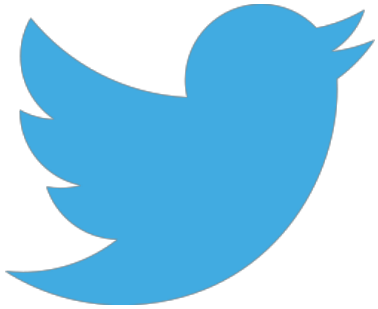
A Random Text Generator based on Markov Chains

Starring:

Twitter Streaming API, Python, Pig



Loris Cro 735424

Luca De Sano 708942

Indice:

1. Le API di Twitter

Twitter offre l'accesso API di vario genere. Tra quelle disponibili le API Streaming permettono di ottenere in tempo reale parte dei tweet che vengono condivisi.

L'uso è semplice:

```
import tweepy

# Impostazione delle chiavi di accesso
auth1 = tweepy.auth.OAuthHandler('xxx','xxx')
auth1.set_access_token('xxx')
api = tweepy.API(auth1)

# Uso della classe
l = StreamListener() # Classe custom che implementa la callback on_message()
streamer = tweepy.Stream(auth=auth1, listener=l, timeout=3000000000 )
streamer.sample()
```

I tweet vengono spediti dagli utenti di Twitter in tutte le lingue e a grande velocità: durante la prima fase di raccolta dei dati in media abbiamo ricevuto 2500 Tweet al minuto, **arrivando a generare 2 GB di dati in meno di 15 minuti**. Abbiamo registrato integralmente i tweet in unicode e con overhead aggiunto dal database (sqlite), ma senza indici.

2. L'obiettivo

Avendo a disposizione molti esempi di frasi, generalmente corte e relativamente facili da decontestualizzare, abbiamo pensato di creare un generatore di frasi randomico.

Per confronto abbiamo poi provato ad applicare il modello alla Divina Commedia ottenendo risultati pessimi.
L'assenza di ripetizioni, la complessità sintattica e le relazioni semantiche tra i versi hanno reso estremamente difficile alla catena la generazione di frasi, oltre a risultare in frasi qualitativamente scadenti.

Esempi presi dalla Divina Commedia:

> da questa tema acciò che tu ti cali, io non perdessi li altri per miei carmi.
>
> che fu la mia, quando vidi ch'i ' era ne l'aere aperto ti solvesti?

Per questo motivo e per gli esempi che abbiamo potuto osservare in internet, siamo convinti che Twitter sia una buona fonte per generare frasi randomiche, addirittura senza la necessità di avere un modello complesso alla base.

3. Il Modello

Per procedere alla generazione abbiamo bisogno che i dati siano memorizzati in maniera precisa e per fare ciò, durante la fase di registrazione dei tweet il testo viene dapprima reso minuscolo e tokenizzato sugli spazi.

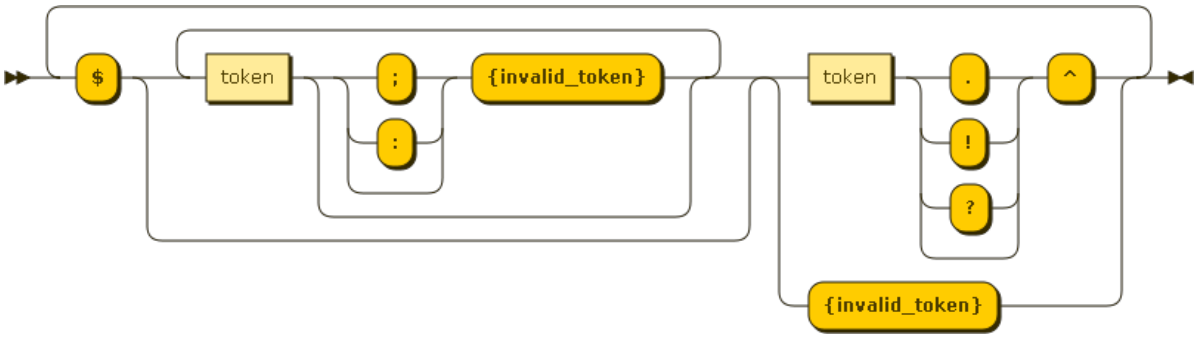
Un token viene considerato valido se rispetta la seguente espressione regolare:

`re.compile(r"^\w+[\w':;.,?!]*$", flags=re.UNICODE)`

Il flag `re.UNICODE` indica che i caratteri unicode vengono considerati come caratteri validi per comporre una parola. In questo modo vengono accettati anche i tweet scritti in caratteri non latini, che sono una grossa fetta del totale dei tweet che circolano.

In secondo luogo la regex esclude URL, hashtag ed acronimi, possibilmente rendendo il testo decontestualizzato. Ciò viene fatto perchè spesso hastag e @nomi vengono utilizzati dagli utenti di Twitter per categorizzare il post come fossero tag, ma capita che niente abbiano a che vedere con la struttura della frase.

A questo punto viene eseguito il parsing dei token secondo questa grammatica (rappresentata leggermente semplificata per comodità):



La grammatica aiuta la catena a generare frasi che abbiamo un inizio ed una fine quantomeno plausibile senza considerazioni sui contenuti.

Infine viene registrata la progressione come una sequenza di coppie, avendo cura di non ritenere come possibili successioni parole tra le quali è stato trovato un token non valido.

4. Gestione di "grosse" moli di dati

Dopo pochi minuti di campionamento ci è parso subito chiaro che non sarebbe stato possibile eseguire le transizioni di stato tramite operazioni matriciali: siamo in poco tempo arrivati ad avere centinaia di migliaia di coppie di token e ciò avrebbe comportato l'avere una matrice di dimensioni non gestibili normalmente dal linguaggio di programmazione utilizzato a meno di librerie particolari che comunque avrebbero rallentato enormemente la manipolazione dei dati.

Come soluzione abbiamo pensato di salvare le tuple in un database SQLite e così facendo abbiamo ottenuto lo stesso risultato dell'utilizzo della matrice ma con il vantaggio di poter ricavare solamente la riga necessaria alla decisione della transizione da compiere da uno stato ad un altro.

Ad ogni step viene eseguita la seguente query per ricavare l'elenco di possibili transizioni da effettuare:

```
SELECT next FROM bigrammi WHERE prev = <token_attuale>
```

Il risultato della query è una lista (**con duplicati**) di possibili token successivi. La catena a questo punto sceglie con probabilità uniforme un elemento dalla lista. **La presenza di duplicati**, al prezzo dell'uso di più spazio su disco, **rende più semplice la selezione pesata del token succesivo**.

Una volta raggiunti ~2GB di database le query hanno iniziato a rallentarsi, l'aggiunta di un indice sulla colonna `prev` ha risolto il problema al prezzo di un ulteriore ingrossamento del database.

5. Migliorie al Modello

Dopo qualche prova ci siamo accorti che una catena di primo ordine presentava il problema di causare cambi inaspettati di lingua.
Alcune parole ('a', 'in', ...) sono comuni a più linguaggi ed essendo presenti molti utenti che scrivono in inglese e spagnolo le transizioni portavano spesso a passare ad una di queste due lingue durante la generazione di frasi

Per risolvere abbiamo semplicemente alzato l'ordine della catena prevedendo transizioni da coppie di token ad altre coppie di token.

In aggiunta, al fine di aumentare la qualità del testo generato abbiamo pensato di 'truccare' il modello nel seguente modo:

Invece di registrare le semplici transizioni successive, nel caso di frasi lunghe almeno 5 token validi, abbiamo registrato anche le possibili transizioni composite:

- Testo: **Speak less and listen more**
- Token: [$, speak, less, and, listen, more, ^]

```
      PREV        -       MIDDLE          -       NEXT

  ( $, speak )    -   "less and listen"   -   ( more, ^ )

  ( $, speak )    -   "less and"          -   ( listen, more )

  ( $, speak )    -   "less"              -   ( and, listen )

  ( $, speak )    -   ""                  -   ( less, and )

  ( speak, less ) -   "and listen"        -   ( more, ^ )

  <eccetera, scorrendo alla base della lista>
```

Le transizioni vengono sempre eseguite da `prev` a `next`, ma in questo modo è possibile eseguire salti in avanti, per i quali il testo intermedio è fornito dal campo `middle`.

Il risultato è che la catena viene generata con la normale logica markoviana, ma il testo finale è 'drogato' dal testo del campo `middle`, risultando in segmenti sensati di testo più probabili.

Da un punto di vista teorico il meccanismo simula lo step di una catena di ordine superiore, dove l'ordine è in base alla lunghezza del salto che si è eseguito.

Ad esempio se dalla tabella precedente si fosse scelto il secondo record, il risultato sarebbe stato equivalente ad uno step normale di una catena di terzo ordine: `( $, speak, less ) ==> ( and, listen, more )`. Più interessante ancora è il caso in cui viene scelto il primo record: in questo caso la transizione avviene tra due ordini diversi.

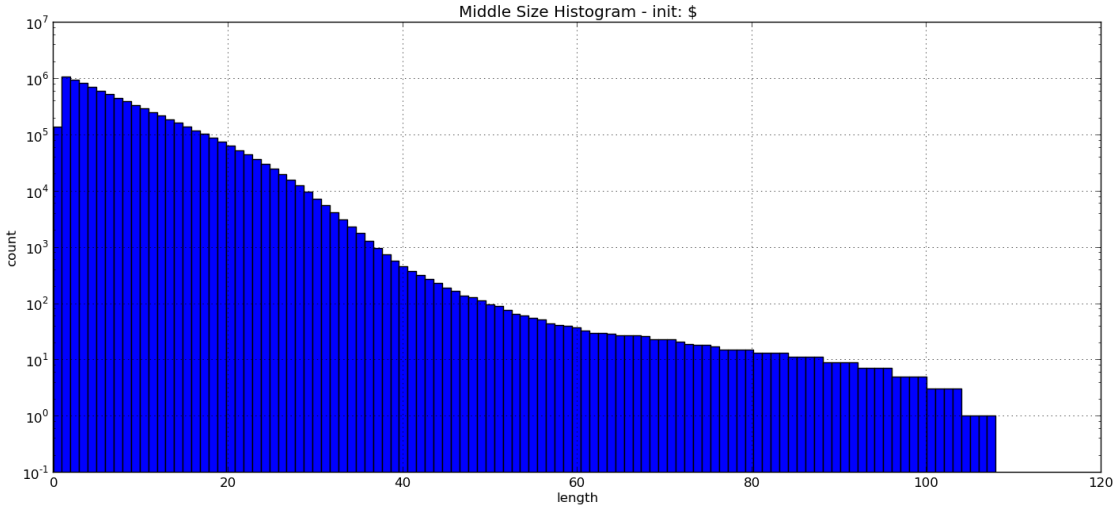
Con questa struttura quindi abbiamo una generazione del testo che è paragonabile ad una catena con la seguente struttura di memoria:

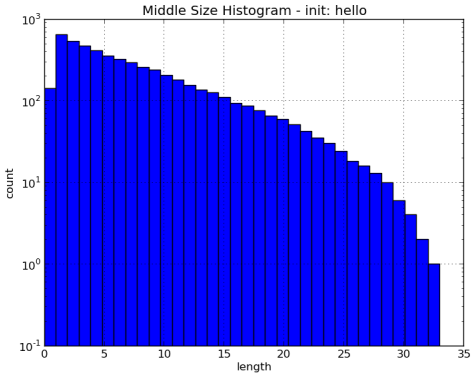
```
2 => N => 2 => N' => 2 => N'' => ...
```

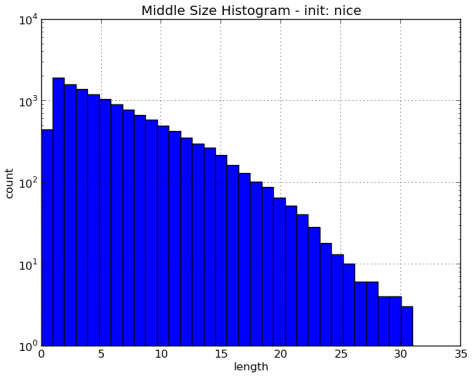
Dove i vari N sono in relazione alla lunghezza del `middle` scelto ad ogni step.

Raffinando ulteriormente:

Osserviamo innanzi tutto alcune distribuzioni delle lunghezze dei middle per alcuni termini:







Le distribuzioni mostrano che con una selezione dello step successivo come descritta nel paragrafo 4, la scelta di uno step corto è più probabile di uno step lungo. Questo fatto fa si che vengano selezionati middle che, pur avendo lunghezza relativamente limitata (7 o 8 token), danno luogo alla selezione di blocchi di testo carichi di significato che una volta inseriti nella frase ne constituiscono una parte troppo lunga, anche in relazione alla tipologia della sorgente (tweet di 140 caratteri).

> I nice mother nature nice . I . can I you handle it , if i I could read it when i was 15 it'd prevent a lot of drama in my life I . my choices . my mistakes . my lessons . I dear future , I v sad as it was the envy of the world , up to us to get in there and get our nhs back on I that eighth grade shit I together over these I girls and even your ex , i'm not fucking I rocket science .

Nell'esempio I indica il cambio di tweet. Si arrivano ad avere in una frase generata sequenze di oltre parole provenienti da un solo tweet.

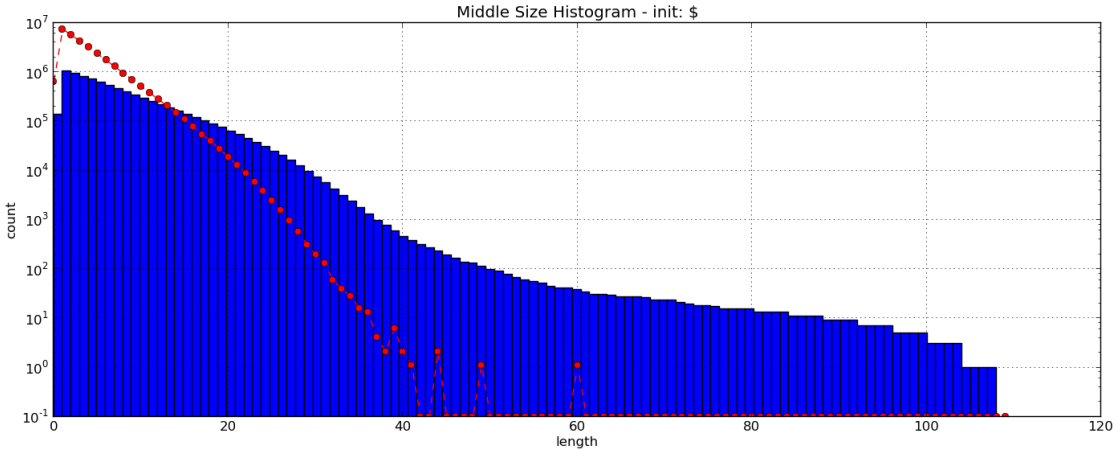
Per ovviare a questo problema e rendere la generazione più varia abbiamo deciso di selezionare lo step successivo anche in funzione della lunghezza del middle secondo una distribuzione di probabilità. In particolare conoscendo la lunghezza del middle di ogni possibile transizione dati gli ultimi due token secondo la seguente query:

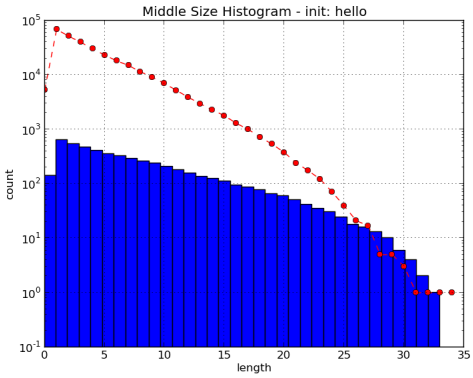
```
select middle_size from from multigrammi where prev = PREV
```

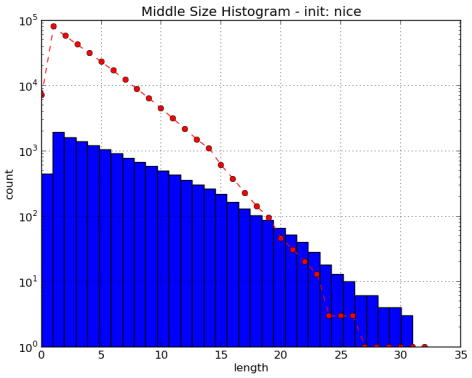
e sapendo quindi il numero di possibili transizioni abbiamo scelto di adottare una distribuzione triangolare sulla lista ordinata rispetto al middle_size delle transizioni possibili di moda pari a

```
1.5/len(set(lunghezze)) * len(lunghezze) # la posizione ~centratata sui middle di lunghezza 2
```

In questo modo la catena ha la possibilità di sfruttare anche i segmenti di testo più lunghi senza compromettere la varietà del testo generato.







6. Considerazioni e possibili migliorie

Twitter è una buona sorgente

In conclusione Twitter è un'ottima fonte per la generazione randomica di testo: le frasi sono generalmente corte e con una struttura semantica mai troppo complessa. Inoltre si trovano molti tweet diversi con combinazioni di parole simili, rendendo interessante il testo generato anche con pochi campioni.

Perchè non abbiamo contato le occorrenze, invece di registrarle duplicate

Una possibile miglioria che sembra ovvia ad una prima occhiata sarebbe quella di aggiungere una colonna nel database in cui si tiene la conta di quante volte si è vista una data progressione. Cosi facendo si ridurrebbe la dimensione del database.

Prima di tutto dobbiamo dire che abbiamo provato a eseguire questo calcolo usando Pig (una piattaforma che facilita la scrittura di computazioni per Hadoop):

```
# Caricamento dei dati
multigrammi = LOAD 'database' as
        (pprev:chararray, prev:chararray, middle:chararray, next:chararray, nnext:chararray, middle_size:int);

# Raggruppo i record identici
gruppi = GROUP multigrammi BY (pprev, prev, middle, next, nnext, middle_size);
```

```
# Genero per ogni gruppo la conta
gruppi_somma = FOREACH gruppi GENERATE FLATTEN(group) as
        (pprev, prev, middle, next, nnext, middle_size), COUNT_STAR(multigrammi) as conta;

# Salvataggio del risultato
STORE gruppi_somma INTO 'result' USING PigStorage();
```

Il risultato di questo calcolo ha portato la riduzione del database dai 5GB originali a ~4.7GB, togliendo circa 3 milioni di record.
Di fatto il guadagno in termini di spazio è risultato trascurabile, in quanto 50 milioni di record (senza dimenticare la grossa ridondanza!) sono un numero irrisorio di campioni in confronto alla dimensione dello spazio campionario, che per vari motivi e per la molteplicità delle lingue in cui viene usato Twitter è un sottoinsieme molto grande di:

$$\bigcup_{n=1}^{140} UNICODE^n$$

In aggiunta questo cambiamento aumenta la complessità dell'estrazione di uno step da `O(1)` a `O(N)` nel caso peggiore con `N` inteso come il numero di possibili transizioni. Questo perchè non è più possibile sfruttare l'accesso diretto ad una chiave della lista (in python le liste sono implementate come vettori).

Le proprietà della catena di Markov

Facendo un'analisi della struttura della catena che viene a generarsi dal campionamento dello stream di tweet, si nota come questa presenti potenzialmente sia stati transienti sia stati assorbenti, i quali presentano le rispettive problematiche:

Stati transienti:

Qualora si usasse un tipo di struttura dati diversa da tabelle relazionali, si potrebbe pensare di comprimere gli stati transienti per facilitare la computazione. Ad esempio si potrebbe pensare di utilizzare un database nosql, in particolare un database su grafi.

Stati assorbenti:

La grammatica permette di immagazzinare parti di testo che non terminano con un carattere di fine frase (^). Questo succede quando una frase termina con un token non valido, cosa che spesso accade nei tweet. Per questo motivo è possibile che la catena vada in uno stato dal quale non è possibile raggiungere un simbolo di fine frase.

La procedura da noi implementata genera il testo ricorsivamente, in questo modo, nel caso in cui ci si trovi in uno stato cattivo, è sempre possibile risalire lo stack per tentare una nuova generazione del testo, similmente ad una procedura di ricerca in profondità.

Migliorie

Indice di Novità(tm)

Supponiamo di raccogliere enormi quantità di dati. Ad un certo punto ci si scontra con l'effetto dei diminishing returns: l'aggiunta di un nuovo tweet non porta più la stessa quantità di informazione dei tweet raccolti quando ancora si avevano pochi dati.
Per rispondere a questa problematica abbiamo pensato di calcolare l'Indice di Novità(tm) così definito:

Sia c1, ..., cn la sequenza di coppie di token che rappresenta il tweet.

$$\frac{1}{n} \cdot \sum_{k=1}^{n} \frac{1}{(occorrenze(c_k)+1)^2}$$

A questo punto si decide se inserire o meno il tweet in memoria con una probabilità pari al valore ottenuto:

Un tweet che contiene (sequenze di) parole nuove viene quasi sicuramente inserito nel database.
Un tweet che contiene solo sequenze già conosciute ha meno probabilità di essere aggiunto tanto più è comune in memoria.

In questo modo il rapporto tra sequenze più comuni e sequenze meno comuni viene mantenuto ma senza la necessità di immagazzinare

necessariamente ogni tweet.

Migliore grammatica

Un'altra miglioria possibile consiste nel raffinare la grammatica per tenere conto di apostrofi (elisione, genitivo sassone, ...) e far leva su altri costrutti sintattici (ad esempio controllare nel tweet citazioni racchiuse tra virgolette).

In aggiunta si potrebbe aggiungere il supporto alle lingue asiatiche che fanno uso di ideogrammi che, èur essendro presenti nella base di dati, a causa delle differenti regole sintattiche rispetto alle lingue che adottano l'alfabeto latino, non possono essere processate con la stessa metodologia utilizzata per queste ultime.

Sono possibili diverse migliorie prima di sfociare nell'analisi logica del linguaggio.