Atoms in MoK: Text Semantic Annotation and Information Representation

Matteo Delvecchio

May 20, 2016

Alma Mater Studiorum – University of Bologna
via Venezia 52, 47023 Cesena, Italy
matteo.delvecchio4@studio.unibo.it

Contents

1 Introduzione a Mok

1.1 Generalità

Mok (Molecules of Knowledge) è un modello di coordinazione orientato all'auto-organizzazione della conoscenza. L'idea di fondo è che la conoscenza si possa aggregare e diffondere spontaneamente per raggiungere i fruitori, piuttosto che dover essere "cercata"; per tale motivo l'ispirazione di questo modello proviene dal mondo biochimico, dove assistiamo continuamente a fenomeni spontanei di interazione: appoggiandosi quindi sulle astrazioni emerse dalla metafora biochimica, è possibile definire un modello di coordinazione.

1.2 Il modello Mok

Attualmente il modello Mok è costituito dai seguenti componenti:

- **Atomi:** sono la più piccola unità di conoscenza in MoK, e contengono informazioni riguardo la sorgente da cui provengono, il contenuto informativo, un attributo che contiene metadati che possono aiutare nell'interpretazione e un identificatore numerico della loro concentrazione.
- **Molecole:** sono aggregazioni spontanee atomi, le quali vengono prodotte continuamente dai compartimenti. È essenziale che gli atomi che concorrono a tale aggregazione siano semanticamente affini.
- **Compartimenti:** sono depositi attivi di informazioni sorgenti (i semi), atomi, molecole ed enzimi.
- **Semi:** Le sorgenti della conoscenza, che autonomamente generano atomi, inserendoli nei compartimenti
- **Catalizzatori:** sono gli utenti del sistema Mok, che mediante l'utilizzo influenzano il sistema.
- **Enzimi:** rappresentano l'influenza degli utenti (i catalizzatori) nel sistema, che veicola lo sviluppo delle razioni, in modo tale da meglio adempiere alle necessità degli utilizzatori.
- **Reazioni:** sono le leggi che governano l'evoluzione del sistema MoK, che permettono l'aggregazione di atomi in molecole. Esse possono essere di quattro tipi:
  - **Aggregazione:** unione di atomi-molecole affini dal punto di vista semantico.

- **Diffusione:** spostamento di molecole tra compartimenti vicini.
- **Rinforzo:** aumento della concentrazione mediante l'effetto degli enzimi
- **Decadimento:** progressiva diminuzione della concentrazione in funzione del tempo.

2 Caratteristiche degli Atomi Mok

Gli atomi, come già evidenziato, sono la forma di conoscenza di base del sistema. Un atomo Mok viene generato da una fonte di conoscenza e possiede al suo interno un contenuto informativo estrapolato dalla sorgente. Possiede inoltre altre informazioni di arricchimento dell'informazione grezza, atte a specificare la semantica del contenuto informativo estratto. Infine, per mantenere la relazione con la fonte che lo ha generato, un atomo contiene anche un identificativo della sorgente.
Un atomo dunque è una tripletta:
atom(src,val,attr)c

- *src* identifica la sorgente dell'informazione.
- *val* è il contenuto informativo generato
- *attr* è il contenuto informativo aggiuntivo che definisce meglio l'ontologia dell'informazione, assegnando sostanzialmente un dominio
- *c* è la concentrazione dell'atomo: indica il numero di atomi identici presenti nel sistema (dove l'identità è identificata dalla tripletta *src,val,attr*).

3 Obiettivi del Progetto

L'obiettivo primario di questo progetto è interrogarsi su quale sia una rappresentazione efficace per modellare l'informazione in un sistema MoK, individuando le possibili annotazioni semantiche d'ausilio all'auto-organizzazione e definendo in particolare un'ontologia consistente per gli atomi: questo elaborato pone infatti il suo focus sull'atomo.
Secondariamente è scopo di questo lavoro mostrare tecniche che permettano di estrarre consocenza a partire da documenti non strutturati.

4 Progetto

In questa sezione verrà descritta in linguaggio naturale e formale l'idea di fondo che ha guidato il progetto, e saranno messe in luce alcune considerazioni e scelte progettuali.

4.1 Feature richieste

L'applicativo deve fornire la possibilità di generare diverse tipologie di atomo, partendo da un documento non strutturato, mettendo in rilievo il funzionamento di tale processo. Il progetto deve evidenziare in particolare le caratteristiche degli atomi, i suoi elementi costituitivi, e soprattutto la tassonomia in cui è possibile strutturare gli atomi,

4.2 Scelta del Tool

Le esigenze scaturite dalle feature hanno reso necessaria la scelta di un tool che si occuppasse di *data-mining* dei sorgenti. Le caratteristiche richieste sono:

- Il tool deve permettere di estrarre informazioni appartenti ad un certo dominio: facendo ciò il frammento di informazione estratto è arricchito dal dominio stesso e dunque può assumere una semantica specifica.
- Possibilmente il tool deve produrre dati in un formato compatibile con java.
- Il tool deve essere supportato.
- (Possibilmente) il tool deve essere di facile utilizzo/integrazione.

La scelta è ricaduta su ***OpenNLP***, poichè, oltre a rispondere alle caratteristiche sopra citate, è largamente diffuso, ed offre un corpus di documentazione ed esempi notevole: ciò si traduce inoltre con la presenza di molti dataset già disponibili, i quali coprono le semantiche più comuni.

4.3 Modello

Iniziamo a descrivere le scelte di rappresentazione dell'informazione, della sua notazione semantica e dunque della struttura specifica degli atomi MoK, e la relativa ontologia.
Prima di definire formalmente l'architettura tuttavia , è rilevante avere una panoramica che metta in evidenza l'iter di caratterizzazione degli atomi Mok:

Interfaccia IAtom

```
package interfaces;

import java.net.URI;




public interface IAtom {

  URI getSource();    //returns the source URI
  IAttribute getAttribute();  //returns the semantic annotation string
  IValue getValue(); //returns the value related to the attribute
  int getConcentration(); //returrsn the concentration of the atom

  void setConcentration(int concentration);//sets the concentration at the specific value
  void incConcentration();//increments the concentration parameter by 1
  void decConcentration();// decrements the concentration parameter by 1
  boolean isEqual(IAtom a);//checks if two atoms are the same.
  String getDefaultRep();//gives a default (textual) representation of the atom


}
```

Dal corpo dell'interfaccia sono già inferibili alcune scelte di modellazione:

- La sorgente dell'informazione è rappresentata da un URI, in modo da identificare in maniera univoca la fonte stessa.
- L' *attr* è stato definito come nuovo tipo di dato, identificato dall'interfaccia IAttribute e definito nella specifica implementazione assegnata nella demo.

Interfaccia IAttribute

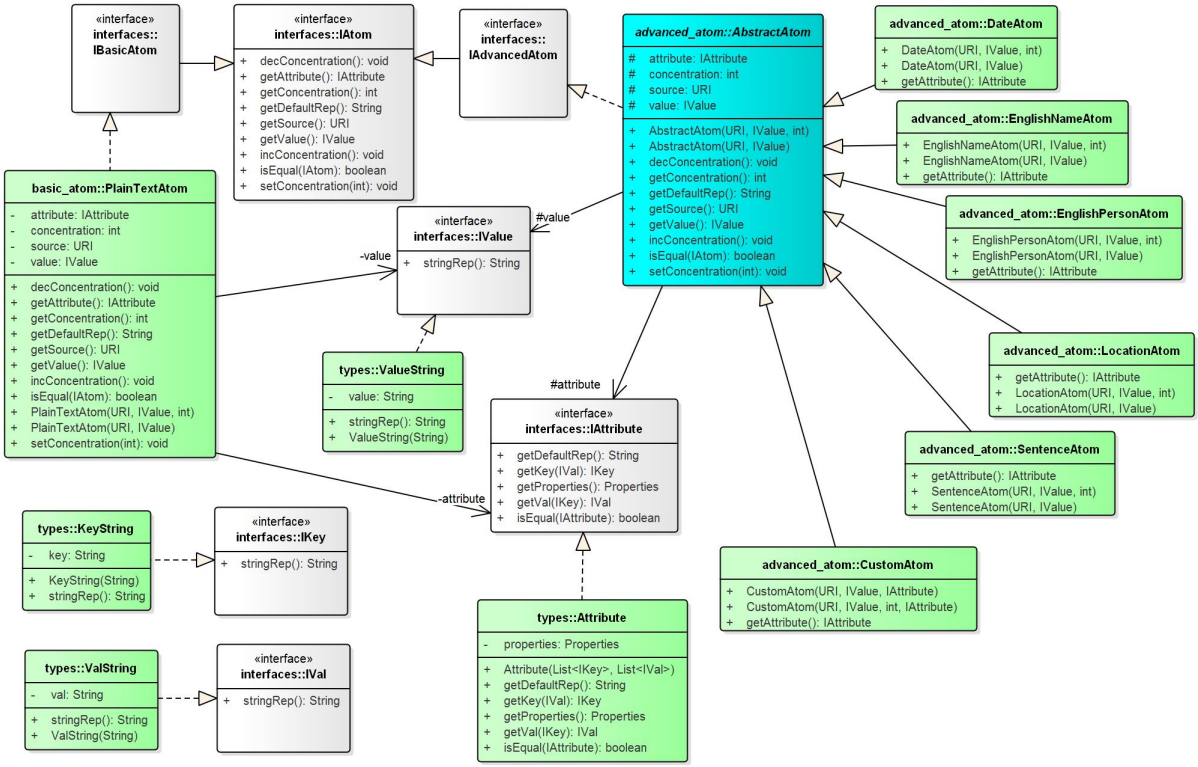
```
package interfaces;


public interface IAttribute {

  //returns the value associated to the key
  public IVal getVal(IKey key);

  //finds out the key related to the value
  public IKey getKey(IVal val);

  //checks if two attribute are equals
  public boolean isEqual(IAttribute a);

    //gives the Properties of the attribute
    public Properties getProperties();

  //gets a textual representation
  public String getDefaultRep();
}
```

Sostanzialmente un attributo è una collezione di coppie *chiave-valore*. Questo tipo di modellazione è voluto per premettere più tag semantici per un singolo

atomo: immaginiamo ad esempio di aver estratto da una sorgente il termine "cat" in riferimento ad un gatto. Il tag semantico che un utente associa a questo termine molto probabilmente sarebbe "animal", tuttavia è subito intuibile che questo arricchimento semantico appare abbastanza generico e debole. Da qui si avverte l'esigenza di generare una sequenza di tag che approfondiscono la semantica a più livelli (in modo da poter rendere anche un'eventuale ricerca per attributo il più flessibile possibile). Ricollegandoci all'esempio sopra proposto, un ulteriore arricchimento potrebbe essere il tag "feline", che in qualche modo rappresenta un livello semantico più specifico rispetto al semplice "animal". Ancora, è possibile aggiungere il tag "english" per evidenziare un altro aspetto semantico dell'informazione "cat", ossia la lingua del vocabolo. L'esigenza di avere una coppia *chiave-valore* anzichè una semplice lista di tag, è dettata da un altra esigenza emersa: se, ad esempio, un atomo possiede un tag "italian", è lampante che tale tag è intrinsecamente ambiguo; potrebbe riferirsi ad un tipo di cucina, alla nazionalità di un individuo, alla lingua, o a tutta un'altra varietà di possibili semantiche interne. Per eliminare questa problematica, si è scelto di adottare appunto questa struttura dati. Nell'esempio proposto il tag diverrebbe *(language, italian)*.

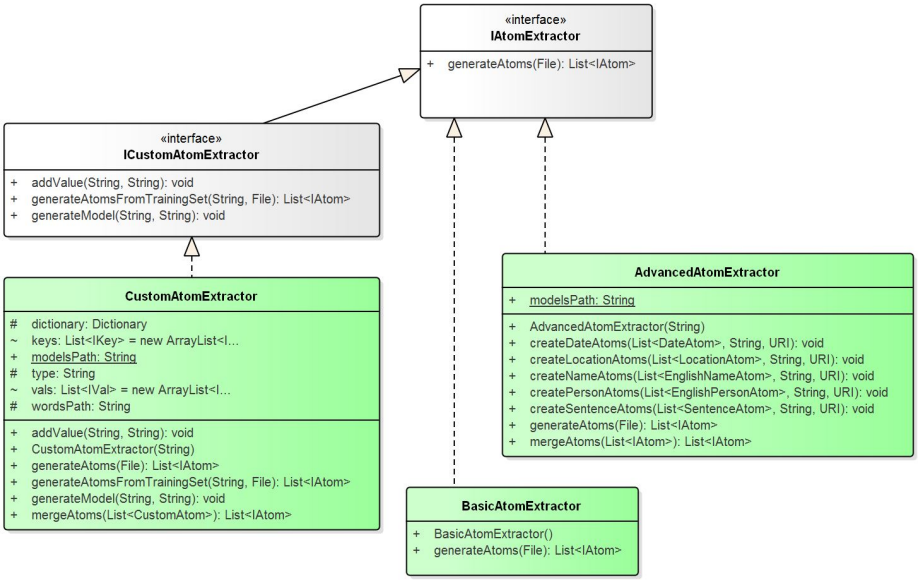
- Il contenuto informativo vero e proprio (il *val*) è associato al nuovo tipo di dato *IValue*: essa nella concreta e specifica implementazione della demo che fondamentalmente è riconducibile ad una stringa, tuttavia ciò è estendibile in sviluppi futuri implementando una classe ad hoc.

4.4 Architettura e tassonomia degli Atomi



Il diagramma strutturale mostra come è stato modellato il dominio degli Atomi in questa esplorazione del modello MOK; la radice di tale gerarchia è l'interfaccia *IAtom*. Da essa abbiamo due diramazioni principali, ossia *IBasicAtom*, che rappresenta la tipologia più generale di atomi, e *IAdvacedAtom*, che rappresenta una granularità più fine nella rappresentazione delle informazioni.
Nella concreta realizzazione, abbiamo la classe *PlainTextAtom* come implementazione di *IBasicAtom*: si considera infatti l'atomo contenente l'intero corpo del documento sorgente come l'atomo più generico e grossolano possibile. *IAdvancedAtom* invece, è implementato da più tipologie di atomi differenti, le quali però sono fattorizzate dalla classe astratta *AbstractAtom*, che mette in comune gli aspetti generali di ogni atomo avanzato. Di particolare rilevanza è il *CustomAtom*, il quale rappresenta il punto di accesso a tutte le altre categorie di atomi non prese in considerazione dalle classi di atomi buit-in.
Dal diagramma emergono anche le già citate caratteristiche interne di un atomo, attraverso le interfacce *IValue* e *IAttribute*, assieme alle loro implementazioni specifiche per questa demo.

4.5 Struttura gerarchica dei generatori/estrattori di Atomi



Per quanto riguarda gli estrattori di conoscenza (da qui in avanti chiamati indisitntamente anche con il nome di generatori, in quanto, oltre ad una parte di estrazione di conoscenza, sono costituiti anche dalla parte di generazioni degli atomi), abbiamo fondamentalmente una dicotomia di entità distinte, *BasicAtomExtractor* e *AdvancedAtomExtractor*, che riflettono esattamente la ripartizione di atomi vista nella sezione precedente.

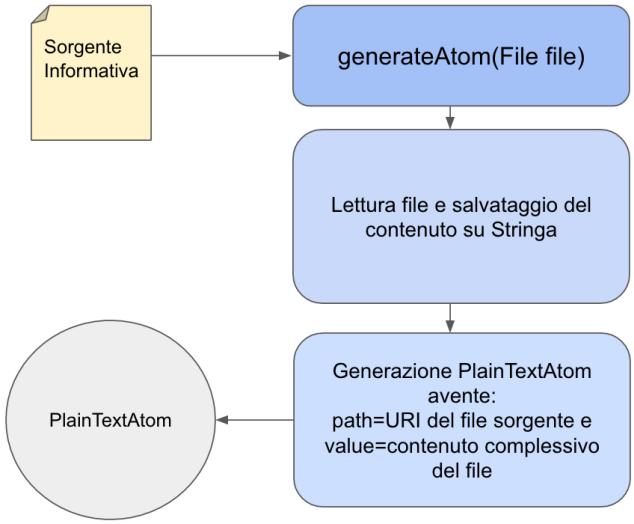


Figure 1: BasicAtomExtractor

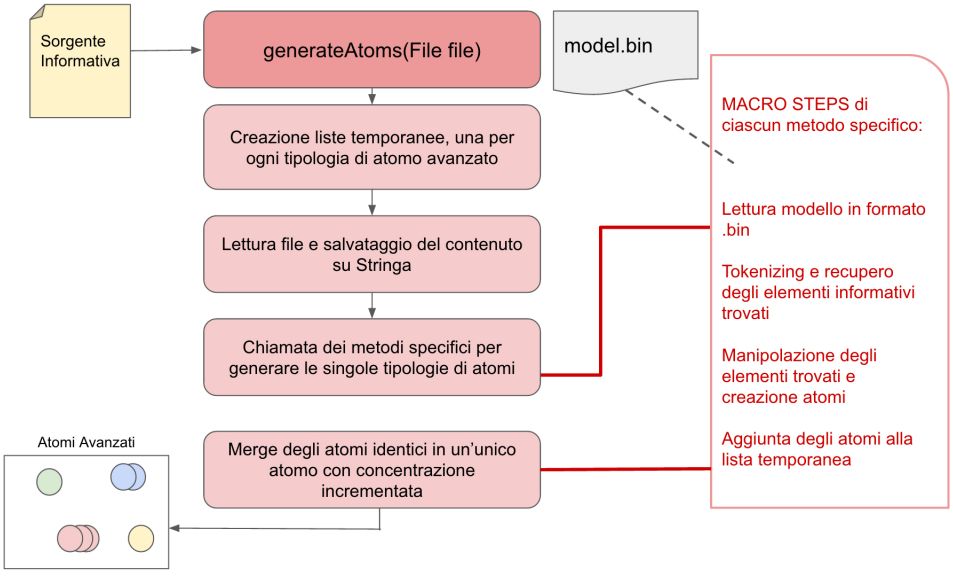


Figure 2: AdvancedAtomExtractor

I due generatori sono tuttavia affiancati in maniera sperimentale da una terza tipologia di generatore, il *CustomAtomGenerator*, il quale permette appunto di generare atomi su definizione dell'utente. Per ottenere questo risultato, l'utente ha due possibili strade:

1. Utilizzo di dizionari. Questa soluzione è più basilare e nasce dall'esigenza di dover generare nuove tipologie di atomi in maniera agevole; OpenNLP mette infatti a disposizione la classe *DictionaryNameFinder* , con la quale è possibile, dato un *Dictionary* rintracciare le parole presenti in un documento corispondenti a quelle presenti nel dizionario definito. Si veda la sezione di tutorial per ulteriori informazioni.

2. Utilizzo di un dataset compatibile con il *NameFinder* di OpenNLP. Questo tipo di soluzione è più sofisticata e congruente con le strategie utilizzate dagli altri estrattori: è possibile infatti caricare un modello (in formato .bin) e generare gli atomi nella medesima procedura utilizzata nell' *AdvancedAtomExtractor*. Il metodo proposto permette all'utente di utlizzare modelli diversi da quelli già built-in, tuttavia ciò non esclude la possibilità di usare gli stessi modelli predefiniti.

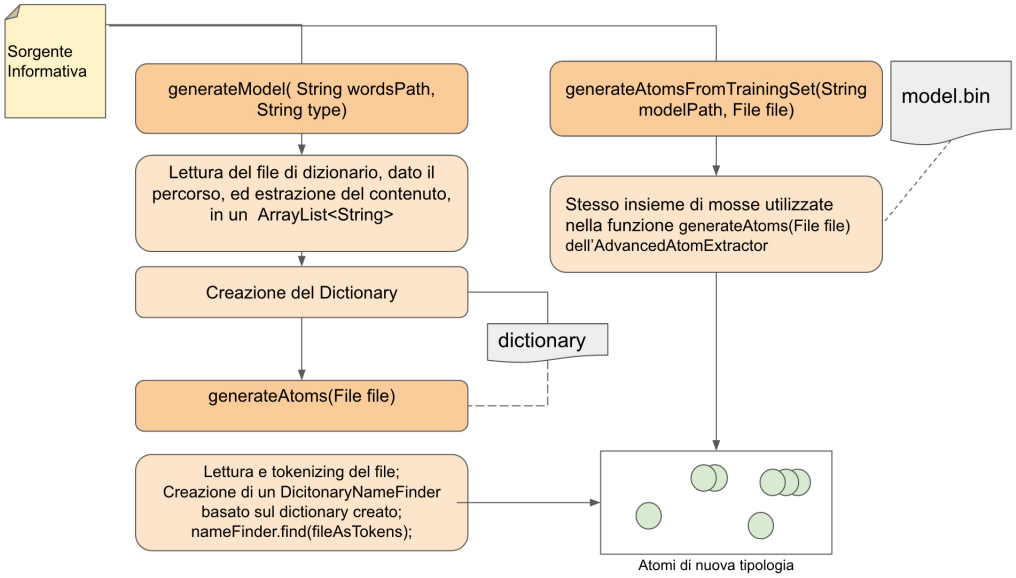


Figure 3: CustomAtomExtractor

4.5.1 Modello per la creazione del dizionario

Per realizzare l'oggetto *Dictionary* offerto dalle API di OpenNLP è necessario definire un file di testo (.txt) in cui viene inserito un vocabolo in ogni riga. Essendo il *Dictionary* utilizzato in modalità case sensitive, si richiede di inserire per ogni parola la versione con iniziale minuscola e maiuscola.

Sezione di un file di training

```
Pasta
pasta
Spaghetti
spaghetti
Hot-Dog
hot-dog
```

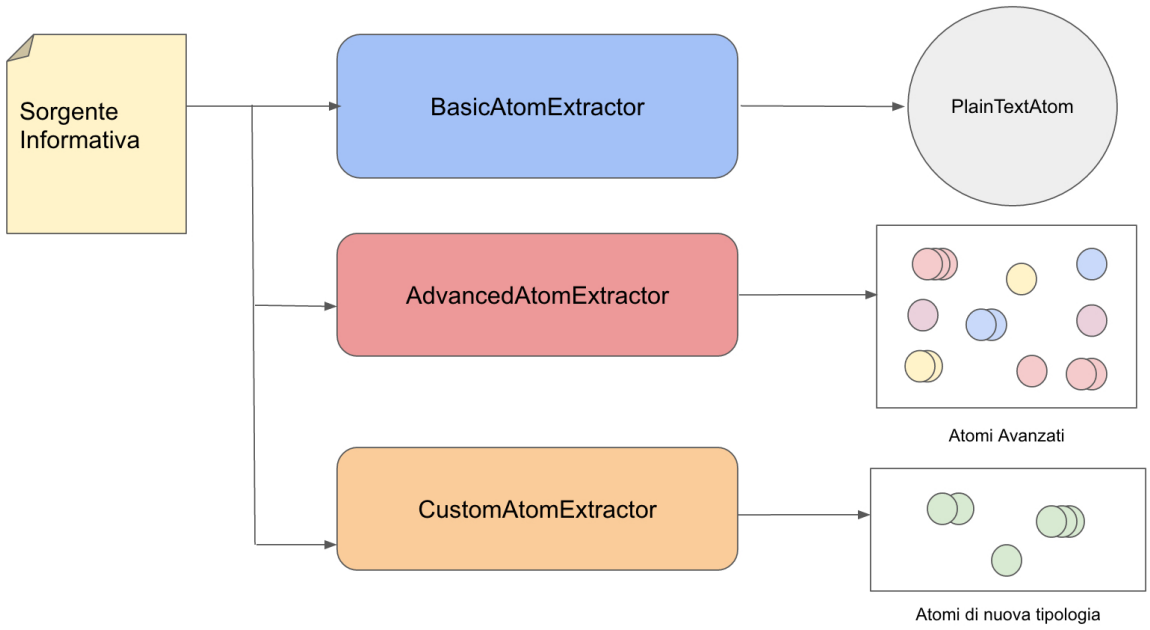


Figure 4: Panoramica riassuntiva degli estrattori

Nota: le fasi di Tokenizing sono state realizzate utilizzando l'oggetto *TokenizerME(tokenModel)* di OpenNLP. Per ulteriori informazioni si veda **https://opennlp.apache.org/documentation/1.6.0/manual/opennlp.html#tools.tokenizer.api**

4.6 Appendice - CustomAtomGenerator: utilizzo di dataset generati dall'utente e generazione di un nuovo modello

L'uso di un dataset definito dall'utente deve avere certe caratteristiche:

- Deve contenere almeno 15000 istanze (frasi contententi la parola sulla quale dovrà essere eseguita la procedura di learning).
- La sintassi che definisce le parole deve essere nella forma <**START:tag-Semantico**> **parola-da-considerare** <**END**>.
- L'estensione del file di training deve essere **.train**.
- Dovrebbero essere inserite istanze con diverse strutture di sintassi grammaticale, in modo da coprire un numero di casistiche più ampio.

Esempio:

Sezione di un dizionario

```
Last Thursday afternoon the <START:thing> soup <END>  spoke to the <START:thing> meat <END> .
The famous <START:thing> potato <END>  bought a foreign <START:thing> key <END> .
A <START:thing> engineer <END>  was authorized to sell the friendly <START:thing> hat <END> .
```

Questo estratto proviene da un file che rispetta i primi tre punti sopra citati; il quarto punto non è stato completamente soddisfatto in quanto le istanze non sono state costruite "a mano" ma generate con un programma java costruito ad hoc, in cui la struttura sintattica delle frasi non possiede più di tanta flessibilità.
Utilizzando OpenNLP da linea di comando, possiamo generare il binario a partire dal dataset di addestramento. Per utilizzare i tool, dopo aver scaricato e decompresso l'archivio (la verisone utilizzata è la 1.6.0), è necessario portarsi dentro la cartella bin, ed utilizzare il comando **./opennlp TokenNameFinderTrainer**. In questo modo è possibile visionare i parametri messi a disposizione dal tool.

```
Arguments description:
        -factory factoryName
                A sub-class of TokenNameFinderFactory
        -resources resourcesDir
                The resources directory
        -type modelType
                The type of the token name finder model
        -featuregen featuregenFile
                The feature generator descriptor file
        -nameTypes types
                name types to use for training
        -sequenceCodec codec
                sequence codec used to code name spans
        -params paramsFile
                training parameters file.
        -lang language
                language which is being processed.
        -model modelFile
                output model file.
        -data sampleData
                data to be used, usually a file name.
        -encoding charsetName
                encoding for reading and writing text, if absent the system defa
ult is used.
```

Gli argomenti necessari e sufficienti per una generazione basilare sono *-model*, *-data* e *-lang*.
Sintassi:
./opennlp TokenNameFinderTrainer -model <file.bin con il relativo path> -lang <lingua> -data <file.train con il relativo path>

Esempio

```
$ ./opennlp TokenNameFinderTrainer
-model /Users/jacob/Desktop/mok/en-ner-pippo.bin
-lang en
-data /Users/jacob/Desktop/mokSources/en-ner-pippo.train
```

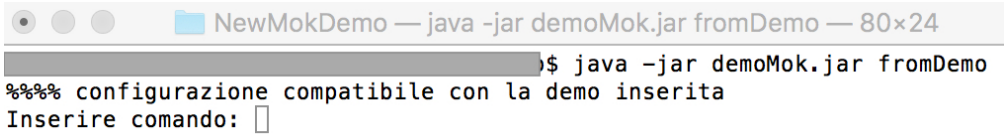
Nota: il comando mostrato, per problemi di formattazione nel paper, è stato suddiviso in più righe, ma è da considerarsi come un'unica riga ovviamente.

Se la procedura è andata a buon fine, verrà mostrato in output il messaggio **Writing name finder model ... done**.

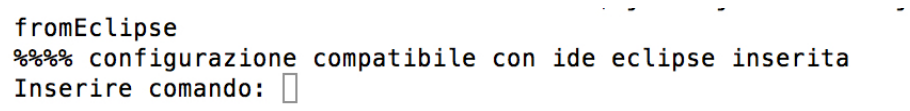
5 Tutorial ed esempi di utilizzo

Questa sezione offre una panoramica concreta delle funzionalità messe a disposizione, e permette di osservare, tramite esempi di utilizzo, alcune scelte implementative.
Per avviare la demo, è necessario inserire un argomento, il quale funge da configuratore per i path ove reperire file, modelli e dizionari:

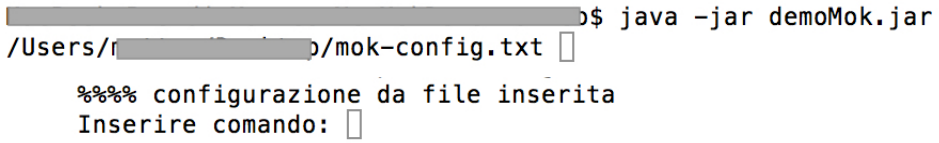
- *fromDemo* - per lanciare il programma con i percorsi prestabiliti della demo.



- *fromEclipse* - per lanciare il programma con i percorsi prestabiliti dell'IDE di sviluppo.



- *<percorso del file di configurazione>* - per lanciare il programma con i percorsi scelti dall'utente.



Il file di configurazione deve avere un'estensione .txt ed essere costituito nel seguente modo:

mok-config.txt

```
FILES
<path della cartella in cui sono contenuti i file>
MODELS
<path della cartella in cui sono contenuti i modelli binari di OpenNLP>
DICTIONARIES
<path della cartella in cui sono contenuti i file sorgenti da cui costituire i dizionari>
```

La demo offre come interfaccia verso l'utente una shell in cui impartire i comandi in maniera testuale. digitando la keyword *help* o una stringa non compresa nell'elenco dei comandi è possibile visualizzare l'effettivo corpus dei comandi messi a disposizione:

```
Inserire comando: help
[comando scritto:]help
Comandi validi:
help                          per avere info sui comandi disponibili e il loro uso
basicAtomGenerator            per utilizzare il generatore di base
advancedAtomGenerator         per utilizzare il generatore avanzato
completeAtomGenerator         per utilizzare entrambi i generatori
customAtomGenerator           per utilizzare il generatore custom (sperimentale)
exit                          per uscire
Inserire comando:
```

Vediamo inoltre i file di prova utilizati nella demo:

moreContents.txt

```
Jason White instead is born on  November 30 1991.
On April 1982 there was a flower.
1955 is a particular year for Marty McFly.
James Tiberius Kirk is a Captain.
James Tiberius Kirk is a Captain.
California is a part of America;
Italy and New Orleans are countries.
I will survive.

Jason White instead is born on  November 30 1991.
On April 1982 there was a flower.
The 1955 is a particular year for Marty McFly.
James Tiberius Kirk is a Captain.
California is a part of America;
Italy and New Orleans are countries.
I will survive.
My favorite dish is pizza , but also spaghetti are good.
this pizza is the best!
```

newfoodtrain.txt

```
Italian food is world-wide famous.
Many restaurants that serves pizza or pasta have been opened all over the world.
There are various traditional plates in Italy but the most common are of course
pizza and pasta.
But not many tourists actually know about the suppli, it is an egg formed food
made of rice with mozzarella and rice with  tomato soup , all of it fried,
it is mostly prepared (and eaten) in mid-Italy.
There are also a lot of famous cheeses in Italy, such as the gorgonzola and the
mozzarella.
The traditional food in Italy also depends from where you live; in south Italy
they are famous for their sweets, such as the  cassata and the mostaccioli,
two very sweet foods.
Instead, up north they're famous for their christmas foods, such as the Panettone
or the Pandoro similiar to cakes but very dry and with caster sugar .
My pizza is better;
```

5.1 basicAtomGenerator

Digitando la keyword *basicAtomGenerator*, verrà richiesto il nome del file sorgente. In questo esempio utilizziamo *moreContents.txt*; per quanto riguarda il percorso di tale file, esso è determinato dalla configurazione inserita al lancio.

```
%%%% configurazione compatibile con la demo inserita
Inserire comando: basicAtomGenerator
[comando scritto:]basicAtomGenerator
inserire nome del file da cui estrarre atomi, preceduto da /
esempio: /moreContents.txt
/moreContents.txt
BasicAtomExtractor Intialized succesfully
```

Il risultato è un *PlainTextAtom*:

```
Atom's Generation terminated
-----------ATOM INFO:-----------
TYPE OF ATOM:
basic_atom.PlainTextAtom

SOURCE:
file:/Users/[redacted]k/NewMokDemo/src/Files/moreContents.txt

ATTRIBUTE:
[ Type ---> Plain_Text ]

CONTENT OF THE ATOM:
Jason White instead is born on  November 30 1991.
On April 1982 there was a flower.
1955 is a particular year for Marty McFly.
James Tiberius Kirk is a Captain.
James Tiberius Kirk is a Captain.
California is a part of America;
Italy and New Orleans are countries.
I will survive.

Jason White instead is born on  November 30 1991.
On April 1982 there was a flower.
The 1955 is a particular year for Marty McFly.
James Tiberius Kirk is a Captain.
California is a part of America;
Italy and New Orleans are countries.
I will survive.
My favorite dish is pizza , but also spaghetti are good.
this pizza is the best!
CONCENTRATION:
1
```

5.2 advancedAtomGenerator

Digitando la keyword *advancedAtomGenerator*, otteniamo, come nel caso precedente, la richiesta del nome del file che fungerà da sogente di informazione.

```
Inserire comando: advancedAtomGenerator
[comando scritto:]advancedAtomGenerator
inserire nome del file da cui estrarre atomi, preceduto da /
esempio: /moreContents.txt
/moreContents.txt
AdvancedAtomExtractor Intialized succesfully
```

Utilizziamo nuovamente *moreContents.txt*, e osserviamo l'output.
(Nota: nel tutorial verranno mostrate solo alcune parti dell'output, ritenute più rilevanti; per controllare l'intero esempio, basta seguire la medesima procedura presentata nel paper).

```
AdvancedAtomExtractor Intialized succesfully
-----------ATOM INFO:-----------
TYPE OF ATOM:
advanced_atom.EnglishNameAtom

SOURCE:
file:/                                        /moreContents.txt

ATTRIBUTE:
[ Type ---> Name ]
[ Language ---> English ]

CONTENT OF THE ATOM:
Jason
CONCENTRATION:
2
```

Figure 5: Estratto di output: generazione Atomo di tipo Name

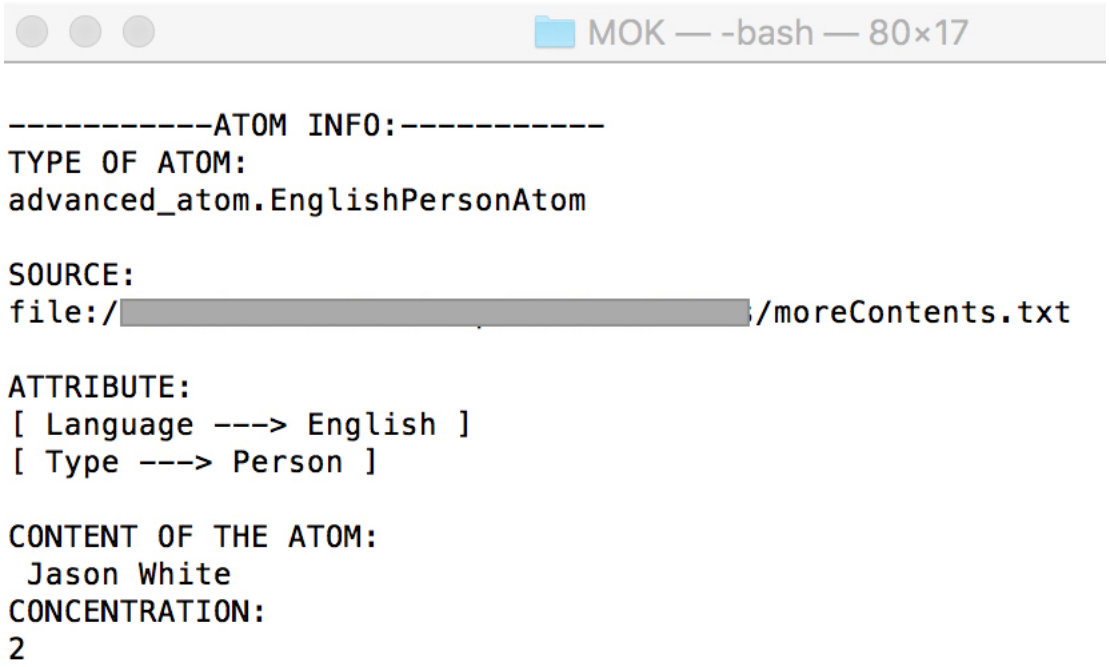


Figure 6: Estratto di output: generazione Atomo di tipo Person

I due frammenti di output proposti mostrano come una stessa minima informazione faccia parte di due tipologie differneti di atomi, mettendo in luce la capacita di OpenNPL di individuare strutture sintattiche di diversa natura. ciò è evidente anche negli esempi sottostanti:

```
----------ATOM INFO:----------
TYPE OF ATOM:
advanced_atom.EnglishPersonAtom

SOURCE:
file:/                                        /moreContents.txt

ATTRIBUTE:
[ Language ---> English ]
[ Type ---> Person ]

CONTENT OF THE ATOM:
 James Tiberius Kirk
CONCENTRATION:
3
```

Figure 7: Estratto di output: generazione Atomo di tipo Person, appartenente ad un individuo il cui nome è complesso

```
----------ATOM INFO:----------
TYPE OF ATOM:
advanced_atom.DateAtom

SOURCE:
file:/                                        /moreContents.txt

ATTRIBUTE:
[ Type ---> Date ]

CONTENT OF THE ATOM:
 November 30 1991
CONCENTRATION:
2
```

Figure 8: Estratto di output: generazione Atomo di tipo Date

5.3 completeAtomGenerator

Molto brevemente, digitando il comando *customAtomGenerator* otteniamo la procedura che unifica le due operazioni precedenti; inserendo il percorso del file sorgente si ottiene sia l'atomo base che gli atomi avanzati associati.

5.4 customAtomGenerator

```
Inserire comando: customAtomGenerator
[comando scritto:]customAtomGenerator
-----
Questa è una demo che mostra la creazione di una tipologia custom di atomi.
-----
Di default il customAtomGenerator funziona con un sistema semplificato basato su
i dizionari, ma è stata aggiunta la possibilità di esplicitare un traing set nel
 formato standard di openNLP (.bin).
PER UTILIZZARE LA FUNZIONE AVANZATA DIGITARE  train , PER USARE IL PROCEDIMENTO
SEMPLIFICATO DIGITARE UN'ALTRA STRINGA
```

5.4.1 Procedimento classico: utilizzo delle funzionalità avanzate

Digitando il comando *train* possiamo utilizzare il medesimo procedimento utilizzato per creare atomi avanzati, ma in questo caso tale procedura è applicabile a modelli differenti da quelli built-it della demo: ciò permette di utlizzare modelli generati dall'utente e definire nuove tipologie di atomi. In questa demo creeremo atomi di tipo *thing*.
Come primo passo, inseriamo il percorso del modello di training:

```
train
inserire nome modello di training, preceduto da /
(esempio--> /en-ner-thing.bin):
/en-ner-thing.bin
```

La procedura guidata richiede anche che vengano inserti i tag semantici degli atomi che siamo in procinto di creare. Il primo campo inserito rappresenta il valore che sarà associato alla chiave *type*, e pertanto è obbligatorio. Si possono inoltre inserire altre coppie chiave-valore, mentre con il comando *exit*, digitato al posto di una chiave, si interrompe tale inserimento.
(In questo tutorial inseriamo la coppia (language, eng))

```
CustomAtomExtractor Intialized succesfully
inserire tipologia principale della semantica dell'atomo
NB almeno un tag è obbligatorio
(esempi--> thing ; food ; name ; date ; city )
thing
inserire altri tag semantici sotto forma di chiave valore. exit per uscite
digitare la chiave (es. language):
language
digitare il valore corrispondente alla chiave (es. it):
eng
digitare la chiave (es. language):
exit
```

```
inserire nome del file da cui estrarre atomi, preceduto da /
esempio: /moreContents.txt
/moreContents.txt
```

MOK — java -jar MOKAtomGenerator.jar — 80×33

```
----------ATOM INFO:----------
TYPE OF ATOM:
advanced_atom.CustomAtom

SOURCE:
file:/Users/matteo/Desktop/MOK/src/Files/moreContents.txt

ATTRIBUTE:
[ language ---> eng ]
[ Type ---> thing ]

CONTENT OF THE ATOM:
flower
CONCENTRATION:
2

$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
----------ATOM INFO:----------
TYPE OF ATOM:
advanced_atom.CustomAtom

SOURCE:
file:/Users/matteo/Desktop/MOK/src/Files/moreContents.txt

ATTRIBUTE:
[ language ---> eng ]
[ Type ---> thing ]

CONTENT OF THE ATOM:
countries
CONCENTRATION:
2
```

Si noti la struttura dell'output soprastante: il campo *ATTRIBUTE* in particolare contiene le coppie digitate. Nota: nella Sez. 4.6 è stato messa in luce la problematica di utilizzare un generatore di frasi casuale. Avendo poca flessibilità nella struttura sintattica delle frasi si ottiene una precisione leggermente inferiore in fase di riconoscimento. Ciò è comunque compensabile se si utilizzano training set realizzati con più accuratezza.

5.4.2 Procedimento semplificato: utilizzo di dizionario dei termini

Digitando una stringa diversa da *train* viene attivata la procedura semplificata (descritta nella Sez. 4.5); utiliziamo in questo esempio un dizionario contenente alcuni cibi.

```
Inserire comando: customAtomGenerator
[comando scritto:]customAtomGenerator
-----
Questa è una demo che mostra la creazione di una tipologia custom di atomi.
-----
Di default il customAtomGenerator funziona con un sistema semplificato basato su
i dizionari, ma è stata aggiunta la possibilità di esplicitare un traing set nel
 formato standard di openNLP (.bin).
PER UTILIZZARE LA FUNZIONE AVANZATA DIGITARE  train , PER USARE IL PROCEDIMENTO
SEMPLIFICATO DIGITARE UN'ALTRA STRINGA
-----
other
la procedura è guidata e offre un tutorial (essa è tuttavia ignorabile)
se si segue la procedura proposta, si otteranno degli atomi che contengono cibi
e pietanze.
-----
nella demo (o nei sorgenti eclipse) è presente il sorgente dictionary_name_finde
r_food_source.txt:
da esso costruiamo il dizionario, per cui procederemo alla creazione di atomi cu
stom che riconoscono i cibi
[NB--> Si può utilizzare anche un dizionario diverso da quello proposto nel tuto
rial.]
PER CONTINUARE DIGITARE next , PER USCIRE DIGITARE UN'ALTRA STRINGA
====>>
```

Come si può notare in figura, viene utilizzato lo stesso schema guidato per inserire gli elementi del campo *ATTRIBUTE*. Ora inseriamo il nome del sorgente, ossia *newfoodtrain.txt* (ma si può utilizzare tranquillamente anche *moreContents.txt*) per confrontare il risultato con gli output delle altre funzioni.

```
next
inserire nome del file contenente i lemmi per il dizionario, preceduto da /
[Utilizzare un dizionario a piacere se non si vuole seguire la procedura guidata.]
(esempio--> /dictionary_name_finder_food_source.txt:)
/dictionary_name_finder_food_source.txt
CustomAtomExtractor Intialized succesfully
inserire tipologia principale della semantica dell'atomo
almeno un tag è obbligatorio
(esempi--> food ; name ; date ; city )
food
inserire altri tag semantici sotto forma di chiave valore. exit per uscite
digitare la chiave (es. language):
language
digitare il valore corrispondente alla chiave (es. it):
eng
digitare la chiave (es. language):
exit

dopo la generazione del modello si eseguirà una ricerca di prova degli atomi del nuovo tipo
digitare next per continuare:
next
inserire percorso completo del file da cui estrarre atomi, preceduto da /
esempio: /newfoodtrain.txt
/newfoodtrain.txt
//=======================
categoria del dizionario:food
//=======================
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
----------ATOM INFO:----------
TYPE OF ATOM:
advanced_atom.CustomAtom

SOURCE:
file:/Users/matteo/Desktop/varie%20mok/NewMokDemo/src/Files/newfoodtrain.txt

ATTRIBUTE:
[ Type ---> food ]
[ language ---> eng ]

CONTENT OF THE ATOM:
pizza
CONCENTRATION:
3

$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
----------ATOM INFO:----------
TYPE OF ATOM:
advanced_atom.CustomAtom

SOURCE:
file:/Users/matteo/Desktop/varie%20mok/NewMokDemo/src/Files/newfoodtrain.txt

ATTRIBUTE:
[ Type ---> food ]
[ language ---> eng ]

CONTENT OF THE ATOM:
pasta
CONCENTRATION:
2
```

6 Appendice: Guida alle API utilizzate

L'appendice qui proposta serve ad elencare le principali API java fornite da OpenNLP in modo da poter replicare il procedimento. Le due classi che sfruttano tali api sono *CustomAtomExtractor* e *AdvancedAtomExtractor*.
Nel caso di *AdvancedAtomExtractor*, la procedura da richiamare è la seguente:

1. **generateAtoms(File file)** : invocando tale metodo, il quale ritorna una List<IAtom>, viene estratto il contenuto e il path del file, e richiamati i metodi specifici di generazione.

2. I metodi chiamati sono:

   createSentenceAtoms(List<SentenceAtom> sentenceAtoms, String content, URI path);

   createNameAtoms(List<EnglishNameAtom> englishNameAtoms, String content, URI path);

   createPersonAtoms(List<EnglishPersonAtom> englishPersonAtoms, String content, URI path);

   createDateAtoms(List<DateAtom> dateAtoms, String content, URI path);

   createLocationAtoms(List<LocationAtom> locationAtoms, String content, URI path);

   Tali metodi non hanno tipo di ritorno ed eseguono manipolazioni del contenuto informativo **content** al fine di estrarre conoscienza e, insieme all'informazione contenuta nel **path**, di generare atomi specifici, che vengono depositati nelle liste specifiche, passate come argomento alle funzioni stesse.

3. All'interno di ciascuno dei metodi presenti, vengono utilizzati i costrutti di OpenNLP;

   a) **FileInputStream(String modelPath)**: con questo metodo creiamo un FileInputStream assegnandolo ad una variabile (esempio, InputStream is = new FileInputStream(modelsPath+"/en-ner-location.bin")).

   b) **TokenNameFinderModel model = new TokenNameFinderModel(is)**; *TonkenNameFinderModel* è un oggetto OpenNLP (il cui riferimento è

opennlp.tools.namefind.TokenNameFinderModel), che consente di applicare il modello di estrazione.

c) Viene eseguita la stessa istruzione proposta nel blocco a), con la differenza che l'argomento non è più un riferimento al modello da utilizzare, bensì al file *en-token.bin*;
(esempio, is = new FileInputStream(modelsPath+"/en-token.bin")).
Nei punti successivi ne vedremo l'utilizzo.

d) **TokenizerModel tokenModel = new TokenizerModel(is);**
tale costrutto, facente riferimento a ***opennlp.tools.tokenize.TokenizerModel***, istanzia un modello di rifrimento per effettuare il tokenizing.

e) **Tokenizer tokenizer = new TokenizerME(tokenModel);**
il tipo *Tokenizer* fa riferimento aall'interfaccia ***opennlp.tools.tokenize.Tokenizer***, mentre TokenizerME è l'istanza di della classe descritta in ***opennlp.tools.tokenize.TokenizerME.TokenizerME(TokenizerModel model)***. Attraverso questa istruzione realizziamo il vero e proprio tokenizer, che permetterà di suddividere i contenuti informativi estratti.

f) **NameFinderME nameFinder = new NameFinderME(model);**
tale costrutto, facente riferimento a ***opennlp.tools.namefind.NameFinderME***, istanzia il name finder vero e proprio, dato in ingresso al costruttore il modello OpenNLP.

g) **String [ ]sentence =tokenizer.tokenize(content);**
il **content** (si veda il punto 2 dell'elenco in questa sezione) viene separato in più **sentence** grazie al metodo **tokenize** del *Tokenizer*

h)**Span nameSpans[ ] = nameFinder.find(sentence);**
Invocando il metodo **find** del **NameFinderME**, vengono individuati gli span (si veda ***opennlp.tools.util.Span***) che identificano le posizioni dei caratterri, all'interno delle frasi ottenute dal tokenizer, che corrispondono alle informazioni trovate; con una semplice procedura si è in grado, dati questi riferimenti, di risalire alle informazioni vere e proprie (in questo caso il codice sorgente è autoesplicativo).

(*Nota: questo schema è generale e ricorrente in tutti i metodi di AdvancedAtomExtractor sopra citati. Per informazioni aggiuntive si osservi il sorgente*

4. Vengono creati gli atomi specifici con un semplice processo iterativo sulle

informazioni estratte.

5. Come ultimo passo, tutti atomi estratti vengono inseriti in un'unica struttura dati e filtrati al fine di unire eventuali atomi identici.

Nel caso del *CustomAtomExtractor* la procedura da utilizzare è:

- in situazioni di estrazione di conoscienza standard, è valido lo schema mostrato nel caso di *AdvancedAtomExtractor*; le differenze sono minimali, e indipendenti dalle API di OpenNLP. Il metodo che avvia la procedura è **generateAtomsFromTrainingSet(String modelPath, File file)**

- Nel caso di procedura semplificata, mediante uso di dizionari, la prima istruzione è **generateModel(String wordsPath, String type)** , che a partire dal precorso del file contenente i lemmi del dizionario, e una stringa che identifica la tipologia di dizionario (ossia la sua natura semantica), genera un oggetto di tipo *Dictionary* (***opennlp.tools.dictionary.Dictionary***): ciò è ottenibile attraverso il metodo **put**, il quale inserisce nel dizionari il set di vocaboli. Affinchè **put** sia applicabile, è necessario che ogni lemma venga inserito in una *StringList* (***opennlp.tools.util.StringList.StringList(String singleToken)***), ad esempio

esempio di creazione e popolazione del dizionario

```
//creazione nuovo Dictionary
dictionary = new Dictionary();

for(int i=0;i<textTokenized.size();i++){
  //System.out.println(textTokenized.get(i));
    dictionary.put(new StringList(textTokenized.get(i)));
}
```

- Una volta generato il dizionario possiamo chiamare il metodo **generateAtoms(File file)**, che sfrutta il dizionario appena creato (il quale fa parte dei campi di *CustomAtomExtractor*):

CustomAtomExtractor.generateAtoms(File file))

```
List <String> fileTokenized= new ArrayList<String>();
    StringTokenizer st = new StringTokenizer(content, " \t\n\r\f,.:;?![]'");

    while (st.hasMoreElements()) {
      fileTokenized.add((String) st.nextElement());
    }

    URI path = file.toURI();

    String[] fileAsTokens = new String[fileTokenized.size()];
    fileAsTokens = fileTokenized.toArray(fileAsTokens);

    DictionaryNameFinder nameFinder = new DictionaryNameFinder(dictionary,type);
    Span[] spans = nameFinder.find(fileAsTokens);
```

In particolare, **DictionaryNameFinder** (opennlp.tools.namefind.DictionaryNameFinder) è una classe messa a disposizione da OpenNLP che permette di trovare una gamma di parole (un dizionario per l'appunto) all'interno di un file su cui è stato fatto tokenizing. Il metodo di riferimento è **find(String[ ] tokens)**, il cui argomento è appunto il pool di vocaboli in cui è stato segmentato il testo, in cui cercare i termini presenti nel dizionario.