**CLUSTKNN**

Collaborative Filtering algorithms for recommendation systems generally provide better quality recommendations and scale easily to accommodate huge amounts of data. One approach to improve the scalability of recommendation systems is to cluster the users and store only the centroids of each cluster. The cluster centers would then be used for prediction purposes. This vastly reduces the amount of users to be stored and leads to faster recommendations as only a small number of cluster centers would be used for computing the prediction score.

Moreover, a scalable recommendation system should not only be able to produce fast and high-quality recommendations with large data, but should also be able to accommodate any influx of users and items without requiring complete re-training of the data. Based on this property, collaborative filtering algorithms can be broadly classified into *memory-based* algorithms and *model-based* algorithms.

**Memory-based Algorithms**

Memory-based collaborative filtering algorithms utilize the entire database of users and their preferences to compute the prediction score of a user towards an item and to make a recommendation. As a result of this, it is trivial to handle addition of users and items into the system as no pre-computation of the training set is ever done. But, as the number of users and items grow, the performance of these algorithms drop, making them unsuitable for large systems.

**Advantages**

* Simple to implement and requires little or no training cost.
* Since the entire dataset is used for recommendation, their performance is generally better, though this majorly depends on the algorithm.
* Can easily accommodate influx of users and items or other changes in the data.

**Disadvantage**

* Does not scale efficiently as no pre-computation is performed.

**Model-based Algorithms**

Model-based collaborative filtering algorithms pre-compute a model of the preference data of the users. The size of this model is generally much smaller compared to the complete dataset. The model is then used to produce recommendations. Since the process of building the model is time consuming, this is done periodically; depending upon how frequently new users and items are added to the system.

**Advantages**

* Recommendations are faster as they are based on the model, and hence these algorithms are highly scalable.
* Better space utilizationas models are compact compared to the datasets.

**Disadvantages**

* Building the model is time consuming.
* Coping up with the influx of users and items is not simple. The model should be periodically re-built.

**Hybrid Approach**

*ClustKNN* algorithm is a hybrid of memory-based and model-based techniques, combining their advantages. This algorithm employs partitional clustering, in which the users are divided into disjoint clusters, to model the users. The number of clusters is a dominating factor in determining the speed and the accuracy of predictions. Since the data is greatly compressed after the model is built, recommendations can be computed quickly, which solves the scalability challenge.

The algorithm uses *bisecting k-means* clustering technique, which is an extension and an improvement to the basic k-means clustering algorithm, to model the users into clusters. An interesting property of ClustKNN is that it can be tuned to fit the size of the data, by varying the parameter k of the clustering technique, to produce different numbers of clusters for different sizes of data, thus making the recommendation system scalable.

**Algorithm**

The algorithm has two phases: model building (offline) and generation of predictions or recommendations (online).

**Model Building**

The training phase takes as input *n* users, where each user is represented by an *m*-dimensional vector of normalized affinities, in the range [-1.0, 1.0], corresponding to *m* items.

1. Select the number of user-clusters, *k*, based on the size of the data.
2. Apply *bisecting k-means* clustering algorithm to partition the users in to *k* disjoint clusters.
3. Build the model by computing the centroid of each cluster. Each centroid represents a surrogate user that is representative of all the users in that cluster.
4. Retain only the *k* centroids (surrogate users): {c1 , c2 , . . . , ck}, where each ci is a vector of size *m*, the number of items. That is, ci = (, , . . . ,), where each is the element in the centroid of the vector ci corresponding to the item aj. Further, since is essentially an average value, it is 0 if nobody in the *i*-th cluster has rated aj­.

**Prediction Generation**

In order to compute the rating prediction, for the target (user, item) pair (ut , a­t), the following steps are taken:

1. Compute the similarity of the target user with each of the *k* surrogate model users using the Pearson correlation coefficient:

and are the means of the *m* values in the target user vector, ut, and the surrogate user vector, ci, respectively.

1. Sort the surrogate users based on the similarity to the target user, ut­, from highest to lowest. Choose the top *l* surrogate users (the ones that are most similar to ut).
2. Compute the prediction based on the *l* surrogate users using the adjusted weighted average:

By choosing an appropriate value for *k*, the time taken to compute the prediction score can be greatly reduced. If *k* is not too large, then the value *l* can be chosen as *k*, as the adjusted weighted average (where the weights are Pearson correlation coefficients) would implicitly assign greater importance to the most similar surrogate users, while the least significant surrogate users would practically be insignificant.

**Bisecting K-Means Clustering**

K-Means algorithm produces partitional clusters of *n* data tuples into *k* clusters by repeatedly assigning each data tuple to the cluster with closest mean. After each iteration, the cluster means are updated to the centroid of the data tuples in the cluster. *Bisecting k-means* is a divisive hierarchical clustering technique and is an extension to and an improved version of the basic k-means clustering algorithm. The algorithm starts by considering all the data points as a single cluster. Then it repeats the following steps (*k* – 1) times to produce *k* clusters.

1. Pick the largest cluster to split.
2. Apply the basic *k­*-means algorithm (with *k* = 2) to produce 2 sub-clusters. This is the bisecting step.
3. Repeat the steps 1 and 2 (*k* – 1) times to produce *k* clusters.

The basic *k*-means algorithm produces clusters of varied sizes. It does not give any guarantee on the size of the clusters and hence it is quite possible that a single cluster contains a large portion of the entire dataset, limiting the usefulness of this algorithm for improving scalability. Since bisecting *k*-means always splits the largest cluster into two sub-clusters, the algorithm tends to produce balanced clusters. As a result, the dataset is evenly partitioned into *k­*-clusters, and hence no one cluster will act as a bottleneck for scalability.

**Complexity**

The time-complexity of the basic *k-*means is reported to be *O(n)* for *n* data points. However, this is assuming the cost of computing the similarity of distance between the data points and the centroids is constant. In ClustKNN, this complexity is *O(m)*, where *m* is the number of items. Hence, the time-complexity of the basic *k-*means in ClustKNN is *O(nm)*. Since bisecting *k-*means applies the basic *k*-means algorithm (*k* – 1) times, its time-complexity is *O((k-1)nm)*. This is the offline complexity of ClustKNN.

During the online stage, *O(k)* similarity weight calculations are needed for the target user. Since computing the Pearson correlation coefficient between the target user and a surrogate user takes *O(m)* time, the time-complexity of the online stage is *O(km)*. If the number of clusters, *k*, is chosen to be sufficiently small compared to the number of items in the dataset, *m*, then the time-complexity of the online stage can be approximated to *O(m)*.

ClustKNN algorithm requires *O(nm)* space during the offline stage to store the entire dataset while producing the *k* clusters. The *k* clusters together asymptotically occupy *O(nm)* space.

For the online stage, only the surrogate users, i.e., the centroids of the clusters are required. Since there are *k* clusters in total, the space-complexity for the online stage is *O(km)*. The remaining data points in the clusters can be discarded after the offline stage. Since each target user takes up *O(m)* space during the online stage, the total space-complexity of the online stage is *O(km + m)*. If *k* is sufficiently small compared to *m*, then the space-complexity of the online stage can be approximated to *O(m)*.

A major advantage of ClustKNN is that it can make recommendations even for users who have not been trained using the algorithm. The online stage only requires the comparing the similarity with the surrogate users. This technique has the properties of a memory-based algorithm, wherein, it is trivial to handle influx of users into the system. The algorithm is also partly model-based as the offline stage builds a model and by storing only the surrogate users, the dataset is made compact. Thus, ClustKNN is a hybrid of model-based and memory-based collaborative filtering algorithms.

**Experiments and Results**

During the first leg of the experiments using ClustKNN, the parameters – the number of clusters, *k*, the number of most similar clusters to be considered for prediction, *l*, and *threshold* score for positive recommendation were optimized using the technique of grid optimization, wherein, equally spaced parameter values within a specific range are applied and the parameter set that produces the best performance over 10-fold cross-validation is chosen.

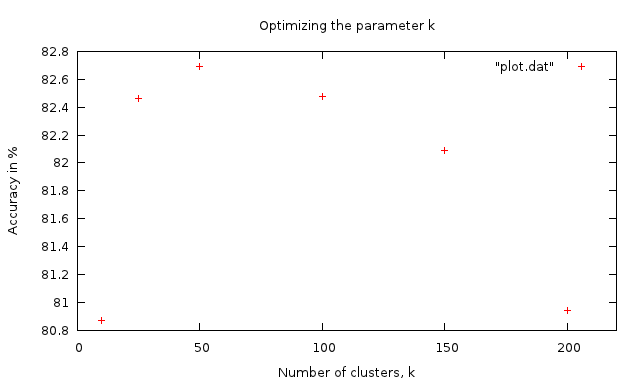
**Optimizing the parameter *k***

The number of clusters, *k¸* plays an important role in deciding the speed and the scalability of the algorithm. If *k* is too small, then the accuracy on real world data would be affected as a result of over-fitting. On the other hand, if *k* is too large, the algorithm would not scale well with the influx of users and categories. Hence, choosing an optimal value of *k* is essential.

|  |  |
| --- | --- |
| ***k*** | **Accuracy in %** |
| 10 | 80.87 |
| 25 | 82.46 |
| 50 | 82.69 |
| 100 | 82.48 |
| 150 | 82.09 |
| 200 | 80.94 |

Optimizing the number of clusters, *k*

The above table shows the accuracies obtained for different values of *k* after 10-fold cross-validation. The algorithm was performed over the *attr100* dataset, with the parameter *l* set to *k*, and *threshold* set to 0.12. The corresponding plot is depicted below:



Parameter *k* vs Accuracy

With 10 clusters, the accuracy obtained is comparatively lower that the accuracies obtained with more than 50 clusters. Although the difference between the accuracies when *k* = 50 and *k* = 200 is not much, we choose 50 as the number of clusters for all the further experiments, as the speed of the algorithm is influenced by it.

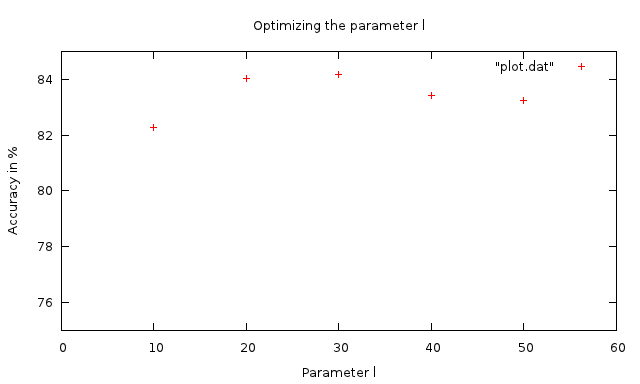
**Optimizing the parameter *l***

During the online stage of the algorithm, the parameter *l* determines the number of most similar clusters to take into account while computing the prediction score. The following table contains the accuracies obtained over 10-fold cross-validation on *attr100* dataset for different values of *l*. The number of clusters, *k*, is set to 50 and the *threshold* is chosen to be 0.12.

|  |  |
| --- | --- |
| ***l*** | **Accuracy in %** |
| 10 | 82.28 |
| 20 | 84.02 |
| 30 | 84.17 |
| 40 | 83.43 |
| 50 | 83.23 |

Optimizing the parameter *l*

The plot for the above table is shown below:



**Parameter *l* vs Accuracy**

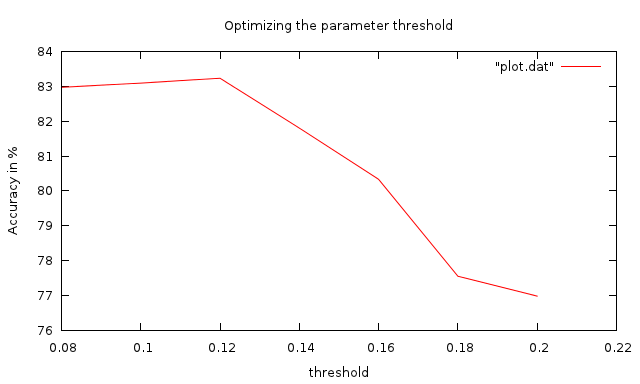
It can be inferred from the table above that, since the number of clusters, *k*, is sufficiently small, the value of the parameter *l* does not result in any major difference in the performance of the algorithm. Moreover, the time-complexity of the online stage, even though dependent on *l,* is not affected much by minor differences in *l,* as the value is sufficiently small. Hence, for all the future experiments, we have set the parameter *l* to be always equal to the number of clusters, *k*.

**Optimizing the parameter *threshold***

The parameter *threshold* marks the division between those target (user, item) pairs that are to be classified as positive predictions and those that are to be classified as negative predictions. The optimal value for *threshold* is chosen by running cross-validation on the *attr100* dataset, with the parameters *k* and *l* set to 50, for different values of *threshold*.

|  |  |
| --- | --- |
| ***threshold*** | **Accuracy in %** |
| 0.08 | 82.97 |
| 0.10 | 83.09 |
| 0.12 | 83.23 |
| 0.14 | 81.80 |
| 0.16 | 80.33 |
| 0.18 | 77.55 |
| 0.20 | 76.98 |

Optimizing the parameter *threshold*



Parameter *threshold* vs Accuracy

The plot depicts a clear picture of the optimal *threshold* value and how the performance of the algorithm varies with the *threshold* value. The optimal *threshold* of 0.12 is chosen for all the experiments.

**Performance of ClustKNN (with and without bisecting k-means)**

Experiments using ClustKNN on both the *attr100* and *attr200* datasets with the optimal parameter set (*k* = *l* = 50, *threshold* = 0.12), gave interesting results when using bisecting k-means compared to basic k-means.

|  |  |  |
| --- | --- | --- |
| **Dataset** | **Basic k-means**  **(Accuracy in %)** | **Bisecting k-means**  **(Accuracy in %)** |
| attr100 | 77.90 | 82.97 |
| attr200 | 64.41 | 82.35 |

Performance of ClustKNN with and without bisecting k-means

The confusion matrices for the *attr100* dataset for both versions of the ClustKNN algorithm (using basic k-means and bisecting k-means) are shown below:

|  |  |
| --- | --- |
| TP = 15715 | FN = 2204 |
| FP = 2757 | TN = 1770 |

Precision = 85.07%, Recall = 87.70%

**Confusion Matrix for *attr100* dataset with basic k-means**

|  |  |
| --- | --- |
| TP = 16507 | FN = 1412 |
| FP = 2409 | TN = 2118 |

Precision = 87.26%, Recall = 92.12%

**Confusion Matrix for *attr100* dataset with bisecting k-means**

On the dataset *attr200*, the confusion matrices obtained with basic and bisecting k-means ClustKNN experiments are:

|  |  |
| --- | --- |
| TP = 9797 | FN = 5276 |
| FP = 1427 | TN = 2334 |

Precision = 87.28%, Recall = 64.99%

**Confusion Matrix for *attr200* dataset with basic k-means**

|  |  |
| --- | --- |
| TP = 12663 | FN = 2410 |
| FP = 913 | TN = 2848 |

Precision = 93.27%, Recall = 84.01%

**Confusion Matrix for *attr200* dataset with bisecting k-means**

From the above tables, it is clear that bisecting k-means results in better performance over the basic k-means algorithm. The accuracy of the basic k-means version of ClustKNN on *attr100* dataset is 77.90%, while on *attr200* dataset is 64.41%. The result of the drop in accuracy with the increase in dataset size is because, for larger datasets, basic k-means produces clusters that vary vastly in their sizes. As a result, even if one cluster is very large, the performance of the algorithm will be affected by that cluster.

But, with bisecting k-means version of the ClustKNN algorithm, the sizes of the clusters produced are more or less equal. As a result, the algorithm is a lot more scalable compared to the basic k-means version. This is experimentally shown by the fact that the accuracies produced for the *attr100* and *attr200* datasets by bisecting k-means version of the algorithm are 82.97% and 82.35% respectively.

ClustKNN is not just scalable, but also produces better accuracies compared to the LSA-SVD algorithm. Moreover, the online phase requires far less space compared to the online phase of the LSA-SVD algorithm, and this is a very significant factor while working with recommendation systems that handle huge data.