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.
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.

TESTING