Bayesian Blocks con funzione esponenziale a tratti

Alessio Rondelli

6 febbraio 2024

Capitolo 1

Report

1.1 Cenni Teorici

Siano dati $t = (t_1, ..., t_n)$ e $x = (x_1, ..., x_n)$ che corrispondono rispettivamente a una sequenza di reali non-negativi crescente e una sequenza di numeri naturali della medesima lunghezza. Noi cerchiamo di trovare la funzione esponenziale a tratti che ottimizzi una certa funzione di fitness rispetto questi dati. Nello specifico t è una sequenza temporale e x è una sequenza di valori provenienti da bin.

Dati t e x si cerca di partizionare l'intervallo identificato dalla sequenza temporale t mediante una successione di sottointervalli (chiamati blocchi) ove in ognuno di questi si cerca la migliore funzione esponenziale da cui i dati di x provengono.

Nello specifico la funzione esponenziale che cerchiamo di fittare in un blocco è nella forma $\gamma e^{a(t-t_n)}$, dove t_n è l'estremo destro del blocco, γ è il valore della funzione alla fine del blocco e a è il termine del decadimento dell'esponenziale. All'interno di ogni blocco si cerca di fittare la funzione ottimizzando i valori γ e a.

La funzione di fitting scelta è la log-likelihood di cui viene riportata la costruzione nel caso di funzione arbitraria $f(n)$ per generalità.

Theorem 1. *Sia $t \in \mathbb{R}^n$ successione crescente di tempi reali. Sia $x \in \mathbb{Z}^{+n}$ variabile aleatoria di legge $\bigotimes_{i=1}^{n} Poisson_{f(t_i)}$, dove $f : [t_1, t_n] \to \mathbb{R}^+$. Evidentemente le marginali sono indipendenti tra loro. Allora la log-likelihood di una realizzazione $\tilde{x}$ sarà:*

$$\log L(\tilde{x}) = \sum_{i=1}^{n} \tilde{x}_i \log f(t_i) - \sum_{i=1}^{n} \log(\tilde{x_i}!) - \sum_{i=1}^{n} f(t_i)$$

Dimostrazione. La likelihood di ognuna delle marginali sarà

$$L_i = \frac{f(t_i)^{\tilde{x}_i} e^{-f(t_i)}}{\tilde{x}_i!}$$

siccome la sua legge è di Poisson. Grazie all'indipendenza tra le componenti di x la likelihood sarà

$$L = \prod_{i=1}^{n} \frac{f(t_i)^{\tilde{x}_i} e^{-f(t_i)}}{\tilde{x}_i!}.$$

Passando alla log-liklihood otteniamo

$$\log L(\tilde{x}) = \sum_{i=1}^{n} (\tilde{x}_i \log f(t_i) - f(t_i) - \log(\tilde{x}_i!))$$

che conclude la dimostrazione. □

Remark 1. *Nel nostro caso useremo la funzione $f(t_i) = \gamma e^{a(t_i - t_n)}$, per cui la nostra log-likelihood sarà*

$$\log L(\tilde{x}) = \sum_{i=1}^{n} \tilde{x}_i(\log \gamma + a \cdot (t_i - t_n)) - \sum_{i=1}^{n} \log(\tilde{x_i}!) - \sum_{i=1}^{n} \gamma e^{a(t_i - t_n)}.$$

Forti di questa base teorica si passa poi alla statistica di Cash ove l'ultima somma si converte in integrale:

$$\log L(\tilde{x}) = \sum_{i=1}^{n} \tilde{x}_i(\log \gamma + a \cdot (t_i - t_n)) - \sum_{i=1}^{n} \log(\tilde{x_i}!) - \int_{t_1}^{t_n} \gamma e^{a(t - t_n)} dt. \quad (1.1)$$

L'uso della log-likelihood è centrale in quanto ci permette di avere la proprietà di additività sui blocchi che è propedeutica all'uso dell'algoritmo di programmazione dinamica ideato da Scargle, ripassiamo i passaggi fondamentali di questo algoritmo:

- Partendo dal primo bin esiste una sola partizione di questo, perciò è nota la partizione ottima di un solo bin.
- Supponiamo di conoscere la partizione ottima dei primi R bins. Analizzando i primi $R + 1$ bins notiamo che grazie alla proprietà di additività delle log-likelihood (o di qualunque funzione di fitness legittima) la partizione ottima sarà divisibile in 2 parti: la partizione ottima dei primi r bins e l'ultimo blocco che va dal bin $r + 1$ fino al bin $R + 1$. Di conseguenza per ogni r da 1 a R calcoliamo la fitness della partizione (che è uguale alla somma delle fitness su ogni blocco della partizione) e selezioniamo r il valore che massimizza la fitness della partizione.
- Abbiamo così trovato la legge induttiva, iterando fino a N (il numero di bin) otterremo l'ottima partizione.

Remark 2. *Notiamo che ad ogni passaggio dell'esecuzione si mettono a confronto le log-likelihood delle partizioni dei primi R bins: in particolare indipendentemente dalla funzione di fitting notiamo che è presente un termine che dopo aver fatto la somma sarà costante su ogni partizione: $-\sum_{i=1}^{n} \log(\tilde{x_i}!)$ e per questa ragione a fini di massimizzazione può essere ignorato.*

Di conseguenza la funzione di fitting che verrà usata nei fatti sarà:

$$\log L(\tilde{x}) = \sum_{i=1}^{n} \tilde{x}_i \log(f(t_i)) - \int_{t_1}^{t_n} f(t) dt \quad (1.2)$$

$$\log L(\tilde{x}) = \sum_{i=1}^{n} \tilde{x}_i(\log \gamma + a \cdot (t_i - t_n)) - \int_{t_1}^{t_n} \gamma e^{a(t - t_n)} dt \quad (1.3)$$

Notiamo però che in generale la log-likelihood dipende dai parametri della funzione da fittare sui dati, questo comporta la necessità di calcolarli mediante massimizzazione della log-likelihood.

1.1.1 Calcolo dei parametri di fitting

In ogni blocco, che possiamo identificare matematicamente come una realizzazione di una variabile aleatoria $x \in \mathbb{N}^n$ e da una successione finita crescente di tempi $t \in \mathbb{R}^n$, è necessario ottimizzare i parametri di fitting che definiscono la funzione $f(n)$. In generale la funzione da scegliersi dovrebbe avere delle proprietà abbastanza chiare e se possibile essere sufficientemente liscia da garantire il calcolo di derivata almeno al primo ordine. Infatti se così dovesse essere sarebbe possibile in generale usare la tecnica della salita del gradiente per trovare i parametri ottimi per massimizzare la log-likelihood.

Nel nostro caso la funzione è C^∞ e perciò siamo in grado di derivare quante volte vogliamo, il che sarà centrale a fini di implementazione, ma non solo: difatti la semplicità della funzione ci permette di trovare analiticamente uno dei parametri ottimi del modello che permette di velocizzare usando un algoritmo molto rapido.

Di conseguenza siccome vogliamo massimizzare (1.3) cerchiamo intanto i punti critici della funzione di fitness sul k-esimo blocco:

$$\begin{aligned}\log L_k(\tilde{x}) &= \sum_{i=1}^{n} \tilde{x}_i(\log\gamma + a\cdot(t_i - t_n)) - \int_{t_1}^{t_n} \gamma e^{a(t-t_n)}dt \\ &= N_k \log\gamma + a\tilde{x}_i \sum_{i=1}^{n}(t_i - t_n) - \gamma\left(\frac{1-e^{-aT_k}}{a}\right),\end{aligned}$$

dove T_k è la lunghezza del temporale del blocco. Cerchiamo ora i punti critici

$$\frac{\partial F_k(\tilde{x})}{\partial\gamma} = \frac{N_k}{\gamma} - \left(\frac{1-e^{-aT_k}}{a}\right)$$

Per cui imponendola uguale a 0 otteniamo

$$\gamma = \frac{aN_k}{1-e^{-aT_k}}.$$

Di conseguenza

$$\begin{aligned}F_{max,k}(\tilde{x}) &= N_k \log\left(\frac{aN_k}{1-e^{-aT_k}}\right) + a\sum_{i=1}^{n}\tilde{x}_i(t_i - t_n) - \frac{aN_k}{1-e^{-aT_k}}\left(\frac{1-e^{-aT_k}}{a}\right) \\ &= N_k \log\left(\frac{aN_k}{1-e^{-aT_k}}\right) + a\sum_{i=1}^{n}\tilde{x}_i(t_i - t_n) - N_k \\ &= N_k\left(\log\left(\frac{aN_k}{1-e^{-aT_k}}\right) + aS_k - 1\right), \qquad (1.4)\end{aligned}$$

dove $S_k = \frac{1}{N_k}\sum_{i=1}^{n}\tilde{x}_i(t_i - t_n)$.

Per valori sensibili di N_k e T_k la funzione è concava e quindi è sufficiente trovare un punto critico (se esiste) per trovare l'unico massimo. Siccome la funzione è di singola variabile ed anche molto semplice da derivare si è scelta una generalizzazione del metodo di Newton, il metodo di Halley, per trovare lo 0 della derivata della nostra funzione $F_{max,k}$.

La scelta di questo algoritmo è dovuta sia a questioni di stabilità numerica che a questioni di velocità di convergenza (è molto stabile e veloce).

1.2 Implementazione

Per fare quanto descritto sopra ci appoggiamo alla libreria astropy per l'implementazione dell'algoritmo di Scargle (Bayesian Blocks) nel caso generale. Per implementare questa nuova funzione di fitness si è scelto di estendere la classe `Events` che implementa nel caso di funzione di fitting costante l'algoritmo. Difatti modificando il metodo `fit` è possibile mantenere tutta la struttura dell'algoritmo e cambiare soltanto il calcolo della funzione di fitness.

Dal punto di vista più applicativo per applicare il metodo implementato è sufficiente fare

```
from astropy import bayesian_blocks
from bb_exponential import ExponentialBlocks_Events
bayesian_blocks(t,x,fitness=ExponentialBlocks_Events,ncp_prior=***)
```

Al momento è fondamentale specificare l'`ncp_prior`, tramite ulteriori sviluppi sarà possibile calcolare l'`ncp_prior` ottimo in funzione di `N` e di `p_0`: rispettivamente la somma di tutti i dati `x` e il false detection rate.

Questo metodo però ritorna solamente gli estremi dei nostri blocchi e non anche i parametri calcolati per la funzione di ottimo fitting sui blocchi. La scelta che si è fatta è stata quella di ricalcolare i parametri a fine esecuzione, si riconosce che dal punto di vista del mero costo computazionale questa non è la scelta migliore (è necessario ricalcolare i parametri in un secondo tempo), tuttavia le motivazioni sono queste:

- Questo è il comportamente del metodo della libreria scipy, estenderlo per avere in output sia gli estremi che i parametri richiede la riscrittura quantomeno del metodo `fit` della classe `FitnessFunc` di astropy affinchè in fase di "peeling off" dell'array `last` oltre al calcolo dell'ottimo estremo si ottengano anche i parametri ottimi, l'implementazione di questo sarebbe lenta e romperebbe la compatibilità con precedenti implementazioni.

- Il costo computazionale è sì maggiore, ma di una quantità abbastanza irrisoria rispetto al costo dell'algoritmo. Infatti l'algoritmo di programmazione dinamica durante l'esecuzione richiede di effettuare il calcolo di fitting sull'ultimo blocco $O(N^2)$, invece il costo di calcolo sul singolo blocco si può pensare come costante, di conseguenza dato che ci si aspetta che il numero di blocchi identificati sia molto più piccolo rispetto al numero di bins il costo di ricalcolo risulta irrisorio rispetto al calcolo di partizionamento ottimo.

Dal punto di vista applicativo il ricalcolo è implementato mediante il metodo `get_parameters` della classe `ExponentialBlocks_Events` in questa maniera:

```
edges = bayesian_blocks(t,x,fitness=ExponentialBlocks_Events, ncp_prior=***)
edge_l,edge_r = edges[0],edges[1]
params = ExponentialBlocks_Events(ncp_prior=***).get_parameters(edge_l,edge_r,t,x)
```

e i parametri vengono restituiti come un dizionario di chiavi `a` e `gamma`.

Evidentemente allora questo metodo deve essere iterato su ogni coppia di estremi per ottenere i parametri di ogni blocco.

1.3 Ottimizzazione iperparametri

Il modello è non parametrico, quindi verrebbe da pensare che non ci siano parametri da ottimizzare in quanto questi vengono già ottimizzati dall'algoritmo in fase di esecuzione, c'è però un problema: il numero di blocchi identificato.

Per semplificare il problema possiamo pensare che il numero dei blocchi identificati e la variabile aleatoria di cui i dati x sono una realizzazione siano indipendenti. Facendo questo la likelihood complessiva sarà il prodotto tra le likelihood dei dati in ogni blocco e la likelihood che il numero di blocchi identificati sia quello corretto.

Sarebbe possibile dare qualunque tipo di distribuzione discreta alla variabile aleatoria del numero di blocchi, si sceglie però di usare una distribuzione geometrica in quanto di fatto ci aspetiamo che ci sia un maggiore peso da dare a piccoli numeri di blocchi rispetto a grandi numeri. Di conseguenza fissiamo che

$$P(N_{blocks} = n) = P_0 \cdot \alpha^n \cdot \mathbb{1}(n \leq N)$$

con $\alpha \in (0, 1)$ e P_0 costante di normalizzazione.

Naturalmente è possibile calcolare che $P_0 = \frac{1-\alpha}{1-\alpha^{N+1}}$.

Avendo calcolata questa likelihood, alla fitness del blocco sarà necessario sommare $\log \alpha$ per normalizzare rispetto al numero di blocchi e rendere più improbabile una partizione con troppi o troppi pochi blocchi.

Il problema sorge quando si rende necessario il calcolo di questa costante $ncp\_prior = \log \alpha$, infatti è evidente la sua dipendenza dal numero di bin N, ma riflettendoci è possibile notare che più è alto il valore atteso della variabile aleatoria N_{blocks}, più è probabile che ci siano falsi positivi, ovvero che venga identificato l'estremo di un blocco quando in realtà questo non dovrebbe succedere. Non sono note formule esatte, però è possibile seguire una di queste tecniche empiriche per trovare la scelta ottima (o quantomeno abbastanza[1] buona) di *ncp_prior*:

1. Se possibile generare dati artificiali (simili a quelli su cui poi l'algoritmo dovrà essere applicato) si può testare l'algoritmo su vari valori possibili di *ncp_prior* e poi calcolare *p_0* rispetto a questi, infine si sceglie il valore di *ncp_prior* più basso (così che l'algoritmo sia il più sensibile possibile) che garantisca un false detection rate minore di un valore fissato in precedenza, 0.05 ad esempio. Tenere conto però della dipendenza del parametro da N, sarà quindi necessario farlo anche per vari valori di N.

2. Se si facesse esattamente come sopra ma su dati reali, questo potrebbe produrre dell'overfitting sui dati reali, e perciò è consigliabile farlo mediante k-cross-validation:

   Presi i dati reali questi si suddividono casualmente in k sottoinsiemi della stessa cardinalità, dopodichè iterativamente si prende l'i-esimo sottoinsieme, che sarà l'insieme di verifica, e tutti gli altri, che saranno i nostri dati

[1] Potrebbe sembrare strano che si sia usata la frase "abbastanza buona", in effetti il filo centrale del metodo è l'ottimizzazione non ottenere un risultato buono ma non perfetto, tuttavia nei fatti il metodo è sufficientemente rigido da restituire i medesimi risultati a patto che questo parametro sia anche soltanto vicino a quello ottimo. Dal punto di vista matematico questo è triviale in quanto il numero di blocchi identificati è una variabile discreta mentre *ncp_prior* è un parametro continuo.

di allenamento. Si ottimizza la scelta di *ncp_prior* sui dati di allenamento rispetto a qualche funzione di fitness (ad esempio il numero di blocchi identificati, l'errore quadratico medio o il false detection rate). Dopo aver fatto questo si calcola la medesima fitness sui dati di verifica e si usa quel valore per confrontare i risultati al variare dell'*ncp_prior*. Quello la cui fitness sui dati di verifica è più alta è l'*ncp_prior* che viene scelto.

3. Esiste anche un'ulteriore tecnica che non viene qui riportata, è presente nella sezione 2.7 di Scargle (2013).

Una volta che sono state fatte queste stime si può interpolare un polinomio su questi dati per avere una stima di *ncp_prior* in funzione di p_0 e di N.

Al momento della scrittura i risultati devono finire di calcolare e quindi non si hanno ancora le formule empiriche precedentemente descritte.

1.4 Osservazioni senza particolare ordine

Remark 3. *In fase di calcolo della log-likelihood la nostra funzione* (1.4) *non ha un buon comportamento computazionale, in particolare se la variabile a risulta negativa il termine* $-aT_k$ *potrebbe essere abbastanza alto da causare un overflow nell'esponenziale, l'algoritmo in tal caso non ha problemi in quanto risulterebbe una log-likelihood sul blocco di* $-\infty$*, però questo potrebbe causare problemi ai fini dell'ottimizzazione. Analogamente se a risulta troppo vicino allo* 0 *potrebbe succedere che l'argomento del* log *possa esplodere se il denominatore dovesse diventare* 0 *esatto. Per questi due casi si propongono due approssimazioni numeriche che dovrebbero evitare questo genere di problemi:*

- *Tramite lo sviluppo di Taylor dell'esponenziale in caso di a vicino allo 0 è possibile approssimare* (1.4) *con*

$$N_k \left(\log \left(\frac{N_k}{T_k} \right) + aS_k - 1 \right).$$

- *Siccome se a dovesse risultare molto minore di* 0 *l'esponenziale dominerebbe il termine* 1 *è possibile ignorare il termine 1 a denominatore, di conseguenza approssimiamo* (1.4) *con*

$$N_k \left(aT_k + \log(-aN_k) + aS_k - 1 \right).$$