Kido, an HMM-driven spelling corrector

Simone Ciccolella[1], Federica Adobbati[2], Daniele Bellani[3], and Francesco Canonaco[4]

[1] *762234, s.ciccolella@campus.unimib.it*
[1] *764300, f.adobbati@campus.unimib.it*
[3] *780675, d.bellani1@campus.unimib.it*
[4] *781239, canonaco@harvard.edu, Full Professor at Harvard e capogruppo di ricerca presso Rotuladores*

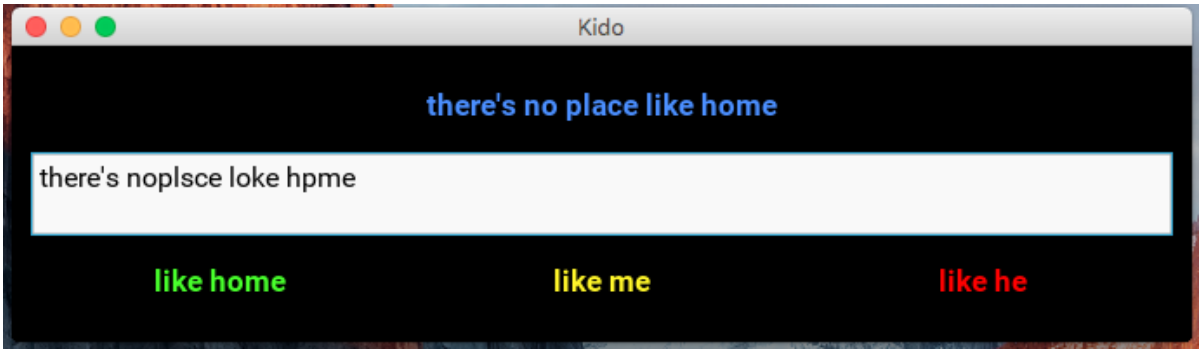
27 giugno 2017

Kido è un correttore di testo in lingua inglese basato su *Hidden Markov Models* e in particolare sull'algoritmo di Viterbi. Il suo principio è ispirato al correttore in [2] in cui viene presentato un correttore che utilizza Viterbi e l'algoritmo di *Forward-Backward* in una HMM generata ad hoc per la correzzione. Prendendo spunto dal suddetto articolo e da diversi algoritmi di confronto di stringhe abbiamo implementato un modello che, sebbene necessiti di ulteriore lavoro e completamento, ha già dei risultati molto promettenti, comparabili a lavori presenti in letteratura e, in alcuni casi, anche migliori. Inoltre, grazie al modo in cui è implementato, Kido è anche in grado, ad ogni parola inserita, di suggerire all'utente tre possibili correzioni e aggiustare le probabilità in coseguenza all'intervento dell'utente.

Abbiamo implementato **Kido** con Python 3 nella sua interezza, per quando riguarda il core del programma (*SmartDictionary* e *HMM*) è sufficiente che i file ad esso relativo siano presenti nella stessa directory del programma. Per poter utilizzare il programma da terminale sono necessarie le seguenti dipendenze:

- Python 3.5.2 +, `https://www.python.org`
- edlib 1.1.2.post2 +, `https://pypi.python.org/pypi/edlib`
- NumPy 1.12.1 + `http://www.numpy.org`
- BioPython 1.69 +, `https://github.com/biopython/biopython.github.io/`

Per l'interfaccia grafica abbiamo deciso di utilizzare il framework multiplatform **Kivy** per Python, reperibile all'indirizzo `https://kivy.org/` che permette una forte customizzazione dell'intera interfaccia oltre alla portabilità su tutti i sistemi operativi, mobile inclusi. Di seguito mostriamo l'immagine di una classica esecuzione di Kido:



1 Metodi

In linea generale Kido legge in input una frase, per ogni parola cerca le parole più vicina ad essa in un dizionario precedente compilato. La parola in input viene poi allineata ad ognuna delle parole trovate e viene calcolata di probabilità che la parola digitata sia, tra tutte quelle trovate, quella analizzata. In seguito calcola la probabilità che la parola corretta si sussegua alla precedente parola nella frase (o che sia la prima parola di una frase). Infine attraverso un algoritmo di Viterbi modificato ad hoc, calcola la frase più corretta. Vediamo ora in dettaglio le diverse fasi.

1.1 Preprocessing

Eseguiamo una fase di preprocessing sulle parole che vengono lette in input con i seguenti step:

- Rendere tutte le lettere minuscole, per non creare problemi di conflitto con il dizioniario.
- Tutte le abbreviazioni vengono sostituite nella forma estesa tenendo in memoria che si tratta di abbreviazioni (utilizzando un underscore _ come simbolo riservato), così da poterle comprimere in fase di post-processing, in particolare le sostituzioni effettuate sono:
  - 's → _s
  - n't → _not
  - 've → _ve
  - 'll → _will
  - 're → _are
  - 'm → _am
  - 'd → _d

In tutti gli altri casi la punteggiatura, i numeri e i caratteri speciali vengono eliminati. Oltre al simbolo riservato di underscore, utilizziamo nel dizionario il simbolo + che distingue i bigrammi dalle altre parole.

1.2 Smart Dictionary

Per controllare la correttezza delle parole della frase è necessario avere un dizionario confronto come *Ground Truth*. Abbiamo preso e unito diversi dizionari:

- **WordsEn**, `http://www-01.sil.org/linguistics/wordlists/english`
- **Names**, `https://github.com/enorvelle/NameDatabases`
- **Bigrams**, generato in modo analogo al training di transizioni. Rimandiamo dunque alla sezione di training per i dettagli.

In totale i lemmi presenti nel dizionario sono più di 709 mila, dunque fare una ricerca sull'intero dizionario risulta essere un'operazione estremamente costosa, nonostante l'utilizzo di una libreria che implementa la distanza di edit in maniera performante riducendo i tempi di calcolo da n^2 a nd, con d distanza di edit.

Per superare questo inconveniente abbiamo definito e implementato **Smart Dictionary**, ovvero un dizionario modificato ad hoc per la ricerca tramite distanza di edit. Lo Smart Dictionary è definito da due array:

- *Dictionary*: contiene i lemmi ordinati per lunghezza e per ordine alfabetico
- *Smart*: è un array di lunghezza pari alla massima lunghezza delle parole contenute in Dictionary ed alla posizione i contiene l'indice j in Dictionary della prima parola di lunghezza i. [• SC: Esprimere meglio il concetto •]

In questo modo se vogliamo tutte le parole che distano al massimo k da una data parola w possiamo ridurre la ricerca alle sole parole che hanno distanza da w di massimo k, ovvero quelle incluse negli indici tra $Smart[|w| - k]$ e $Smart[|w| + k + 1]$, riducendo notevolmente la quantità di parole da processare e di conseguenza i tempi di esecuzione.

Inoltre la classe contiene il metodo:

Edit-Search(w,k) che restituisce la lista ordinata per distanza di tutte le parole a distanza di edit k da w.

1.3 Matrice di perturbazione

Le probabilità di errore nel digitare una lettera sono date la seguente assunzione:

- La lettera c ha il 90% di probabilità di essere correttamente digitata
- Le lettere geograficamente vicine a c sulla tastiera, hanno complessivamente probabilità 0.5% di essere digitate, perciò ogni singola vicina ha probabilità $\frac{0.05}{|vicine_c|}$
- Le rimanenti lettere hanno probabilità complessiva 0.5%, dunque ogni altra lettera ha probabilità $\frac{0.05}{26 - |vicine_c| + 1}$

1.4 HMM Custom Class

Non abbiamo utilizzato nessuna libreria già fatta per gli HMM, ma abbiamo deciso di implementarne una noi adatta alle nostre esigenze. Gli elementi principali della classe sono:

Prior : Prior[w] array delle probabilità a priori che la parola w sia ad inizio frase, imparata dal training.

Señor Prob : Prob[w_1, w_2] contiene la probabilità di passare dalla parola w_1 alla parola w_2, imparata dal training.

Viterbi : applica l'algoritmo di Viterbi modificato ad hoc, per i dettagli rimandiamo alla sezione 1.5.

Train : allena la rete a partire dal training test, rimandiamo a 2.

A livello implementativo le matrici sono in realtà dei *dict* di Python in quanto risultano molto più efficienti delle matrici.

1.5 Custom Viterbi

L'algortimo **Custom Viterbi** si basa su Viterbi, ma i passaggi di calcolo delle probabilità sono differenti, vediamo come funziona l'algoritmo supponendo che l'utente abbia digitato la parola *hpme* sbagliando a scrivere *home*

- Prendiamo le prime l parole $w_1, \ldots, w_l$ che distano al massimo k da *hpme*, utilizzando Edit-Search(*hpme*, k). Sicuramente tale lista conterrà nelle prime posizioni *home* in quanto ha distanza di edit 1. Ogni parola w_i forma dunque uno stato del processo markoviano.

- Ogni parola w_i viene allineata a *hpme* utilizzando l'algoritmo di Needleman–Wunsch [1] generando la parola v, in sequito le parole w_i e v vengono confrontate lettera a lettera utilizzando la matrice di perturbazione, viene calcolata la probabilità di errore della parola inserita, in questo caso *hpme*, con la formula $\prod_{j=0}^{|v|} P(v[j]|w_i[j])$. Chiameremo questo valore P_i nel seguito. Dunque P_i è il corrispettivo della probabilità di una data osservazione nello stato i nell'algorutimo di Viterbi classico.

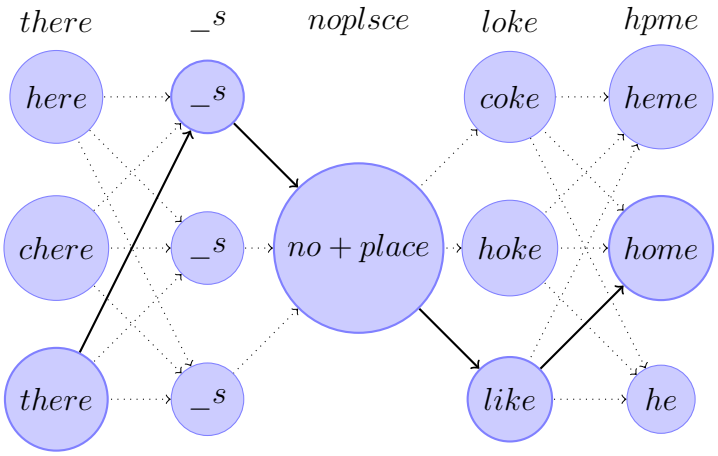
- Ora dobbiamo distinguere tra:

  - Nel caso in cui w_i risulti la prima parola della frase, allora si ottiene il valore che tale parola sia ad inizio frase dalla matrice delle priori, Prior[w_i]. E tale valore viene moltiplicato per P_i ed assegnato nella matrice di calcolo di Viterbi.
  - Altrimenti si ottiene la probabilità di transizione dalla parola precedente a w_i dalla matrice Prob, tale valore viene moltiplicato per P_i e per i risultati delle precedenti iterazioni di Viterbi ed il massimo viene scelto come valore attuale di Viterbi.

- Una volta calcolata tutta la matrice di Viterbi, viene ricostruita la sequenza migliore come nel classico algoritmo.

Una ovvia considerazione da tenere a mente è che nel caso la parola corretta sia un bigramma, ad esempio *istoo* con *is+too*, l'eventuale priori e le transizioni sono valutate di conseguenza; ovvero la transizione tra la precendente parola ed il bigramma è valutata tra la parola precedente e la prima parola del bigramma, mentre la transizione tra il bigramma e la parola successiva viene valutata dalla seconda parola del bigramma; ugualmente la priori viene data dalla prima parola del bigramma, nel caso in cui tale bigramma si trovi ad inizio frase. [• SC: ci sono una marea di ripetizioni, ma d'altronde.. •] Dobbiamo inoltre far notare che le operazioni in realtà vengono effettuate con i logaritmi e quindi le moltiplicazioni diventano somme, in quanto tali probabilità tendono a zero ad una velocità esponenziale.

Vediamo ora un esempio di risultato di Custom Viterbi per la frase *there's noplsce loke hpme*, cominciando con il processo generato:



e concludendo con la tabella di probabilità ad essa relativa:

-12.71720564	**-12.79279799**	**-22.79630544**	-41.38889364	-62.02404146
-27.2382859	-12.79279799	...	-42.83453579	**-58.53913481**
-10.48440738	-12.79279799	...	**-40.16889086**	-59.36092873

1.6 Layered Custom Viterbi

L'algoritmo appena descritto, così come il classico algoritmo di Viterbi, viene eseguito alla fine valutando tutta la frase nella sua interezza e per tale motivo è ottimo in fase di testing. Tuttavia per una applicazione a run-time quale, ad esempio, una tastiera con correttore non si comporta in maniera ottimale, in quanto l'esecuzione risulta molto rallentata.

Per ovviare a questo inconveniente abbiamo sviluppato una versione a *layer* per la costruzione della matrice di Viterbi. Ovvero una volta inserita la prima parola della frase viene

costruita la prima colonna della matrice con le probabilità a priori, mentre all'inserimento di ogni altra parola viene valutata ed aggiunta la colonna relativa alla parola digitata come descritto in 1.5.

Questa particolare versione dell'algoritmo ci ha permesso anche di introdurre una caratteristica molto importante ed utile nell' interfaccia, ovvero la possibilità di suggerire all'utente tre possibili alternative di correzione, che se verranno confermate cambieranno le probabilità della matrice di Viterbi di conseguenza.

2 Training

La fase di training ci ha permesso di ricavare la probabilità a priori delle parole osservate e la matrice di transizione da una parola ad un'altra. In un primo momento abbiamo eseguito il preprocessing sui testi, oltre a quello sulle singole parole descritto in 1.1: abbiamo spezzato i testi in proposizioni in presenza dei caratteri . ; ? ! e abbiamo eliminato tutte le proposizioni che contenevano caratteri speciali, numeri o parole non appartenenti al dizionario. Dai testi così processati abbiamo ricavato la probabilità a priori contando il numero di volte in cui ogni parola compare come prima della frase, assegnando una costante $\pi = 2 * 10^{-4}$ come probabilità a priori delle parole senza osservazioni e normalizzando in modo che la somma delle entrate del vettore facesse 1. Analogamente abbiamo ricavato la matrice di transizione contando il numero di volte in cui due parole sono consecutive in una proposizione, impostando a $\alpha = 10^{-5}$ la probabilità di una transizione non osservata e normalizzando in modo che la somma degli elementi di ogni riga facesse 1. Infine abbiamo generato una lista di bigrammi, in cui sono state inserite tutte le coppie di parole consecutive distinte presenti nel training set.

2.1 Training Set

Abbiamo scelto di utilizzare un training set costituito da:

- un set di articoli della BBC,
- alcuni libri da Project Gutenberg,
- altri libri (Harry Potter, Dune...).

Questa scelta è stata dettata dall'esigenza di apprendere su frasi che utilizzano un inglese moderno e sostanzialmente corretto. Abbiamo infatti evitato romanzi troppo datati che utilizzassero un linguaggio arcaico e frasi dalla messaggistica istantanea, cariche di neologismi e abbreviazioni.

2.2 Considerazioni

Il problema dei nomi propri Abbiamo già evidenziato come ogni frase che contenesse parole non presenti nel dizionario venisse eliminata durante il preprocessing. Per fare in modo

che questa scelta di progetto non risultasse troppo restrittiva, abbiamo integrato il dizionario con un dizionario dei nomi propri. Questo ci ha permesso sia di mantenere un ampio range di frasi per allenare il nostro modello, sia di riconoscere i nomi nella fase di test successiva. A questo punto però si sono palesate altre problematiche, come la preponderanza di alcuni nomi rispetto ad altri nel training set. Nomi come "Larry" ad esempio, sono corretti con "Harry" a causa della saga di Harry Potter nel training set. Per ovviare a questo problema si potrebbe ampliare ulteriormente il training set, in questo modo l'ampia varietà di nomi che si andrebbe a formare riequilibrerebbe la situazione.

Possibili miglioramenti Una fase di training più accurata potrebbe senz'altro portare a migliori risultati nella correzione. Oltre ad ampliare il training set, altri possibili miglioramenti si potrebbero ottenere dall'analisi grammaticale delle frasi, ad esempio sappiamo che il verbo non può mai seguire un articolo Un'idea potrebbe essere quella di inserire questo e altri vincoli grammaticali in modo da rendere più precisa la correzione.

3 Testing

L'ultima fase è stata quella di testing. Anche in questo caso abbiamo eseguito un piccolo preprocessing sul testo per eliminare la punteggiatura e trattare le parole con apostrofi come in 1.1. Utilizzando la matrice di perturbazione 1.3 abbiamo generato delle frasi con circa il 10% di errore. Abbiamo valutato l'errore confrontando la correzione generata tramite il metodo Custom Viterbi 1.5 con la frase originale. [• **FA:** io continuo a preferire super-vitello •] [• **SC:** anche io •] In particolare l'errore viene calcolato in due modi:

- Verificando il numero di lettere corrette:
  1. Per ogni frase si trova la distanza di edit fra la frase originale e quella perturbata;
  2. Si trova la distanza di edit fra la frase originale e quella corretta con Custom Viterbi;
  3. Si sommano fra loro i valori le distanze relative alla stessa tipologia di frase (perturbata e corretta);
  4. Il rapporto tra la distanza relativa alle frasi perturbate e la somma delle lunghezze delle frasi originali restituisce l'errore generato, mentre il rapporto tra la distanza relativa alle frasi corrette e la somma delle lunghezze delle frasi originali restituisce l'errore dopo la correzione.
- Verificando il numero di parole corrette:
  1. Si contano le parole nelle frasi perturbate non presenti nelle frasi originali;
  2. Si contano le parole nelle frasi corrette non presenti nelle frasi originali;
  3. il rapporto fra il valore trovato al punto 1 e il numero di parole totali nelle frasi originali restituisce l'errore commesso perturbando;

4. il rapporto fra il valore trovato al punto 2 e il numero di parole totali nelle frasi originali restituisce l'errore commesso dopo la correzione.

3.1 Testing Set

Come testing set abbiamo utilizzato una parte del dataset Simple Test, che contiene brevi frasi in inglese con un lessico abbastanza variegato. In particolare, ne abbiamo testate oltre 2000. [•SC: referenza al dataset•] [•SC: Aumentare i test!!!! di piuuuuuuuuuuu•]

3.2 Risultati

Coerentemente con il metodo di perturbazione utilizzato, l'errore generato è sempre intorno al 10% sia per quanto riguarda l'errore sulle lettere che quello sulle parole. L'errore dopo la correzione invece in entrambi i casi è circa il 5%. Questo vuol dire che tramite la correzione rimuoviamo circa il 50% dell'errore commesso inizialmente. Questo miglioramento è apprezzabile, anche se potrebbe essere ulteriormente incrementato ad esempio migliorando la fase di training 2. Anche la possibilità data al nostro ipotetico utente di contribuire alla correzione è stata implementata proprio per fornire una maggior accuratezza nell'applicazione pratica rispetto alla sola correzione automatica. [•SC: Una bella tabellozza ci starebbe•]

Esempi Vediamo qualche esempio. Gli errori riportati si riferiscono al calcolo con la distanza di edit.

1. jim had two separate sheets of paper
   jim yad tyosepcratd sheetsof paper
   i had two separate sheets of paper

   Errore generato 0.1667
   Errore correzione 0.0556

   In questo esempio vediamo che ritorna il problema del riconoscimento del nome proprio. L'algoritmo riesce ad individuare e correggere tutte le perturbazioni nella frase, ma evidentemente la probabilità a priori di "jim" e la probabilità di transizione "jim - had" risultano inferiori rispettivamente a quella di "i", "i - had". Questo è sensato e dipende dal training set.

2. the tv person said it was a big storm
   thetv person shid it was a bigztodm
   the tv person said it was a big storm

   Errore generato 0.1351
   Errore correzione 0

   In questo caso la correzione viene eseguita esattamente.

3. mark sits in the front seat when he drives his new car
   mark sits in the frott uepy khef hedlives his new car
   mark sits in the front up he he lives his new car

   Errore generato 0.1481
   Errore correzione 0.1481

   In questo esempio l'algoritmo sbaglia la correzione. Questo avviene per l'elevata concentrazione di errori nello stesso intervallo della frase che rende difficile il riconoscimento delle delle parole corrette anche ad un lettore umano. In questo caso avrebbe forse migliorato la correzione l'apprendimento secondo regole della grammatica come spiegato in 2, così facendo ad esempio difficilmente ci sarebbe stato nella correzione "he he".

4. when the hairdresser finished susan looked at her hair
   whyn te hairdresserfinished sasan lookdd at her hair
   when he had finished a san looked at her hair

   Errore generato 0.0925
   Errore correzione 0.1011

   Questo esempio mostra un altro limite del correttore; non c'è motivo di immaginare che nel training set comparisse il bigramma "hairdresser finished" ed in effetti questo bigramma viene sostituito con "had finished", molto più comune sebbene abbastanza diverso.

4 Sviluppi futuri

Abbiamo fin qui presentato le nostre scelte implementative e i risultati del nostro progetto. Vediamo ora brevemente come ci immaginiamo che potrebbe essere ampliato e migliorato:

- Ottimizzazione del codice; nella costruzione dei metodi abbiamo già adottato alcuni accorgimenti per minimizzare la quantità di calcoli da eseguire, come ad esempio creare metodi appositi per non dover rieseguire ad ogni iterazione l'intero algoritmo di Viterbi 1.6, ma potrebbero essere apportabili ulteriori miglioramenti.

- Sviluppo della fase di preprocessing e di training, di cui abbiamo già parlato 2.

- Gestione degli spazi intra-token (es. "e xcercise"); per semplicità abbiamo supposto che quando ci fosse uno spazio, questo stesse ad indicare la fine di una parola o di un bigramma, con uno studio più approfondito del problema si potrebbe eliminare questa semplificazione.

- Gestione di un numero maggiore di spazi intra-token; per ogni parola; abbiamo supposto che l'utente dimenticasse al più di inserire uno spazio, si potrebbe aggiungere un analisi anche dei trigrammi o addirittura di quadrigrammi nel caso ne dimentichi un numero maggiore.

- Miglioramento dell'interfaccia; si potrebbe ad esempio aggiungere la possibilità di cambiare parole all'interno di una frase senza dover riscrivere tutto daccapo.

Riferimenti bibliografici

[1] Saul B. Needleman and Christian D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443 – 453, 1970.

[2] A. Tarniceriu, B. Rimoldi, and P. Dillenbourg. Hmm-based error correction mechanism for five-key chording keyboards. In *2015 International Symposium on Signals, Circuits and Systems (ISSCS)*, pages 1–4, July 2015.