Esercitazione di IA

Fela Winkelmolen

6 marzo 2010

Indroduzione[1]

Lo scopo dell'esercitazione è stato quello di iniziare a lavorare sul problema dell'identificazione della lingua di stringhe di testo, argomento intorno al quale svilupperò la mia prova finale.

Sebbene l'identificazione della lingua di un testo, anche multilinguistico, sia ormai considerato un problema risolto, la precisione degli algoritmi esistenti degrada molto con la diminuzione della lunghezza del testo dato in input. Inoltre anche quando si riesce a determinare che una stringa contiene testo in più lingue non è detto che sia facile capire quali parti esattamente sono scritte in quale lingua.

Nei casi di utilizzo reali si hanno spesso informazione esterne (come potrebbe essere la lingua del software installato in un sistema) che potrebbero essere usati per ottenere un risultato più preciso. È mia intenzione esplorare questa possibilità.

In questa esercitazione mi sono focalizzato sull'acquisire familiarità con il codice di identificazione del linguaggio presente all'interno del repository di KDE, con possibili algoritmi e con problematiche varie legate al problema trattato. Ho inoltre sviluppato un test per determinare la precisione di un certo algoritmo all'accorciarsi della stringa di input e un'applicazione per testare visivamente algoritmi che tentano di determinare la lingua di ogni parola contenuta in un testo.

Ho provato a lasciare, per quanto possibile, alla prova finale le tutto ciò che richiedesse una consistente analisi teorica, nella relazione viene comunque già accennato qualcosa.

Dipendenze e compilazione

Per compilare il progetto servono gli header file di Qt 4, su linux sono presenti pacchetti per tutte le principali distribuzioni (il pacchetto si chiama libqt4-dev per distribuzioni basati su debian, libqt4-devel per distribuzioni basati su rpm)[2].

[1] devo ancora analizzare molta della ricerca che è stata fatta fino ad ora sull'argomento, quello che segue sono le mie deduzione a seguito di quanto ho letto fino ad ora

[2] spero di non aver accidentalmente usato funzioni non presenti in versioni più vecchie, ho testato il codice nei laboratorio dei DISI e con la versione installata lì funziona tutto

Per eseguire la compilazione[3], dalla directory *languageid* lanciare:

```
./build
```

Verranno creati due eseguibili: precisiontest, e visualizer. Il primo, se lanciato dalla linea di comando esegue il test di precisione sull'algoritmo implementato e stampa in standard output i risultati, il secondo è l'eseguibile del visualizzatore. I comandi vanno lanciati dalla directory radice dell'esercitazione (dove vengono creati anche gli eseguibili), in caso contrario non viene trovato il file contenente i trigrammi.

Identificazione della lingua tramite trigrammi

Refactoring del codice esistente

Il codice all'interno del repository KDE si basa a sua volta su uno script di Maciej Ceglowski[4] che implementa l'algoritmo descritto in [Cavnar94]. In pratica vengono confrontati i trigrammi (triple di tre lettere consecutive) più frequenti nel testo dato con i trigrammi più frequenti nelle varie lingue.

In realtà il codice da cui sono partivo lavorava in due fasi utilizzando prima le informazioni sui caratteri Unicode, in cui si suppone sia codificato l'input, per determinare il tipo di scrittura (latina, araba, cinese, ecc.) per successivamente confrontare la stringa soltanto con le lingue che usano le scritture trovate. Io per semplicità ho considerato solo i casi un cui i caratteri sono latini e ho per ora eliminato i controlli sul tipo di scrittura. I miei test case contengono solo testi in inglese, italiano e olandese, ovviamente il programma non ha questa informazione e confronta l'input con tutte le lingue a caratteri latini.

È poi relativamente facile generalizzare di nuovo i risultati ottenuti al caso in cui si hanno più tipi di scritture.

Calcolo della confidenza

Oltre a correggere alcuni errori e imprecisioni nel codice originale ho modificato in modo sostanziale il modo in cui viene calcolata la confidenza. Anziché ritornare le n lingue più probabili l'algoritmo ritorna tutte le coppie $(lingua, confidenza)$ la cui confidenza è strettamente maggiore di 0. Le confidenze vengono normalizzate in modo che la loro somma sia 1[5]. La stima della confidenza è ancora da molto irrealistica, ma migliorarla richiede un lavoro teorico consistente. Per ora mi interessava avere una base da cui poter partire.

[3] il modo in cui viene compilato il programma non è molto elegante, ma è il modo più semplice che mi sia venuto in mente mantenendo le dipendenze al minimo

[4] il cui codice sorgente non sembra essere più disponibile in rete, ma su cui sembrano basate quasi tutte le implementazioni libere che ho trovato

[5] non è detto che questo sia realistico perché, in particolare nella implementazione attuale, c'è una probabilità che nessuna delle lingue ritornate sia corretta, ma è una approssimazione accettabile

Test di precisione

Ho scritto un test per poter analizzare in che misura degrada la precisione all'accorciarsi delle stringhe. Viene testato solo la precisione relativa all'identificazione della lingua prevalente di una stringa.

Diventa così facile analizzare quanto una modifica al codice o l'uso di un'algoritmo diverso influenza la precisione.

I test eseguiti evidenziano una precisione leggermente inferiore a quello menzionato in [Prager99] per un'algoritmo basato su trigrammi. Probabilmente perché l'algoritmo utilizzato da Prager calcola la distanza tra modelli in modo diverso. È da rilevare che Prager non considera la precisione di stringhe più corte di 20 caratteri.

L'output del test è, per ogni lingua, una lista di righe ognuna della forma:

```
m-n: test_corretti/test_totali (percentuale_di_test_corretti)
```

Dove m-n è l'intervallo della lunghezza delle stringhe testate (ho aggregato le lunghezze tre a tre per avere risultati più significativi)

Esempio:

```
3-5: 4/24(16.6667%)
6-8: 9/26(34.6154%)
9-11: 8/22(36.3636%)
[...]
```

La prima riga indica che sono state testate 24 stringhe con lunghezza compresa tra 3 e 5 caratteri ciascuna, di queste è stato identificata correttamente la lingua in 4.

Lingue multiple

Implementazione

Ho implementato un algoritmo molto semplice, che dato una stringa tenta di determinare la lingua in cui è scritta ogni parola, partendo dal presupposto che parole vicine sono generalmente scritte nella stessa lingua.

Vengono calcolate prima tutte le probabilità a priori di ogni singola parola (cioè a prescindere dall'influenza delle parole vicine)[6]. Poi data una parola la sua probabilità di essere in una certa lingua è stata calcolata combinando le probabilità a priori delle altre parole, diminuendo il loro peso con l'aumentare della distanza.

L'algoritmo è relativamente inefficiente e non particolarmente preciso. Lavora sempre in tempo $\Theta(n^2)$, dove n è il numero di parole. Impedire che la lingua

[6] viene considerata parte della parola anche la prima lettera della parola successiva, per sfruttare anche i trigrammi comprendenti l'ultima lettera di una parola e la prima della successiva

di una parola influenzi parole oltre una certa distanza renderebbe l'algoritmo lineare per testi sufficientemente lunghi.

Per una maggiore precisione si potrebbe usare la teoria della probabilità per ottenere un'algoritmo ottimale[7] quando le probabilità di partenza sono corrette.

Un semplice modo per migliorare l'affidabilità nel mondo reale potrebbe essere usare il fatto che è più probabile che la lingua cambia dopo alcuni simboli specifici come il punto, il punto a capo, le virgolette. Si vede infatti che nel mio algoritmo, anche quando vengono individuati correttamente le lingue usate in un testo, in generale il punto in cui il programma suppone avvenga il cambiamento di lingua è abbastanza arbitrario. Un motivo per cui, anche se semplice, non ho ancora implementato questa idea è perché è più interessante vedere la misura dell'(im)precisione senza questo aiuto.

Un'ulteriore complicazione si pone quando si vuole avere un'algoritmo efficiente che operi online, cioè che aggiorni la lingua del testo man mano che questo viene scritto. Per ora non mi sono posto il problema.

Anche in questo caso l'algoritmo è solo una base di partenza per qualcosa di migliore.

Visualizzatore

Ho sviluppato una piccola applicazione che permette di inserire del testo e ne colora le parole in base alla lingua (supposta). Queste informazione vengono per ora aggiornate soltanto quando lo richiede l'utente.

Struttura del codice

Il codice è diviso in tre directory principali

/**lib** contiene il codice che effettivamente tenta di identificare la lingua;

/**tests** contiene il test;

/**ui** contiene il codice relativo al visualizzatore.

Ho cercato di rendere il codice il più possibile estendibile e riutilizzabile, le classi principali sono:

LanguageGuesser una classe astratta con un unico metodo che dato una stringa ritorna un insieme di coppie $(lingua, confidenza)$;

TrigramLanguageGuesser implementa LanguageGuesser utilizzando l'algoritmo descritto sopra;

MultiLanguageGuesser classe astratta con un unico metodo che dato una stringa la divide in sottostringhe ognuna con associata una lingua;

[7] in realtà non è banale definire cosa sia ottimale e l'ottimalità può essere definita in diversi modi

SimpleMultiLanguageGuesser classe che implementa MultiLanguageGuesser e implementata come descritto sopra, il costruttore prende come argomento un oggetto di tipo LanguageGuesser.

Il test è implementato da un metodo che prende un LanguageGuesser come argomento. Nell'eseguibile dell'esercitazione viene eseguito con un TrigramLanguageGuesser.

Il visualizzatore è una classe il cui costruttore prende un MultiLanguageGuesser.

Ho aggiunto anche il codice da cui ero partito, nella cartella *kde*.

References

[Cavnar94] N-Gram-Based Text Categorization, William B. Cavnar and John M. Trenkle, 1994

[Prager99] Linguini: Language Identification for Multilingual Documents, John M. Prager, 1999