RELAZIONE SULLO STATO DELL'ARTE ED EFFICACIA DEI COMPITI DI TOKENIZZAZIONE TRAMITE LA LIBRERIA NLTK (NATURAL LANGUAGE PROCESSING TOOLKIT)

Sommario

Scopo

Lo scopo di questa ricerca è l'individuazione e il testing dei compiti di sents tokenize e di words tokenize di un testo passato come parametro in ingresso, permettendo così di effettuare i compiti successivi di natural language processing

Le risorse utilizzate

Per effettuare i tests si è scelto di utilizzare la risorsa ***Paisa***, corpus in formato conll, così da poter avere un parametro di confronto per verificare la precisione dei singoli tokenizzatori presi in esame.

La scelta è stata dettata dalla mancanza di corpora nella libreria per la lingua italiana.

Ottenere il codice utilizzato

Tutto il codice è liberamente scaricabile. si trova disponibile all'indirizzo:

https://github.com/patriziobellan86/testerTokenizers

Il formato Conll del corpora Paisa

Data Format CoNLL-X

Il formato CONLL è un particolare formato di memorizzazione delle frasi utilizzato per condividere i dati in modo standard tra differenti programmi di elaborazione del linguaggio naturale. Nella libreria nltk, è possibile importare un corpora di questo formato tramite il metodo ConllCorpusReader della classe corpus.

Ogni parola in questo formato è costituita dai seguenti dieci campi:

1	**ID**	Token counter, con indice 1 per ogni nuova frase.
2	**FORM**	Forma della parola
3	**LEMMA**	Lemma della parola
4	**CPOSTAG**	Coarse-grained part-of-speech tag
5	**POSTAG**	Fine-grained part-of-speech tag
6	**FEATS**	Insieme non ordinato di features sintattiche e morfologiche; i parametri di questo insieme possono essere separati da una bar | o un underscore _
7	**HEAD**	L'indice della testa del tokens in caso di parole formate da più tokens
8	**DEPREL**	Dependency relation to the HEAD. Questo parametro è utilizzato per la ricostruzione dell'albero della frase
9	**PHEAD**	Projective head of current token, questo campo indica il valore dell'ID a cui il token corrente è riferito. Se 0 o è un underscore, il parametro non è utilizzato.

10	**PDEPREL**	Dependency relation to the PHEAD, o underscore se non è avviabile. Il set delle relazioni di dipendenza differisce per ogni lingua. Se vi è indicato il tag ROOT, questo indica che questa parola è la testa dell'albero della frase

Il corpora PAISA'

Il corpora PAISA' è una collezione di testi in lingua italiana, raccolti da internet nell'ambito progetto PAISÀ (Piattaforma per l'Apprendimento dell'Italiano Su corpora Annotati) allo scopo di fornire materiale autentico e disponibile gratuitamente per l'apprendimento dell'italiano. I testi sono stati raccolti nel 2010. Tutto il materiale è disponibile e diffuso gratuitamente ed ha una dimensione di circa 250 milioni di tokens.

Tutto il corpus è annotato secondo il formato CONLL, preannotato automaticamente e raffinato manualmente. Il corpus contiene in totale circa 380.000 documenti da circa 1.000 siti distinti per un totale di circa 250 milioni di parole. Circa 260.000 documenti provengono dal Wikipedia, circa 5.600 da altri progetti Wikimedia Foundation. Circa 9.300 documenti provengono da Indymedia, e si stima che circa 65.000 documenti provengano da blog.

Il corpus PAISÀ è messo a disposizione dal progetto PAISÀ (www.corpusitaliano.it) attraverso una licenza creative commons non commerciale.

È possibile scaricare questa risorsa

In formato conll:

http://www.corpusitaliano.it/static/documents/paisa.annotated.CoNLL.utf8.gz

o in formato testuale :

http://www.corpusitaliano.it/static/documents/paisa.raw.utf8.gz

il tagset utilizzato è ISST-TANL Tagsets, la cui specifica è disponibile all'indirizzo:

http://medialab.di.unipi.it/wiki/Tanl_POS_Tagset

La costruzione del corpus nel programma per effettuare i test

Per effettuare i tests si effettua la ricostruzione del corpus, per passare dal formato conll ad una stringa di testo. Per poter effettuare l'efficienza dei singoli tokenizzatori, il testo è stato ricreato valutando differenti parametri come "separatori" di frasi e differenti modalità di separazione tra le parole quando si incontra un carattere non alfabetico (come ad esempio , . ' " ,…) tenendo conto delle possibili condizione che si possono incontrare in un normale testo, come ad esempio in un compito di web crawling

I parametri di ricostruzione delle frasi

indicati come tagS, utilizzati nel programma di test sono:

- *'NONE':*
  il testo è ricostruito senza aggiungere nessun carattere tra una frase e la successiva - u""

- *'NEW LINE':*
  il testo è ricostruito aggiungendo un carattere di Carrige Return Line Feed (sistemi Windows) tra due frasi - u"\n"
- *'TABS':*
  Il testo è ricostruito aggiungendo un carattere di tabulazione tra le frasi - u"\t"
- *'PARAG':*
  Il testo è ricostruito aggiungendo un carattere di tabulazione tra le frasi - u"\n\t"
- *'PARAG_2':*
  Il testo è ricostruito aggiungendo un carattere di tabulazione tra le frasi - u"\n\n\t\t"

I parametri di ricostruzione della frase

Indicati come tagW, utilizzati nel programma di test sono:

- *'SPACE'*
  Viene posto un carattere di spaziatura singola tra ogni parola e carattere non alfabetico –
  Es. Quest ' esempio
- *'AFTER'*
  Viene posto un carattere di spaziatura tra il carattere non alfabetico e la parola successiva, mentre viene omesso lo spazio tra il carattere e la parola precedente –
  Es. Quest' esempio
- *'BEFORE'*
  Viene posto un carattere di spaziatura tra il carattere non alfabetico e la parola precedente, mentre viene omesso lo spazio tra il carattere e la parola successiva –
  Es. Quest 'esempio

Il compito di ricostruzione del corpus è affidato al metodo **CreaPlainText** della classe **Tools**.

Il Codice del metodo CreaPlainText

```
def CreaPlainText (self, tagS='NONE', tagW='SPACE'):
    r"""
        Questo metodo si occupa di creare il corpus da utilizzare per i tests

        # option 'SPACE'|'BEFORE'|'AFTER'
        #SPACE uno spazio tra ogni parola
        #BEFORE niente spazio tra parola e segno dopo
        #AFTER niente spazio tra parola e segno, ma tra segno e parola
            es. "wordPunct word"
    """
```

```
#li registro per poterli utilizzare dopo

self.tagW = tagW
self.tagS = tagS
self.corpusLst = list()

corpus=u''''
for sent in self.sents:
    if tagW == self.SPACE:
        frase = u'' ''.join(sent) + self.TAGS[tagS]
        corpus = corpus + frase
        self.corpusLst.append(frase)
    ############################ ok
    elif tagW == self.AFTER:
        frase = u''''
        for i in xrange(len(sent)):
            if (i+1) < len (sent):
                if not sent[i+1].isalpha() and len(sent[i+1]) == 1:
                    frase = frase + sent[i]
                else:
                    frase = frase + sent[i] + u'' ''
            else:
                 frase = frase + sent[i] + u'' ''
        frase = frase + self.TAGS[tagS]
        self.corpusLst.append(frase)
        corpus=corpus+frase
    ######################## ok
    elif tagW == self.BEFORE:
        frase = u''''
        for i in xrange(len(sent)):
            if (i+1) < len (sent):
                if not sent[i].isalpha() and len(sent[i]) == 1:
                    frase = frase + sent[i]
                else:
                    frase = frase + sent[i] + u'' ''
            else:
                 frase = frase + sent[i] + u'' ''
        frase = frase + self.TAGS[tagS]
        self.corpusLst.append(frase)
        corpus=corpus+frase+self.TAGS[tagS]
    else:
```

```
            print "ATTENZIONE: parametro %s non valido" % tagW
        self.corpusTxt = corpus


        return self.corpusTxt
```

PaisaSentsExtractor.py

Per estrarre le frasi dal corpus di Paisà si è creata la classe paisaSentsExtractor.

Questa classe, durante l'istanziazione dell'oggetto, continua ad estrarre tante frasi fino ad arrivare al numero di parole voluto. Durante la sua dichiarazione è anche possibile specificare più di una cartella in cui salvare le frasi estratte.

I Parametri della Funzione

I parametri impostabili sono:

- paisa: path al file del corpus paisa
- nwords: numero di parole totali che si vuole estrarre
- folderdst: folder di destinazione delle frasi estratte
- folderList: dizionario formato da:
  key -> da quale numero di parole iniziare a salvare
  value->la folder dove salvare

se non impostati, di default la classe provvede ad estrarre tutto il corpus nella cartella di default "corpus raw"

Il Codice

Si riporta il codice integrale della classe

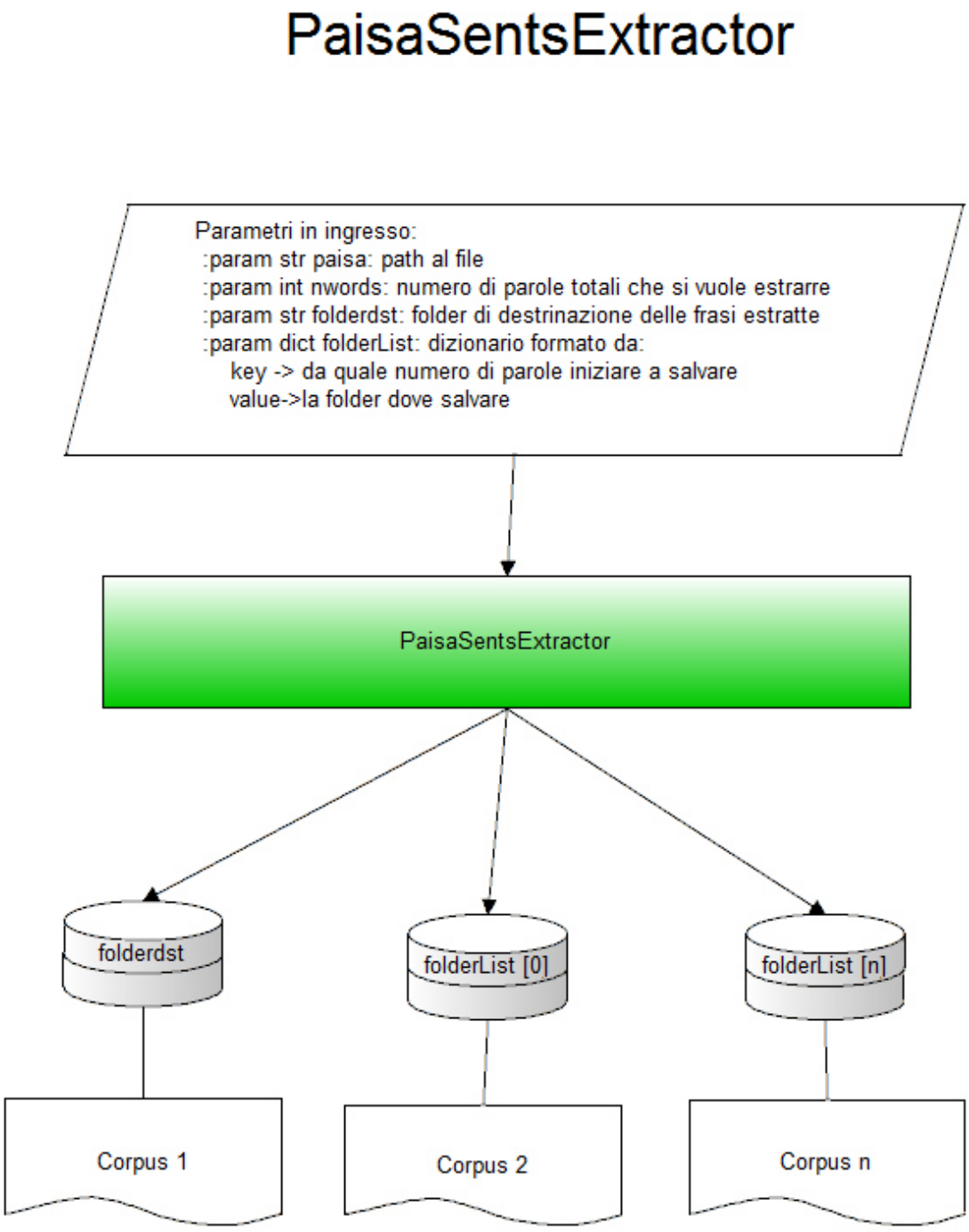
```
class PaisaSentsExtractor ():
    """
        questa classe si occupa di estrarre i dati dal file paisa e di salvarli
        in files separati. uno per ogni frase
    """
    def __author__(self):
        return "Patrizio Bellan \n patrizio.bellan@gmail.com"
    def __version__(self):
        return "0.4.1.b"


    def __init__(self, paisa = "paisa.annotated.CoNLL.utf8", nwords = -1,
```

```
            folderdst = "corpus raw" + os.path.sep, folderList = {-1: "corpus raw" + os.path.sep}):
        r"""
            direttamente durante la creazione dell'oggetto parte l'elaborazione dei dati

            :param str paisa: path al file
            :param int nwords: numero di parole totali che si vuole estrarre
            :param str folderdst: folder di destrinazione delle frasi estratte
            :param dict folderList: dizionario formato da:
                        key -> da quale numero di parole iniziare a salvare
                        value->la folder dove salvare

            :return: None

            esempio:

            >>>  PaisaSentsExtractor (nwords = 15000, paisa = paisaFilename, folderdst = "a" + os.path.sep,
folderList = {5000 : 'b' + os.path.sep, 10000 : 'c' + os.path.sep})
            estrarrà:
            15000 parole, dal file paisaFilename
            salvandole per prima nella folder a
            giunto a 5000 le salva nella folder b
            giunto a 10000 le salva nella folder c



        """

        #per i conteggi uso i float per evitare overflow
        self.folderList = folderList
        self.folderdst = folderdst
        self.extfile = ".conll.txt"
        self.paisa_corpus = paisa
        self.nwords = nwords
        self.__Elabora()


    def __Elabora(self):
        period = []
        nfile = float (0) #n di files scritti
        nwords = float (0) #n parole scritte
```

```
fpaisa = codecs.open(self.paisa_corpus, mode='r', encoding='utf-8')
while True:
   line = fpaisa.read (1)
   if line[0] == u"<":
      if period[0] != u'#' and period[0] != u"" and len(period) > 1:
         frase = []
         period = u"".join (period)
         if len(period.strip ()) > 1:
            try:
               #uso il costrutto try per evitare errori quando la substring manca
               period = period[period.index (u">")+1:]

               for s in period.split (u"\n"):
                  if s != u'\n' and s != u"":
                     if  len(s.split(u"\t")) == 8:
                        s = s + u"\t_\t_\n"
                        frase.append (s)
                        nwords += 1

                  elif frase != []:
                     filename = self.folderdst + str(nfile) + self.extfile
                     print "saving file: ", filename
                     with codecs.open(filename, mode = 'a', encoding = 'utf-8') as out:
                        out.writelines (frase)

                     nfile += 1

                     #controllo se ho salvato il numero di parole desiderate
                     if nwords >= self.nwords:
                        return

                     #controllo se devo cambiare folders
                     if len(self.folderList.keys ()) > 0 and nwords >= min(self.folderList.keys ()):
                        self.folderdst = self.folderList [min(self.folderList.keys())]
                        del self.folderList[min(self.folderList.keys())]

                     frase = []
               except ValueError:
                  period = []
            else:
               period = []
```

```
        period = []
    period.append (line)
fpaisa.close ()
```

Breve schema logico della classe



La Funzione di Score

Per verificare la bontà di ogni singolo test è stata sviluppata la seguente funzione di score per andare a valutare in modo appurato e preciso ogni singolo test, confrontandolo con il testo utilizzato in ingresso per i test, riuscendo ad individuare esattamente come e dove è presente ogni eventuale errore.

Il metodo che assolve questo compito è ScoreTest della classe Tools.

Il Codice del Metodo ScoreTest

```
def ScoreTest (self, fOriginale, fTest, tag):
    r""" Questa funzione calcola lo score di un test
        Ripestto alla funzione utilizzata in Tools, questa verifica la bontà esatta del test
        tenendo conto di eventuali errori non scopribili semplicemente dal rapporto
                                        float(ottenuti / giusti)
          :param list fOriginale: lista dati di partenza
          :param list fTest: lista risultante dall'applocazione del tokenizer a dati di partenze
          :return: lo score effettivo del test
          :rtype: float

    """
    ERROR = 0
    NO_ERROR = 1

    score = list()
    assert type(fOriginale) == type(fTest) and type(fOriginale) == type(list())

    i = 0   #indice di ciclo
    indOrig = 0  #indice a File Originale
    indTest = 0  #indice a file di Test

    while i < len (fOriginale):
        if indOrig >= len(fOriginale):
            break
        elif indTest >= len(fTest):
            break

        jo = 0
        jt = 0
        if fOriginale[indOrig] == fTest[indTest]:
            score.append(NO_ERROR)
        elif fOriginale[indOrig].startswith(fTest[indTest]):
```

```
    #attivo lo sfasamento jt
    tmpTest = fTest[indTest]
    jt = 1   #variabile temporanea di sfasamento nella lista fTest

    while True:
        #controllo fine lista
        if (indTest + jt) >= len(fTest):
            break
        #######new aggiunto: + tag +
        tmpTest = tmpTest + tag + fTest[indTest + jt]
        if fOriginale[indOrig] == tmpTest:
            #aggingo l'error allo score
            #aggiungo tanti error quanti sono gli j
            #ed esco dal ciclo più interno
            score.extend([ERROR] * jt)
            break
        elif fOriginale[indOrig].startswith(tmpTest):
                jt += 1
        else:
            #se sono qui, la precedente di jt era contenuta
            #quindi decremento di uno jt e registro l'errore
            #ed esco dal ciclo
            jt -= 1
            score.extend([ERROR] * jt)
            break
elif fTest[indTest].startswith(fOriginale[indOrig]):
    #fTest potrebbe aver incluso due parole di fOrig
    #Controllo che anche la successiva si la continuazione
    #in fTest
    #Metto il tutto in un ciclo
    #attivo lo sfasamento jo
    jo = 1
    tmpOrig = fOriginale[indOrig]

    while True:
        if (indOrig + jo) >= len (fOriginale):
            break
        #######new aggiunto: + tag +
        tmpOrig = tmpOrig + tag + fOriginale[indOrig + jo]

        if fTest[indTest]  == tmpOrig:
```

```
                #se sono arrivato a far combaciare origine e test
                #registro gli errori
                score.extend([ERROR] * jo)
                break
            elif fTest[indTest].startswith(tmpOrig):
                jo += 1
            else:
                #se sono qui, la precedente di jo era contenuta
                #quindi decremento di uno jo e registro l'errore
                #ed esco dal ciclo
                jo -= 1
                score.extend([ERROR] * jo)

                #se jo è zero un errore lo devo segnare
                if jo < 1 :
                    score.extend([ERROR])

                break
        else:
            #il successivo di orig non è contenuta in test quindi
            #registro l'errore e continuo
            score.append(ERROR)

        indTest += jt + 1
        indOrig += jo + 1
        i += 1

    if indOrig > indTest:
        #arrivato alla fine del ciclo, se indOrig è maggiore di indTest
        #aggiungo tanti errori quanti sono la loro differenze
        score.extend([ERROR] * (indOrig - indTest))
    elif indOrig < indTest:
        #arrivato alla fine del ciclo, se indTest è maggiore di indOrig
        #aggiungo tanti errori quanti sono la loro differenze
        score.extend([ERROR] * (indTest - indOrig))

    try:
        return float(sum(score) / len(score))
    except ZeroDivisionError:
        return float(0)
```

Il Ciclo di Testing

Ogni singolo tokenizzatore effettua un ciclo di test in cui se ne testa la sua efficacia secondo due dimensioni distinte:

- PARAMS: le possibili combinazioni di ricostruzione del testo
- DIMS: varianti sulle dimensioni del testo da tokenizzare.

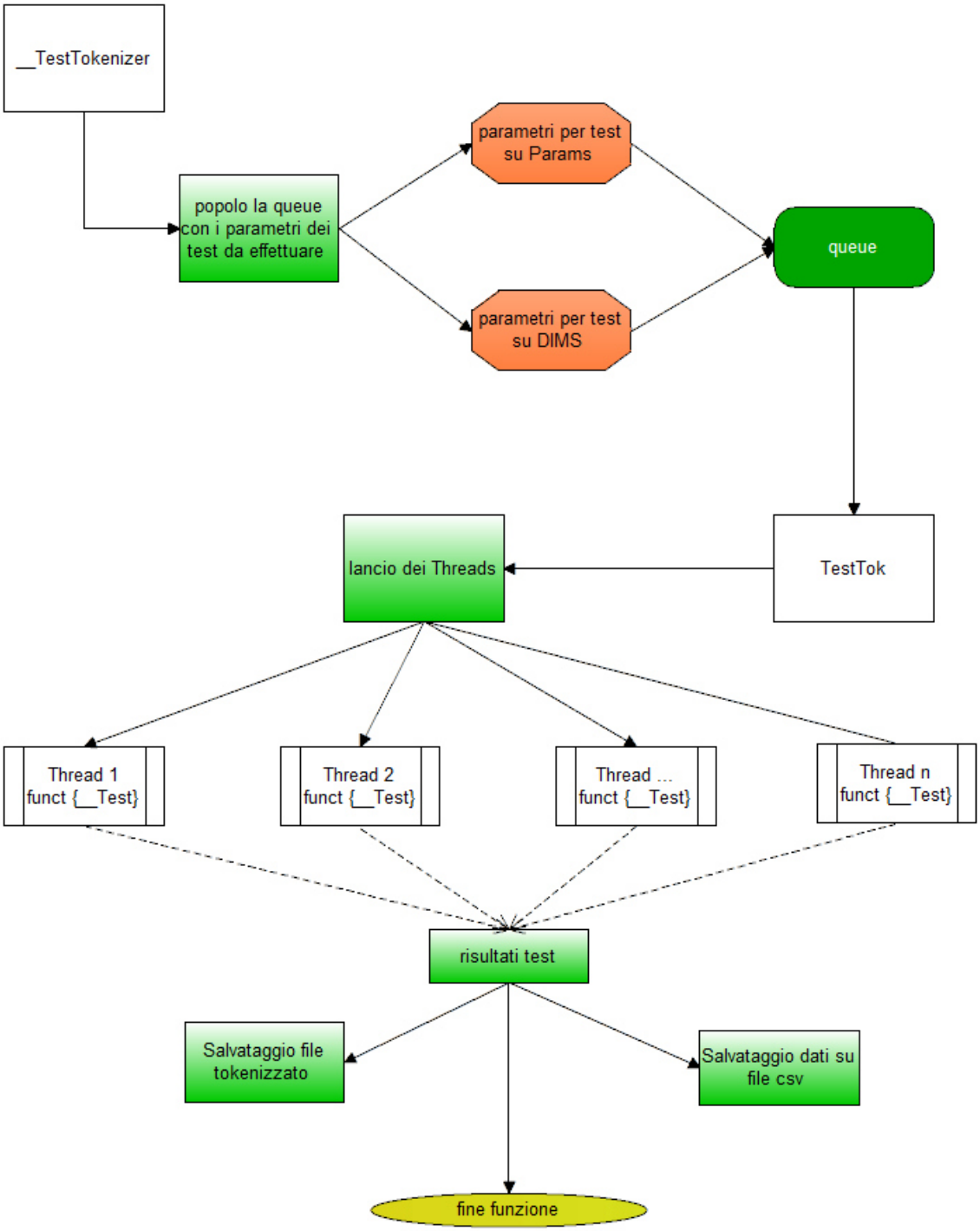
Ogni metodo di test invia il proprio oggetto tokenizzatore direttamente al metodo __***TestTokenizer*** il quale, per prima cosa crea le possibili varianti del test secondo le dimensioni Params e Dims; successivamente invia tutti i dati stimati al metodo ***TestTok*** il quale si occupa di lanciare in esecuzione, con modalità multithread, i test da effettuare.

Il metodo __***Test*** è il core di questa sezione; qui viene effettuato il test vero e proprio.

Schema logico di funzionamento del ciclo di testing

Il flusso logico dell'esecuzione dei test segue il seguente schema:

CICLO DI TEST



Il Codice della funzione di test

```
def __TestTokenizer (self, testName, dimTests, tok, tipo, attributi = None, attrFilename = None):
    def Tests ():
        self.queue =  Queue.Queue ()
        #popolo la coda di test
        dim = dimTests[0]
        for paramS in self.paramCorpusCreationS:
            for paramW in self.paramCorpusCreationW:
                self.queue.put((testName, tok, dim, paramS, paramW,
                    self.TIPO_PARAMS, tipo, attributi, attrFilename))
        for dim in dimTests:
            self.queue.put((testName, tok, dim, self.normalParamS,
                    self.normalParamW, self.TIPO_DIMENS, tipo, attributi, attrFilename))
        return self.TestTok ()
    return Tests ()
def TestTok (self):
    class MyThread  (threading.Thread):
        def __init__ (self, testFunction, numberTh, dati, r):
            threading.Thread.__init__(self)
            self.name = numberTh
            self.__Test = testFunction
            self.dati = dati
            self.r = r
        def run (self):
            try:
                print "Inizio thread", self.name
                self.r.append(self.__Test (*self.dati))
                print "Fine thread", self.name
            except:
                print "break thread", self.name
                #global r
                self.r.append (False)
                return
    i=0
    r = []
    while not self.queue.empty ():
        lt = []
        #utilizzo tanti thread quanti sono i processori logici dell'elaboratore
        for  i in xrange (multiprocessing.cpu_count()):
```

```
        #for  i in xrange (self.queue.qsize ()):
            try:
                if self.queue.empty ():
                    #se non ci sono più dati da elaborare esco dal ciclo di elaborazione
                    break
                t = MyThread (self.__Test, i, self.queue.get (), r)
                #thl.release ()
                t.daemon = True
                t.start ()
                #t.join ()
                i+=1
                lt.append (t)
            except:
                pass
        #aspetto finchè ci sono thread vivi
        while sum([l.isAlive() for l in lt]):
            pass
        #r = [r.r for r in t]
        r.sort()
        #dopo sort i False sono messi per primi
        if not r[0]:
            return False
    return True
#     return r


  def __Test (self, testName, tok, dim, paramS, paramW, tipoTest, tipo, attributi, attrFilename, registra =
True):
    try:
        s = u"\nTEST : {}".format (testName)
        print s
        s = s + "\nTest su %d in condizioni paramS: %s paramW: %s" % (dim, paramS, paramW)
        #self.tools.PrintOut (s)
        #Oggetto per il corpus
        corpusObj = Tools (dim)
        corpusObj.CaricaCorpus ()
        datiOut = tok.tokenize (corpusObj.CreaPlainText (paramS, paramW))
        if tipo == self.tools.SENT:
            tag = paramS
        else:
            tag = u""
        r, score = self.tools.RisultatiTest(testName, datiOut, tipo, corpusObj.words, corpusObj.corpusLst, tag)
```

```
            self.tools.PrintOut (s + r)
            #Salvo il file elaborato dal tokenizzatore
            fileTest = None
            if self.save and registra:
                if not attrFilename:
                    attrFilename = u''''
                filename = self.folderTestFiles + testName + u'' '' + attrFilename + u'' '' + paramS + u'' '' + paramW +
u'' '' +  unicode(len(corpusObj.words)) + u'' '' + u''.txt''
                fileTest = filename


                self.tools.SaveFile (filename = filename, dati = datiOut)
            if self.EuristicaNoZero (score):
                self.tools.PrintOut (''Euristica NoZero superata'')
            else:
                self.tools.PrintOut (''Euristica NoZero Non superata'')
            test = {'paramS':paramS, 'paramW':paramW, 'dim':len(corpusObj.words),
                    'score':score, 'euristicaNoZero': self.EuristicaNoZero (score),
                    'tipoTest': tipoTest, 'fileTest': fileTest, 'attributiTok': attributi}
            if registra:
                self.risultatiTest[testName].append(test)
            return (score, test)
        except:
            print ''il testo non è elaborabile dal tokenizzatore %s''% testName
            return False
```

Moduli presenti in nltk

La libreria nltk presenta al suo interno una serie di classi per effettuare le operazioni di tokenizzazione in modo efficiente ed efficacie, evitando di dover riscrivere le procedure ad hoc per questa tipologia di compiti che presentano una soluzione comune.

Grazie all'individuazione dei parametri specifici per ogni singolo compito, è possibile andare a programmare opportunamente gli strumenti presenti.

Nella libreria sono presenti le seguenti classi che svolgono i suddetti compiti:

- nltk.tokenize.simple
- nltk.tokenize.regexp
- nltk.tokenize.punkt
- nltk.tokenize.sexpr
- nltk.tokenize.treebank
- nltk.tokenize.stanford
- nltk.tokenize.texttiling

- nltk.tokenize.casual
- nltk.tokenize.mwe

Classi prese in esame durante la fase di test

Durante la fase di testing, sono stati presi in esame soltanto i moduli base; i moduli di funzioni derivate (come ad esempio il BlanklineTokenizer) sono testati come varianti del modulo di base (che nel caso del BlanklineTokenizer, risulta essere una variante del modulo di base nltk.tokenize.regexp).

I moduli di base presi in esame sono:

- nltk.tokenize
- nltk.tokenize.simple
- nltk.tokenize.regexp
- nltk.tokenize.punkt
- nltk.tokenize.treebank
- nltk.tokenize.texttiling

La Classe nltk.tokenize

La classe nltk.tokenize è la classe base per assolvere i compiti presi in esame.

Essa, oltre a fornire la possibilità di accedere alle classi specifiche , offre i riferimenti ai metodi standard per svolgere i compiti di words e sents tokenization

nltk.tokenize.word_tokenize (corpus)

Questo metodo di classe permette di suddividere un corpus in parole.

Il compito viene assolto tramite il richiamo a due metodi specifici:

- ***PunktSentenceTokenizer*** che permette di suddividere il corpus in frasi
- ***TreebankWordTokenizer*** che permette di suddividere le frasi in parole

return [token for sent in sent_tokenize(text, language) for token in _treebank_word_tokenize(sent)]

Questo compito è svolto tramite l'impostazione standard calibrata sulla lingua inglese, come percepibile dal parametro in ingresso di default *language* nella dichiarazione del metodo

def word_tokenize(text, language='english'):

Il parametro language permette di informare il ***PunktSentenceTokenizer*** di caricare il pickle del Punkt pre-addestrato per la lingua desiderata

All’interno del programma di test creato, questa funzione è svolta dal metodo TestSimpleWordTokenizer della classe TestTokenizer tramite il seguente segmento di codice:

Il Codice del metodo SIMPLE WORD TOKENIZER

```
def TestSimpleWordTokenizer (self):
    r"""
       questo metodo effettua il test sullo standard word tokenizer
    """
    class Tok :
       def tokenize (self, sents):
          return nltk.tokenize.word_tokenize (sents)

    testName = u"STANDARD WORD TOKENIZER"
    dimTests = self.dimTests
    tok = Tok ()
    tipo = self.tools.WORD
    self.__TestTokenizer (testName, dimTests, tok, tipo)
```

Il programma prevede anche il test basato sulla lingua italiana, specificandola direttamente durante la chiamata del metodo tramite il metodo di classe TestSimpleWordTokenizerIta, come mostrato dal seguente codice:

Il Codice del metodo SIMPLE WORD TOKENIZER ITA

```
def TestSimpleWordTokenizerIta (self):
    r"""
       questo metodo effettua il test sullo standard word tokenizer ita
    """
    class Tok :
       def tokenize (self, sents):
          return nltk.tokenize.word_tokenize (sents, language='italian')

    testName = u"STANDARD WORD TOKENIZER ITA"
    dimTests = self.dimTests
    tok = Tok ()
    tipo = self.tools.WORD
    self.__TestTokenizer (testName, dimTests, tok, tipo)
```

La classe Tok

All'interno dei due metodi precedenti è presente la dichiarazione della classe Tok. Questo è stato necessario per definire un oggetto che abbia il metodo tokenize, inquanto i due tokenizzatori suddetti ne erano sprovvisti.

nltk.tokenize.sent_tokenize (corpus)

Questo metodo di classe permette di suddividere un corpus in frasi.

Il compito viene assolto tramite il richiamo al metodo:

- ***PunktSentenceTokenizer*** che permette di suddividere il corpus in frasi

tokenizer = load('tokenizers/punkt/{0}.pickle'.format(language))

return tokenizer.tokenize(text)

Questo compito è svolto tramite l'impostazione standard calibrata sulla lingua inglese, come percepibile dal parametro in ingresso di default *language* nella dichiarazione del metodo

def sent_tokenize(text, language='english'):

Il parametro language permette di informare il ***PunktSentenceTokenizer*** di caricare il pickle del Punkt pre-addestrato per la lingua desiderata

All'interno del programma di test creato, questa funzione è svolta dal metodo TestSimpleTokenizer della classe TestTokenizer tramite il seguente segmento di codice:

SIMPLE TOKENIZER

```
def TestSimpleTokenizer (self):
    r"""
        questo metodo effettua il test sullo standard sent tokenizer
    """
    class Tok :
        def tokenize (self, sents):
            return nltk.tokenize.sent_tokenize (sents)

    testName =  u"SIMPLE SENT TOKENIZER"
    dimTests = self.dimTests
    tok = Tok ()
    tipo = self.tools.SENT
    self.__TestTokenizer (testName, dimTests, tok, tipo)
```

Il programma prevede anche il test basato sulla lingua italiana, specificandola direttamente durante la chiamata del metodo, tramite il metodo di classe TestSimpleTokenizerIta, come mostrato dal seguente codice:

SIMPLE TOKENIZER ITA

```
def TestSimpleTokenizerIta (self):
    r"""
       questo metodo effettua il test sul simple sent tokenizer ita
    """
    class Tok :
       def tokenize (self, sents):
          return nltk.tokenize.sent_tokenize (sents, language='italian')

    testName = u"SIMPLE SENT TOKENIZER ITA"
    dimTests = self.dimTests
    tok = Tok ()
    tipo = self.tools.SENT
    self.__TestTokenizer (testName, dimTests, tok, tipo)
```

La Classe nltk.tokenize.simple

La classe nltk.tokenize.simple permette di effettuare semplicissimi compiti di tokenizzazione. Essenzialmente i metodi di questa classe sono un'interfaccia ai metodi della classe String di python per lavorare sulle stringhe

nltk.tokenize.simple.StringTokenizer

Questo metodo utilizza il carattere di spazio come delimitatore di parole. Questo metodo è l'interfaccia al metodo split(" ") della classe String. All'interno del programma questo compito è testato dal seguente codice:

Il Codice del metodo SIMPLE SPACE TOKENIZER

```
def TestSimpleSpaceTokenizerWord (self):
    r"""
       questo metodo effettua il test sullo simple space word tokenize
    """
    testName = u"SIMPLE SPACE WORD TOKENIZER"
```

```
dimTests = self.dimTests
tok = nltk.tokenize.simple.SpaceTokenizer()
tipo = self.tools.WORD
self.__TestTokenizer (testName, dimTests, tok, tipo)
```

nltk.tokenize.simple.TabTokenizer

Questo metodo utilizza il carattere di tabulazione come delimitatore di parole. Questo metodo è l'interfaccia al metodo split("\t") della classe String. All'interno del programma questo compito è testato dal seguente codice:

Il Codice del metodo SIMPLE TAB TOKENIZER

```
def TestSimpleTabTokenizerWord (self):
    r"""
       questo metodo effettua il test sullo simple tab tokenize
    """
    testName = u"SIMPLE TAB SENT TOKENIZER"
    dimTests = self.dimTests
    tok = nltk.tokenize.simple.TabTokenizer()
    tipo = self.tools.SENT
    self.__TestTokenizer (testName, dimTests, tok, tipo)
```

nltk.tokenize.simple.LineTokenizer

Questo metodo utilizza il carattere di CRLF come delimitatore di parole. Questo metodo è l'interfaccia al metodo split("\n") della classe String. All'interno del programma questo compito è testato dal seguente codice:

Il Codice del metodo SIMPLE LINE TOKENIZER

```
def TestSimpleLineTokenizerWord (self):
    r"""
       questo metodo effettua il test sullo simple line tokenize
    """
    testName = u"SIMPLE LINE SENT TOKENIZER"
    dimTests = self.dimTests
    tok = nltk.tokenize.simple.LineTokenizer()
    tipo = self.tools.SENT
    self.__TestTokenizer (testName, dimTests, tok, tipo)
```

La Classe nltk.tokenize.treebank

La classe nltk.tokenize.treebank permette di effettuare la tokenizzazione di una frase in parole tramite l'ausilio della classe nltk.re, effettuando la tokenizzazione tramite l'applicazione e sostituzione di regular expression preconfigurate, in modo sequenziale alla frase passata come parametro in ingresso, come visibile dal codice sorgente in nltk:

TOKENIZE

```
def tokenize(self, text):
    for regexp, substitution in self.STARTING_QUOTES:
        text = regexp.sub(substitution, text)
    for regexp, substitution in self.PUNCTUATION:
        text = regexp.sub(substitution, text)
    for regexp, substitution in self.PARENS_BRACKETS:
        text = regexp.sub(substitution, text)
    text = " " + text + " "
    for regexp, substitution in self.ENDING_QUOTES:
        text = regexp.sub(substitution, text)
    for regexp in self.CONTRACTIONS2:
        text = regexp.sub(r' \1 \2 ', text)
    for regexp in self.CONTRACTIONS3:
        text = regexp.sub(r' \1 \2 ', text)
    return text.split()
```

le regular expression del treebank tokenizer

Le regular expression utilizzate per assolvere questo compito sono:

```
#starting quotes
STARTING_QUOTES = [
        (re.compile(r'^\"'), r'``'),
        (re.compile(r'(``)'), r' \1 '),
        (re.compile(r'([ (\[{<])"'), r'\1 `` '),]

#punctuation
PUNCTUATION = [
        (re.compile(r'([:,])([^\d])'), r' \1 \2'),
        (re.compile(r'([:,])$'), r' \1 '),
        (re.compile(r'\.\.\.'), r' ... '),
```

```
        (re.compile(r'[;@#$%&]'), r' \g<0> '),
        (re.compile(r'([^\.])(\.)([\]\)}>"\']*)\s*$'), r'\1 \2\3 '),
        (re.compile(r'[?!]'), r' \g<0> '),
        (re.compile(r"([^'])' "), r"\1 ' "),]

#parens, brackets, etc.
PARENS_BRACKETS = [
        (re.compile(r'[\]\[\(\)\{\}\<\>]'), r' \g<0> '),
        (re.compile(r'--'), r' -- '),]

#ending quotes
ENDING_QUOTES = [
        (re.compile(r'"'), " '' "),
        (re.compile(r'(\S)(\'\')'), r'\1 \2 '),
        (re.compile(r"([^' ])('[sS]|'[mM]|'[dD]|') "), r"\1 \2 "),
        (re.compile(r"([^' ])('ll|'LL|'re|'RE|'ve|'VE|n't|N'T) "), r"\1 \2 "),]

# List of contractions adapted from Robert MacIntyre's tokenizer.
CONTRACTIONS2 = [re.compile(r"(?i)\b(can)(not)\b"),
        re.compile(r"(?i)\b(d)('ye)\b"),
        re.compile(r"(?i)\b(gim)(me)\b"),
        re.compile(r"(?i)\b(gon)(na)\b"),
        re.compile(r"(?i)\b(got)(ta)\b"),
        re.compile(r"(?i)\b(lem)(me)\b"),
        re.compile(r"(?i)\b(mor)('n)\b"),
        re.compile(r"(?i)\b(wan)(na) ")]

CONTRACTIONS3 = [re.compile(r"(?i) ('t)(is)\b"),
        re.compile(r"(?i) ('t)(was)\b")]

CONTRACTIONS4 = [re.compile(r"(?i)\b(whad)(dd)(ya)\b"),
        re.compile(r"(?i)\b(wha)(t)(cha)\b")]
```

Nel programma di test, questo compito viene effettuato tramite il metodo TestTreeBankTokenizer della classe TestTokenizer tramite il seguente segmento di codice:

Il Codice del metodo TREEBANK TOKENIZER

```
    def TestTreeBankTokenizer (self):
    r"""
```

```
    questo metodo effettua il test sul treebank tokenizer
    """
    testName = u"STANDARD WORD TOKENIZER ITA"
    dimTests = self.dimTests
    tok = nltk.tokenize.TreebankWordTokenizer()
    tipo = self.tools.WORD
    self.__TestTokenizer (testName, dimTests, tok, tipo)
```

La Classe nltk.tokenize.RegexpTokenizer

La classe nltk.tokenize.RegexpTokenizer fornisce gli strumenti adatti per poter configurare un tokenizzatore che assolva il compito, tramite l'ausilio delle regular expression.

Grazie a questa classe è possibile configurare in ogni parametro di comportamento.

I Parametri Standard configurabili

Come si evince dal codice sorgente di questa classe, ogni oggetto creato può essere configurato nel comportamento tramite i seguenti parametri:

- pattern
- gaps
- discard_empty
- flags

Il Parametro *pattern*

Questo parametro rappresenta una stringa valida di regular expression

Il Parametro *gaps*

Questo parametro può assumere soltanto valori booleani (**True** or **False**).

True
Il pattern del tokenizzatore è utilizzato per trovare i separatori tra i ***tokens***

False
Il pattern del tokenizzatore è utilizzato per trovare i ***tokens***

Il Parametro *discard_empty*

Questo parametro può assumere soltanto valori booleani (**True** or **False**). I tokens vuoti vengono generati solo ed esclusivamente se il parametro gaps è True

True
I Tokens vuoti vengono eliminati

False
I Tokens vuoti non vengono eliminati

Il Parametro *flag*

Questo parametro è utilizzato per definire il tipo di compilazione del pattern, può assumere solo i seguenti valori:

re.UNICODE

Cerca la corrispondenza usando le impostazioni locali. Questa impostazione permette di far includere al compilatore delle regular expression le lettere proprie di una determinata lingua, come facente parte di una parola. Ad esempio se come impostazione locale avessi impostato la lingua francese, tramite questo parametro, il carattere "ç" verrà incluso come carattere di una parola, come i caratteri [a-Z].

re.MULTILINE

Il comportamento normale dei metacaratteri ^ e & utilizzati per la costruzione del pattern di ricerca in una regular expression è quello di cercare il pattern con corrispondenza all'inizio e/o alla fine del testo su cui effettuare la ricerca. Tramite l'impostazione di questo parametro, la ricerca viene effettuata non solo all'inizio/fine ma in tutte le righe di testo del testo di ricerca

re.DOTALL

Il carattere speciale " **.** " cerca ogni carattere, incluso il fine riga, senza questa opzione, " **.** " cercherà tutto *eccetto* il fine riga.

I Regular Expression del Programma di Test

Nel programma di tests sono state creati i seguenti metodi per testare le varie opzioni configurabili del RegexpTokenizer:

- TestREWordTok_patter_w_gaps_T_discardEmpty_T_flags_reUNI
- TestREWordTok_patter_w_gaps_T_discardEmpty_T_flags_reMULTI
- TestREWordTok_patter_w_gaps_T_discardEmpty_T_flags_reDOTALL
- TestREWordTok_patter_w_gaps_T_discardEmpty_F_flags_reUNI
- TestREWordTok_patter_w_gaps_T_discardEmpty_F_flags_reMULTI
- TestREWordTok_patter_w_gaps_T_discardEmpty_F_flags_reDOTALL

- TestREWordTok_patter_w_gaps_F_discardEmpty_T_flags_reUNI

- TestREWordTok_patter_w_gaps_F_discardEmpty_T_flags_reMULTI
- TestREWordTok_patter_w_gaps_F_discardEmpty_T_flags_reDOTALL
- TestREWordTok_patter_w_gaps_F_discardEmpty_F_flags_reUNI
- TestREWordTok_patter_w_gaps_F_discardEmpty_F_flags_reMULTI
- TestREWordTok_patter_w_gaps_F_discardEmpty_F_flags_reDOTALL

Tutte queste funzioni sono richiamate dal metodo ***AvviaTestREWordTok***, il quale si occupa di gestire il pattern in ingresso in base alla tipologia di test (discrimina tra i test da effettuare sulle frasi da quelli da effettuare sulle parole) e richiama direttamente i metodi associati sopraelencati

Tramite l'utilizzo di questi metodi è possibile testare un pattern in tutte le sue opzioni su uno stesso corpus.

Il Codice del metodo AvviaTestREWordTok

```
def AvviaTestREWordTok(self, tipo):
    r"""
        self.patterns
        dict([tuple(patternName, tipo)]= pattern)
    """
    s=u"\n\nINIZIO SESSIONE  Regular Expression Tokenizers"
    self.tools.PrintOut(s)
    for pattern in self.patterns.keys():
        if tipo == pattern[1]:
            #Avvio i Tests per il tipo
            self.TestREWordTok_patter_w_gaps_F_discard_empty_T_flags_reUNI (
                pattern = self.patterns[pattern], patternName = pattern[0], tipo = tipo)
            self.TestREWordTok_patter_w_gaps_F_discardEmpty_T_flags_reMULTI (
                    pattern = self.patterns[pattern], patternName = pattern[0], tipo = tipo)
            self.TestREWordTok_patter_w_gaps_F_discardEmpty_T_flags_reDOTALL (
                    pattern = self.patterns[pattern], patternName = pattern[0], tipo = tipo)

            self.TestREWordTok_patter_w_gaps_F_discard_empty_F_flags_reUNI(
                    pattern = self.patterns[pattern], patternName = pattern[0], tipo = tipo)
            self.TestREWordTok_patter_w_gaps_F_discardEmpty_F_flags_reMULTI (
                    pattern = self.patterns[pattern], patternName = pattern[0], tipo = tipo)
            self.TestREWordTok_patter_w_gaps_F_discardEmpty_F_flags_reDOTALL (
                    pattern = self.patterns[pattern], patternName = pattern[0], tipo = tipo)

            self.TestREWordTok_patter_w_gaps_T_discardEmpty_T_flags_reUNI (
                    pattern = self.patterns[pattern], patternName = pattern[0], tipo = tipo)
            self.TestREWordTok_patter_w_gaps_T_discardEmpty_T_flags_reMULTI (
```

```
            pattern = self.patterns[pattern], patternName = pattern[0], tipo = tipo)
        self.TestREWordTok_patter_w_gaps_T_discardEmpty_T_flags_reDOTALL (
            pattern = self.patterns[pattern], patternName = pattern[0], tipo = tipo)

        self.TestREWordTok_patter_w_gaps_T_discardEmpty_F_flags_reUNI (
            pattern = self.patterns[pattern], patternName = pattern[0], tipo = tipo)
        self.TestREWordTok_patter_w_gaps_T_discardEmpty_F_flags_reMULTI (
            pattern = self.patterns[pattern], patternName = pattern[0], tipo = tipo)
        self.TestREWordTok_patter_w_gaps_T_discardEmpty_F_flags_reDOTALL (
            pattern = self.patterns[pattern], patternName = pattern[0], tipo = tipo)
```

TestREWordTok_patter_w_gaps_T_discardEmpty_T_flags_reUNI

In questo metodo viene testato il pattern con le seguenti configurazioni:

- Gaps = True
- Discard_Empty = True
- Flag = re.UNICODE
-

Il Codice del metodo

```
def TestREWordTok_patter_w_gaps_T_discardEmpty_T_flags_reUNI (self,
                                pattern, patternName, tipo):
    r"""
        questo metodo effettua sui RE tokenizer

        :param str pattern: il pattern re da testare
        :param str patternName: il nome del patten
    """

    testName = patternName
    dimTests = self.dimTests
    tok = nltk.tokenize.RegexpTokenizer (pattern, gaps=True, discard_empty=True, flags=re.UNICODE)
    tipo = tipo
    attributiTok = {'gap': True, 'discardEmpty': True, 'flags': 're.UNI'}
    attrfn = u"gap=True discardEmpty=True flags=re.UNI"
    self.__TestTokenizer (testName, dimTests, tok, tipo, attributiTok, attrfn)
```

TestREWordTok_patter_w_gaps_T_discardEmpty_T_flags_reMULTI

In questo metodo viene testato il pattern con le seguenti configurazioni:

- Gaps = True
- Discard_Empty = True
- Flag = re. MULTILINE

Il codice del metodo

```
def TestREWordTok_patter_w_gaps_T_discardEmpty_T_flags_reMULTI (self,
                                pattern, patternName, tipo):
    r"""
        questo metodo effettua sui RE tokenizer

        :param str pattern: il pattern re da testare
        :param str patternName: il nome del patten
    """

    testName = patternName
    dimTests = self.dimTests
    tok = nltk.tokenize.RegexpTokenizer (pattern, gaps=True, discard_empty=True, flags=re.MULTILINE)
    tipo = tipo
    attributiTok ={'gap': True, 'discardEmpty': True, 'flags': 're.MULTI'}
    attrfn =  u"gap=True discardEmpty=True flags=re.MULTI"
    self.__TestTokenizer (testName, dimTests, tok, tipo, attributiTok, attrfn)
```

TestREWordTok_patter_w_gaps_T_discardEmpty_T_flags_reDOTALL

In questo metodo viene testato il pattern con le seguenti configurazioni:

- Gaps = True
- Discard_Empty = True
- Flag = re. DOTALL'

Il codice del metodo

```
def TestREWordTok_patter_w_gaps_T_discardEmpty_T_flags_reDOTALL (self,
                                pattern, patternName, tipo):
    r"""
        questo metodo effettua sui RE tokenizer

        :param str pattern: il pattern re da testare
        :param str patternName: il nome del patten
```

```
        """

        testName = patternName
        dimTests = self.dimTests
        tok = nltk.tokenize.RegexpTokenizer (pattern, gaps=True, discard_empty=True, flags=re.DOTALL)
        tipo = tipo
        attributiTok = {'gap': True, 'discardEmpty': True, 'flags': 're.DOTALL'}
        attrfn = u"gap=True discardEmpty=True flags=re.DOTALL"
        self.__TestTokenizer (testName, dimTests, tok, tipo, attributiTok, attrfn)
```

TestREWordTok_patter_w_gaps_T_discardEmpty_F_flags_reUNI

In questo metodo viene testato il pattern con le seguenti configurazioni:

- Gaps = True
- Discard_Empty = False
- Flag = re.UNICODE

Il codice del metodo

```
    def TestREWordTok_patter_w_gaps_T_discard_empty_F_flags_reUNI (self,
                                    pattern, patternName, tipo):
        r"""
           Standard Regular Expression Tokenizer

           questo metodo effettua sui RE tokenizer

           :param str pattern: il pattern re da testare
           :param str patternName: il nome del patten
        """

        testName = patternName
        dimTests = self.dimTests
        tok = nltk.tokenize.RegexpTokenizer (pattern, gaps=True, discard_empty=False, flags=re.UNICODE)
        tipo = tipo
        attributiTok = {'gap': True, 'discard_empty': False, 'flags': 're.UNI'}
        attrfn = u"gap=True discard_empty=False flags=re.UNI"
        self.__TestTokenizer (testName, dimTests, tok, tipo, attributiTok, attrfn)
```

TestREWordTok_patter_w_gaps_T_discardEmpty_F_flags_reMULTI

In questo metodo viene testato il pattern con le seguenti configurazioni:

- Gaps = True
- Discard_Empty = False
- Flag = re. MULTILINE

Il codice del metodo

```
def TestREWordTok_patter_w_gaps_T_discardEmpty_F_flags_reMULTI (self,
                                pattern, patternName, tipo):
    r"""
        questo metodo effettua sui RE tokenizer

        :param str pattern: il pattern re da testare
        :param str patternName: il nome del patten
    """

    testName = patternName
    dimTests = self.dimTests
    tok = nltk.tokenize.RegexpTokenizer (pattern, gaps=True, discard_empty=False, flags=re.MULTILINE)
    tipo = tipo
    attributiTok = {'gap': True, 'discardEmpty': False, 'flags': 're.MULTI'}
    attrfn = u"gap=True discardEmpty=False flags=re.MULTI"
    self.__TestTokenizer (testName, dimTests, tok, tipo, attributiTok, attrfn)
```

TestREWordTok_patter_w_gaps_T_discardEmpty_F_flags_reDOTALL

In questo metodo viene testato il pattern con le seguenti configurazioni:

- Gaps = True
- Discard_Empty = False
- Flag = re. DOTALL'

Il codice del metodo

```
def TestREWordTok_patter_w_gaps_T_discardEmpty_F_flags_reDOTALL (self,
                                pattern, patternName, tipo):
    r"""
        questo metodo effettua sui RE tokenizer

        :param str pattern: il pattern re da testare
        :param str patternName: il nome del patten
```

```
        """

        testName = patternName
        dimTests = self.dimTests
        tok = nltk.tokenize.RegexpTokenizer (pattern, gaps=True, discard_empty=False, flags=re.DOTALL)
        tipo = tipo
        attributiTok = {'gap': True, 'discardEmpty': False, 'flags': 're.DOTALL'}
        attrfn =  u'' gap=True discardEmpty= False flags=re.DOTALL''
        self.__TestTokenizer (testName, dimTests, tok, tipo, attributiTok, attrfn)
```

TestREWordTok_patter_w_gaps_F_discardEmpty_T_flags_reUNI

In questo metodo viene testato il pattern con le seguenti configurazioni:

- Gaps = False
- Discard_Empty = True
- Flag = re.UNICODE
-

Il Codice del metodo

```
    def TestREWordTok_patter_w_gaps_F_discardEmpty_T_flags_reUNI (self,
                                    pattern, patternName, tipo):
        r"""
           questo metodo effettua sui RE tokenizer

           :param str pattern: il pattern re da testare
           :param str patternName: il nome del patten
        """

        testName = patternName
        dimTests = self.dimTests
        tok = nltk.tokenize.RegexpTokenizer (pattern, gaps=False, discard_empty=True, flags=re.UNICODE)
        tipo = tipo
        attributiTok = {'gap': False, 'discardEmpty': True, 'flags': 're.UNI'}
        attrfn = u''gap=False discardEmpty=True flags=re.UNI''
        self.__TestTokenizer (testName, dimTests, tok, tipo, attributiTok, attrfn)
```

TestREWordTok_patter_w_gaps_F_discardEmpty_T_flags_reMULTI

In questo metodo viene testato il pattern con le seguenti configurazioni:

- Gaps = False

- Discard_Empty = True
- Flag = re. MULTILINE

Il codice del metodo

```
def TestREWordTok_patter_w_gaps_F_discardEmpty_T_flags_reMULTI (self,
                                pattern, patternName, tipo):
    r"""
       questo metodo effettua sui RE tokenizer

       :param str pattern: il pattern re da testare
       :param str patternName: il nome del patten
    """

    testName = patternName
    dimTests = self.dimTests
    tok = nltk.tokenize.RegexpTokenizer (pattern, gaps=False, discard_empty=True, flags=re.MULTILINE)
    tipo = tipo
    attributiTok ={'gap': False, 'discardEmpty': True, 'flags': 're.MULTI'}
    attrfn =  u"gap=False discardEmpty=True flags=re.MULTI"
    self.__TestTokenizer (testName, dimTests, tok, tipo, attributiTok, attrfn)
```

TestREWordTok_patter_w_gaps_F_discardEmpty_T_flags_reDOTALL

In questo metodo viene testato il pattern con le seguenti configurazioni:

- Gaps = False
- Discard_Empty = True
- Flag = re. DOTALL'

Il codice del metodo

```
def TestREWordTok_patter_w_gaps_F_discardEmpty_T_flags_reDOTALL (self,
                                pattern, patternName, tipo):
    r"""
       questo metodo effettua sui RE tokenizer

       :param str pattern: il pattern re da testare
       :param str patternName: il nome del patten
    """

    testName = patternName
```

```
dimTests = self.dimTests
tok = nltk.tokenize.RegexpTokenizer (pattern, gaps=False, discard_empty=True, flags=re.DOTALL)
tipo = tipo
attributiTok = {'gap': False, 'discardEmpty': True, 'flags': 're.DOTALL'}
attrfn = u"gap=False discardEmpty=True flags=re.DOTALL"
self.__TestTokenizer (testName, dimTests, tok, tipo, attributiTok, attrfn)
```

TestREWordTok_patter_w_gaps_F_discardEmpty_F_flags_reUNI

In questo metodo viene testato il pattern con le seguenti configurazioni:

- Gaps = False
- Discard_Empty = False
- Flag = re.UNICODE

Il codice del metodo

```
def TestREWordTok_patter_w_gaps_F_discard_empty_F_flags_reUNI (self,
                                pattern, patternName, tipo):
    r"""
       Standard Regular Expression Tokenizer

       questo metodo effettua sui RE tokenizer

       :param str pattern: il pattern re da testare
       :param str patternName: il nome del patten
    """

    testName = patternName
    dimTests = self.dimTests
    tok = nltk.tokenize.RegexpTokenizer (pattern, gaps=False, discard_empty=False, flags=re.UNICODE)
    tipo = tipo
    attributiTok = {'gap': False, 'discard_empty': False, 'flags': 're.UNI'}
    attrfn = u"gap=False discard_empty=False flags=re.UNI"
    self.__TestTokenizer (testName, dimTests, tok, tipo, attributiTok, attrfn)
```

TestREWordTok_patter_w_gaps_F_discardEmpty_F_flags_reMULTI

In questo metodo viene testato il pattern con le seguenti configurazioni:

- Gaps = False
- Discard_Empty = False

- Flag = re. MULTILINE

Il codice del metodo

```
def TestREWordTok_patter_w_gaps_F_discardEmpty_F_flags_reMULTI (self,
                                pattern, patternName, tipo):
    r"""
       questo metodo effettua sui RE tokenizer

       :param str pattern: il pattern re da testare
       :param str patternName: il nome del patten
    """

    testName = patternName
    dimTests = self.dimTests
    tok = nltk.tokenize.RegexpTokenizer (pattern, gaps=False, discard_empty=False, flags=re.MULTILINE)
    tipo = tipo
    attributiTok = {'gap': False, 'discardEmpty': False, 'flags': 're.MULTI'}
    attrfn = u"gap=False discardEmpty=False flags=re.MULTI"
    self.__TestTokenizer (testName, dimTests, tok, tipo, attributiTok, attrfn)
```

TestREWordTok_patter_w_gaps_F_discardEmpty_F_flags_reDOTALL

In questo metodo viene testato il pattern con le seguenti configurazioni:

- Gaps = False
- Discard_Empty = False
- Flag = re. DOTALL'

Il codice del metodo

```
def TestREWordTok_patter_w_gaps_F_discardEmpty_F_flags_reDOTALL (self,
                                pattern, patternName, tipo):
    r"""
       questo metodo effettua sui RE tokenizer

       :param str pattern: il pattern re da testare
       :param str patternName: il nome del patten
    """

    testName = patternName
```

```
dimTests = self.dimTests
tok = nltk.tokenize.RegexpTokenizer (pattern, gaps=False, discard_empty=False, flags=re.DOTALL)
tipo = tipo
attributiTok = {'gap': False, 'discardEmpty': False, 'flags': 're.DOTALL'}
attrfn =  u'' gap=False discardEmpty= False flags=re.DOTALL''
self.__TestTokenizer (testName, dimTests, tok, tipo, attributiTok, attrfn)
```

La classe CreatorePatternRE

Si è ritenuto conveniente sviluppare questa classe per poter utilizzare differenti patterns su cui effettuare i tests, dando da una parte la possibilità di testare tutti i metodi specializzati propri della classe nltk.tokenize.regexp e dall'altro la possibilità espandere l'insieme dei patterns testati. La classe si occupa di registrare il nome, pattern e campo di applicazione (parole o frasi) da utilizzare durante la fase di testing. I dati inseriti vengono salvati nel file RegularExpression.tag

Il metodo WhitespaceTokenizer

Questo metodo utilizza il pattern **r'\s+'**

Il pattern utilizza gli spazi come delimitatori delle parole

È equivalente al metodo from nltk.tokenize.simple.SpaceTokenizer

Il metodo BlanklineTokenizer

Questo metodo utilizza il pattern **r'\s*\n\s*\n\s*'**

Il pattern utilizza ogni sequenza di riga vuota come delimitatori delle frasi

È equivalente al metodo from nltk.tokenize.simple. LineTokenizer

Il metodo WordPunctTokenizer

Questo metodo utilizza il pattern **r'\w+|[^\w\s]+'**

Il pattern divide le parole trattando gli insiemi di caratteri alfabetici e non come singole parole

La classe nltk.tokenize.sexpr tokenizer

Questa classe definisce l'astrazione per un particolare tipo di tokenizzatore usato per trovare le sottostringhe e le espressioni all'interno delle parentesi. Lavora dividendo il testo in una sequenza di sottostringhe, rappresentanti la sottostringa presente nelle parentesi.
Questa classe, essendo specifica non è stata testata dal programma poiché non risulta testabile su un testo di natura generale.

La classe nltk.tokenize.punkt

Questo tipo di tokenizzatore utilizza un algoritmo non supervisionato per svolgere il compito. È possibile parametrizzarlo per la lingua presa in esame, andando a migliorare la sua efficacia.

Il compito si divide perciò in due fasi distinte:

- Addestramento
- Testing

La fase di addestramento dei punkt tokenizers

Durante la fase di addestramento è possibile definire ed espandere i parametri linguistici.

Uno dei parametri fondamentali per questa fase è la definizione del set di abbreviazioni comunemente trovate nei testi, perciò si è ritenuto necessario creare una classe ad – hoc che vada a registrarle in memoria per l'utilizzo in questa fase. Questo compito è svolto dalla classe Abbreviazione.

La Classe Abbreviazione

Questa classe si occupa di estrare campioni di abbreviazioni di dimensioni differenti da il corpus Paisa e dalla risorsa MorphIt. Il processo è effettuato da due metodi differenti: RegistraDaPaisa e RegistraDaMorphIt

Il Codice del metodo RegistraDaPaisa

```
def RegistraDaPaisa (self, dim = -1):
    r"""
        Questo metodo ricerca e registra le abbreviazioni dal corpus paisa


        il parametro dim, se è un intero rappresenta quanti file utilizzare
        se è pari a -1 utilizza tutto il campione
        se è una lista, è la lista contenente il numero di files da utilizzare
        per creare i files di abbreaviazione
```

```
        questo è stato fatto per poter effettuare prove differenti
    """
    abbrs = set ()
    #ciclo su tutte le frasi
    files = glob.glob (self.folderCorpus + u'*.*')
    if dim != -1:
        files = files[:dim]

    for file in files:
        lines = self.tools.LoadFile(file)
        for line in lines:
            line = line.split (u"\t")
            if line[4] == self.ABBR:
                abbrs.add (line[1])
    filename = u"paisa"
    self.SaveAbbrFile (filename, abbrs)
```

Il Codice del metodo RegistraDaMorphIt

```
def RegistraDaMorphIt (self):
    r"""
        Questo metodo ricerca e registra le abbreviazioni da Morphit
    """
    abbrs = set ()
    #Creo il pattern di ricerca
    pattern_abl = r'^ABL+'   #r'^ABL+'
    pat_abl = re.compile(pattern_abl)
    #Leggo morphIt
    print self.fileMorphIt
    for line in self.tools.LoadFile(self.fileMorphIt):
        line = line.split ()
        if len(line) == 3:
            match=re.match(pat_abl, line[2])
            if match:
```

```
            abbrs.add (line[0])
    #salvo le abbreviazioni
    self.SaveAbbrFile (u'morhpit', list(abbrs))
```

I Parametri Impostabili

ABBREV

Questo parametro è rappresentato da un valore intero, il cut-off value indicante quando un token deve essere considerato una abbreviazione.

ABBREV_BACKOFF

Questo parametro è rappresentato da un valore intero, questo parametro rappresenta il upper cut-off, utilizzato dall'algortmo di Mikheev atto ad individuare le abbreviazioni.

COLLOCATION

Questo parametro, espresso da un valore float, rappesenta il minimal log-likelihood value che determina quando due tokens vengono definiti come collocazione.

IGNORE_ABBREV_PENALTY

Questo parametro booleano abilita la abbreviation penalty heuristic. Questa euristica sfavorisce in modo esponenziale le parole che sarebbero considerato come abbreviazioni anche in caso di mancanza del simbolo di periodo (.)

INCLUDE_ABBREV_COLLOCS

Questo parametro booleano considera come potenziali collocazioni tutte le coppie di parole in cui la prima è una abbreviazione. Tale parametro sostituisce l'euristica ortografica ma non la abbreviation penalty heuristic. Queto parametro viene bypassato dal parametro INCLUDE_ALL_COLLOCS. Se entrambi sono di valore falso, tutte le collocazioni con iniziali ordinali sono considerate come collocazione

INCLUDE_ALL_COLLOCS

Questo parametro booleano rappresenta la scelta di includere tutte le coppie di parole (bigrams) di cui la prima termina con un simbolo di periodo (.) . Questo risulta molto efficacie in presenza di corpora in cui si ha un elevato numero di abbreviazioni senza periodo, difficilmente individuabili (ad esempio dott o mr o miss,…),

MIN_COLLOC_FREQ

Questo parametro, espresso da un valore intero, rappresenta il numero di volte minimo per cui un bigram deve apparire prima di essere considerato una collocazione.

SENT_STARTER

Questo parametro, espresso tramite un numero float, rappresenta il minimal log-likelihood value da cui un token è considerato come token di start frequente.

Log – Likelihood - Dunning log-likelihood

Per effettuare la stima del parametro log-likelihood, parametro che definisce quando un bigrams è definibile come collocations, si è sviluppata la classe specifica LogLikelihood. Questa classe utilizza l'algoritmo di Dunning, il quale risulta statisticamente più significativo rispetto agli altri tests statistici, come ad esempio quello del Chi-quadro o F di Fisher. Un'ottima dispensa inerente a questo metodo è reperibile all'indirizzo:

http://research-srv.microsoft.com/pubs/68957/rare-events-final-rev.pdf

e all'indirizzo:

http://aclweb.org/anthology/J93-1003

La classe Loglikelihood

Lo scopo di questa classe è quello di stimare le soglie di loglikelihood, dati macinati dalla classe TestLogLikelihood.

Il codice della classe LogLikelihood

```
class LogLikelihood ():
    r"""
        Questa classe si occupa di effettuare la stima del parametro
        log-likelihood inerente alle collocations
```

```
    da utilizzare come parametro per la creazione dei MyPunkt Tokenizers
"""

def VERSION (self):
    return "vers.3.8.b"


def __init__ (self, n = -1):
    r"""

        :param int n: la dimensione su cui effettuare il calcolo del logl.

    """
    self.folderDati = "dati" + os.path.sep
    self.loglFilename = self.folderDati + "loglikelihood.pickle"
    self.__tools = Tools (n) #default -1, cioè tutto il campione
    self.__col_logl = list ()
    self.queue = Queue.Queue ()

    self.__tools.CaricaCorpus ()



def __FreqFromCorpus (self):
    r"""
        Questo metodo estrae le frequenze dal corpus
    """
    bi = FreqDist(bigrams(self.__tools.words))
    wfr = FreqDist(self.__tools.words)

    #popolo la coda
    for eles in bi.keys():
        a = wfr[eles[0]]
        b = wfr[eles[1]]
        ab = bi[eles]
        N = wfr.N()

        self.queue.put(tuple([a, b, ab, N]))


def __CalcolaLogL (self):
```

```
    r"""
        Questo metodo calcola il LogLikelihood
    """
    while True:
        a, b, ab, N = self.queue.get ()

        self.__col_logl.append (nltk.tokenize.punkt.PunktTrainer()._col_log_likelihood  (a, b, ab, N))

        self.queue.task_done()


#lancio tutti i threads
def __MThreard (self):
    r"""
        Questo metodo lancia in esecuzione i threads per il calcolo
    """
    #numero di threads
    nThread = multiprocessing.cpu_count()

    while not self.queue.empty ():
        #avvio tanti threads quanti sono il numero di processori logici
        #disponibili
        for  i in xrange (nThread):
            t = threading.Thread(target = self.__CalcolaLogL)
            t.daemon = True
            t.start ()


def LogLikelihood (self):
    r"""
        Questa funzione calcola la media dei logl.

        essendo k > 30 / 50 la distribuzione Gamma si approssima alla normale
        quindi sfrutto le proprietà della normale e stabilisco che una
        collocations è tale se rappresenta almeno circa il 30% del campione
    """
    self.__FreqFromCorpus ()
    self.__MThreard ()


    mean = float( sum(self.__col_logl) / len(self.__col_logl))
```

```
        var = sum ([abs(x - mean) for x in self.__col_logl]) / len (self.__col_logl)

        logl = mean - sqrt(var) * 1

        print "mean:", mean
        print "var:%f   sigm:%f" %(var, sqrt(var))
        print "logl:", logl

        #salvo il logl trovato
        self.__tools.SaveByte ([logl], self.loglFilename)

        return [logl]

    def LogLikelihoods (self):
        r"""
            Questo metodo calcola 3 loglikel. in base alle 3 dimensioni disponibili
            3* - tutto il campione
            2* - 2/3 del campione
            1* - 1/3 del campione
        """
        ress = []
        dimcorp = len(glob.glob (self.__tools.folderCorpus + '*.*'))
        #suddivido gli step
        for i in range(1,4):
            #calcolo il logl
            dim = int (i / 3 * dimcorp )
            self.__tools.n = dim
            self.__tools.CaricaCorpus ()
            ress.append (self.LogLikelihood ()[0])
        #registro i dati
        self.__tools.DelFile (self.loglFilename)
        self.__tools.SaveByte (ress, self.loglFilename)

        return ress
```

La classe TestLogLikelihood

Questa classe è stata sviluppata per testare la classe precedente secondo differenti condizioni, al fine di trovare la soluzione ottimale da utilizzare come parametro

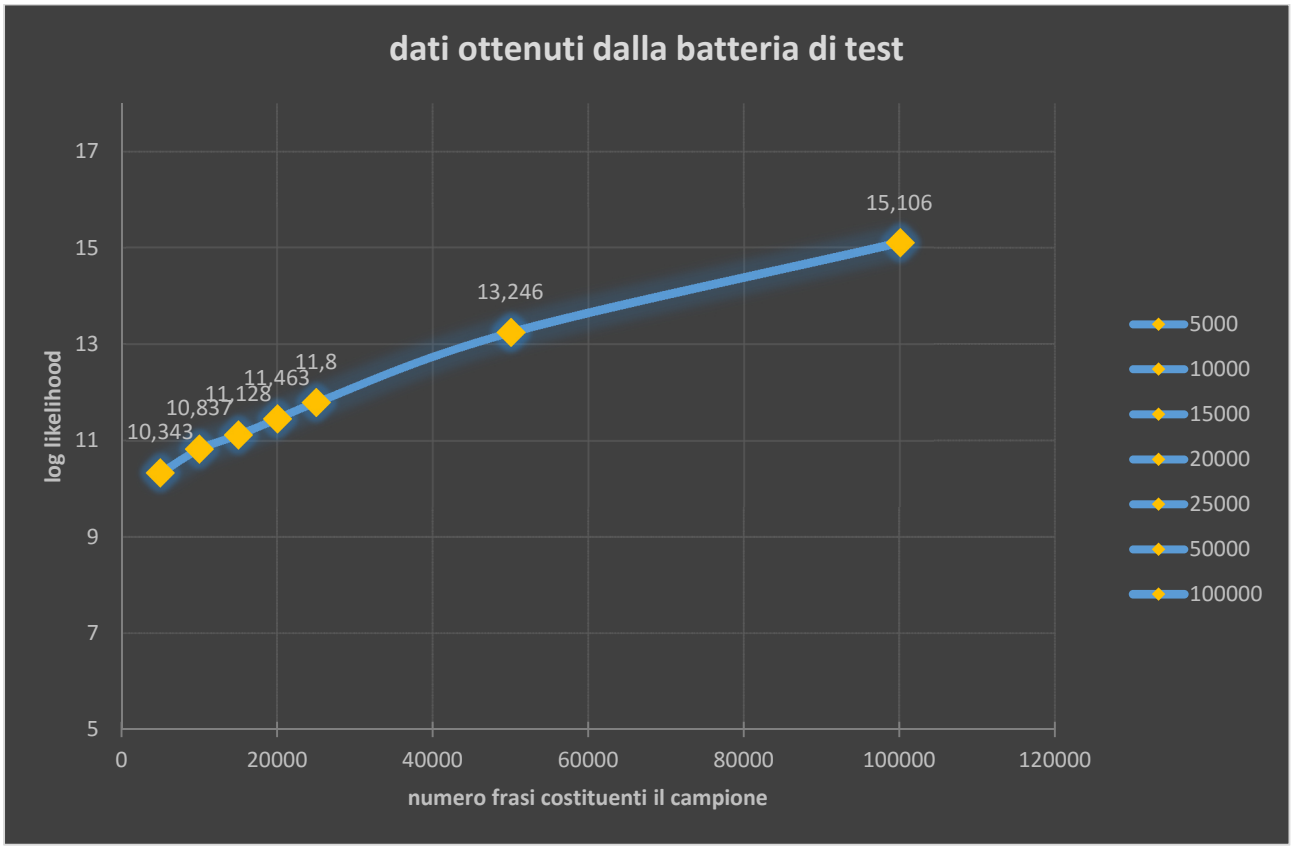
Il Codice della classe TestLogLikelihood

```
class TestLogLikelihood ():
    r"""
        Questa classe modella la batteria di test da effettuare
    """
    def __init__ (self, batterie = [-1], filename = "TestLogLikelihood_newTests.csv"):
        r"""
            :param list batterie: lista contenente le dimensioni dei test
            :param str filename: il nome del file su cui salvare i test

                :return: i risultati dei test
                :rtype: tuple
        """
        self.folderDati = "dati" + os.path.sep
        self.testFilename = self.folderDati + filename

        self.tools = Tools(1)

        self.AvviaTests (batterie)



    def AvviaTests (self, batterie):
        r"""
            Questo metodo avvia tutti i test
        """
        for dim in batterie:
            print "Avvio Test su %d campioni" % dim
            #Effettuo il test
            test = LogLikelihood (n = dim)
            #registro i dati
            filename = self.testFilename
            result = test.LogLikelihood ()

            print "LogLikelihood pari a %f" % result
            #salvo i log trovati
            self.tools.SaveTestCsv (filename = filename,testName = "Loglikelihood",
                                    nSamples = dim, result = result)

            self.tools.SaveByte (result, self.loglFilename)
```

```
print "Tests Eseguiti correttamente"
```

I Risultati del Test

Il grafico seguente riporta i risultati ottenuti da una batteria di test a differenti dimensioni del campione.



I Parametri Linguistici

internal_punctuation

Questo parametro, rappresentato da una stringa, rappresenta tutti i simboli di punteggiatura che possono comparire all'interno di una frase

sent_end_chars

Questo parametro, rappresentato da una tupla, rappresenta tutti i simboli che occupano il ruolo di marcatore forte di fine frase

La creazione dei PunktTokenizers

Per l'addestramento di questo tokenizzatore si è ricorso alla creazione della classe CreatorePunktTokenize.

Tramite questa classe è possibile variare i parametri linguistici da utilizzare durante la fase di training.

La fase di creazione dei PunktTokenizers

Questa fase si occupa di creare ed addestrare il tokenizzatore. Durante la fase di test il tokenizzatore viene creato in modo dinamico, addestrato e testato. Solo il tokenizzatore creato con i parametri stimati migliori verrà salvato (se ne salverà la serializzazione, cioè il pickle). Questa fase è svolta dal metodo __CreaTok della classe CreatorePunktTokenizer

_CreaTok

Questo metodo è il core della classe. Qui è dove avviene la parametrizzazione e l'addestramento vero e proprio del tokenizzatore.

Per prima cosa viene creato l'oggetto che ospiterà i parametri linguistici. Questo oggetto è modellabile, parametrizzando la classe nltk.tokenize.punkt.PunktLanguageVars

Successivamente, una volta impostati i parametri linguistici, si crea un oggetto contenitore, grazie alla classe nltk.tokenize.punkt.PunktBaseClass il quale sarà utilizzato dall'oggetto che si occuperà di effettuare il training.

La fase di training è stata suddivisa in tre momenti separati:

- Addestramento delle abbreviazioni
- Addestramento sul corpora
- Finalizzazione dell'addestramento

L'oggetto di trainer così ottenuto verrà utilizzato dal tokenizzatore come parametrizzazione dello stesso.

Questo è reso possibile passandolo come parametro all'atto della creazione di quest'ultimo.

punktTok = nltk.tokenize.punkt.PunktSentenceTokenizer (trainer.get_params ())

Il Codice del metodo __CreaTok

```
def _CreaTok (self, internal_punctuation, end_sent_punct,
        abbrsFilename, ABBREV,
        IGNORE_ABBREV_PENALITY,     #·controllare bene i parametri in ingresso
        ABBREV_BACKOFF, COLLOCATION,
        SENT_STARTER, INCLUDE_ALL_COLLOCS,
        INCLUDE_ABBREV_COLLOCS, MIN_COLLOC_FREQ):
    r"""
      Questa funzione, dati i parametri in ingresso crea il tokenizzatore
```

```
"""
#Language Params
#inizio creando gli oggetti del punkt
languageParam = nltk.tokenize.punkt.PunktLanguageVars
#internal punctuactions    - type string  -
languageParam.internal_punctuation = internal_punctuation
#end sent punctuactions    - type tuple
languageParam.sent_end_chars = end_sent_punct
#Punkt Parameters
punktParameters = nltk.tokenize.punkt.PunktParameters()
#Base Class
baseClass = nltk.tokenize.punkt.PunktBaseClass (lang_vars=languageParam,  params=punktParameters)
#Trainer
#Effettuo il training delle abbreviazioni
if not abbrsFilename:
   abbrs = self.Dabbs
abbrs = [ele.strip() for ele in self.tools.LoadByte(abbrsFilename)]
abbrs = "\n".join (abbrs)
abbrs = "Start" + abbrs  + "End."
try:
   trainer = nltk.tokenize.punkt.PunktTrainer (abbrs, baseClass)
except ValueError:
   return -1
#fase 2 aggiungo i parametri
trainer.ABBREV = ABBREV
trainer._IGNORE_ABBREV_PENALITY = IGNORE_ABBREV_PENALITY
trainer.ABBREV_BACKOFF = ABBREV_BACKOFF
trainer.COLLOCATION = COLLOCATION
trainer.SENT_STARTER = SENT_STARTER
trainer.INCLUDE_ALL_COLLOCS = INCLUDE_ALL_COLLOCS
trainer.INCLUDE_ABBREV_COLLOCS = INCLUDE_ABBREV_COLLOCS
trainer.MIN_COLLOC_FREQ = MIN_COLLOC_FREQ

return trainer
```

La fase di Stima dei parametri

Per effettuare la stima dei parametri migliori con cui addestrare il tokenizzatore si effettuano una serie di cicli di test. Durante ogni ciclo si provano tutte le varianti di un parametro; grazie al metodo Best si identifica quale variante risulta essere la più performante.

La fase di Test dei MyPunktTokenizers

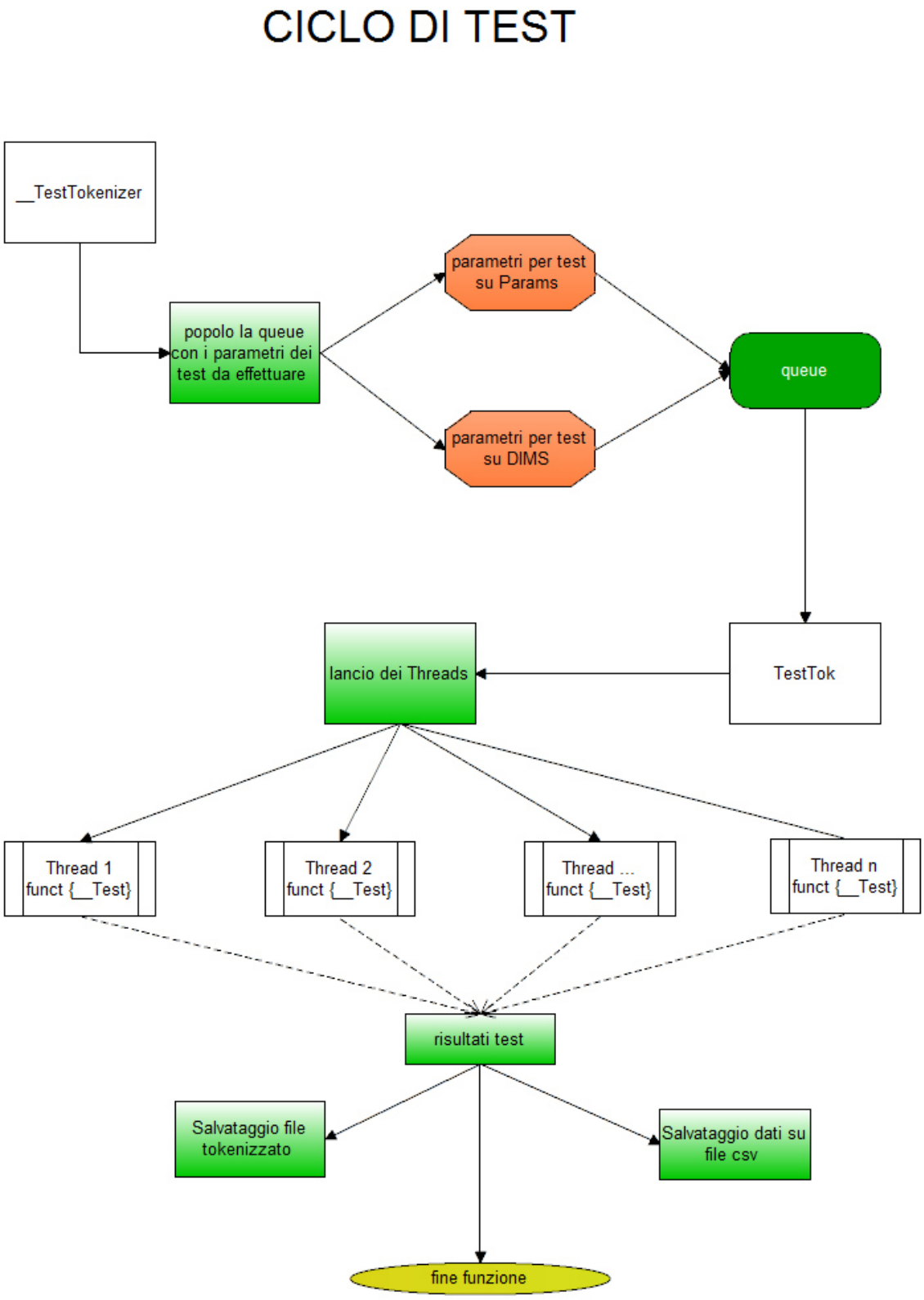
Il primo ciclo di test viene effettuato per determinare quale sia la dimensione di training migliore, utilizzando i parametri di default.

Durante il ciclo successivo verrà effettuata la stima dei parametri linguistici

Infine, viene effettuato un ciclo per determinare nuovamente la dimensione del campione di training dati i parametri appena stimanti

A questo punto, tutti i parametri sono stati stimati e si procede con l'esecuzione dei test su questo tokenizzatore.

Breve schema logico di funzionamento del processo di stima dei parametri



Il Codice del metodo MyPunkt

```
  def TestMyPunkt (self):
      r"""
         questo metodo effettua i tests sui Punkt Tokenizers
      """
      def CreaTokenizzatore (dimTraining, params):
         r"""
            Questo metodo crea il tokenizzatore con i parametri richiesti
            :param int dimTraining: la dimensione di training del tok
            :param tuple params: la tupla contenente tutti i parametri con cui modellare il tok
            :return: il tokenizzatore modellato
            :rtype: tok
         """
         obj = Tools (dimTraining)
         obj.CaricaCorpus (folder = self.folderCorpusTraining)
         #test mode
         #obj.CaricaCorpus ()
         return MyPunktTokenize().CreaMyPunkt (obj.CreaPlainText (self.simpleParamS, self.simpleParamW),
*params)


      def TestTagsTok (testName, dim, tok, registra = False, paramTest = None):
         r"""
            Questo metodo effettua il ciclo dei test sulla dimensione Params
            è utilizzato durante la fase di stima dei parametri
            Questo metodo deve restituire solo True o False
            o list(tuple(paramTest, score))
            :param str testName: nome del tok
            :param int dim: dimensione di training
            :param tok tok: tokenizzatore da testare
            :param bool registra: flag che indica se il test è da registrare
            :param     paramTest: il parametro in fase di test

            :return: i risultati dei test
            :rtype: tuple
         """
         tupleScores = list ()
         #Oggetto per il corpus
         corpusObj = Tools (dim)
         corpusObj.CaricaCorpus ()
```

```
        for paramS in self.paramCorpusCreationS:
            for paramW in self.paramCorpusCreationW:
                datiOut =  tok.tokenize (corpusObj.CreaPlainText (paramS, paramW))
                r, score = self.tools.RisultatiTest(testName, datiOut, self.tools.SENT, corpusObj.words,
corpusObj.corpusLst, tag = paramS)
                tupleScores.append (tuple([paramTest, score]))
        return tupleScores


    def TestDimsTok (testName, dims, tok, registra = False, paramTest = None):
        r"""
            Questo metodo effettua il ciclo dei test sulla dimensione Dims
            è utilizzato durante la fase di stima dei parametri
            Questo metodo deve restituire solo True o False
            o list(tuple(paramTest, score))
            :param str testName: nome del tok
            :param int dim: dimensione di training
            :param tok tok: tokenizzatore da testare
            :param bool registra: flag che indica se il test è da registrare
            :param     paramTest: il parametro in fase di test

            :return: i risultati dei test
            :rtype: tuple
        """
        tupleScores = list ()
        for dim in dims:
            corpusObj = Tools (dim)
            corpusObj.CaricaCorpus ()
            datiOut =  tok.tokenize (corpusObj.CreaPlainText (self.normalParamS, self.normalParamW))
            r, score = self.tools.RisultatiTest(testName, datiOut, self.tools.SENT, corpusObj.words,
corpusObj.corpusLst, tag = self.normalParamS)
            tupleScores.append (tuple([paramTest, score]))
        return tupleScores


    def Best (tuplaScores):
        """
          Questa funzione restituisce il primo parametro della tupla migliore
            :param list(tuple) tuplaScores: lista di tuple (param, score)
            :return: il parametro con il punteggio più alto
            :rtype: param
        """
```

```
    best = (0, 0)
    for i in tuplaScores:
        if best[1] < i[1]:
            best = i
    return best[0]

# questi due arrays vengono passati direttamente dall'oggetto creatore MyPunktTokenize
parametri = MyPunktTokenize().GetAllParams ()  # contiene tutte le varianti dei parametri
params = MyPunktTokenize().GetStandardParams ()  #all'avvio del metodo continene i parametri standard

########PARTE STANDARD DEL TESTS  #################
#per prima cosa addestro e testo il tokenizzatore con i parametri standard
#Stimo il dimensionamento di training migliore, calcolato sul primo valore delle dimensioni dei test
#questi parametri servono per effettuare almeno una volta il ciclo
precScore = 0
attScore = self.sogliaMiglioramento
dim = self.passoTraining
dimsent = DimSamplesPunkt().nSents (dim)
nsentprec = 0 #questa var mi serve per controllare di non eccedere oltre le dimensioini del corpus di training
print "inizio euristica miglioramento default punkt"
while self.EuristicaMiglioramento (precScore, attScore):
    precScore = attScore
    if nsentprec == dimsent:
        break
    testName =   u"DEFAULT PUNKT TOKENIZER"
    #creo il tokenizzatore
    tok = CreaTokenizzatore (dimsent, params)
    if tok == -1:
        continue
    tipo = self.tools.SENT
    attributiTok =  {'dimTrainingWords': dim}
    attrfn = unicode(dim)
    attScore, test_ = self.__Test (testName, tok, self.dimTests[0],
            self.normalParamS, self.normalParamW, self.TIPO_DIMENS,
            self.tools.SENT, attributiTok, attrfn)
    nsentprec = dimsent
    #precScore = attScore
    dim = dim + self.passoTraining
    dimsent = DimSamplesPunkt().nSents (dim)
print "fine stima dimensione di training"
print "inizio test su default punkt tok"
```

```
#Effettuo e registro il test con i parametri standard
testName =   u"DEFAULT PUNKT TOKENIZER"
#effettuo i test
dimTests = self.dimTests
#creo il tokenizzatore
tok = CreaTokenizzatore (dimsent, params)
tipo = self.tools.SENT
attributiTok =  {'dimTrainingWords': dim}
attrfn = unicode(dim)
self.__TestTokenizer (testName, dimTests, tok, tipo, attributiTok, attrfn)

#ora inizio a cercare i parametri migliori per il tokenizzatore
#utilizzando come dimensione di training la minore
print "ciclo di tests per la stima dei parametri migliori"
# ciclo su tutti i parametri
#j è posizione su params
j=0
while j < len (parametri):
   #azzero le variabili interne del ciclo
   # ciclo su tutte le varianti del parametro
   tmpParams = list ()
   tmpDims = list ()
   for iParam in parametri[j].keys():
      #modifico un parametro
      params[j] = parametri[j][iParam]
      # Creo il tok
      tok = CreaTokenizzatore (dimsent, params)
      if tok == -1:
         print "Parametri non utilizzabili per creare il tokenizzatore"
         print params
         continue
      paramTest = parametri[j][iParam]
      #test Params
      tmpParams.extend (TestTagsTok (testName, self.dimTests[0], tok, False, paramTest = paramTest))
      #test Dims
      tmpDims.extend (TestDimsTok (testName, self.dimTests, tok, False, paramTest = paramTest))

   #sostituisco il parametro migliore nell'array dei parametri da passare all'atto della creazione
   scoreBestParams ={ele[1]:ele[0] for ele in tmpParams}
   scoreBestDims ={ele[1]:ele[0] for ele in tmpDims}
```

```
    if max(scoreBestParams.keys()) >= max(scoreBestDims.keys()):
        params[j] = scoreBestParams[max(scoreBestParams.keys())]
    else:
        params[j] = scoreBestDims[max(scoreBestDims.keys())]
    j += 1
#giunto a questo punto ho stimato i parametri ottimali per il tokenizzatore
# l'ultimo test dovrebbe essere  quello con i risultati migliori

#Calcolo la dimensione di Training Migliore
precScore = 0
attScore = self.sogliaMiglioramento
dim = self.passoTraining
dimsent = DimSamplesPunkt().nSents (dim)
nsentprec = 0 #questa var mi serve per controllare di non eccedere oltre le dimensioini del corpus di training
print "inizio stima dim training my punkt"
while self.EuristicaMiglioramento (precScore, attScore):
    precScore = attScore
    if nsentprec == dimsent:
        break
    testName =   u"MY PUNKT TOKENIZER"
    #creo il tokenizzatore
    tok = CreaTokenizzatore (dimsent, params)
    tipo = self.tools.SENT
    attributiTok =  {'dimTrainingWords': dim}
    attrfn = unicode(dim)
    attScore, test_ = self.__Test (testName, tok, self.dimTests[0],
          self.normalParamS, self.normalParamW, self.TIPO_DIMENS,
          self.tools.SENT, attributiTok, attrfn)
    nsentprec = dimsent
    dim = dim + self.passoTraining
    dimsent = DimSamplesPunkt().nSents (dim)
print "fine stima dimensione di training"
print "inizio test my punkt"
#Test con il dimensionamento del tok già effettuato dall'istr prec
testName = u"MY BEST PUNKT TOKENIZER"
#creo il tokenizzatore
tok = CreaTokenizzatore (dimsent, params)
#salvo il tokenizzatore
tokfilename = self.folderPunkt + testName + self.fileExtPnkt
self.tools.SaveByte (tok, tokfilename)
#devo implementare tokenize method!!!!
```

```
tipo = self.tools.SENT
attributiTok =  {'dimTrainingWords': dim}
attrfn = unicode(dimsent)
self.__TestTokenizer (testName, [self.dimTests[0]], tok, tipo, attributiTok, attrfn)
```

La classe nltk.tokenize.texttiling

La classe texttiling rappresenta l'astrazione di un oggetto atto a suddividere il documento passato come parametro in ingresso in sotto argomenti (subtopic), utilizzando l'algoritmo TextTiling. Questo particolare algoritmo è in grado di trovare suddividere il testo tramite l'analisi delle co-occorenze ottenute tramite l'analisi lessicale.

Il processo parte suddividendo il testo in pseudofrasi di lunghezza fissa w. Viene assegnato uno score di similarità tra le pseudo frasi.

L'algoritmo procede computando le differenze tra gli scores ottenuti nella fase precedente, definendo in questo modo i boundaries.

I boundaries (confini) trovati vengono normalizzati e definiscono il paragrafo contenente il topic.

Infine, tutti i dati calcolati vengono restituiti come risultato del processo.

I Parametri del tokenizzatore

- **w** rappresenta la dimensione (prefissata), intesa come lunghezza in numero di parole, delle pseudofrasi
- **k** rappresenta il numero di frasi utilizzato durante la fase di comparazione tra i blocchi di pseudofrasi
- **similarity_method** questa costante indica all'oggetto quale metodo utilizzare per determinare la similarità degli score:
    - *BLOCK_COMPARISON*
    - *VOCABULARY_INTRODUCTION*
- **Stopwords** questo parametro rappresenta una lista di stopwords, utilizzate per filtrare il testo in ingresso per ridurre la complessità computazionale
- **smoothing_method** rappresenta il metodo usato per effettuare la normalizzazione degli scores
- **smoothing_width** rappresenta la dimensinoe della finestra usata per la normalizzazione degli scores
- **smoothing_rounds** questa parametro rappresenta il numero di passaggi di normalizzazione degli score da effettuare
- **cutoff_policy** questo parametro è utilizzato per determinale la policy (il procedimento) di determinazione dei boundaries. Può essere:
    - HC
    - LC

Stopwords

Le stopwords sono un elenco di parole che si trovano con una frequenza maggiore in tutti i documenti. Data questa caratteristica, risultano inversamente proporzionali al loro grado informativo all'interno del documento stesso in cui appaiono. Questo elenco può essere costruito secondo il dominio di applicazione oppure direttamente stilato in base alle caratteristiche stesse dei suoi componenti.

Essenzialmente esistono tre modi distinti di formare l'elenco delle stopwords:

1. I termini più frequenti
2. Dominio specifico
3. Termini con IDF (Inverse document frequency) basso

Stopwords come termini più frequenti

Per stimare le stopwords da un corpora, si considera come caratteristica comune di appartenenza, la frequenza elevata. In altre parole più volte un termine appare nel corpora e maggiore saraà la sua frequenza. Per calcolare ciò per prima cosa si andrà a calcolare la frequenza di ogni parola, successivamente si manterranno solo le prime n parole.

Per svolgere questo compito per la lingua italiana, si è ritenuto necessario scrivere una classe si che occupi di maneggiare questo compito. La classe presa in esame è ItalianStopWords. Il metodo che assolve questo compito è StopWordsFrequenza.

Stopwords Dominio Specifico

Ogni elemento appartenente a questo elenco condivide con gli altri alcune caratteristiche morfosintattiche specifiche. Per il compito in esame si sono scelte come caratteristiche:

- Articoli
- Preposizioni
- Congiunzioni

Questo compito è svolto dal metodo StopWordsDomainSpecific.

Stopwords con IDF basso

Come discriminante di appartenenza all'elenco delle stopwords si considera un valore molto basso della caratteristica IDF

La Classe ItalianStopwords

La classe ItalianStopWords è stata scritta per derivare le stopwords dal due differenti risorse: Paisà e Morphit. Mentre la prima risorsa è utilizzata per stimare le stopwords dato il valore di frequenza e dato il valore Idf, la seconda risorsa è stata impiegata per la stima date le caratteristiche morfologiche.

All'interno della classe i tre metodi che si occupano di effettuare i calcoli sono :

- StopWordsDomainSpecific
- StopWordsFrequenza
- StopWordsIDF

Il Codice del metodo StopWordsDomainSpecific

```
def __StopWordsDomainSpecific (self):
    r"""
        Questo metodo si occupa di stimare le stopwords in base alle
        caratteristiche morfosintattiche

        Si utilizza la base di dati morphIt per estrarre i dati in
        base alle seguenti caratteristiche:
        - Determiners
        - Coordinating conjunctions
        - Prepositions
    """

    #compilo i patterns re
    #Articoli
    pat_art = re.compile(r'^ART+')
    #Congiunzioni
    pat_con = re.compile(r'^CON+')
    #Preposizioni
    pat_pre = re.compile( r'^PRE+')
    patts = [pat_art, pat_con, pat_pre ]
    stopws = set ()
    #Leggo morphIt
    for line in self.tools.LoadFile(self.morphItFileName):
        line = line.split ()
```

```
            if len(line) == 3:
                #verifico se la parola presa in esame appartiene al gruppo di stopwords
                for p in patts:
                    m = re.match(p, line[2])
                    if m:
                        stopws.add (line[0])
        stopws = list (stopws)
        #salvo le abbreviazioni
        filename = self.folderDati + "stopwordsDominSpec" + self.fileExtStopW
        self.__SaveStopWords (filename = filename, stopwords = stopws)
```

Il Codice del metodo StopWordsFrequenza

```
    def __StopWordsFrequenza (self):
        r"""
            Questo metodo si occupa di stimare le stopwords, stimandole in
            base alla loro frequenza assoluta rispetto al corpus analizzato
        """

        self.tools.CaricaCorpus ()
        #calcolo la distribuzione di frequenza del corpus
        freqs = nltk.FreqDist ([w.lower() for w in self.tools.words])

        #candidati = [w for w in freqs.keys() if IsAlpha(w) and freqs.freq(w)>0]
        #traformo le distribuzione in tuple  - usato in fase di sviluppo del metodo
        tupleAleatorie = [tuple([k, freqs[k]]) for k in freqs.keys()]# if k in candidati]
        mean = sum([x[1] for x in tupleAleatorie]) / len(tupleAleatorie)

        #calcolo la dispersione dei dati intorno alla media
        #e utilizzo lo scostamento come limite per determinare i valori potenziali come stop words

        #calcolo sigma e var
        scartiQuadratici = [pow(x[1] - mean,2) for x in tupleAleatorie]
        #print scartiQuadratici

        scartoQuadraticoMedio = sqrt (sum(scartiQuadratici) / len(tupleAleatorie))
        print "scarto quadratico medio:", scartoQuadraticoMedio
```

```
print "media:", mean
s1 = mean + scartoQuadraticoMedio * 1
s2 = mean + scartoQuadraticoMedio * 2
s3 = mean + scartoQuadraticoMedio * 3
zs1p=[x[0] for x in tupleAleatorie if x[1]>=s1 and IsAlpha(x[0])]
zs2p=[x[0] for x in tupleAleatorie if x[1]>=s2 and IsAlpha(x[0])]
zs3p=[x[0] for x in tupleAleatorie if x[1]>=s3 and IsAlpha(x[0])]
#salvo le abbreviazioni
filename = self.folderDati + "stopwordsFreq_sigma_1" + self.fileExtStopW
self.__SaveStopWords (filename = filename, stopwords = zs1p)
filename = self.folderDati + "stopwordsFreq_sigma_2" + self.fileExtStopW
self.__SaveStopWords (filename = filename, stopwords = zs2p)
filename = self.folderDati + "stopwordsFreq_sigma_3" + self.fileExtStopW
self.__SaveStopWords (filename = filename, stopwords = zs3p)
```

StopWordsIDF

Per assolvere questo compito è stata sviluppata una classe speficica, la classe IDF.

La Classe IDF

Tramite la modifica all'ultima riga del metodo SelezionaStopWords è possibile definire quale metodologia di selezione utilizzare, o un metodo proporzionale rispetto alle dimensioni del corpo o uno selettivo in base alla dispersione dei valori rispetto alla media, delineando come limite di selezione i valori estremi.

Di default si è scelto la seconda metodologia.

Il Codice della Classe IDF

```
class IDF():
def __init__(self, n = -1):
    self.folderCorpus = "corpus" + os.path.sep
    self.tools = Tools(n = n)
    self.D = 0
    self.KD = collections.defaultdict (int)
    self.idfs = dict ()
    self.tools.CaricaCorpus()
```

```
def AvviaCalcoli (self):
    self.VocsDocs ()
    self.IDFs ()
def VocsDocs (self):
    docs = self.CreaDocs (self.tools.sents)
    self.D =len(docs)
    for doc in docs:
        for voc in set(doc):
            if IsAlpha (voc):
                self.KD[voc.lower()] += 1
def IDFs (self):
    r"""
        Questo metodo computa il valore idf per ogni termine
    """
    for voc in self.KD.keys ():
        self.idfs[voc] = self.CalcolaIdf (self.D, self.KD[voc])
def CalcolaIdf (self, D, kd):
    return log(D / kd)
def CreaDocs (self, sents):
    down = float (0)   #uso float per evitare overflow
    up = float (0)
    PASSO = 500
    up = PASSO
    docs = list ()  #lista di documenti
    while up <= len (sents):
        doc = list ()
        for j in xrange(PASSO):
            doc.extend ([w.lower() for w in sents[j + down]])
        docs.append (doc)
        down = up
        up += PASSO
    else:
        #aggiungo l'ultima frase
        doc = list ()
        for j in xrange(len (sents) - down):
            doc.extend ([w.lower() for w in sents[j + down]])
        docs.append (doc)
    return docs

def SelezionaStopWords (self, metodo = 1, val = 1):
    r"""
```

```
        Questo metodo calcola le stopwords
        sicuramente valori di log pari a 0 sono stopwords perchè presenti in
        tutti i documenti, le altre da stimare sono quelle sotto la soglia
    """
    def Metodo_1 (percentuale):
        """ percentuale rappresenta la porzione di testi in cui una parola
            deva comparire per essere considerata stopwords
        """
        LIMITE = log (self.D * (1 - percentuale))
        stopws = list ()
        for voc in self.idfs.keys ():
            c = self.idfs[voc]
#             print "%s : %f" % (voc, c)
            if c <= LIMITE:
                stopws.append (voc)
        return stopws
    def Metodo_2 (nVar):
        """
            nVar rappresenta l'indice di quale limite scegliere
        """
        mean = sum([self.idfs[voc] for voc in self.idfs.keys ()]) / len(self.idfs.keys ())
        scartiQuadratici = [pow(self.idfs[voc] - mean,2) for voc in self.idfs.keys ()]
        scartoQuadraticoMedio = sqrt (sum(scartiQuadratici) / len(self.idfs.keys ()))
        s = mean - scartoQuadraticoMedio * int(nVar)
        stopws = [voc for voc in self.idfs.keys() if self.idfs[voc] <= s]
        return stopws
    if metodo == 1:
        return Metodo_1 (val)
    elif metodo == 2:
        return Metodo_2 (val)
```

Le stima delle Stopwords

Per effettuare la stima da tutti gli elenchi formati grazie alla classe e i metodi precedenti si è reso necessario scrivere la classe specifica ConfrontaStopwords.

Il Codice della classe ConfrontaStopwords

```
class ConfrontaStopwords ():
```

```
def __init__ (self, filenameStopwords = "ItalianStopwords", perc = 0.75):
    self.fileNameStopwords = filenameStopwords
    self.perc = perc
    self.folderDati = u"dati" + os.path.sep
    self.fileExtStopW = u".stopWords"
    self.tools = Tools (1) # uso solo gli strimenti di tools
    stp = self.Confronta (self.LoadStopwords (), escludiNltk = True)
    self.PrintStp (stp)
    self.SalvaFileStopwords(stp)
    print "\nProcesso di selezione terminato con successo"
def LoadStopwords (self):
    r"""
        Questo metodo carica tutti i files ottenuti ed effettua un confronto
        tra le liste
    """
    stopwords = dict ()
    for file in glob.glob (self.folderDati + '*' + self.fileExtStopW):
        print file
        k = os.path.basename(file)[:-len (self.fileExtStopW)]
        stopws = self.tools.LoadByte (file)
        stopwords[k] = stopws
        #old
        #stopwords[k] = set (stopws)
    return stopwords
def Confronta (self, italianStopwords = None, escludiNltk = False):
    r"""
        Questo metodo effettua una selezione tra gli elenchi delle stopwords
        precedentemente calcolati.
        Considera una parola come stopwords solo se è presente in almento una
        percentuale PERC nei files
    """

    meanStopws = collections.defaultdict (int)
    if not escludiNltk:
        for w in nltk.corpus.stopwords.words ("italian"):
            meanStopws [w.lower()] += 1
    for k in italianStopwords.keys ():
        for w in italianStopwords[k]:
            meanStopws [w.lower()] += 1
    stp=[]
    #Considero stopwords solo quelle che sono presenti in almeno il x % dei casi
```

```
        mean = sum([1 for k in italianStopwords.keys () if italianStopwords[k] != list ()]) * (1 - self.perc)
        for w in meanStopws.keys() :
            if meanStopws[w] >= mean:
                stp.append(meanStopws[k])
        stp = [w for w in meanStopws.keys() if meanStopws[w] >= mean]
        return stp
    def PrintStp (self, stps):
        print "Le Stopwords stimate sono"
        for i in stps:
            print i
        print "dimensione:", len(stps)
    def SalvaFileStopwords (self, dati):
        filename = self.folderDati + self.fileNameStopwords + self.fileExtStopW
        self.tools.DelFile (filename)
        self.tools.SaveByte (filename = filename, dati = dati)
```

La fase di Test del TestTextTiling

La fase di test di questo particolare tipo di tokenizzatore è composta da più sottofasi successive.

Per prima cosa effettuo un test generale del tokenizzatore con i parametri di default. Lo scopo di questa sessione è di determinare se il tokenizzatore è applicabile ad ogni tipologia di corpus presa in esame.

Successiva a questa fase vi è il ciclo di stima dei parametri migliori. La logica di funzionamento segue lo schema del mypunktTokenizer; effettuo un ciclo di test per ogni parametro con la finalità di individuare i parametri più performanti

Una volta stimati tutti i parametri si passa alla fase di testing vera e propria del tokenizzatore.

Il Codice del metodo TextTilingTokenizer

```
    def AvviaTestTextTilingTokenizer (self):
        r"""
            questo metodo effettua il test sul TextTiling Tokenizer
            Questo metodo crea e avvia i test sul texttililng tokenize
            :param str corpus: il corpus su cui testare il tokenizzatore
        """
        #utilizzo la stessa procedura di stima dei parametri del mypunkt tokenizer
        def CreaTokenizzatore (params):
            r"""
                Questo metodo crea il tokenizzatore con i parametri richiesti
                :param tuple params: la tupla contenente tutti i parametri con cui modellare il tok
```

```
            :return: il tokenizzatore modellato
            :rtype: tok
        """
        return TextTiling().CreaTextTilingTokenizer (*params)
    def TestTagsTok (testName, dim, tok, registra = False, paramTest = None):
        r"""
            Questo metodo effettua il ciclo dei test sulla dimensione Params
            è utilizzato durante la fase di stima dei parametri

            Questo metodo deve restituire solo True o False
            o list(tuple(paramTest, score))
            :param str testName: nome del tok
            :param int dim: dimensione di training
            :param tok tok: tokenizzatore da testare
            :param bool registra: flag che indica se il test è da registrare
            :param     paramTest: il parametro in fase di test

            :return: i risultati dei test
            :rtype: tuple
        """
        tupleScores = list ()
        #Oggetto per il corpus
        corpusObj = Tools (dim)
        corpusObj.CaricaCorpus ()
        for paramS in self.paramCorpusCreationS:
            for paramW in self.paramCorpusCreationW:
                try:
                    datiOut =  tok.tokenize (corpusObj.CreaPlainText (paramS, paramW))
                    r, score = self.tools.RisultatiTest(testName, datiOut, self.tools.SENT, corpusObj.words,
corpusObj.corpusLst, tag = paramS)
                    tupleScores.append (tuple([paramTest, score]))
                except:
                    #se il tokenizzatore non è applicabile al testo
                    tupleScores.append (tuple([paramTest, 0.0]))
        return tupleScores

    def TestDimsTok (testName, dims, tok, registra = False, paramTest = None):
        r"""
            Questo metodo effettua il ciclo dei test sulla dimensione Dims
            è utilizzato durante la fase di stima dei parametri
```

```
            Questo metodo deve restituire solo True o False
            oppure list(tuple(paramTest, score))

            :param str testName: nome del tok
            :param int dim: dimensione di training
            :param tok tok: tokenizzatore da testare
            :param bool registra: flag che indica se il test è da registrare
            :param     paramTest: il parametro in fase di test

            :return: i risultati dei test
            :rtype: tuple
        """
        #in questo caso per passare deve superare l'euristica delle prestazioni medie
        tupleScores = list ()
        for dim in dims:
            corpusObj = Tools (dim)
            corpusObj.CaricaCorpus ()
            try:
                datiOut =  tok.tokenize (corpusObj.CreaPlainText (self.normalParamS, self.normalParamW))
                r, score = self.tools.RisultatiTest(testName, datiOut, self.tools.SENT, corpusObj.words,
corpusObj.corpusLst, tag = self.normalParamS)
                tupleScores.append (tuple([paramTest, score]))
            except:
                #se il tokenizzatore non è applicabile al testo
                tupleScores.append (tuple([paramTest, 0.0]))
        return tupleScores


    def Best (tuplaScores):
        """
        Questa funzione restituisce il primo parametro della tupla migliore

            :param list(tuple) tuplaScores: lista di tuple (param, score)

            :return: il parametro con il punteggio più alto
            :rtype: param
        """

        best = (0, 0)
        for i in tuplaScores:
            if best[1] < i[1]:
```

```
            best = i
    return best[0]

# questi due arrays vengono passati direttamente dall'oggetto creatore MyPunktTokenize
parametri = TextTiling().GetAllParams ()  # contiene tutte le varianti dei parametri
params = TextTiling().GetStandardParams ()  #all'avvio del metodo continene i parametri standard

########PARTE STANDARD DEL TESTS  #################
#per prima cosa addestro e testo il tokenizzatore con i parametri standard
testName =   u"DEFAULT TEXTITILING TOKENIZER"
#creo il tokenizzatore
tok = CreaTokenizzatore (params)
tipo = self.tools.SENT
attributiTok =  {'params default': params}
attrfn = "_default_params"
#inserita espressione di controllo - dato che questo tipo di tokenizzatore è applicabile solo
# ad un campo ristretto di tipologie di testo
if not self.__TestTokenizer (testName, self.dimTests, tok, tipo, attributiTok, attrfn):
    print "Fine funzione di test ", testName
    return

print "fine prima parte test"

#ora inizio a cercare i parametri migliori per il tokenizzatore
print "ciclo di tests per la stima dei parametri migliori"
# ciclo su tutti i parametri
#j è posizione su params
j = 0
while j < len (parametri):
    #azzero le variabili interne del ciclo
    # ciclo su tutte le varianti del parametro
    tmpParams = list ()
    tmpDims = list ()
    for iParam in parametri[j].keys():
        #modifico un parametro
        params[j] = parametri[j][iParam]
        # Creo il tok
        tok = CreaTokenizzatore (params)
        paramTest = parametri[j][iParam]
        #test Params
        tmpParams.extend (TestTagsTok (testName, self.dimTests[0], tok, False, paramTest = paramTest))
```

```
            #test Dims
            tmpDims.extend (TestDimsTok (testName, self.dimTests, tok, False, paramTest = paramTest))


        #sostituisco il parametro migliore nell'array dei parametri da passare all'atto della creazione
        scoreBestParams ={ele[1]:ele[0] for ele in tmpParams}
        scoreBestDims ={ele[1]:ele[0] for ele in tmpDims}
        if max(scoreBestParams.keys()) >= max(scoreBestDims.keys()):
            params[j] = scoreBestParams[max(scoreBestParams.keys())]
        else:
            params[j] = scoreBestDims[max(scoreBestDims.keys())]
        j += 1
    #giunto a questo punto ho stimato i parametri ottimali per il tokenizzatore
    # l'ultimo test dovrebbe essere  quello con i risultati migliori
    #testo con i parametri stimati
    testName =   u"MY TEXTITILING TOKENIZER"
    #creo il tokenizzatore
    tok = CreaTokenizzatore (params)

    tipo = self.tools.SENT
    attributiTok =  {'my params': params}
    attrfn = "_my_params"
    self.__TestTokenizer (testName, self.dimTests, tok, tipo, attributiTok, attrfn)
```

Note dei pre-tests

Come risultante dai tests preliminari, questa tipologia di tokenizzatore non risulta adatta allo scopo di questa ricerca, ma si è scelto di includerlo per testarne l'efficacia durante il processo di testing con un numero elevato di frasi costituenti il corpus. Ci si aspetta che l'oggetto in questione dia risultati solo con il parametro di ricostruzione del corpus tagS pari a PARG_2

La classe nltk.tokenize.causal.TweetTokenizer

TweetTokenizer

Questa particolare classe è stata sviluppata per processare in modo efficiente i posts provenienti da Tweeter.

Questo tipo di tokenizzatore opera sequenzialmente il post passato in ingresso tramite operazioni di Regular Expression

Per prima cosa elimina i caratteri di markup del codice HTML, successivamente identifica e raggruppa gli insiemi di simboli rappresentanti le emoticons, trattandoli come una singola parola.

Essendo specifico il campo di utilizzo, ovverosia adatto ai posts di tweeter, non si è ritenuto opportuno testare questo tipo di tokenizzatore

La classe nltk.tokenize.mwe.MWETokenizer (Multi-Word Expression tokenizer)

MWETokenizer

Questo particolare tipo di tokenizzatore effettua una sorta di "*tokenizzazione al contrario".* Differisce da tutti gli altri tipi di tokenizzatori inquanto accetta come parametro in ingresso non più un testo da tokenizzare ma una lista ti tokens.

Partendo dai dati in ingresso e da un lexicon restituisce un lista di tokens in cui i tokens presenti nel lexicon sono raggruppati in un'unica parola sperati da un segno di separazione che per default è l'underscore.

Questo tokenizzatore è utilizzato per raggruppare le parole o espressioni da considerare come un'unica espressione; come si evince dall'esempio ricavato dalla documentazione interna della classe:

>>> tokenizer.tokenize('In a little or a little bit or a lot in spite of'.split())

*['In', **'a_little'**, 'or', **'a_little_bit'**, 'or', **'a_lot'**, **'in_spite_of'**]*

Nella creazione dell'oggetto, di fatto è possibile impostare il sepratore, tramite il parametro separator, e l'insieme del lexicon, costituita da una lista di liste, tramite il parametro mwes.

Il lexicon, sia esso quello di default o quello impostato, è modellato grazie alla classe nltk.util.Trie.

Data la natura specifica di questo tokenizzatore, non verranno effettuati tests su di esso.

La classe nltk.tokenize.stanford

Questa classe è un wrapper per python per la classe sviluppata in linguaggio Java dalla Standford University.

Il tokenizzatore in questione è stato sviluppato per supportare le lingue: Cinese (tradizionale e semplificato) e inglese.

Dato che lo scopo di questa ricerca è di testare i tokenizzatori, modificando e testando tutti i possibili parametri al fine di individuare la risorsa migliore per la lingua italiana, specialmente quelli del pacchetto nltk; essendo questo tokenizzatore esterno al package e sviluppato su lingue diverse da quella italiana non verrà escluso dai tests.

La sessione di Test del programma

Il Problema della Complessità

Uno dei principali problemi da affrontare immediatamente prima di avviare l'eseguzione di tutti i tests è quello legato alla complessità.

Il Metodo Brute Force

Il metodo Brute force rappresenta il tentativo di calcolare ogni possibile combinazione, senza effettuare nessuna operazione di filtraggio per scegliere quelle che potrebbero rappresentare una soluzione ottima o almeno sub-ottima.

Il problema della complessità rende impraticabile questa strada.

Per maggiore chiarezze se ne riporta un esempio esemplificativo

La complessità nella creazione dei punkt tokenizers

Durante la fase di creazione dei punkt tokenizers (vedi sezione "La fase di addestramento dei punkt tokenizers") è possibile definire i seguenti insiemi di parametri:

- _internal_punctuation
- _end_sent_punct
- _ABBREV
- _IGNORE_ABBREV_PENALITY
- _ABBREV_BACKOFF
- _COLLOCATION
- _SENT_STARTER
- _INCLUDE_ALL_COLLOCS
- _INCLUDE_ABBREV_COLLOCS
- _MIN_COLLOC_FREQ

Per esempio, assumiamo di voler testare i seguenti parametri, così definiti:

_internal_punctuation = {'default': ',:;', 'estesa': '_-@#,;:'}
_end_sent_punct = {'default': ('.', '?', '!'), 'estesa':('.', '?', '!','}',']')}
_ABBREV = [0.2, 0.3, 0.5]
_IGNORE_ABBREV_PENALITY = [True, False]
_ABBREV_BACKOFF = [3, 5 ,7]
_COLLOCATION = [6.47, 7.88, 8.11]
_SENT_STARTER = [25, 30, 45]
_INCLUDE_ALL_COLLOCS = [True, False]
_INCLUDE_ABBREV_COLLOCS = [True, False]

_MIN_COLLOC_FREQ = [1, 5]

Di fatto otterremo che per ogni combinazione di tipologia di ricostruzione corpora avremo:

O(_internal_punctuation * 2 * _end_sent_punct * 2 *_ABBREV * 3 * _IGNORE_ABBREV_PENALITY * 2 * _ABBREV_BACKOFF * 3 * _COLLOCATION * 3 * _SENT_STARTER * 3 *_INCLUDE_ALL_COLLOCS * 2 * _INCLUDE_ABBREV_COLLOCS * 2 * _MIN_COLLOC_FREQ * 2**)**

Quindi **O** è pari a (2 * 2 * 3 * 2 * 3 * 3 * 3 * 2 * 2 * 2), cioè pari a **5184** combinazioni, tutte da testare per ogni combinazione di ricostruzione del corpus. Se ad esempio considerassimo 3 possibili combinazioni come parametri tagS (vedi sezione "La costruzione del corpus nel programma per effettuare i tests") e 2 parametri per tagW otteniamo

O (5184 * tagS * 3 * tagW * 2) = 5184 * 6 = **31104**

Se ora consideriamo che vogliamo monitorare l'andamento dei vari tokenizzatori in una dimensione di dimensione del campione del corpus di test, abbiamo

31104 * nCorpus

Quindi, sempre a titolo esemplificativo, considerando 5 differenti dimensioni abbiamo:

31104 * 5 = **155520.**

Tutto questo, avendo considerato solo i punkt tokenizers.

Come si evince subito, il numero non possiede un rapporto lineare e il problema della complessità temporale diventa ben presto ingestibile.

Per ovviare a questo problema, si rende quindi necessario ricorrere ad una euristica che effettui una selezione e riduca in modo drastico questo numero. Tutto questo si ripercuote sui tempi di esecuzione dei tests.

Un Metodo per riduzione dei tempi di esecuzione

Passando da una logica sequenziale ad una in parallelo, riusciamo ad ottenere un sostanziale miglioramento delle prestazioni durante l'esecuzione dei tests.

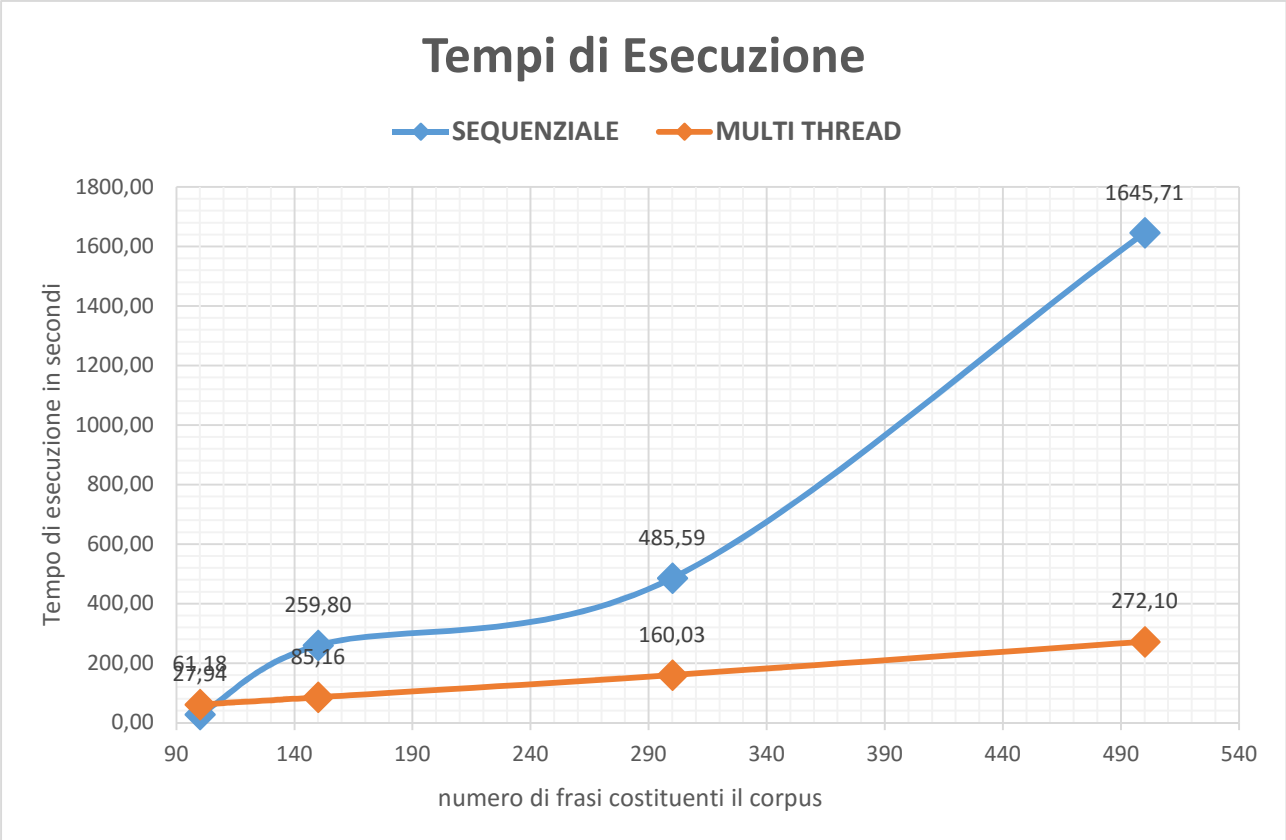
Per verificare ciò si è scelto di riscrivere la classe TestTextTiling in modo da renderla multithreading.

Questa tipologia di test è stata effettuata grazie al metodo TestTimeThreadVsNormal della classe TestTokenizer. Se ne riportano i risultati:

		tipologia di test	
		SEQUENZIALE	**MULTI THREAD**
numero	**100**	*27,94*	*61,18*

	150	*259,80*	*85,16*
	300	*485,59*	*160,03*
	500	*1645,71*	*272,10*

Il test consisteva nel effettuare un test completo (tramite tutte le possibili combinazioni di costruzione dei corpus) su un corpus composto da un numero di frasi variabile.



Il test per la sessione in multi threads è stata effettuata con 4 thread in eseguzione dato che l'elaboratore con cui sono stati effettuati i test possiede 4 unità logiche di calcolo su OS Win 10).

Per effettuare il test si è utilizzato il metodo

1. Tempo Start
2. Eseguzione
3. Tempo End

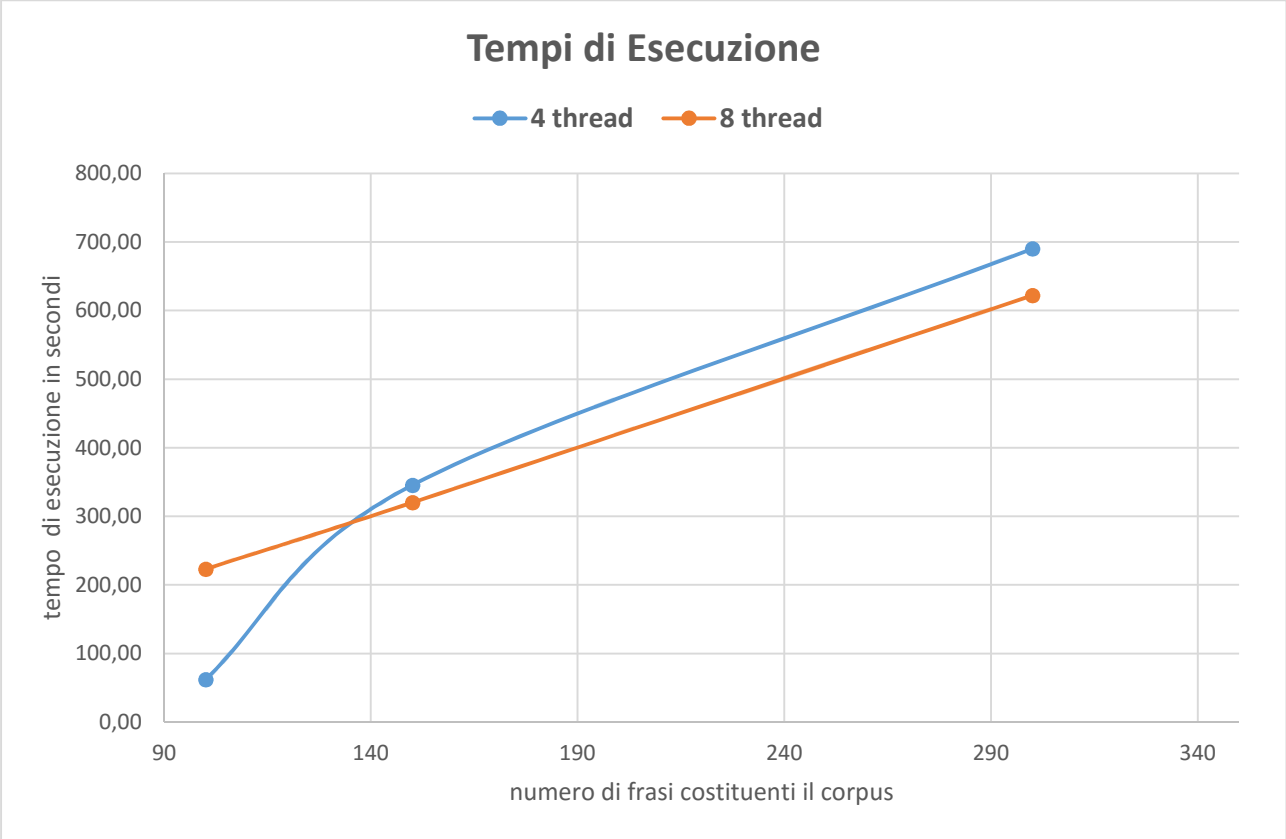
Tempo = Tempo Start – Tempo End

Benchè questo non risulti essere un metodo preciso ma essendo lo scopo di questi test indicativo, si è ritenuto un ottimo compromesso.

Come si evince dai dati, mentre quella rappresentante i test in multithread è una retta con una pendenza pari all'incirca a **0.534**, di contro la curva del metodo sequenziale ricorda quasi un curva esponenziale.

La fase successiva consiste nel valutare se il raddoppio del numero dei threads comporti un sostanziale miglioramento del processo. Si è proceduto confrontando lo stesso compito eseguito a 4 e a 8 thread.

		Numero Thread	
	n	4 thread	8 thread
	100	62,20	223,00
	150	345,45	320,00
	300	690,00	622,00
proiezione	383	993,68	787,33
	483	1307,58	986,83
	583	1621,48	1186,33
	683	1935,38	1385,83
	783	2249,28	1585,33



Come si può vedere, dopo la soglia critica di 150 frasi, non risulta particolarmente efficiente utilizzare 8 thread al posto di 4. La discrepanza potrebbe essere dovuta alla non precisione del metodo di registrazione del tempo di esecuzione utilizzato.

Il campione di Test

Per effettuare i test si è scelto di utilizzare una parte del corpus Paisa. Il corpus testato sarà così costituito:

DIMENSIONE DEL CAMPIONE MASSIMA ESPRESSA IN NUMERO DI PAROLE:
METTERE_NUM_WORD_USATE_PER_TEST

i test sulle dimensioni è effettuato prendendo 1/3, 2/3 e 3/3 della dimensione precedentemente impostata

La sessione di test

Tutti i tests sono stati raggruppati nel metodo AvviaTestWordsTokenizers il quale si occupa di richiamare i singoli test. Per semplicità si sono suddivisi i test in due funzioni distinte, le quali si occupano di richiamare ed eseguire i metodi associati ai test.

I TESTS SUI WORDS TOKENIZERS

Durante questa fase vengono eseguiti i seguenti test:

- TestSimpleSpaceTokenizerWord
- TestSimpleWordTokenizer
- TestSimpleWordTokenizerIta
- TestTreeBankTokenizer
- AvviaTestREWordTok (tipo = WORD)

I TEST SUI SENTS TOKENIZERS

Durante questa fase vengono eseguiti i seguenti test:

- TestSimpleLineTokenizerWord
- TestSimpleTokenizer
- TestSimpleTokenizerIta
- AvviaTestREWordTok (tipo = SENT)
- AvviaTestTextTilingTokenizer
- TestMyPunkt

I RISULTATI DEI TEST EFFETTUATI

TODO AGGIUNGERE I PDFS DEI TEST EFFETTUATI