



UNIVERSITÀ  
DEGLI STUDI  
FIRENZE

**Architetture  
Avanzate**

# Rankboost: implementazione e testing di un algoritmo learning-to-rank

Tommaso PAPINI  
tommaso.papini1@stud.unifi.it  
5537529

Gabriele BANI  
gabriele.bani@stud.unifi.it  
5719258

## IMPLEMENTAZIONE

---

Il progetto che abbiamo realizzato prevede l'implementazione dell'algoritmo **RankBoost** all'interno del framework QuickRank, utilizzando C++ come linguaggio di programmazione. In seguito sono stati effettuati anche dei test di comparazione con altri algoritmi presenti nei software Quickrank, jForests, RT-Rank e RankLib.

È stata, dunque, inizialmente implementata una versione sequenziale dell'algoritmo RankBoost all'interno di `custom_ltr.cc`. Prima di far partire l'algoritmo, tuttavia, è stato necessario creare un metodo `init()` per permetta di inizializzare tutte le varie strutture dati che poi verranno utilizzate in seguito. All'interno del metodo `init()` vengono anche controllati l'orientamento dei dataset che sono stati passati come parametri al metodo `learn()`. Se, infatti, l'orientamento dei dataset non è orizzontale, allora tali dataset vengono opportunamente trasformati in dataset con orientamento orizzontale (istanze  $\times$  features).

Il "**cuore**" principale dell'algoritmo RankBoost risiede nel metodo `learn()` ed, in particolare, all'interno del ciclo `for` che si occupa di scandire le varie iterazioni. Ad ogni iterazione, infatti, viene calcolato  $\alpha$  (peso del weak ranker) e vengono utilizzati i metodi `compute_pi` e `compute_weak_ranker` per calcolare, rispettivamente, la matrice potenziale ed il Weak Ranker migliore che poi verrà utilizzato per aggiornare la matrice distribuzione  $D$ .

Le metriche relative ai dataset di training e validation vengono calcolate durante le varie iterazioni ed il risultato viene stampato al termine del ciclo `for`. Non sapevamo, inizialmente, se fosse corretto fare ciò, ovvero calcolare le metriche relative al training ed al validation durante le varie iterazioni dell'algoritmo anziché in seguito dopo la terminazione delle varie iterazioni. Tuttavia, calcolando gli score relativi al training ed al validation iterazione per iterazione, riusciamo ad evitare di dover effettuare ulteriori cicli.

Per poter gestire più semplicemente i vari Weak Ranker è stata creata una sottoclasse **Weak Ranker** all'interno di `custom_ltr.h` e caratterizzata dalle variabili `theta`, `feature_id` e `sign` e dai metodi `clone()`, utile per creare un nuovo Weak Ranker con gli stessi valori del Weak Ranker corrente, `get_feature_id()`, `get_theta()` e `score_document()`.

Dopo aver verificato il corretto funzionamento di tale implementazione, abbiamo provveduto a parallelizzare alcune porzioni del codice utilizzando il tool **OpenMP**.

In alcuni casi, le versioni sequenziali di alcune porzioni del codice risultano migliori in termini di velocità di esecuzione rispetto alle rispettive versioni parallele e, pertanto, non sono state parallelizzate. Metodi come *compute\_weak\_ranker*, invece, hanno ottenuto un significativo incremento delle prestazioni.

Abbiamo provveduto anche ad analizzare le prestazioni dell'algoritmo utilizzando le varie tipologie di scheduling parallelo: *guided*, *dynamic* e *static*. Lo scheduling *dynamic* è risultato il migliore tra le varie tipologie di scheduling e, per questo motivo, abbiamo provveduto a modificare la variabile di sistema *OMP\_SCHEDULE* in modo che tutte le porzioni di codice parallelizzate utilizzino lo scheduling di tipo *dynamic*. Similmente, abbiamo provveduto ad utilizzare la variabile *n\_thread* (dichiarata in *custom\_ltr.h*) per poter gestire più semplicemente il numero di thread creati in ogni porzione di codice parallela.

## PROBLEMI INCONTRATI

I principali problemi riscontrati nell'implementare l'algoritmo RankBoost sono sorti al momento della parallelizzazione ed, in particolare, all'interno del metodo *compute\_weak\_ranker*. Parallelizzare il ciclo più esterno, ad esempio, portava a risultati differenti (e spesso erronei) rispetto alla versione sequenziale dell'algoritmo. Ciò non avveniva, tuttavia, se il ciclo parallelizzato era il ciclo più interno. In tal caso, infatti, sono stati ottenuti gli stessi risultati della versione sequenziale, pur osservando uno speedup notevole. Tale comportamento era inusuale ed, infatti, è stato risolto semplicemente effettuando una gestione più opportuna delle variabili condivise.

Una mal gestione delle **variabili condivise** aveva portato anche ad ottenere risultati non deterministici sia in termini di tempo di esecuzione dell'algoritmo, sia in termini di valori di NDCG. La principale causa di tale comportamento è stata individuata nella mancata dichiarazione della Feature *Feat* (variabile d'appoggio durante il calcolo del weak ranker) di tipo *firstprivate*.

Un'altra situazione controversa che abbiamo affrontato durante l'implementazione di tale algoritmo riguarda la gestione di **R** all'interno del metodo *compute\_weak\_ranker*. Seguendo le slides del corso, infatti, avevamo inizialmente implementato il metodo *compute\_weak\_ranker* in modo da avere valori di R crescenti durante le varie iterazioni effettuate dall'algoritmo. Tale discorso, infatti, era consistente con il concetto di bontà (o *goodness*) del weak ranker.

Confrontando i risultati da noi ottenuti con i risultati della libreria Java **RankLib** abbiamo potuto notare che, in RankLib, i valori di R non sono crescenti, bensì decrescenti e che questo non viene associato alla **bontà** del weak ranker, bensì all'**errore**. Tale comportamento deriva principalmente dal mancato controllo degli R negativi all'interno del metodo che provvede al calcolo del weak ranker.

Abbiamo, dunque, provato a commentare (e dunque eliminare) la porzione di codice che si occupava di effettuare il suddetto controllo (in *compute\_weak\_ranker*) ed abbiamo registrato, oltre ai valori decrescenti di R, anche valori di NDCG molto maggiori rispetto a prima. Abbiamo avuto delle difficoltà ad interpretare il corretto significato di R: se seguivamo le slides del corso ottenevamo valori di R crescenti ed NDCG più bassi, se seguivamo invece RankLib ottenevamo R decrescenti ed NDCG più alti. Riscontrando valori di NDCG più alti senza l'utilizzo di un controllo sugli R negativi, abbiamo preferito adottare tale approccio. Un altro dubbio che abbiamo avuto durante la fase di implementazione dell'algoritmo riguarda il significato del campo **offset**, in particolare all'interno del metodo *score\_document*. Ogni algoritmo, infatti, potrebbe aver bisogno di lavorare con un orientamento differente del dataset (orizzontale o verticale). Il campo offset indica solamente la posizione della prossima feature all'interno dello stesso documento e serve, appunto, ad ottenere un accesso più diretto ed efficace ai dati. Nel caso di orientamento orizzontale del dataset (documenti x features), la seconda feature è immediatamente dopo la prima e dunque il campo *offset* = 1. Nel caso di orientamento verticale del dataset (features x documenti), invece, il campo offset è uguale al numero di documenti del dataset poiché, per trovare la feature successiva, occorre prima scorrere tutti i documenti relativi alla feature precedente.

Allo scopo di velocizzare l'algoritmo sono state introdotte le strutture dati **SDF** (Sorted Document Features) e **last**.

In particolare, la matrice **SDF** contiene, per ogni feature e per ogni query, gli indici dei documenti ordinati in senso decrescente. Tale matrice viene inizializzata all'interno del metodo *init()* e viene utilizzata soprattutto all'interno del metodo *compute\_weak\_ranker* per poter calcolare in maniera più efficiente lo weak ranker migliore per ogni feature.

Il vettore **last**, invece, conterrà l'ultima posizione considerata nei documenti di ogni query ed è particolarmente utile all'interno del metodo *compute\_weak\_ranker* per poter "saltare" tutti i documenti già precedentemente considerati ed evitare, dunque, inutili iterazioni. È possibile fare ciò, naturalmente, grazie alla presenza ed all'utilizzo della matrice SDF che permette di avere i documenti ordinati in senso decrescente. TESTING

Vediamo in questo capitolo i test eseguiti sui vari software ed algoritmi suggeriti. Diciamo innanzitutto che i test sono stati eseguiti su un sottoinsieme dei dataset proposti, in particolare abbiamo preso le prime 50k righe da ogni dataset. Abbiamo fatto questo in quanto utilizzare i dataset originali (più di 2 milioni di righe ciascuno) avrebbe richiesto tempi d'esecuzione troppo lunghi.

Per i vari software e algoritmi testati abbiamo cercato di mantenere configurazioni più simili possibile. Abbiamo specificato il massimo numero di iterazioni per ogni algoritmo (100, 200 e 400) e, negli algoritmi che supportavano parallelizzazione (ovvero il nostro Rankboost e gli algoritmi RT-Rank) sono stati parallelizzati con 50 processori.

I primi test che abbiamo eseguito sono quelli relativi a 100 iterazioni massime (o massimo 100 alberi negli algoritmi tree-based).

Nella Tabella 1 possiamo vedere i tempi di esecuzione totali, in secondi, dei vari algoritmi sui vari dataset.

		<b>Fold1</b>	<b>Fold2</b>	<b>Fold3</b>	<b>Fold4</b>	<b>Fold5</b>	<b>Yahoo</b>
<b>Quickrank</b>	<b>Rankboost</b>	1893.44	1406.33	1377.14	1432.61	1454.83	8720.14
	<b>Mart</b>	27.58	27.57	27.34	26.66	28.32	72.41
	<b>LambdaMart</b>	28.34	28.55	27.99	29.16	28.69	74.45
	<b>OBVMart</b>	22.44	22.49	22.84	22.11	21.94	74.90
	<b>OBVLambdaMart</b>	24.42	23.68	23.39	23.26	25.24	75.51
<b>RankLib</b>	<b>Rankboost</b>	8976.79	9557.26	9165.96	9128.61	9508.41	167704.94
<b>RT-Rank</b>	<b>Random Forests</b>	705.14	5.36	4.75	4.89	4.95	18.91
	<b>IGBRT</b>	952.59	28.76	27.64	28.31	28.19	51.52
<b>jForests</b>	<b>LambdaMart</b>	44.32	42.83	43.53	43.35	44.77	146.68

Tabella 1: Tempi di esecuzione (in secondi) dopo 100 iterazioni massime.

Notiamo innanzitutto che gli algoritmi basati su alberi (ovvero tutti tranne Rankboost) sono estremamente più veloci. Per questo motivo abbiamo deciso di rappresentare in Figura 1 soltanto i tempi di esecuzione più veloci, al fine di avere una lettura più chiara.

In Figura 2 vediamo invece i tempi di esecuzione in percentuale sui vari dataset per ogni algoritmo. Da questo grafico vediamo subito come il dataset Yahoo sia quello più "impegnativo" per i vari algoritmi, ad

eccezione degli algoritmi RT-Rank che impiegano la maggior parte del tempo sul Fold1.

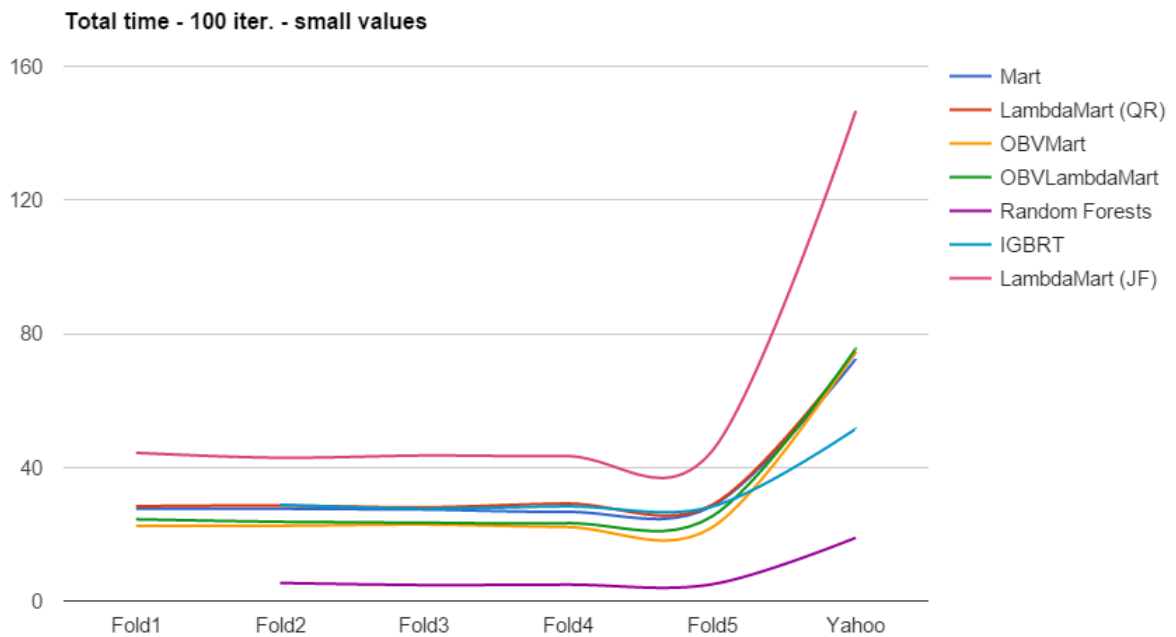


Figura 1: Tempi di esecuzione bassi (in secondi) dopo 100 iterazioni massime.

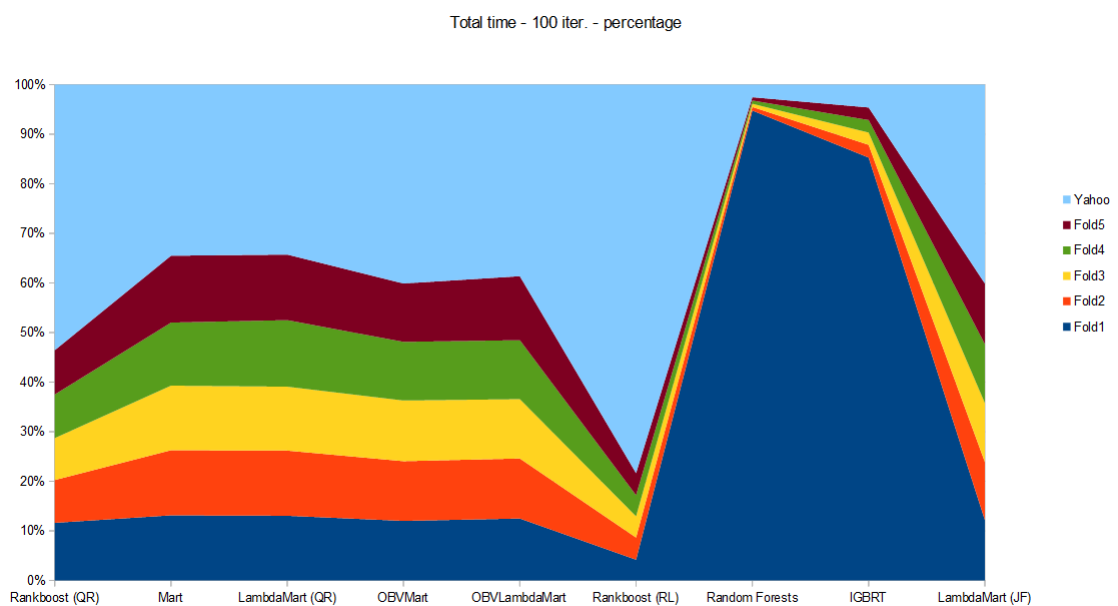


Figura 2: Tempi di esecuzione in percentuale dopo 100 iterazioni massime.

Vediamo adesso i grafici dei valori NDCG sui set di training, validation e test per ogni algoritmo ed ogni dataset. Per chiarezza di lettura non abbiamo incluso le tabelle relative ai valori NDCG ottenuti con questi test. Per consultarli visitare il seguente indirizzo: <https://docs.google.com/spreadsheets/d/1vXvkFW0HsGrn0sFNtyqgyLr0Lo-dTIV9bmV4sTKwVL4/edit?usp=sharing>.

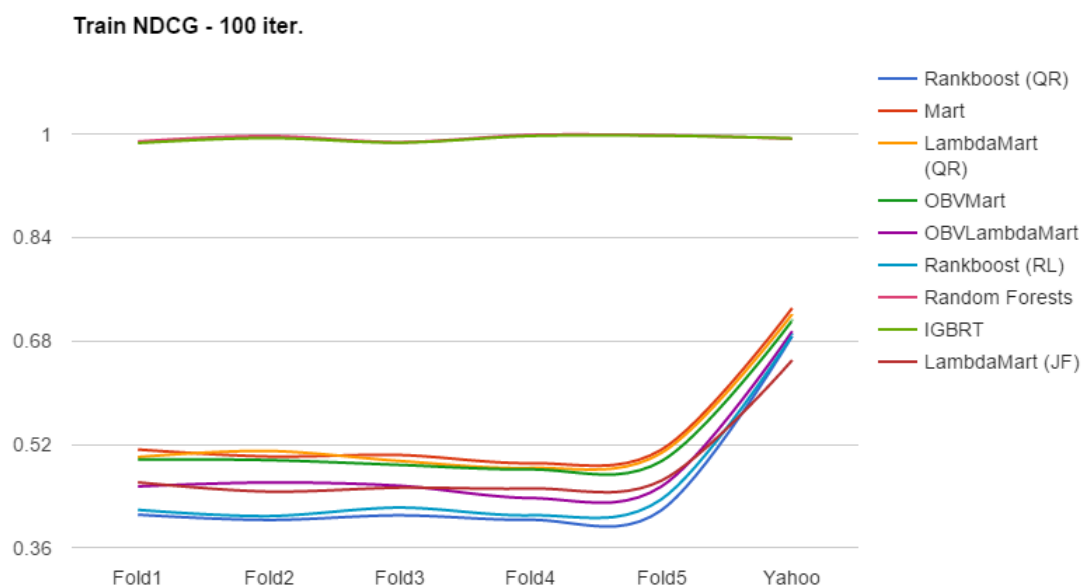


Figura 3: Metriche NDCG del Training Set dopo 100 iterazioni massime.

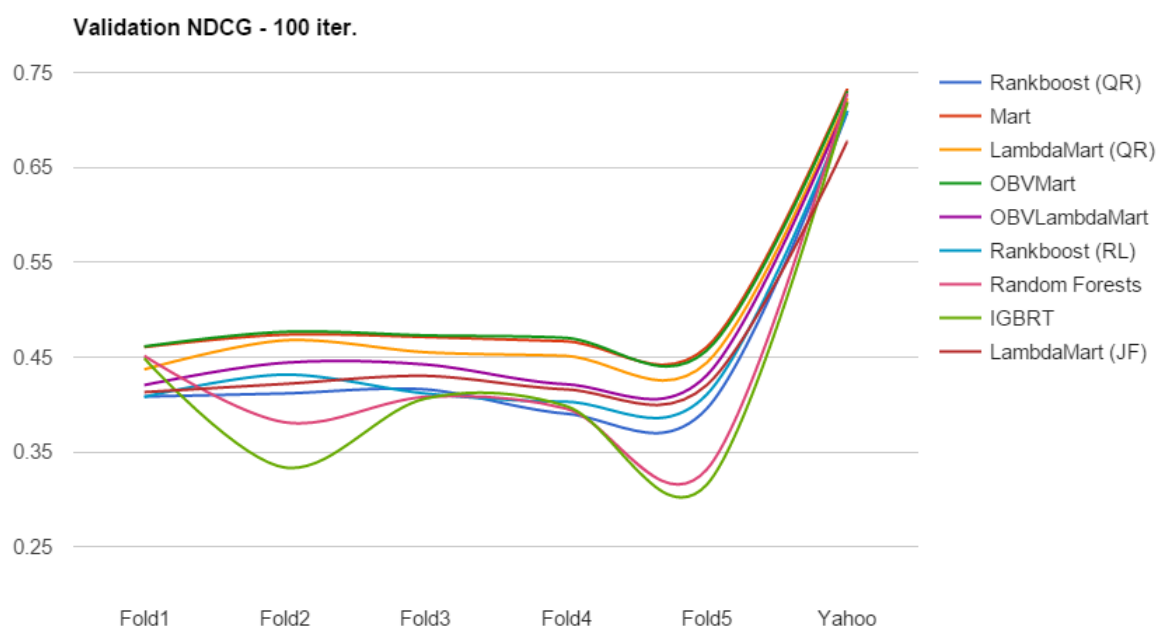


Figura 4: Metriche NDCG del Validation Set dopo 100 iterazioni massime.

Confrontando i tre grafici notiamo come gli algoritmi di RT-Rank, nonostante raggiungano valori altissimi sulla predizione del Training Set (sono le due linee in alto in Figura 3, sovrapposte), non riescono tuttavia a raggiungere gli stessi risultati nei set di Validation e Test. Al contrario, hanno dei valori di NDCG anche più bassi rispetto agli altri algoritmi, quindi possiamo affermare che RT-Rank è il software che inferisce il peggio possibile su dati sconosciuti, imparando “troppo” del Training Set. Inoltre si può osservare come, in generale, la fase di boosting dell’algoritmo IGBRT peggiori i risultati già ottenuti con Random Forests, aumentando però i tempi di esecuzione.

Abbiamo ripetuto i test anche con 200 e 400 iterazioni massime (per i soli algoritmi basati su alberi)

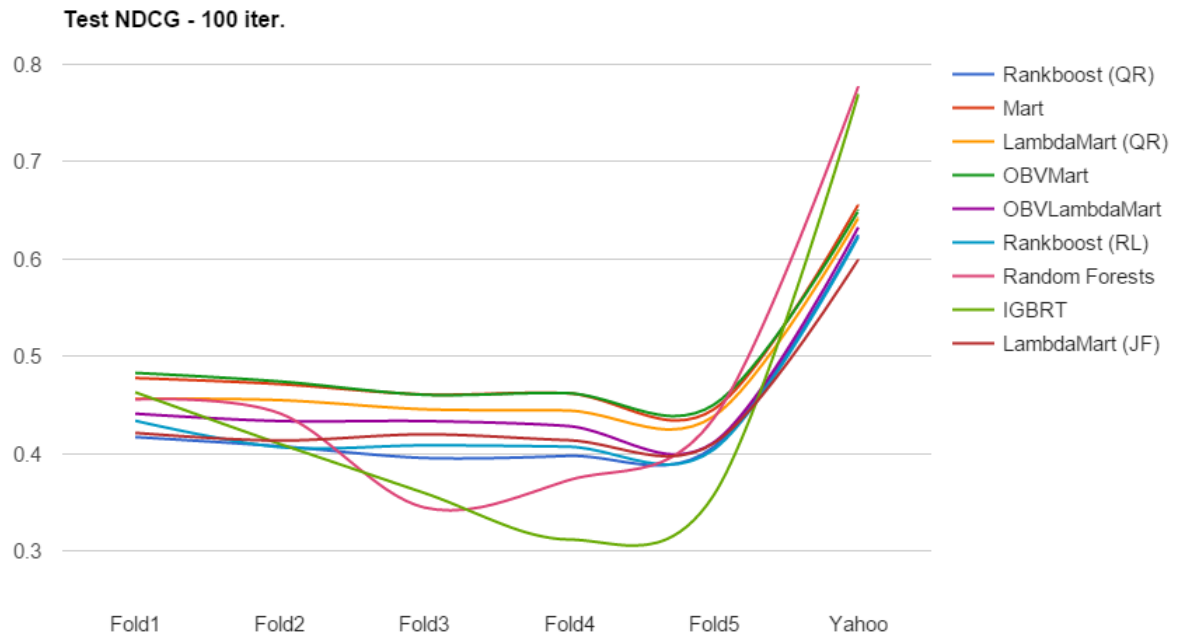


Figura 5: Metriche NDCG del Test Set dopo 100 iterazioni massime.

ed abbiamo ottenuto risultati molto simili a questi primi test (a parte tempi d'esecuzione leggermente più lunghi). Inoltre alcuni algoritmi utilizzavano meno di 400 alberi, quindi non siamo andati oltre con i test. Non riportiamo nessuna tabella o grafico (essendo molto simili ai precedenti). Per una consultazione in dettaglio dei tempi di esecuzione e dei valori NDCG ottenuti si vada al seguente indirizzo: <https://docs.google.com/spreadsheets/d/1vXvFW0HsGrn0sFNtyqgyLr0Lo-dTIV9bmV4sTKwVL4/edit?usp=sharing>.