Hybrid Collaborative Filtering Recommendation Engine using Spark and ElasticSearch

Shivraj Gill

April 8, 2018

Introduction

A recommendation engine attempts to intelligently recommend products to users based on what they would like the most. While there are different implementations of recommendation engines, all of them are evaluated by their predictive accuracy, scalability, and ability to handle sparseness. In this report, I will examine the performance of a Hybrid Distributed Collaborative Filtering model and compare it to a commonly used collaborative recommendation system. To underscore the motivations behind the hybrid collaborative filtering model, I will give a brief overview of traditional recommendation engines and how they perform.

Traditional Recommendation Algorithms

(1) Content-Based

- **Idea**: Uses contextual data of an item (i.e movie genre) that a user has liked and recommends similar items
- **Algorithms**: Decision Trees, Neural Networks, and vector-based representations
- **Evaluation**:
    - **Predictive Accuracy**: Not always accurate because it is limited by the features that are explicitly associated with the items. Also, its accuracy depends on the amount of data available about the user and item. (Cold start Problem)
    - **Scalability**: The time it takes to train the algorithm varies, but it usually produces fast predictions.

(2) Collaborative Filtering

- **Idea**: Based entirely on historical data about user-item interactions, such as ratings and clicks. This makes it one of the most commonly used algorithms as it is not dependent on any additional information like metadata about the item
- **Algorithms**:
  1 User-User Collaborative filtering
    - **Approach**: Finds similar users and recommends common items that those users had liked in the past. It is a memory-based model because it needs to store a user similarity matrix.
    - **Evaluation**:
      - **Predictive Accuracy**: Generally performs better than content-based algorithms but has inaccurate predictions for new users/items that have little historical data. (Cold Start Problem)
      - **Scalability**: Does not scale well because it has to first compute every user pair similarity score and then compute the user-item score for the most similar users.
      - **Spareness**: Performs poorly when the user-item matrix is sparse

  2 Item-Item Collaborative filtering
    - **Approach**: Recommends similar items to items that a user had liked. It is a memory-based model because it needs to store an item similarity matrix.
    - **Evaluation**:
      - **Predictive Accuracy**: Generally performs better than content based algorithms but has inaccurate predictions for new users/items which have little historical data. (Cold Start Problem)
      - **Scalability**: Scales very well because the item similarity matrix can be computed offline and usually does not change much once there is enough data to calculate reliable similarity scores
      - **Spareness**: Performs poorly when the user-item matrix is sparse

  3 Low-Rank Matrix Factorization

    - **Approach**: Finds low rank matrix representations of users and items and uses them to fill in recommendation scores for unseen user-item pairs
    - **Evaluation**:
      - **Predictive Accuracy**: Generally performs better than both user-user and item-item based collaborative filtering.
      - **Scalability**: Training takes awhile when the user-item matrix is large. Also, there is usually not enough memory to store the prediction scores for the user-item matrix. Thus, predictions are pretty slow since they must be calculated real-time.
      - **Sparseness**: Handles sparseness very well because it fills in missing values by performing calculations of factorized user and item vectors

(3) Hybrid Models

  - **Idea**: Combines techniques from content-based, collaborative filtering, and other machine learning methods to provide more relevant recommendations to the user
  - **Examples**: Factorization Machines, Weighted methods, Switching, Feature Augmentation, Meta-Level
  - **Evaluation**:
    - **Predictive Accuracy**: Tends to show more accurate results depending on the application. It alleviates some of the problems associated with collaborative filtering and content-based recommender such as the Cold-Start Problem ,
    - **Scalability**: Tends to be slower because it is incorporating two or more techniques
    - **Sparseness**: Tends to handle sparseness better, but it depends on the application. For example, factorization machines perform poorly with highly sparse data.

Hybrid Collaborative Filtering Recommendation Engine

Motivations

By combining the techniques from different hybrid methods, I plan to show how using a distributed search platform like ElasticSearch on top of Apache

Spark can be used to implement a scalable hybrid collaborative filtering recommendation engine that can be personalized according to context filtering. To provide context to the results, I will compare the hybrid implementation to the traditional matrix factorization based collaborative filtering model.

Overview

(1) Apply Spark's Matrix Factorization based Collaborative Filtering package on User-Item Matrix to obtain a condense low-rank vector representation of users and items

(2) Apply preprocessing techniques such as normalization and removing outliers to low-rank user vectors

(3) Apply Bisecting K-Means clustering to post-processed user vectors to obtain clusters of similar users

(4) Build a cluster-item matrix by taking top ratings from k closest points to cluster centroid

(5) Save the matrix to ElasticSearch and show recommendations using Elasticsearch queries

Implementation and Results

All the experiments were performed on Mac OSX 10.11 operating system running on 2.90GHz processors with 4 processing cores. The master node of a cluster was allocated 4GB RAM while each slave node was allocated 2GB RAM. I used the Pyspark Shell with the following settings Apache Hadoop-2.7.3, Spark-2.2.1, Scala-2.11.8, and elasticsearch-hadoop-5.3.0.

To see the code for the classic and hybrid recommendation engines, please refer to the notebooks folder in this repository. We compare our model performances on the MovieLens dataset with 100k ratings. Before we run matrix factorization on our user-movie ratings, we preprocess the data by removing users who have rated less than 5 times and movies with an average rating less than 2.5. The results of the preprocessing on the 100k sized dataset in Table 1.

(1) The data is split into 70% training, 10% validation, and 10% test. We train and tune the matrix factorization model using the ALS recommendation module in Spark. Note that we replicate the approach used

in . To find the approximate optimal values, we train the model over the following settings:

rank: {5, 10, 25, 35, 45, 50, 70, 100}
λ: {0.01, 0.1, 1, 10}

We found that the model performs best when rank = 50 and $\lambda = 0.1$ with an RMSE value of . Figure 1 shows the resulting RMSE values for the different ranks when $\lambda = 0.1$.

(2) To optimize clustering performance, it is important to normalize our features and remove outliers. For our model. we normalize each latent feature of the user vectors returned from our matrix factorization model. These vectors will be normalized to have 0 mean and unit variance. Next, we remove any outliers whose features lie outside [-3.15, 3.15].

(3) We use the K-Means model package in Spark to cluster the post-processed user feature vector. To choose the best number of clusters, we use the elbow method. We found that the WSSE is minimized when the cluster size is 20.

(4) To build the cluster-movie matrix we first find the k closest users for each cluster that are closest to the cluster centroid. Next, we will use the results from our matrix factorization model to obtain each of these k users' ratings for every movie. Lastly, we will take the top rating for every movie from these k users and remove duplicate ratings.

(5) Save the matrix to ElasticSearch and show recommendations using Elasticsearch queries. In the notebook, we show examples of post-contextual filtering through ElasticSearch queries and how it can achieve more personalized results.

Conclusions

We saw that our hybrid engine generates recommends twice as fast as our classic matrix factorization model. The ElasticSearch function query for the classic matrix factorization model computes a similarity score by multiplying the user matrix factorization vector with each movie's factorization vector. On the other hand, the ElasticSearch function query for the hybrid model just needs to query for a cluster's movie ratings and filter out movies that have already been watched for that user. This is why the hybrid model achieves very impressive speed results.

Nonetheless there are major weaknesses in the hybrid engine such as accuracy. Because a cluster represents each user's preferences for movies, we end up recommending the same movies to users in the same cluster. We can improve upon this by using a neighborhood approach to our clustering results to create more personalized results for each user. We could also average out the ratings for the k closest users to each centroid instead of taking the max. Further, more data could improve our results because K-Means is known to get stuck at local minimums. This could lead to higher quality clusters and thus better recommendations. Lastly, we can improve our model by taking advantage of ElasticSearch's query functionality and creating customized function queries that weight certain movie metadata such as genres.

Influences

For anyone that wants to gain a stronger understanding of the techniques and technologies used, please refer to Hybrid Collaborative Filtering Engine in Spark and Building a Recommendation Engine with ElasticSearch and Spark. The former applies K-Means clustering to a collaborative filtering model exactly like our hybrid recommendation engine. The latter shows how to use ElasticSearch with Spark to build a scalable recommendation system. We combine these two techniques to build our hybrid engine and compare the results to a traditional matrix factorization based collaborative filtering model.