UNIVERSITÀ DEGLI STUDI FIRENZE

Architetture Avanzate

Rankboost: implementazione e testing di un algorimo learning-to-rank

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

Gabriele Bani
gabriele.bani@stud.unifi.it
???????

IMPLEMENTAZIONE

Il progetto che abbiamo realizzato prevede l'implementazione dell'algoritmo **RankBoost** all'interno del framework quickrank, utilizzando C++ come linguaggio di programmazione.
All'interno del metodo **learn**() della classe custom_ltr, è stata inizialmente implementata una versione sequenziale dell'algoritmo. In particolare, il metodo init() che viene richiamato all'interno del metodo learn(), prima di effettuare il ciclo for che scandisce le varie iterazioni dell'algoritmo, viene utilizzato per instanziare tutte le varie strutture dati che verranno utilizzate dall'algoritmo. 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 vengono opportunamente trasformati in dataset con orientamento orizzontale (istanze x features).
Il **cuore** principale dell'algoritmo risiede all'interno del ciclo for che si occupa di scandire le varie iterazioni dell'algoritmo. Ad ogni iterazione, infatti, vengono utilizzati i metodi compute_pi e compute_weak_ranker per calcolare, rispettivamente, la matrice potenziale ed il Weak Ranker che poi verrà utilizzato per aggiornare la matrice distribuzione D. Le metriche relative al dataset di training e validation vengono calcolate durante le varie iterazioni ed il risultato viene stampato al termine del ciclo for.
Per poter gestire più semplicemente i vari Weak Ranker è stata creata una subclasse **Weak Ranker** all'interno della classe custom_ltr.h, 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, infatti, le versioni sequenziali di alcune porzioni del codice risultano migliori in termini di velocità di esecuzione rispetto alle rispettive versioni parallele e, pertanto, non è stato necessario parallelizzarle.
Abbiamo provveduto anche ad analizzare le prestazioni dell'algoritmo utilizzando le varie tipologie di scheduling parallelo: runtime, guided, dynamic e static. Lo scheduling guided risulta 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 guided. 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, all'interno del framework quickrank, sono sorti 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 problematica è stata risolta 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 score. La principale causa di tale comportamento è stata individuata nella mancata dichiarazione della Feature Feat 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** (come da Lei suggerito) 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 viene chiamato **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 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 descrescenti ed NDCG più alti.

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.

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

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/1vXvkFWOHsGrn0sFNtyqgyLr0Lo-dTIV9bmV4sTKwVL4/edit?usp=sharing`.

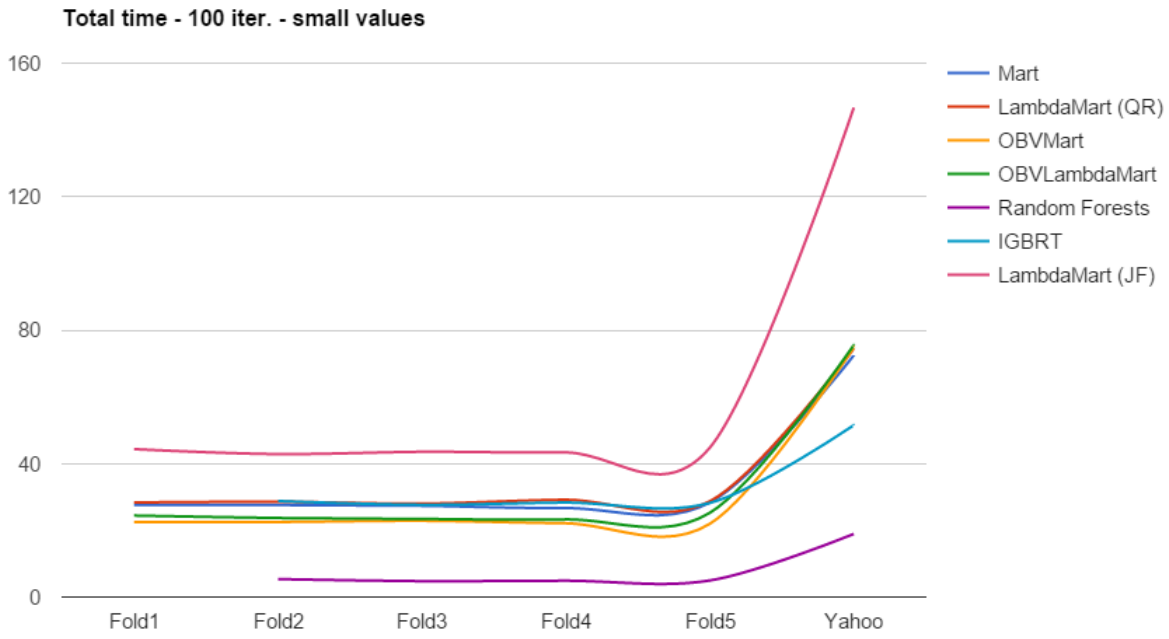

Figura 1: Tempi di esecuzione bassi (in secondi) dopo 100 iterazioni massime.

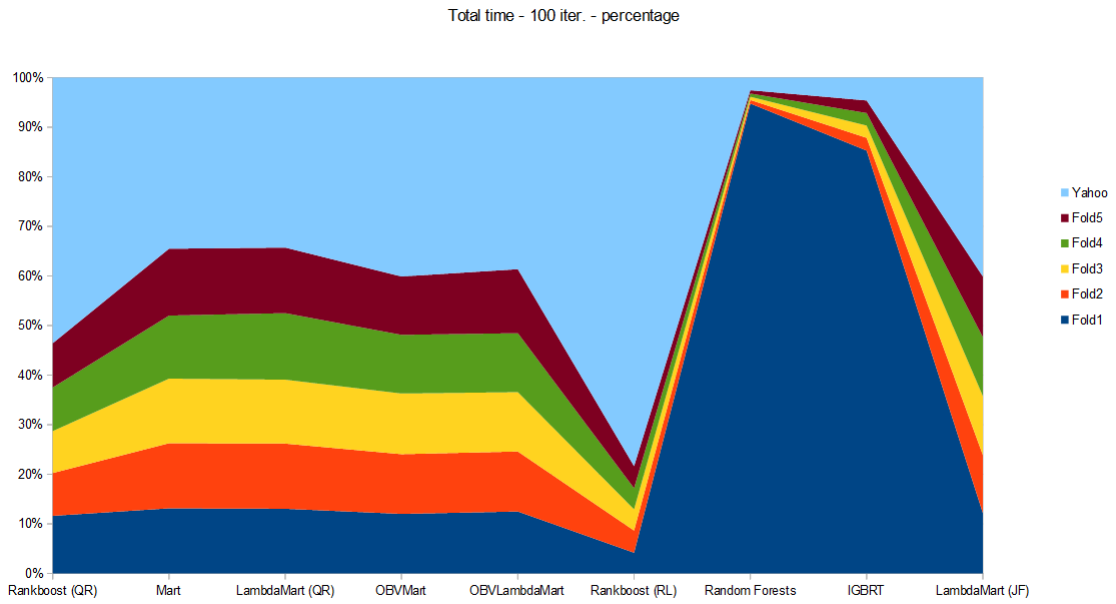

Figura 2: Tempi di esecuzione in percentuale dopo 100 iterazioni massime.

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

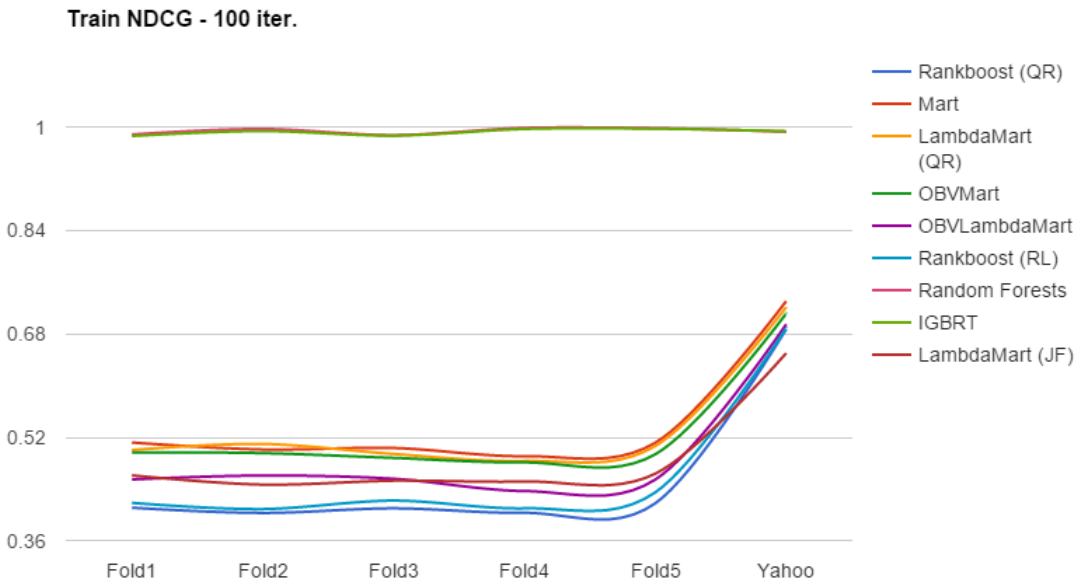

Figura 3: Metriche NDCG del Training Set dopo 100 iterazioni massime.

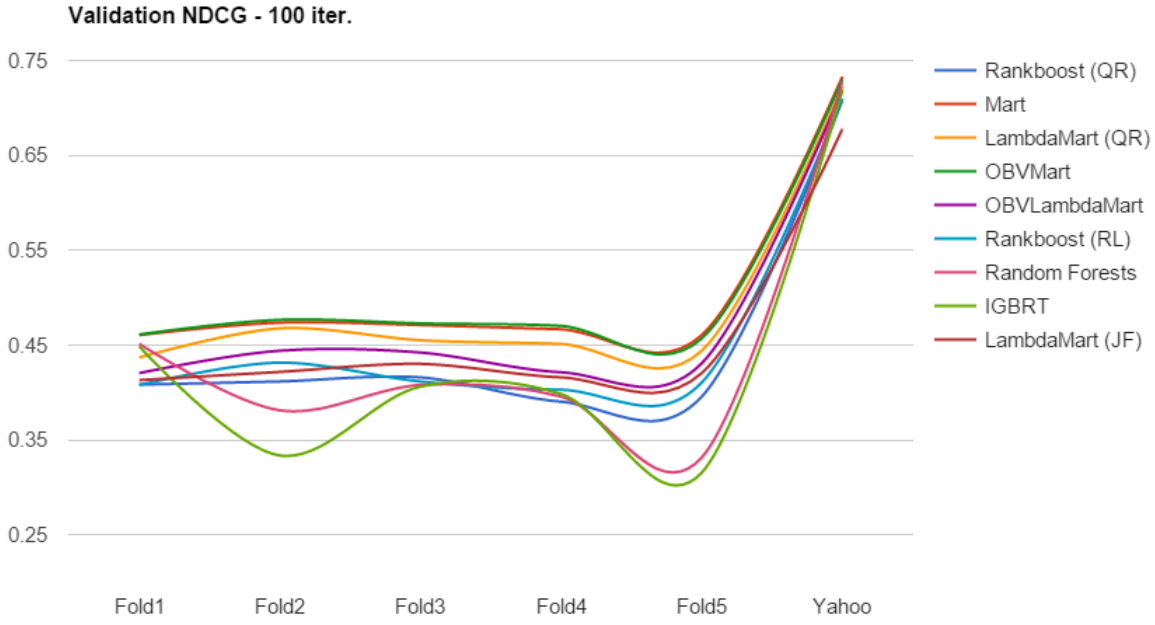

Figura 4: Metriche NDCG del Validation Set dopo 100 iterazioni massime.

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

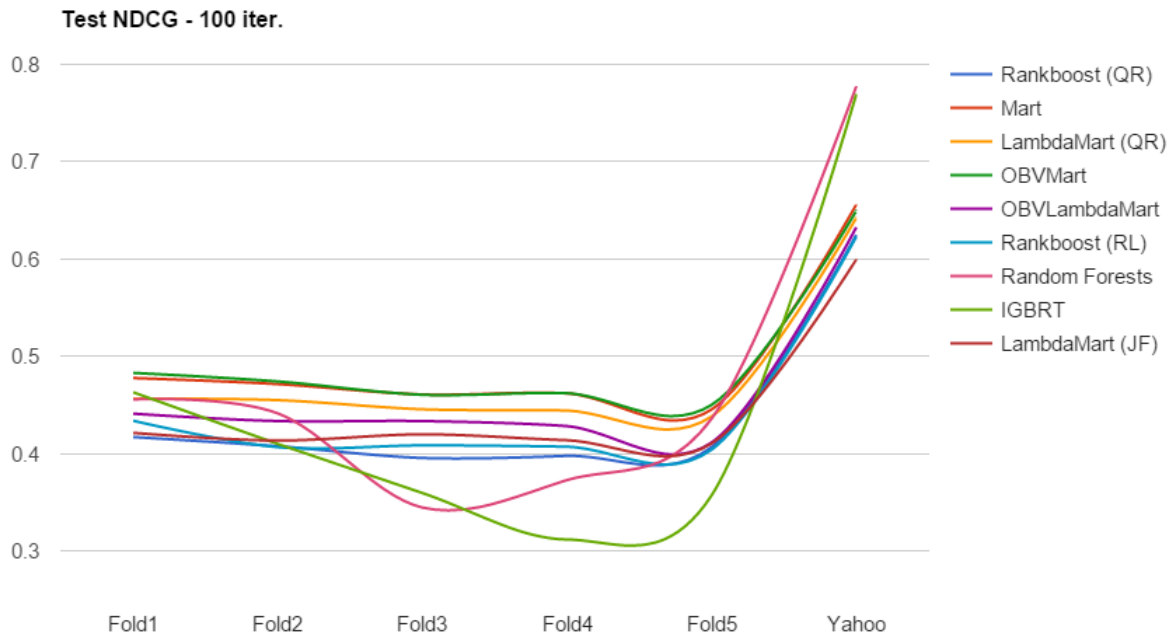

Figura 5: Metriche NDCG del Test Set dopo 100 iterazioni massime.

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) 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/1vXvkFWOHsGrn0sFNtyqgyLr0Lo-dTIV9bmV4sTKwVL4/edit?usp=sharing`.