Bayesian Blocks con funzione esponenziale a tratti

Alessio Rondelli

12 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. 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), in ognuno dei quali 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 alla statistica di Cash, ove l'ultima somma si converte in integrale:

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

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 così che massimizzi la fitness della partizione.
- Abbiamo in questo modo 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:

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

e nel nostro caso specifico

$$\log L(\tilde{x}) = \sum_{i=1}^{n} \tilde{x}_i(\log\gamma + a\cdot(t_i - t_n)) - \int_{t_1}^{t_n} \gamma e^{a(t-t_n)} dt. \tag{1.2}$$

Tuttavia notiamo che in generale la log-likelihood dipende dai parametri della funzione da fittare sui dati: ciò 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 teoriche abbastanza note 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 il calcolo usando un algoritmo molto rapido.

Di conseguenza, al fine di massimizzare (1.2) cerchiamo 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,

$$\begin{aligned} F_{max,k}(\tilde{x}) &= N_k \log\left(\frac{aN_k}{1 - e^{-aT_k}}\right) + a\sum_{i=1}^{n} \tilde{x}_i(t_i - t_n) - \frac{aN_k}{1 - e^{-aT_k}}\left(\frac{1 - e^{-aT_k}}{a}\right) \\ &= N_k \log\left(\frac{aN_k}{1 - e^{-aT_k}}\right) + a\sum_{i=1}^{n} \tilde{x}_i(t_i - t_n) - N_k \\ &= N_k\left(\log\left(\frac{aN_k}{1 - e^{-aT_k}}\right) + aS_k - 1\right), \end{aligned} \tag{1.3}$$

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, perciò è sufficiente trovare un punto critico (se esiste) per trovare l'unico massimo. Siccome la funzione è di singola variabile e 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.

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 la funzione di fitness per il caso esponenziale si è scelto di estendere la classe `Events`, che implementa l'algoritmo nel caso di funzione di fitting costante. Difatti modificando il metodo `fit` è possibile mantenere tutta la struttura dell'algoritmo e cambiare soltanto il calcolo della funzione di fitness.

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=***)
```

Il parametro `ncp_prior` è possibile specificarlo a priori oppure lasciare la sua determinazione al calcolatore. Le specifiche sul calcolo automatico di un buon `ncp_prior` verranno trattate nei prossimi capitoli.

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 le seguenti:

- Il ricalcolo dei parametri è il comportamento della libreria Astropy; 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 procedimento sarebbe lenta e romperebbe la compatibilità con precedenti implementazioni.

- Il costo computazionale è sì maggiore, ma di una quantità piuttosto 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` nella seguente 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`.

Tale 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 sembrerebbe che non ci siano parametri da ottimizzare in quanto questi vengono già ottimizzati dall'algoritmo in fase di esecuzione. Sorge però un problema: è necessario controllare il numero di blocchi identificati.

Per semplificare il processo possiamo pensare che il numero di 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 del numero di blocchi identificati.

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

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

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.

Si presenta ora il punto cruciale, ovvero il calcolo della costante $ncp\_prior = \log \alpha$. É evidente la sua dipendenza dal numero di bin N, ma 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 è 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. Bisogna tenere conto tuttavia della dipendenza del parametro da N: sarà quindi necessario farlo per molteplici valori di N.

2. Se si facesse esattamente come da punto 1 bensì su dati reali, questo potrebbe produrre dell'overfitting, e perciò è preferibile utilizzare k-cross-validation[2].

3. Esiste un'ulteriore tecnica che non viene qui riportata, che si può consultare nella sezione 2.7 di Scargle (2013).

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

Nell'implementazione si è scelta una interpolazione tramite spline sui dati costruiti con *p_0* uguale a 0 per semplicità. Tali dati sono elencati nella seguente tabella.

[1] 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 ciò è triviale in quanto il numero di blocchi identificati è una variabile discreta, mentre *ncp_prior* è un parametro continuo.

[2] 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 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). Successivamente si calcola la medesima fitness sui dati di verifica e si usa quel valore per confrontare i risultati al variare dell'*ncp_prior*. Quello che massimizza la fitness sui dati di verifica è l'*ncp_prior* che viene scelto.

N	ncp_prior
140000	300
165450	700
190900	1050
216350	1300
241800	1550
267250	1800
292700	2050
318150	2300
343600	2400
369050	2600
394500	2950
419950	3000
445400	3200
470850	3300
496300	3700
521750	3800
547200	4000
572650	4150
598100	4200
623550	4400
649000	4550
674450	4700
699900	4800
725350	5000
750800	5100

N	ncp_prior
776250	5300
801700	5250
827150	5400
852600	5600
878050	5700
903500	5900
928950	5900
954400	6100
979850	6300
1005300	6300
1030750	6350
1056200	6350
1081650	6500
1107100	6500
1132550	6700
1158000	6600
1183450	6650
1208900	6650
1234350	6650
1259800	6700
1285250	6750
1310700	6750
1336150	6750
1361600	6750
1387050	6850

N	ncp_prior
1412500	6850
1437950	6850
1463400	6900
1488850	6900
1514300	6900
1539750	7000
1565200	7000
1590650	7000
1616100	7000
1641550	7000
1667000	7000
1692450	7000
1717900	7000
1743350	7000
1768800	7000
1794250	7000
1819700	7000
1845150	7000
1870600	7000
1896050	7000
1921500	7000
1946950	7000
1972400	7000
1997850	7000
2023300	7000

N	ncp_prior
2048750	7000
2074200	7000
2099650	7000
2125100	7000
2150550	7000
2176000	7000
2201450	7000
2226900	7000
2252350	7000
2277800	7000
2303250	7000
2328700	7000
2354150	7000
2379600	7000
2405050	7000
2430500	7000
2455950	7000
2481400	7000
2506850	7000
2532300	7000
2557750	7000
2583200	7000
2608650	7000
2634100	7000
2659550	7000

1.4 Osservazioni senza particolare ordine

Remark 3. *In fase di calcolo della log-likelihood la nostra funzione* (1.3) *non ha un buon comportamento computazionale: ad esempio, 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 ostacolare l'ottimizzazione. Analogamente se a risulta troppo vicino allo* 0 *potrebbe succedere che l'argomento del* log *può 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* (1.3) *con*
$$N_k \left(\log \left(\frac{N_k}{T_k} \right) + aS_k - 1 \right).$$
- *Se a dovesse risultare molto minore di* 0 *l'esponenziale dominerebbe il termine* 1*; in tal caso si può ignorare il termine 1 a denominatore, di conseguenza approssimiamo* (1.3) *con*
$$N_k \left(aT_k + \log(-aN_k) + aS_k - 1 \right).$$

Remark 4. *Quando si usa questo algoritmo è importante notare che le funzioni esponenziali ci permettono di avere molta più flessibilità. Tuttavia, messo a confronto anche semplicemente con Bayesian Blocks con funzione di fitting costante questo algoritmo non è strettamente migliore: ha infatti sia alcune potenzialità che alcuni problemi che nel sopracitato metodo non compaiono.*

- *É proprio della natura della funzione esponenziale la velocità nel crescere: questo permette di rappresentare alcuni tipi particolari di dati con grande fedeltà. Bisogna comunque fare attenzione al pericolo che ciò comporta: se il metodo fallisse, ad esempio per una scelta sbagliata del parametro ncp_prior, questo potrebbe avvenire in maniera esplosiva. Perciò è fondamentale scegliere correttamente il parametro in questione per avere dei risultati attendibili.*

- *Il calcolo dei parametri è molto più costoso a causa della non esistenza di una soluzione analitica del massimo della log-likelihood. Per cui se il costo computazionale dovesse essere una priorità il metodo trattato potrebbe non essere (almeno direttamente) il più adatto.*