiasi dispositivo connesso ad internet. Poi, mi sono reso conto che in alcuni casi, la fase principale, quella d'analisi con i servizi di Opener, poteva richiedere molto tempo, a seconda della quantità di dati forniti in input. Sarebbe stato scomodo usare l'applicazione dal web e la gestione ed il riutilizzo degli output intermedi prodotti dai vari moduli risultavano particolarmente complessi. Ho quindi optato per l'implementazione di un CLI che permettesse all'utente di eseguire comunque ogni operazione che avrebbe potuto eseguire sul web ma da riga di comando, in modo da poter gestire con semplicità tempistiche e outputs. Ovviamente si potrebbe comunque pensare una implementazione in cloud, lo vedremo in seguito.

I 4 moduli principali, elencati in ordine di come vengono eseguiti sono: *scraper*, *analizer*, *visualizer*, *rdfer*. In realtà, gli ultimi due moduli possono lavorare in maniera indipendente sull'ouput dell'analizer. La seguente figura illustra meglio l'architettura dell'applicazione.

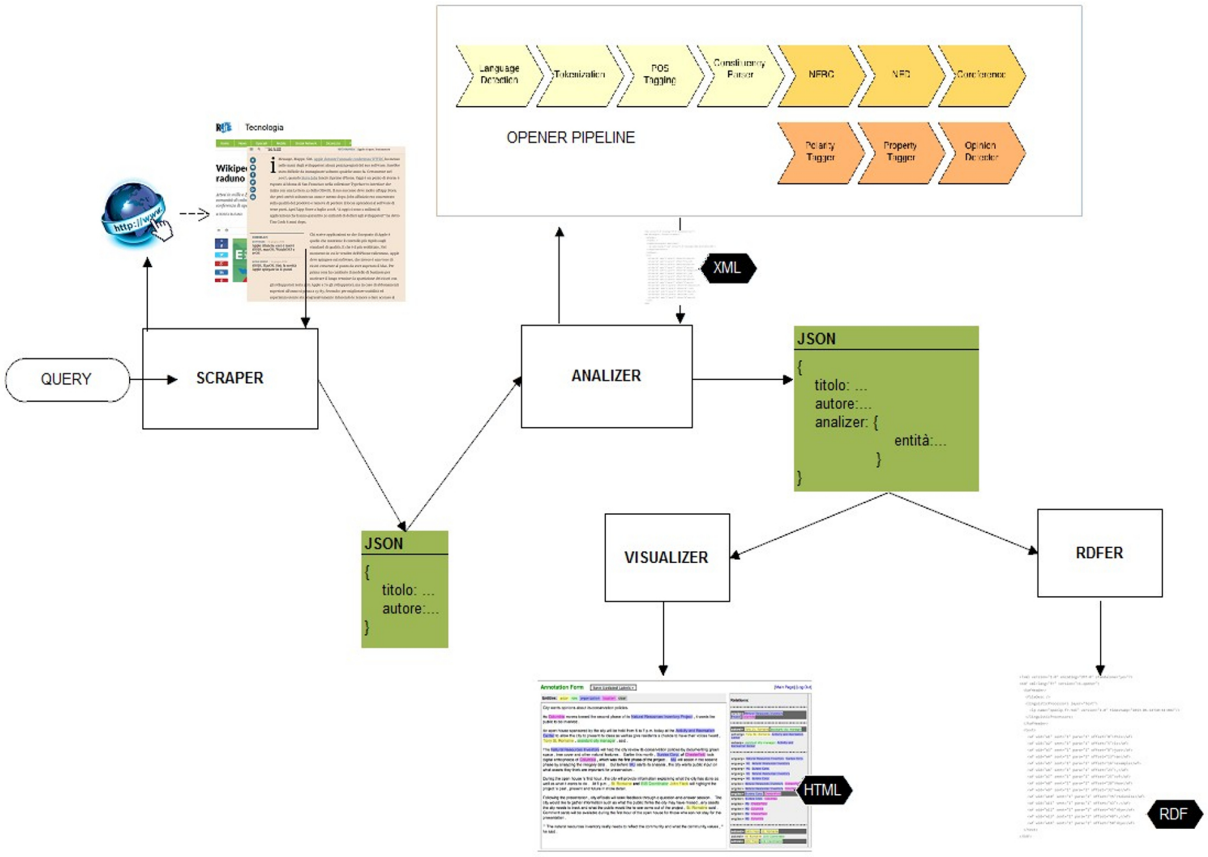


Figura 2.1: Architettura di PARLEN, input e output.

Come è evidente, tutto inizia da una query. L'utente inserisce una o più parole chiave da ricercare all'interno dei giornali supportati. Lo scraper viene subito chiamato in causa e recupera le pagine contenenti gli articoli, estrae le informazioni essenziali (titolo, testo dell'articolo, autore etc) e le salva su disco in formato JSON.

Il JSON generato dallo scraper viene dato in input all'analizer che, servendosi dei servizi offerti da Opener, opera un'analisi sul contenuto di ogni articolo, provando ad estrar-

re le entità contenute in essi. Conclusa l'analisi, l'analizer salva i risultati estendendo il JSON precedente con le nuove informazioni appena ottenute.

Questo secondo JSON è un input valido tanto per il visualizer quanto per l'rdfer: se pure questi due moduli hanno due ruoli completamente diversi, entrambi trovano le informazioni di cui hanno necessitano nello stesso file. In particolare, il visualizer produce un documento HTML per ogni articolo trovato, evidenziando però anche le entità riconosciute e permettendo all'utente una lettura interattiva. L'rdfer produce una ontologia in rdf sulla quale sarà poi possibile in seguito effettuare query in SPARQL ed operare qualsiasi altro tipo di analisi più sofisticata sfruttando le potenzialità dei Linked Open Data.

Un'architettura del genere permette una certa flessibilità sia nell'utilizzo dell'applicazione sia per quanto concerne eventuali espansioni. L'utilizzo è semplificato dal fatto che si può scegliere di eseguire i vari moduli separatamente ed in momenti diversi; è quindi possibile suddividere l'analisi, ripeterla solo in alcune fasi ed ottenere con facilità output intermedi. Inoltre è possibile scegliere quali sorgenti di dati (giornali) utilizzare ed anche quale/i tool per l'analisi.

PARLEN è attualmente in grado di analizzare articoli provenienti da Repubblica e Sole 24 Ore ed utilizza Opener come tool per l'estrazione di entità: vedremo però come è sarà possibile espandere la compatibilità a nuovi giornali e nuovi tool per l'analisi del linguaggio naturale.

2.1 Scraper: repubblica.js e sole24ore.js

Si tratta di un modulo (per essere precisi un insieme di moduli, uno per ogni giornale) che hanno un compito preciso: recuperare dall'archivio del giornale relativo tutti gli articoli che parlano di un determinato argomento, in un range di date. In particolare, i parametri che prende in input sono i seguenti:

- **query**: una stringa contenente la query di ricerca;
- **data-inizio**: una data di pubblicazione degli articoli dalla quale partire: tutti gli articoli pubblicati precedentemente saranno ignorati;

- **data-fine**: una data che insieme a quella d'inizio definisce un range: gli articoli pubblicati in giorno fuori dal range saranno ignorati;

- **connettore logico**: se la query contiene più di una parola, è possibile specificare il tipo di connettore logico da utilizzare nel corso della ricerca; se scelto l'AND, saranno selezionati solo gli articoli in cui compaiono esattamente tutte le parole contenute nella query. Con l'OR, saranno scelti tutti gli articoli che ne conterranno almeno una. Ovviamente, se la query contiene una sola parola (es "inflazione"), questo parametro è inutile.

A seconda dell'archivio interrogato, sarà inoltre necessario autenticarsi. Repubblica non richiede questa esigenza, in quanto l'archivio è aperto e gratuito. Il Sole 24 Ore, invece, ha un archivio a pagamento ed il modulo che lo gestisce si deve occupare anche dell'autenticazione senza la quale non sarà possibile avere accesso ad alcun dato. L'Università di Bologna ha accesso ai dati dell'archivio storico del Sole 24 Ore ed interrogando i server del giornale con un IP della rete di Unibo, risulteremo autenticati. Questo però, costringe l'utilizzo di PARLEN o direttamente da una macchina dell'Università di Bologna o con un proxy attivo, cosa che sono stato costretto a fare in fase di sviluppo.

Il modulo procede col download delle pagine html (o dei JSON a seconda dell'architettura del backend del giornale interrogato). Quest'operazione è eseguita con parsimonia: non è consigliato ad esempio scaricare tutti gli articoli che parlano d'inflazione dalla nascita del giornale in questione; il rischio è quello di ritrovarsi con l'IP bannato. Basti pensare che, ad esempio, Repubblica risponde con i "soli" primi 2500 risultati in caso di query che ritornano un numero di articoli maggiore ed invita gli utenti a spezzettare la ricerca, magari utilizzando il filtro delle date. Inoltre, onde evitare problemi di ban, per il download dei risultati è stata utilizzata la tecnica del throttling: se c'è la necessità di scaricare 200 articoli, piuttosto che fare 200 richieste contemporaneamente, questo modulo suddivide le richieste in gruppi più piccoli. In particolare, ho preferito tenere una connessione attiva per volta (e quindi richiedere un articolo alla volta) per evitare ogni problema. Attenzione però: è possibile richiedere a PARLEN di procedere con un'analisi su più giornali contemporaneamente: in quel caso, sarà tenuta aperta una connessione

alla volta per ogni giornale, in modo da parallelizzare l'operazione ed ottenere i risultati in tempi ragionevolmente brevi.

In seguito all'ottenimento degli articoli, viene eseguita una routine altamente personalizzata a seconda del giornale che recupera dalla pagina ottenuta solo le informazioni essenziali. Quest'operazione è tra le più critiche ed in alcuni casi può generare errori.

Per capirne il motivo, facciamo un po' di chiarezza: un sito web dinamico, come quello di un giornale, può essere costruito in tanti modi diversi. Fino a qualche anno fa, la maggior parte dei siti web facevano rendering lato server. Questo vuol dire che il server creava una pagina html a partire da un template (scheletro) e dai dati (contenuto). Il client riceveva quindi una pagina HTML, contenente già tutti i dati. Approcci più moderni preferiscono il rendering lato client: il client non richiede l'intera pagina contenente l'articolo ma soltanto i dati (tipicamente in formato JSON) da renderizzare all'interno di un template, servito precedentemente. Questo approccio limita l'utilizzo di banda da parte sia del server che del client e distribuisce il carico dell'operazione di rendering ai vari client, appesantendo meno la CPU del server.

Per lo scraper di PARLEN è fondamentale sapere che tipo di rendering operano i server dei giornali contattati: nel caso di rendering lato client (Sole 24 Ore), è molto più facile recuperare i dati necessari. La risposta in JSON è standard e tra le varie proprietà del file troveremo sempre quello che stiamo cercando, dal titolo dell'articolo, al body, all'autore etc. Quando invece si fa rendering lato server, come nel caso di Repubblica, non sarà più sufficiente andare a leggere un valore all'interno di un JSON ma bisognerà analizzare con un'apposita libreria l'HTML contenente l'articolo, distinguendo attraverso i selettori css le varie parti della pagina e recuperando solo quelle che interessano.

Quest'operazione sarebbe relativamente semplice nel caso in cui il server contattato mantenesse un formato standard all'interno delle proprie pagine. Tuttavia, nel caso di Repubblica non è cosi: esistono decine di formati diversi con cui il sito presenta i vari articoli e non mancano errori strutturali all'interno delle pagine HTML. Inoltre, in alcuni casi, i metadati non sono contenuti all'interno di alcun tag speciale e questa cosa rende quasi impossibile la distinzione tra il testo dell'articolo ed il suo titolo, ad esempio. Gli

articoli pubblicati nella versione cartacea, poi, talvolta sono incompleti e comunque nel 100% dei casi non hanno alcuna struttura vera e propria. Per differenziare titolo da testo, autore e data è necessario applicare non pochi stratagemmi. Tuttavia, come vedremo più in dettaglio, repubblica.js raggiunge un buon compromesso e riesce a supportare quante più tipologie d'articolo ed errori diversi, fornendo in output un risultato accettabile.

Invece, come già accennato, con il Sole 24 Ore ci sono molti problemi in meno, in quanto le informazioni di cui necessitiamo sono già in un formato (JSON) facile da leggere.

Ottenuti gli articoli e differenziate le varie parti, vengono eseguite ulteriori operazioni di filtraggio. Alcuni articoli, ad esempio, fanno riferimento a pagine che non esistono e saranno esclusi. Saranno esclusi inoltre gli articoli con una lunghezza inferiore ai 300 caratteri (in quanto non possono essere considerati veri e propri articoli) e saranno corretti quegli articoli che presentano caratteri speciali (privi di significato) all'inizio o alla fine del body o del titolo.

Dopo le ultime operazioni di filtraggio, il JSON è pronto e viene salvato su disco. Si tratta di un file che per ogni articolo distingue:

- url;
- fonte;
- autore;
- titolo;
- body;
- data di pubblicazione;
- footer;
- città.

Non sempre tutti i campi saranno completi: in alcuni articoli non viene specificato l'autore, in altri addirittura il titolo. In alcuni casi, invece, le informazioni sono specificate

ma con un formato completamente diverso da quelli "standard" e lo scraper non è in grado di riconoscerle.

Figura 2.2: Esempio di articolo su Repubblica.it

Nei paragrafi successivi vedremo come i dati di quest'articolo saranno estratti, rielaborati ed estesi.

2.2 Analizer: opener.js

Questo modulo legge il JSON generato dallo scraper ed utilizza un tool per l'analisi del linguaggio naturale per riconoscere le entità contenute negli articoli. Attualmente ad essere supportato è solo Opener tool già descritto in precedenza.

Il body di ogni articolo viene processato dalla pipeline di Opener. Come avevamo visto, sono veramente tante le possibilità ed i servizi che Opener offre e lo sviluppatore che si approccia con esso può costruire la pipeline come meglio preferisce. Nel mio caso, ho utilizzato nell'ordine i seguenti servizi di Opener:

- language-identifier (vedremo dopo il perchè);
- tokenizer;
- pos-tagger;
- ner;
- ned.

I primi tre moduli sono alla base di ogni analisi eseguita con questo tool e costruiscono un contesto attraverso cui i moduli successivi possono lavorare. Gli ultimi due si occupano poi di riconoscere, estrarre e collegare ad una risorsa DBpedia le entità riconosciute. Opener riconosce Persone, Organizzazioni, Luoghi ed Posizioni Geopolitiche.

Inoltre, il compito eseguito dal tokenizer, ci torna utile per i moduli successivi. Esso infatti individua l'offset di ogni parola a partire dall'inizio del testo e distingue tutte le frasi contenute in ogni articolo. Attraverso quest'informazione saremo in grado di costruire un'ontologia più completa con l'rdfer ed avremo i dati necessari da fornire al visualizer che senza l'informazione dell'offset non potrebbe in alcun modo funzionare.

Anche l'analizer può generare errori: in particolare, il problema non sta nel modulo di PARLEN (opener.js) ma nella pipeline di Opener. In alcuni rari casi che non sempre sono riuscito a distinguere, il modulo NER crasha. Fortunatamente questa è una eventualità più rara che altro. Ad ogni modo, ho quanto meno gestito l'errore filtrando eventuali articoli problematici e garantendo il proseguo dell'esecuzione.

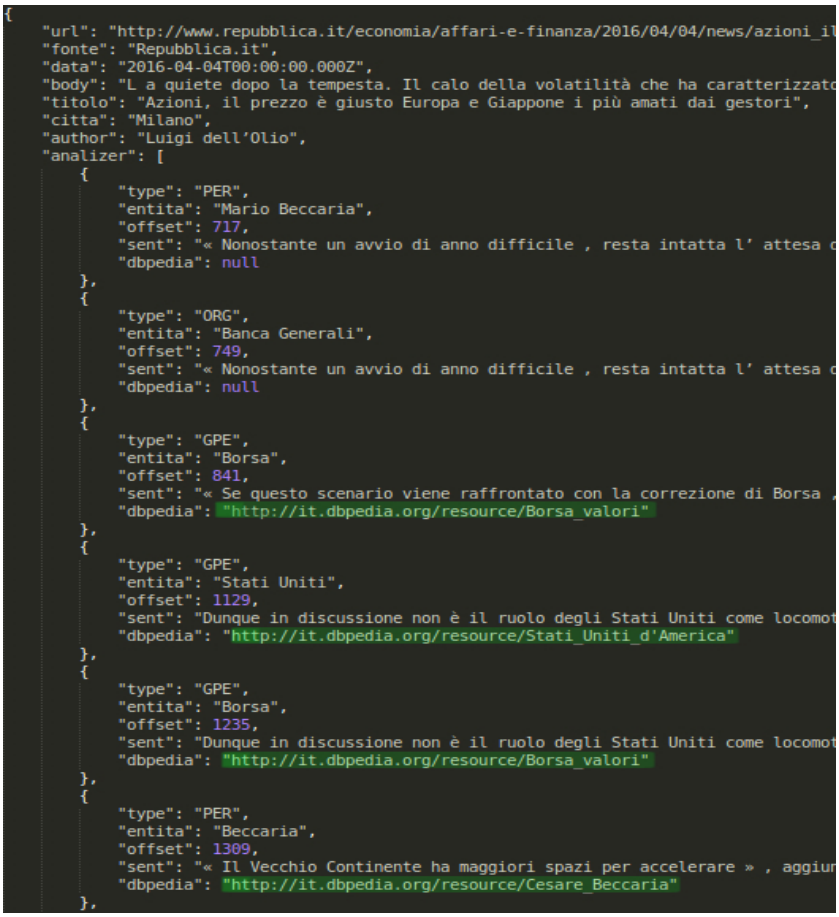
Conclusa l'analisi di ogni articolo in input e filtrati eventuali casi d'errore, l'analizer procede con l'estensione e la successiva scrittura su disco del JSON. Si tratta di un file identico al precedente meno che per il fatto che ogni articolo contiene un campo in più: un vettore di entità. Ogni elemento di questo vettore contiene:

- il nome attraverso cui l'entità è citata;
- il link a DBpedia (quando disponibile);
- il tipo di entità;
- offset;
- frase in cui l'entità è citata;
- log.

Nella proprietà log sono indicati dati sulla durata dell'esecuzione e sullo stack degli errori generati da Opener, nel caso in cui ne esistano.

Gli errori che sono gestiti sono quelli che causano il crash dei servizi di Opener. Eventuali falsi positivi o falsi negativi non possono essere individuati se non reimplementando quasi del tutto il sistema di estrazione e disambiguazione delle entità. PARLEN utilizza Opener ed ha fiducia nei suoi risultati. Analizzeremo meglio quali situazioni mettono più in difficoltà i servizi di Opener.

Segue un sample dell'output prodotto dal modulo sull'articolo della figura 2.2.



```
{
    "url": "http://www.repubblica.it/economia/affari-e-finanza/2016/04/04/news/azioni_il
    "fonte": "Repubblica.it",
    "data": "2016-04-04T00:00:00.000Z",
    "body": "L a quiete dopo la tempesta. Il calo della volatilità che ha caratterizzato
    "titolo": "Azioni, il prezzo è giusto Europa e Giappone i più amati dai gestori",
    "citta": "Milano",
    "author": "Luigi dell'Olio",
    "analizer": [
        {
            "type": "PER",
            "entita": "Mario Beccaria",
            "offset": 717,
            "sent": "« Nonostante un avvio di anno difficile , resta intatta l' attesa d
            "dbpedia": null
        },
        {
            "type": "ORG",
            "entita": "Banca Generali",
            "offset": 749,
            "sent": "« Nonostante un avvio di anno difficile , resta intatta l' attesa d
            "dbpedia": null
        },
        {
            "type": "GPE",
            "entita": "Borsa",
            "offset": 841,
            "sent": "« Se questo scenario viene raffrontato con la correzione di Borsa ,
            "dbpedia": "http://it.dbpedia.org/resource/Borsa_valori"
        },
        {
            "type": "GPE",
            "entita": "Stati Uniti",
            "offset": 1129,
            "sent": "Dunque in discussione non è il ruolo degli Stati Uniti come locomot
            "dbpedia": "http://it.dbpedia.org/resource/Stati_Uniti_d'America"
        },
        {
            "type": "GPE",
            "entita": "Borsa",
            "offset": 1235,
            "sent": "Dunque in discussione non è il ruolo degli Stati Uniti come locomot
            "dbpedia": "http://it.dbpedia.org/resource/Borsa_valori"
        },
        {
            "type": "PER",
            "entita": "Beccaria",
            "offset": 1309,
            "sent": "« Il Vecchio Continente ha maggiori spazi per accelerare » , aggiun
            "dbpedia": "http://it.dbpedia.org/resource/Cesare_Beccaria"
        },
```


Figura 2.3: Sample di output dell'analizer. Contiene anche i dati dello scraper. In verde, i link alle risorse collegate con DBpedia.

2.3 Visualizer

Questo modulo prende in input il JSON generato dall'analizer e produce in output un file HTML per ogni articolo precedentemente analizzato.

Il file HTML è una pagina che contiene tutti i dati relativi all'articolo in questione: titolo, autore, giornale, url, body, data di pubblicazione. In più, però, tutte le entità riconosciute dall'analizer sono evidenziate, di colore diverso a seconda della tipologia.

Inoltre, quelle entità di cui è stato riconosciuto un riferimento su DBpedia, sono cliccabili e rimandano alla risorsa in questione.

Il modulo quindi non si occupa di operare ulteriori analisi ma è stato creato per visualizzare in un formato più consono ad un essere umano i risultati ottenuti dall'analisi di Opener. Ad ogni modo, con i file HTML ottenuti non siamo in grado di incrociare i risultati con altre ricerche ne tanto meno siamo in grado di sfruttare le potenzialità dei Linked Open Data. In sostanza, questo modulo sfrutta solo una parte delle potenzialità offerte dai risultati dell'analisi operata.

Fatta eccezione per qualche raro caso, questo modulo svolge il suo lavoro velocemente e senza generare mai errori. Raramente è capitato che qualche articolo presentasse dei problemi: gli offset risultavano essere sfalsati di uno o due indici. Il problema parte dallo scraper ed è legato al tokenizer di Opener che in rari casi, quando riceve input che presentano caratteri speciali o andate a capo consecutive, sbaglia a calcolare gli offset di appunto 1, 2 indici. Fidandoci di Opener ed utilizzando i suoi dati, per forza si va incontro ad un errore. Comunque sia (vedremo in seguito i dettagli), questo problema si verifica talmente poche volte che possiamo considerarlo trascurabile.

Segue l'output del modulo, relativo sempre all'articolo della figura 2.2.

Figura 2.4: Sample di output del visualizer. Le entità sottolineate sono cliccabili e rimandano alla risorsa DBpedia.

2.4 Rdfer

Questo modulo è particolarmente importante: senza la sua esistenza, l'analisi eseguita in precedenza servirebbe a ben poco e sarebbe difficile avere un quadro chiaro della "situazione semantica" dei documenti analizzati.

Prendendo in input il file generato dall'analizer, l'rdfer genera una ontologia in formato RDF che contiene articoli ed entità (con relativi offset, tipologia e link DBpedia). Potrebbero sembrare semplicemente gli stessi dati in un formato diverso, ma non è cosi. L'RDF è un formato che dà significato semantico alle entità. I dati sono espressi sotto forma di triple composte da soggetto, predicato e oggetto.

In particolare, per ogni articolo abbiamo:

- una tripla che lo identifica appunto come un articolo;
- una tripla per ogni frase di ogni articolo in cui viene citata una entità;
- due o tre triple per ogni entità riconosciuta (servono a definire la tipologia dell'entità e a collegarle a DPpedia quando è possibile);;
- una tripla che collega le entità alle frasi in cui sono citate;
- un insieme di triple che descrivono l'offset relativo alla frase in cui l'entità è citata.

Con un'ontologia così costruita possono essere eseguite una serie di analisi supplementari. Inoltre, ontologie provenienti da analisi differenti possono essere accorpate sulla stessa base di dati estendendo le possibilità. Attraverso il linguaggio di interrogazione SPARQL, è possibile eseguire delle query ed ottenere informazioni di ogni tipo come:

- conoscere la quantità di volte che una entità viene citata;
- conoscere il numero di articoli che citano una entità;
- conoscere il tipo di entità più citate tra articoli che parlano di un argomento;
- utilizzare gli operatori logici ed ottenere, ad esempio, il numero di articoli che citano o Berlusconi o Renzi;
- ... e qualsiasi altra informazione che si può ottenere interrogando una base di dati composta come descritto.

Infine, per le entità di cui abbiamo anche un riferimento su DBpedia, è possibile utilizzare le opportunità offerte dai Linked Open Data. Se il sistema è stato in grado di riconoscere una persona famosa, attraverso le interrogazioni semantiche possiamo riconoscere i motivi per cui quella persona è famosa, altre persone o cose collegate a quella di partenza e qualsiasi altra informazione di cui nelle ontologie di LOD è fornita una rappresentazione semantica.

Queste ultime possibilità sarebbero parzialmente offerte da un modulo di Opener di cui abbiamo già parlato, il Coreference. Tuttavia, ho ritenuto più pratico lasciare libertà all'utilizzatore di PARLEN di lavorare con l'ontologia prodotta, evitando di individuare collegamenti superflui (es. l'entità Roma collegata all'entità "capitale") ed evitando anche di tralasciare collegamenti che invece possono potenzialmente essere utili all'utente.

Vediamo nella figura che segue un frammento dell'ultimo output prodotto dalla pipeline, sempre relativo all'articolo della figura 2.2.

```
ent:Beccaria a foaf:Person;
    owl:sameAs <http://it.dbpedia.org/resource/Cesare_Beccaria>.
<http://articoli.it/sent/bc7f646c6f61068bcbc2eb6499ecc666/3/span-70-78> a nif:String;
    nif:beginIndex "70"^^<http://www.w3.org/2001/XMLSchema#nonNegativeInteger>;
    nif:endIndex "8"^^<http://www.w3.org/2001/XMLSchema#nonNegativeInteger>;
    nif:referenceContext <http://articoli.it/sent/bc7f646c6f61068bcbc2eb6499ecc666/3>;
    itsrdf:taldentRef ent:Beccaria.
<http://articoli.it/sent/bc7f646c6f61068bcbc2eb6499ecc666/3/span-174-177> a nif:String;
    nif:beginIndex "174"^^<http://www.w3.org/2001/XMLSchema#nonNegativeInteger>;
    nif:endIndex "3"^^<http://www.w3.org/2001/XMLSchema#nonNegativeInteger>;
    nif:referenceContext <http://articoli.it/sent/bc7f646c6f61068bcbc2eb6499ecc666/3>;
    itsrdf:taldentRef ent:Bce.
<http://articoli.it/entita/Maria%20Bruna%20Riccardi> a foaf:Person.
<http://articoli.it/sent/bc7f646c6f61068bcbc2eb6499ecc666/4/span-115-135> a nif:String;
    nif:beginIndex "115"^^<http://www.w3.org/2001/XMLSchema#nonNegativeInteger>;
    nif:endIndex "20"^^<http://www.w3.org/2001/XMLSchema#nonNegativeInteger>;
    nif:referenceContext <http://articoli.it/sent/bc7f646c6f61068bcbc2eb6499ecc666/4>;
    itsrdf:taldentRef <http://articoli.it/entita/Maria%20Bruna%20Riccardi>.
```

Figura 2.5: Frammento di output dell'rdfer. In verde, una tripla che definisce il collegamento tra l'entità e la risorsa DBpedia.

2.5 CLI: Command Line Interface

La command line interface di PARLEN è pensata per consentire l'esecuzione dei vari moduli sia sotto forma di pipeline, sia singolarmente.

La seconda possibilità permette sostanzialmente di eseguire in momenti diversi ed organizzati le varie fasi che compongono l'estrazione e la manipolazione di dati operata da PARLEN Le operazioni di scraping ed esrazione di entità sono esponenzialmente più dispendiose di tempo e risorse rispetto a quelle operate dagli altri due moduli. Inoltre, eseguendo i moduli singolarmente, è possibile ottenere solo e soltanto gli output che all'interno della pipeline abbiamo definito "intermedi", permettendo quindi la possibilita di riutilizzo da parte di un potenziale nuovo modulo o di un altro script.

Per quanto concerne invece la modalità *pipeline*, sarà sufficiente inserire una query e, dopo l'esecuzione dei vari moduli, otterremo direttamente gli ultimi outputs della catena: i file html e l'ontologia rdf.

I comandi con cui interagire con la command line interface sono della forma:

node cli.js COMMAND [OPTS]

I possibili *command* sono:

- pipeline;
- scrape;
- analize;
- visualize;
- rdf.

Vediamo alcuni esempi concreti di utilizzo:

node cli.js pipeline -q inflazione -s 01-01-2015 -o out/

Questo comando esegue l'intera pipeline: cercherà su tutti i giornali disponibili articoli che contengano la parola "inflazione" pubblicati a partire dal 2015. In seguito a tutta l'analisi dei risultati, nella cartella "out", saranno salvati un' ontologia unica relativa all'intera ricerca ed un file html per ogni articolo trovato. Se avessimo voluto ricercare solo su Repubblica:

node cli.js pipeline -q inflazione -s 01-01-2015 -o out/ -g repubblica

Vediamo ora uno dei 4 comandi che permettono l'esecuzione standalone del modulo a cui si riferiscono:

node cli.js analize -i input.json -o out/

I comandi diversi da pipeline funzionano tutti con un path di input ed uno di output, fatta eccezione per il comando scrape che prende in input i parametri per la query:

node cli.js scrape -q inflazione -o out/

Gli ultimi due comandi eseguono rispettivamente solo il modulo dell'analizer e dello scraper, e salvano i risultati in un file JSON, nella cartella out.

Per la lista completa delle opzioni e dei defaults, fare riferimento alla figura 3.6

Capitolo 3

Dettagli sull'implementazione di PARLEN

3.1 Le tecnologie

PARLEN è interamente sviluppato in *Node.js*. I servizi offerti da Opener, invece, sono sviluppati in più linguaggi, a seconda del modulo considerato. Tra i principali, abbiamo Java, Python e jRuby.

Sostanzialmente, la parte da me sviluppata, quella in Node.js, consiste in una serie di operazioni asincrone come richieste http, lettura e scrittura su disco e comunicazione con i servizi di Opener. Questo permette di sfruttare a pieno le potenzialità offerte dal framework Javascript che è single threaded e fortemente orientato verso la programmazione asincrona. Inoltre, l'unico momento della computazione in cui c'è un alto utilizzo di CPU, cosa in cui Node.js non eccelle, è proprio quando vengono chiamati in causa i servizi di Opener.

Per le librerie ho utilizzato npm, il Node Package Manager[24]. Molte di esse mi hanno aiutato a velocizzare lo sviluppo e ad effettuare le operazioni più specializzate di ogni modulo. Per citare i moduli più importanti che ho utilizzato:

- **cheerio**[25]: utilizzato per lo scraping html;
- **command-line-args**[26]: serve per leggere i parametri da riga di comando;

- **html-entities**[27]: serve per effettuare encoding e decoding di stringhe html;
- **request**[28]: serve per la gestione avanzata di richieste http;
- **xml2json**[29]: converte file XML in JSON, utile per gestire l'output di Opener;
- **N3**[30]: utilizzato per la creazione dell'ontologia in RDF;
- **child_process**[31]: libreria nativa di Node.js per la gestione di processi.

Questi sono solo i moduli più specifici utilizzati. Ovviamente non mancano poi le inclusioni di moduli base come *fs*, per la gestione del file system o *math* per svolgere semplici calcoli matematici. Inoltre, è fondamentale il ruolo di *async*: questo modulo serve per migliorare la gestione di codice asincrono. Mi è tornato utile molti casi, vedremo in dettaglio il perché. Per conoscere tutti i moduli inclusi, fare riferimento al package.json, nella directory principale del progetto.

L'ambiente in cui ho lavorato è *Debian 8*. PARLEN, essendo interamente scritto in Node.js sarebbe compatibile sia con Linux che con Windows che con Osx. Tuttavia la compatibilità è attualmente ristretta a Linux e Osx, in quanto l'applicazione per funzionare presuppone l'installazione di Opener, compatibile solo con sistemi UNIX.

Per l'installazione dei componenti di Opener consiglio di visitare il sito ufficiale del progetto. Bisognerà soddisfare un bel po' di requisiti per la preparazione dell'ambiente necessario, viste le diverse tecnologie con cui Opener è implementato.

3.2 La struttura modulare

In questo paragrafo vedremo come sono organizzati i moduli e i path di PARLEN

Nella directory principale abbiamo:

- **package.json**: file contiene i riferimenti alle dipendenze necessarie;
- **node_modules**: directory contenente le dipendenze;
- **cli.js**;

- **lib**: directory contenente tutti gli altri moduli;
  - **scraper**: directory contenente tutti gli scraper di PARLEN;
    - **repubblica.js**;
    - **sole24ore.js**;
    - **città.json**: JSON contenente la lista di tutte le città italiane;
  - **analizer**: directory contenente tutti gli analizer di PARLEN;
    - **opener.js**;
  - **rdfer**: directory;
    - **rdfer.js**;
  - **visualizer**: directory;
    - **visualizer.js**;
    - **template.hbs** template dell'output del modulo;

Abbiamo già abbondantemente parlato dei motivi per cui la struttura di PARLEN è fortemente modulare ed anche delle possibilità che si aprono con questi presupposti.

Nello specifico, per estendere ad esempio la compatibilità con nuovi giornali o nuovi tool come Opener, sarà sufficiente implementare un modulo in Node.js che rispetti le politiche di input/output del progetto e che si chiami esattamente come il giornale o il tool a cui si riferisce (es. corriere.js o polyglot.js).

Per quanto la struttura dell'applicazione sia modulare e volta a future espansioni e compatibilità, ho preferito sviluppare i vari moduli con un approccio procedurale, senza tuttavia trascurare la stabilità del sistema anche grazie all'utilizzo del modulo async che consente un'ottima gestione dell'errore a cascata, tipico di quest'approccio.

3.3 L'utilizzo di async

Questa libreria è stata ripetutamente utilizzata all'interno di PARLEN Comprenderne le caratteristiche principali tornerà molto utile quando andremo ad analizzare nello specifico il codice.

Async è uno tra i moduli più scaricati di npm. Permette una gestione chiara del codice asincrono ed offre delle performance eccezionali. Sono tante le possibilità ed i casi d'uso offerti da questa libreria ed il suo utilizzo diventa indispensabile quando si fa un forte utilizzo di funzioni asincrone. Vediamo in dettaglio i motivi che mi hanno spinto ad usarlo.

3.3.1 Risoluzione del callback hell

Uno dei problemi più noti quando si sviluppa in maniera asincrona è quello del *callback hell*. Consideriamo un caso semplice: immaginiamo di voler implementare una routine per eseguire una query su un database, scrivere su file il risultato della query, leggere dal file appena scritto e stampare a video il suo contenuto. Consideriamo questa implementazione che non utilizza async:

```
var fs = require('fs');
var db = require('db').connect('localhost:9000');

db.find('articles', function(err, articles) {
    fs.writeFile('/tmp/myfile', articles, function(err) {
        fs.readFile('/tmp/myfile', function(err, file) {
            console.log(file);
        })
    })
});
```

Figura 3.1: Esempio tipico di callback hell.

Quì è evidente come, a lungo andare, il codice diventi ingestibile e si tenda sempre più ad andare a scrivere verso destra. Una delle possibili implementazioni con async del codice sopra è quella che utilizza il metodo *waterfall* che, appunto, vuol dire cascata. Esso prende in input un array di funzioni in ognuna delle quali eseguire le chiamate asincrone, ed una ultima funzione che sarà eseguita quando l'ultima chiamata asincrona (quella che legge il file dal disco) avrà terminato la sua esecuzione.

```
var fs = require('fs');
var db = require('db').connect('localhost:9000');
var async = require('async');

async.waterfall([

    function(callback) {
        db.find('articles', function(err, articles) {
            callback(articles);
        });
    },

    function(callback, articles) {
        fs.writeFile('/tmp/myfile', articles, function(err) {
            callback()
        });
    },

    function() {
        fs.readFile('/tmp/myfile', function(err, file) {
            callback()
        });

    }], function(err, articles) {

        console.log(articles);
    }
);
```

Figura 3.2: Utilizzo del metodo waterfall di Async per evitare il callback hell.

In un primo momento questa implementazione può sembrare confusionaria. In realtà, quando si ha la necessità di implementare una lunga cascata di funzioni asincrone, questo è il modo più performante che abbiamo per evitare il callback hell.

3.3.2 Gestione parallela di gruppi di chiamate a funzioni asincrone

Consideriamo il caso in cui volessimo effettuare 10 richieste HTTP, per poi, solo dopo aver effettuato anche l'ultima, salvare su disco un file contenete il body di tutte le richieste. Tenendo presente il fatto che comunque non ci interessa l'ordine in cui le richieste sono soddisfatte, consideriamo la seguente implementazione (errata):

```
var request = require('request');
var urls = ['url1', 'url2' ... ];
var results = [];

for (var i = 0; i < urls.length; i++) {

    request.get(urls[i], function(err, res, body) {
        results.push(body);
    })
}

fs.writeFile('/tmp/file', results, function(err) {
    process.exit();
})
```

Figura 3.3: Implementazione errata di richieste http multiple.

Il problema sta nel fatto che il file risulterà essere quasi sempre vuoto. Questo perché, concluso il ciclo for, nessuno ci assicura che tutte le richieste sono state soddisfatte, eppure viene richiesto di scrivere su disco il contenuto della variabile results. Essa sarà verosimilmente vuota o incompleta, dato che per l'esecuzione dell'intero for sono necessari pochi millisecondi e tendenzialmente per la risposta di anche una sola richiesta HTTP, il tempo richiesto è maggiore.

Vediamo ora una implementazione con async:

```
var request = require('request');
var async = require('async');
var urls = ['url1', 'url2' ... ];

async.map(urls, function(url, cb) {

    request.get(url, function(err, res, body) {
        callback(null, body);
    });

    },

    function(err, results) {
        fs.writeFile('/tmp/file', results, function(err) {
            process.exit();
        })
    }
);
```

Figura 3.4: Metodo map di Async utilizzato per richieste HTTP multiple

Anche in questo caso, il codice pare essere meno leggibile. Tuttavia la logica è molto semplice da comprendere: il metodo prende in input l'array con gli url e due funzioni: la prima sarà eseguita per ogni elemento dell'array ed al suo interno viene effettuata la richiesta http; la seconda è una funzione che sarà eseguita soltanto quando tute le richieste saranno terminate.

Con questo paragrafo ho voluto spiegare il funzionamento dei due metodi di async che più ho utilizzato all'interno di PARLEN Async può essere utilizzato per tantissimi altri scopi, alcuni dei quali analizzeremo in seguito. La documentazione della libreria è comunque chiara ed esaustiva. Inoltre le performance sono eccellenti. Fino al 2013, Async infatti riusciva a gestire funzioni asincrone meglio di altre librerie che utilizzano approcci differenti[32] (promise), sebbene queste ultime abbiano una sintassi d'utilizzo molto più intuitiva. Ora Bluebird, un'altra libreria che utilizza un approccio differente, ha addirittura superato le prestazioni di Async, comunque tutt'ora eccellenti.

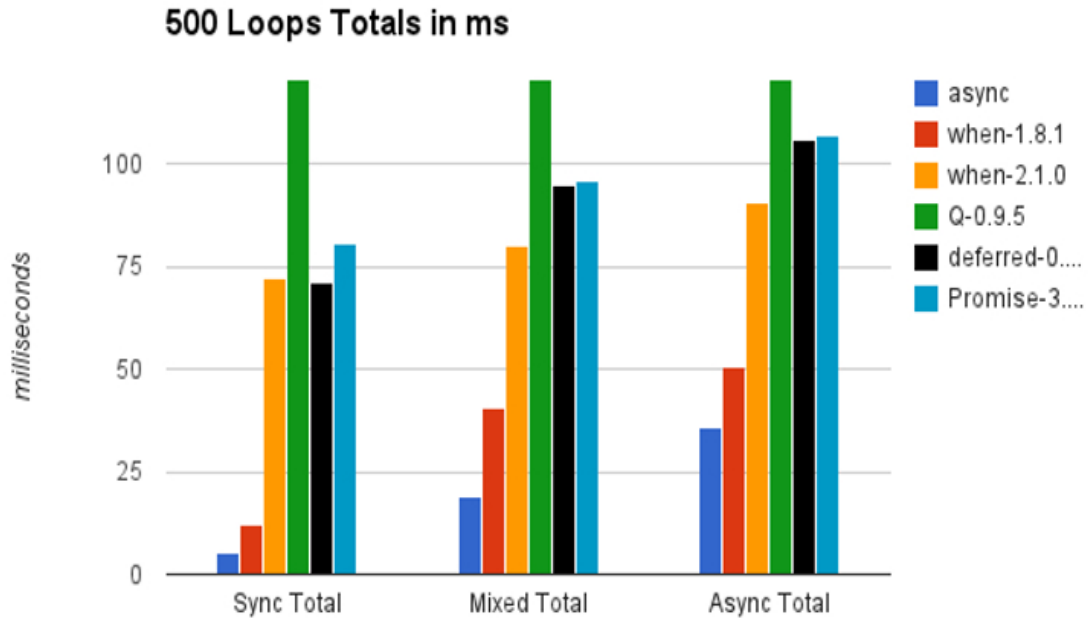


Figura 3.5: Confronto performance tra async e altre librerie per la gestione di chiamate asincrone

3.4 Implementazione

3.4.1 La CLI

Abbiamo visto nel capitolo precedente come funziona la command line interface di PARLEN Vediamone ora i punti salienti dell'implementazione.

```
var cli = commandLineArgs([
  { name: 'command', alias: 'c', type: String, defaultOption: true },
  { name: 'query', alias: 'q', type: String },
  { name: 'startD', alias: 's', type: String, defaultValue: ''},
  { name: 'endD', alias: 'e', type: String, defaultValue: '' },
  { name: 'mode', alias: 'm', type: String, defaultValue: 'and' },
  { name: 'input', alias: 'i', type: String, defaultValue: '' },
  { name: 'output', alias: 'o', type: String, defaultValue: '' },
  { name: 'giornali', alias: 'g', type: String, defaultValue: ['repubblica', 'sole24ore'], multiple: true },
  { name: 'analizer', alias: 'g', type: String, defaultValue: 'opener' },
]);

var args = cli.parse();
```

Figura 3.6: Comandi, opzioni e defaults del modulo cli.js

Questo snippet si occupa della gestione dei parametri da riga di comando, attraverso l'utilizzo della libreria command-line-args. Essa permette di definire in modo intuitivo tutti i parametri di cui necessitiamo, tipizzandoli, assegnandogli un eventuale valore di default oppure definendone uno come il principale. In questo caso, *command* è il parametro principale (default option). Da notare il parametro *giornale*: se sarà sviluppato un modulo per supportare un nuovo giornale, e si vorrà includere quest'ultimo all'interno della pipeline di default, sarà necessario modificare i relativi valori nell'array, inserendo in questo caso la stringa "corriere", se il modulo si chiama corriere.js.

Dopo la chiamata alla funzione *parse()*, otteniamo un oggetto con i parametri passati dall'utente. Attraverso l'utilizzo di un costrutto *switch-case*, distinguiamo i cinque possibili commands, eseguendo la routine corrispondente. Sono particolarmente interessanti i casi *scrape* e *pipeline*. Il primo due deve eseguire tanti processi quanti sono i giornali specificati nella query. La chiamata con cui eseguire un processo (spawn) con Node.js è asincrona. Abbiamo quindi bisogno di utilizzare la funzione map di async per gestire in maniera ottimale l'output e gli errori dei moduli chiamati in causa.

```
case 'scrape':
    async.map(args.giornali, function(g, cb){
        var child = child_process.spawn('node', scraperArgs(args, false, g));

        child.stdout.on('data', function (data) {
            log.info(g + ' output: ' + data);
        });

        child.stderr.on('data', function (data) {
            log.error(g +' err: ' + data);
            cb(data);
        });

        child.on('exit', function (exitCode) {
            log.info("Child exited with code: " + exitCode);
            cb(null, undefined)
        });

    }, function(err, data) {
        if (err)
            console.log('Si è verificato un errore', err);
        else
            console.log('Fine Scraping');
    });
    break;
```

Figura 3.7: Snippet: implementazione del comando “scrape” in cli.js

Nella proprietà “giornali” dell’oggetto *args* c’è un array di stringhe (che rappresentano i giornali). Sarà quindi ripetutamente chiamata la funzione spawn, che creerà un nuovo processo per ogni stringa. Il comando eseguito sarà “node” concatenato con la stringa calcolata dalla funzione *ScraperArgs*. Esempio:

node /lib/scraper/repubblica.js -q inflazione

Per ogni command della CLI, è definita una funzione simile alla ScraperArgs. Queste funzioni servono per generare una stringa che altro non è che il comando che avremmo digitato sul terminale per eseguire il modulo relativo. Infatti anche lo scraper, l’analizer il visualizer e l’rdfer funzionano attraverso parametri passati da riga di comando. Tuttavia questa funzionamento viene nascosto all’utente che si interfaccia solo con la CLI che possiamo dire funga da “wrapper di comandi”.

Gli snippets che gestiscono gli altri comandi sono simili se non più semplici di quello mostrato, che deve gestire anche la possibilità di dover eseguire più di un processo. Caso particolare per il comando pipeline, quello che costruisce appunto la pipeline: esso

è implementato utilizzando la chiamata waterfall di async. Viene (o vengono) quindi prima eseguiti lo scraper, poi l'analizer e così via.

Un'altro dettaglio importante riguarda la differente gestione dei file in input ed output a seconda che si stia utilizzando un comando specifico o il comando pipeline. Il primo caso è più semplice: infatti, è necessario specificare il path del file in input (o la query se si sta utilizzando lo scraper) ed il path di una directory di output. Anche nel secondo caso saremo obbligati ad inserire una query (perché la pipeline inizia sempre dallo scraper) e bisognerà specificare un path di output. Tuttavia, i file intermedi generati da scraper ed analizer saranno salvati in /tmp. Considerando un esempio concreto: se volessi utilizzare solo l'analizer passandogli in input un file (precedentemente generato dallo scraper), i path di input/output utilizzati saranno quelli specificati dall'utente. Quando invece si utilizza il comando pipaline e sarà il turno dell'analizer, esso dovrà leggere e scrivere in /tmp. Serve un modo per comunicare ai vari moduli se essi sono stati eseguiti all'interno di una pipeline o no, in modo da regolarsi sui path di lettura e scrittura. Per risolvere questo problema ho utilizzato un parametro supplementare che ho chiamato *tmp*. Esso è un booleano, viene gestito nella CLI ed è utilizzato all'interno di tutti gli altri moduli.

3.4.2 Lo Scraper

Come già si può intuire dalla descrizione fatta in precedenza di questo insieme di moduli, il codice dei due scraper implementati differisce per certi aspetti. Vediamo prima però in cosa i moduli di Repubblica e del Sole 24 Ore si assomigliano.

In entrambi, all'inizio del file, sono dichiarate alcune funzioni che possiamo definire *helpers*. Servono a leggere le date, effettuare operazioni con stringhe ed espressioni regolari, rimuovere duplicati da un array etc. Inizialmente avevo pensato di sviluppare un ulteriore modulo che contenesse questi helpers. Tuttavia, mi sono reso conto che i differenti formati utilizzati dai due archivi non mi consentivano di rendere tali funzioni così tanto generiche. Per questo ho preferito tenere all'interno di ogni scraper la definizione degli helpers di cui esso necessita.

Un altro punto in comune riguarda l'utilizzo di request per effettuare chiamate HTTP

e di async, sia con il metodo waterfall che con il metodo map che, lo abbiamo visto, serve per eseguire gruppi di chiamate asincrone. In particolare, in questo modulo sono stato costretto ad usare una variante della funzione map, la *mapLimit*. Essa funziona come la map, con la differenza che limita ad un intero specificato il numero massimo di chiamate asincrone attive nello stesso tempo. In altri termini, nel nostro caso, viene utilizzata per limitare il numero di connessioni attive contemporaneamente, implementando un sistema di throttling ed evitando di essere bannati dai server degli archivi.

```
async.mapLimit(articles, 1, function(article, cb) {
    request(article.url, function(err, res, body) {
        article.err = err, article.res = res, article.body = body;
            cb(null, article);
        });

    },function(err, articles) {
        callback(null, articles)
    }
);
```

Figura 3.8: Implementazione di un sistema di throttling con async e request

Da notare come all'interno dell'oggetto *article* viene creata la proprietà "err", che conterrà l'eventuale errore verificatosi in seguito ad una particolare richiesta. Quest'approccio evita di dover terminare l'esecuzione nel caso una o più richieste dovessero fallire e permette comunque la gestione dell'errore in seguito, attraverso un' helper.

Altro aspetto comune ai due moduli riguarda il fatto che prima di eseguire le richieste http per scaricare gli articoli, bisogna effettuarne una (o più) preliminare per capire quali e quanti sono gli url da contattare per il download degli articoli. La gestione è particolarmente macchinosa: bisogna fare una prima richiesta per ottenere il numero totale degli articoli che corrispondono alla query. Essi sono a loro volta suddivisi in pagine che ne contengono 10 ciascuna (nel caso di Repubblica). Bisognerà quindi effettuare ulteriori n/10 richieste per ottenere i veri e propri link agli articoli.

Se tuttavia con l'archivio storico di Repubblica che, lo ricordo, opera rendering lato server, non è possibile ottenere più di 10 articoli alla volta, attraverso le API dell'archivio storico del Sole 24 Ore è possibile aumentare questo limite fino a 1000: all'interno del

sito web gli articoli sono suddivisi in pagine da 20 ma attraverso la sola modifica di un parametro si può ad aggirare il limite, ottenendo migliori performance ed un utilizzo minore di risorse e soprattutto tempo.

Una volta ottenuti gli articoli, il codice dei due moduli differisce sostanzialmente. Vengono eseguite le routine che si occupano dello scraping vero e proprio e che differiscono a seconda di quanto differiscono i formati dei dati in input.

Il caso del Sole 24 Ore è più semplice da gestire: gli articoli sono in formato JSON, anche se le stringhe in esso contenute contengono tag ed entità HTML. Per recuperare i dati è sufficiente leggere i valori interessati dal JSON per poi effettuare alcune semplici operazioni di pulizia dei dati prevalentemente attraverso l'utilizzo di espressioni regolari. Tra le principali abbiamo:

- sostituzione di tutti i tag "br" con un carattere \n;
- eliminazione di tutti i tag html;
- eliminazione di caratteri speciali a inizio e fine stringa;
- decoding delle entità html, attraverso la libreria html-entities.

Nel caso di Repubblica bisogna effettuare non poche operazioni, molte delle quali chiamano in causa la libreria cheerio per il parsing di HTML. Tra gli helpers dichiarati all'inizio del modulo, abbiamo due metodi: *parseWeb()* e *parseCarta()*. Entrambi prendono in input l'intera pagina HTML di un articolo e la parsano, cercando di estrarre le varie informazioni di cui necessitiamo. L'esistenza di questi due metodi è solo il primo filtro attraverso cui differenziare l'analisi da operare su una pagina HTML di un articolo pubblicato nel sito web del giornale dall'analisi di una pagina HTML di un articolo pubblicato nella versione cartacea e riportato poi nell'archivio. I due formati sono completamente diversi e comunque, all'interno dei due gruppi, esistono ulteriori differenze strutturali o errori da gestire. Entrambi i metodi in primo luogo rimuovono tag HTML che sicuramente non conterranno nulla di utile come script, iframe, .adv, figure, etc.

In seguito il metodo parseWeb() cerca il titolo, il body e l'autore dell'articolo all'interno di tag che abbiano una determinata classe, id o attributo a seconda della categoria

del sito in cui l'articolo è stato pubblicato. Questo perché, a seconda che l'articolo sia stato pubblicato nella sezione Tecnologia, piuttosto che Economia, piuttosto che Cronaca e via dicendo, la struttura delle pagine cambia.

Se però con il metodo parseWeb() bisogna soltanto cercare di supportare quanti più formati è possibile, il metodo parseCarta() non ha nessun riferimento forte da cui partire, come può essere un div con una determinata classe. Gli articoli pubblicati in questa sezione non hanno una struttura ben definita. Tendenzialmente i titoli sono dei tag h1 ed il body è un insieme disordinato di paragrafi, senza alcuna classe. L'autore spesso è citato alla fine del testo, in maiuscolo ed è attraverso questa informazione che lo scraper prova a recuperare il dato. Inoltre, spesso alla fine degli articoli compare la stringa "Riproduzione Riservata". L'ho utilizzata come delimitatore per distinguere il body dell'articolo dal footer.

Entrambi i metodi appena spiegati operano inoltre operazioni con espressioni regolari per mantenere i dati quanto più puliti possibile. Molte di esse sono simili a quelle citate in precedenza per il modulo del Sole 24 Ore.

Inoltre, durante le operazioni di scraping, c'è un'altra routine comune ad entrambi i moduli: nella cartella scraper, il file città.json contiene la lista di tutte le città italiane. Attraverso una funzione e con l'aiuto di un'espressione regolare, gli script riescono a comprendere se il body di un articolo inizia con il nome di una città. In quel caso, essa viene tagliata dal body e trattata come un dato ulteriore. Interessante l'espressione regolare utilizzata:

```
for (var i = 0; i < citta.length; i++) {
    var regex = new RegExp('^' + citta[i] + '[\r\n ,.-]', 'i');
    if (regex.test(article.body)) {
        /*elimina la città all'inizio dell'articolo*/
        article.body = article.body.replace(regex, '');
        /*elimina eventuali spazi, punti o trattini rimasti a inizio stringa*/
        article.body = article.body.replace(/[\.\-\s\,]*/i, '').trim();
        article.citta = citta[i]
    }
}
```

Figura 3.9: Snippet per l'estrapolazione della città dal body di un articolo

Come si evince anche dai commenti, sono stati utilizzati alcuni accorgimenti per riconoscere la città senza tuttavia rischiare di riconoscere dei falsi negativi (es *Romano Prodi ha dichiarato...* Riconoscere Roma come città dell'articolo sarebbe un errore).

In seguito alle operazioni di scraping, il vettore d'articoli viene filtratto attraverso ulteriori operazioni elementari. Tra le principali abbiamo:

- eliminazione articoli duplicati;
- eliminazione articoli con body troppo corto;
- eliminazione articoli con url non funzionante;
- eliminazione articoli scritti a metà (riconosciuti dalla presenza di stringhe particolari).

Il JSON è pronto per essere scritto sul disco, nel path di output specificato dall'utente o nella cartella /tmp (in caso si stesse eseguendo lo scraper all'interno dell'intera pipeline).

3.4.3 L'Analizer

L'implementazione di questo modulo inizialmente credevo sarebbe stata particolarmente veloce. Tuttavia ho riscontrato qualche problema che mi ha costretto a scrivere qualche riga di codice in più.

L'idea iniziale è molto semplice: leggere il file generato dallo scraper, darlo in input ai servizi di Opener, aspettare la fine dell'esecuzione e salvare un nuovo JSON, contenete i risultati dell'analisi. Due problemi, causati da Opener, hanno reso però l'implementazione più sofisticata:

1. Servizi web di Opener non funzionanti correttamente;
2. modulo kaf2json di Opener rallenta drasticamente l'esecuzione.

Come avevamo accennato, tutti i moduli di Opener possono funzionare come servizi HTTP (attraverso un server HTTP dedicato ad ogni componente) oppure possono essere eseguiti all'occorrenza da riga di comando. La prima soluzione mi sembrava molto più comoda da gestire ed anche molto più performante. Avevo anche implementato una routine preliminare che, prima di procedere con l'analisi, lanciava un'istanza server di ogni componente di Opener di cui PARLEN avesse avuto bisogno. Questo rallentava leggermente l'avvio di opener.js. Tuttavia, i servizi dovevano caricare in memoria le dipendenze di cui necessitavano solo all'inizio, garantendo una maggiore velocità con il proseguire dell'esecuzione.

Non ho comunque potuto procedere con questo approccio perché i servizi web di Opener non funzionano correttamente. Nonostante svariati tentativi e reinstallazioni dell'intero pacchetto software, i servizi web assumevano comportamenti anomali: talvolta partivano nel giro di qualche secondo, senza causare alcun problema. Altre volte, invece, prima che il servizio web fosse pronto, dal momento del lancio passavano anche 4, 5 minuti, e la CPU era in idle. Il comportamento era anomalo e le attese inaccettabili.

Per questo motivo sono stato costretto ad eseguire, per ogni artiolo da analizzare, il comando per lanciare i servizi del tool.

Mi sono quindi servito della libreria child_process e del suo metodo *exec* per gestire l'esecuzione dei servizi di Opener e, ovviamente, di async per gestire la concorrenza.

```
async.mapLimit(data, 1, function(article, cb) {
    var start = process.hrtime();
    /*var cmd = buildCmd(article)*/
    exec(cmd, {maxBuffer: 1024 * 2000}, function (error, stdout, stderr) {
        var end = process.hrtime(start);
        var words = prettyHrtime(end); // '1.2 ms'
        if (error) {
            console.log(error);
            article.log = {
                time: parseFloat(words),
                error: error,
                stderr: stderr,
            }
            return cb(null, article);
        }

        article.analizer = JSON.parse(xml2json.toJson(stdout))
        article.log = {}
        article.log.time = words;
        return cb(null, article);
    })
}, function(err, articles) {
    /*...*/
);
```

Figura 3.10: Snippet per l'esecuzione dei servizi di Opener

La funzione *mapLimit()* è limitata ad 1: in altri termini sarà eseguita l'analisi un articolo alla volta: i servizi di Opener sfruttano di default tutti i core a disposizione. Parallelizzare le analisi non avrebbe avuto alcun effetto positivo. L'istruzione commentata alla terza riga è una semplificazione che mi sono permesso di fare: al posto di quella funzione, nel codice ci sono alcune righe di codice che compongono il comando da dare in input alla funzione exec. Non essendo interessanti ho preferito non riportarle. Il comando generato è comunque qualcosa del tipo:

cat /tmp/tmpfile.txt | *language-identifier* | *tokenizer* | *pos-tagger* | *ner* | *ner*

La semplicità di creazione della pipeline è davvero notevole. Il file temporaneo contiene il body dell'articolo da analizzare, mentre un altro appunto va fatto sul modulo del language-identifier: sarebbe possibile non utilizzarlo, risparmiando tempo e risorse, specificando al tokenizer che la lingua del testo in input è l'italiano. Tuttavia il tokenizer

pare non voler funzionare col parametro della lingua: nonostante abbia seguito alla perfezione il tutorial sul sito, riscontro un errore sulla gestione della memoria, probabilmente un segmentation fault.

Ho riportato invece le istruzioni attraverso cui vado al calcolare il tempo d'esecuzione dei servizi per ogni articolo: il metodo hrtime ha una precisione al millisecondo. Non è una feature utile ai nostri scopi ma è comunque un dato interessante.

Ancora una volta la gestione dell'errore viene posticipata: se per qualsiasi motivo, durante l'analisi di uno degli articoli, i servizi di Opener dovessero andare in crash, l'esecuzione proseguirà e nel file output potremmo analizzare il problema verificatosi.

L'istruzione *xml2json.toJson(stdout)* è invece correlata col secondo problema, quello legato al modulo kaf2json di Opener. Una precisazione: Opener salva tutti gli outputs dei suoi componenti nel formato KAF, un formato praticamente identico all'XML. Il componente kaf2json si occupa di convertire in JSON questo formato. Purtroppo però, per la sua esecuzione, talvolta sono necessari anche 6 o 7 minuti, ancora una volta con la CPU in idle. Anche in questo caso ogni tentativo di fix non è andato a buon fine ed alla fine sono stato costretto ad utilizzare la libreria in node.js xml2json. Le performance sono eccezionali: per quanto la dimensione l'Output di opener talvolta superi il Megabyte, questa libreria converte in qualche decimo di secondo i dati in JSON. Purtroppo però, i tag XML vengono interpolati in un JSON con una struttura particolarmente complessa, contrariamente a quanto fa il kaf2json nativo di Opener.Questa complessità nella struttura dell'oggetto mi ha costretto ad implementare ulteriori routine per recuperare entità, frasi in cui vengono citate, offsets e links DBpdia.

Il codice che segue l'esecuzione di Opener consiste in alcuni cicli for attraverso cui costruire l'oggetto che diventerà poi l'output del modulo. Ad ogni articolo viene aggiunta la proprietà "analizer" che include i dati a noi utili provenienti dell'analisi di Opener, riorganizzati in un nuovo formato più pratico per l'utilizzo con i moduli successivi.

Un ultimo dettaglio interessante sull'implementazione di Opener.js:

```
var steps = [
    'language-identifier',
    'tokenizer',
    'pos-tagger',
    'tree-tagger',
    'ner',
    'ned'
];
```

Figura 3.11: Snippet contenente i moduli della pipeline di Opener eseguiti da PARLEN

Quest'array è dichiarato all'inizio del file e contiene, in ordine, il nome dei vari componenti chiamati da noi in causa durante la pipeline di Opener. Qualora si volessero includere nuovi servizi, dopo averli installati, sarà sufficiente modificare il contenuto della variabile.

3.4.4 Il Visualizer

L'implementazione di questo modulo non è risultata particolarmente difficoltosa, complice anche il fatto che, non dovendo utilizzare alcuna chiamata asincrona, la libreria async non è inclusa ed il codice risulta essere più leggibile.

Per generare i file HTML mi sono servito di *handlebars*[33], una libreria per eseguire rendering di pagine web.

Dopo la dichiarazione dei soliti helpers, l'intero script è contenuto in un ciclo for che itera sugli articoli letti dal file in input. All'interno di ogni interazione vengono eseguite sostanzialmente due operazioni.

In primis si va ad aggiungere l'interattività all'articolo. Il body viene ricostruito e trasformato in una stringa HTML in cui tutte le entità sono contenute all'interno di un tag "a" che ne permette la colorazione ed il collegamento ipertestuale alle risorse di DBpedia. Nel corso di questa operazione, i caratteri speciali contenuti nel body, vengono ritrasformati in entità HTML per la loro corretta visualizzazione sul browser. Per effettuare l'encoding, ho utilizzato la stessa libreria con cui nello scraper del Sole 24

Ore avevo effettuato il decoding, html-entities. Inoltre, tutte le occorrenze del carattere \n vengono risostituite con il tag “br”.

Creato il nuovo body, si passa alla creazione del file HTML vero e proprio. Utilizzare handlebars è molto semplice: all’interno della cartella “visualizer” c’è il file *template.hbs*. Si tratta di un file HTML contenente solo la struttura della pagina ed al posto dei contenuti ci sono delle variabili, un classico esempio di template con una sintassi inoltre molto simile a quella utilizzata in *angular.js*, versione 1. Attraverso le API di handlebars, riusciamo ad ottenere in memoria una stringa HTML che altro non è che il template compilato con l’oggetto passato in input alla libreria. Si tratta dell’oggetto article, contenente tulle le informazioni relative all’articolo opportunamente codificate ed il body ricostruito in precedenza.

A questo punto la stringa viene trasformata in un file attraverso l’utilizzo della libreria *fs* nativa di Node.js. Per precisazione, la funzione *template* è legata ad handlebars e

```
fs.writeFileSync(outputPath, template(context));
```

Figura 3.12: Chiamata alla funzione writeSync della libreria nativa “fs” di Node.js

context è un oggetto contenente tutti i dati dell’articolo.

Ho preferito utilizzare la funzione *writeFileSync* piuttosto che la più comune versione asincrona perché non volevo complicare la gestione del codice dovendo ricorrere nuovamente all’utilizzo di async. Bisogna tenere presente che l’istruzione in figura è l’ultima del ciclo for. Viene eseguita per ogni articolo e, fosse asincrona, ricade quindi nel caso già affrontato di gruppi di chiamate asincrone.

Per quanto questa soluzione dovrebbe degradare le prestazioni, dato che i files generati sono di piccolissime dimensioni, il modulo riesce comunque a scriverne anche 100 in qualche frazione di secondo.

Continuando a parlare di prestazioni, va fatta una piccola precisazione su una istruzione correlata ad handlebars:

```
var handlebars = require('handlebars')
var template = fs.readFileSync(__dirname + '/template.hbs').toString();
template = handlebars.compile(template);
```

Figura 3.13: Chiamata alla funzione compile della libreria "handlebars" di Node.js

All'inizio del modulo, per predisporre l'utilizzo di handlebars, vengono eseguite queste tre istruzioni. La seconda ci permette di capire da dove proveniva la variabile dello snippet in figura 3.12, l'ultima è la più interessante: il metodo *compile* precompila il template in maniera tale che tutte le volte che sarà associato ad un oggetto per la compilazione vera e propria, le performance saranno nettamente superiori. Ho fatto degli esperimenti ed effettivamente si guadagnano fino a 300ms per chiamata, soprattutto quando il numero file da compilare supera i 100.

3.4.5 L'Rdfer

Per quanto il compito svolto da questo modulo è completamente diverso da quello svolto dal visualizer, le somiglianze implementative sono tante. Anche qui non ho avuto la necessità di utilizzare async e l'approccio utilizzato è quindi estremamente sincrono e procedurale. La libreria più utilizzata è *N3* che permette la lettura, creazione e manipolazione di file RDF.

Letto il file JSON in input, la prima operazione effettuata consiste nella dichiarazione di tutti i prefissi che saranno utilizzati all'interno dell'ontologia.

```
var writer = N3.Writer({ prefixes:
    {
        bibo: 'http://purl.org/ontology/bibo/#',
        foaf: 'http://xmlns.com/foaf/0.1/',
        owl:  'http://www.w3.org/2002/07/owl#',
        po:   'http://www.essepuntato.it/2008/12/pattern#',
        doco: 'http://purl.org/spar/doco/',
        c4o:  'http://purl.org/spar/c4o/',
        art:  'http://articoli.it/articolo/',
        /*........*/
    },
});
```

Figura 3.14: Dichiarazione dei prefissi utilizzati nell'ontologia prodotta dal modulo "rdfer" di PARLEN

Poi, con una iterazione su tutti gli articoli, utilizzando il metodo *addTriple()* di N3, si costruiscono le varie triple che comporranno il file in output, seguendo la struttura mostrata nel paragrafo 2.4. Per tutta la durata dell'esecuzione il file viene tenuto in memoria e solo alla fine viene scritto su disco, utilizzando questa volta il più comune metodo asincrono. Questo perché in questo caso i dati vengono accorpati tutti in un unico file e la chiamata è soltanto una, fuori dall'interazione.

Durante il corso di tutta la procedura, entità che si ripetono più volte nei vari articoli vengono trattate creando un solo riferimento. Tuttavia, su analisi separate questo comportamento non può essere replicato. Nel caso in cui si volessero accorpare le ontologie prodotte da PARLEN in una unica base di dati, bisognerà tener conto di ciò prendendo le dovute precauzioni.

Gli unici due helpers utilizzati all'interno del modulo sono la funzione *hash()* e la funzione *getType()*.

La prima calcola l'hash dell'url dell'articolo: esso sarà utilizzato come identificativo unico nell'URI che lo rappresenta.

La seconda associa alla tipologia di un'entità il corrispettivo URI che la descrive semanticamente.

Per la creazione di alcune triple, sono stato costretto ad utilizzare il metodo nativo di Node.js *encodeURI()* che opera l'encoding dell'URI passato in input, rimpiazzando

spazi e caratteri speciali coi simboli corrispondenti.

```
if (obj.analizer[i].dbpedia)
        writer.addTriple(
                encodeURI('http://articoli.it/entita/' + obj.analizer[i].entita),
                'http://www.w3.org/2002/07/owl#sameAs',
                encodeURI(obj.analizer[i].dbpedia)
        );
```

Figura 3.15: Utilizzo del metodo nativo “encodeURI()” di Node.js

Nessun’altra istruzione è degna di particolare nota all’interno di questo modulo. D’altronde, lo sforzo implementativo maggiore sta nello sviluppo della logica con cui collegare le entità ad articoli, frasi, link DBpedia, offsets, tipologia ecc, evitando ricorrenti duplicati.

Capitolo 4

Risultati e Performance

Dopo averne visto anche i dettagli sull'implementazione, valuteremo PARLEN sia dal punto di vista della qualità dei dati prodotti, sia dal punto di vista della stabilità e delle performance. Certamente terremo conto anche della qualità del lavoro svolto dai servizi Opener che hanno un ruolo centrale, forse il più importante all'interno dell'intera pipeline.

Analizzeremo ancora una volta i moduli singolarmente: soprattutto per quanto concerne la qualità dei dati, è fondamentale analizzare i risultati dei vari componenti della pipeline in ordine di come vengono eseguiti dato che eventuali errori che potrebbero verificarsi nelle prime fasi dell'esecuzione possono ripercuotersi negativamente quando saranno eseguiti i moduli successivi.

Per quello che riguarda l'ambiente, è stata utilizzata una macchina virtuale con sistema operativo Debian 8 e Node.js ver. 5, 8 core virtuali a 4Ghz, 8gb di RAM. La potenza dell'hardware sarà realmente sfruttata solo da Opener in quanto Node.js è single threaded e comunque le fasi di CPU burst non sono mai particolarmente dispendiose di risorse.

4.1 Scraper

Per valutare la qualità dei dati prodotti dai due scaper, ho effettuato una query che restituisse in entrambi i giornali circa 100 articoli. Ho confrontato poi gli output dei moduli con le pagine web degli archivi dei giornali, verificando la presenza e la correttezza dei dati effettivamente disponibili.

La query che ho effettuato, sia per Repubblica che per il Sole 24 Ore è la seguente:

node cli.js scrape -q inflazione -s 01-04-2015 -e 30-04-2015

Ossia: tutti gli articoli in cui compare la parola inflazione pubblicati nel mese di aprile 2015.

Nell'archivio di Repubblica, gli articoli trovati sono 94, in quello del Sole 24 Ore sono 144. Per comodità, analizzeremo i risultati dei primi 90 articoli provenienti da entrambi i giornali.

Nella valutazione non terremo conto nè del dato relativo alla città nè di quello relativo al footer. Nel primo caso, è sufficiente sapere che se un articolo inizia col nome della città a cui si riferisce, questa sarà sempre riconosciuta, a patto che sia una città italiana o che comunque sia inclusa nel file citta.json (vedi paragrafo 3.4.2). Per quanto riguarda il footer, esso è un dato supplementare contenuto solo in alcuni articoli provenienti dall'archivio di Repubblica; in particolare è contenuto solo negli articoli pubblicati nella versione cartacea del giornale. Contiene spesso informazioni relative all'autore del giornale e comunque, il suo contenuto non viene preso in considerazione dall'analizer.

Inoltre, nelle pagine di alcuni articoli provenienti da entrambi i giornali non viene specificato talvolta il titolo, più spesso l'autore. Non essendo un bug ma un'informazione mancante, non ho tenuto conto di questi casi.

	Repubblica	**Sole 24 Ore**
Dimensione campione	90	90
Falsi positivi	1	1
Falsi negativi	0	0
Problemi con l'autore	9	0
Problemi con il titolo	1	0
Problemi con il body	4	0

Tabella 4.1: Qualità dati prodotti dallo scraper

Come si evince dalle tabella, i dati prodotti in output dallo scraper del Sole 24 Ore hanno sicuramente una qualità migliore. La cosa non stupisce: avevamo già ripetutamente parlato dei problemi relativi ai formati degli articoli di Repubblica e come ci potevamo aspettare, l'output di repubblica.js non è pulitissimo ma non per questo non accettabile.

Entrando nel dettaglio, per problemi con l'autore vogliamo intendere sia i casi in cui al posto dell'autore del giornale, lo scraper riconosce un'altra informazione, sia quelli in cui, nonostante nella pagina dell'articolo l'autore sia citato, lo scraper non è in grado di riconoscerlo o, più spesso, lo confonde con il footer o lo incude nel body. Per problemi con titolo e con body, invece, intendiamo dire che lo scraper ha riconosciuto l'informazione ma quest'ultima presenta ancora qualche carattere speciale privo di significato. Tali caratteri ovviamente non sono generati erroneamente dal modulo ma sono già presenti negli articoli. Ho provato comunque a ripulire i dati, riuscendoci in molti casi ma non in tutti. Ovviamente sono sempre possibili ulteriori migliorie ma, per rendere questo modulo perfetto bisognerebbe prevedere ogni errore strutturale, senza comunque avere la garanzia di riuscire a recuperare tutti i dati visto che, alcuni articoli, una vera e propria struttura non ce l'hanno.

Figura 4.1: Esempio di articolo senza struttura e con errori ortografici ad inizio body

I risultati sono comunque più che soddisfacenti, sia perché il numero di incongruenze o problemi è comunque basso, sia perché il problema maggiore, quello con gli autori, non si ripercuote in alcun modo all'interno degli altri componenti della pipeline: il dato è pensato per essere recuperato e messo a disposizione dell'utente. Nessuna ulteriore operazione d'analisi viene eseguita con l'autore dell'articolo.

Caso diverso per il body: la presenza di caratteri speciali dovrebbe in teoria incidere sui risultati della pipeline di Opener, che lavora proprio su questo dato. Tuttavia, come vedremo, i caratteri speciali non causeranno particolari problemi, a differenza invece degli errori ortografici talvolta presenti.

Riguardo alle performance, nonostante l'implementazione del throttling, l'esecuzione termina in tempi ragionevoli. Ovviamente non si possono stabilire delle tempistiche assolute, visto che la parte onerosa di tempo è quella in cui avvengono le varie richieste HTTP. Il tempo di risposta per ogni articolo è influenzato dalla qualità, stabilità e velocità della connessione del client ed anche dal traffico e dai tempi di risposta del server.

Con una connessione a 100mbps, entrambi i moduli scaricano gli articoli in un tempo medio di 0,5s. La fase di scraping vero e proprio, poi, avviene in qualche frazione di

secondo, complici anche l'utilizzo di espressioni regolari come principale strumento per la pulizia dei dati.

La stabilità di entrambi i moduli è ottima: le ultime versioni non hanno mai presentato casi di crash ed eventuali errori sono sempre stati gestiti correttamente e non hanno compromesso il proseguo dell'esecuzione.

Sia le routine che si occupano di scaricare gli articoli che quelle che effettuano lo scraping vero e proprio sono altamente specializzate e rendono quindi il codice difficilmente riutilizzabile. Tuttavia, chiunque volesse sviluppare uno scraper ulteriore, può riutilizzare le parti per la lettura dei parametri e la struttura generica del modulo, organizzata con async e perfetta per effettuare richieste HTTP multiple.

4.2 Opener pipeline

Vediamo ora come si comporta la pipeline di Opener i cui risultati sono sicuramente i più importanti da considerare, visto il ruolo centrale che svolge all'interno dell'architettura di PARLEN

Vediamo innanzitutto i dati relativi al numero di entità che Opener riesce ad estrarre, tenendo conto anche dei tipi e dell'eventuale link a DBpedia.

	Repubblica	**Sole 24 Ore**
Dimensione campione	90	90
Lunghezza media body	4000 caratteri	4280 caratteri
Totale entità riconosciute	2174	1530
Media entità per articolo	24	17
Max entità per articolo	90	54
Articoli senza alcuna entità riconosciuta	0	3 (tra cui un falso positivo)
Entità collegate a DBpedia	1835 (84,4%)	1267 (82,8%)
Entità di tipo Persona	549 (25,3%)	410 (26,9%)
Entità di tipo Organizzazione	828 (38%)	662 (42,3%)
Entità di tipo Luogo Geopolitico	779 (35,8%)	448 (29,3%)
Entità di tipo Luogo (vie, piazze, etc)	17 (0,8%)	6 (0,4%)

Tabella 4.2: Dati numerici sulle entità riconosciute dalla pipeline di Opener

Il dato più utile ed interessante è sicuramente quello relativo alle entità collegate con DBpedia: la percentuale supera in entrambi i casi l'80% e questo è sicuramente positivo. Quando le entità non sono collegate ad una risorsa che le identifica, le successive possibilità di analisi si riducono drasticamente, principalmente perché non si può usufruire dei vantaggi offerti dal progetto Linked Open Data di cui abbiamo precedentemente parlato.

Bisogna comunque tenere presente che i risultati riportati derivano un'analisi effettuata su testi in italiano e non sappiamo se con altre lingue la situazione migliora o peggiora.

Per valutare invece la qualità dei dati, ho preso in considerazione alcuni degli articoli facenti parte del campione ed ho controllato la presenza di falsi negativi e di falsi positivi. Ho tenuto in considerazione inoltre anche della qualità del lavoro svolto dal ned (Named Entity Disambiguation). Per la valutazione di questi risultati mi servirò di un articolo che mette in rilievo sia gli aspetti buoni che quelli cattivi del lavoro svolto da Opener. Utilizziamo l'output del visualizer per facilitare la visualizzazione.


Il direttore del Fondo monetario, Christine Lagarde, è intervenuta ieri a «forte sostegno» della politica monetaria della Banca centrale europea, oggetto nei giorni scorsi di pesanti critiche del ministro delle Finanze tedesco, Wolfgang Schaeuble. Un incontro fra Schaeuble e il presidente della Bce, Mario Draghi, per discutere la questione è fissato per il momento per stasera a Washington, a quanto risulta al Sole 24 Ore. Anche se difficilmente le divergenze potranno essere appianate – il ministro è da tempo contrario alla «eccessiva liquidità» creata dalla misure della Bce e alla politica di bassi tassi d'interesse – il faccia a faccia risponde probabilmente ala necessità di abbassare i toni, che hanno recentemente assunto connotati virulenti da parte dell'establishment e dei media tedeschi. Il direttore dell'Fmi ha detto che la politica monetaria accomodante ha avuto «un ruolo cruciale nella ripresa» e anche i tassi d'interesse negativi (che la Bce ha adottato sui depositi delle banche presso l'istituzione di Francoforte) sono, tutto considerato, un fatto positivo e possono aiutare. «Sosteniamo con forza le decisioni prese dalla Bce – ha detto la signora Lagarde – l'inflazione è molto bassa. La crescita è inferiore al potenziale. Quindi crediamo che una politica monetaria innovativa sia legittima». Il capo dell'Fmi ha ammesso che ci sono possibili effetti collaterali, in particolare sui margini delle banche, e che i tassi negativi non possono continuare per sempre, ma si è chiesta: senza queste decisioni «non ci sarebbero meno crescita, meno credito, meno occupazione?». Dopo l'intervento dai toni molti aspri del ministro tedesco, il presidente della Bundesbank, Jens Weidmann, spesso in dissenso con le scelte della Bce, si è schierato nettamente a difesa dell'indipendenza della banca centrale. Ieri i principali istituti di ricerca economica della Germania, nel loro rapporto di primavera, hanno a loro volta sostenuto che la politica monetaria accomodante della Bce è appropriata e hanno anzi criticato il Governo tedesco per la sua politica fiscale eccessivamente restrittiva, che non fa abbastanza per gli investimenti e per sostenere la crescita. La signora Lagarde ha ripetuto la tesi del Fondo monetario, secondo cui i Paesi che hanno spazio nei bilanci (la Germania ha i conti in pareggio) devono adottare politiche fiscali di sostegno alla crescita. Come Weidmann, il direttore dell'Fmi ha ricordato che è sbagliato pensare solo alle conseguenze dei tassi bassi sui risparmiatori, ma che si devono considerare i benefici per i consumatori e per i lavoratori. Il pareggio di bilancio in Germania, sostengono gli istituti di ricerca, è stato ottenuto anche grazie al risparmio della spesa per interessi. La signora Lagarde ha insistito ancora una volta che il rilancio della crescita, sulla quale crescono i rischi al ribasso, passa da una strategia a tre punte: la politica monetaria non può fare tutto da sola, ma ha bisogno del sostegno della politica di bilancio e delle riforme strutturali. Su quest'ultimo punto, ha proposto che le riforme, concordate dal G-20 due anni fa e che avrebbero dovuto portare una crescita globale addizionale del 2,1% entro il 2018, vengano accelerate al 2016. L'obiettivo è di fatto irraggiungibile, come le prime discussioni nel G-20, iniziate ieri sera, riconoscono. Il caso Grecia sta emergendo come l'altro tema caldo degli incontri. Dopo la sospensione dei colloqui ad Atene la settimana scorsa, la signora Lagarde ha messo nuovamente in dubbio l'impianto dell'accordo raggiunto nel luglio scorso per il terzo salvataggio e che l'Fmi non ha per ora sottoscritto. Il surplus primario dei conti pubblici del 3,5% entro il 2018, può forse essere raggiunto attraverso sforzi "eroici" del popolo greco, ma non può essere mantenuto stabilmente per decenni, ha detto il capo del Fondo. «Il programma deve essere realistico e sostenibile», ha affermato, ripetendo che deve comporsi di due elementi, le riforme da parte di Atene e la ristrutturazione del debito detenuto dai Paesi europei. L'opposizione più dura su questo punto viene dalla Germania, che però insiste che l'Fmi deve partecipare al salvataggio della Grecia. Le due posizioni appaiono al momento inconciliabili, anche se l'obiettivo è di cercare di avvicinarle nei colloqui di questi giorni a Washington.

Figura 4.2: Un output del visualizer

Delle 32 entità riconosciute, nessuna è un falso negativo. Tuttavia c'è una incongruenza: l'entità "Grecia" viene riconosciuta come una organizzazione, anche se poi il link alla risorsa DBPedia è corretto. Il lavoro svolto dal ned è buono: l'entità "Banca Centrale Europea" viene correttamente riconosciuta e collegata a DBpedia anche quando il riferimento testuale è "BCE". Stesso discorso va fatto per "Christine Legarde" e per "Fondo monetario Internazionale": la prima viene riconosciuta anche quando viene

citata solo per mezzo del suo cognome, la seconda viene riconosciuta anche quando viene utilizzata la sigla "Fmi".

Parlando invece di falsi negativi, l'entità "Grecia", già una volta riconosciuta come persona, non viene riconosciuta quando viene citata in un'altra parte del testo. Non vengono inoltre riconosciute "Atene" e "G-20", due entità che ci aspettiamo vengano correttamente individuate.

Bisogna considerare però che il mancato riconoscimento di una entità non sempre è prova del fatto che Opener non sia in grado di riconoscerla mai (vedi entità Grecia). Non conosciamo il motivo di questo comportamento; esso potrebbe verosimilmente essere legato al fatto che Opener, prima di riconoscere una entità nel testo, individua il contesto grammaticale in cui l'entità stessa è citata e cerca di capire se il riferimento è appropriato, sbagliandosi in alcuni casi.

Possiamo comunque considerare i risultati prodotti da Opener assolutamente buoni: lo sporadico mancato riconoscimento di una entità compromette solo lievemente la bontà dei dati estratti, complice anche la presenza di un elevato numero di istanze positive che contribuiscono a dare un significato semantico più che esaustivo ai testi in analisi.

Parlando di performance, sulla nostra macchina virtuale, Opener è in grado di concludere l'analisi con una media di 11,5 secondi ad articolo, utilizzando tutti e 8 i core. Interessante il fatto che se si riduce il numero dei core utilizzati a 6, il risultato è esattamente lo stesso e per iniziare a notare un degrado delle prestazioni, dobbiamo scendere a 4 core: in questo caso la media sale a 15 secondi. Se però si esegue l'analisi di 2 articoli in concorrenza, il tempo medio richiesto sale esattamente al doppio: infatti, non guadagnando nulla in termini di tempo, ho preferito che la pipeline operasse l'analisi degli articoli uno alla volta. Sicuramente, se avessi potuto utilizzare i servizi web di Opener, l'analisi si sarebbe conclusa in minor tempo dato che l'allocazione delle librerie in memoria sarebbe avvenuta una volta sola, all'avvio del server web.

L'utilizzo di risorse è relativamente elevato: durante tutta la fase di analisi, Opener usa tutti i core a disposizione al 100% e circa 750mb di RAM.

Per quello che concerne la stabilità, non ho avuto alcun tipo di problema con la

pipeline di Opener quando ho eseguito questi test. Tuttavia è doveroso ricordare che in sporadici casi che non sono riuscito a distinguere, in fase di sviluppo, mi è capitato che il componente che opera la Named Entity Disambiguation andasse in crash.

4.3 Visualizer

Utilizzando lo stesso campione di 90 articoli per giornale, ho generato i rispettivi 90 files HTML e controllato per ognuno se la visualizzazione delle entità evidenziate risultasse corretta. Di seguito i risultati:

	Repubblica	**Sole 24 Ore**
Dimensione campione	90	90
Files con offset errati	1	2

Tabella 4.3: Risultati del modulo visualizer

L'unico problema riscontrato riguarda il fatto che, in rarissimi casi, soprattutto quando la pulizia dei dati in input non è ottima, la parte che evidenzia le entità è sfalsata di 1 o 2 indici, a dimostrazione del fatto che Opener ha calcolato male gli offset. Nel caso di Repubblica, l'articolo che presenta il problema è anche uno di quelli che aveva avuto dei problemi con lo scraper. In particolare, all'inizio del body compare un carattere speciale, seguito da due \n, il simbolo per andare a capo. Probabilmente questo ha creato problemi con il tokenier di Opener, che ha calcolato male l'offset. Nei due casi del Sole 24 Ore, invece, i dati in input erano apparentemente puliti e non è chiaro quale motivo gli offset di alcune entità risultano errati di 1 indice.

Ad ogni modo, anche in questo caso il verificarsi del problema è cosi raro che possiamo comunque considerare buono il lavoro svolto sia dal modulo da me implementato che dal tokenizer di Opener.

In questo caso, parlare di performance è superfluo: le risorse impiegate dalla procedura sono minime e nonostante l'utilizzo di una chiamata sincrona per la scrittura del file su disco, sulla macchina virtuale lo script genera 90 file in qualche frazione di secondo.

4.4 Rdfer

Questo è il modulo in assoluto più stabile di tutti. Sul nostro campione ma anche su altri dati in input, non hai generato mai un errore ed ha terminato la sua esecuzione in qualche frazione di secondo. Questi risultati sono dovuti al fatto che non esiste alcun punto critico in nessun momento della routine che esegue.

Le uniche incongruenze che possono verificarsi non sono legate all'implementazione dell'rdfer ma ad errori logici verificatisi nella fase precedente della computazione (es. Grecia identificata come organizzazione).

Per verificare la correttezza sintattica del file RDF in output ho utilizzato un linter online[34] che, ancora una volta, non ha individuato alcun errore.

Conclusioni e futuri sviluppi

Abbiamo visto quali sono le possibilità offerte dal Semantic Web e la conseguente necessità di lavorare con dati strutturati e collegati a risorse che li identifichino.

Abbiamo inoltre analizzato ed utilizzato alcuni tra i più importanti tool che operano l'analisi del linguaggio naturale e abbiamo visto quanto sono avanzate le funzionalità che offrono. In particolare, è stato interessante scoprire l'esistenza di algoritmi che operano la disambiguazione, necessità alla quale non avevo inizialmente pensato.

Tra gli altri componenti più funzionali che abbiamo visto, ci sono sicuramente quelli che individuano un sentimento di positività o di negatività all'interno di una frase: nessuno di essi è stato integrato per il momento in PARLEN ma la futura integrazione di un modulo simile sarà sicuramente uno dei prossimi passi. I sistemi per valutare la web reputation sono sempre più richiesti da piccole e grandi realtà e per rendere PARLEN competitivo e sperare in una maggior diffusione del software, sarà assolutamente necessario aggiungere questo tipo di funzionalità.

Sarebbe stato interessante poter integrare Dandelion API, il servizio made in Italy che pareva essere più interessante e funzionale. Tuttavia Opener è riuscito a svolgere un buon lavoro come analizer di PARLEN e l'ontologia generata con i risultati della sua analisi permetterà comunque ai ricercatori dell'Università di Bologna di avere un quadro semantico approfondito degli articoli che forniranno in input al programma.

L'utilizzo di Node.js si è rivelata una scelta vincente: nonostante esso sia prevalentemente pensato per lo sviluppo di API RESTful, la sua natura asincrona, la sua stabilità e la sua facilità d'uso lo hanno reso ideale anche in un contesto in cui i vari moduli sono script standalone e non fanno parte di un Server Web.

L'implementazione è risultata veloce e non ci sono stati quasi mai grossi problemi se non con lo sviluppo di alcune routine che avevano una logica particolarmente complessa, come quelle che operano lo scraping delle pagine di Repubblica.

Dover implementare un'architettura modulare è stato l'aspetto più stimolante, sia perché ciò ha richiesto delle conoscenze preliminari che ho potuto mettere in gioco, sia perché questo garantisce la possibilità di futuri sviluppi, da parte mia o di chiunque vorrà collaborare.

Come oramai sappiamo, infatti, sarà possibile aggiungere il supporto a qualsiasi altro giornale o fonte di dati testuali; la cosa più interessante, però, sarà vedere come lavoreranno altri tool per l'analisi del linguaggio naturale (come Dandelion) sull'output prodotto dagli attuali scraper.

Prendendo spunto dall'architettura dei servizi di Opener, sarebbe comodo in futuro rendere tutti i moduli di PARLEN utilizzabili anche come servizi web, senza tuttavia compromettere l'attuale struttura modulare.

In seguito, sarebbe anche utile sviluppare un'applicazione web che si interfacci con le API dei futuri servizi HTTP di PARLEN: questa ulteriore espansione richiederebbe l'utilizzo di nuove tecnologie client side che possano garantire una buona user experience nonostante i fisiologici tempi d'attesa richiesti dall'analizer utilizzato, qualsiasi esso sarà.

La speranza è quella che PARLEN sarà utilizzato anche in contesti esterni da quello dell'Università di Bologna: il software è in grado di lavorare su qualsiasi tipo di testo, proveniente anche da una fonte diversa da un giornale online. Questa caratteristica lo rende versatile ed utilizzabile per diversi scopi.

Tuttavia, bisogna essere consapevoli che, prima di poter contare in una maggior diffusione del programma, sarebbe necessario espandere il supporto a più giornali, a più tool per l'analisi e soprattutto sarebbe necessaria l'implementazione dell'interfaccia web che lo renderebbe utilizzabile anche da chi non ha troppa dimestichezza con la riga di comando.

In conclusione, il lavoro svolto è più che buono: dopo ripetuti test e bug fix, il software ha raggiunto un ottimo livello di stabilità e gli output prodotti rispecchiano

abbondantemente le aspettative.

Come avevo scritto nelle prime righe di questa tesi, il web è un contenitore immenso di dati, molti dei quali sono testuali. In molti casi, almeno da un punto di vista informatico, essi sono privi di semantica e questo può essere considerato uno spreco. È per questo che considero utile l'aver sviluppato un software che riesce a dare significato a testi che prima non avevano alcuna informazione semantica.

Bibliografia

[1] W3c semantic web activity homepage. https://www.w3.org/2001/sw/.

[2] Archivio - la repubblica.it. http://ricerca.repubblica.it/.

[3] Data - w3c. https://www.w3.org/standards/semanticweb/data.

[4] Archivio storico — il sole 24 ore. http://www.archiviostorico.ilsole24ore.com/.

[5] Rdf - semantic web standards. https://www.w3.org/RDF/.

[6] Owl - semantic web standards. https://www.w3.org/2001/sw/wiki/OWL.

[7] Foaf vocabulary specification. http://xmlns.com/foaf/spec/.

[8] La repubblica.it - news in tempo reale - le notizie e i video di politica, cronaca, economia, sport. http://www.repubblica.it/.

[9] Il sole 24 ore: notizie di economia, finanza, borsa, fisco, cronaca italiana ed esteri. http://www.ilsole24ore.com/.

[10] Apache stanbol - welcome to apache stanbol! https://stanbol.apache.org/.

[11] Celi: Language technology. https://www.celi.it/.

[12] Linguagrid. http://www.linguagrid.org/.

[13] Dandelion api - semantic text analytics as a service. https://dandelion.eu/.

[14] Spaziodati ˜ smart data now! http://www.spaziodati.eu/it/.

[15] plans and pricing — dandelion api. https://dandelion.eu/profile/plans-and-pricing/.

[16] demo — entity extraction: find places, persons, brands, and events in documents and social media — dandelion api. https://dandelion.eu/semantic-text/entity-extraction-demo/.

[17] Welcome to polyglot's documentation! — polyglot 15.10.03 documentation. http://polyglot.readthedocs.io/en/latest/.

[18] Opener · github. https://github.com/opener-project.

[19] The opener project. http://www.opener-project.eu/.

[20] Github - dbpedia-spotlight/dbpedia-spotlight: Dbpedia spotlight is a tool for automatically annotating mentions of dbpedia resources in text. https://github.com/dbpedia-spotlight/dbpedia-spotlight.

[21] Spaziodati ˜ smart data now! http://www.spaziodati.eu/it/#services.

[22] Prodotti: Language technology — celi. https://www.celi.it/it/prodotti/.

[23] Evalita — evaluation of nlp and speech tools for italian. http://www.evalita.it/.

[24] npm. https://www.npmjs.com/.

[25] Github - cheeriojs/cheerio: Fast, flexible, and lean implementation of core jquery designed specifically for the server. https://github.com/cheeriojs/cheerio.

[26] command-line-args. https://www.npmjs.com/package/command-line-args.

[27] html-entities. https://www.npmjs.com/package/html-entities.

[28] request. https://www.npmjs.com/package/request.

[29] Github - buglabs/node-xml2json: Converts xml to json using node-expat. https://github.com/buglabs/node-xml2json.

[30] n3. https://www.npmjs.com/package/n3.

[31] Child process node.js v6.2.2 manual & documentation. https://nodejs.org/api/child_process.html.

[32] Promises/a+ performance hits you should be aware of. http://thanpol.as/javascript/promises-a-performance-hits-you-should-be-aware-of/.

[33] Handlebars.js: Minimal templating on steroids. http://handlebarsjs.com/.

[34] W3c rdf validation results. https://www.w3.org/RDF/Validator/rdfval.

Ringraziamenti

Ringrazio tutta la mia famiglia.

I miei genitori che mi hanno sostenuto economicamente e soprattutto moralmente, che hanno avuto sempre fiducia in me e che continuano a dedicarmi la loro vita intera.

I miei nonni, quelli che ci sono ancora e quelli che non ci sono più, che hanno contribuito alla mia formazione e che mi hanno insegnato i valori più importanti.

I miei zii: due fratelli e una sorella più grandi, che mi hanno donato tante risate e tanti consigli.

I miei amici e compagni di corso: una seconda famiglia con cui ho trascorso gli anni più belli della mia vita e condiviso momenti di gioia e tristezza.

Ringrazio Martina, mi ha sostenuto più di chiunque altro qualsiasi scelta io intraprendessi. Mi ha insegnato tanto e le ho voluto bene.

Ringrazio il dott. Di Iorio. Ha saputo guidarmi perfettamente in questa ed in altre attività universitarie. Mi auguro di lavorarci ancora assieme.

Infine ringrazio Bologna, non serve aggiungere altro.