Speeding Up k-NN Subreddit Recommendation

Dimitrios Tzannetos
Carnegie Mellon University
dimitrios.tzannetos@sv.cmu.edu

Darren Yin
Carnegie Mellon University
darren.yin@sv.cmu.edu

Ali Yousafzai
Carnegie Mellon University
ali.yousafzai@sv.cmu.edu

***Abstract*—Reddit has become a very popular social news website, but there is no good way to discover "subreddits" - online communities based on specific discussion topics. Although a lot of approaches have been implemented in the area of Recommender Systems, most of them perform poorly when dealing with large datasets. This leads us to other alternatives that can generate recommendations in a parallel way. This paper tackles the subreddit discovery problem by adopting a collaborative filtering approach and examines two different Map Reduce implementations of k-Nearest Neighbors algorithm to recommend subreddits based on users' past subreddit subscriptions and activity. Leveraging a more scalable approach, we were able to achieve approximately 30x speedup compared to the sequential implementation.**

***Index Terms*—Recommender Systems, Distributed Computing, k-Nearest Neighbors, Reddit**

1. Introduction

With the proliferation of social and information networks, the cost of content creation and distribution has become negligible, giving rise to a vast amount of human-generated content available from multiple sources across the Internet. In the Information Age, users nowadays require tools to assist them in finding and filtering the available information into manageable quantities of interesting material. This need for content filtering led to the emergence of Recommender Systems. Such tools rely on the historical patterns of users to predict the preference of a user for new content and focus on suggesting content most likely to be of interest to that individual user. In this work, we examine the use of a specific technique widely used in the area of Recommender Systems, named collaborative filtering and apply it in the context of Reddit.

Reddit, often called "The front page of the Internet" is a popular link aggregator and social media website where users share and discuss their topics of interest. These entries are divided into "subreddits" based on areas of interest (e.g. programming, gaming, sport or politics). Each user can choose which sub-community they subscribe to, based on their interests, though upon registration each user is automatically subscribed to a default set of 20 subreddits.

As of 2017, Reddit has 234 million unique users from 217 different countries and generates 8.19 billion views a month, making it the 4th most visited sited in the United States and 7th in the world. Across 2015, Reddit saw 82.54 billion pageviews, 73.15 million submissions, 725.85 million comments. With over 67,000 active subreddits monthly, it is difficult for both new and old users to find subreddits that would be of interest to them and to know which subreddit it would be best to submit posts to.

The aforementioned features make the task of subreddit discovery a challenging problem, calling for a technique in parallel computing environments that shows scalability and manages to exploit concurrency.

This paper uses an unsupervised machine learning model to examine extensive experiment results in a large corpus of comments derived from Reddit. The k-Nearest Neighbors technique employed, yields promising results concerning time efficiency, with respect to Big Data workload processing time limitations.

The rest of the document is organized as follows: Section 2 presents recent related work and studies on Recommender Systems using variations of k-Nearest Neighbors. The problem description and the addressed challenges are explained in Section 3. Our approach is presented in Section 4. Subsequently, the evaluation results are discussed in Section 5.

We conclude with a view on future work in Section 6.

2. Related Work

Finding nearest neighbors is a critical component in Recommender Systems. In this section, we discuss recent trends in scaling up nearest neighbor identification to Big Data. Filtering and Approximate Nearest Neighbor (ANN) based approaches rely on data structures to reduce the number of computed similarities, yet provide error guarantees on identified neighborhoods. Other approaches simply choose a subset of users or items to search for neighbors in, without guaranteeing high quality results. Additional performance gains are achieved via multithreaded and distributed systems. For the rest of this Section, we will refer to the user or item rating profile as an object and the values it contains as features.

Identifying neighbors has been studied extensively for several decades in the form of two related problems. The k-nearest neighbor (k-NN) search problem seeks to find the k objects that have the highest proximity to a query object. The ϵ-nearest neighbor (ϵNN) search (or similarity search) problem finds all objects with a similarity of at least ϵ to the query. When the proximity function is a metric, the problem is also known as radius search, and seeks to find all objects within distance to the query. Recommender systems initially compute and store the k-NN or ϵNN for each user or item, which are known as k-NN graph construction (k-NNG) and all-pairs similarity search (APSS) or similarity join problems, respectively. In recent years, a number of nearest neighbor methods have been proposed for sparse vectors that work by ignoring (or filtering) object pairs that cannot be neighbors. The vector representing a user or item rating profile is inherently sparse, as a user often consumes and rates few of the overall items.

Search methods avoid comparing objects that have no features in common by using an inverted index data structure. The index consists of a set of lists, one for each feature among all objects, such that the jth list contains pairs $(i, r_{i,j})$ for all objects i that have a non-zero $r_{i,j}$ value for the jth feature. Additional savings can be achieved by relying on proximity thresholds. Chaudhuri et al. [**?**] were first to show that, given a predefined feature processing order, there comes a point in the feature processing when the un-processed query features do not have enough weight to lead to a proximity of at least ϵ with any object in the index. One can thus identify all potential neighbors for an object by checking only the inverted lists associated with the first few query features.

Bayardo et al. [**?**] used the idea in [**?**], along with a predefined object processing order, to design a filtering framework for solving the ANN problem. For each object, the framework first identifies a list of candidates by consulting the inverted index lists for the first few features in the query object. Each candidate is then vetted by computing a quick upper bound on its similarity with the query and comparing it to the threshold ϵ. Finally, those candidates that pass this filtering test have their similarities with the query computed and checked against ϵ. Note that the framework has a null error guarantee. It outputs the same result as when computing all pairwise similarities and then filtering those below ϵ.

The framework proposed by Bayardo et al. [**?**] has been extended for cosine similarity by a number of different methods [**?**], [**?**], [**?**], [**?**]. In their method L2AP, Anastasiu and Karypis [**?**] introduced more stringent bounds and new filtering strategies within the filtering framework, which lead to an order of magnitude efficiency improvement over previous state-of-the-art methods. The key to the success of L2AP, and its namesake (L2-norm All-Pairs), is the use of the L2-norm of the remaining (unprocessed) features in the candidate and query vectors to compute several powerful similarity upper bounds which lead to significant efficiency savings in the filtering framework.

In the k-NNG construction context, most proposed methods are approximate, relying on data [**?**] or neighborhood improvement [**?**] heuristics to find some of the nearest neighbors. In contrast, Anastasiu and Karypis [**?**] developed L2k-NNg, an exact k-NNG construction filtering-based method. The main idea in L2k-NNg is to bootstrap the search with a quickly constructed approximate graph and then use the minimum similarities in neighborhoods as filtering criteria.

Finding nearest neighbors is inherently a memory

bound operation. As a result, shared memory parallel methods [**?**] focus on keeping threads busy and minimizing resource contention. Existing distributed solutions for nearest neighbor graph construction generally use the MapReduce framework. Most of these methods rely on the frameworks built-in features to aggregate (reduce) partial similarities of object pairs computed in mappers [**?**], [**?**], [**?**], [**?**]. While some filtering strategies can be used to avoid generating some partial similarities, these methods often suffer from high communication costs which make them inefficient for large datasets [**?**]. Another category of MapReduce methods use a mapper-only scheme, with no reducers [**?**], [**?**],[**?**]. They partition the set of objects into subsets (blocks) and apply serial nearest-neighbor search methods on block pairs. Certain block comparisons can be eliminated by relying on block-level filtering techniques. Alabduljalil et al. also investigated distributed load balancing strategies [**?**] and cache-conscious performance optimizations for the local searches [**?**].

3. Problem Formulation

3.1. Collaborative Filtering

As one of the most successful approaches to building Recommender Systems, collaborative filtering (CF) uses the known preferences of a group of users to make recommendations or predictions of the unknown preferences for other users.

Collaborative filtering can be applied to various problems depending on the goal of the recommender system. In the context of Reddit, there are two types of recommenders:

(i) **Rating Prediction:** the goal is to compute the rating that a user would give for a subreddit.

(ii) **Top-N recommendation:** the goal is to provide the user with a list of N items that they will find interesting and will likely subscribe to.

In the context of the current work, we study the analysis of collaborative filtering for top-N recommendation, as we are focusing on Reddit as a community, aiming to provide each user with a most N recommended subreddits.

The outcome of the aforementioned analysis is delivered using a neighborhood-based approach generate recommendations. Thus, two similarity approaches can be defined:

(i) **User-Based:** firstly identify the k most similar users (nearest neighbors) to the active user using the Pearson correlation or vector-space model [9, 27], in which each user is treated as a vector in the m-dimensional item space and the similarities between the active user and other users are computed between the vectors. After the k most similar users have been discovered, their corresponding rows in the user-item matrix R are aggregated to identify a set of subreddits , C subscribed by the group together with their frequency. With the set C, user-based CF techniques then recommend the top-N most frequent subreddits in that the active user has not subscribed to.

(ii) **Subreddit-Based:** firstly compute the k most similar subreddits for each subreddits according to the similarities; then identify the set, C, as candidates of recommended subreddits by taking the union of the k most similar subreddits and removing each of the subreddits in the set, U, that the user has already subscribed to; then calculate the similarities between each subreddit of the set C and the set U. The resulting set of the subreddits in C, sorted in decreasing order of the similarity, will be the recommended subreddit-based Top-N list.

3.2. k-Nearest Neighbors

In the current paper, we focus on the k-NN distributed implementation using an user-based approach to generated recommendations. k-NN is a non-parametric lazy learning algorithm. Being a non-parametric algorithm it does not make any assumptions on the underlying data distribution. This is a major advantage because majority of the practical data does not obey theoretical assumptions made and this is where non-parametric algorithms like k-NN come to the rescue. k-NN is also a lazy algorithm this implies that it does not use the training data points to do any generalization. So, the training phase is pretty fast. Lack of generalization means that k-NN keeps all the training data. k-NN makes decision based on the entire training data set.

Unlike other cases where users' ratings are available, in the case of Reddit, such information is not explicitly provided. To estimate the relationship between a user and a particular subreddit, we define

the following affinity score a metric between a user u_i and a subreddit s_j as :

$$a(u_i, s_j) = \frac{\mu_j(i)}{\sum_{i=1}^{n} \mu_j(k)}$$

where $\mu_j(i)$ denotes the number of comments user u_i has posted to subreddit s_j and N refers to the total number of users.

3.3. MapReduce

MapReduce is a programming model for processing large amounts of data. It is essentially broken into a Map and Reduce step. The Map step performs some transformation over each element of the dataset input. The Reduce step performs an aggregation over the output from the map step.

Hadoop is a commonly used framework that supports the storing of and processing of extremely large data utilizing the MapReduce programming model in a distributed computing environment. The framework coordinates the map and reduce phases. Since the framework utilizes a distributed environment, multiple mappers and reducers can be run in parallel.

One of the biggest problems with distributed computing, is the almost guarantee that some component of the distributed system will fail. Hadoop implements two important fail-safes to provide reliability and scalability to distributed MapReduce computing. Fail-safe storage is provided with HDFS, Hadoop Distributed File System, where three separate copies of each data block are stored, by default. The architecture also provides fail-safe task management, which will reschedule failed tasks on different nodes up to four times, by default, before failing completely.

4. ALGORITHMS

4.1. Sequential

A sequential algorithm of k-Nearest Neighbors is used as a proof of concept to show the differences in performance when using Map-Reduce and compare when to use one approach over the other.

A baseline sequential version of the Subreddit Recommender was used as a starting point and modified to run with our dataset. An implementation written in Python was found on GitHub [1]. The algorithm recommends subreddits for a user based on what subreddits the neighbors of the user are subscribed to. Algorithm **??** provides the pseudocode of the Sequential version which is quite simple to follow. The input to the algorithm can be a file containing a list of users with their subreddits.

Algorithm 1 Sequential Implementation

```
1: Parse Input file
2: Parse target
3: function k-NN(target, k)
4:     Calculate userVectors[N][M]
5:     for user ∈ users do
6:         array[user] = euclDist(user, target)
7:     sort(array)
8:     output = parseTopKSubs(array)
```

4.2. Map Reduce

We essentially can break down the code into three distinct steps. (1) Data Processing step, (2) Model Training step, and (3) Recommendation step.

Preprocessing. Since only the "author" and "subreddit" fields of the raw reddit comments were needed for the project, the comments were preprocessed bringing the approximately 30GB file down to roughly 340MB. Each line contained the author, subreddit, and an affinity score that we add during the preprocessing step.

This preprocessing step was implemented both sequentially in Python and distributedly using Hadoop.

Training vs. Test Data. In general, it is difficult to calculate the accuracy of a recommendation engine without experimentally testing it with actual users. We utilize a standard method of measuring the accuracy of our recommendation strategies by using roughly 80% of our preprocessed data as training for our k-NN models and then running the remaining 20% of our preprocessed data as a test set.

The test set is constructed by taking 20% of the subreddits for each user from the training data, at minimum one subreddit for a user that has at least 2 subreddits. This ensures that all users exist in the training set with at least one subreddit as a correct

[1] https://github.com/logicx24/SubredditRecommendationEngine

recommendation and also that the test will cover as many users as possible.

Accuracy is calculated by outputting $n + 5$ recommendations, where n is the number of subreddits for a test user and the resultant recommendations are compared to the actual subreddits of the test user.

4.3. Model Training

Strategy 1 : Distributed Cache. To improve the performance and expand the scope of the k-NN implementation from its current sequential version, we wanted to use a distributed environment and thus implemented MapReduce jobs.

As described in the preprocessing section, we use training and test data, where test data includes all the users who we want a recommendation for. In this implementation, our algorithm is very similar to that in the sequential version, with a few key differences.

We split our algorithm in the following way. First, the mapper calculates the similarity (inner product) for every test user with all of the training users. Then it sorts the distances and writes them to a temporary file. The reducer then picks the top k training users (neighbors) for each test user from the temporary file and outputs those users.

Algorithm 2 Distributed Cache

```
1: function MAP(key, value)
2:     data = value.split()
3:     testUser = data[0]
4:     MaptestSubs = parseSubVector(data[1])
5:     for s ∈ trainVectors do
6:         trainSubs = parseSubVector(s)
7:         sim = innerProduct(trainSubs, testSubs)
8:     list = sort(sim)
9:     outKey = newStringBuilder()
10:    for e ∈ list do
11:        outKey.append(e.Key, e.Value)
12:    emit(testUser, outKey)
13: function REDUCE(key, List < value >)
14:    k = parseConfig()
15:    topK = parseTopKNeighbors(value, k)
16:    emit(key, topK)
```

To pass the training data to the mappers, we first used the configuration. This however, only allowed us to pass up to 100,000 training users. Increasing this number gave us java heap errors. We then tried to use the Distributed Cache, which did not give heap errors.

Strategy 2 : Preprocess All Pairs. The limitation with Strategy 1 above was that side data was required for calculation, which was facilitated using Distributed Cache. By using order inversion, we can compute the marginal distance between each user for each subreddit. This allows us to bypass the need for side data, as the preprocessed data set is sufficient to calculate the distances between all users.

Algorithm 3 SubCount Step

```
1: function MAP(key, value)
2:     data = value.split()
3:     emit(data[1], data[0] + "," + data[2])
4: function REDUCE(key, List < value >)
5:     outKey = newStringBuilder()
6:     for value ∈ List < value > do
7:         outKey.append(subreddit).append(value)
8:     emit(key, outKey)
```

Algorithm 4 Neighbor Step

```
1: function MAP(key, value)
2:     aScores = parse(value)
3:     Initialize parsedScores
4:     for u ∈ parsedScores do
5:         Calculate Final Key fKey
6:         Calculate Partial aScore
7:         emit(fKey, aScore)
8: function REDUCE(key, List < value >)
9:     Initialize distance
10:    for val ∈ List < value > do distance += val
11:    Calculate euclDistance
12:    emit(key, euclDistance)
```

4.4. Recommendation

The recommendation step for both strategies is essentially the same. The database contains the preprocessed data of users with a list of their subreddits and respective affinity scores. For strategy 1, we obtain the k nearest neighbors for all users from the MapReduce job and query the users name for a list of the k-nearest users. In Strategy 2, we obtain the k nearest neighbors by querying from all the preprocessed pairs the pairs that contain the users name and have the k shortest distances.

Algorithm 5 Recommendation Step

```
1: function DistributedCache(username)
2:     kNN = db.getNeighborhood(username)
3:     for u ∈ kNN do
4:         neighbors = db.getUserInfo(u)
5:     weightedSum(neighbors)
6: function AllPairs(username)
7:     pairs = db.getAllPairs(username)
8:     kNN = getKClosestNeighbors(pairs)
9:     for u ∈ kNN do
10:        neighbors = db.getUserInfo(u)
11:    weightedSum(neighbors)
12: function WeightedSum(neighbors)
13:    for u ∈ neigbors do
14:        for s ∈ u.subredditList do
15:            sum(aScore)
16:    return subreddit with greatest weighted sum
```

With the k-nearest neighbors, the recommendation engine then queries the list of subreddits and their respective affinity scores for each nearest neighbor and sum the affinity scores for each subreddit. The subreddit with the highest weighted score, is recommended.

5. Experiments and Evaluation

5.1. Setup

Dataset. The data used for this project comprised of the entire month of reddit comments made in January 2015. The dataset can be found here: http://bit.ly/1RmhQdJ.

Each comment in the data rest on a single line as a JSON block containing information about the comment, as shown in Listing **??**.


```
{
  "gilded": 0,
  "author_flair_text": "Male",
  "author_flair_css_class": "male",
  "retrieved_on": 1425124228,
  "ups": 3,
  "subreddit_id": "t5_2s30g",
  "edited": false,
  "controversiality": 0,
  "parent_id": "t1_cnapn0k",
  "subreddit": "programming",
  "body": "Hello World!",
  "created_utc": "1420070668",
  "downs": 0,
  "score": 3,
  "author": "thedoc",
  "archived": false,
  "distinguished": null,
  "id": "cnasd6x",
  "score_hidden": false,
  "name": "t1_cnasd6x",
  "link_id": "t3_2qyhmp"
}
```


Listing 1: Example JSON block

For this project, only the "author" and "subreddit" fields were used.

All of the analysis is performed with 1 million rows of preprocessed affinity score data. Where each row contains `<user, subreddit, affinity score>`. 80% of the data is utilized as a training set and 20% was turned into a test set. Due to resources and time constraints, we utilized a small test set, of 1 thousand rows, and a large test set, of 10 thousand rows. We checked to ensure that accuracy did not degrade with our implementations.

Infrastructure. We used a machine with an Intel Core™ i5-2557M Processor, 1.7 GHz, 4GB of RAM memory to run the sequential algorithm for the different versions of the dataset. All distributed computing tasks were performed on Amazon EMR with 1 master node and 4 slave nodes using Amazons c1.xlarge machines.

5.2. Results

1) Sequential kNN: We ran the sequential implementation on the preprocessed affinity score data. From the results shown in Table **??**, we see that the sequential version is nearly linear with the amount of input data.

Size	Execution Time (min)
1K	967
10K	10002

Table 1: Sequential Execution

2) Distributed Cache: Similarly, the distributed version (using distributed cache) takes considerably less time but is also linear with the amount of test data. This is because the test data is split between mappers, therefore, as the load on the mappers increase linearly, the program would take linearly more time. Comparing the performance of the sequential and distributed version(using distributed cache), we observe a speedup of around 27x, as shown in Table **??**.

Size	Execution Time (min)	Speedup
1K	34.66	27.89
10K	369.66	27.05

Table 2: Distributed Cache Execution

It is important to note that we have used 4 nodes in the distributed version, so if the sequential version alone was run on 4 nodes, we would expect a runtime of around 360 minutes (for 1K user data). This value is still considerably larger than the 34 mins it takes on the parallel distributed cache version. We can thus conclude, that the speed up observed is due to distributed computing as well as using a better algorithm.

3) Preprocessed All Pairs: For the Preprocessed All Pairs algorithm, a different approach was taken to overcome the file size limitations of distributed cache. In doing so, it incurs a greater upfront time cost to calculate the distances between all users. Larger speedups, as shown in Table **??**, are seen when larger test sets are given as input to the recommendation engine.

Size	Execution Time (min)	Speedup
1K	105.06	9.20
10K	255.06	39.21

Table 3: All-Pairs Execution

Overall, the speed up factor is less than the distributed cache method, with both strategies converging as the size of the test set grows towards the size of the training set.

Algorithm	Preprocess	Train	Recommend
Sequential	134	1000	0
Distributed Cache	20.5	18	16.67
Preprocessed All Pairs	20.5	88.4	16.67

Table 4: Stepwise Result Comparison in minutes

5.3. Limitations

Throughout the discussion of our work, limitations of the methodology have been identified that exist as points of improvements for further development of our proposed model. These points can be summarized as follows :

(i) The sequential version gives recommendations for only one input user. Any additional recommendations would require running the whole program again.

(ii) Using configuration to pass training data to mappers limited the size of the training data file. Passing more than around 100K user data would give Java Heap Error since the maximum job configuration size is around 5MB.

(iii) Distributed cache allowed faster access to training data than passing through configuration, however, this too had a size limit. Having 1 million users in the training file (64.4MB resulted in Java Heap Error). Therefore, the 20%-80% split for testing and training data respectively could not be practiced using Distributed Cache.

(iv) We had limited resources available on AWS, therefore we could only run our jobs for limited times, and were unable to experiment further with larger Instance types.

(v) The preprocessing all data strategy outputs exponentially increasing data files.

- 10GB output size for 100K training data
- 78GB output size for 1M training data

6. Conclusion and Future Work

Our distributed strategies, both using distributed cache and by preprocessing all data, have shown speedups over the naive sequential version. The distributed cache version followed a similar algorithm to the sequential version, and showed around a 27x speedup. In the second approach of preprocessing all data, we have an initial overhead of precalculating distances (appx. 88mins for 1M training data), however, subsequent recommendations only take appx. 1sec/user. Overall, this approach has a 9x speedup over the sequential version (for 1K test data) and a 39x speedup for 10K test data.

There are limitations to both the distributed versions. The distributed cache version needs to be given all the test data upfront along with the training data. If any other test data is to be used, the job needs to be run again with the other test data included in the input. On the other hand, the preprocessed all-pairs version will give recommendations for any set of existing users, without the need to run the job again, but outputs extremely large amount of data.

In future work, we would like to combine our distributed strategies. We can utilize the all pairs algorithm to calculate distances between users without the need for side data as in the distributed cache version. Then augment the all pairs algorithm by

restricting its output to the k neighbors for each user as done in the distributed cache version. This would allow recommendations to be made for all users in the dataset and output much smaller, manageable files. We also plan to experimentally find the optimal k values for calculations as well as the optimal number of distributed nodes for Hadoop calculations. Finally, we would like to port our algorithms to Spark to achieve even greater speedups.