# Analysis and performance optimization of k-nearest neighbor approach for movie queries

Project Report - CSE 5522

Manjari Akella
Gaurav Singh
Harsh Shah

12/5/2013

## Problem Definition

The idea of our project was two part – the first was to pursue the knn algorithm to predict movies in response to a user query, the second was to look at ways to optimize the knn algorithm to better the performance time.

The user starts the query with a movie they have already seen. They can then add and/or subtract the content of each genre to get a new movie prediction which fits the description they asked for. The following is an example query that the user can ask –

- *"When Harry Met Sally" – Romance + Thriller =* ***"From Dusk till Dawn"***

The system returns the movie in bold.

We have tried 2 performance optimizing techniques – pre-clustering using knn and k-d trees. We propose the use of another technique – Locality Sensitivity Hashing; which given the constraint of time, we couldn't implement.

We also plan to upload a demo version online with an interactive front-end to represent our analysis in an intuitive format.

## Machine Learning Algorithms Used

Our project uses the knn algorithm to first predict the movie based on the user query. We then try to optimize its performance using 3 techniques –

- Pre-clustering using k-means *(Implemented)*
- k-d trees *(Implemented)*
- Locality Sensitivity Hashing *(Future Scope)*

## knn

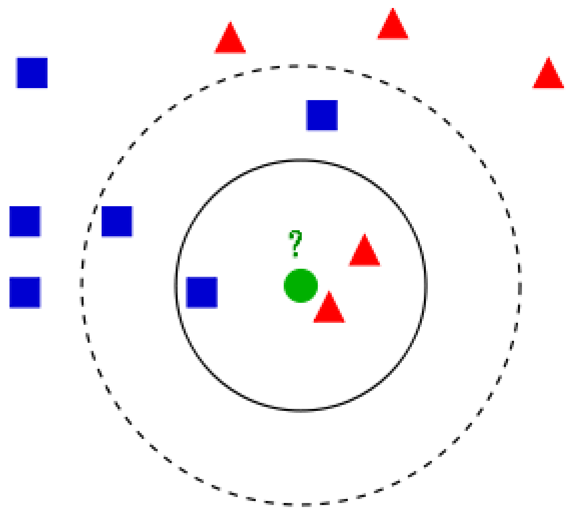

**Figure 1: knn**

The knn (aka k-nearest neighbor) approach is one of the simplest classification techniques used. It is a type of lazy learning technique wherein all computation is deferred until a new test point actually arrives. It operates based on the k closest training points to the new test point in the feature space. It looks to find the k nearest neighbors of the new test point based on some distance metric. It is a type of majority classifier which takes into account the k nearest neighbors' votes. It assigns to the new point, the label most common across its k nearest neighbors.

## Optimization: Pre-clustering using k-means

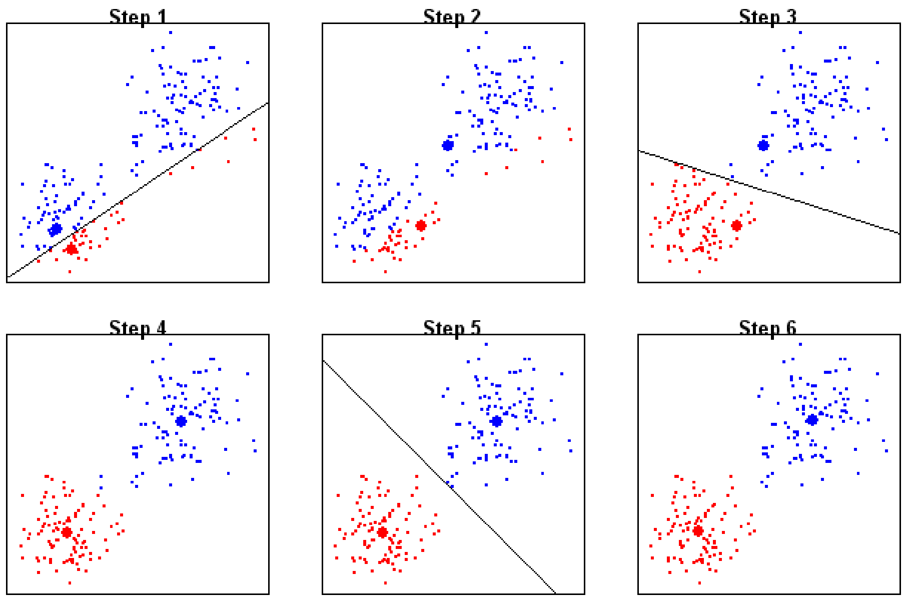

**Figure 2: k-means clustering [1]**

The k-means algorithm is quite similar to knn. However, instead of looking at nearest neighbors, it looks to find the nearest mean. Each class (cluster) is assigned a mean. The mean to which a new test point is closest to represents the prediction answer. Closest mean is again found out on the basis of some distance metric.

## Optimization: k-d trees

k-d trees (aka k-dimensional tree) are a type of space partitioning data structure used for organizing points in a k-dimensional space. It basically splits the data into left and right subtrees iteratively over each axis.

The construction of a k-d tree is pretty straight forward. We construct a tree with a split at each level on one of the k (number of dimensions) axes. Each split of the tree represents a split in the corresponding axis.

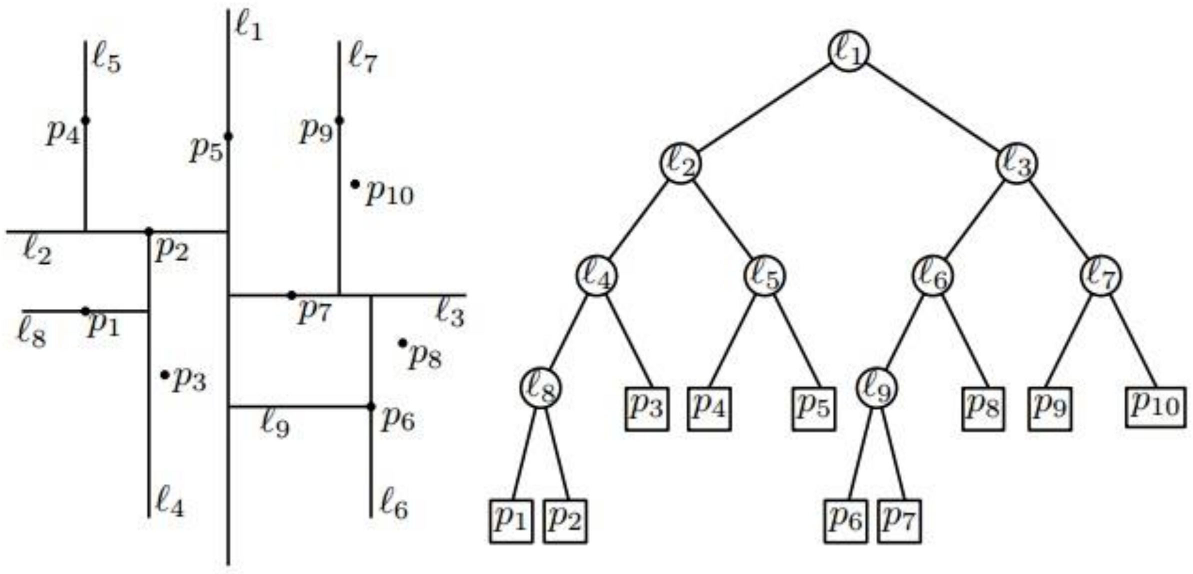

**Figure 3: k-d tree (split in feature space, tree representation)**

At each level, the median value of the axis being split on is computed. All those points whose axis value is less than the median go to the left subtree and those which are higher go to the right subtree. However, the computation of median makes no sense in our case; where the feature vector is a combination of zeros and ones. Hence we decided to split based on zeros and ones. Those points whose axis value is zero go in the left subtree and those whose axis value is one go to the right subtree.

One of the major applications of k-d trees is its use in nearest neighbor searches. If we have enough train data points, the number of searches made significantly reduces. Given a new test point, we can eliminate a huge part of the search space which reduces the query time significantly.

## Optimization: Locality Sensitivity Hashing

The locality sensitivity hashing technique (aka LSH technique) is a method of performing probabilistic dimension reduction of high-dimensional data. LSH is based on the simple idea that, if two points are close together, then after a “projection” operation these two points will remain close together.

We start with a random projection from the high dimensional space to a lower dimensional subspace. The first step involves a scalar product of the query point in the high dimensional space with a vector in a lower dimension with components that are selected at random from a Gaussian distribution, for example N(0,1). This value obtained is then quantized into a set of bins such that nearby items in the original space fall into the same bin. [2]

Hence the nearest neighbor search would now have to look through the points only in the bin where the new query point has been classified instead of looking through all the points which makes the process a lot faster.

## Training and Testing of the System

We collected our data from various sources online. Our data set consists of 3484 movies which were custom crawled from the Freebase and MovieLens 10M databases.

Our feature vector consists of 376 columns for each movie. Each of the columns represents a genre (comedy, drama, romance, action, thriller, animation etc.)

Each column takes a value of 0 or 1. If a movie belongs to a genre, it takes a value of 1, if not; it takes a value of 0. This combination of zeros and ones represent each movie.

We used the movie set collected as the train points. We then added/subtracted the genres to get a new test point.

The performance time of each of the methods was tested.

## Results and Discussion

We ran our code in different scenarios to evaluate each of the models, viz. the plain K-Nearest Neighbor, K-Nearest on clustered data, and K-Nearest Neighbor using k-d trees, in terms of time taken to classify a new incoming query. Additionally, for the clustering process, we have also analyze the quality of our clustering algorithm using three different distance or similarity measures, namely Manhattan distance, Euclidian Distance and Cosine Similarity. The IDE cache was cleaned prior to each run of the program, just to make sure no older values remained stuck in the memory buffers.

Time taken for each of the three algorithms to classify a new query can be compared from Figure 4.

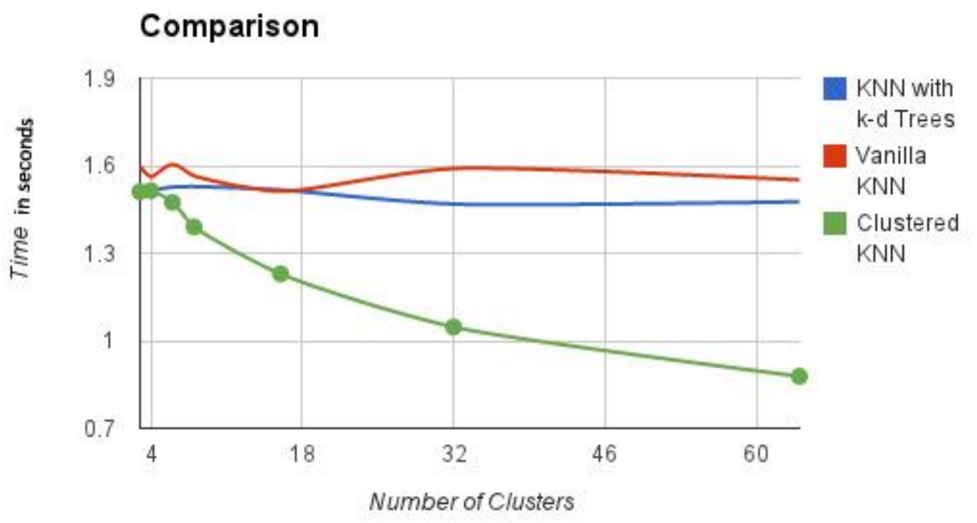

**Figure 4: Comparision - kNN vs clustered kNN vs KNN with k-d Trees**

Overall, we observed that kNN on clustered data did the best job in terms of performance. Qualitatively, too, it seemed to provide better results, and more closer to what we provided as our input query. We have tried k-means clustering for varied values of k (3, 4, 6, 8, 16, 32 and 64), and the time to classify expectedly decreases, since with increasing clusters, the kNN algorithm had to progressively work with a reduced set of feature vectors. Unexpectedly, K-d trees did not provide much improvement over the plain KNN algorithm in our case. This could be attributed to the imbalance in our data, i.e. comparatively less data points when compared with the dimensionality. If the dimensionality of the data is k, a k-d tree works well only if the number of data points is more than $2^k$. Otherwise, we end up testing nearly all the nodes in the dataset making the complexity O(n) instead of the desired O(log(n)).

To evaluate our quality of clustering, rather than finding the Mean Square Error within a cluster, we calculated a more intuitive metric, and we call it our Genre Entropy. It determines how pure our clusters are with respect to the genres present in it. The different genres for the movie set are identified within a cluster and their counts are calculated.

Entropy is then calculated as:

$$S = -\sum \frac{xi}{C} \log \frac{xi}{C}$$

where, $x_i$ = the count for a genre ‘i’ in a cluster

C = Number of movies in the cluster

For ‘k’, clusters we have k different entropies which are used to calculate the overall weighted entropy of the entire clustering using the following function,

$$Total\ Entropy = -\sum (Si\ x\ Ni)/M$$

where, Si = Entropy of the ‘i’th cluster,

Ni = Total number of movies in the ‘i’th cluster

M = Total number of movies in the corpus

Graphs between ‘k’ (using different distance metrics) and the genre entropy is then calculated to get a curve as follows:

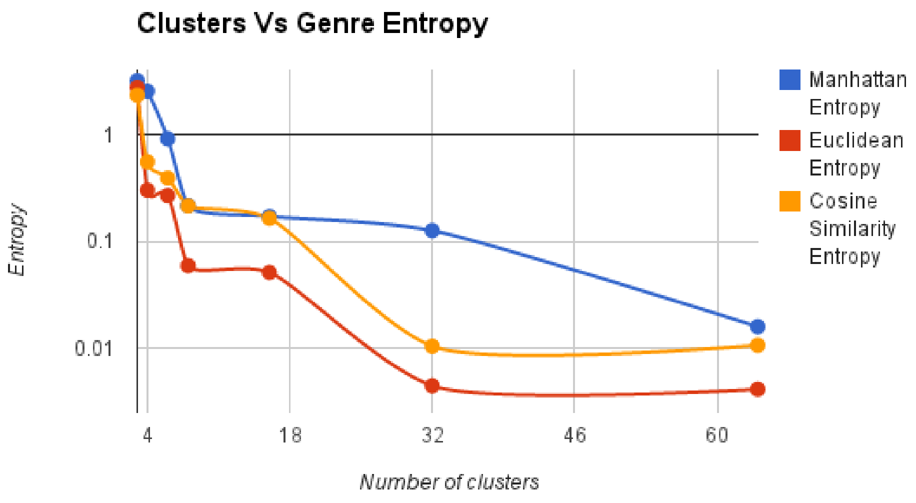

**Figure 5: Clusters vs genre entropy - k-means algorithm**

The entropy decreases with the increase in ‘k’, the number of clusters. This is expected, because with lesser number of ‘k’ various movies with different genres will be included within a cluster, and as such there will less homogeneity, leading to higher entropies. Also, Euclidian distance seems to perform better than Cosine similarity and Manhattan distance.

As for the time to converge, between all the three cases, the graph is mostly linear in the Number of clusters ‘k’ (for k = 3, 4, 6, 8, 16, 32 and 64). Cosine Similarity expectedly takes the maximum time because of the computation involved.

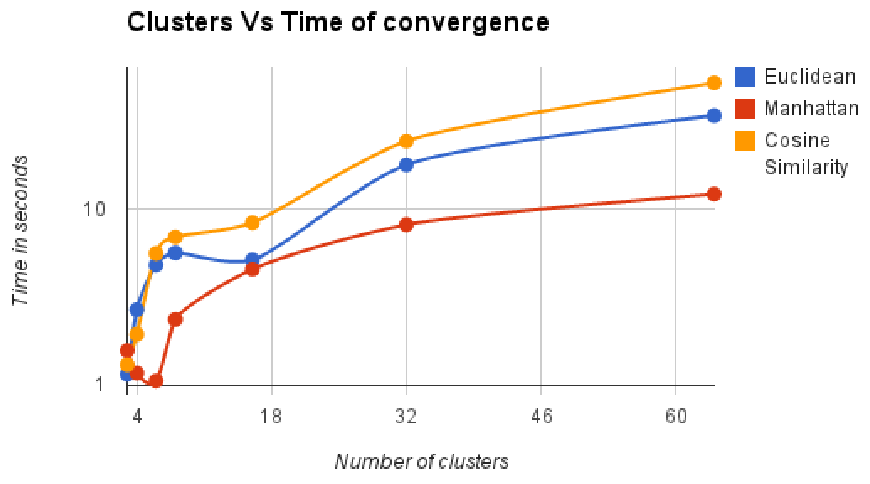

**Figure 6: Clusters vs Time Convergence - k-means algorithm**

## Future Scope

Due to the unexpected roadblock we encountered when dealing with k-d trees, we propose to use the locality sensitivity hashing to see if it works any better. However, due to the limited time at hand, we couldn't get to the implementation stage of it. Looking at the research paper we studied [2], we expect the LSH algorithm to perform significantly better.

## References

[1] http://astrostatistics.psu.edu/su09/lecturenotes/clus2.html

[2] [Malcolm Slaney and Michael Casey – "Locality-Sensitive Hashing for Finding Nearest Neighbors ", IEEE Signal Processing Magazine – March 2008]