International Baccalaureate®
Baccalauréat International
Bachillerato Internacional

Extended essay cover

Candidates must complete this page and then give this cover and their final version of the extended essay to their supervisor.

Candidate session number			
Candidate name			
School name			
Examination session (May or November)	May	Year	2015

Diploma Programme subject in which this extended essay is registered: Computer Science

(For an extended essay in the area of languages, state the language and whether it is group 1 or group 2.)

Title of the extended essay: Recommender Systems for a more accurate estimate of user's preferences.

Candidate's declaration

This declaration must be signed by the candidate; otherwise a mark of zero will be issued.

The extended essay I am submitting is my own work (apart from guidance allowed by the International Baccalaureate).

I have acknowledged each use of the words, graphics or ideas of another person, whether written, oral or visual.

I am aware that the word limit for all extended essays is 4000 words and that examiners are not required to read beyond this limit.

This is the final version of my extended essay.

Candidate's signature: ______ Date: 20 February 2015

Supervisor's report and declaration

The supervisor must complete this report, sign the declaration and then give the final version of the extended essay, with this cover attached, to the Diploma Programme coordinator.

Name of supervisor (CAPITAL letters) ______________________________

Please comment, as appropriate, on the candidate's performance, the context in which the candidate undertook the research for the extended essay, any difficulties encountered and how these were overcome (see page 13 of the extended essay guide). The concluding interview (viva voce) may provide useful information. These comments can help the examiner award a level for criterion K (holistic judgment). Do not comment on any adverse personal circumstances that may have affected the candidate. If the amount of time spent with the candidate was zero, you must explain this, in particular how it was then possible to authenticate the essay as the candidate's own work. You may attach an additional sheet if there is insufficient space here.

- The student understood the difference between the algorithms that he compared and their usefulness in recommendation based systems.
- He faced technical difficulties related to the size of the sample, wich he solved without altering the validity of the experiment.
- He invested a great amount of time into executing the tests to validate his experiments, obtaining conclusive results.
- He made a complete and extensive documentary research to formulate his research questions based on solid statements.

This declaration must be signed by the supervisor; otherwise a mark of zero will be issued.

I have read the final version of the extended essay that will be submitted to the examiner.

To the best of my knowledge, the extended essay is the authentic work of the candidate.

As per the section entitled "Responsibilities of the Supervisor" in the EE guide, the recommended number of hours spent with candidates is between 3 and 5 hours. Schools will be contacted when the number of hours is left blank, or where O hours are stated and there lacks an explanation. Schools will also be contacted in the event that number of hours spent is significantly excessive compared to the recommendation.

I spent 5 hours with the candidate discussing the progress of the extended essay.

Supervisor's signature: ______________ ______________ Date: 23 FEBRUARY 2015

Abstract

Recommender systems, which predict how high a user would rate an item, have been researched for a long time in the industry and academia due to the advantages they present on computer-user interaction as a way of filtering data overload to the user. They have also been used extensively by media and e-commerce services in order to achieve *personalization* of these services, catering to many users and increasing sales. Four main types of recommendation systems are identified currently: *content-based memory-based*, *content-based model-based*, *collaborative content-based*, and *collaborative model-based.* Each of them presenting advantages and disadvantages. But, how do current recommender system types compare to each when evaluated against each other in controlled environment? This essay will identify the strengths and weaknesses of each through experimentation and analysis in order to compare them objectively. A representative algorithm will be chosen for each of these recommender systems, and then implemented on a controlled environment to be run against the *Netflix dataset*. Data will be gathered on algorithm speed and accuracy, for a varying sample size. The results show that content-based algorithms performed poorly against their counterparts. This gives insight into the type of datased used. It was determined that the Netflix dataset is fundamentally collaborative, as it has favors these types of algorithms. By analyzing how the results varied with the sample size, the potential benefits of each algorithm were shown. Content-based systems perform better on environments which provide extra metadata relating to items. Collaborative systems perform better on systems with a large number of users, memory-based systems are more accurate than model-based systems with smaller datasets. These results reinforce how these algorithms are used in the industry, the data provided by the tests appears to reinforce the way these algorithms are thought of.

A=
RQ
Met
?

Word Count: [291]

J -> 2

Assessment form (for examiner use only)

Candidate session number		

Achievement level

Criteria	Examiner 1	maximum	Examiner 2	maximum	Examiner 3
A research question	~~1~~ 2	2		2	
B introduction	2	2		2	
C investigation	4	4		4	
D knowledge and understanding	3	4		4	
E reasoned argument	3	4		4	
F analysis and evaluation	3	4		4	
G use of subject language	3	4		4	
H conclusion	2	2		2	
I formal presentation	3	4		4	
J abstract	2	2		2	
K holistic judgment	3	4		4	
Total out of 36	30				

of examiner 1: ____ Examiner number: ____
AL letters)

of examiner 2: ______________________ Examiner number: ______________
AL letters)

of examiner 3: ______________________ Examiner number: ______________
AL letters)

IB Assessment Centre use only: **B:** ________

IB Assessment Centre use only: **A:** ____________

RQ is never
precisely stated
A→ 1(2

Recommender systems for a more accurate estimate of user's preferences

Word Count: [3312]

Table of Contents

? ? ? ? - - - - - - - - - - - 4

Introduction ? ?

With the rise of the Internet, information available has increased to a scale never seen before. The information overload experienced by customers of today's web services gave rise to the need of *personalization* or technologies that present customers with information they deem useful [1][3][5]. Many techniques have been developed in order to address this multifaceted problem, with *recommender systems* being one of these techniques [1].

Recommender systems are expert systems that address the specific problem of providing item recommendations for users [1]. They date back to the mid-90's, when the recommendation problem was formally established by independent research [2][10]. Recommender systems have found a big use in the industry due to the advantages they present on predicting user behavior patterns [2][4]. Many businesses require such insight into their users in order to provide a better service. Recommender systems have been an active area of research due to the growing number of media and e-commerce services on the web [4]. The ability of being able to predict what a user may like or not may determine the difference between maximizing sales for an e-commerce site or missing potential customers [6]. The search for the best algorithm to provide recommendations is an ongoing one because only through the use of clever algorithms will we achieve true *personalization* in human-computer interaction [3]. The objective of this paper is to provide a complete and extensive analysis of the current available techniques for constructing recommender systems, and to expose the advantages, disadvantages, and extent of each through experimentation.

As the aforementioned recommendation problem is a very general one and includes many different areas, different approaches have been used to create recommender systems, depending on the context and requirements of the industry that implemented these systems [6]. Techniques derived from machine learning and approximation theory have long been utilized to build such systems [1], where the machine learning methods have been proven to be the most successful in the majority of areas due to their scalability [4]. These different techniques have evolved over time and differ on the way they approach the problem. According to [11] these approaches can be categorized into the following two broad categories: content-based filtering (also known as individual filtering) and collaborative filtering. A hybrid category is also proposed for those recommender systems which use techniques from both aforementioned categories.

B → 2 3/4

need a brief intro to the methods

1. Content-based vs Collaborative Filtering

Recommender systems aim to predict the *utility* a user would find from an item [2], This utility is often presented in the form of a *rating* which can be given explicitly by the user or implicitly through his behavior [2]. Recommender systems work on the assumption that there exists a correlation between user preferences and ratings given to items.

The first approach to constructing recommender systems is collaborative filtering. Collaborative filtering algorithms predict users' preferences through the use of data collected from other users on the system. They work on the assumption that a user *c* will rate an item s in a similar way that a similar user *c'* would. They effectively are able to predict the preferences of any user based on the preferences of other users deemed similar [3][4]. Formally, collaborative filtering tries to predict the utility *u(c,s)* for user *c* and item *s* by approaching it by *u(c',s)* [2].

how do they know they are similar?

On the other hand, content-based filtering does not rely on other users to provide recommendations, but instead aims to provide them by comparing items previously rated by the user [2][4][5]. It relies on the assumption that user *c* will rate item *s* in a similar way as was done with a similar item *s'* It is the opposite to collaborative filtering in that it tries tries to predict the utility *u(c,s)* for user *c* and item *s* by approaching it with *u(c,s')* [2][8].

good k+

Content-based algorithms perform best on contexts where there is an abundance in data pertaining to items [3]. An example of a service where content-based filtering could be useful could be a news article aggregator that personalizes content for the user. This is because items can be easily described by tags or keywords from the article. For many applications pure content-based filtering can prove insufficient, because a person is likely to have many interests and preferences that may not show through the items they have rated.[3]

There is a problem, however, with the two methods mentioned above known as the cold-start problem [3]. This problem emerges when there is no data in the recommender system and thus it's not able to make recommendations. Since these systems are dependent on past experiences, they will not be able to provide recommendations if there is no data present.

This is why, to solve the cold-start problem, many businesses opt to provide simple recommendations based on best sellers, or most viewed. Which are examples of additive recommendations [1]. The data feed top the recommender system will determine the outcome of its predictions on the users, so choosing carefully this will provide a better context for their users.

Ci —

?

not proof-read K-

2. Memory-based vs Model-based Filtering

Aside from these two categories, further two categories have been proposed by [9] that describe the implementation of recommender systems. These categories describe the way ratings are computed instead of focusing kind of data used to compute ratings. These are memory-based (also known as heuristic-based) and model-based.

Memory-based algorithms were among the first used to predict user preferences [4]. They analyze present data and manipulate it in a meaningful way to obtain a result. Because of this, memory-based algorithms depend directly on the data stored in a database [9].

An example of a memory-based algorithms could be the *K-Nearest Neighbor algorithm* which predicts a user's preferences by combing the preferences other users graded by their closeness to the user in question [4]. Memory-based techniques use available system memory to store data needed to provide recommendations. This can lead to manageability problems, as database sizes tend to become very large [9]. Services with a database that is neither sparse nor fragmented may benefit from such recommender systems.

explain

Memory-based algorithms are considered comparatively easier to implement, so they have been used extensively for many different applications [4]. But with a growing

- it is not clear how this functions ?
- does the candidate understand it ?

demand for accurate and fast recommendations, other methods that improve the speed of recommendations were looked into.

The other type of algorithms used to predict users' preferences are model-based algorithms [11]. These algorithms use techniques from machine learning and data mining to operate [3][6]. They do not depend directly on present data to calculate ratings, and because of this tend to be faster. Model-based algorithms differ from memory-based techniques in that they require previous training from the database to generate a model. This model can then be applied to any user and item and predict the rating given [3].

Model-based algorithms have been proven to be much less vulnerable to malicious data insertion and spam due to their nature [12]. Model-based algorithms take the **system as a whole** and are able to overlook noise. This is one of the primary reasons these algorithms are among the most popular in recommender systems [12]. This way, the system cannot be tricked into giving false recommendations to a user based on how other users manipulated the recommendations into fitting their objective [12]. ?

Examples of model-based algorithms are *data clustering* algorithms, *Bayesian network* algorithms [2], and recently *singular value decomposition* algorithms [13]. Model-based algorithms aim to establish relationships between items and users with data taken from their behavior, but doesn't depend on the stored data [4]. The data is only used in order to generate a model that predicts accurately user behavior patterns. Model-based techniques uncover the underlying patterns of user behavior, but this information is not useful outside the context specified by the algorithm [3].

There exist hybrid systems which combine memory-based and model-based techniques [11]. They create an interesting possibility because they take elements from both techniques in order to better handle data [14]. These hybrid systems are often the most used one because of their advantages, and are used when a specific set of algorithms doesn't fit an application's goal [3].

The candidate is too dependent on the source material.

3. Algorithm implementations

Excluding the hybrid systems, four types of recommender systems can be identified: memory-based collaborative, model-based collaborative, memory-based content-based, and model-based content-based. With these different types of recommender systems, the following question arises: "How do current recommender system types compare to each when evaluated against each other in controlled environment?" In order to get such insight, tests were performed with each algorithm in a controlled environment to compare them in a quantitative way and to analyze their implications.

? In 2011 *Netflix* created a recommender system contest known as the *Netflix Price*. It consisted of making a recommender system capable of accurately predicting ratings present on their site. To test their algorithms, contestants trained their recommender systems with an anonymous dataset provided by Netflix and compared the predictions with a set of answers also given in the dataset. The ratings provided in this dataset are of 17770 movies and were made by 480189 users. Further information concerning the dataset is given in the Appendix.

To provide real results, the Netflix dataset was used. Concerning the algorithms, the standard implementation for each recommender system was evaluated, having speed and accuracy as the main criteria. An average rating recommendation, along with a random recommendation were also tested to serve as a control for the experiment. ? Algorithm accuracy was measured by applying the RMSE criteria to the resulting predictions.

3.1 Memory-based collaborative filtering

Memory-based collaborative filtering was used for the first test. The most common implementation for this type of recommender systems is done by defining the utility function for user c and item s as an aggregate function of the ratings $r_{c',s}$ for every similar user c' inside users C [2].

$$u(c,s) = \underset{c'\in C}{\mathrm{aggr}} r_{c',s}$$

The aggregate function used for this method was a simple weighted, as shown below, where *k* is a normalizer,$\bar{r}_{c'}$is the average rating given by a user *c'*, and *sim* is a function that determines the similarity between two users. This method is known as the KNN algorithm or K-Nearest Neighbor algorithm as the value for the prediction is based on the nearest neighbors weighted by their closeness or, in this case, similarity [2].

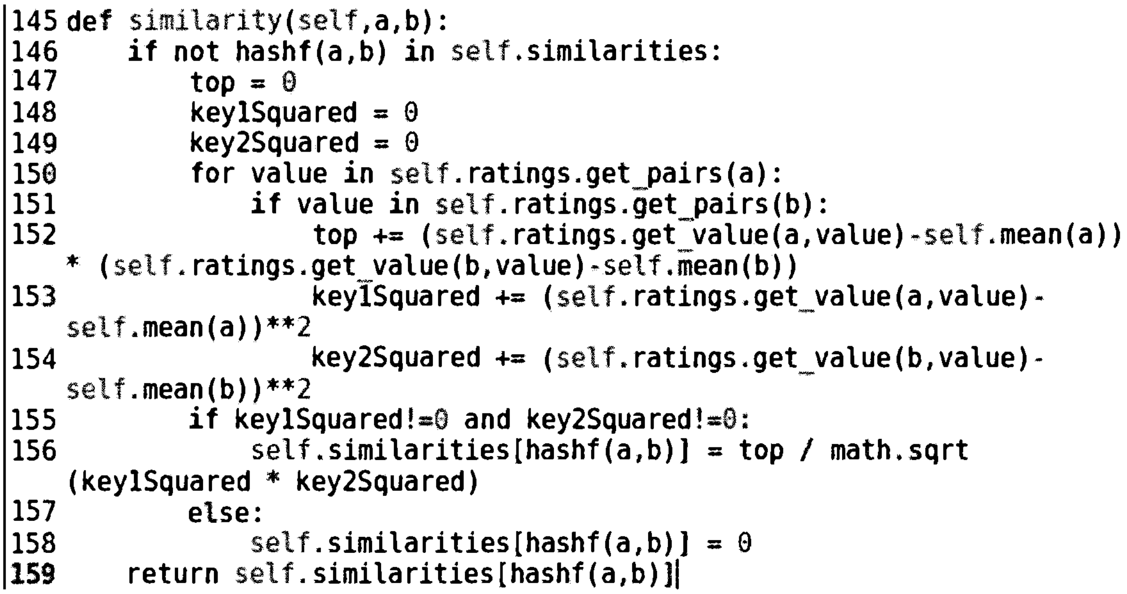
$$u(c,s) = \bar{r}_c + k \sum_{c' \in C} \operatorname{sim}(c,c')\left(r_{c',s} - \bar{r}_{c'}\right)$$

The similarity function is most commonly implemented by computing the Pearson correlation coefficient [2][3]. This coefficient is computed by the following formula, where $S_{xy}$is the set of items rated by both user *x* and user *y*.

$$\operatorname{sim}(x,y) = \frac{\sum_{s \in S_{xy}} \left(r_{x,s} - \bar{r}_x\right)\left(r_{y,s} - \bar{r}_y\right)}{\sqrt{\sum_{s \in S_{xy}} \left(r_{x,s} - \bar{r}_x\right)^2 \sum_{s \in S_{xy}} \left(r_{y,s} - \bar{r}_y\right)^2}}$$

?

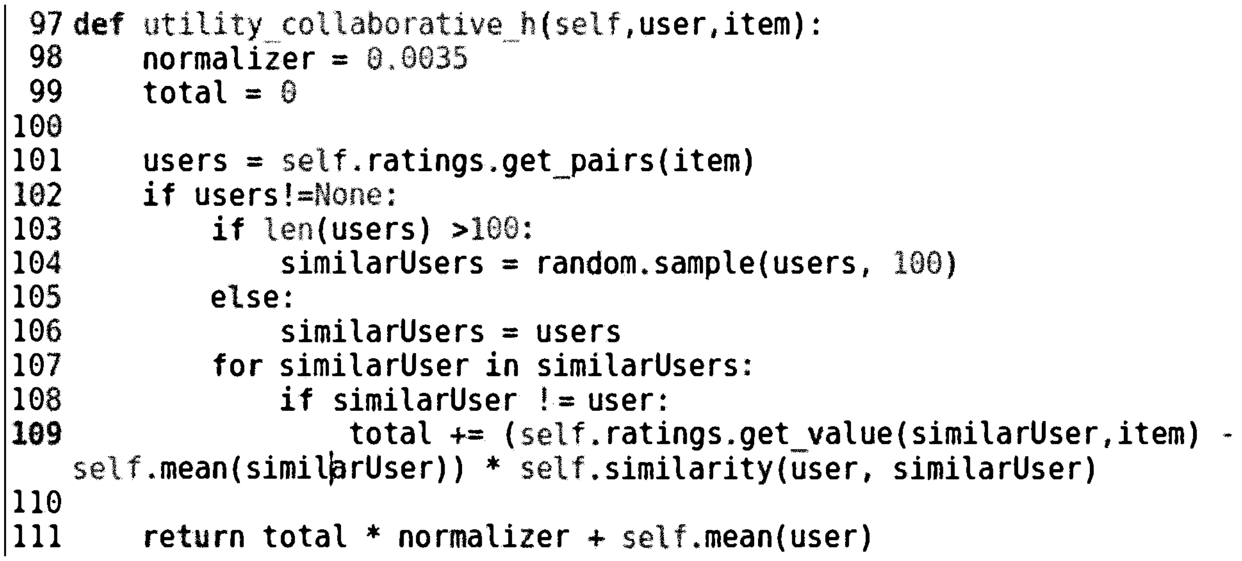
The cosine similarity between n-dimensional vectors can be also used to obtain the similarity as pointed out in [2], but it should be noted that the Pearson correlation coefficient is equivalent to the cosine similarity if the mean ratings were taken as the origin for each vector. Implementing this formula in python results with the following:



```
def similarity(self,a,b):
    if not hashf(a,b) in self.similarities:
        top = 0
        key1Squared = 0
        key2Squared = 0
        for value in self.ratings.get_pairs(a):
            if value in self.ratings.get_pairs(b):
                top += (self.ratings.get_value(a,value)-self.mean(a)) * (self.ratings.get_value(b,value)-self.mean(b))
                key1Squared += (self.ratings.get_value(a,value)-self.mean(a))**2
                key2Squared += (self.ratings.get_value(b,value)-self.mean(b))**2
        if key1Squared!=0 and key2Squared!=0:
            self.similarities[hashf(a,b)] = top / math.sqrt(key1Squared * key2Squared)
        else:
            self.similarities[hashf(a,b)] = 0
    return self.similarities[hashf(a,b)]
```

The resulting algorithm is known as the *Pearson Nearest Neighbor* algorithm [13]. This implementation has an order of O(m x n) per recommendation where m is the amount of ratings for user *c* and *n* is the amount of ratings in *C*. This can prove to be very expensive as the value of *n* is around 10 million. A solution for this problem is to reduce the size of *C* used in the utility function. This was done by taking a randomly distributed sample of 100 or less users from *C* instead. This sample size would be changed in order to obtain different results from the RMSE. In order to reduce discrepancies in the data due to the random factor, 5 runs of the program were performed per measure and the average was taken as the result. ✓

The following is the resulting implementation of the utility function:



```
def utility_collaborative_h(self,user,item):
    normalizer = 0.0035
    total = 0

    users = self.ratings.get_pairs(item)
    if users!=None:
        if len(users) >100:
            similarUsers = random.sample(users, 100)
        else:
            similarUsers = users
        for similarUser in similarUsers:
            if similarUser != user:
                total += (self.ratings.get_value(similarUser,item) -
  self.mean(similarUser)) * self.similarity(user, similarUser)

    return total * normalizer + self.mean(user)
```

3.2 Model-based Collaborative Filtering

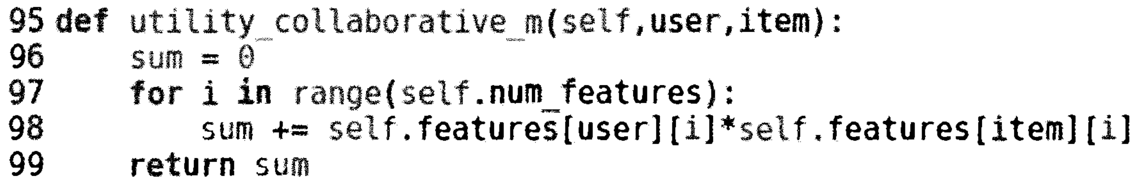
For the model-based collaborative approach, we start by having an error *E* defined as the error between the real ratings *r* and the ratings predicted by *u*. The problem is then an optimization one with the goal of minimizing *E* for the predictions given by *u* [7]. The error *E* can thus be expressed as some kind of aggregate of errors.

? The SVD algorithm is one of the most commonly used algorithms for recommender systems, this is due to both its accuracy and speed [7][13][16]. This algorithm works by defining *feature vectors* for each user and item. Feature vectors refer to the features an item might have. A movie might, for example, have a feature vector consisting of action, drama and comedy components expressed as follows <*amount_of_action, amount_of_comedy,drama*>. An action movie that has a little comedy and is not dramatic may have a feature vector of <4,-1,1>. Similarly, a user who likes action and drama would have a feature vector of <4,3,0>.

By combing these vectors in the utility function, one would get the rating a user would give to an item. One way to do this is by simply obtaining the dot product of the feature vectors [13]. The utility function would then be defined the following way where F_c represents the features of user c and F_s represents the features of item s:

$$u(c,s) = F_c \cdot F_s$$

In code, the utility function was implemented the following way:



```
def utility_collaborative_m(self,user,item):
    sum = 0
    for i in range(self.num_features):
        sum += self.features[user][i]*self.features[item][i]
    return sum
```

The order of the utility function for this algorithm is thus O(1) because the amount of features per vector is a constant.

As can already be guessed, the performance of this algorithm would be dependent on the performance of its *training*. One of the most common approaches to reducing time taken by SVD while keeping accuracy high is called *incremental SVD* [13][15]. This algorithm defines the error function for a user *c* as the sum of the errors of items rated by that user:

$$E_c = \sum_{s \in S_c} \left(r_{c,s} - u(c,s)\right)^2 + ||F_s||^2 + ||F_c||^2$$

The problem then becomes one of finding the feature vectors, instead of predicting ratings. To obtain these feature vectors, SVD algorithms factorize the ratings. If the ratings are viewed as a matrix, this becomes a *matrix factorization* problem [7][13]. As can be seen in the error function above, one is able to obtain the feature vectors *c* and *s*, by performing optimization on the function. The reason the squared error was chosen as the error function is that minimizing the squared error function is equivalent to minimizing the RMSE (the metric used to evaluate the accuracy).

This error function was implemented as follows:

```
    def squaredError(self,user,items):
        squaredError = 0
        for item in items:
            squaredError += (self.utility_collaborative_m
  (user,item) - self.ratings.get_value(user,item))**2 + sum
  (self.item_features[item]) + sum(self.user_features[user])
        return squaredError
```

why?

?

The training for the algorithm had an order of O(m^3), but due to the nature of the algorithm, there was no need for the training to ~~be~~ complete, viz. partial training would still yield accurate predictions. Instead, some time was given to the algorithm to 'learn' the features of these vectors. The more time given to the algorithm, the more accurate the predictions and the less RMSE.

3.3 Content-based model-based Filtering

Content-based filtering techniques have mostly been used for recommendations pertaining to services with an abundance of item information. This item information was not present in the dataset tested, but could be obtained from external sources. Despite this, no external sources for data were used. This was with the hope of keeping results comparable.

Content-based model-based filtering can be implemented through the use of the same SVD algorithm as the collaborative model-based implementation, replacing the error function with a content-based analog, Note that the error is now for all users C_s that have rated item s.

$$E_s = \sum_{c \in C_s} \left(r_{c,s} - u(c,s)\right)^2 + \|F_s\|^2 + \|F_c\|^2$$

The python implementation was kept polymorphic in order for it to be used for both techniques. The function would only have to be called differently.

3.4 Content-based memory-based Filtering

Finally a content-based memory-based algorithm was tested. This recommender system was implemented the same as the memory-based collaborative algorithm but using the similarity between two items for the weighted sum instead of the similarity between two users.

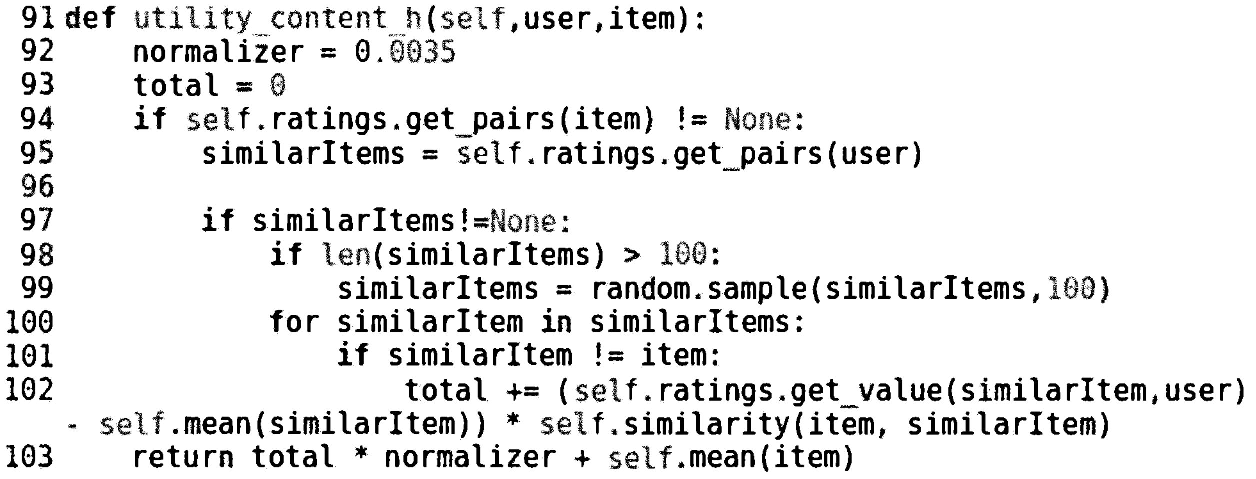
$$\text{sim}(x,y) = \frac{\sum_{c\in C_{xy}}(r_{c,x} - \bar{r_x})(r_{c,y} - \bar{r_y})}{\sqrt{\sum_{c\in C_{xy}}(r_{c,x} - \bar{r_x})^2 \sum_{c\in C_{xy}}(r_{c,y} - \bar{r_y})^2}}$$

The similarity between items uses the same implementation shown above for the similarity, which was also made polymorphic in order to be used for items and users alike. Again a sample of 100 was used when appropriate in order to reduce time taken.

The utility function was thus the same PNN algorithm, but was modified to focuss on items instead of users:

$$u(c,s) = \bar{r_s} + k \sum_{s\prime\in S} \text{sim}\,(s,s')(r_{c,s\prime} - \bar{r_{s\prime}})$$

The python implementation is shown below.



```
def utility_content_h(self,user,item):
    normalizer = 0.0035
    total = 0
    if self.ratings.get_pairs(item) != None:
        similarItems = self.ratings.get_pairs(user)

        if similarItems!=None:
            if len(similarItems) > 100:
                similarItems = random.sample(similarItems,100)
            for similarItem in similarItems:
                if similarItem != item:
                    total += (self.ratings.get_value(similarItem,user)
 - self.mean(similarItem)) * self.similarity(item, similarItem)
    return total * normalizer + self.mean(item)
```

4. Results

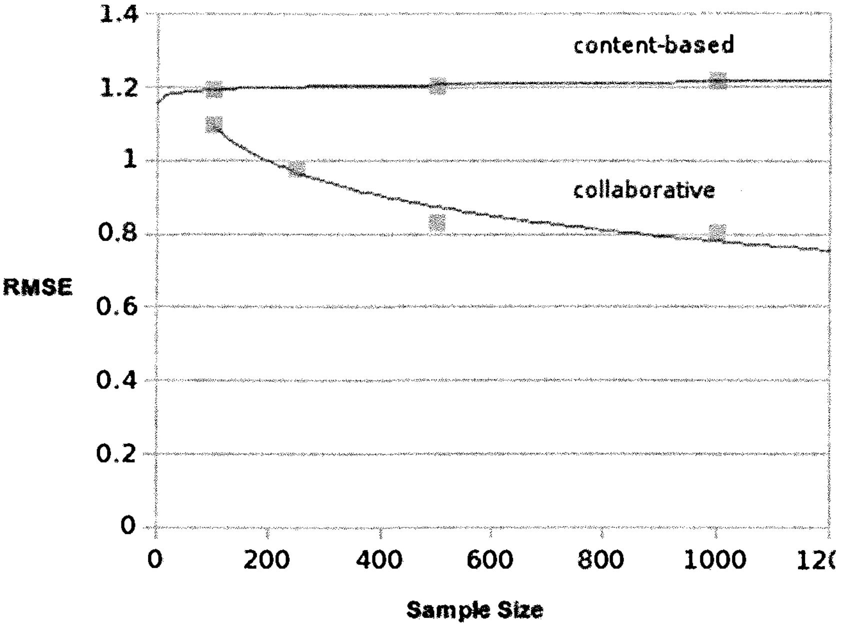
The results obtained by running the algorithms against the test probe provided in the Netflix dataset are shown below. The control algorithms are also provided. The following table shows the Order of each algorithm as well as the least RMSE measured.

	Minimum error (RMSE)	Order per recommendation	Training Order

Random recommendations	1.732051	O(1)	-
Average rating of user	1.000000	O(1)	-
Average rating of item	1.225819	O(1)	-
Memory-based Collaborative Filtering	1.094864	O(m x n)	-
Model-based Collaborative Filtering	0.915002	O(1)	$O(m^3)$
Memory-Based Content-based Filtering	1.192138	O(m x n)	-
Model-Based Content based-filtering	0.974348	O(1)	$O(m^3)$

In order to demonstrate the behavior of the RMSE in the memory-based algorithms implemented the sample size was increased. The following chart illustrates the behavior of the RMSE when groups were formed before applying the operation. The top part of the data refers to content-based algorithm, while the bottom part refers to collaborative predictions.

Illustration

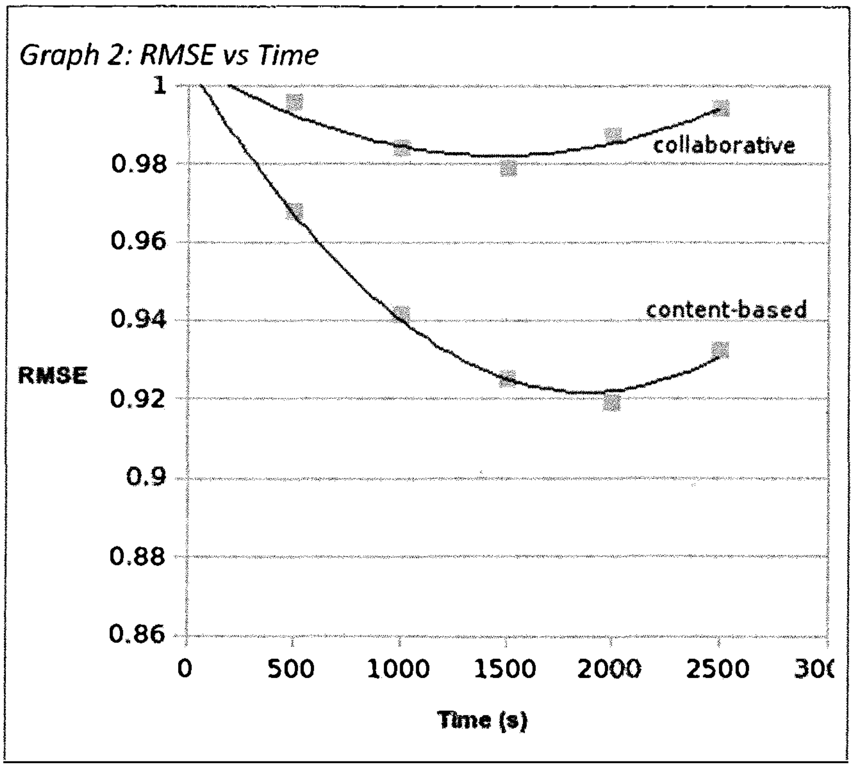


1: Graph 1: RMSE vs. Sample Size

On average, content-based predictions were 31% less accurate than their counterparts.

Memory-based filtering accuracy seems to perform better when more data is available to the algorithm. This is because the bigger the sample size, the more similar users available to the algorithm. These algorithms, although proved to be accurate, also proved to not scale. The time taken for a complete run with 100 as the sample size was, in both cases, around 2600s. The time taken is proportional to the number of samples chosen. With a sample size of 1000, the time taken was around 26,000s.

Further tests were conducted to determine the behavior of the RMSE in the model-based algorithms. The training time was used as the independent variable. The results are as shown in the following chart:



Again, the content-based technique was the less accurate one. On average, the content-based algorithm was 4.3% less accurate than the collaborative algorithm in this case.

Even though content-based filtering performed worse than collaborative filtering, its performance is surprising. These algorithms thrive in situations where the content features are provided explicitly to them. But due to the nature of the dataset used, these features had to be measured indirectly. Even in an environment where this data was not present, their performance was not far behind from their collaborative counterparts.

Following these results it is clear why memory-based filtering use is reduced for larger databases. These algorithms are able to provide an accurate prediction, but at a high time cost. As services continue to increase in size, these algorithms will not able to keep up. Memory-based algorithms are more suited to smaller databases where they are able to predict with accuracy user preferences.

5. Conclusion

With the need of a fast, yet accurate recommender system, the best algorithm is one that manages to find balance between these qualities, which seem to oppose each other. The results obtained through the tests performed give a clear answer as to why the best algorithms are the model-based algorithms. But it must be noted that the environment at which this algorithms were able to perform well was a specific one that may not fit every need. As the nature of the Netflix dataset was a collaborative one, collaborative-algorithms were able to outperform the others.

Having this limitation in mind, it is important to weight the benefits of each algorithm for a given context: Based on the results, content-based algorithms are the ones that should be used for databases that have additional metadata concerning their items, as they have a more accurate similarity function on which to operate. Collaborative algorithms should be used when the nature of the database permits a high number of ratings from multiple users as the similarity function will also be more accurate, model-based algorithms should be used for large databases where recommendation speed is required and accuracy is a major concern, and finally memory-based algorithms should be best employed on smaller datasets with a high degree of similar users.

Further improvements on recommender systems which incorporate the knowledge of the nature of these techniques to create hybrid systems should be looked into. Identifying the right areas on where to use these tools is the first step on this direction. That is achieving true personalization for a better utilization of the information available to the users.

4 -> 1

The candidate tended to follow his sources too closely in the early part of the essay, and not all terms have been adequately explained. However, overall this is a good essay tackling a topic which is quite original showing a good understanding through his analysis

References

1. Resnick, P., & Varian, H. R. (1997). Recommender Systems. *Communications Of The ACM, 40*(3), 56-58. doi:10.1145/245108.245121

2. Tuzhilin, Alexander, and Gediminas Adomavicius. "Towards the Next Generation of Recommender Systems."*IEEE Transactions on Knowledge and Data Engineering* 17 (2005): 734. Print.

3. Anand, Sarabjot Singh, and Bamshad Mobasher. "Intelligent techniques for web personalization." *Proceedings of the 2003 international conference on Intelligent Techniques for Web Personalization*. Springer-Verlag, 2003.

4. "Recommender Systems Introduction."*Stanford Infolab*. N.p., n.d. Web. 6 Mar. 2014. <http://infolab.stanford.edu/~ullman/mmds/ch9>

5. Resnick, P., & Varian, H. R. (1997). Recommender Systems. *Communications Of The ACM, 40*(3), 56-58. doi:10.1145/245108.245121

6. Ricci, F., Rokach, L., & Shapira, B. (2011). *Introduction to recommender systems handbook* (pp. 1-35). Springer US.

7. Ma, C. C. (2008). Large-scale collaborative filtering algorithms. *Master's thesis, National Taiwan University.*

8. Van Meteren, R., & Van Someren, M. (2000, May). Using content-based filtering for recommendation. In *Proceedings of the Machine Learning in the New Information Age: MLnet/ECML2000 Workshop.*

9. Breese, John S., David Heckerman, and Carl Kadie. "Empirical analysis of predictive algorithms for collaborative filtering." *Proceedings of the Fourteenth conference on Uncertainty in artificial intelligence.* Morgan Kaufmann Publishers Inc., 1998.

10. Goldberg, D., Nichols, D., Oki, B. M., & Terry, D. (1992). Using collaborative filtering to weave an information tapestry. *Communications of the ACM, 35*(12), 61-70.

11.M. Balabanovic and Y. Shoham, "Fab: Content-Based, Collaborative Recommendation," Comm. ACM, vol. 40, no. 3, pp. 66-72, 1997.

12. Mehta, Bhaskar, Thomas Hofmann, and Wolfgang Nejdl. "Robust collaborative filtering." *Proceedings of the 2007 ACM conference on Recommender systems.* ACM, 2007

13. Sarwar, B., Karypis, G., Konstan, J., & Riedl, J. (2000). *Application of dimensionality reduction in recommender system-a case study* (No. TR-00-043). Minnesota Univ Minneapolis Dept of Computer Science.

14. Kleinberg, J., & Sandler, M. (2004, June). Using mixture models for collaborative filtering. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing* (pp. 569-578). ACM.

15. Su, X., & Khoshgoftaar, T. M. (2009). A survey of collaborative filtering techniques. *Advances in artificial intelligence, 2009*, 4.

16. Pryor, M. H. (1998). The effects of singular value decomposition on collaborative filtering.

Appendix I

Netflix Price README file

SUMMARY
==
========

This dataset was constructed to support participants in the Netflix Prize. See http://www.netflixprize.com for details about the prize.

The movie rating files contain over 100 million ratings from 480 thousand randomly-chosen, anonymous Netflix customers over 17 thousand movie titles. The data were collected between October, 1998 and December, 2005 and reflect the distribution of all ratings received during this period. The ratings are on a scale from 1 to 5 (integral) stars. To protect customer privacy, each customer id has been replaced with a randomly-assigned id. The date of each rating and the title and year of release for each movie id are also provided.

USAGE LICENSE
==
========

Netflix can not guarantee the correctness of the data, its suitability for any particular purpose, or the validity of results based on the use of the data set. The data set may be used for any research purposes under the following conditions:

* The user may not state or imply any endorsement from Netflix.

* The user must acknowledge the use of the data set in publications resulting from the use of the data set.

* The user may not redistribute the data without separate permission.

* The user may not use this information for any commercial or revenue-bearing purposes without first obtaining permission from Netflix.

If you have any further questions or comments, please contact the Prize administrator <prizemaster@netflix.com>

TRAINING DATASET FILE DESCRIPTION
==
========

The file "training_set.tar" is a tar of a directory containing 17770 files, one per movie. The first line of each file contains the movie id followed by a colon. Each subsequent line in the file corresponds to a rating from a customer and its date in the following format:

CustomerID,Rating,Date

- MovieIDs range from 1 to 17770 sequentially.

- CustomerIDs range from 1 to 2649429, with gaps. There are 480189 users.
- Ratings are on a five star (integral) scale from 1 to 5.
- Dates have the format YYYY-MM-DD.

MOVIES FILE DESCRIPTION
==
========

Movie information in "movie_titles.txt" is in the following format:

MovieID,YearOfRelease,Title

- MovieID do not correspond to actual Netflix movie ids or IMDB movie ids.
- YearOfRelease can range from 1890 to 2005 and may correspond to the release of corresponding DVD, not necessarily its theaterical release.
- Title is the Netflix movie title and may not correspond to titles used on other sites. Titles are in English.

QUALIFYING AND PREDICTION DATASET FILE DESCRIPTION
==
========

The qualifying dataset for the Netflix Prize is contained in the text file "qualifying.txt". It consists of lines indicating a movie id, followed by a colon, and then customer ids and rating dates, one per line for that movie id. The movie and customer ids are contained in the training set. Of course the ratings are withheld. There are no empty lines in the file.

MovieID1:
CustomerID11,Date11
CustomerID12,Date12
...
MovieID2:
CustomerID21,Date21
CustomerID22,Date22

For the Netflix Prize, your program must predict the all ratings the customers gave the movies in the qualifying dataset based on the information in the training dataset.

The format of your submitted prediction file follows the movie and customer id, date order of the qualifying dataset. However, your predicted rating takes the place of the corresponding customer id (and date), one per line.

For example, if the qualifying dataset looked like:

111:
3245,2005-12-19
5666,2005-12-23
6789,2005-03-14
225:
1234,2005-05-26
3456,2005-11-07

then a prediction file should look something like:

111:
3.0
3.4
4.0
225:
1.0
2.0

which predicts that customer 3245 would have rated movie 111 3.0 stars on the 19th of Decemeber, 2005, that customer 5666 would have rated it slightly higher at 3.4 stars on the 23rd of Decemeber, 2005, etc.

You must make predictions for all customers for all movies in the qualifying dataset.

THE PROBE DATASET FILE DESCRIPTION

==
========

To allow you to test your system before you submit a prediction set based on the qualifying dataset, we have provided a probe dataset in the file "probe.txt". This text file contains lines indicating a movie id, followed by a colon, and then customer ids, one per line for that movie id.

MovieID1:
CustomerID11
CustomerID12
...
MovieID2:
CustomerID21
CustomerID22

Like the qualifying dataset, the movie and customer id pairs are contained in the training set. However, unlike the qualifying dataset, the ratings (and dates) for each pair are contained in the training dataset.

If you wish, you may calculate the RMSE of your predictions against those ratings and compare your RMSE against the Cinematch RMSE on the same data. See http://www.netflixprize.com/faq#probe for that value.

Good luck!

MD5 SIGNATURES AND FILE SIZES

==
========

```
d2b86d3d9ba8b491d62a85c9cf6aea39      577547 movie_titles.txt
ed843ae92adbc70db64edbf825024514    10782692 probe.txt
88be8340ad7b3c31dfd7b6f87e7b9022    52452386 qualifying.txt
0e13d39f97b93e2534104afc3408c68c          567 rmse.pl
0098ee8997ffda361a59bc0dd1bdad8b  2081556480 training_set.tar
```

Appendix II

Memory-based filtering implementation

```
import math
import random

def User(a):
    return int(a)*2


def Item(a):
    return int(a)*2+1

#Simple hashing function where the order of inputs doesn't matter
def hashf(a,b):
    return a*b

#Helper class to store ratings
class DoubleKeyDict(object):
    def __init__(self):
        self._key1_dict = {}
        self._key2_dict = {}
        self._values = {}
    #Establish relation between user a and item b of rating c
    def set(self,a,b,c):
        if not a in self._key1_dict:
            self._key1_dict[a] = []
        self._key1_dict[a].append(b)
        if not b in self._key2_dict:
            self._key2_dict[b] = []
        self._key2_dict[b].append(a)

        #Add rating 'c' to user 'a' and item 'b' inside dict 'ratings'
        self._values[hashf(a,b)] = c

    def get_value(self,a,b):
        return self._values[hashf(a,b)]

    #Return items rated by user a or users that rated item a
    def get_pairs(self, a):
        if a in self._key1_dict:
            return self._key1_dict[a]
        elif a in self._key2_dict:
            return self._key2_dict[a]
        else:
            print("ERROR: Key not found")

    #Return user list
    def get_first_keys(self):
        return self._key1_dict

    #Return item list
    def get_second_keys(self):
        return self._key2_dict

#Main class
```

```
class Group(object):
    def __init__(self):
        self.ratings = DoubleKeyDict()
        self.leader = 0
        self.means = {}
        self.similarities = {}

    #Add rating for user and item to self.ratings
    def add(self,user, item, rating):
        self.ratings.set(user,item,rating)

    def utility_random(self, user, item):
        return random.randint(1,5)

    def utility_mean_user(self, user, item):
        return self.mean(user)

    def utility_mean_item(self, user, item):
        return self.mean(item)


    def utility_collaborative_h(self,user,item):
        normalizer = 0.0035
        total = 0

        similarUsers = self.ratings.get_pairs(item)
        if similarUsers!=None:
            if len(similarUsers) >100:
                similarUsers = random.sample(similarUsers, 100)
            for similarUser in similarUsers:
                if similarUser != user:

                    total += (self.ratings.get_value(similarUser,item) - self.mean(similarUser)) *
self.similarity(user, similarUser)

        return total * normalizer + self.mean(user)

    def utility_content_h(self,user,item):
        normalizer = 0.0035
        total = 0
        if self.ratings.get_pairs(item) != None:
            similarItems = self.ratings.get_pairs(user)

            if similarItems!=None:
                if len(similarItems) > 100:
                    similarItems = random.sample(similarItems,100)
                for similarItem in similarItems:
                    if similarItem != item:
                        total += (self.ratings.get_value(similarItem,user) - self.mean(similarItem)) *
self.similarity(item, similarItem)
        return total * normalizer + self.mean(item)

    def similarity(self,a,b):
        if not hashf(a,b) in self.similarities:
            top = 0
            key1Squared = 0
            key2Squared = 0
            for value in self.ratings.get_pairs(a):
                if value in self.ratings.get_pairs(b):
                    top += (self.ratings.get_value(a,value)-self.mean(a)) *
(self.ratings.get_value(b,value)-self.mean(b))
                    key1Squared += (self.ratings.get_value(a,value)-self.mean(a))**2
                    key2Squared += (self.ratings.get_value(b,value)-self.mean(b))**2
            if key1Squared!=0 and key2Squared!=0:
                self.similarities[hashf(a,b)] = top / math.sqrt(key1Squared * key2Squared)
            else:
                self.similarities[hashf(a,b)] = 0
        return self.similarities[hashf(a,b)]

    def mean(self, a):
        if not a in self.means:
            total = 0
            count = 0
            if self.ratings.get_pairs(a) != None:
                for pair in self.ratings.get_pairs(a):
                    total += self.ratings.get_value(a, pair)
                    count += 1
                self.means[a]=total/count
            else:
                #print "ERROR: No ratings present for given data"
                self.means[a]=2.5
        return self.means[a]

main = Group()

def load(d):
    filenames = os.listdir(d)
    total = float(len(filenames))
    current = 0
    for filename in filenames:
        f = open(d+filename)
        movieId = f.readline()[:-2]
        if os.path.isfile('status'):
            print "Loading... [%i%%]" %(current/total*100)
        for line in f:
            (user, rating, date) = line.split(",")
            main.add(User(user), Item(movieId),int(rating))
        current += 1
        f.close()
        if current>10000 or os.path.isfile('continue'):
            break

def results(d):
    f = open(d)
    results = open("results.txt", "w")
    current = 1
    total = float(sum(1 for line in f))
    movieId = 0
    f.seek(4)
    rMSE = 0
    squaredError = 0
    batch_output = ""
    total_valued = 1
    for line in f:
        print "Predicting user ratings...[%f%%] with error of:%f" %((current/total*100),rMSE)

        if line[-2:-1] == ":":
            movieId = int(line[:-2])
            batch_output += line+""
        else:
            result = main.utility_content_h(User(line), Item(movieId))

            try:
                squaredError += (result-main.ratings.get_value(User(line), Item(movieId)))**2
                rMSE = math.sqrt(squaredError/total_valued)
                total_valued += 1
            except:
                pass

            batch_output += "%f\n" %result

        if current % 1000 == 0:
            results.write(batch_output)
            batch_output = ""
        current+=1
    print "Predicting user ratings...[%f%%] with error of:%f" %((current/total*100),rMSE)
    f.close()

load("NF_Dataset/training_set/")
results("NF_Dataset/probe.txt")
```

Appendix III

Model-based filtering implementation

```
#Incremental SVD algorithm

import math
import random

def User(a):
    return int(a)*2

def Item(a):
    return int(a)*2+1

#Simple hashing function where the order of inputs doesn't matter
def hashf(a,b):
    return a*b

#Helper class to store ratings
class DoubleKeyDict(object):
    def __init__(self):
        self._key1_dict = {}
        self._key2_dict = {}
        self._values = {}
    #Establish relation between user a and item b of rating c
    def set(self,a,b,c):
        if not a in self._key1_dict:
            self._key1_dict[a] = []
        self._key1_dict[a].append(b)
        if not b in self._key2_dict:
            self._key2_dict[b] = []
        self._key2_dict[b].append(a)

        #Add rating 'c' to user 'a' and item 'b' inside dict 'ratings'
        self._values[hashf(a,b)] = c

    def get_value(self,a,b):
        return self._values[hashf(a,b)]

    #Return items rated by user a or users that rated item a
    def get_pairs(self, a):
        if a in self._key1_dict:
            return self._key1_dict[a]
        elif a in self._key2_dict:
            return self._key2_dict[a]
        else:
            print("ERROR: Key not found")
            return []
    #Return user list
    def get_first_keys(self):
        return self._key1_dict
    #Return item list
    def get_second_keys(self):
        return self._key2_dict
```

```
#Minimizing function.
def minimize(errorfunc, parameters, feature, array):
    rate = 0.002
    right = errorfunc(*parameters)
    array[feature] -= rate
    left = errorfunc(*parameters)
    slope = (right-left)/rate

    if slope>0:
        while (slope>0.0005):
            right = left
            array[feature] -= slope*rate
            left = errorfunc(*parameters)
            slope = (right-left)/slope
    elif slope<0:
        array[feature] += rate
        while (slope<-0.0005):
            left = right
            array[feature] -= slope*rate
            right = errorfunc(*parameters)
            slope = (left-right)/slope

#Main class
class Group(object):
    def __init__(self):
        self.ratings = DoubleKeyDict()
        self.user_features = {}
        self.item_features = {}
        self.num_features = 7
        self.predictions = {}

    #Add rating for user and item to self.ratings
    def add(self,user, item, rating):
        self.ratings.set(user,item,rating)

        #Assign a 7 dimension feature vector to each user and item.
        #0.598 is a nice default and was calculated by sqrt(meanScore/numberOfFeatures).
        #No check was done as it is faster this way and doesn't affect the end result
        self.user_features[user] = [0.598 for x in range(self.num_features)]
        self.item_features[item] = [0.598 for x in range(self.num_features)]

    #Simple dot product of the feature vectors
    def utility_model(self,user,item):
        sum = 0
        for i in range(self.num_features):
            sum += self.user_features[user][i]*self.item_features[item][i]
        return sum

    #Squared Error function
    def squaredError(self,user,items):
        squaredError = 0
        for item in items:
            squaredError += (self.utility_collaborative_m(user,item) -
self.ratings.get_value(user,item))**2 + sum(self.item_features[item]) +
sum(self.user_features[user])
        return squaredError
```

```
main = Group()

#Load database
def load(d):
    filenames = os.listdir(d)
    total = float(len(filenames))
    current = 0
    for filename in filenames:
        f = open(d+filename)
        movieId = f.readline()[:-2]
        if os.path.isfile('status'):
            print "Loading... [%i%%]" %(current/total*100)
        for line in f:
            (user, rating, date) = line.split(",")
            main.add(User(user), Item(movieId),int(rating))
        current += 1
        f.close()
        if current>10000 or os.path.isfile('continue'):
            break

#Load data to be predicted
def load_predictions(d):
    f = open(d)
    current = 1
    total = float(sum(1 for line in f))
    movieId = 0
    f.seek(0)
    for line in f:
        if os.path.isfile('continue2'):
            print "Determining values to predict [%i%%]" %(current/total*100)
            break
        if line[-2:-1] == ":":
            movieId = int(line[:-2])
        else:
            main.predictions[User(line)]=Item(movieId)
        current += 1
    f.close()

#Calculate the feature vectors for every user and item
def calculate_features():
      total = float(len(main.user_features))
      current = 0
    for user in main.user_features:
        print "Calculating features... [%f%%]" %(current/total*100)
        if user in main.predictions:
            items = main.ratings.get_pairs(user)
            for i in range(len(main.user_features[user])):
                minimize(main.squaredError, (user,items), i, main.user_features[user])
            for item in items:
                for i in range(len(main.item_features[item])):
                    minimize(main.squaredError, (user,items), i, main.item_features[item])
        current+=1

print "Loading ratings..."
load("NF_Dataset/training_set/")
```

```
print "Loading predictions..."
load_predictions("NF_Dataset/probe.txt")
print "Calculating Features..."
calculate_features();

squaredError = 0
total = float(len(main.predictions))
current = 0
print total
for user,item in main.predictions.items():
    print "Loading... [%i%%] rmse of %f"
%(current/total*100,math.sqrt(squaredError/len(main.predictions)))
    if item in main.ratings.get_pairs(user):
        try:
            squaredError += (main.ratings.get_value(user,item) -
main.utility_collaborative_m(user,item))**2
        except:
               pass
    current+=1
print "Loading... [%i%%] rmse of %f"
%(current/total*100,math.sqrt(squaredError/len(main.predictions)))
print math.sqrt(squaredError/len(main.predictions))
```