UNIVERSITÀ DI PISA

Corso di Laurea in Informatica Umanistica

RELAZIONE

Correttore automatico applicato su testi storici

Candidato: *Michele Mallia*

Relatore: *Alessandro Lenci*

Correlatore: *Felice Dell'Orletta*

Anno Accademico 2014-2015

INDICE

Introduzione.....1
1. La spelling-correction applicata a testi storici.....2
1.1. I diari di guerra.....2
1.1.1. Stato dei testi.....2
1.1.2. Presenza di errori ortografici.....3
1.2. Spelling-Correction.....3
1.2.1. Problemi nella spelling-correction.....4
1.2.2. Studi sulla spelling-correction.....4
1.2.3. La distanza metrica.....5
1.3. Minimum edit distance.....5
1.3.1. Operazioni elementari.....5
1.3.2. L'allineamento.....5
1.3.3. La programmazione dinamica.....6
1.3.4. Costruzione della tabella di edit-distance.....6
1.4. Algoritmo di Norvig.....8
1.4.1. Operazioni “primitive”.....8
1.4.2. Alfabeto.....9
1.4.3. Struttura dell'algoritmo.....9
1.4.4. Come opera l'algoritmo.....10
1.4.5. Numero di candidati generati.....11
1.4.6. Ottenere candidati a distanza massima 2.....11
1.4.7. Metodi alternativi per la ricerca dei candidati.....12
2. Struttura, componenti del programma e metodologie di correzione.....13
2.1. Moduli python implementati.....13
2.1.1. Codecs.....13
2.1.2. WordPunctTokenizer e bigrams.....14
2.1.3. Collection.....14
2.1.4. Tag.....14
2.1.5 Re.....15
2.1.5 Difflib e SequenceMatcher.....15

2.2. Preparazione dei dizionari....15
2.2.1. Dizionario degli unigrammi (NWORDS)....16
2.2.1.1. Funzione creaDizionario()....17
2.2.1.2. Termini a bassa frequenza....18
2.2.2. Dizionario di bigrammi (BIG)....19
2.2.2.1. Pattern matching....20
2.2.2.2. Preparazione dei bigrammi....20
2.2.2.3. Scrittura del dizionario esterno....21
2.2.2.4. Caricamento del dizionario dei bigrammi....21
2.3. Preparazione del testo e tokenizzazione....22
2.4. Metodi di correzione....23
2.4.1. Probabilità con parola a sinistra e a destra....24
2.4.1.1. Determinazione di un verbo pronominale....24
2.4.1.2. Determinazione della fusione fra due parole....25
2.4.1.3. Selezione, generazione di parole e filtraggio dei candidati....26
2.4.1.4. Calcolo delle probabilità.... 26
2.4.1.5. Ordinamento e selezione del candidato....27
2.4.1.6. Ripristino dei valori iniziali....27
2.4.2. Possibilità con parola solo a sinistra - possibilità con parola solo a destra....28
2.4.3. Non word Zero....28
3. Valutazioni e performance del programma....30
3.1. Esempio di output....30
3.2. Risultati della correzione automatica....32
3.3. Pregi e difetti del progetto....33
3.3.1. Tempo e memoria....33
3.3.2. Assenza di un sistema context-sensitive....34
3.3.3. Impossibilità nell'implementare le catene di Markov....35
Conclusioni....36
Bibliografia e webliografia....37
Ringraziamenti....38

INTRODUZIONE

La linguistica computazionale è una scienza che unisce due mondi che, a primo impatto, sembrano distanti anni luce l'uno dall'altro: un mondo è popolato da algoritmi e strutture dati; l'altro è popolato da parole e da strutture linguistiche, le cui regole, al contrario del primo, sono insite nella natura dell'essere umano.

L'unione di questi due mondi genera nuove prospettive per molti versi pionieristiche. Negli ultimi anni, con l'evoluzione del *web 2.0* e con lo sviluppo di tecnologie che si avvicinano ancora di più alle esigenze dell'uomo, questa disciplina si rivela sempre più importante.

Il progetto presentato consiste in un correttore automatico scritto in Python, applicato a lettere risalenti alla Prima Guerra Mondiale, scritte da soldati con uno scarso livello di istruzione e con molti errori ortografici e di sintassi.

Con la ricorrenza del centenario della Prima Guerra Mondiale (1915-2015), e con la partecipazione del candidato al progetto *Memorie di Guerra*, realizzato dall'Istituto di Linguistica Computazionale A. Zampolli e l'Università di Pisa in collaborazione con l'Università di Siena[1], si è deciso di proseguire per questa linea tematica e di creare qualcosa che possa utilizzare le tecnologie informatiche applicate alla lingua, al fine di ricostruire gli schemi del linguaggio naturale e di poterli usare per la risoluzione di problemi linguistici complessi.

Questa tesi rappresenta un punto di partenza per uno studio futuro più approfondito della lingua e delle tecniche computazionali per la realizzazione di sistemi e metodi più intelligenti per la correzione automatica o per altri task per il trattamento dei testi. Una risorsa sempre più importante in un mondo dove l'umanizzazione della tecnologia diventa sempre più preponderante.

1 Per vedere il progetto, si prega di visitare la pagina *Memorie di Guerra* all'indirizzo: http://memoriediguerra.it/

CAPITOLO I

La spelling-correction applicata a testi storici

Il progetto di questa tesi di laurea si pone come obbiettivo quello di poter analizzare testi appartenenti a diari scritti dai soldati durante la prima guerra mondiale e apportare correzioni a errori ortografici. Esso consiste nella realizzazione di un piccolo correttore automatico scritto in Python, con l'aggiunta di alcune librerie utili per l'analisi del testo.

La correzione automatica, in inglese *spelling correction*, pone le sue radici nella seconda metà degli anni Cinquanta e negli ultimi vent'anni, grazie allo sviluppo delle tecnologie del linguaggio applicate al mondo di internet, ha avuto uno sviluppo molto sostenuto.

Grazie alla digitalizzazione di questi testi è stato possibile applicare direttamente al testo algoritmi capaci di prevedere un candidato (o più candidati) per una parola errata.

Per parlare del progetto è necessario affrontare la questione secondo una visione panoramica, andando ad individuare quali sono gli strumenti e gli studi che trattano questa tematica.

1.1. I diari di guerra

I testi sottoposti ad analisi appartengono a diari scritti da soldati durante la Prima Guerra Mondiale. Questi documenti sono stati raccolti alla fine della guerra e custoditi in varie zone, fra la Toscana, l'Umbria e le Marche. Al centenario dell'entrata in guerra dell'Italia, tutte le lettere sono state digitalizzate e pubblicate su internet[2], con il contributo del gruppo editoriale L'Espresso e l'Archivio diaristico nazionale.

1.1.1. Stato dei testi

I testi sono stati mantenuti nella loro forma originaria, senza omettere o correggere errori presenti nel testo; in questo modo si sono mantenute le tracce vive della cultura

2 *La Grande Guerra* – L'Espresso con l'Archivio diaristico nazionale: http://espresso.repubblica.it/grandeguerra/index.php

e della lingua italiana di quel periodo.

1.1.2. Presenza di errori ortografici

Molte lettere sono state scritte da semi-analfabeti, pertanto sono presenti nei testi errori di ortografia e costruzioni lessicali di origine dialettale, nomi di luoghi modificati e tempi verbali non corretti.

In questi testi viene raccontata la guerra con gli occhi di chi la combatteva e non solo: gli autori di queste lettere sono soldati semplici, medici, infermiere e soldati di grado superiore. Ognuno di essi racconta un particolare evento o situazione che sta vivendo, e viene ricostruito così uno scenario più completo degli avvenimenti della guerra.

Un esempio di una lettera di un soldato viene riportata qua sotto:

> *"Così, entrammo in quello posto, che c'erino sacchette piene di terra e poi c'era uno raficello con una crocetella che l'avevino scavato i soldate. Così, il sercente mi ha detto piano piano: - Rabito,prente solo il fucile, il pognale e li bombe, e il zaino lo mette lì, dentra a quello picolo nascontiglio, che poi lo prente."*[3]

Come si può ben notare, in questo testo sono presenti molti errori, alcuni dei quali difficili da correggere.

La correzione automatica applicata a questa tipologia di documenti è risultata più difficile rispetto ad altri testi per via di tanti fattori; per questo sono state applicate delle regole universali, che possano correggere errori su una grande varietà di testi.

Esistono tuttavia molte delle tecniche di *spelling-correction* che possono risolvere, seppur parzialmente, problemi di questo tipo.

E' necessario anche capire in che cosa consiste questa tecnica che può permetterci di effettuare modifiche utili in un testo.

1.2. Spelling-correction

Per *spelling-correction* si intende quel processo di correzione di parole errate in documenti di testo. Tutti gli algoritmi di correzione automatica sono probabilistici;

3 *La Grande Guerra* – L'Espresso con l'Archivio diaristico nazionale
Vincenzo Rabito, *Il primo giorno in trincea*
http://espresso.repubblica.it/grandeguerra/index.php?page=estratto&id=286

non esiste un algoritmo che riesca a correggere tutti gli errori presenti in un testo.

Il rilevamento e la correzione di errori è parte integrante dei motori di ricerca e di applicazioni che operano su testi, come ad esempio le applicazioni di O*ptical Character Recognition* (*OCR*).

1.2.1. Problemi nella *spelling-correction*

Nella *spelling-correction* si possono distinguere tre tipologie di problemi:

1. Rilevamento di *non-words*: consiste nell'individuare parole che non risultano presenti in un dizionario;

2. Correzione di parole isolate: ovvero la correzione della parola che non risulta nella lista indipendente dal contesto;

3. Rilevamento e correzione di parole con contesto: si tratta di quel procedimento con il quale viene individuata una parola (*real word*) che è stata erroneamente inserita in un contesto a cui non appartiene e viene corretta in base al contesto in cui si trova (un classico esempio può essere rappresentato dal caso degli omofoni: "*ci anno detto che"*, *"si e detto che"* ecc...)

1.2.2. Studi sulla *spelling-correction*

Molti sono stati gli studi condotti su tale argomento: degni di nota sono gli studi condotti da Peterson nel 1986, il quale propose di usare un dizionario dei termini (*spelling dictionaries*) minore rispetto ad altri dizionari più grandi, poiché questi ultimi possono contenere termini rari che somigliano a errori ortografici di altre parole (ipotizziamo di avere una *word-list* e di applicare un filtro per raccogliere i termini con frequenza maggiore o uguale a 2 e creare un nuovo dizionario; si rischierebbe di aggiungere parole a bassissima frequenza come *ciorno*, invece di *giorno*, e *otel*, invece di *hotel*, che sono errori di ortografia che in un corpus di parole possono comparire; in maniera conseguenziale, queste parole non verrebbero riconosciute come errori, lasciandole non corrette); Damerau e Mays, nel 1989, scoprono che invece è meglio usare grandi dizionari, poiché con l'utilizzo di questi

dizionari si evita di interpretare come errori parole a bassa frequenza e, quindi, evitare l'incremento di ulteriori errori generati inconsapevolmente dal programma (attualmente i sistemi di correzione automatica tendono ad usare dizionari di grandi dimensioni[4]); attualmente, lo stato dell'arte è stato raggiunto con la ricerca di Andrew R. Golding e Dan Roth, che ha portato alla realizzazione di un metodo efficace per correggere le parole in base al contesto[5].

1.2.3. La distanza metrica

Per la correzione automatica, in genere, si fa ricorso a misure che calcolano la similarità fra una parola ed un'altra. Per scegliere i potenziali candidati si fa ricorso ad una misura di *distanza*[6] fra una parola sorgente (in questi casi, l'errore ortografico) e una parola di destinazione (il candidato). L'algoritmo che permette di calcolare la *distanza* è quello della *minimum edit distance*.

1.3. Minimum edit distance

L'algoritmo di *minimum edit-distance* serve per quantificare la *distanza*, ovvero la differenza fra due stringhe.

1.3.1. Operazioni elementari

La distanza fra due stringhe si calcola prendendo in considerazione tre operazioni elementari: *cancellazione* di un carattere, *inserimento* di un carattere e *sostituzione* di un carattere. La distanza non è altro che il numero di operazioni che si devono operare su una stringa in entrata per ottenere una stringa in uscita.

1.3.2. L'allineamento

Si considerino due stringe di caratteri come *intenzione* ed *esecuzione*; entrambe le stringhe vengono sottoposte ad un processo di *allineamento*, dove le due sotto-

4 Daniel Jurafsky & James H. Martin, *Speech and Language Processing: An introduction to natural language processing, computational linguistics, and speech recognition*, p.30

5 Andrew R. Golding, Dan Roth, *A Winnow-Based Approach to Context-Sensitive Spelling Correction*: http://arxiv.org/pdf/cs/9811003.pdf

6 Si veda il paragrafo successivo per la definizione delle misure di *distanza*

stringhe (ovvero le lettere che compongono la parola) vengono confrontate.

```
I N T E N Z I O N E
| | | | | | | | | |
E S E C U Z I O N E
```

Prendendo in considerazione le tre operazioni fondamentali citate precedentemente, si assegna un costo computazionale di valore 1per ogni modifica che viene fatta per ottenere la stringa in uscita.

```
I N T E N Z I O N E
| | | | | | | | | |
E S E C U Z I O N E
1 1 1 1 1 0 0 0 0 0
```

In questo caso, il costo o *distanza* fra le due stringhe è di 5, valore ottenuto sommando tutti i costi di *sostituzione* fra una sotto-stringa e l'altra. Una versione alternativa dell'algoritmo di Levenshtein prevede le seguenti regole sui costi di trasforrmazione: inserimento ed eliminazione valgono 1, mentre la sostituzione può non essere ammessa o valere 2.[7]

1.3.3. Programmazione dinamica

In informatica, la *minimum-edit-distance* viene calcolata col metodo della *programmazione dinamica*, una tecnica di progettazione di algoritmi basata sulla risoluzione dei problemi con la suddivisione del problema principale in sotto-problemi e con la combinazione della soluzione con questi ultimi usando metodi tabellari.

1.3.4. Costruzione della tabella di *edit-distance*

Si prenda in considerazione l'algoritmo di Levenshtein usato in questo progetto (cornice 1); l'algoritmo provvederà a creare una matrice [x+1,y+1], al cui interno sono presenti la *source string* (nella colonna x), la *target string* (nella riga y) e i

7 *Daniel Jurafsky, James H. Martin*, Speech and Language Processing: An introduction to natural language processing, computational linguistics and speech recognition, 2006.

valori corrispondenti alla somma dei costi computazionali fra le diverse sotto-stringhe.

Viene riportato l'esempio di una matrice (si veda la tabella 2) che viene prodotta una volta eseguito l'algoritmo di Levenshtein; ogni cella contiene la distanza fra il primo carattere della stringa *i* e il primo carattere della stringa *j*. Ogni cella viene calcolata come una semplice funzione delle celle circostanti; avviato l'algoritmo, viene riempita ogni singola cella con un valore.

```
 def LD(seq1, seq2):
     oneago = None
     thisrow = range(1, len(seq2) + 1) + [0]
     for x in xrange(len(seq1)):
         twoago, oneago, thisrow = oneago, thisrow, [0] * len(seq2)
+ [x + 1]
         for y in xrange(len(seq2)):
             delcost = oneago[y] + 1
             addcost = thisrow[y - 1] + 1
             subcost = oneago[y - 1] + (seq1[x] != seq2[y])
             thisrow[y] = min(delcost, addcost, subcost)
     return thisrow[len(seq2) - 1]
```

Cornice 1: Algoritmo di Levenshtein usato in questo progetto

Il valore in ogni cella viene calcolato considerando il valore minimo dei tre percorsi possibili attraverso la matrice che arriva in quel punto.

e	10	9	9	8	9	9	9	8	7	6	5
n	9	8	8	8	8	8	8	7	6	5	6
o	8	7	7	7	7	7	7	6	5	6	7
i	7	6	6	6	6	6	6	5	6	7	8
z	6	5	5	5	5	5	5	6	7	8	9
n	5	4	4	4	4	5	6	7	8	8	9
e	4	3	4	3	4	5	6	7	8	9	8
t	3	3	3	3	4	5	6	7	8	8	8
n	2	2	2	3	4	5	6	7	7	7	8
i	1	1	2	3	4	5	6	6	7	8	9
#	0	1	2	3	4	5	6	7	8	9	10
	#	**e**	**s**	**e**	**c**	**u**	**z**	**i**	**o**	**n**	**e**

Tabella 2: matrice dell'edit-distance con i relativi costi computazionali

Il valore nell'ultima cella corrisponderà alla distanza minima fra le due stringhe.

Qui sotto è riportata l'equazione che effettua il calcolo dei costi, il cui valore viene inserito in ogni cella:

$$distance[i,j] = min \begin{cases} distance[i-1,j] + ins - cost(target_{i-1}) \\ distance[i-1,j-1] + subst - cost(source_{j-1}, target_{i-1}) \\ distance[i,j-1] + del - cost(source_{j-1}) \end{cases}$$

E' ben chiaro adesso come sia possibile stabilire, dal punto di vista computazionale, la somiglianza fra due parole.

Per questo progetto, l'algoritmo di Levenshtein è di fondamentale importanza, ma non è l'unico vero protagonista del lavoro svolto.

Esiste infatti un'altra componente importante del progetto che ci permette di ottimizzare il tempo tempo di generazione di parole simili dato un errore ortografico e che si basa, per l'appunto, sull'algoritmo della *minimum edit distance*: l'algoritmo in questione è quello di Peter Norvig, inteso anche come "generatore di candidati".

1.4. Algoritmo di Norvig

Il secondo componente fondamentale di questo progetto è l'algoritmo di Peter Norvig. Sviluppato dall'omonimo direttore del centro di ricerca dei laboratori di Google, l'algoritmo si occupa di generare parole (o candidati) data una parola che non risulta essere presente in un dizionario. La generazione dei candidati avviene attraverso l'uso di operazioni *primitive* di modifica della parola e, tramite delle condizioni di verifica, vengono filtrati i risultati che hanno una frequenza maggiore a zero (vengono filtrate le parole che sono presenti nel dizionario, scartando così le parole inesistenti e restringendo il numero degli elementi nell'insieme dei possibili candidati). Data una *mispelled*-word, il programma genera 10 candidati al secondo.

In questo progetto è stata usata una piccola parte dell'algoritmo originale, in quanto alcune componenti del programma di Norvig non si sono rivelate utili ai fini del progetto.

L'algoritmo è scritto originariamente in Python, ma esistono altre versioni del programma scritte in diversi linguaggi di programmazione.

1.4.1. Operazioni "primitive"

L'algoritmo per la generazione delle parole segue le stesse regole *primitive* usate per

calcolare la distanza di Levenshtein; vengono infatti generati candidati mediante operazioni di *inserimento*, *eliminazione*, *sostituzione* e una regola in più, ovvero la *trasposizione* (spostamento di due parole adiacenti, una a fianco all'altra).

Inizialmente, le parole vengono generate seguendo queste regole, producendo anche parole che non esistono nel dizionario. Viene in seguito eseguita un'operazione di filtraggio, restituendo parole corrette. Con questa funzione vengono prodotte parole che hanno distanza 1 da quella data in input.

1.4.2. Alfabeto

Per prima cosa, si deve dare all'algoritmo una stringa con le lettere dell'alfabeto, con le quali l'algoritmo si occuperà di costruire i token.

```
alphabet = u'abcdefghijklmnopqrstuvwxyzàèìòùé'.encode('utf-8')
```

Nell'alfabeto vengono incluse, rispetto all'algoritmo originale, anche le lettere accentante, visto e considerato che si sta trattando di una correzione automatica di testi in lingua italiana. Per consentire all'algoritmo di generare parole con caratteri diverse dallo standard *ASCII*[8], la stringa deve essere convertita e codificata nel formato *unicode*, in modo da poter leggere anche i caratteri accentati senza avere problemi di compatibilità.

1.4.3. Struttura dell'algoritmo

L'algoritmo di generazione di candidati si presenta in questa forma:

```
def edits1(word):
   splits     = [(word[:i], word[i:]) for i in
range(len(word) + 1)]
   deletes    = [a + b[1:] for a, b in splits if b]
   transposes = [a + b[1] + b[0] + b[2:] for a, b in
splits if len(b)>1]
   replaces   = [a + c + b[1:] for a, b in splits for c
in alphabet if b]
```

8 ASCII: acronimo di American Standard Code for Information Interchange, è il primo standard per l'assegnazione di codici a caratteri; creato nel 1963, è un set di caratteri riconosciuto da tutti i computer, contenente 128 caratteri, sufficienti per rappresentare l'inglese, ma non per rappresentare altri caratteri occidentali.

```
    inserts    = [a + c + b     for a, b in splits for c
in alphabet]
    return set(deletes + transposes + replaces + inserts)
```

I caratteri in grassetto rappresentano le operazioni elementari che l'algoritmo opera su una parola. *Splits* non è una vera e propria regola elementare; questa variabile serve per dividere la stringa in entrata (`word`) seguendo un ciclo che parte da 0 e si conclude con un numero pari alla lunghezza della parola più 1 (che corrisponde allo spazio vuoto); questo per permettere alle altre variabili di *inserimento*, *eliminazione*, *trasposizione* e *sostituzione* di poter operare su porzioni di stringa che le vengono passate da *split*.

1.4.4. Come opera l'algoritmo

Per capire meglio come opera l'algoritmo, è bene osservare la stampa di ogni singola variabile presente nell'algoritmo. Si consideri come stringa di esempio la stringa *casa*.

Data in input una stringa di testo, la variabile *splits* restituirà questo output:

```
[('', 'casa'), ('c', 'asa'), ('ca', 'sa'), ('cas', 'a'), ('casa',
'')]
```

Si tratta di una lista contenente cinque tuple con due elementi; ogni tupla contiene due stringhe, che corrispondono alla stringa data in input, divisa in due per scorrimento, seguendo un ciclo che parte da 0 fino ad un valore che è pari alla lunghezza della stringa, ovvero 4.

Una volta divisa la stringa data in input, entrano in gioco gli altri componenti dell'algoritmo che si occupano della vera a propria modifica della stringa.

Qui sotto vengono riportati gli output delle diverse variabili di modifica dell'algoritmo:

deletes ==>
```
['asa', 'csa', 'caa', 'cas']
```

transposes ==>
```
['acsa', 'csaa', 'caas']
```

replaces ==>
```
['aasa',  'basa',  'casa',  'dasa',  'easa',
'fasa',  'gasa',  'hasa',  'iasa',  'jasa',
'kasa',  'lasa',  'masa',  'nasa',  'oasa',
'pasa', 'qasa', 'rasa', ...]
```

inserts ==>

```
['acasa', 'bcasa', 'ccasa', 'dcasa', 'ecasa',
'fcasa', 'gcasa', 'hcasa', 'icasa', 'jcasa',
'kcasa', 'lcasa', 'mcasa', 'ncasa', 'ocasa',
'pcasa', ...]
```

Dopo che le variabili hanno eseguito il loro compito di modifica della stringa data in input, l'algoritmo restituirà un insieme di elementi diversi (diversi fra loro) sommando i prodotti dalle variabili di inserimento, trasposizione, eliminazione e sostituzione.

1.4.5. Numero di candidati generati

Il numero delle parole prodotte da quest'algoritmo si può calcolare con una formula: si consideri *n* come numero pari alla lunghezza di una parola, il numero dei risultati è pari alla somma di *n* cancellazioni, *n*-1 trasposizioni, $32^{9}n$ alterazioni e $32(n+1)$ inserimenti. Ad esempio, il numero degli elementi restituiti dall'algoritmo prendendo in input la stringa *casa* è di circa 295.

Precedentemente si è detto che risultati dell'algoritmo sono una serie di parole con distanza 1 e per lo più sgrammaticate. Generalmente, un correttore automatico ricopre solo il 76% della possibilità di generare una parola corretta con distanza 1; per questo è necessario poter generare parole con distanza 2 (in rarissimi casi con distanza 3) per poter coprire il 98,9% delle probabilità di trovare il candidato perfetto.[10]

1.4.6. Ottenere candidati a distanza massima 2

Per ottenere parole non sgrammaticate e con distanza minore o uguale a 2 si usa la funzione `known_edits2`, riportata qua sotto, il cui compito è quello di effettuare un doppio ciclo sulla funzione `edits1`, e di restituire soltanto le parole che sono presenti in `NWORDS`, ovvero il dizionario di riferimento:

9 Solitamente questo numero è pari a 26, ovvero il numero delle lettere che compongono l'alfabeto inglese; per l'italiano si considera il numero delle lettere dell'alfabeto inglese più il numero delle lettere accentate (àèìòùé); così facendo si ottiene 32 (26+6).

10 Si veda la pagina *How to write a spelling corrector* al seguente indirizzo: http://norvig.com/spell-correct.html

```
 def known_edits2(word):
     return set(e2 for e1 in edits1(word) for e2 in edits1(e1) if e2
in NWORDS)
```

Ecco un esempio di parole generate dall'algoritmo prendendo in input la stringa *casa*:

```
 (['coda', 'casane', 'sca', 'ala', 'osa', 'kata', 'caco',
'scarsa',  'saga',  'castra',  'vasi',  'vaso',  'fas',
'cascia',  'vasa',  'risa',  'caos',  'casas',  'vast',
'cuasa', 'sana', 'lisa', 'cose', 'cap', 'pasa', 'cosa',
'vasai', 'coso', 'paso', 'cosi', 'past', 'carla'...)]
```

1.4.7. Metodi alternativi per la ricerca dei candidati

Un'alternativa alla generazione di parole candidate può essere rappresentata dalla ricerca, all'interno del dizionario dei monogrammi, delle parole con distanza minore o uguale a due; tuttavia è una soluzione molto inefficiente in termini di tempo, per cui l'uso di questo algoritmo speciale ci permette di ottenere buoni risultati in tempi molto brevi.

CAPITOLO II

Struttura, componenti del programma e metodologie di correzione

In questo capitolo si tratterà del progetto dal punto di vista della sua struttura e delle metodologie di correzione; verranno elencati tutti i componenti, da quelli più elementari, come i moduli in python implementati, a quelli più particolari, come gli algoritmi decisionali per la correzione degli errori ortografici.

Si discuterà del flusso di operazioni che verranno applicate al testo, da immaginare come un percorso che ha un inizio, caratterizzato dalla *lettura* del testo in entrata, un intermezzo, ovvero la correzione dei vari errori ortografici, e una fine, rappresentata dal confronto fra il testo originale e il testo modificato.

2.1. Moduli python implementati

Prima di iniziare a descrivere la struttura del programma e le procedure di correzione, è bene fare un piccolo riassunto dei moduli in Python usati nel progetto; questo perché, anche se non sono i protagonisti principali della correzione automatica, si rivelano strumenti utili per un buon proseguimento di queste operazioni, aumentando le capacità e gli strumenti di analisi per Python.

2.1.1. Codecs

Il primo modulo che deve essere preso in considerazione è il modulo *codecs,* che è lo strumento che permette al programma di poter effettuare la lettura e la codifica di un testo preso in input; questo modulo permette la lettura di un testo in *unicode*[11], qualora quest'ultimo presenti caratteri o segni grafici particolari (come ad esempio i caratteri accentati, tipici della lingua italiana). La *lettura* e la *codifica* sono le prime operazioni che vengono effettuate all'interno del progetto.

11 Unicode, o *Universal Character Set*, è uno standard internazionale di codifica che contiene un set di caratteri molto grande (100.000 caratteri grafici rappresentati) e che permette quindi di poter rappresentare qualsiasi tipo di carattere appartenente ai sistemi grafici esistenti (dalle lingue europee alle lingue asiatiche); il sistema funziona con l'assegnazione di un codice univoco ad ogni carattere, risolvendo così i problemi di incompatibilità con i precedenti sistemi di rappresentazione dei caratteri (ISO-8859). Con Unicode si possono realizzare e/o leggere documenti scritti in più lingue diverse senza riscontrare problemi di codifica.

2.1.2. WordPunctTokenizer e bigrams

Entrambi i moduli fanno parte della libreria di NLTK (Natural Language ToolKit); il primo modulo è un *tokenizzatore* che si occupa di dividere il testo preso in input in singoli *token*, creando una lista contenente parole ed altri segni di punteggiatura. La particolarità che contraddistingue questo metodo di tokenizzazione rispetto agli altri è la capacità di poter effettuare una divisione del testo più profonda[12], spezzettando anche le parole unite da elisione.

Bigrams, invece, è un modulo che crea, data una lista di token, una seconda lista composta da coppie di parole adiacenti.

2.1.3. Collection

Serve per la creazione di *dizionari*: un dizionario è un particolare tipo di struttura dati che serve ad immagazzinare una grandissima quantità di *chiavi* (in questo caso, token/parole o bigrammi) accompagnate da un *valore* (che corrisponde, in questo caso, alla frequenza di una chiave nel corpus). Usare un dizionario può essere in molti casi più utile di una semplice di una lista, poiché le operazioni di accesso e ordinamento sono molto più veloci e meno costosi in termini di memoria, risparmiando così molto tempo nell'esecuzione del programma.

2.1.4. Tag

Dal modulo *pattern.it* viene importato un modulo molto utile, ovvero *tag*. Questo strumento ci permette di poter estrarre una *pos*[13] da una parola; eseguito il modulo, esso ci restituisce una *tupla* composta da due elementi: il primo è il token di cui ci interessa sapere la classe sintattica; il secondo elemento invece è la classe sintattica stessa.

La precisione di questo strumento si aggira attorno al 92%, ed è stato allenato con

12 Si prenda in considerazione la sequenza di testo *"c'era un'altra ragazza"*: un tokenizzatore standard come *PunktTokenizer* provvede a dividere il testo nella seguente maniera (token 1: *"c'era"*, token 2: *"un'altra"*, token 3: *"ragazza"*); *WordPunctTokenizer*, invece, divide il testo in testo in questo modo (token 1: *"c"* , token 2: *" ' "* , token 3: *"era"* , token 4: *"un"*, token 5: *" ' "* , token 6: *"altra"* , token 7: *"ragazza"*).

13 Pos: sta per *part of speech*, è un codice che serve ad identificare la classe sintattica di una parola. Ad esempio, VB (*verb*) indicherà un verbo, NN (*noun*) un nome comune ed NNP (*proper noun*) un nome proprio. Le *pos* non sono tutte uguali, non seguono uno standard internazionale ed esistono sistemi che hanno *pos* nominate diversamente.

una collezione di termini presi da Wikitionary.[14]

2.1.5. Re

Il modulo *re* è uno dei più importanti usati nel programma: esso dà a Python la possibilità di poter estendere le sue manipolazioni del testo con l'utilizzo delle *regex* (o espressioni regolari). In questo progetto, il modulo *re* viene usato per poter verificare se un token che si sta analizzando è una parola o meno, e per evitare di fornire eventuali simboli di punteggiatura ad algoritmi che si occupano della correzione del testo (pena, la compromissione dei risultati del programma). Questa operazione di riconoscimento/selezione dei token viene chiamata *string matching*.

2.1.6. Difflib e SequenceMatcher

Il modulo *difflib*, con il suo sotto-modulo *SequenceMatcher*, si occupa di verificare quanto due stringhe di testo sono più o meno simili. Esso restituisce un valore che va da 0 (che indica la totale diversità di una parola dall'altra) a 1 (viceversa, la totale uguaglianza). Utilizza una strategia simile a quella adoperata dall'algoritmo di Levenshtein, anche se tuttavia opera in maniera diversa, in quanto non restituisce la distanza effettiva fra una parola ed un'altra, ma la similarità fra le due verificando la loro lunghezza e i caratteri usati per la composizione della parola. Questo modulo servirà per filtrare i candidati generati dall'algoritmo di Peter Norvig.

2.2. Preparazione dei dizionari

Per poter effettuare una correzione automatica efficiente è necessario avere come componente di base un dizionario contenente vocaboli corretti della lingua italiana. Il programma, infatti, deve avere una *conoscenza* su quali siano i vocaboli corretti e quali no. Per prima cosa, dunque, è necessario costruire un dizionario.

Rispetto alle prime versioni precedenti di questo progetto, che prevedevano la costruzione di un dizionario una volta avviato il programma, si è alla fine optato per la costruzione di un dizionario esterno[15], in modo da non ricreare la struttura dati

14 Percentuale riportata dal sito di *pattern.it* alla voce *Parser*: http://www.clips.ua.ac.be/pages/pattern-it

15 Il dizionario esterno a cui si sta facendo riferimento è il dizionario dei *bigrammi*, e non il dizionario degli unigrammi (ovvero, quello dei vocaboli), poiché quest'ultimo è già presente (il file *formario*), e, non dovendo essere stampato, viene semplicemente letto da una funzione del

contenente i vocaboli a cui fare riferimento ogni volta che viene eseguito il programma. Con questa soluzione si guadagna parecchio tempo e memoria, perché il dizionario viene creato una sola volta e si può accedere ad esso ogni volta che ce n'è bisogno.

Per la creazione del dizionario dei bigrammi è necessario fare una scansione del corpus di testi, accedendo ad esso, ma senza aprirlo, perché ciò provocherebbe un sovraccarico in termini memoria in quanto il corpus ha un notevole peso in megabyte (il corpus originale ha un peso di 1.45 gigabyte).

Per la creazione del dizionario degli unigrammi si deve accedere al file *formario*, che è un file di testo di circa 70 megabyte e contiene soltanto i termini del corpus principale di PAISA'[16].

Per la creazione dei dizionario dei *bigrammi* si accederà invece al file del corpus, in modo da analizzare interamente i testi e poter estrarre le coppie di parole da essi.

2.2.1. Dizionario degli unigrammi (NWORDS)

Il file *formario* è strutturato in questo modo: per ogni riga del file ci sono informazioni relative al lemma della parola con la rispettiva *pos* affiancata (colonna uno), il codice del lemma (colonna due), il termine nella sua forma flessa non lemmatizzata (colonna tre) e la frequenza relativa al termine (colonna quattro).

Qui sotto viene fornito un estratto del file *formario*:

```
essere_V   3394893     è      1507918
essere_V   3394893     sono   458303
essere_V   3394893     era    269959
essere_V   3394893     essere      166260
...
```

programma. Le istruzioni sulla creazione e scrittura del file con i dati relativi al dizionario dei bigrammi verranno illustrate in seguito.

16 Il corpus PAISÀ è un'ampia collezione di testi in lingua italiana tratti da Internet; esso contiene una notevole quantità di testi (con circa 250 milioni di *token* provenienti da circa 380.000 documenti) ed è anch'esso disponibile gratuitamente online. Per questo progetto non è stato usato il corpus nel formato CoNLL, bensì è stato usato un *formario*, ovvero un elenco di parole presenti nel corpus con a fianco la relativa frequenza e la rispettiva classe sintattica. Inoltre si è usato anche il corpus nel formato *plain-text*, contenente i testi originali dei documenti raccolti, per la ricerca delle *collocazioni* e dei *bigrammi*.
Sergio Scalise, Claudia Borghetti, Francesca Masini, Vito Pirrelli, Alessandro Lenci, Felice Dell'Orletta, Andrea Abel, Chris Culy, Henrik Dittmann, Verena Lyding, Marco Baroni, Marco Brunello, Sara Castagnoli, Egon Stemle, *Corpus dell'Italiano – Progetto PAISÀ*
http://www.corpusitaliano.it/it/index.html

2.2.1.1. Funzione creaDizionario()

La funzione `creaDizionario()` prevede l'esecuzione delle seguenti istruzioni: non appena viene invocata la funzione (che non richiede alcun parametro in entrata) viene eseguito l'accesso al file *formario* con `with open("formario") as formario`; questo metodo permette l'accesso al file, ma non implica la sua apertura o lettura, che richiederebbe all'interprete Python un notevole impiego di memoria per leggere tutte le righe. Una volta avuto l'accesso al file, l'algoritmo si preoccupa di creare una variabile `dizionario` che rappresenta il dizionario stesso; `defaultdict` serve ad indicare che la struttura dati che si vuole realizzare è quella di un dizionario che, a differenza dei dizionari tradizionali, restituisce un valore di *default* nel caso in cui si stia effettuando una ricerca di un termine o chiave non presente nel dizionario; in questi casi il valore di default viene espresso con `lambda:0`, che, in questo caso, sta ad indicare che il valore di default che si deve restituire è pari a 0 qualora la ricerca non dovesse andare a buon fine; un notevole vantaggio di questa scelta è che, con questo tipo di struttura dati, non si potrà mai incorrere in errori di tipo *key error* per le chiavi non presenti nel dizionario, e quindi compromettere l'andamento del flusso di istruzioni.

```
def creaDizionario():
     with open("formario") as formario:
           dizionario = collections.defaultdict(lambda:0)
           for line in formario:
                 key = line.split()[2]
                 value = int(line.split()[3])
                 if value > 1:
                       dizionario[key] += value
     return dizionario
```

Funzione per creare il dizionario dei monogrammi (NWORDS)

La prossima istruzione da seguire è un ciclo `for` all'interno del variabile `formario`, che rappresenta il file al quale si ha avuto accesso; il ciclo scorre tutto il file partendo dalla prima riga del dizionario e si ferma soltanto quando ha raggiunto l'ultima; la variabile `line` rappresenta la riga del dizionario che viene analizzata ad ogni *step* del ciclo. All'interno del ciclo vengono dichiarate due variabili: `key` e `value`. La

prima variabile rappresenta il termine che deve essere aggiunto al dizionario, e per poterlo indicare bisogna indicare all'algoritmo di prendere in considerazione la terza colonna del file di testo, assegnando alla variabile `key` la seguente istruzione: `line.split()[2]`.

Ogni *linea* o riga del dizionario viene *splittata* (nel gergo comune, spezzettata) con il metodo `split()`, che genera una struttura dati di tipo *list* contenente i dati separati da uno spazio (o in questo caso, da una tabulazione). Si consideri la prima riga del file *formario* contenente le seguenti informazioni:

```
il_RD      15591484    il    3888938
```

si può notare la presenza del *lemma* più la sua *pos* nella prima colonna, il codice identificativo del lemma, il token/parola nella sua forma flessa non lemmatizzata nella terza colonna e, infine, la frequenza relativa al token nella quarta colonna; con il metodo `split()` applicato alla variabile `line` si ottiene la seguente lista di elementi:

```
['il_RD', '15591484', 'il', '3888938']
```

ogni singolo elemento viene indicizzato separatamente dagli altri elementi in base alla presenza di spazi o altre tabulazioni nella riga di testo presa in esame; per ottenere l'elemento della lista che ci interessa basterà accedere al suo *indice*, ovvero il valore racchiuso fra le due parentesi quadre accanto al metodo `split()`, che in questo caso è pari a 2[17].

Sotto la variabile `key` c'è la variabile `value`, all'interno della quale verrà assegnato il valore relativo alla frequenza del token nel corpus.

2.2.1.2. Termini a bassa frequenza

All'interno del *formario* sono presenti moltissimi termini a bassa frequenza; la maggior parte di questi termini sono forme verbali rare, termini dialettali, parole che non appartengono alla lingua italiana e, in molti casi, parole con errori ortografici; non rappresentano quindi una risorsa utile ai fini della correzione automatica, in

17 La regola canonica prevede che il valore dell'indice di un elemento è pari al valore della sua posizione nella lista meno uno; se `'il'` è il terzo elemento di una lista, il suo indice è uguale a due.

quanto molti di questi termini scorretti potrebbero comparire nei testi da analizzare e non risultare quindi errori. D'altro canto, però, bisogna pure aggiungere i verbi con tempi verbali poco frequenti, per evitare che il programma riconosca, ad esempio, verbi al passato remoto come errori ortografici, generando di conseguenza altri errori.

Per tale motivo è stato applicato un filtro: vengono inserite nel dizionario soltanto parole che hanno una frequenza maggiore a 1; questa condizione viene espressa con l'istruzione `if value > 1`. Gli *hapax*[18] vengono esclusi e non vengono immagazzinati nel dizionario principale.

Alla fine di questa funzione, verrà restituito (`return`) il dizionario contenente un numero cospicuo di termini.

2.2.2. Dizionario di bigrammi (BIG)

La creazione del dizionario dei bigrammi prevede la scrittura di un file esterno contenente i bigrammi stessi con la loro relativa frequenza. La procedura, simile a quella per la creazione del dizionario degli unigrammi, prevede, in primis, la creazione di un dizionario:

```
def creaDizionarioBigrammi():
      with open("xaa.txt") as xaa:
            c = 0
            dizionarioBig = collections.defaultdict(lambda:0)
            parola = re.compile("(\w+'|\w+,|\w+:|\w+;|\W+|\w+\d)")
            for frasi in xaa:
                  frasiU = unicode(frasi, "utf-8")
                  frasiS = frasiU.split()
                  bigrammi = list(bigrams(frasiS))
                  for big in bigrammi:
                  if parola.match(big[0]) or parola.match(big[1]):
                        c = c
                  else:
                     if NWORDS[big[0].encode('utf-8')] and
               NWORDS[big[1].encode('utf-8')] >= 10:
                        big = (big[0].encode('utf-8').lower(),
                     big[1].encode('utf-8').lower())
                        dizionarioBig[big] += 1
      return dizionarioBig
```

Funzione per creare il dizionario dei bigrammi (BIG)

18 Hapax: parole che ricorrono nel testo soltanto una volta.

A differenza della funzione per creare il dizionario degli unigrammi, questa funzione accede direttamente al file contenente la collezione di testi.

2.2.2.1. Pattern matching

Prima di effettuare il ciclo all'interno del corpus, si definisce la variabile `parola`, al cui interno viene compilata un'espressione regolare che ci servirà a scartare le parole che contengono virgole, apostrofi, numeri decimali e altri simboli di punteggiatura. Con questa variabile applichiamo un filtro che ci consentirà di scartare le parole che non sono considerate utili ai fini della correzione automatica, rendendo più snella e "pulita" la struttura dati del dizionario in uscita.

2.2.2.2. Preparazione dei bigrammi

Eseguito l'accesso all'interno del corpus, ogni frase viene codificata in *unicode* (`frasiU`) e splittata (`frasiS`). Dopo aver creato la lista delle singole parole, si procede a scomporre il testo in bigrammi, creando una variabile apposita (`bigrammi`) con all'interno le istruzioni per applicare la funzione di divisione del testo (`bigrams(frasiS)`) da inserire in una *lista*.

Si esegue un secondo ciclo all'interno della lista creata (quella dei bigrammi) e si dettano le seguenti condizioni:

- se la prima o la seconda parola del bigramma contengono segni di punteggiatura, numeri o, sinteticamente, soddisfano i requisiti compilati nella variabile `parola`, il bigramma non viene inserito nel dizionario (aggiornando a se stessa la variabile `c`, e quindi non eseguendo alcuna operazione particolare);
- oppure, se non vengono soddisfatte le condizioni della regex, si provvede a porre un'ulteriore condizione, espressa nei seguenti termini: se ognuna delle due parole del bigramma ha una frequenza (nel dizionario dei monogrammi) maggiore o uguale a 10, il bigramma viene aggiunto al dizionario.

Le parole del bigramma vengono aggiunte nella loro forma minuscola, in modo da snellire il dizionario. Per ogni nuova occorrenza del bigramma, il valore correlato ad

esso viene incrementato di 1. Alla fine dell'esecuzione dell'algoritmo viene restituito il dizionario con tutti i bigrammi.

2.2.2.3. Scrittura del dizionario esterno

La seconda operazione da fare è la scrittura del dizionario. Per farlo, dobbiamo assegnare ad una variabile il contenuto del dizionario, come viene riportato qui sotto:

```
BIG = creaDizionarioBigrammi()
```

Una volta fatto, bisognerà creare uno script apposito per la scrittura del file partendo dai dati che abbiamo raccolto:

```
with open('big', 'w') as big:

     for i in BIG:

           big.write('{:<17} {:<17} {:<17}\n'.format(i[0], i[1],
int(BIG[i])))
```

Con queste righe di codice si eseguono le seguenti istruzioni: viene aperto un file (che però ancora non esiste) su cui scrivere dati (`'w'` sta per *write*) e si scorre il dizionario di bigrammi appena creato (`BIG`) con un ciclo: ad ogni riga del dizionario si usa il metodo `write()` per scrivere i dati nel file che abbiamo creato o dobbiamo creare. Per ottenere una stampa ordinata e in formato colonnare si usa il metodo `format()`, con il quale dichiariamo quanti valori devono essere scritti e a quale distanza (si veda la notazione delle parentesi graffe) e, all'interno delle parentesi del metodo, i valori che devono essere scritti. In questi casi vengono scritti i valori corrispondenti a: prima parola del bigramma, seconda parola del bigramma e frequenza del bigramma, distanziati dal un valore numerico espresso in intero (17, nell'esempio) e con l'allineamento indicato dal tipo dell'operatore di confronto usato(con `<` si intende che le stringhe vengono scritte con allineamento a sinistra, viceversa con `>` si intende che esse verranno scritte con l'allineamento a destra).

2.2.2.4. Caricamento del dizionario dei bigrammi

Una volta finito il ciclo, il file contenente i bigrammi sarà pronto e sarà visualizzabile nella cartella del programma che ha eseguito le istruzioni di scrittura.

Il dizionario dei bigrammi verrà caricato prendendo spunto dal file appena creato (`big`) e seguendo le stesse istruzioni per caricare il dizionario degli unigrammi:

```
def dizionarioBigrammi():
     with open("big") as big:
           diziBig = collections.defaultdict(lambda:0)
           for line in big:
                 keyUno = line.split()[0]
                 keyDue = line.split()[1]
                 value = int(line.split()[2])
                 bigramma = (keyUno, keyDue)
                 diziBig[bigramma] += value
     return diziBig
```

Si può notare la notevole similarità con l'algoritmo per creare il dizionario degli unigrammi, ad eccezione del fatto che qui servono due variabili *key* per il primo e il secondo elemento del bigramma. Con questo metodo si risparmia notevolmente tempo. Se prima per creare un dizionario dei bigrammi ci volevano circa 60 secondi, adesso ci vogliono soltanto meno di 20 secondi.

Le istruzioni per la creazione e la scrittura del dizionario dei bigrammi sono state scritte in un programma esterno (*big.py*), mentre quelle per il caricamento sono state scritte all'interno del programma principale *(sp.py)*.

2.3. Preparazione del testo e tokenizzazione

Il primo compito che svolge il programma è quello di leggere il testo in entrata. Questo compito viene svolto dalla funzione principale che regola il flusso di tutte le altre istruzioni, che prende il testo stesso come unico argomento:

```
def main(testo):

      inputTesto = codecs.open(testo, 'r', 'utf-8')
      rawTesto = inputTesto.read()
      ...
```

La codifica in *unicode* permette al programma di poter codificare parole o simboli particolari. Con il metodo `read()` l'oggetto in entrata viene convertito in una stringa di testo, pronta per essere tokenizzata:

```
def estraiTokens(raw):

     tokens = WordPunctTokenizer().tokenize(raw)
     return tokens
```

```
tokenEstratti = estraiTokens(rawTesto)
```

La funzione estrae tutti i tokens dalla stringa e restituisce una lista di elementi (ovvero le parole tokenizzate); questa lista di elementi verrà *assegnata* su una variabile, sulla quale verranno, da ora in poi, applicate le analisi e i metodi di correzione.

Questa variabile, essendo sottoposta a modifiche, non contiene più, alla fine di tutte le istruzioni del programma, il testo originale dei diari di guerra; per questo, viene dichiarata una seconda variabile che viene utilizzata in seguito per poter confrontare i due testi e capire quali modifiche sono state apportate al testo:

```
testoOriginale = estraiTokens(rawTesto)
```

```
[u'Cos\xec', u',', u'entrammo', u'in', u'quello', u'posto', u',',
u'che', u'c', u"'", u'erino', u'sacchette', u'piene', u'di',
u'terra', u'e', u'poi', u'c', u"'", u'era', u'uno', u'raficello',
u'con', u'una', u'crocetella', u'che', u'l', u"'", u'avevino',
u'scavato', u'i', u'soldate', u'.'....]
```

Lista restituita dalla funzione estraiToken prendendo in input una lettera di Vincenzo Rabito

2.4. Metodi di correzione

Tipicamente, la rilevazione di una *non-word* è abbastanza semplice, poiché basta fare una scansione del testo in entrata ed evidenziare le parole che non sono presenti nel dizionario.

In molti casi vengono etichettate come *non-word* parole particolari che in realtà non

sono errori: ad esempio, spesso il programma ha difficoltà a riconoscere i verbi pronominali (verbi come *ricordar-le*, *pensar-ci*, *amar-ci* ecc...) e le fusioni fra due o più parole (esempi ricorrenti in alcuni testi: *misono*, *maperò*, *dicoraggio* ecc...). Per questo vengono introdotti alcuni criteri che permettono di riconoscere una buona parte di questi tipi di parole, evitando così la generazione di ulteriori errori che possono essere facilmente evitabili.

Poiché per ogni *non-word* viene creata una lista di candidati che, oltre alla parola corretta, contiene anche altre possibili alternative (*serpente*, *sergente*, *esercente* possono essere benissimo dei buoni candidati per *sercente*), è doveroso calcolare le probabilità di ciascun candidato in base alla posizione del testo che occupa, e restituire il candidato con la probabilità[19] più alta. Ancor prima di questa operazione, è necessario che, per ogni metodo di correzione, la generazione di candidati segua determinati criteri: da una parte per non inserire nella lista parole che non sono lontanamente buoni candidati, dall'altra per semplificare le operazioni di scelta e garantire una quanto più esatta scelta del candidato.

Per il calcolo delle probabilità vengono usati diversi metodi, che verranno qui elencati.

2.4.1. Probabilità con parola a sinistra e a destra

Il primo *task* di correzione automatica si occupa di verificare qual'è la probabilità massima di ogni candidato mediante l'osservazione della parola che la precede e che la segue.

Dopo aver eseguito l'accesso alla lista di token che costituiscono il testo, l'algoritmo si occupa dell'individuazione della *non word*. Prima di aggiungere una parola non presente nel dizionario degli unigrammi in una lista (`assente`), bisogna verificare se si tratta di un verbo pronominale o di una fusione fra due parole.

2.4.1.1. Determinazione di un verbo pronominale

Per determinare se si tratta di un verbo pronominale si inseriscono queste condizioni:

- la parola non deve essere presente nel dizionario

19 Il valore che esprime la possibilità di un candidato è espressa nel numero delle sue occorrenze con la parola che la precede e la parole che la segue.

- la parola viene divisa in due stringhe; la stringa che contiene le ultime due lettere della parola deve avere una classe sintattica, o *pos*, corrispondente a un pronome (PRP) o ad un articolo (DT)
- la parola senza le due lettere finali (o con l'aggiunta del carattere 'e' qualora si tratti di un verbo pronominale all'infinito) deve essere presente nel dizionario

Se queste condizioni vengono soddisfatte, la parola non viene trattata dall'algoritmo di correzione automatica grazie ad una variabile booleana (il cui valore è False in partenza) alla quale gli viene assegnato il valore True.

Le successive condizioni dell'algoritmo di correzione conterranno questa variabile: se questa risulterà avere un valore pari a *false*, la parola verrà sottoposta a correzione; se invece sarà *true*, non verrà corretta.

Questa sequenza di condizioni sarà presente in ogni task di correzione automatica, assieme a quella della determinazione di una parola composta da due parole.

2.4.1.2. Determinazione della fusione fra due parole

Per determinare se una parola è composta da due parole unite per fusione è necessario fare:

- un ciclo all'interno della stringa analizzata
- una divisione, ad ogni ciclo, della stringa in due sotto stringhe[20]
- una verifica che consente di misurare l'esistenza di un bigramma, composto dalle due sotto-stringhe, che sia valido e che possa confermare l'effettiva fusione fra due parole nel token analizzato
- una sostituzione del token analizzato con il bigramma valido (qualora il bigramma abbia una frequenza maggiore a 20)

Se si riscontra una fusione fra due parole, oltre alla sostituzione viene pure assegnato un valore *true* alla variabile *split*. Con questa ultima variabile, si fa capire al programma che non devono essere effettuate ulteriori correzioni alla parola analizzata.

20 Il sistema è molto simile a quello della variabile *splits* dell'algoritmo di Norvig.

2.4.1.3. Selezione, generazione di parole e filtraggio dei candidati

Se la parola non viene considerata né un verbo pronominale né una fusione fra due parole, viene aggiunta nella lista `assente`, e, a partire dal singolo elemento di quella lista, viene generata un'altra lista di candidati tramite una funzione esterna denominata `filtraCandidati`.

I candidati vengono generati con l'algoritmo di Peter Norvig, e vengono poi filtrati mediante altri parametri di controllo. Vengono poste delle condizioni nella selezione dei candidati:

1. la lunghezza della stringa generata deve avere una lunghezza pari a quella della *non word* o, al massimo, con una lettera in più o una in meno;
2. il numero di caratteri differenti fra la stringa generata e la parola errata non deve essere superiore a due[21];
3. la differenza fra le due parole, espressa con una ratio tramite i moduli `difflib` e `SequenceMatcher`, deve essere uguale o superiore a 60 (questo significa che un candidato generato deve essere simile graficamente alla *non word* per una percentuale che supera la metà; deve essere *sufficientemente* simile).

Una volta finita l'operazione di filtraggio, la funzione restituisce una lista con i candidati.

2.4.1.4. Calcolo delle probabilità

Per poter calcolare la probabilità dei candidati con i "vicini" viene eseguito un ciclo all'interno della variabile `scelte` per poter effettuare i confronti. Vengono, innanzitutto, definite due variabili (`probSx`, `probDx`), con un valore iniziale pari a zero, che serviranno a immagazzinare i dati relativi alle occorrenze con la parola di sinistra e con la parola alla sua destra.

Dato un candidato, le operazioni per il calcolo, delle probabilità a sinistra e a destra seguono questo iter:

21 Per non inserire fra i candidati parole che risulterebbero troppo disomogenee fra loro.

1. se il token a sinistra è presente nel dizionario e non è un simbolo di punteggiatura, si procede alla creazione di un bigramma contenente il token a sinistra e la parola candidata, si calcola la sua frequenza delle due parole col dizionario dei bigrammi e si assegna il valore a `probSx`;
2. viceversa, si eseguono le stesse operazioni per il token a destra, con l'unica differenza che il valore della frequenza viene assegnato a `probDx`;
3. se il candidato ottiene delle probabilità superiori a zero sia a destra che a sinistra, si pone una seconda condizione annidata che verifica se la distanza del candidato è minore a 2; in caso positivo, viene inserito in una tupla con altri due elementi, ovvero la *distanza* e la probabilità di sinistra moltiplicata per quella di destra.
4. Successivamente, la tupla viene inserita in un array di appoggio, che conterrà tante tuple quanto il numero di candidati che avranno soddisfatto le condizioni precedenti.

2.4.1.5. Ordinamento e selezione del candidato

Alla fine di questo percorso caratterizzato da condizioni, se ne pone un'ultima che verifica se l'array di appoggio è vuoto o meno (nel caso non si riuscisse a trovare un candidato valido per la parola sbagliata); in caso positivo viene effettuato un ordinamento a due versi sull'array di appoggio, che prevede l'ordine ascendente per i valori relativi alla distanza e discendente per i valori relativi alla probabilità, in modo che, come primo elemento dell'array, risulti la parola candidata con distanza minima e con la probabilità più alta (esempio rappresentato qui sotto).

```
appoggio = sorted(appoggio,key=lambda x:(-x[1],x[2]),reverse=True)
```

L'istruzione successiva è quella di modificare la *non word* assegnando il valore del primo elemento della prima tupla della lista, ovvero la parola con maggiore probabilità di essere corretta.

2.4.1.6. Ripristino dei valori iniziali

Le ultime istruzioni dell'algoritmo sono quelle che riguardano lo svuotamento degli array precedentemente usati (`assente`, `scelte` e `appoggio`) e del risettaggio di tutte

le variabili booleane con i valore *false*, in modo che questi elementi possano essere riusati in caso di rilevamento di altre *non word*. Successivamente incrementata di 1 la variabile `c` che funge da numero/indice per la lista del testo in entrata, per la selezione della *non word* in analisi e per la selezione della parola di sinistra (`tokenEstratti[c-1]`) e quella di destra (`tokenEstratti[c+1]`).

2.4.2. Possibilità con parola solo a sinistra – possibilità con parola solo a destra

Il calcolo delle possibilità a sinistra e a destra, in contemporanea, scartano molte parole errate che trovano, come loro vicini, segni di punteggiatura o altre *non word*. Ciò implica che vengono scartate dalla prima analisi e correzione, poiché una delle due probabilità avrebbe valore zero. Per questo vengono introdotti due algoritmi che si occupano di effettuare due analisi separate: la prima prevede di calcolare le possibilità dei candidati soltanto con il vicino di sinistra, la seconda prevede invece il calcolo delle possibilità per il vicino di destra. Le struttura dell'algoritmo e le funzioni per la generazione dei candidati sono identiche al primo algoritmo di controllo, con l'unica differenza che, la funzione che verifica se la parola analizzata è una fusione di due parole, funziona in maniera leggermente diverso, nel senso che suddivide parola in sotto-stringhe cancellando un carattere (tranne il carattere iniziale), operando in maniera quasi identica alla variabile *splits* dell'algoritmo di Norvig.

2.4.3. Non word Zero

Dopo aver effettuato queste tre correzioni, il testo dovrebbe aver ricevuto qualche piccolo miglioramento. Si possono adesso fare altre correzioni che prima, per le regole che si erano dettate negli algoritmi, non si potevano effettuare. Ad esempio, per molte *non word*, i candidati più validi non hanno distanza 1, bensì distanza 2; inoltre, molte parole errate che prima si trovavano in un contesto di isolamento (parole sbagliate che hanno accanto segni di punteggiatura o *non word* stesse), adesso possono ricevere lo stesso trattamento ricevuto per le altre in precedenza.

Ci sono alcune differenze con i metodi di correzione precedenti che è bene elencare in maniera schematica:

1. vengono imposte delle condizioni che riassumono quelle precedenti; vengono fatti controlli sulle probabilità:
    - di sinistra e destra insieme;
    - solo di sinistra e solo di destra;
    - parole che hanno probabilità a destra e a sinistra pari a zero
2. per ogni condizione soddisfatta, possono essere ammessi candidati che hanno una distanza minore o uguale a due (nelle precedenti potevano essere ammessi soltanto candidati a distanza uno);
3. se viene soddisfatta la condizione delle parole con probabilità a sinistra e a destra pari a zero, viene introdotto un ulteriore controllo sulla similitudine fra i candidati e la parola errata.

Rimangono identiche le operazioni di ordinamento e assegnamento del candidato migliore.

Alla fine di tutte queste operazioni, vengono confrontati i due testi, quello originale e quello sottoposto a modifiche, per evidenziare le differenze fra i due e capire quali modifiche ha fatto il programma.

CAPITOLO III

Valutazioni e performance del programma

In questo capitolo si parlerà delle prestazioni del programma una volta avviate le procedure di correzione automatica. Sono stati raccolti dati sugli errori, sulle correzioni giuste e sbagliate che il programma attua e sono state realizzate delle tabelle per rendere esplicite le qualità e i difetti del progetto.

Sono stati raccolti e sottoposti ad analisi ventidue testi[22], ognuno con caratteristiche peculiari dal punto di vista linguistico e stilistico: infatti, per ogni autore, si evidenziano diverse tipologie di errori ortografici. Alcuni scrivono forme di parole che sono il frutto di una fusione fra due (*"misono"* invece che *"mi sono"*) o tre parole (*"emianno"* invece che *"e mi hanno"*); altri usano forme dialettali miste all'italiano (come *"sceparese li capille"* invece che *"scipparsi i capelli"*); altri spezzano le parole (*"in sieme"* o *"a posetamente"* invece di *"insieme"* e *"appostitamente"*).

Esiste quindi una quantità variegata di errori e di stili di errori, e ogni autore ha un propria forma di semi-analfabetismo.

Il programma è stato concepito per poter correggere una buona parte di questi errori, adottando regole universali che possano affrontare anche le diverse tipologie di errori ortografici. Soprattutto, si è fatto in modo di non inserire all'interno del programma regole che possano generare ulteriori errori.

3.1. Esempio di output

Viene riportato qui un esempio di output generato dal programma:

- per prima cosa, vengono riportati i verbi pronominali:

```
calarsi
salvatemi
ferirmi
spararmi
tenermi
```

22 Gli autori delle lettere sono: Severino Bartalini (? - ?; cinque lettere), Francesco Berrettoni (1898 – 1963; due lettere), Paolo Capecchi (1892 – 1915; due lettere), Duilio Faustinelli (1893 – 1991; sei lettere), Luigi Giappesi (1882 – 1962; quattro lettere), Eugenio Lavatori (1886 – 1916; una lettera) e Vincenzo Rabito (1899 – 1981; quattro lettere). In termini di lunghezza, quasi tutte le lettere non superano una pagina, e sono state scritte nei campi di battaglia.

```
farsi
…
```

- successivamente vengono riportate le parole che sono il frutto di una fusione fra due parole:

```
SPLITZERO: a penna 117 apenna
```

- viene poi riportata una stampa degli algoritmi che si occupano del controllo dei vicini (a sinistra e a destra, solo a sinistra, solo a destra); i risultati sono da interpretare in questo modo:

    - nome della funzione;
    - lista dei candidati (parentesi quadre);
    - all'interno delle parentesi tonde sono presenti, in sequenza, il candidato, la distanza dalla *non-word*, frequenza/probabilità
    - fuori dalle parentesi, c'è la *non-word*

```
NON WORD SX DX: [('pensiero', 1, 40160)] penziero
NON WORD SX: [('scrivo', 1, 156), ('privo', 1, 7)] crivo
NON WORD DX: [('vostro', 1, 43)] votro
```

- vengono stampate pure le parole che sono il frutto della fusione di due parole (questa seconda funzione aggiunge uno spazio all'interno della sottostringa):

```
SPLIT UNO: a mano 3155 ammano
```

- viene illustrata la sequenza di correzioni che riassumono tutte le condizioni di controllo delle funzioni precedenti; la stringa finale di testo è stata inserita per capire quale condizione è stata seguita:

    - `'sx'` per indicare che il candidato ha una probabilità con il vicino di sinistra;
    - `'dx'` per la probabilità con il vicino di destra;
    - `'sxdx'` per tutte e due le probabilità;
    - `'zero'` nel caso in cui la *non-word* si trovi in un contesto di *isolamento*, ovvero non abbia vicini validi per un calcolo delle probabilità;

```
NON WORD ZERO: [('ginocchio', 2, 172, 'sx'),
('giacchio', 2, 8, 'zero')] gioncchio
NON WORD ZERO: [('automobile', 2, 40, 'sx')] ottomobile
NON WORD ZERO: [('illeso', 2, 318, 'zero'), ('plesso',
2, 236, 'zero'), ('annesso', 2, 125, 'dx'), ('amplesso',
2, 124, 'zero')allesso
```

- infine, vengono stampati, in due colonne separate, il testo originale e il testo modificato; le frecce servono ad indicare, in maniera intuitiva, i punti del testo in cui è stata fatta una modifica:

```
...
Giappesi                        ""
che                             ""
ora                             ""
tocca                           ""
```

```
anoi                            -----> a noi
,                               ""
bisogna                         ""
andare                          ""
rinforsare                      -----> rinforzare
lalinia                         ""
di                              ""
destra                          ""
che                             ""
…
```

In fondo ai testi, viene stampato anche il tempo impiegato dal programma nell'eseguire tutte le task di correzione automatica.

3.2. Risultati della correzione automatica

In questo paragrafo è presente una tabella con i dati relativi ai risultati della correzione automatica. I testi sono stati divisi per autore e numerati in base alla quantità dei testi riconducibili ad esso. In fondo alla tabella sono riportate anche le percentuali riguardanti la correzione di errori e gli errori non corretti.

Testo	**Errori rilevati**	**Errori corretti**	**Errori non corretti**
bartalini	**4**	**2**	**2**
bartaliniDue	**2**	**1**	**1**
bartaliniTre	**3**	**0**	**3**
bartaliniQuattro	**6**	**4**	**2**
bartaliniCinque	**5**	**3**	**2**
berrettoni	**5**	**4**	**1**
berrettoniDue	**15**	**8**	**7**
capecchi	**13**	**9**	**5**
capecchiDue	**5**	**4**	**1**
faustinelli	**10**	**4**	**6**
faustinelliDue	**4**	**3**	**1**
faustinelliTre	**5**	**2**	**3**
faustinelliQuattro	**3**	**2**	**1**
faustinelliCinque	**3**	**2**	**1**
faustinelliSei	**26**	**10**	**16**
giappesi	**109**	**70**	**39**
giappesiDue	**26**	**20**	**6**
giappesiTre	**4**	**1**	**3**

giappesiQuattro	**13**	**6**	**7**
rabito	**25**	**15**	**10**
rabitoDue	**46**	**23**	**23**
rabitoTre	**29**	**21**	**8**
rabitoQuattro	**38**	**21**	**17**
	Totale errori rilevati	**Totale errori corretti**	**Totale errori non corretti**
	399	**235**	**164**
Percentuali	**/**	**58,89%**	**41,11%**

Come si può ben notare, circa il 60% degli errori presenti nella totalità dei testi viene corretta in maniera adeguata. Questi possono essere considerati buoni risultati, data la semplicità del correttore automatico, in quanto viene superata la soglia della sufficienza delle buone prestazioni.

3.3. Pregi e difetti del progetto

Visti i risultati del progetto, si pone adesso l'esigenza di fare il punto della situazione, e constatare quali possono essere i suoi pregi e difetti e, soprattutto, quali alternative potrebbero essere usate in altri casi per migliorare l'efficienza nella correzione automatica.

3.3.1. Tempo e memoria

In termini di tempo e di memoria, il programma lavora molto bene, in quanto sono state ridotte al minimo le operazioni di *lettura*, che avrebbero compromesso notevolmente le prestazioni, in quanto si opera con dei testi molto grandi (il corpus principale a cui si fa riferimento è grande un gigabyte e mezzo, e il dizionario dei bigrammi è poco meno di ottocento megabyte). Le buone prestazioni sono garantite anche dal fatto che il dizionario dei bigrammi viene creato una volta sola (in un file esterno), e semplicemente letto dal programma, eliminando così la ripetizione di operazioni faticose ogni volta che vengono avviate le procedure di correzione automatica (con questa tecnica si risparmia più del 50% del tempo).

Inoltre, l'algoritmo di Norvig (contrariamente a quanto farebbe una ricerca all'interno del dizionario dei monogrammi), opera molto velocemente e senza

problemi.

Complessivamente, il programma esegue tutte le operazioni di caricamento, ricerca e selezione in un tempo che oscilla dai 67 secondi (per i testi brevi) ai 140 secondi (per i testi più lunghi).

3.3.2. Assenza di un sistema *context-sensitive*

Uno dei maggiori limiti del progetto è quella di non poter effettuare una selezione del candidato in base al contesto. Non è stato possibile, infatti, poter pesare in maniera più efficiente i candidati in base alle parole vicine, in quanto l'ampiezza del corpus non ha permesso il conseguimento di queste operazioni in tempi ragionevoli.

Nelle versioni precedenti di questo progetto si è provato a realizzare un sistema di selezione dei candidati in base al contesto, prendendo spunto dal procedimento proposto da Dan Roth e Andrew R. Golding[23], che è molto semplice ed elementare:

- viene selezionata una *non-word* o una *real-word* sospetta con una probabilità pari a zero;
- per ogni parola scorretta, vengono ricercate le parole vicine (nomi, aggettivi e altre parole che possano creare un contesto);
- viene generata una lista di candidati;
- viene effettuata una ricerca all'interno di un corpus, prendendo in considerazione il candidato e le parole vicine (la ricerca viene effettuata tramite *espressioni regolari* per poterne effettuare una più raffinata e precisa);
- se esistono delle occorrenze fra un candidato e le parole di quel contesto, si dà ad esso un peso maggiore rispetto agli altri potenziali candidati
- viene scelto, infine, il candidato con minor distanza, con maggior frequenza e con maggiore probabilità di trovarsi in quel contesto

Ad esempio, si consideri la parola *sercente* e la lista dei suoi candidati (*sergente, serpente, esercente*, *servente* ecc...); attorno a questa *non word*, con una distanza massima di 10 parole, sono presenti termini come *pugnale*, *caporale*, *bombe*. Si

23 Andrew R. Golding, Dan Roth, *A Winnow-Based Approach to Context-Sensitive Spelling Correction*
http://arxiv.org/pdf/cs/9811003.pdf

capisce bene che queste sono termini che si riferiscono ad un contesto di tipo bellico o militare. Ci si pone, quindi, la domanda: quante probabilità ci sono che una parola come *sergente*, rispetto a *serpente*, compaia in una frase assieme a parole di gergo militare? Oppure: quante probabilità ci sono di trovare una parola come *sergente* seguita da una forma verbale del verbo *dire*?

Sicuramente, se si usasse questo sistema, il programma potrebbe assegnare un peso maggiore alla parola *sergente*, invece che a *serpente*, e riconoscerebbe che il potenziale candidato non potrebbe essere mai (o quasi mai) un animale, bensì un essere umano; e per di più, un essere umano che riveste un ruolo militare.

In poche parole, con questo sistema, il programma avrebbe capacità maggiori, di effettuare correzioni adeguate.

3.3.3. Impossibilità nell'implementare le catene di Markov:

Un'altra grande pecca è rappresentata dal fatto che non è stato possibile usare le catene Markoviane di ordine superiore a 1 per verificare quali siano le probabilità di una frase o di un candidato. Trattandosi di testi storici, scritti da persone poco istruite se non quasi analfabete, in un italiano, peraltro, leggermente diverso da quello scritto o parlato da noi contemporanei, è stato praticamente impossibile fare un calcolo delle probabilità (nonostante la massiva quantità di testi contenuti nel corpus) in quanto quasi tutte avevano 0 come risultato.

CONCLUSIONI
Un punto di partenza per studi futuri

Questo progetto affronta, con strumenti molto semplici, problemi estremamente complessi e di grande rilevanza nell'ambito della linguistica computazionale.

Durante la realizzazione di questo lavoro ci si è accorti della grandissima difficoltà nel far apprendere ad uno strumento non umano come comportarsi di fronte a fenomeni linguistici e ai suoi errori; i metodi decisionali su cui il programma si basa nella scelta del candidato migliore, ovvero sul binomio *minor distanza* e *maggior frequenza*, sono basilari e necessitano di ulteriori raffinamenti, nonostante i buoni risultati ottenuti con essi.

Essendo la lingua uno strumento molto complesso, a pari livello di complessità dovranno stare gli strumenti e metodi informatici per il suo trattamento.

Sebbene molti problemi rimangano aperti, il lavoro realizzato rappresenta comunque la volontà di realizzare un programma che possa costituire un punto di partenza nello studio dei problemi della lingua e nella sua manipolazione ai fini della correzione automatica.

Per questo, esso si può considerare la base per la realizzazione futura di sistemi più complessi, che possano accomunare l'efficienza in tempi e memoria e l'efficienza nei risultati.

Inoltre, questo lavoro è il frutto di una passione, che ancora si deve sviluppare nella sua forma più totale e che, in poche parole, deve ancora crescere.

BIBLIOGRAFIA E WEBLIOGRAFIA

– Gruppo editoriale L'Espresso, Archivio diaristico Nazionale, *La Grande Guerra*: http://espresso.repubblica.it/grandeguerra/index.php

– Daniel Jurafsky & James H. Martin, *Speech and Language Processing: An introduction to natural language processing, computational linguistics, and speech recognition*.

– Andrew R. Golding, Dan Roth, *A Winnow-Based Approach to Context-Sensitive Spelling Correction*: http://arxiv.org/pdf/cs/9811003.pdf

– Peter Norvig, *How to write a spelling corrector*: http://norvig.com/spell-correct.html

– Computational Linguistics & Psycholinguistics Research Center (CLiPS), *Pattern.it*, http://www.clips.ua.ac.be/pages/pattern-it

– Sergio Scalise, Claudia Borghetti, Francesca Masini, Vito Pirrelli, Alessandro Lenci, Felice Dell'Orletta, Andrea Abel, Chris Culy, Henrik Dittmann, Verena Lyding, Marco Baroni, Marco Brunello, Sara Castagnoli, Egon Stemle, *Corpus dell'Italiano – Progetto PAISÀ*

RINGRAZIAMENTI

Si passa adesso alla fase dei ringraziamenti, nei quali vengono menzionate le persone che più degli altri hanno saputo influenzarmi positivamente nella realizzazione di questo lavoro e nella realizzazione personale al di fuori dell'università.

Al mio relatore, Alessandro Lenci, che mi ha accettato nonostante non abbia dimostrato in passato grandi risultati nel campo, ma che, grazie a questo gesto, mi ha fatto innamorare di una materia bellissima e mi ha fatto ricredere su molte cose (in ambito universitario e non); per la sua pazienza, disponibilità e impeccabilità nello svolgimento della sua professione. Lo ringrazio adesso, e lo ringrazierò anche in future occasioni.

A Lucia Passaro, per l'aiuto che mi ha dato durante l'attività di tirocinio; grazie a lei, un'attività che per me sembrava difficile è stata invece una bella esperienza. Ringrazio anche lei per la sua presenza e per la sua disponibilità, spero di incontrarla presto durante il mio percorso universitario.

A Felice Dell'Orletta, che assieme al relatore mi ha insegnato le basi della Linguistica Computazionale e che mi ha dato molte idee sulla realizzazione del progetto. E' stata una vera fortuna averla trovata in questo momento, e ammiro il suo modus operandi, che unisce la semplicità, la simpatia e la competenza: tre doti per me fondamentali nell'arte dell'insegnamento che, spero, di intraprendere. In un futuro meno buio.

A tutti i docenti del corso di laurea di Informatica Umanistica: a pari merito, avete contribuito alla mia formazione professionale, universitaria e anche umana. E' grazie a voi e alla vostra disponibilità che questo corso pionieristico rimane ancora in vita nonostante i tempi di crisi: consapevoli o meno, state portando avanti una vera rivoluzione nel mondo degli studi.

A Pisa, città amata e odiata, città che mi ha preso e sbattuto, cambiato e

rivoluzionato, dove ho conosciuto mille persone e mille personalità, dove ho conosciuto me stesso. Una città che mi ha messo a dura prova e che mi ha fatto conoscere la tranquillità, nella quale mi sono spaccato e ricomposto. Alle sue vie, ai suoi locali, borghi e angoli, piazze e palazzi, che sono e saranno parte della mia vita.

A mia madre e mio padre, a loro devo tutto questo tesoro infinito, difficilmente riassumibile a parole. Grazie a voi sono uscito dalla mia terra, per provare nuove emozioni e nuove esperienze, nuove mentalità e nuovi stimoli. Voi avete permesso tutto ciò, e ve ne sarò infinitamente grato. Un giorno, spero di ripagarvi tutto quello che mi avete offerto, anche se è così grande che ci vorrebbero due vite per potervi restituire tutte le gioie che mi avete dato. Vi amo dal profondo del cuore.

A mia sorella Sara, che il cielo ti ha mandato per potermi fare da sostegno nel mio (particolare) cammino; grazie a te non sarò mai solo per tutta la vita, e neanche tu. Ci siamo resi conto di essere un po' diversi nel corso degli anni, ma questa diversità mi ha permesso di poterti prendere come modello di ispirazione, per perfezionarmi e correggermi. Hai molte qualità che io non possiedo e che vorrei avere; a volte ti invidio per tutta la forza e la grinta che hai. Sono fiero di te.

Ai miei parenti, zii, cugini e a mia nonna, grazie a voi sono cresciuto bene e mi sono sempre sentito protetto. Il calore familiare mi riscalda ancora oggi, nonostante io sia lontano mille miglia da casa. Il solo pensare a voi mi rasserena e mi tiene collegato con la mia terra e i miei ricordi dell'infanzia, la più bella che si possa avere.

A Lorenzo, Simone, Loris e Giulio, i migliori coinquilini nella storia del mondo; con voi c'è armonia, anarchia, rispetto e amicizia; mi sento fortunato a vivere con voi in quella magnifica casa. Non avrei mai potuto avere di meglio.

A Paolo e Alba, che mi hanno aiutato i primi momenti in cui sono stato a Pisa e anche dopo, che mi hanno dato ospitalità e sostegno: siete delle persone splendide e vi ammiro tantissimo e vi voglio un sacco di bene.

A tutti gli amici che ho conosciuto in questo “buco” di città, siete troppi per essere elencati e siete (e sarete) sempre nel mio cuore e vi penserò finché avrò la forza di pensare. Sono stato molto fortunato ad incontrare voi e non generici stronzi come, purtroppo, se ne incontrano molti. Siete il mio punto di forza, di sfogo e di felicità. E' bello potervi dare tutto quello che posso, senza in cambio ricevere nulla se non affetto e rispetto. A voi che siete tantissimi e che a fatica riesco a ricordare tutti, che le stelle vi guidino verso mari calmi e il vento possa tirare a vostro favore.

E poi ringrazio te Chiara, amore mio grandissimo; quando ti ho incontrato, la mia terra era arida, nera, bruciata, e tu eri come una Spiga dorata in mezzo al buio; sei stata preziosa come il pane e generosa con la mia terra. Mi hai sfamato, nutrito, rigenerato, fatto di me una persona migliore. Sei il mio centro di gravità permanente. Mi rendo conto che ti direi mille altre cose, che questo ringraziamento per te è incompleto: spero di poterti dire tutto nel corso della vita. Grazie amore mio, questo lavoro lo dedico anche a te.

Infine, ringrazio tutte quelle cose, persone, idee e luoghi che mi hanno fatto crescere e per cui provo una grande simpatia: ringrazio Ernesto Che Guevara, la Resistenza, Enrico Berlinguer, il P.C.I., la Musica, Ableton, la mia chitarra, i bengalesi che vendono birra in piazza dei Cavalieri, Fabrizio De Andrè, i Radiohead, Johnny Cash, Bob Dylan, gli Who, i Joy Division, Franco Battiato, Francesco De Gregori, Giorgio Gaber, la poesia, il trash in generale, i Partigiani, la Sinistra italiana (che è morta e spero risorga), Lucio Battisti, David Bowie, i Queen, i Pink Floyd, la musica emergente, Maurizio Crozza, Guccini, il non sense, l'Aula Studio Pacinotti, la Biblioteca di Storia e Filosofia, il Sud, Piazza delle Vettovaglie, i kebabbari di Pisa, il Palazzo Blu, il Parco della Cittadella, i vagabondi e i senza tetto, i poveri senza una casa ma con un cuore grande e tanti altri ancora.

Grazie infinite a tutti voi per aver reso la mia vita meravigliosa.