UNIVERSITÀ DEGLI STUDI FIRENZE

Architetture Avanzate

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

Tommaso Papini
tommaso.papini1@stud.unifi.it

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. In seguito sono stati effettuati anche dei test di comparazione con altri algoritmi presenti nei software Quickrank, JForests e Rt-Rank.
E' stata, dunque, inizialmente implementata una versione sequenziale dell'algoritmo RankBoost all'interno della classe custom_ltr. 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 x 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 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 matriche 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 subclasse **Weak Ranker** all'interno della classe 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 significato incremento delle prestazioni.
Abbiamo provveduto anche ad analizzare le prestazioni dell'algoritmo utilizzando le varie tipologie di scheduling parallelo: runtime, guided, dynamic e static. Lo scheduling dynamic 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 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 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" (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 descrescenti 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.

SPEEDUP E MIGLIORAMENTI

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 features 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. E' possibile fare ciò, naturalmente, grazie alla presenza ed all'utilizzo della matrice SDF che permette di avere i documenti ordinati in senso decrescente.

TESTING