Hadoop-based Movie Recommendation Engine:

A comparison of the Apriori algorithm (for building association rules) vs. the k-means method (for clustering)

by Sofia Parfenovich,
Data Scientist at Altoros

Table of contents

1. Executive Summary

Building a data analytics service for a large online store involves a number of challenges. Product data may range in size from several to hundreds of terabytes and be stored in different locations. In addition, when looking for patterns in massive data sets, for some algorithms, the required amount of memory can grow exponentially relative to the number of analyzed elements. Time is another critical issue—this is why companies adopt high-performance tools that support parallel computing, such as Hadoop.

One of our customers needed a recommendation engine for their e-commerce service that would provide on-demand media streaming. The solution was to increase sales and attract a larger audience by making targeted recommendations of movies. Our task was to develop a mathematical model that would be able to handle massive data sets and provide relevant suggestions to each user. Due to the large size of data, it was clear that we would have to use Hadoop for parallel processing.

Due to the extremely large size of data sets, the customer initially decided to build movie recommendations with association rules, a rather unconventional method for this type of application. However, as a rule, these kinds of systems are based on collaborative filtering methods, such as clustering, that find groups of similar goods and similar user profiles. The recommendations are either built based on a selection of goods from the same cluster, or the service suggests goods previously bought by a user with a similar purchase history. In general, association analysis makes it possible to find data patterns and create simple rules based on them.

In addition, there are some algorithms for building association rules that support the MapReduce paradigm. In theory, this approach would have a number of advantages:

- First, data pre-processing would be much simpler and faster, compared to data pre-processing for clustering methods. This is very important, if you deal with big data.
- Second, it would be easier to store a set of rules than find which cluster each user belongs to and what goods can be recommended to them every time.

This white paper gives a detailed account of how we implemented a recommendation system using two methods that support Hadoop distributed processing: a) building **association rules** with the *Apriori* algorithm and b) **clustering** data with the *k-means* method. Below, we will compare the two methods and overview the recommendations they produce.

2. The Basics of Association Analysis

2.1 Algorithms for building association rules

We were to implement the association rules method, so that we would be able to scale it for big data later on. After all, we were going to use MapReduce (Hadoop) distributed computing for processing. Unfortunately, not every algorithm is suitable for the MapReduce approach, and this greatly limited the number of methods we could choose from. Below, we will take a closer look at some of the algorithms that could be implemented, but first let's define several notions necessary to describe how they work:

- *Transaction*. Each movie watched by a customer was considered to be a transaction performed by this customer. Thus, several movies comprised a sequence of transactions. The table that contained the lists with all the movies watched by each customer was called the transaction database.
- *Rule*. The result of calculations performed with algorithms was called a set of association rules *{X=>Y}* composed of two sets of movies, X and Y. A rule was to be interpreted in the following way: if a user has watched the movies that belong to set X, there is a certain probability that they will also watch a movie from set Y. We limited the Y part of our rules to a single element.
- *Minimum support level* corresponded to the minimum number of occurrences for a particular sequence of transactions in the transaction database that were considered sufficient to make a rule.

There are several algorithms for building association rules that can be implemented with Hadoop. The most popular ones are **AIS**, **SETM**, **Apriori**, and **AprioriTiD** (a modified version of Apriori). All of these use the same general principle to find association rules. In brief, they first look for frequent sequences among the transactions performed by all users. The sequences are defined in iterations. During each iteration, a set of movies of a particular length, called "candidates," is selected. Then the algorithm calculates how many times they occur among all the transactions. If a candidate exceeds the pre-set minimum support threshold, it is considered to be a frequent sequence. The association rules are then built based on these sequences. The difference between the algorithms is in the details of their implementation. They use various approaches to scanning the transaction table, building lists of candidates, and calculating the number of occurrences.

The main drawback of the **AIS** and **SETM** algorithms is that—due to their nature—they generate redundant candidates in frequent sequences that would not pass the minimum support threshold regardless of its value (see descriptions of the AIS and SETM algorithms). In a real-life system, this would result in too many operations and more time to perform the calculations. **Apriori** finds sequences using a more rational approach that reduces the number of candidates thanks to the antimonotone property. We will describe how this algorithm works in more detail below. **AprioriTid** is a modified version of Apriori with an additional table where the candidates are stored. At the beginning, this somewhat slows down the calculations, but it makes them faster towards the end, when we need to find the longest sequences.

The customer initially selected unmodified Apriori as the most suitable algorithm for an implementation with Hadoop, since all the advantages of the additional table with candidates would become negligible due to distributed processing.

2.2 Two ways to speed up data analysis

In addition to parallel processing that would speed up data analysis, there are two ways to further decrease computational efforts with data filtering and aggregation:

a) *Filtering*—eliminating part of information that will never be used by the algorithms. This approach not only helps to considerably shrink data size, but—in some cases—it can be necessary (e.g., for noisy datasets). Below, we will show how pre-processing data can change the results produced by the Apriori algorithm.

b) *Data aggregation*—grouping data with partial loss of information. If we take movies as an example, "Startrek/Season 1/Episode 1" and "Startrek/Season 1/Episode 13" can be replaced by "Startrek/Season 1" or even simply "The Startrek Series." Data aggregation not only helps to decrease the amount of information but also eliminates data redundancy, which is evident in case of TV series.

2.3 Association analysis and big data

Association analysis was first implemented to analyze the so-called "market basket." Association rules can be effectively used to segment customers based on their shopping behavior, analysis of customer preferences, planning patterns for merchandise placement in supermarkets, cross-marketing, targeted advertising by mail, etc. However, these algorithms are not limited to applications in the retail business. Other fields that use association rules for data analysis include:

- credit card and insurance fraud detection
- finding the reasons for failures in telecommunication networks
- DNA analysis
- processing data in sociological studies
- text analysis (in term definition and related applications), and much more

To build association rules, we need to find dependencies between related events in data sets. We can say that there is an association link between event A and event B, if event B is the consequence of event A. Unlike with other types of data analysis, we do not take into account object properties. Instead, we look at a series of simultaneous events. Here, we are interested in whether the event happened or not, rather than in its peculiarities. Association rules were first used to find typical purchasing patterns in supermarkets. This is why this method is sometimes called market basket analysis [1].

Questions in a Shopping Cart

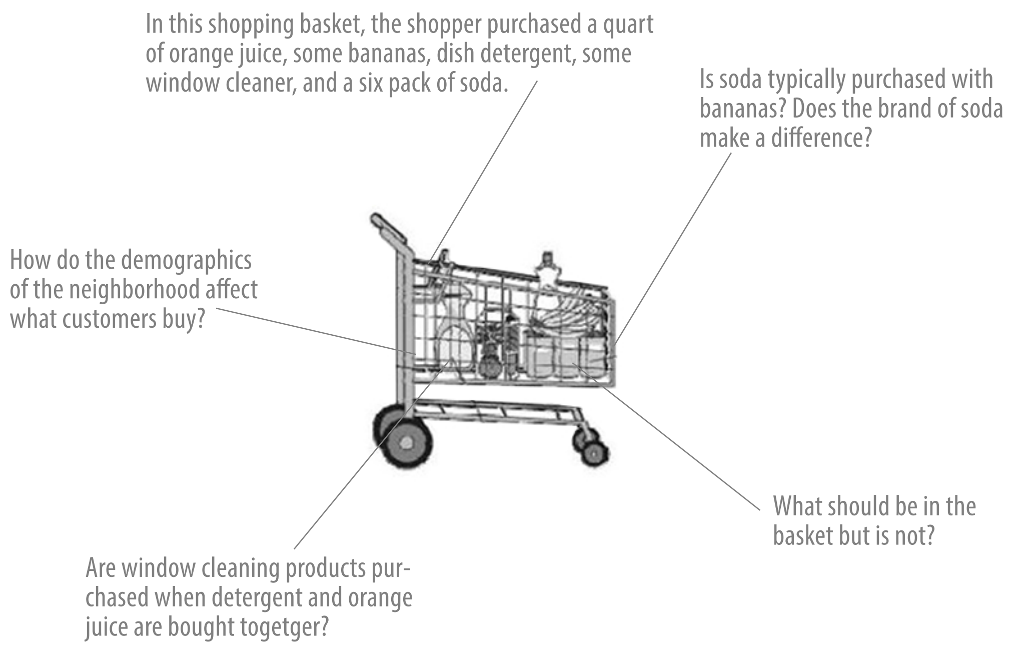

In this shopping basket, the shopper purchased a quart of orange juice, some bananas, dish detergent, some window cleaner, and a six pack of soda.
Is soda typically purchased with bananas? Does the brand of soda make a difference?
How do the demographics of the neighborhood affect what customers buy?
What should be in the basket but is not?
Are window cleaning products purchased when detergent and orange juice are bought togetger?

Figure 1: Questions in a shopping cart. Source: *Gordon S. Linoff and Michael J. Berry*

In their simplest form, association rules only indicate whether there is an association. This is why they are called Boolean association rules [2]. If we take a movie recommendation service as an example, a typical Boolean association rule would be that: those users who watched "Pretty Woman" usually also watch "Bridget Jones's Diary." We are considering two situations: whether a user has or has not watched a movie (and in case of an online store—whether they have purchased certain goods). At the same time, we ignore additional information about the movie or the piece of merchandise, user characteristics, etc. The main advantage of association rules is that they are very simple to understand for humans and can be easily interpreted with programming languages. Nevertheless, just like any other method, they are not always applicable. It depends on the particular task and type of data. If your goal is to create a statistically reliable recommendation and it is acceptable that there might be no relevant recommendations for some situations, association rules will work. If you need to recommend as many products as possible and there must be a recommendation for each user, you should probably use other methods.

2.4 Types of association rules and evaluation of results

Overall, there are three types of association rules [3]:

- *Useful rules* contain real information that used to be unknown but has a logical explanation. Such rules can be used in decision making to gain profit. For example, we can recommend the "Firefly" series or "Serenity" to a person who has watched "Star Wars" and "Startrek." This will be a useful rule, since, although "Serenity" it is less popular than "Startrek," the user will probably like it.
- *Trivial rules* contain real information that has a simple explanation and is already known. Such rules cannot be useful because they either reflect well-known dependencies or repeat the results

of previous research. Sometimes, such rules can be used to prove, if decisions based on earlier analysis have been applied. Once again, taking movies as an example, we can recommend "Pirates of the Caribbean 2" to those who have watched the first part. This is a correct dependency; nevertheless, it is likely that users will watch the first part without this recommendation.

- *Inexplicable rules* contain information that cannot be explained. Such rules may be generated from anomalous values or only hidden knowledge. They cannot be used in decision-making as their inexplicability can lead to unpredictable results. Further analysis is required to get a better understanding. In fact, it is hard to anticipate a situation where an algorithm produces inexplicable rules, especially, in a use case like a movie recommendation service. Developers do not check each particular rule, so only end users can say how relevant a rule is. For instance, a recommendation engine can suggest a "trending" movie without taking into account user preferences, because a lot of others have watched it at a certain moment and because it is found in most frequent sequences. We cannot say that this rule is inexplicable. Still, there is no clear relation to what movies the user has watched. Thus, this suggestion may not be interesting to the user.

Among the Boolean association rules that only indicate whether there is an association, we should single out the models that can be used to find *negative association rules* [4]. A negative association rule predicts an absence of the following set: *Y: {X => not Y}*. These are specific associations with a much narrower field of application than that of the classical Boolean rules. However, they should be mentioned, since negative rules can serve as an additional source of information when solving certain types of problems. Apart from that, there are also temporal association rules that take into account the fact that transactions are time dependent by nature. This can be important, for instance, when studying how users navigate across Web sites or when analyzing sales, taking into account the dates when merchandise was released to the market.

Associations can be found using a variety of algorithms, but before considering the algorithms we should take a closer look at the criteria we will use and how the rules will be interpreted. Let's call the rules *{X => Y}*, where *X* is the left and *Y* is the right part of the rule. *X* and *Y* may consist of several components. Association rules can be evaluated by the following criteria:

- *Length* (the number of components in the left part of the rule *(X)*)
- *Confidence* (how reliable the rule is, i.e. in what cases the right part of the rule will be the consequence of the left part)
- *Support* (the number of users, whose data correspond to both the left and the right part of the rule)
- *Lift* (how much better is the rule than a randomly suggested component)

2.4.1 Length—the length of a rule

Often, the *X* part of a rule includes 4–5 or more components. Researchers justify such length by the fact that it provides a more detailed description of each particular case. Such rules do seem to be more reliable since the conclusion stems from a whole range of events instead of just one or two. In reality, long rules are statistically less effective than short ones. In addition, if you fix the length of the left part, you will not be able to recommend anything to a new user until they accumulate a sufficient number of transaction components. This is also known as the "cold start" problem. That is why we think it is reasonable to use short rules for analysis but give them, for instance, a lower priority. We should only exclude rules like *{Ø =>Y}*, where the length of the left part equals zero.

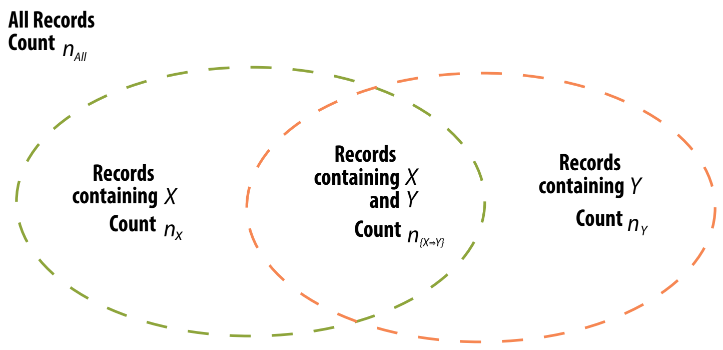


Figure 2: Calculating rule confidence and lift. Source: Altoros

2.4.2 Support—a rule's frequency in the general transaction database

Support indicates how often the sequence that the rule is based on occurs relative to all the transactions. If you look at Figure 2, you will see that *Support(X) = nX/nAll*. This is a key parameter used when looking for frequent patterns with the Apriori algorithm, for instance. It serves as the threshold criterion. This means, when creating rules, we do not take into account the sequences that have not reached a certain pre-set support threshold. On the one hand, this reduces the time used by the algorithm, since part of all possible combinations will be excluded. On the other hand, such a great support value will ensure statistical confidence of the rules. As a result, the recommendations will be more reliable. Still, there is a downside. Support is a relative indicator, i.e. a lot of rules will not be able to pass the threshold, if there are a lot of transactions, although it is clear that the number of occurrences is enough to ensure statistical confidence. This may result in the system only building statistically reliable but trivial rules that could be made without any calculations.

2.4.3 Confidence—validity of the rule

This is another key criterion for algorithms designed to find association rules. It can help to adjust the resulting number of rules. Confidence can be calculated by the following formula: *Support({X => Y})/ Support (X)*. We can also put it down as *n{X => Y}/nX* (see Figure 2). Confidence indicates the probability that the right part *(Y)* of the rule will be the consequence of the left part *(X)*. This example is the easiest to interpret: the greater the confidence, the better, and the more statistically reliable the recommendation will be.

2.4.4 Lift—the rule's value

Lift can be calculated by the following formula: *Support({X => Y})/Support(X)Support(Y)*. It indicates how much more often the right part of the rule occurs after the left part, compared to a randomly suggested element from the transactions set. For instance, if *Lift({X => Y})=10*, it is 10 times more likely that the *X* sequence will be followed by Y than by a random transaction. The higher the lift value, the better.

3. The Apriori Algorithm

In this section, we will clarify the basic principles of the Apriori algorithm. This will help you to understand what results could be expected. You will also get a better idea of how to distribute Apriori calculations, so that it can work on large data sets.

First of all, let's take a look at Figure 3 that shows all the possible candidates for a set of five transactions (*a*, *b*, *c*, *d*, and *e*). We can use the antimonotone support property to decrease the number of candidates, in the same way as the Apriori algorithm does. Let's assume that a set that includes *{a, b}* has a support value below the threshold and, accordingly, is not frequent. Then, in accordance with the antimonotone property, all of its supersets are also not frequent and should not be considered. This means, the whole branch, beginning with *{a, b}* can be pruned. This kind of heuristics helps to shrink the search space considerably.

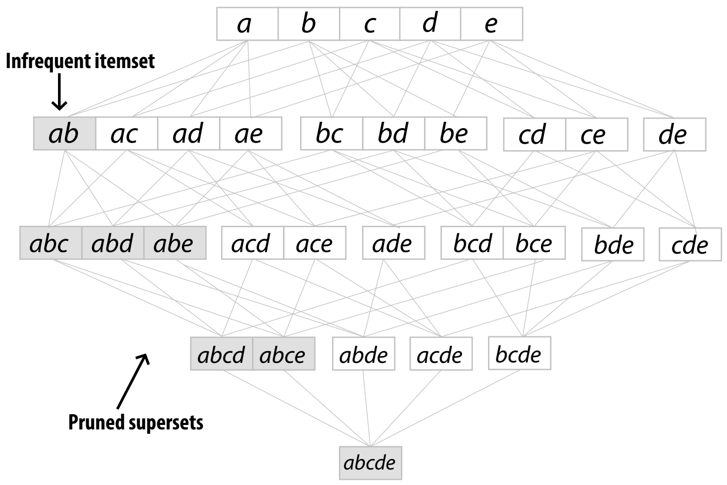
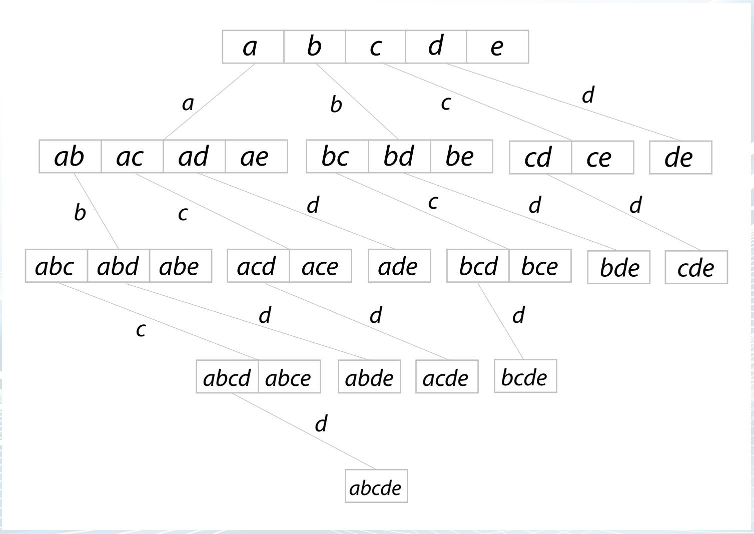


Figure 3: Decreasing the number of candidates in a sequence. Source: Efficient Implementations of Apriori and Eclat by Chrisitian Borglet

In the algorithm, this set of candidates will be expressed as a prefix tree. This helps to find transactions and calculate support values in a more efficient way. In addition, a relation of order is introduced to the transaction set. As a result, in our example, the prefix tree with five transactions will look like this (the order is *a>b>c>d>e* and the empty set is omitted in the beginning):



a b c d e
a b c d
ab ac ad ae bc bd be cd ce de
b c d c d d
abc abd abe acd ace ade bcd bce bde cde
c d d d
abcd abce abde acde bcde
d
abcde

Figure 4: A set of candidates in the form of a prefix tree. Source: Chrisitian Borglet

Order is introduced in a purposeful manner: the set of candidates is sorted by support, so that components with similar values are placed next to each other. This helps to find transactions in the tree faster, because there will be no large gaps inside, if we prune the branches with insufficient support values. Moreover, if we present the set in the form of a prefix tree, the only difference between a child and a parent will be in one element. Also, we will not create new branches, unless necessary, i.e. if a parent's support value is insufficient, we will not generate or analyze any children.

Now let's move on to the algorithm itself. The calculations are performed in iterations and can be described as follows:

Generating candidates is the first stage where the algorithm creates a set of candidates that consist of an *i* number of components (*i* stands for stage number). At this time, they include all the transactions with only one component. On later stages, candidates are generated based on the set of sequences from previous stages. To explain it in more detail, we can divide the process in two steps:

- *Merging*. Each candidate is generated by extending the frequent set of size (k-1) with one component from another set that also consists of (k-1) components.
- *Discarding redundant candidates.* Based on the antimonotone property, we delete all the sets from the list of candidates, if at least one of their (k-1) subsets is not frequent.

Counting the candidates is the stage where we calculate support values for each candidate that consists of an *i* number of components. At this stage, we eliminate the candidates whose support value is below the minimum set by the user. The remaining *i*-component sets are considered to be frequent.

Here is an example of how this works. Let us assume that after three iterations we received an ordered set of frequent sequences:

Frequent3={{a,b,c},{a,b,d},{a,c,d},{a,c,e},{b,c,d}}

According to the aggregation procedure, we can merge the first and the second set, as well as the third and fourth one, but there is no correspondence between the first two components for the fifth set. After aggregation, we will have:

Candidate4={{a,b,c,d},{a,c,d,e}}

After we delete all the redundant candidates, there will only be one set left:

Candidate4={{a,b,c,d}},

The subset *{a,d,e}* from the second set of *Candidate4* is not included into *Frequent3*.

3.1 Implementing a recommendation engine with the Apriori algorithm

3.1.1 Overview of input data

Now that we have an understanding of the principle behind the Apriori algorithm, we can return to solving the problem. Since the customer had no database (at the moment when we received the task and started working on it, the online store had not yet been developed), we had to use a test data set. For this purpose, we took the database of Netflix, a popular American Internet service. The goods our customer's online store was to sell were similar to those in Netflix's database.

Netflix's data consisted of multiple text files. Each text file corresponded to a certain movie and contained user IDs, as well as ratings they gave to various movies. The data included ratings on a five-point grading scale given to 17,770 movies by 480,000 users from December 1998 to December 2005.

In a classical case of collaborative filtering, data is presented in the form of a matrix of a size equal to *N*M*, where *N* is the number of users and *M* is the total number of movies. The matrix is filled using the following principle: *a[i][j]=rate*, if user i has watched movie j, and *a[i][j]=0* otherwise. As a result, we get a very sparse matrix where most of the values are equal to zero, because, on average, each user has watched 300 movies and the total number of movies is 17,770. This data format generates a very large initial dataset. This is why we have to convert the data into transactions. This means, we simply need to gather all the movies each user has watched (record the IDs of those movies for which we have data on users and ratings). The new converted file will look like this:

MovieID1, MovieID2, MovieID3, …

MovieID1, MovieID3, …

Each new line corresponds to a new user. User ID is of no importance—it is sufficient for us to know that these are different people. After we had converted the data into the required format, the size shrunk from 2 GB to 899 MB. In fact, we could use a local machine to find rules in 800 MB of data, but this was only a test dataset. In reality, we would have to deal with terabytes.

3.1.2 Technologies behind the engine

To solve our problem, we used Hadoop Streaming, the R language, and a well-known R-package for finding association rules called arules. Those who are more used to working with other languages, such as Java, can do the same with the Apriori implementation available in Apache Mahout (see FPGrowth).

First, we converted source files with movies into files with transactions (as already mentioned above). In order to do this, we created two scripts (Map and Reduce jobs).

To compare different types of pre-processing, we also implemented a modified version of the Map script that could take into account user ratings. In addition, we wrote a small script that searched for trending movies by the distribution of support density among all movies.

After the data pre-processing stage, we found frequent sequences among all transactions and calculated support values for each sequence. To detect frequent sequences, we used an existing function from arules in the Map script. To find the rules, we wrote two scripts (Map and Reduce jobs), since the arules package was not suitable for this task due to the peculiarities of the MapReduce paradigm.

Then we analyzed the resulting rules. These were initially based on three different data sources: the source database with movies, data on movies and their user ratings, and the database with deleted trending movies. We wanted to see how pre-processing affected the quality and quantity of association rules relative to the recommendation system we were to build.

In our research, we used the R language, a powerful tool for gathering statistics and analyzing data. The calculations were performed by four virtual machines (each with two cores and 8 GB of RAM). It took five hours to convert test data into transactions and ten minutes to find frequent sequences. The process of building association rules took about four hours.

3.1.3 Preliminary results and conclusions

The preliminary results generated by Apriori were disappointing. The association rules it had found were mostly trivial and could not be used to build an efficient recommendation engine. The resulting frequent sequences only included movies with high support levels. As a result, the association rules provided very few options. If we had a large database, the system would only recommend very popular movies. For example, if there were 100,000 transactions in the database and the support level was equal to 0.1, a movie would have to be watched 10,000 times to be included into a rule. It is statistically correct to recommend such movies, but rules of this kind are self-evident. Here is an example of a rule produced by this algorithm:

> *{Kill Bill: Vol. 2, Lord of the Rings: The Return of the King, Lord of the Rings: The Two Towers, The Bourne Identity} => {Lord of the Rings: The Fellowship of the Ring} or {Kill Bill: Vol. 2, Lord of the Rings: The Return of the King, Lord of the Rings: The Two Towers, The Bourne Identity} => {Pirates of the Caribbean: The Curse of the Black Pearl}.*

As you can see, we received combinations of popular movies that users are likely to watch without a recommendation. This does not mean, however, that all the suggestions were completely useless. Some of the rules were, in fact, helpful:

{Armageddon, Pearl Harbor, The Rock, Lethal Weapon 4, The Day After Tomorrow} => {Con Air}

Unfortunately, they were very few compared to the total number.

In order to demonstrate this, we can simply scan the right parts of the rules, to understand what movies such a system will recommend. For instance, here is a result received based on data without pre-processing:

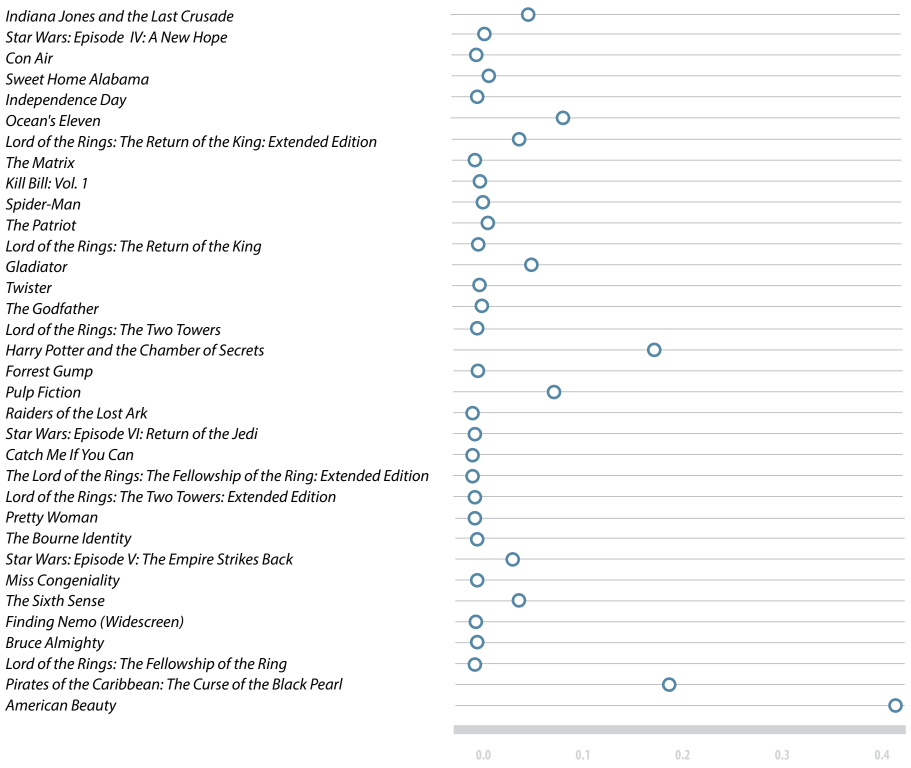


Figure 5: Movie recommendations based on data without pre-processing. Source: Altoros

As you can see, in more than half of all cases, the system recommends only three movies, "American Beauty," "Pirates of the Caribbean: The Curse of the Black Pearl," and "Harry Potter and Chamber of Secrets." Other movies are much less likely to appear in suggestions, which was unacceptable for the recommendation system that our customers wanted to build.

3.2 Improving the quality of recommendations

3.2.1 Data pre-processing

In an attempt to improve the quality of recommendations, we tried several types of data pre-processing. In particular, deleting trending movies and using additional information derived from user ratings. Deleting trending movies was supposed to bring more variety into the rules. We also expected that the system would be less likely to suggest a popular movie instead of a movie selected based on what a user has watched. Information on user ratings would better reflect user preferences, at least in theory.

When we calculated how rules intersect after different types of pre-processing, we saw the following ratios:

	Apriori	Apriori + Trend	Apriori + Rate
Rules (total)	*73,683*	*55,487*	*3,899*
General rules	*3,589 (0.05)*	*3,589 (0.06)*	*3,589 (0.92)*
Unique rules	*18,406 (0.25)*	*431 (0.01)*	*90 (0.02)*

Table 1: Ratios of unique and general rules to the total number of rules for each algorithm. Source: Altoros

In this table, the share of general and unique rules relative to the total number of rules received after each algorithm run is indicated in brackets. Unique rules are the ones that are only found in one sequence. General rules can be found in each of the three sets.

As you can see, filtering based on user ratings (after we had eliminated transactions with low ratings) had the greatest effect on the number of rules. Despite that, it did not improve movie suggestions. We managed to receive very few rules and only 90 of them were unique. This is not enough for a recommendation system to work. With 17,000 movies available, we expected to get at least one rule per movie.

3.2.2 Deleting trending movies

After we deleted trending movies (the ones that occurred much more frequently than others), we were able to receive a satisfactory number of rules. In order to evaluate their variety, we took a closer look at the right parts after pre-processing. It turned out that the initial dataset contained all the elements found in the right parts of these rules (that is exactly what we hoped to avoid). The distribution of movies in the right parts of the rules was as follows:

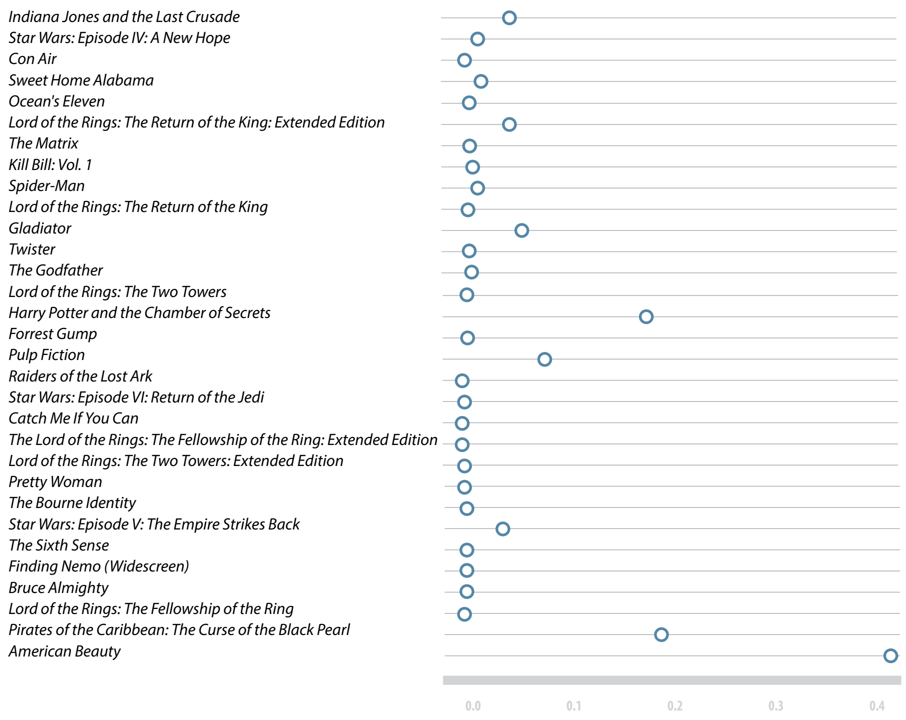


Figure 6: Movie recommendations after deleting trending movies. Source: Altoros

The picture has not changed much. Several movies are no longer included, but the distribution did not become more even. Almost 80% of all the rules include the same three suggestions. We can say that deleting trends has not helped.

3.2.3 Using ratings

The second approach to data pre-processing that we tried used transaction ratings. In theory, if we exclude the movies that users have already watched, giving them a low rating, the recommendations will more accurately reflect their preferences. This kind of pre-processing has resulted in a greatly reduced number of rules (the resulting number was 19 times smaller compared to what we would get without data pre-processing). We also expected that the quality and relevancy of recommendations would improve.

First we looked at the right parts of the rules, i.e. all the movies that the system could suggest. Again, they turned out to be a subset of the rules before pre-processing. However, their number decreased from 34 to 15. The distribution of the number of rules per movie from the right parts of the rules is shown on the diagram below.

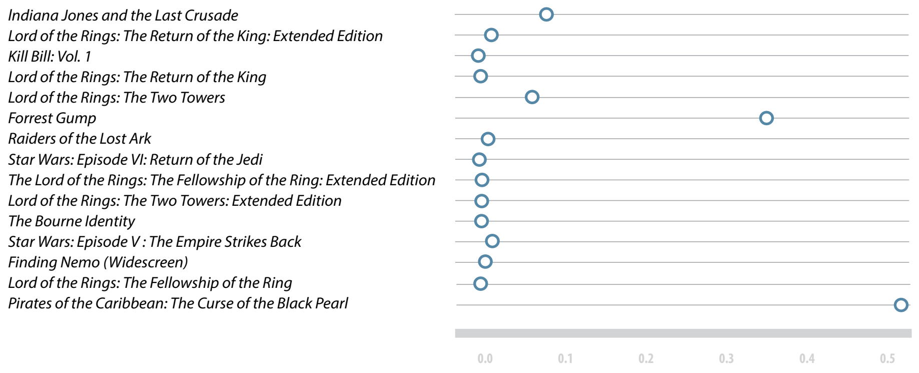


Figure 7: Movie recommendations taking into account transaction ratings. Source: Altoros

From Figure 7, you can see that the results have become even worse. Now two movies take up almost 90% of all the rules. This means user ratings did not help to improve the quality of the system. Deleting trending movies did not help either. Now let's consider other properties of the rules.

3.2.4 Using a rule quality filter

We decided that it would be enough to use only support and confidence values when calculating qualitative properties to assess the quality of the system. When building rules, we used the following quality filter: the rules with a confidence value below 0.9 and support value below 0.01 were not considered. As a result, the distribution of support density for different types of data (the initial dataset, the dataset without trend, and the dataset with user ratings) looked like this:

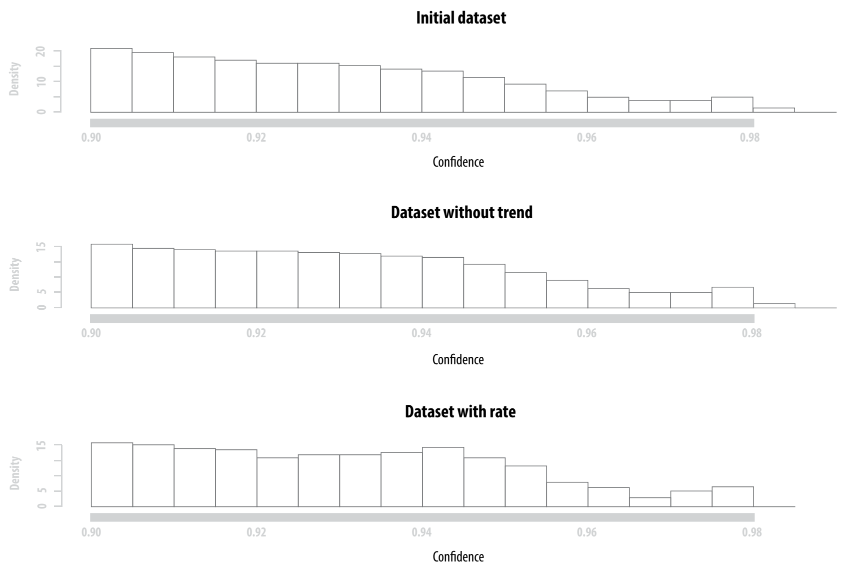


Figure 8. Distribution of support density for different types of data. Source: Altoros

Here you can see slight differences in distribution, but the ratio of high quality rules (above 0.95) to average quality rules (0.9 to 0.95) is the same despite pre-processing.

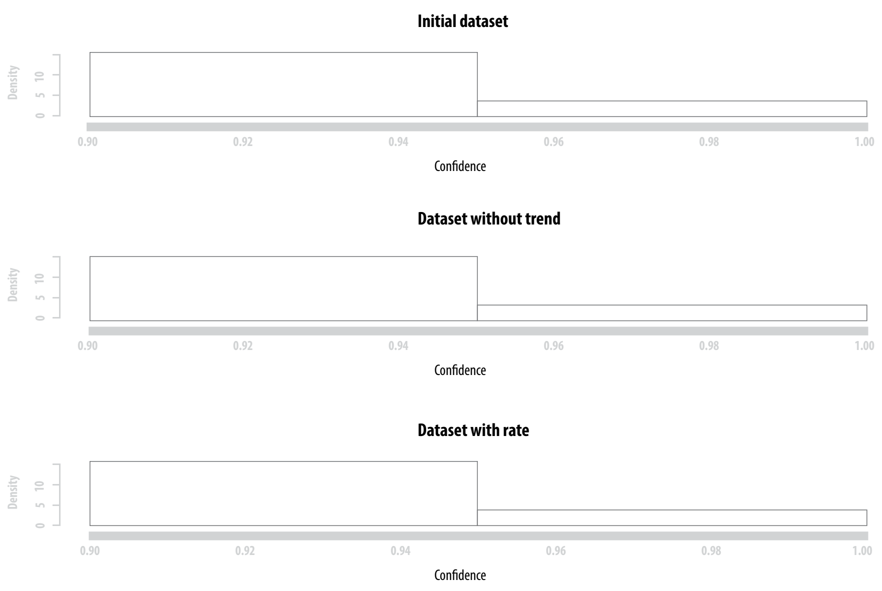


Figure 9. Distribution of support density for medium and high quality rules. Source: Altoros

3.2.5 Using high-confidence rules

In our last attempt to improve movie suggestions, we checked the degree of recommendation variety for rules with high confidence (0.95). In case of a movie database, the support value is not as important as confidence. As a result, we had 13,000, 11,000, and 700 rules per iteration accordingly (previously we had 74,000; 55,000; and 4,000). The distribution of the right parts of the rules was as follows:

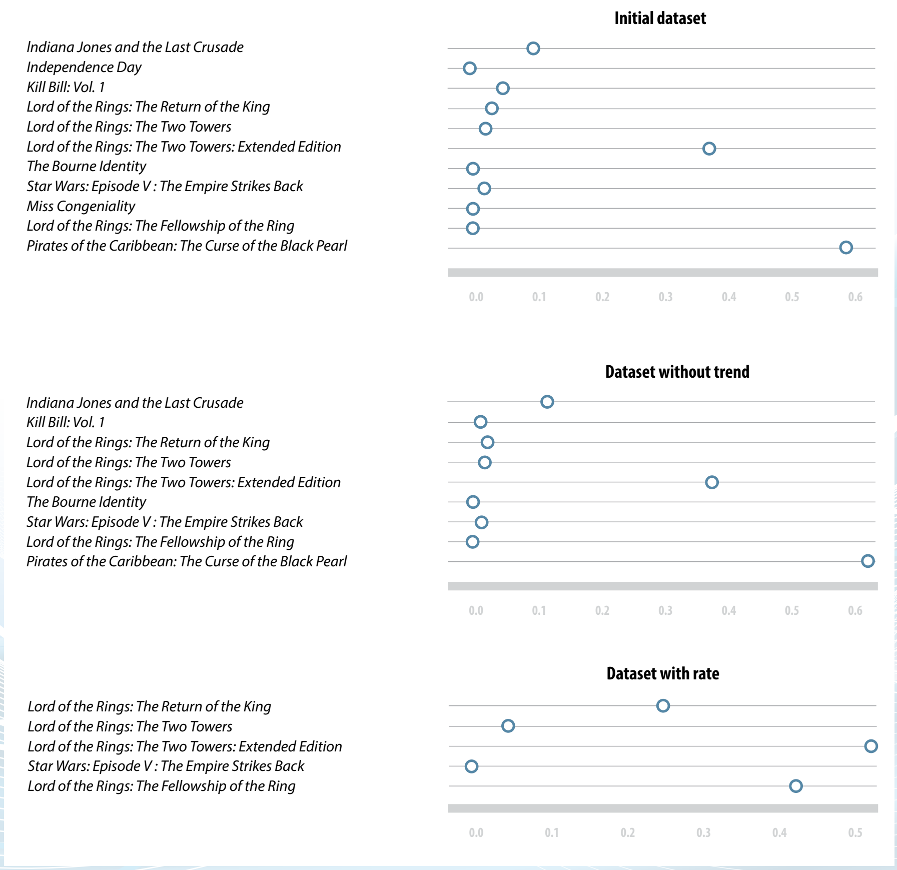


Figure 10. Recommendations taking into account the variety of high-confidence rules (0.95). Source: Altoros

The results were not surprising: we got 13,000 rules that mostly recommended three movies out of 11 or 700 rules for only five movies. This is not an acceptable solution for the problem.

3.3 Conclusions for using association rules

Although we have tried to improve the quality of recommendations using different data pre-processing approaches before building the rules, they did not change much. We could still try data aggregation (grouping similar transactions), but it seems to be unreasonable with our test data set. In theory, we could also try joining different parts of one movie, such as, for instance, the parts of the Lord of the Rings trilogy. Although, in our case, it would probably decrease the variety in the right parts of the rules, since the support value of such an aggregated movie would be much higher. This approach is only reasonable if the system deals with TV series. Still, it would be more appropriate to implement this at the moment the movies are being watched in the inner system rather than group movies by name afterwards.

Thus, despite the customer's expectations, the Apriori method could not be used to build a production-ready recommendation engine. Those few unique and high-quality rules that we received were absolutely not worth the effort. In the end, we managed to create an efficient Hadoop-based recommendation system using a different approach—clustering.

4. Clustering with Hadoop

Collaborative filtering methods, such as clustering, are perhaps the most popular among all machine-learning algorithms. They can work with data as it is, i.e. there is no need to create learning or test samples. Clustering algorithms are rather simple to implement and usually do not have much underlying theory, so even a person new to this field may understand them. What is more important, some of the clustering algorithms support Hadoop and the MapReduce paradigm and are suitable for big data applications.

When implementing clustering, we wanted to re-use the results of data pre-processing for association rules. Since we had previously converted all data into transactions for the *Apriori* algorithm, at that moment we had lists of watched movies for each user. All we needed to do was to group users with similar interests. Each group would have its own unique centroid. To build a recommendation, we would need to determine to what group a user belongs and suggest him movies watched by other people from this group.

Since we were going to use Hadoop distributed processing with the MapReduce paradigm, it was decided to use the k-means method for clustering. This is the easiest clustering method to implement with parallel computing. In addition, there are plenty of ready-to-use solutions that could potentially save us a lot of time. Below, we will explain the main principles of the *k-means* algorithm, how it was implemented for a recommendation engine, and the result—a production-ready system.

4.1 The k-means algorithm

4.1.1 Basic concepts of the k-means algorithm

k-means is the most well-known of all clustering algorithms. Despite its simplicity, this method is highly efficient and yields good results in most use cases. However, before describing k-means, we need to define three notions that we will be using a lot: points, clusters, and centroids.

- *A point* is one record from a table with data.
- *A cluster* consists of multiple points that the algorithm attached to one centroid.
- *A centroid* is a point that serves as the center of a mass cluster.

In the k-means algorithm, a centroid creates a unique cluster. This means, knowing the centroids, we can determine to what cluster a new point belongs.

When implementing the k-means algorithm, you must take into account two factors that might have a strong influence on the outcome:

1) Calculations performed by the k-means algorithm will give us cluster centroids. But these are only basic results. In fact, it is also possible to sort the data by clusters. In our use case, we only need to know the centroids.
2) Another thing to take into account when working with the k-means algorithm is that the number of clusters is a user-defined parameter. Unfortunately, unless it is clearly stated in your problem's specification, it is hard to evaluate how many clusters you need. The most popular way to determine this is to run the algorithm several times with different cluster quantity settings. Preliminary data analysis can also help. Yet, in most cases, it is easier to simply use the first way.

4.1.2 Pre-processing input data

Before using the k-means method on a data set, we must convert the latter into points of a multidimensional space. This means the data format must be as close to coordinates as possible:

1) It must be arranged into columns.
2) It must be homogenous.

Few researchers pay attention to this aspect. Due to its simplicity, the k-means algorithm will work even on a data set that does not suit it, but the results might be unsatisfactory. Let's look at each point in more detail:

1) If your data is arranged into columns, every record has the same length and all values are arranged in a particular order that does not change. For example, a table where different objects are listed as rows and the columns contain their properties. In this kind of data set, each object is followed by values of properties arranged in the same order. A data set that contains rows of customers and columns with their purchases does not fit into this pattern.

2) Homogenous, means that all your data is in the digital format and has the same scale. It would be inappropriate to cluster data of different formats, e.g. digital and categorical, or data where some properties have values close to zero while others have values above one million.

In addition to data pre-processing, before using the algorithm, we must set the initial coordinates for the centroids. In fact, it is possible to simply choose random points. A more careful approach—choosing initial points that are close to real ones—might save the algorithm some processing time. We decided start with random centroids.

4.1.3 Working principles of the k-means algorithm

As already mentioned before, the k-means algorithm is extremely simple. It operates in two stages:

1) First, it sorts data into clusters based on existing centroids. In order to determine which cluster a particular point belongs to, you need to calculate the distance between this point and each centroid and then choose the smallest value. The procedure is repeated for each data point until they are all placed into certain clusters.

2) The second step is to specify the centroids. To do this, we need to calculate new centroids after all the points have been sorted. In most cases, you only need to calculate the mean value inside each cluster. If Xi stands for the points inside a cluster (the cluster consists of N points), each point can be expressed by the following coordinates: Xi=(xi1, xi2, ..xiM). Here, M is he number of values in each data record. Given this, we can calculate a new centroid using this formula:

$$C = \sum_{i=1}^{N} \frac{Xi}{N}$$

We can also calculate it in coordinates:

$$cj = \sum_{i=1}^{N} \frac{Xji}{N}$$

3) The algorithm stops, when new centroids differ from the old ones by an insignificant pre-defined amount.

Below are two diagrams. The left one shows the initial data set with random centroids. On the right, you can see the clustered data and the centroids of the resulting clusters.

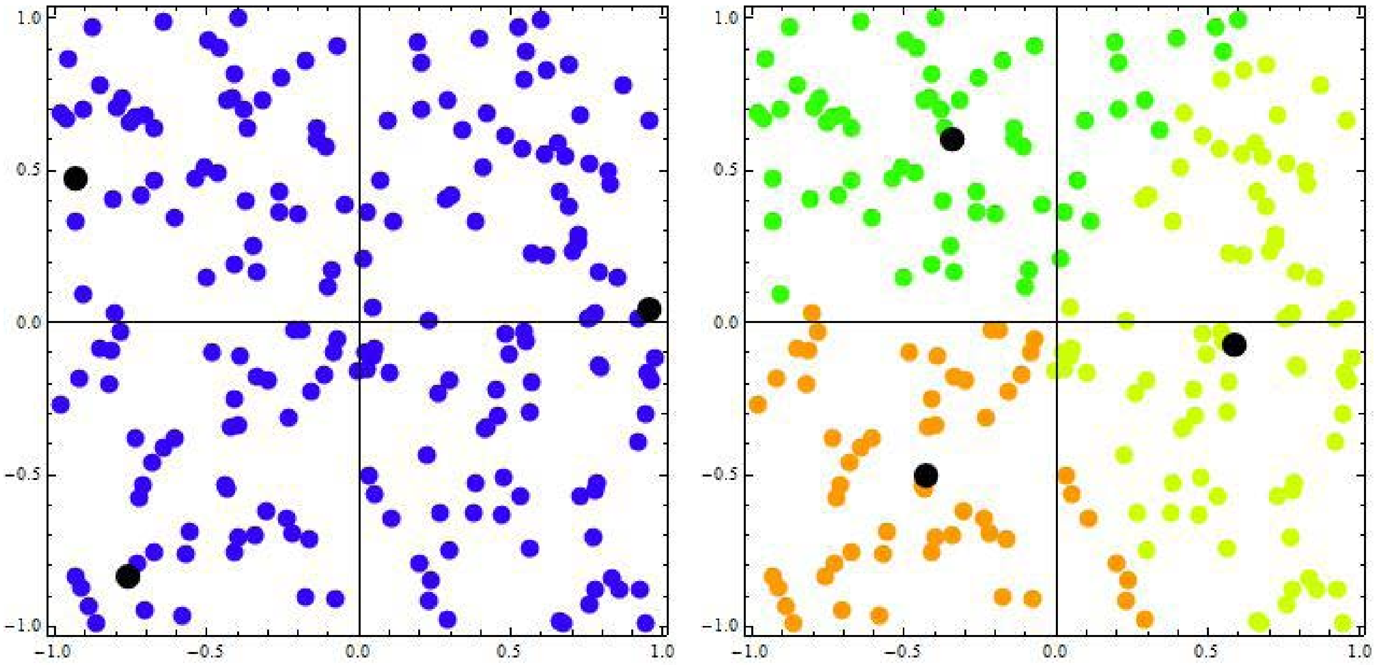


Figure 11. A sample data set with centroids before (left) and after (right) clustering. Source: Altoros

4.2 Implementing a recommendation engine with the k-means algorithm

4.2.1 Peculiarities of implementation

When implementing the algorithm from scratch or using an existing solution, you should pay attention to the following two aspects. The first one is calculating the distance between a point and a cluster centroid. In most cases, you can do this using regular Euclidean metric or Mahalanobis distance, but some types of data require other metrics. In fact, two approaches exist. You can convert your data into a standard format and use Euclidean distance. If that is too hard to do, you can select another metric based on your data set.

The second aspect you should pay attention to is cluster centroids. If your data is standard, a centroid can be calculated as the mean value of all the points in a cluster. The formula is provided in this document. However, if your data is in a specific format (the points have varying values/priority), you will have to calculate each centroid as the center of mass for the points in a cluster.

On the one hand, unconventional data formats may require additional efforts to convert it to the standard format. On the other hand, this might simplify or speed up the algorithm.

4.2.2 Clustering test data

4.2.2.1 Data pre-processing

Our initial data set consisted of multiple files. Each file contained information on one movie arranged in the following order: CustomerID, Rate, Date. To find association rules, we converted this data into transactions. After that, for each user, we had a list of movies they have watched. At the same time, this helped to shrink the size of our test data set from 2 GB to 800 MB.

An implementation of the k-means method is available in Mahout, an Apache project that provides scalable implementations of distributed machine learning algorithms. However, the format of transactions was not suitable for Mahout, unless modified. In order to use Mahout, we would have to convert the data into a sparse matrix *A* of size *N*M* (where *N* is the number of users and *M* is the total number of movies in the database). In this matrix, *a[i][j] = 1*, if user *i* has watched movie *j*. In all other cases, *a[i][j] = 0*. Our test data set contained 17,770 movies (in reality, we would deal with a lot more) and, on average, each of the 480,000 users watched 700 of them. So, if we converted this data set into a matrix, we would only get about 4% of useful information for a huge amount of information.

This is why we decided to avoid converting data for Mahout and created our own implementation of k-means for Hadoop Streaming using the R language.

4.2.2.2 Implementation of the k-means algorithm

We wanted to use the data that we had previously converted into transactions for the Apriori algorithm (see above). As you remember, we had written two simple scripts for Hadoop Streaming using the R language and it took several hours for them to convert the data. Our problem was that the standard Euclidean distance formula was not suitable for this format. The following example illustrates this. There are two records:

$$Cust1 = (2, 6, 8)$$

$$Cust2 = (6, 11)$$

They indicate that the first customer watched movies with ID numbers 2, 6, and 8, while the second one watched movies 6 and 11. Given this, we can calculate the distance from one user to another using the following formula:

$$Length\ (Cust1) + Length\ (Cust2) - 2\ Length\ (Cust1 \cap Cust2)$$

In our case, the difference between two users will be 3+2-2=3. If we also extract a square root from this value, we will get the value that is equal to the Euclidean distance between two vectors of 17,770 in length. These vectors have 1 on the positions presented in sets *a* and *b*, and 0 everywhere else.

Now we can move on to the cluster centroid. We cannot calculate it as a mean value regardless of the data format, because the result might turn out to be outside our data space. As an illustration, let's assume that there are two vectors:

$a = (1, 0, 1, 1)$

$b = (1, 0, 0, 1)$

They indicate that user a has watched the first, third, and fourth movie out of four in the database while user b only watched the first and the fourth one. If we calculate the centroid of the cluster that consists of these users as a mean value, the resulting point will have the following coordinates: (1, 0, 0.5, 1). Obviously, it does not belong to our data space, because inside this data space only two values are possible—0 or 1 (indicating whether a user has or has not watched a movie respectively).

However, if we assume that our data reflects how likely a user is to watch a movie, all values from 0 to 1 will belong to our data space. We could use this method to calculate the cluster centroid, but the data would have to be in the form of a sparse matrix or we would have to store both the locations of values not equal to zero and the values in those locations. In any case, this would increase the amount of data to store and process, which is undesirable.

We can make another assumption and consider any likelihood value greater than a certain threshold to be equal to 1. Accordingly, in case the value is lower than this threshold, it will be equal to 0. In other words, we will sample the data space again. However, this time we will have to put up with errors. In addition, the resulting point will not be the true centroid of the cluster but an approximation. In this scenario, we can also store the centroids as a list of positions with values not equal to zero. This fit the general approach we wanted to use.

To implement the k-means algorithm, we wrote two scripts with Map and Reduce jobs for Hadoop Streaming using the R language. We also made another small script that helped to launch jobs in iterations and coordinated the operation of the algorithm. It took around 60 hours and four virtual machines (with double-core CPUs and 8 GB of RAM) to sort our data set of 500,000 points into 2,000 clusters. The number of centroids was selected after running the algorithm on a test data set several times.

4.3 Results and analysis for the k-means method

Below is a chart that shows how many users were included into each of the 2,000 clusters produced by the algorithm.

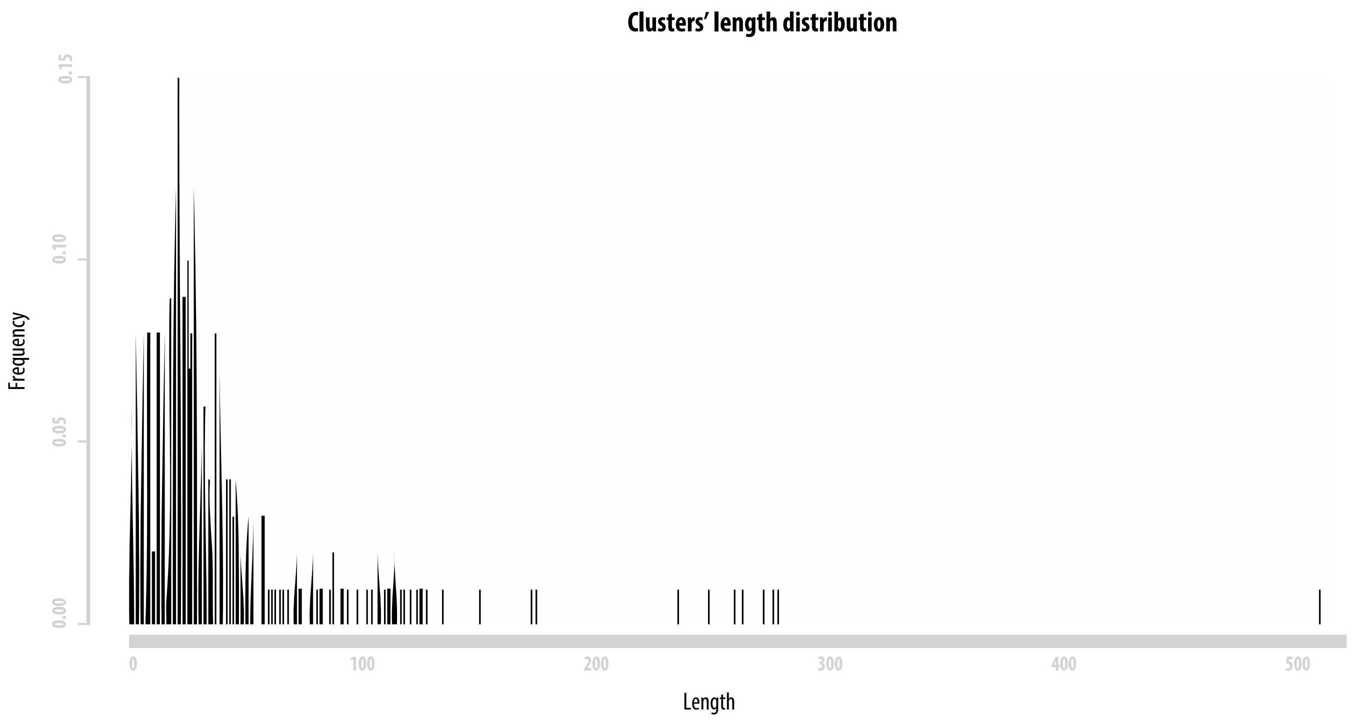


Figure 12. Distribution of clusters by size. The x axis indicates cluster size. The y axis indicates how often a cluster of this size occurs relative to all other cluster sizes. Source: Altoros

As you can see, clusters of around 50 users are the most frequent. If we would get many clusters of ten or fewer users, this would indicate that clustering was not sufficiently optimized (too many clusters). Large clusters (of 300 and more users) would be too few, resulting in rough recommendations.

Each cluster contains users, whose tastes are as similar as possible—lists of watched movies inside one cluster are more similar than lists of watched movies taken from different clusters. As we have already mentioned, a cluster is formed around a unique centroid. The lists of movies that comprise the centroids of each cluster have varying lengths (10–250 movies). Clusters with centroids that have short movie lists allow for making recommendations to new users, who have watched few movies. As new users watch more movies, they will be moved to other clusters with more precise recommendations.

Recommendations are made in the following way:

1) First, the system checks a user who needs a recommendation and includes them into a cluster, i.e. it finds the centroid closest to this user.
2) Then, the system tries to make a recommendation based on the differences between the centroid and the user. From the list of movies attached to the centroid, it selects the ones that the user has not watched yet. Since the number of recommendations you can give at a time is limited, if there

are more than necessary, the recommendation engine randomly displays as many as required.

3) If a user has watched all the movies from the centroid's list, the system selects a random user from the same cluster and makes recommendations using the method described in the first two points.

Below are several examples of users and recommendations selected for them:

Movies watched by a user:	Recommendations:
"Independence Day," "The Fast and the Furious," "The General's Daughter," "Indiana Jones and the Last Crusade," "Miss Congeniality," "Con Air," "Air Force One," "Men of Honor," "John Q," "Swordfish," "Pearl Harbor," "Enemy of the State," "Ghost," "Entrapment," "The Patriot," "Lethal Weapon 4," "A Few Good Men," "Twister," "The Green Mile," "Armageddon," "The Day After Tomorrow," "The Rock," "I," "Nothing to Lose," "Miracle on 34th Street," "Rules of Engagement," "The Negotiator," "First Knight"	*"S.W.A.T.," "Men in Black II," "Lethal Weapon 3," "Gone in 60 Seconds," "Along Came a Spider," "Lethal Weapon 2," "Double Jeopardy"*
"Lord of the Rings: The Fellowship of the Ring," "Collateral," "Lord of the Rings: The Return of the King," "Garden State," "Van Helsing," "Mystic River," "X2: X-Men United," "Tomb Raider," "Lara Croft: Tomb Raider: The Cradle of Life," "Big Fish," "Terminator 3: Rise of the Machines," "The Bourne Identity," "The Bourne Supremacy," "Gothika"	*"Ocean's Eleven," "Minority Report," "Gangs of New York," "Lord of the Rings: The Two Towers," "The Matrix: Reloaded"*

Table 2. Clustering: Sample recommendations based on watched movies. Source: Altoros

This kind of system requires periodical tuning. In time, you will have to adjust the number of centroids, because the number of movies and users in the system will grow. At some point your cluster centroids will no longer reflect the real distribution of users by their preferences. Perhaps, you will need more or fewer clusters. In order to make this adjustment, you will have to analyze your data and find the optimal number again. Unfortunately, it is rather difficult to do this automatically.

There are two ways to determine whether your system has become outdated:

1) You can gather user feedback—your users can evaluate how helpful the recommendations are. When the quality of the recommendations falls below an acceptable value (that satisfies the owner of the online store), the system needs to be readjusted.

2) You can watch the number of new movies and users in the database. As soon as they reach a certain threshold, you will have re-adjust the recommendation system. In this document, we only provide general advice. In reality, to calculate these threshold values, you will need to conduct a research on an operating online store.

5. Final Comparison and Conclusions

When building a Hadoop-based recommendation engine, we tried out two approaches: the Apriori algorithm (for building association rules) and the k-means method (for clustering). The results can be summed up in the following table:

	Association Rules (Apriori)	Clustering (k-means)
Number of recommendations	Just 74,000 recommendations for 480,000 users	More than 10 recommendations per single user
Recommendation quality	Unsatisfactory, trivial recommendations	Good
Implementation with Hadoop/MapReduce	Possible	Possible
Processing time	~4 hours	~60 hours
Additional maintenance	Rules have to be re-calculated periodically, using new available data	The number of clusters has to be tuned periodically

Table 3. The Apriori algorithm vs. k-means in a movie recommendation engine. Source: Altoros

Apriori:

Despite the positive expectations, the Apriori algorithm turned out to be too rough to generate useful recommendations. Most of the rules were trivial and had low variety. The number of movies that the system could recommend was rather small while most suggestions included less than 100 movies popular among all users.

k-means:

The clustering method was much more useful. Based on simple clustering of the initial data set, you can get a working recommendation engine that takes into account user preferences and provides more recommendations than you actually need. This system can be further improved, if we take into account user ratings given to movies and assign weight to each point (the higher the rating, the more weight a movie receives). Then the formula for calculating the center of the cluster will change, since the points in the data space will no longer be equal.

However, after some time, the recommendation engine based on clustering will require additional adjustments. Since our prototype was implemented focusing on Hadoop, calculations on a larger data set will not be a problem. In addition, to define the centroids more precisely, you will probably need several iterations with the algorithm. This is because we start with centroids that are already approximations of real ones. Initial tuning of the system and re-tuning it when the number of clusters changes are the most time-consuming stages.

All of the said above demonstrates that clustering with k-means seems to be a better option for building a movie recommendation engine. Still, creating association rules with the Apriori algorithm may work for other purposes, such as credit card and insurance fraud detection, finding the reasons for failures in telecommunication networks, etc. (see section 2.3 *Association analysis and big data*).

6. References

[1] I. A. Chubukova. Data Mining. Study Course.

[2] Raymond Chi-Wing Wong, Ada Wai-Chee Fu. Association Rule Mining and its Application to MPIS .

[3] A. Pavlenko. Lectures on Data Analysis. 2012

[4] V. I. Gorodetsky, V. V. Samoilov. Association and Causation Analysis and Association Bias Networks. Tr. SPIIRAN, 9 (2009), 13–65.

[5] Chrisitian Borglet. Efficient Implementations of Apriori and Eclat

[6] Vattani., A. (2011). k-means requires exponentially many iterations even in the plane. *Discrete and Computational Geometry*. 45(4): 596–616. doi:10.1007/s00454-011-9340-1

[7] Frahling, G.; Sohler, C. (2006). A fast k-means implementation using coresets. Proceedings of the twenty-second annual symposium on Computational geometry (SoCG).

[8] K-Means Clustering at the Apache Mahout project documentation

[9] Large Scale Machine Learning and Other Animals, a blog by Danny Bickson

[10] The UH Data Mining Hypertextbook by Rakesh Verma

7. About the Authors

Sofia Parfenovich *is a Data Scientist at Altoros with 3+ years of experience in solving complex data science problems and creating algorithms for Hadoop-driven applications. Having a strong background in math, computer science, artificial intelligence, and machine learning algorithms, Sofia also used to teach computer science at Belarusian State University.*

Altoros *is a big data and Platform-as-a-Service specialist that focuses on system integration for IaaS/cloud providers, software companies, and information-driven enterprises. The company provides consulting and managed services with Cloud Foundry PaaS, multi-cloud deployment automation, and complex Java/.NET/Ruby architectures that make heavy use of SQL/NoSQL/Hadoop clusters. Altoros employs 250+ senior and mid-level engineers across 7 countries, including one of the largest pools of Cloud Foundry expertise on the market. Fast-track services (4–12 weeks) include single-cloud deployments of Cloud Foundry and integration with relational and NoSQL databases. For more, please visit www.altoros.com or follow @altoros.*

Liked this white paper?
Share it on the Web!