Recommendation system implementation based on clustering, text similarity and collaborative filtering

Alan Ramponi
University of Trento
Via Sommarive 9, 38123 - Povo (TN), Italy
alan.ramponi@studenti.unitn.it

Davide Martintoni
University of Trento
Via Sommarive 9, 38123 - Povo (TN), Italy
davide.martintoni@studenti.unitn.it

ABSTRACT

In this report we present our projects for both Data Mining and Big Data courses. Using a massive dataset of Amazon reviews we implemented three types of algorithms in order to propose suggestions in a e-commerce website.

We developed a recommendation system with a collaborative filtering using five different distance measures in order to find the best one. We have also tried to recommend items using clustering techniques on the relationship between the items, so we implemented three different types of clustering algorithms in order to find which one works better. As last attempt we tried to work on the text of the reviews representing them as TFIDF vectors and compare them in order to get suggestion based on similar opinions.

Once we have implemented all these algorithms we tested them and we evaluated their performance and correctness comparing each variant of each method. At the end, based on that Amazon data, we have also figured out which of these techniques fits better for our purpose.

Obviously Amazon contains a huge amount of reviews and ratings on its system and if we are going to analyze all this information we need to distribute the computation. In fact, it is impossible that a single computer process all these data in a reasonable time. Thus we implemented, using the Hadoop framework, a MapReduce algorithm that computes the TFIDF of the textual reviews distributing the workload among a cluster of computers.

After that, we compared the parallel reduce implementation with the serial one in order to discover if the real speed-up is actually high as expected.

Categories and Subject Descriptors

H.2.8 [**Database management**]: Database applications—*data mining*; H.3 [**Information storage and retrieval**]: Content analysis and indexing, Information search and retrieval, Systems—*information filtering, clustering, linguistic processing, distributed systems*; I.5.3 [**Pattern recognition**]: Clustering—*algorithms, similarity measures*; I.2.7 [**Artificial intelligence**]: Natural language processing—*language parsing and understanding, text analysis*; D.1.3 [**Programming techniques**]: Concurrent programming—*distributed programming, parallel programming*.

Keywords

Data mining, Big data, Information retrieval, Distributed systems, Distributed algorithms.

1. INTRODUCTION

Nowadays every e-commerce website uses a recommendation system to suggest to the people what they should buy. This problem can be approached in a lot of way: for example, a possible solution can be simply suggest random item hoping that the user who see that advertise will buy that particular object. However we can be smarter than that, thus we can think to something more sophisticated.

Based on a dataset of Amazon items that contains data about every object (e.g. price, image url, brand name) and data about what users think about those items (e.g. numerical rating, textual reviews and other items seen in the same user session), we built some algorithms to produce high quality recommendations for millions of customers and products. All these information are really meaningful and by exploiting the dataset we can provide an improved recommendation system that can lead to bigger outcome for the reseller. The only problem is that we have to choose which information we need from the aforementioned dataset and how to use them to suggest appropriate items to our customers.

Considering that we couldn't state a priori which of the many techniques better fits our problem, we have implemented three of them and subsequently we have compared every result. We have written a recommendation system based on collaborative filtering using the rating assigned to the objects by users. We have also tried to suggest new items based on the correlation between objects (e.g. which are bought together, which are seen in the same session and which are bought in a previous order) using such relations with a clustering algorithm to create "bags" of similar objects. The last approach implemented consists in suggesting items to the target user based on the text reviews written by other users. After a preliminary stage of dimensionality reduction (tokenization and stemming) we compared tex-

Project report, Department of Information Engineering and Computer Science, University of Trento, Italy. Courses: Data mining, Big data and social networks. Academic Year: 2015/16. Authors: Davide Martintoni (171076) and Alan Ramponi (151369).

tual reviews using the TFIDF technique in order to detect similar items to recommend.

Despite the algorithms implemented work with remarkable accuracy, by using very large datasets they were still too slow to give recommendations within a reasonable time. For that reason we have implemented also a parallel version of the text based recommendation algorithm (the slowest one) in a MapReduce framework, using the power of distributed computing. According to Amdahl's law, in order to obtain the maximum portion of parallel work we have divided that algorithm in five MapReduce instances. Moreover, we have compared the parallel version with the serial one in terms of time, for various data sizes and types.

2. MOTIVATING EXAMPLE

A user bought a new umbrella on our e-commerce, so we insert this information into our recommendation system that instantly compare the user with other users in the system, and based on that and on the item that he just bought we can suggest to him for example some rain boots (maybe with the same colour of the umbrella), a raincoat or some items that the user probably needs. These suggestions are better fitting this user than random advertising, so this system will increase the outcome of our e-commerce web site using only the information that it already knows.

3. PROBLEM STATEMENT

Now the problem is quite big and there are a lot of possible solutions. We need to find some way to exploit all the data that we can gather from the web site (in our case a dataset kindly granted to us by the University of California, San Diego as an open source resource for educational purposes [17]). The problem is that all these information needs to be parsed and interpreted to be useful for any purpose. So with our algorithms we tried to exploit the knowledge contained in these data and we use it to improve our business leading users to buy more items with a properly designed recommendation system.

4. SOLUTION

We have designed three different ways to use these information. Each technique uses a different part of the available data and a different method to suggest to the final user something to buy that he should be interested in. So we used information that are actually available on every e-commerce website to improve the service by suggesting products to users based on ratings that they and other similar users have given. Moreover we can use textual reviews and information related to the items that a user is looking to suggest correlated products that probably can be useful with the target object.

So we have implemented and compared some techniques in order to find the best method to recommend the right products to users. We hope that this system will improve the business of our hypothetical e-commerce website leading the users to buy more items.

4.1 Collaborative filtering

The first implemented technique falls under the collaborative filtering (CF) family. CF techniques are usually divided into two big categories: memory-based and model-based. The main difference between the two approaches is that memory-based methods uses all the data to generate a prediction, whereas model-based methods use the data to learn or train a model which can be used in a second time to make a prediction. Our user-based algorithm falls into the first mentioned category.

In particular, our technique is a user-based collaborative filtering based on ratings given by users to products they like or dislike. This approach allows to make powerful predictions about the interest of a particular user by collecting tastes and preferences from other users in the system, and then by suggest to the target user the items that probably it should like, taken by the top similar people. The Amazon product dataset fits perfectly this purpose: in fact it provides a complete review data, including text reviews, summary and, among the other things, numerical ratings given by users to items (that vary in the range of 1.0 - 5.0, with half votes allowed). With that information we have mined knowledge from data identifying users that are similar on tastes to the target one and thus we have given him a list of recommendations.

4.1.1 Neighbors and distance measures

Although the basic idea is very simple, the problem to find a set of k nearest neighbors (k-NN) in a recommender system is very challenging due to many things. First of all, to compute a k-NN algorithm an appropriate distance metric has to be defined to measure the similarity between two patterns (see Figure 1 for a graphical representation of a user neighborhood formation).

	i_1	i_2	i_3	i_4	i_5	i_6	i_7	i_8
u_1	?	4.0	4.0	2.0	1.0	2.0	?	?
u_2	3.0	?	?	?	5.0	1.0	?	?
u_3	3.0	?	?	3.0	2.0	2.0	?	3.0
u_4	4.0	?	?	2.0	1.0	1.0	2.0	4.0
u_5	1.0	1.0	?	?	?	?	?	1.0
u_6	?	1.0	?	?	1.0	1.0	?	1.0
u_a	?	?	4.0	3.0	?	1.0	?	5.0
r_a	3.5	4.0			1.3		2.0	

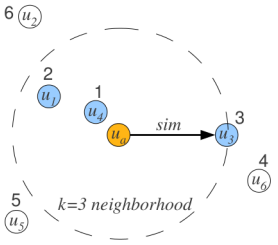


Figure 1: User-based collaborative filtering example with a rating matrix and estimated ratings for the active user (on the left) and a user neighborhood formation (on the right).

Simple distances like both the euclidean and the manhattan distance work well if the data is dense and the magnitude of the attribute values is important, but they don't suite our e-commerce website, where users clearly don't rate all the items they find in the store.

Another problem that arises in our case is that different users have very different behaviors when it comes to rating items. A way to fix this problem is to use the Pearson correlation coefficient that measures the correlation between two users, so it outputs a value that determines how well they are related in spite of "grade inflation" problem.

Pearson correlation coefficient has an outstanding drawback too. Such problem is the fact that it takes into account every 0-0 users rating pairs (that indicates an object that is not

rated yet) that, according to our goals, leads to an inaccurate recommendation. As we want to recommend items with high accuracy, we don't want to use these "shared zeros" when we are computing similarity.

A similarity measure that avoids 0-0 matches is represented by the cosine similarity, however computing similarity using basic cosine measure reintroduces the "grade inflation" problem.

The smart way we used to address the issue is to use a cosine similarity variant that offsets also this last drawback by subtracting the corresponding user average from each co-rated pair. Formally, the resulting similarity between items i and j for a user u is given by the following formula:

$$sim(i,j) = \frac{\sum_{u \in U}(R_{u,i} - \overline{R}_u)(R_{u,j} - \overline{R}_u)}{\sqrt{\sum_{u \in U}(R_{u,i} - \overline{R}_u)^2}\sqrt{\sum_{u \in U}(R_{u,j} - \overline{R}_u)^2}}$$

A deeper analysis of the measures implemented and their performance comparisons are discussed in the evaluation section (7.2.1).

4.1.2 Implementation

In details, the algorithm first collects in a nested dictionary all the ratings given to items by every user (Algorithm 1, lines 1-2). Secondly, it searches the k nearest neighbors (where k is given by input) among all the other users (Algorithm 1, line 4) using one among the five distance measures we have implemented (i.e. adjusted cosine similarity, cosine similarity, pearson correlation coefficient, manhattan distance, euclidean distance and then the generic minkowski distance of the last two mentioned). A performance comparison of them is discussed in 7.2.1. In the third place, the sorted list of users obtained in the previous step is used to efficiently build the recommendation system by assigning a weight to each neighbor; that weight is relative to the total distance (i.e. the sum of the k-NN distances) and it is designed to give more weight to more significant items (Algorithm 1, lines 6-8). Finally, items from the k neighbors that the target user didn't rate are weighted according to its own reviewer and they are recommended to the user in a decreasing order, by proximity (Algorithm 1, lines 10-17).

Algorithm 1 User-based collaborative filtering

Require: k, $n > 0$, a valid *user*
Ensure: max n recommendations based on k-NN to *user*

```
1:  for all reviews ∈ dataset do
2:      data ← (item,rating) pairs for each user

3:  function RECOMMEND(user)
4:      nearest ← NEARESTNEIGHBORS(user, k, measure)
5:      userratings ← (item,rating) pairs of user
6:      totdistance ← sum of k-NN distances

7:      for i ← 0 to k do
8:          weight ← nearest_i / totdistance
9:          neighratings ← (item,rating) of neighbor
10:         for all items ∈ neighratings do
11:             if item_i ∉ userratings then
12:                 current ← neighratings_i * weight
13:                 if item_i ∉ suggestions then
14:                     suggestions_i ← current
15:                 else
16:                     suggestions_i ← suggestions_i + current

17:     return top n suggestions
```

4.2 Hierarchical clustering

The second implemented technique is a bottom-up clustering method based on relation between items. The dataset contains a metadata table that for each item provides, among other information, a list of other items that are *also_bought* (at least one user has bought these items in different sessions), *also_seen* (at least one user has seen these items in the same session) and *bought_together* (at least one user has bought these two items in the very same session).

As we can see in Figure 2, for every relation type we have heuristically assigned a weight. These values are based on our own experience, in fact after running our algorithms we have seen better outcomes with these values so after a few tests we have decided to use them in this way.

Now with all these data we need to decide how to cluster them. In order to better understand which method fits better the data we have tried three different bottom-up clustering algorithms. So with this three clustering methods we have tried to use the metadata that we found in our dataset to provide item-based suggestions by exploiting the relationship between distinct items.

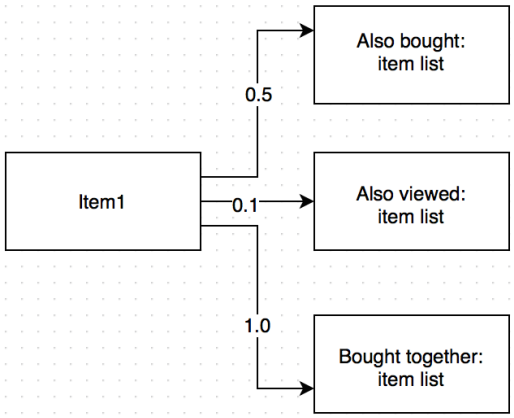


Figure 2: Relationships listed in the raw dataset with the weight that we have heuristically given to them.

So let's take a closer look on the used clustering techniques.

4.2.1 Weight-based clustering

The first approach that we have implemented is a bottom-up clustering algorithm that uses as distance measure the weight of the "relationship" between clusters. So first of all we built a distance matrix in which in the entry (item1, item2) we placed the sum of every entry in our dataset structure that we can see in Figure 2.

We decided that we don't care about the direction of this relationship, thus the entry (i_1, i_2) of the matrix should be the same of the entry (i_2, i_1). For this purpose at line 3 of the Algorithm 2 we sum the distance matrix with its transposition. After that we can easily delete all the useless values above the principle diagonal and the diagonal itself.

Algorithm 2 Weight-based clustering algorithm

Require: a *threshold* heuristic value

1: $dist\_matrix \leftarrow$ ImportData()
2: $dist\_matrix \leftarrow dist\_matrix + dist\_matrix^{\intercal}$

3: $dist\_matrix \leftarrow dist\_matrix.triu$
4: $max \leftarrow 0.5$
5: **while** $max > threshold$ **do**
6: $\quad max, cords \leftarrow$ FindMax($dist\_matrix$)
7: $\quad dist\_matrix \leftarrow$ UpdateMatrix($dist\_matrix, cords$)

Now the clustering technique that we used is trivial: we find bigger values in the matrix and their coordinates. Then we can easily update the matrix by setting to zero all the distances of one of the two clusters and updating the other. We need to set to zero every entry in the matrix that has the (x, y) coordinates equal to the deleted cluster. Instead, for the other cluster we use the mean between the existing distances, hence if an item has a relation value towards both of them we set the average value, otherwise, if it's in relation with only one of them, we simply divide this value by two.

We keep clustering until our maximum distance reach the threshold value and here we stop. The threshold we have setted in our implementation is really small (0.00000005) because, due to the fact that our data are really spread, we have a lot of entries with zero as value that keep dividing by two our values and so the distances decrease fast.

4.2.2 Set-based clustering

The second idea is to represent every item as a set of items. For this algorithm we tried to leave the "weight idea" and we have built for each item a set of "related items". With these sets we have implemented a hierarchical clustering algorithm quite similar to the first one but using the Jaccard similarity coefficient [19] for the initialization of the distance matrix. Then, like before we found the maximum value and we use the mean value between the two clusters to evaluate the new distance.

4.2.3 Graph-based clustering

The last idea that we implemented about the clustering recommendation system is graph-based. Thus, we built an undirected graph with the values on the edges imported from the metadata with the structure that we can see in Figure 2. In this graph we tried to use the detection of community network. A community detection is a set of nodes such that each set of nodes is densely connected internally. This implies that the graph divides itself naturally into groups of nodes with dense internal connections and sparser connections between groups (see Figure 3 for a graphical representation).

We implemented a hierarchical clustering method that finds this kind of communities in our graph. However, finding communities can be a computationally difficult task. The number of communities within the network, if any, is typically unknown and communities are of unequal size and (or) density, thus our implementation works really slow. So, in order to run this algorithm within the same range of times of the others we decided to use a library that fits quite well our problem.

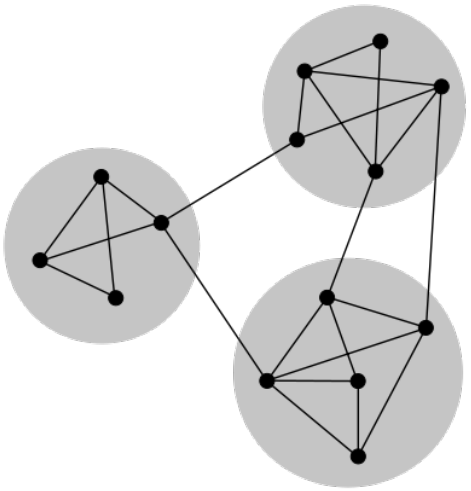


Figure 3: An example of community detection in a network.

Then, we imported "Networkx" [12], an open source implementation of a lot of functions on graphs. Among all these features we have found a function that computes the "community best partition" over a graph that, according to the documentation, uses an algorithm based on an approximation of the well-known maximum-cut algorithm. This means that is based on a NP problem approximated with a polynomial time algorithm that gives an estimation of the right solution.

4.3 Text based recommendation

Another interesting information that are available in our dataset are the textual reviews provided by users about items. Based on this data we have implemented another method to suggest similar items to the target user. Firstly we applied some dimensionality reduction techniques such as tokenization, filtering the text to remove stop words and stemming to normalize words. After this phase we have calculated the TFIDF representation of every review in order to obtain a vector that can be compared with other reviews.

4.3.1 TFIDF

The "term frequency – inverse document frequency" is a well-known information retrieval algorithm used to get weights for terms of some texts. The TFIDF is described as follows:

Algorithm 3 TFIDF algorithm definition

Require: a term w, a text t

1: $weight(w, t) \leftarrow tf * idf$
2: $tf \leftarrow (\#w \in t)$ / $(tot\_words \in t)$
3: $idf \leftarrow \log((\#texts)$ / $(\#texts \ni w))$

Now, after that we ran the Algorithm 3 on every word of every review, we obtained a vector representation of each text. Once we have this matrix containing the TFIDF representation of every text we can easily evaluate the similarity between reviews (hence, between items).

4.3.2 *Implementation*

The first phase of the algorithm uses the NLTK library [13] for filtering the text from stop words and to perform stemming and tokenization. After that, there is only the matter of counting the number of word occurrences, so it's pretty easy. The only problem that we have encountered in the development of this technique is that every documentation that we have found about the TFIDF representation talks about normalize every text to obtain a vector of the same size (adding a zero entry for all the words it actually doesn't contain). With this trick it's easy to group every vector representation in a matrix and to compute the distances with any distance measure that fits for vectors.

In our case this method cannot work because the filtered reviews had an average length of twenty words and in our dataset we have found almost 100K words. This will lead to a huge amount of wasted memory and computational power. So we developed a distance function that does not require normalized vectors (hence, vectors with the same length).

Algorithm 4 Euclidean distance for unnormalized TFIDF representation

Require: lists of tuples d_1, d_2 in the form <*word, TFIDF*>, ordered by word

```
1:  s, i1, i2 ← 0.0
2:  while i1 < len(d1) OR i2 < len(d2) do
3:      if i1 = len(d1) then
4:          v ← d2[i2].TFIDF
5:          i2 ← i2 + 1
6:      else if i2 = len(d2) then
7:          v ← d1[i1].TFIDF
8:          i1 ← i1 + 1
9:      else if d1[i1].word < d2[i2].word then
10:         v ← d1[i1].TFIDF
11:         i1 ← i1 + 1
12:     else if d2[i2].word < d1[i1].word then
13:         v ← d2[i1].TFIDF
14:         i2 ← i2 + 1
15:     else
16:         v ← d1[i1].TFIDF - d2[i2].TFIDF
17:         i1 ← i1 + 1
18:         i2 ← i2 + 1
19:     s ← s + (v * v)
20: EuclideanDistance ← √s
```

As we can see in the Algorithm 4 we scan the two lists and if one of the two considered word is smaller than the other (hence, if it arrives before the other in alphabetical order) we add its value, squared to our total sum. Otherwise (Algorithm 4, line 15), if they are equals we subtract the values. If one of the two lists reaches its end, we simply add the remaining values of the other list (Algorithm 4, lines 3 and 6). At the end, we simply return the square root of the obtained sum.

We used the very same technique also for the implementation of the cosine similarity algorithm.

5. MAPREDUCE IMPLEMENTATION

All our algorithms works quite well on medium size datasets. However, if we are going to process every review and every item hosted for example on the Amazon website, we will surely fail. For this kind of data size (in the order of terabytes) we need to distribute the computation among a cluster of computers in order to meet the time requirement.

5.1 Expected speed-up

According to Amdahl's law [18] we noticed that we can expect a substantial speed-up in the computation based on the parallel fraction of the work. As we can see in the following formula we can expect a maximum increase in the speed with N processor equal to the total amount of work (normalized to 1) divided by the parallel amount of work P divided among N processor plus the serial amount 1-P:

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}}$$

In the chart in Figure 4 we can notice how the speed-up changes based on the parallelized portion in the work. So we need to maximize the amount of parallelized work in our algorithm in order to obtain better results.

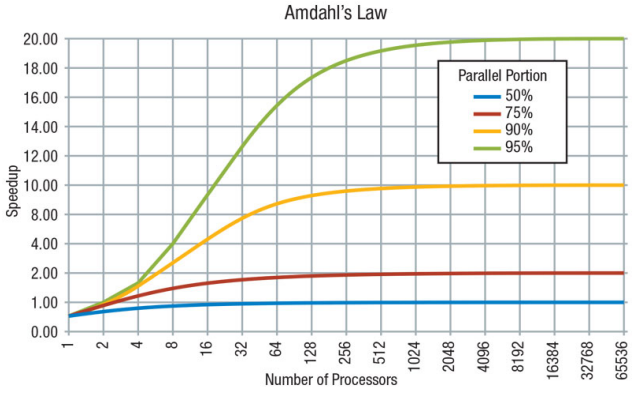


Figure 4: The speed-up of a program using multiple processors in parallel computing is limited by the sequential fraction of the program. For example, if 95% of the program can be parallelized, the theoretical maximum speed-up using parallel computing would be 20x as shown in the diagram.

5.2 Phases of algorithm

In our specific case we reimplemented in parallel the TFIDF evaluation of all the textual reviews in the dataset, that it is the slowest algorithm that we have. For this purpose we used Apache Hadoop [16], an open source software framework written in Java for distributed storage and distributed processing of very large datasets on computer clusters built from commodity hardware. In particular we used the "Hadoop

streaming" utility that allows us to write our map and reduce code in Python, the language on which we wrote all the project.

So, in order to obtain the maximum portion of parallel work we have planned to divide the work in five distinct MapReduce instances:

1. **Word count**: the first MapReduce phase consists simply in the count of words repetitions in each text. We have built a mapper that performs some dimensionality reductions on the text (i.e. tokenization, stop words deletion and stemming), and then it emits tuples in the form <word@itemID, 1>. After that, the results are sorted by the key (i.e. the first element in the tuple) and they arrive to the reduce function. This function takes every tuple with the same key and sum them emitting tuple like <word@itemID, 4>.

2. **Term frequency**: now that from the previous phase we have the number of occurrences of every word in every document we need to calculate the term frequency. So with the mapper we change the form of the data simply emitting <itemID, word=n_occurrences>. With this, mapping data is sorted by itemID and arrive to the reducer. Here the reducer can simply use a counter to maintain the total number of words in the item and with a data structure temporary store every "word=n_occurrences" pair. Once every tuple of this item has been processed, we only need to divide every stored tuple by the total number of words, thus we emit them as <word@itemID, n_occurrences/n_words_item> that means <word@itemID, term_frequency>.

3. **Reviews count**: in this step we need to compute the IDF (inverse document frequency), but for this purpose we need to know how many reviews we have parsed. Hadoop streaming utility lets us to use not only Python as scripting language to implement our functions, but also shell commands. As we know that we have one review for each line in our JSON dataset, in order to discover the number of reviews we only need to count the number of lines in the document. So to this end we used the "cat" shell command as mapper (that it will read every line that Hadoop streaming pass to it and print it to standard output) and the "wc -l" shell command as reducer (that will count the number of lines with the -l option). Notice that we need the total amount of lines so we need to specify that the reducer will be only one, thus we will obtain only one result.

4. **TFIDF evaluation**: now that from previous steps we have collected every information that we need to calculate the TFIDF value of each word in each document, we take the results of the TF phase in the form of <word@itemID, term_frequency> and we map it with <word, itemID=term_frequency>, so now to our reducer will arrive tuples sorted by word. Now we only need to count for each word how many reviews contain it, divide it by the result of the previous phase (i.e. the total number of reviews) and perform a logarithm on it. Finally we get the IDF of this word. Then, it is easy to multiply it with the TF value of that tuple and emit it as a tuple in the <word@itemID, TFIDF> form.

5. **Reorder phase**: the last phase is simple a reordering of these data. They will present each document as a list of each word that contains and its TFIDF value.

5.3 Hadoop streaming dataflow

Hadoop Streaming is an utility that provides an interface to the Hadoop system that can be accessed not only in Java but also with other scripting languages. Moreover the word "streaming" suggests that this system provides data as a "stream of data", hence it reads the input line by line from the specified directory and then it pass it to the mapper. Once the map phase ends (i.e. when the system prints to standard output its results), in a standard MapReduce implementation we will have a shuffling phase that will group every <k, v> tuple in the <k, list(v)> form for each key. However, this is not going to happen in the Hadoop Streaming interface that will only sort our data by key without grouping its values. The only ensured thing is that all the tuples with the same key will arrive to the same reducer.

The absence of the grouping feature caused some problems on the design of our reducers. However, as we can see in the Algorithm 5, for each line that arrives from the standard input we log its value in a temporary dictionary that will be printed when the key of the current line is different from the previous one (Algorithm 5, lines 13-16).

Algorithm 5 Reducer of the TFIDF evaluation phase

Require: <*word, itemID=TF*> pairs in standard input, ordered by key

```
1: tot_n_reviews ← ReviewsCountResult()
2: target_key ← None
3: temp_dict ← Empty
4: count ← 0

5: for all line ∈ standard input do
6:     word, itemID, TF ← ParseInput(line)
   ▷ First iteration case
7:     if target_key = None then
8:         target_key ← word
   ▷ They are the same so log it
9:     if target_key = word then
10:        temp_dict ← add(itemID, TF)
11:        count ← count + 1
   ▷ Otherwise print and reset
12:    else
13:        for all itemID, TF ∈ temp_dict do
14:            newKey ← "target_key@itemID"
15:            newVal ← TF * (count/tot_n_reviews)
16:            print newKey, newVal ▷ Reset for next word
17:            temp_dict, target_key ← init to next word
18:            count ← 0
   ▷ Catch final counts after all have been received
19: for all itemID, TF ∈ temp_dict do
20:    newKey ← "target_key@itemID"
21:    newVal ← TF * (count/tot_n_reviews)
22:    print newKey, newVal
```

Another important thing to remember about the streaming interface is that not all the available libraries for Python can be used in the implementation of the map and reduce functions. Unfortunately, this utility is only a ".jar" file written in Java that translates scripts into acceptable map and reduce functions. Therefore, this translate feature only understands standard libraries of Python and, for example, the Natural Language Toolkit (NLTK) [13] that we used in our serial implementation for both the stemming and the tokenization phases cannot be imported because is an open source implementation that is not provided in the default version of the language. Thus, regarding the Hadoop implementation we have written from scratch our own function of stemming and tokenization.

5.4 Hadoop Distributed File System

Another problem that we have solved concerning the parallel implementation is that our dataset originally were compressed with GZIP. This kind of file cannot be used as input for the Hadoop MapReduce framework because is not partitionable due to the high level of compression. When we decompressed this file, we have seen that the size of the resulting JSON is more or less 5 times bigger than the original ".gz" file. However, that is not a big problem because the easiest way in which we can provide an input to the Hadoop MapReduce is to store the desired file on the cluster in the distributed system provided by the framework.

So we have uploaded this uncompressed file on the HDFS (Hadoop Distributed File System) layer, spread among the cluster that is easily managed through Hadoop. In fact, among the other things, this framework lets us access the file system through its commands that include lots of shell commands. Moreover, we used as development environment a Cloudera[1] virtual machine that provides Apache Hadoop with an integrated HDFS system that can be managed and monitored not only through shell commands but also with a visual interface named HUE (Hadoop User Experience) that lets we monitor every phase of our MapReduce and provides a simple way to inspect logs and results.

However, we used this user interface only in debugging phase to inspect logs, but for all the other phases we used the command line because gives us more customization power: for example, in the phase four we use as input the results of the phase two that gives for every pair "word-itemID" the term frequency value, but we also need that every reducer knows the result of the third phase (i.e. the total number of reviews). We have solved this problem by passing the third phase output as a custom argument, as you can see in the seventh line of Listing 1 that contains the Hadoop command that starts the fourth MapReduce phase.

```
1  hadoop jar hadoop-mapreduce/streaming-2.1.jar
2  -mapper tfidf_map.py \
3  -reducer tfidf_red.py \
4  -input tfidf/wordindoc/output1 \
5  -output tfidf/word_tfidf/output1 \
6  -file idf_map.py -file idf_red.py \
7  -cmdenv TOT_LINE=tfidf_v1/n_reviews/output1
```

Listing 1: Our shell command used to start a MapReduce job with the streaming utility.

[1]Cloudera home page: `http://www.cloudera.com`

6. RELATED WORK

Recommendation systems are achieving widespread success nowadays thanks, among the other things, to the growth of e-commerce services and to evolution of the Internet. For that reason many companies and research universities have designed and developed a wide range of smart techniques, that vary from the classical content based filtering, collaborative filtering and hybrid recommender systems to various modern recommendation approaches such as semantic based techniques [3].

A well-known example that uses a recommendation system to personalize its online store is Amazon.com[2]. In particular, it uses an item-to-item collaborative filtering approach to radically change the online market for each customer based on its interests [10].

In the modern literature there are some works that analyze collaborative filtering techniques, such as [4] that proposes a combination of different state-of-the-art methods to obtain an hybrid recommendation system in order to improve performance, and [1] that presents an academic overview of the field of recommender systems by describing limitations of current methods. Among many others, [5] highlights the principal strengths and weaknesses of many collaborative filtering approaches; however, there are no works in literature in which there is a formal comparison of clustering, collaborative filtering and text based techniques for the same recommendation task. Furthermore, other research works frequently use other datasets, such as Netflix prize dataset [11] and Movielens data [7]. Thus, our work attempts to find some relevant information about the techniques used and their performance.

7. EXPERIMENTAL EVALUATION

In this section we present some details concerning our experimental evaluation, and then we describe our major findings. First of all we discuss all the serial implementations and their implications by showing performance comparison and some stress-test to higher data sizes. In particular, since we have implemented many versions of the same algorithm, we compare all of them to demonstrate the approach of every technique that works better, and then the best approach among all the three main methods. Secondly, we present our evaluation of the parallel implementation in the MapReduce framework, with a consistent number of experiments that compare the serial version of the text based recommendation system with the distributed one in terms of time, for various data sizes.

7.1 Result evaluation methodology

The dataset that we have used to develop our algorithms does not contain any control set with already labelled results. Thus, we cannot compare our outcomes with something that we know it's right in order to evaluate the precision of our algorithms.

Regarding the user-based collaborative filtering method we used a widely known information retrieval technique named cross-validation that consists into splitting the original raw dataset in two main partitions: the test set and the training

[2]Amazon website: `http://www.amazon.com`

set. By running our algorithms only on the data within the training set, we can compare on a later stage the predicted result with the expected result contained in the test set, and then we can evaluate the algorithm performance.

Among the other things, our dataset contains actually real data about Amazon, so we used the Amazon recommendation system to compare our results as regards the item-based algorithms (i.e. clustering and text based). In fact, if we search suggestions for an item with our algorithm, we can easily find this item also on the Amazon platform and then look what they are suggesting based on this very object. This kind of comparison works quite well for our item based recommendation algorithms.

We decided that our item-based algorithms will give us ten suggestions for the selected items, so we compare these results with the first ten that Amazon proposes. As we can see in Figure 5, Amazon usually gives a quite long list of suggestions (e.g. for the glasses the list is composed of six rows of five elements each). However we use only the first ten in the comparison with our results. Moreover we know that our data has been recorded over a year ago so we don't expect an high correspondence ratio (and we hope that Amazon algorithm is cleverer than our academic one).

Figure 5: Example of items suggested by Amazon if we are looking to the page of the item "Spiegelau Vino Grande Burgundy Wine Glasses, Set of 6".

In order to better understand our results, we used this control set to evaluate two parameters of our algorithms:

- **Precision**: the fraction of retrieved instances that are relevant. In other words, an high precision means that the algorithm return substantially more relevant results than irrelevant:

$$Prec(N) = \frac{|relevant \cap retrieved|}{|retrieved|}$$

- **Recall**: the fraction of relevant instances that are retrieved. In simple terms, high recall means that an algorithm returned most of the relevant results:

$$Recall(N) = \frac{|relevant \cap retrieved|}{|relevant|}$$

Once we made the experiments with these values we can compare our item-based algorithms in order to see which fits better following the Amazon example. At a later stage, an overall evaluation are presented to determine the best technique implemented.

7.2 Results and performance evaluation

It is important to evaluate each of the techniques implemented to demonstrate that our approaches work well and fit our problem. To do so, here we compare each of our methods and then we describe the main challenges that led us to design different versions of each approach. Furthermore, some experiments are conducted to evaluate them individually and finally we did an overall evaluation of all the developed techniques.

7.2.1 Collaborative filtering

In order to assess the goodness of our collaborative filtering algorithm we have done several computation tasks on various data sizes and types taken from the raw Amazon reviews dataset. Two different evaluation metrics could provide us a different performance analysis of our implemented distance measures, thus these made it easier for us to see the problem from various angles. To address both types of evaluation we have used the k-fold cross-validation method [14], a validation technique for assessing how well our model will generalize to an independent dataset. In particular we splitted the raw dataset into five equal-sized subsamples (after we shuffled the complete dataset to avoid bad partitioning), and then we merged four different folds for five times to form the training set and the test set for every evaluation experiment (see Figure 6).

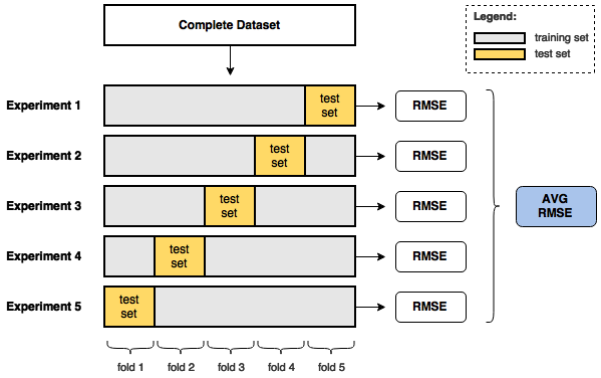


Figure 6: 5-fold cross-validation technique. The raw dataset is firstly divided into 5 subsamples or "folds". One fold is designated as the test set, while the remaining four folds are all combined and used for training. The test accuracy is computed for each of the test sets by using the RMSE metric, and then averaged to get a final cross-validation accuracy.

The two main tests we have done are the following:

- Predicting items that users actually bought: for this task we have evaluated how many items i for a user u in the training set are also present in the test set (i.e. items that are actually bought by the user u).

- Predicting correct ratings: for this task we have evaluated how well the system provides a set of predicted ratings on items i by a user u (i.e. by comparing expected ratings in the test set with predicted ratings in the training set).

With regard to the first test task, we have used the complete clothing reviews dataset (approximately 3GB of data) and we have run our recommendation algorithm five times (one for each experiment) in a single machine to predict how many items within the test set are effectively recommended to the target user. This task has proved to be so challenging due to the execution time and, among the other things, to the impossibility to guess items that users have actually bought. However, we have found an interesting thing: on average, for every user processed the adjusted cosine similarity finds approximately 2x of the actually bought items compared to those found by using the Pearson correlation coefficient, the manhattan distance and the euclidean distance, for any number of considered neighbors. Lastly, the basic cosine distance works slightly worse than the adjusted one in this task.

Regarding the task on predicting ratings, we have taken the same raw dataset as before, and then we have cleaned and preprocessed it. More deeply, after the shuffle phase we have run some of our implemented preprocessing scripts to reduce data sparsity for evaluation purposes. According to statistics on the average number of Amazon users and items reviews from the Stanford Network Analysis Project [9], we have reduced our original dataset to 5413 users and 29159 items by filtering users with less than 20 reviews and items with less than 10 reviews. After that, we have created five new folds from reduced dataset and we have run our designed evaluation algorithm to those data.

As we can see in Figure 6, we have performed an experiment for each of the five training and test combinations for each of the five distance measures that we have implemented. To evaluate them we have used a widely known machine learning and statistics evaluation metric known as the Root Mean Square Error (RMSE) [8]. Given a set $K = (u, i)$ of hidden user-item ratings, if $r_{u,i}$ is the predicted rating for user u over item i and $\overline{r_{u,i}}$ is the expected rating, then the RMSE is defined as follows:

$$RMSE = \sqrt{\frac{\sum_{(u,i)\in K}(r_{u,i} - \overline{r_{u,i}})^2}{\mid K \mid}}$$

Each experiment performed in a single machine took about 24940 seconds to finish (approximately 7 hours/experiment) and it consisted in calculating the RMSE for each of the distance measures for k nearest neighbors, with $k = 20$, 40, 60, 80 and 100 for that particular combination of training and test sets. Lastly, the obtained results were averaged to get the final RMSE for each of the distance metrics, for each k (the results for each single experiment are in Appendix A).

The obtained results, as we can see in Figure 7 and in Table **??**, show us that, for each k, the best accuracy is reached by the cosine similarity that always remains below RMSE=1.0. The plot additionally shows that manhattan distance and euclidean distance are very similar in performance, but the number of predicted items is slightly different for $k = 20$, 40. To be precise, both the distances slightly improve their accuracy as the k becomes larger, and the euclidean distance performs slightly worse in both the number of predicted items and the accuracy (in the order of 0.001) than the manhattan metric does. Furthermore, Pearson correlation coefficient is considered a good metric in the modern literature but it seems to be unsatisfactory for our task, while adjusted cosine similarity converges to the classical cosine similarity for larger k.

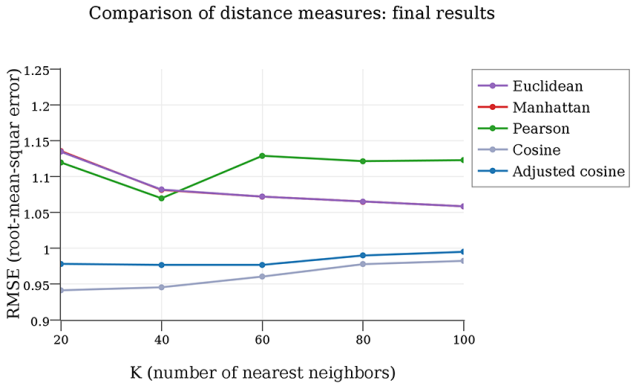


Figure 7: Comparison of distance measures: final results on accuracy given by the average of all five experiments done for each combination of training and test sets, according to the 5-fold cross-validation methodology.

	Adj cos.	Cosine	Pearson	Manh.	Eucl.
k=20	844	961	135	950	944
k=40	1211	1485	172	1330	1329
k=60	1398	1783	203	1545	1545
k=80	1549	1975	239	1701	1701
k=100	1685	2124	264	1844	1844

Table 1: Comparison of distance measures: final results on number of predictions given by the average of all five experiments done for each combination of training and test sets, according to the 5-fold cross-validation methodology.

In conclusion, our experiments show that the user-based collaborative filtering is constrained by the use of a good metric to effectively find neighbors, and in particular attests that the best distance measure for this kind of recommendation system is the pure cosine similarity. Despite the adjusted variant takes into account the "grade inflation" problem and it performs better in predicting items that users actually bought, for this very dataset it is dominated by the pure cosine. The use of RMSE as a performance measure is justified firstly by our needs to penalize larger errors more severely than other error metrics do, and secondly for its suitability for the prediction tasks (it measures inaccuracies on all ratings, either negative or positive) [6]. Finally, the obtained results are so promising since the actual state-of-the-art RMSE on collaborative filtering algorithms are very close to our RMSE results [15].

7.2.2 *Hierarchical clustering*

As regards the clustering techniques we have considered average values of precision and recall on the proposed suggestion on 20 items. The results are not so satisfactory:

- Weight based clustering: for this algorithm the precision is 0.12 and the recall is 0.08.
- Set based clustering: for this one we are going a little better: the precision is 0.22 and the recall is 0.15.
- Graph based: for this last one the precision is 0.13 and recall is 0.09.

These parameters have quite low values, however we have an explanation for that: first of all the Amazon review dataset is really spread: it means that information are not connected well and so we have lots of "small item cluster". This leads our algorithms to retrieve less than 10 suggestions for some items. For example, in one of the twenty cases the first algorithm gets 3 suggestions, the second one gets 4 and the last one gets only one. This leads our precision and recall to decrease a lot. However, even the items that get the right number of results does not excite: the best case among the twenty tries is for the weight-based with a precision and recall of 0.4, for the set-based with both values of 0.4, and for the graph-based with both values of 0.3. Notice that if the relevant results and the retrieved are the same number, we have the same value for both the recall and precision.

We have notice that, with this technique for measuring the results, among the clustering methods the "set based" works a little better than the other two, and surely the one that exploits the graph representation is the worst.

However, this evaluation method does not completely convince us, so we have tried to look for every result that we get in this experiment what is exactly the item corresponding to that ID. We discover that, according to our experience and judge, more than 80% are items slightly correlated with the target. For example, we think that is quite a good suggestion that if we search a "Green Sugars Spinkles" for baking cake we get other colours variation of the same product and other products filling designs on cakes.

Thus, in summary we think that this algorithm works quite well. The only real problem encountered is that this very dataset is too spread and so we have items in small or even single item clusters and this leads to a small amount of suggestions from our algorithms for those products.

7.2.3 Text based recommendation

With this algorithm we have found that this method does not work for the implementation of a recommendation system. With all the distance measure that we have used (Euclidean, Cosine and Jaccard) the result is the same: precision and recall are near to zero.

We have tried to investigate in this fact, and by looking at the items that this system suggests to us we have found out that they are obviously wrong. For example, with as target a bottle of an alcoholic beverage our algorithm tried to suggest a "stainless-steel herb mill" or some bars at vanilla and yoghurt taste. By deeply examining the reviews of the example we have seen, if we remove the stop words and we use a stemming process we have some words in common. However, they are not very explicative about the product reviewed. For example, some common terms found are: "strong", "money" and "product".

So we discovered that this method does not fit into the goal of recommending products in an e-commerce environment. However we think that this algorithm can be used to produce some other important results in other applications.

7.2.4 Overall evaluation

Based on the results of our experiments we can state that the method that works better among the implemented techniques is the collaborative filtering based algorithm. This method produce surely the best results based on our measurements.

The other two techniques are not suitable for the development of a recommendation system. Precisely, we think that the clustering based techniques can surely work, but our data are too spread so our clustering-based algorithms can suggest items only for well connected products. Among the other things, the presence of lots of single object clusters can lead to the failure of our system (hence, we don't suggest anything for some products).

Instead, as we have noticed, the analysis on similar reviews does not lead to a working recommendation system because similar textual evaluation does not mean that we are talking about similar products. However, the algorithm that computes the TFIDF can be used for other purposes and is still useful. For example, one future work can be try to characterize users based on their writing style in order to improve the collaborative filtering technique.

So we suggest, based on our experiments and our experience, the use of the collaborative filtering with the cosine similarity as distance measure for the real implementation of a recommendation system. This method is the only one that provides to us outstanding proof of its accuracy and correctness. The clustering method seems to work but with some drawbacks so we cannot evaluate it as completely positive and the text-based algorithm cannot be used because in our evaluation we have seen that provides mostly random suggestions.

7.3 Comparison with MapReduce

About the correctness of the text-based algorithm we have seen that on our small testing datasets we get exactly the same results from the MapReduce implementation and from the serial one. So we had asses that these algorithms are equally working on small data sizes. On the performance side, we are expecting an improvement by our parallel implementation. So we compared the execution time between the two algorithms.

As we have seen in the section 5.1, we can use the Amdahl's law[18] to compute an expected speed-up of our algorithm and then compare it with the real results that we have obtained. After the development and testing in a Cloudera environment we decided to try our algorithm also on a real cluster. Using the free trial that Amazon AWS [2] grants to us, we have configured a cluster with four working nodes (they are unfortunately "micro instances", that are the only free available). Once configured our system we have down-

loaded on the distributed file system provided by Amazon our entire dataset (about 18GB).

Once we were ready, we have calculated with the Amdahl's law the expected speed-up. Considering that we have parallelized all the TFIDF algorithm that produces the values and we only keep in serial the comparison within vectors, we can compare only the first part. So, without any serial part, the Amdhal's law becomes the following:

$$S(N) = \frac{1}{\frac{P}{N}} = \frac{1}{\frac{1}{4}} = 4$$

So we expect an algorithm that process this data more or less four times faster.

In our serial implementation we added some logging functions that we can use to monitor the time that this will take (obviously we cannot expect that our personal laptops actually process 18GB of data). However, we have simulated this execution by loading all the data like in the parallel implementation and by running the first phase on them. After ten minutes we have interrupt the execution and looked at logs in order to know how much time every iteration has taken. Then, as we know how many of them we need (i.e. the number of reviews), we have calculated the final value. For the second phase we have cut and paste the first results until the desired length and then we have done the same simulation. According to our estimation, processing all these reviews and calculating all the TFIDF coefficients will take on our single machines about three months and a half.

On the MapReduce implementation we used the same trick (we simulated the execution because we can't run that cluster for a long time with our free AWS trial account). In the end, after the simulation we have calculated that the same dataset with our MapReduce will take slightly more than two months. These numbers correspond approximately to a speed-up of 1,75 that is less than half our expectation. Probably this difference between expected and real value is caused by the power of the instances on our cluster. Every "micro instance" that we used has in fact only 2 processor and only 2GB of RAM. Instead, our used laptop has 8GB of memory and a quad-core processor. Thus, this difference may have mitigated the speed-up.

Another thing that can mitigates the speed-up is that every phase of MapReduce has a master node that partitions the input data, sorts the output of the map phase and redistributes it to the reducer. These are sequential operation (hence, we have not counted them within the serial portion in the expected speed-up calculation). So MapReduce cannot be faster as an hypotetical parallelized algorithm due to its serial phases that it uses to coordinate the cluster.

In the end we think that by increasing the performance of every cluster component we can probably reach a result that approaches the expected speed-up. Anyhow, we had a substantial improvement in the performance so our goal with this implementation is reached.

8. CONCLUSIONS

In conclusion, for this project we developed three different types of algorithms for a recommendation system implementation. First of all, we have deeply analysed our data structure in order to figure out how to use all that information. Not all our ideas were good because, as we have seen in the experimental evaluation, the text-based recommendation system does not work well. However, the user-based collaborative filtering and the clustering techniques exploit well the knowledge contained in the dataset. Unfortunately, for the clustering techniques we have too much "single node clusters" that lead our results to a bad precision evaluation.

The implementation of these algorithms has been long because we have tried more than one distance measurement for each technique in order to better understand if these methods can or cannot really work. So we have deeply analysed every single method by more than one point of view, seeking for the best possible result.

We have also performed some advanced evaluation techniques on our algorithms in order to better understand and measure the power, the correctness and the performance of our implementations. Even for this task we have found a lot of interesting state-of-the-art techniques, such as the k-fold cross-validation. We have tried some of them in order to evaluate our work by different angles.

In the end, we have also implemented a parallel version of the text-based algorithm using a MapReduce chain that evaluates the TFIDF value of a list of texts. This distributed framework allowed us to process a large amount of data that usually are quite impossible even to store on a single computer. Also this part of our project was really interesting because we think that this kind of computation patterns represent the future of data processing in the era of big data. In fact, datasets keep growing and the computational power of machines have some limits, so we need to learn how to distribute computation and storage. The Hadoop framework is quite a strong tool for this kind of jobs and we enjoyed using it.

Thus, thanks to this project we learned how the data can be a powerful source of power and we realised that, if we exploit it well, this very source of power can leads to amazing results.

9. REFERENCES

[1] G. Adomavicius and A. Tuzhilin. Toward the next generation of recommender systems: a survey of the state-of-the-art and possible extensions. *Knowledge and Data Engineering*, 2005.

[2] Amazon Web Services, Inc. Amazon web services webpage. `http://aws.amazon.com/it/`, 2015.

[3] D. Asanov. Algorithms and methods in recommender systems. 2011. Available at: `https://www.snet.tu-berlin.de/fileadmin/fg220/courses/SS11/snet-project/recommender-systems_asanov.pdf`.

[4] R. Burke. Hybrid recommender systems: Survey and experiments. *User Modeling and User-Adapted Interaction*, 2002.

[5] F. Cacheda, V. Carneiro, D. Fernandez, and

V. Formoso. Comparison of collaborative filtering algorithms: limitations of current techniques and proposals for scalable, high-performance recommender systems. *ACM Transactions on the Web (TWEB)*, 2011.

[6] T. Chai and R. R. Draxler. Root mean square error (rmse) or mean absolute error (mae)? *Geoscientific model development*, 2014.

[7] GroupLens Research Lab, University of Minnesota. Movielens datasets. `https://grouplens.org/datasets/movielens/`, 2010.

[8] M. Hibon and S. Makridakis. Evaluating accuracy (or error) measures. *Insead*, 1995.

[9] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, 2014.

[10] G. Linden, B. Smith, and J. York. Amazon.com recommendations. item-to-item collaborative filtering. *IEEE Internet Computing*, 2011.

[11] Netflix, Inc. Netflix price dataset. `http://www.netflixprize.com/`, 2009.

[12] NetworkX developer team. Open source high-productivity software for complex networks. `http://networkx.github.io`, 2014.

[13] NLTK Project. Natural language toolkit (nltk) library home page. `http://www.nltk.org/index.html`, 2011.

[14] P. Refaeilzadeh, L. Tang, and H. Liu. Cross-validation. *Encyclopedia of Database Systems*, 2008.

[15] H. Su and T. M. Khoshgoftaar. A survey of collaborative filtering techniques. *Advances in Artificial Intelligence*, 2009.

[16] The Apache Software Foundation. Apache hadoop project webpage. `https://hadoop.apache.org`, 2015.

[17] University of California, San Diego. Amazon review data: This dataset contains product reviews and metadata from amazon, including 143.7 million reviews spanning may 1996 - july 2014. `http://jmcauley.ucsd.edu/data/amazon/links.html`, 2014.

[18] Wikipedia. Amdahl's law page on wikipedia. `https://en.wikipedia.org/wiki/Amdahl's_law`, 2015.

[19] Wikipedia. Jaccard index page on wikipedia. `https://en.wikipedia.org/wiki/Jaccard_index`, 2015.

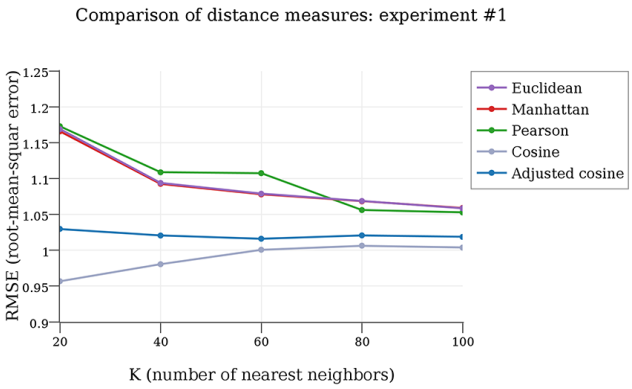
APPENDIX

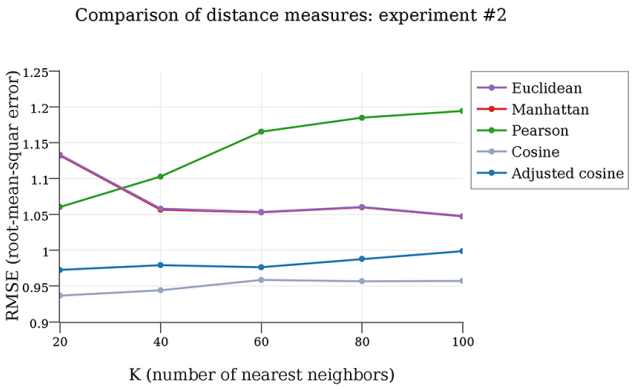
A. CF EVALUATION: EXPERIMENTS

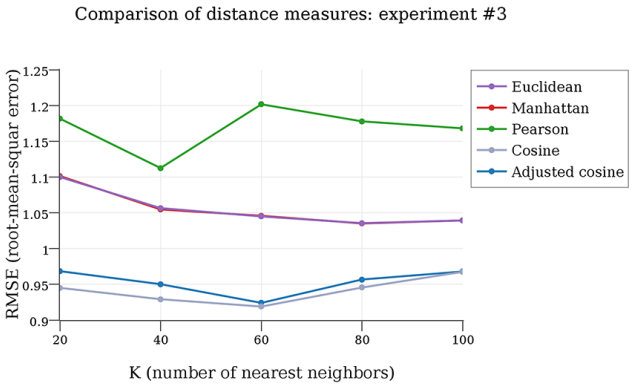
In this section we present some of our experiments concerning the user-based collaborative filtering evaluation.

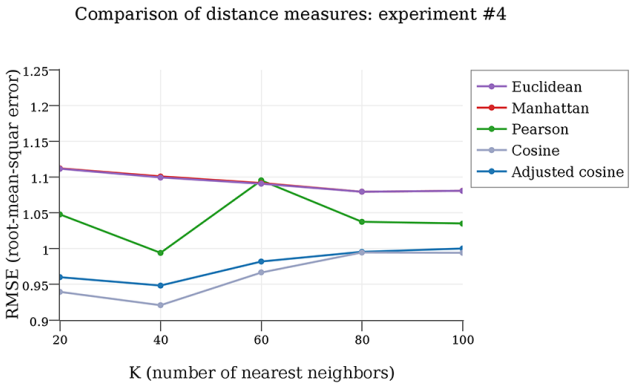
As we have seen in 7.2.1, we conducted five different tests for each combination of training and test sets, according to the adopted k-fold cross-validation methodology, with k=5. These five steps are summarized in Figure 7, that represents a resulting plot given by the resulting RMSE average from each of the following tests.

As regards the experiments themselves, this section contains all of them in order to demonstrate that a single training and test partition might lead to inaccurate results.









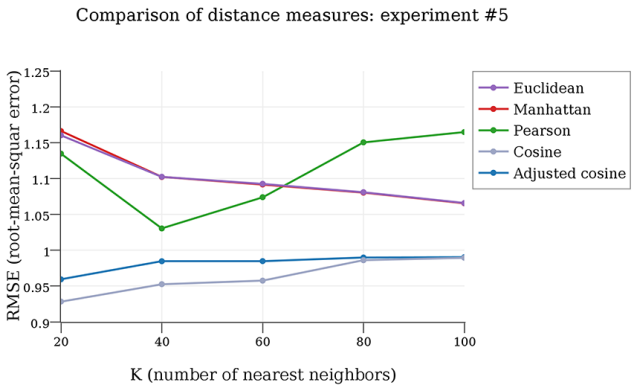


Figure 8: Experiments 1, 2, 3, 4 and 5: comparison of distance measures on accuracy, done for all the combinations of training and test sets, according to the 5-fold cross-validation methodology.