ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Academic year 2013-2014

Mining 35 Million product reviews with MapReduce

Semester Project in Communication Sciences

Author: Amato Van Geyt

Professor: Matthias Grossglauser
Contact Assistant: Lucas Maystre

Laboratory for Communications and Applications

Contents

1 Introduction

1.1 Context of the Project

In the last 10, 20, years our society has seen a lot of changes in what the average customer buys, but even more how. Traditional stores have lost a lot of terrain against the still growing number of e-commerces. The main reasons of this can be found not only in the ease of buying, the huge and easily searchable collections, but also the ease of checking the opinions of other users on a particular product, and as such deciding whether you want to buy that product or rather not.
A more recent phenomenon or even hype is called Big Data. The absolute size of data, on-line and off-line, keeps increasing without a stop. Datasets are getting bigger and bigger and many more insights can be extracted from them. Recommendation systems, smart placing of products in supermarkets, user profiling are only a small fraction of the countless applications. The huge amount of this data is also its biggest enemy. Saving the data and processing the data among others have become harder and harder. It is for this purpose that completely different architectures in software and architecture have been created to cope with these amounts of data; Spark, Hadoop, Storm, MapReduce and many more.

It is in this setting that the Laboratory for Communications and Applications (LCA) of the École Polytechnique Féderale de Lausanne (EPFL) obtained 2 different resources: a huge dataset of about 35 million reviews from the leading e-commerce site `www.Amazon.com` from 1995 to 2013 obtained from Stanford combined with a Hadoop MapReduce computing cluster. The combination of those two, a huge amount of product review data and a powerful computing unit made for demanding parallel computations, gave us the idea that this would lead us to new insights on product reviews.

1.2 Goal of the Project

The goal of this project was open-ended and not set in stone. There were some initial ideas for analyses we could conduct, but in the end there was still a lot of choice left for us to go in any direction which we preferred. One possibility might be to use Natural Language Processing (NLP) tools to extract and analyse various language features from the review texts (sentiment analysis, rating prediction,...). Another possible idea would be to build create co-rating graphs from the data. This is a graph in which the nodes represent users and edges connect users who have rated the same product. We could then try to extract all kinds of information from this graph such as strongly related products or tightly knit communities. A last idea would be to look to the evolution of ratings; is there an overall change of how products are reviewed? How do the reviews of a user change when he gets more experienced? These are all possibile ideas for us to try to implement while makiing use of MapReduce.

1.3 Technicalities

The Hadoop MapReduce computing cluster which we used consisted of up to 48 machines, each with a quad-core processor and 16GB of RAM and over all of those terabytes of replicated HDFS storage. This cluster was accessible through a private user-account on `icsil1-hdp1.epfl.ch`.

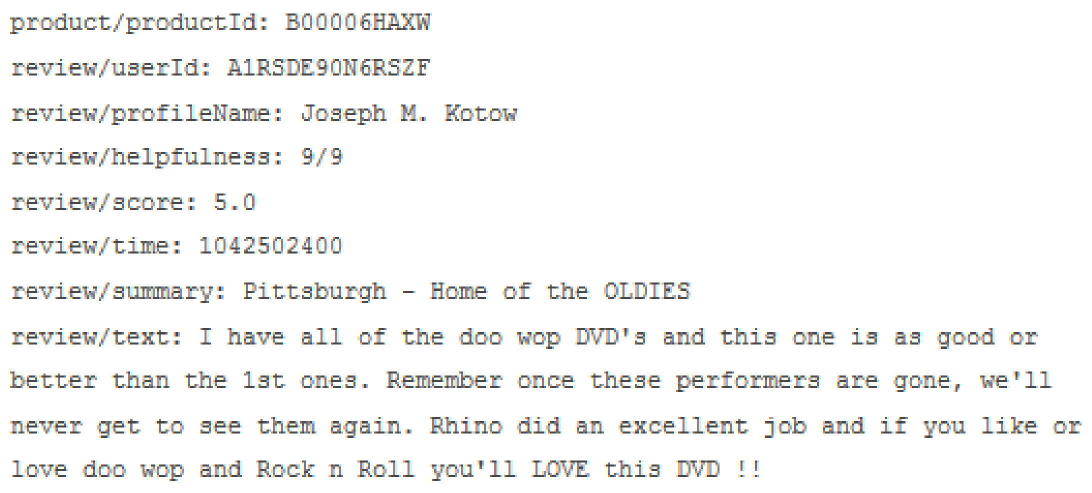
The properties which made us choose for Hadoop MapReduce for these analytics were mainly that very large files could be saved distributed over all the nodes by using the Hadoop Distributed File System (HDFS) and that MapReduce is a programming model made to run very well in parallel on multiple cores while still having a very simple and logical way of working. More details on the workings of Hadoop and MapReduce can be found on `http://hadoop.apache.org/`. Understanding these workings is not necessary to interpret the results presented in this report but are useful to understand the ways how they were obtained.

2 The Dataset and its Representation

The dataset containing the 35 million Amazon Reviews has a well-defined structure; it containes 35 million text blocks, separated by an empty line, with each text block representing one review as shown in Figure 1. The `productId` and `userId` are unique identifiers for the product and user and are main parts of the reviews as we'll see later. The number representing the time is in Unix-time and such requires an intermediate step in order to view the exact date this review has been made. Unfortunately it seems that only the year, month and day were saved, making it impossible to analyse the relationship between reviews and for example the hour in which they were created.

The biggest problem with this was though that MapReduce is by default only capable to process datasets line for line making it impossible to study the correlation between different fields of a review, basically disallowing any real data analytics. After changing the default delimiter from `\n` to `\n\n` and as such looking for the empty line instead of the end of the line we solved this issue locally but for some very weird reason this did not work on the cluster. As we realised that solving this bug would be needlessly time-consuming, we decided to create a work-around by transforming the dataset into JSON-format and as such putting every single review on one single line while combining this with using a format which allows for easy parsing due to the availability of many libraries involving JSON (the one we used was `com.google.gson.Gson`). The transformation was done by using a script created by the responsible assistant Lucas Maystre and can be found in the repository under `utils/process.py`.
In order to allow easy programming and processing on this dataset we created a container class called `AmazonReview.java` which we could easily construct by using the parsing method of `Gson` and later on quickly access without using get-methods.

A final comment to make; as the size of this whole, transformed, uncompressed, dataset exceeds 30 GB (32.310 GB more precisely) it made a lot of sense to keep this whole dataset on the HDFS (`all.txt` and only download a subset of it, preferably the smallest one, for local testing of the code before running it on the whole dataset. This smallest subset we could find was contained in the file `Jewelry.txt` which was about 30.80 MB.
The exact amount of reviews seemed to be 34,686,770 in the whole dataset and 58,621 in `Jewelry.txt`



```
product/productId: B00006HAXW
review/userId: A1RSDE90N6RSZF
review/profileName: Joseph M. Kotow
review/helpfulness: 9/9
review/score: 5.0
review/time: 1042502400
review/summary: Pittsburgh - Home of the OLDIES
review/text: I have all of the doo wop DVD's and this one is as good or
better than the 1st ones. Remember once these performers are gone, we'll
never get to see them again. Rhino did an excellent job and if you like or
love doo wop and Rock n Roll you'll LOVE this DVD !!
```

Figure 1: The representation of an Amazon-review in the dataset

3 Basic Analytics

As we first of all wanted to get familiar with the infrastructure which had to our disposal we started with some very basic MapReduce-jobs who computed some basic statistics. We used MapReduce to get our results in columns in our final text file and then in the end we used Python (and the library `matplotlib` in specific) to plot and visualise our results. All the code, its documentation on how to use it and how to reproduce the results, the scripts, the final results and its plots can all be found in the repository.

3.1 Score Distribution

The very first MapReduce-job that we programmed was one to compute the score distribution over all the reviews. This was done in a very simple manner:

Mapper Firsty we parse the review into an `AmazonReview`-object, then we extract the score from it and this score we use then as `outputKey`. The corresponding `outputValue` is always equal to 1.

Reducer In the Reducer we set as `outputKey` its `inputKey` (and as such the score the reducer represents) while as `outputValue` we simply take the sum over all the `inputValues` received in that Reducer.

As such, in the end we end up with a text-file with 5 lines: every line represents a score and the times that score occurred among the reviews in the dataset. The code can be found in the package `amazon.scorecount`, the resulting text-file as `results/all/ScoreCount.txt` and the corresponding plot, which is shown in Figure 2, can be found as `plots/all/ScoreCount.png`. The exact values of the points in the graph are shown in the table below.

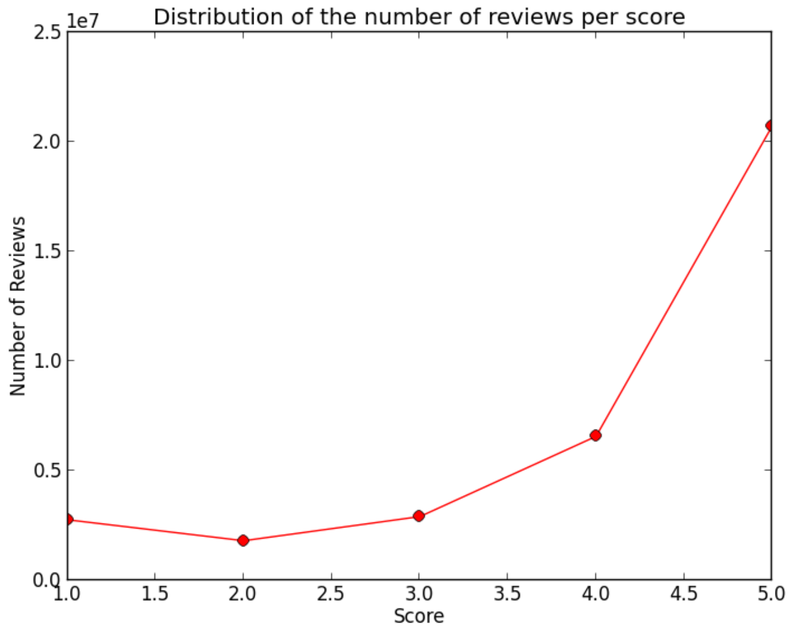


Figure 2: The distribution of the review-scores over the whole dataset.

1	2,746,559
2	1,791,219
3	2,892,566
4	6,551,166
5	20,705,260

As we can clearly see there is very high tendency for users to give a product the maximum score in its review. The amount of scores for 5 is higher than all the others combined. It also shouldn't surprise too much that the rating 1 breaks the tendency seen in the review scores (the lower the score the less reviews) as users tend to give extreme ratings more easily: a single bad experience with a product is often enough reason to give it the minimum score.

3.2 Length Distribution

The next thing we did is similar to the previous statistic but then we didn't look to the score of the review but to the length of its text, what would it tell us? As there are two different ways of measuring the length of a text, the number of characters and the number of words, we implemented two different options between which could be chosen by passing the corresponding argument through the command line. Also, because the length of a text in characters can quickly become very high we also demanded a precision-argument which should be an integer. This number is then used to define the bin-size of the bins in which the lengths will be put. If for example the precision is 10, then only values for the length which are a multiple of 10 will be seen, this is done by rounding values down to the closest bin-limit. In the end we saw that the results with precision 1 were still more than ok so we went further with these results.
The coding was again straightforward:

Mapper First we parse the review into an `AmazonReview`-object and determine the values of our global variables `type` (`word` or `char`) and `precision` (an integer). If the `type` is `word` then we split the String `review.text` into an array with a non alphanumerical character used as splitter and then we calculate the length of this array. If the `type` is `char` then we just calculate the length of `review.text`. In the very end we round our found value down, based on the precision and it is this value that we use as `outputKey` while we use the constant 1 as `outputValue`.

Reducer This Reducer is completely the same as the one used in Score Distribution.

The code can be found in the package `amazon.length`, the resulting text-files as `results/all/LengthWord.txt` and `results/all/LengthChar.txt` the corresponding plots, which are shown in Figures 3 and 4, can be found as `plots/all/LengthWord.png` and `plots/all/LengthChar.png`.

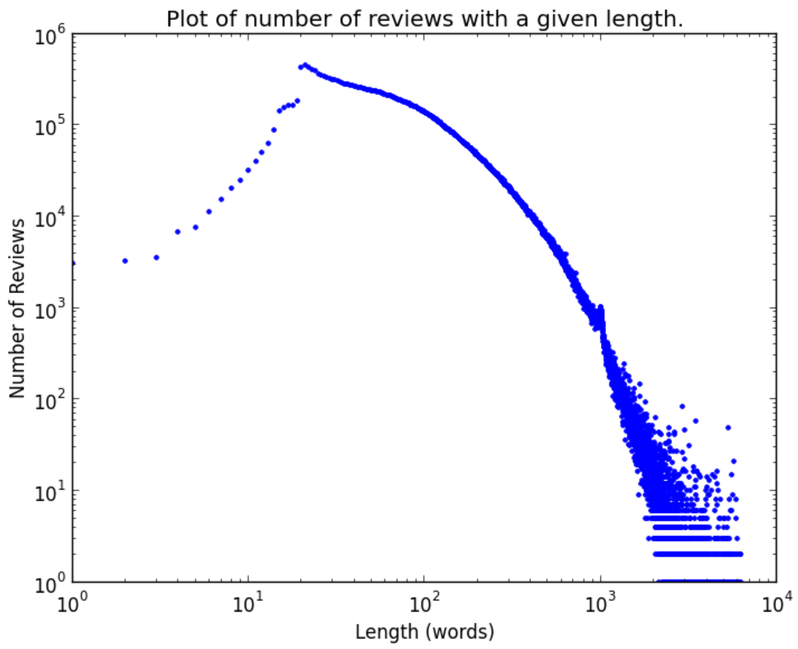


Figure 3: The distribution of the amount of reviews in function of its length (words)

We can clearly see a tendency here: in the beginning the number of reviews increases with an increasing length, but after reaching its peak it declines rapidly although it still has a very heavy tail with some reviews even being as long as 50,000 characters or 7,000 words. As this plot is logarithmic in both the x- and the y-axis and we can see a somewhat linear tendency for higher lengths we are inclined to think that we see the power law here at work. For the completeness we also provided the cumulative distribution plots made from these 2 results and showed them in Figures 5 and 6.

3.3 Score vs. Length

After having done these simple counting analytics of score and length we'd like to combine the two of them. How does the score of a review changes when it becomes longer? In order

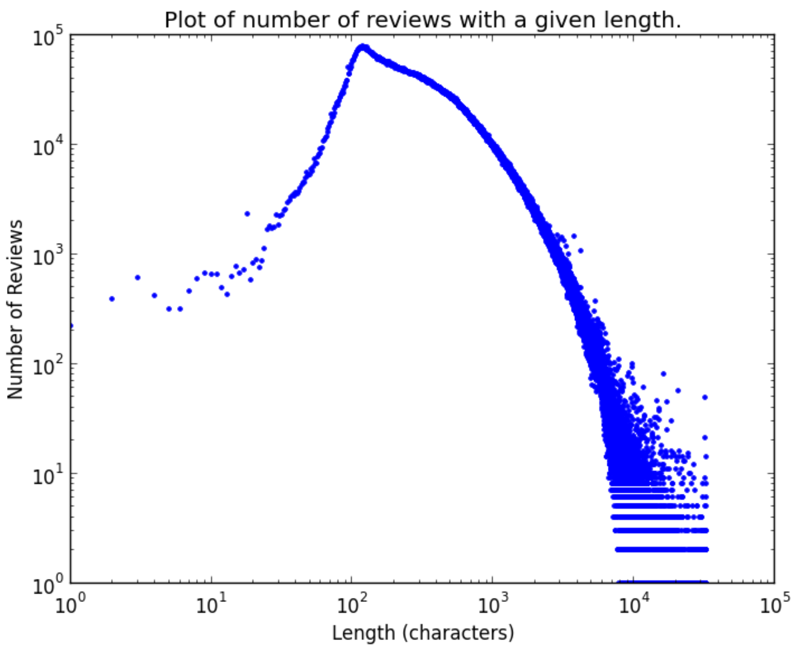


Figure 4: The distribution of the amount of reviews in function of its length (characters)

to find this out we created the following MapReduce-job with again the possibility to express the length in characters or words.

Mapper First we computed the length of the review in exactly the same way as in the previous job. The precision wasn't constant though as the length is plotted in logarithmic scale and as such the density of data-points would vary a lot between low and high values. In the previous job this wasn't much of a problem but as we now want to calculate the average of the score for a given length we wanted our binning to be a bit more efficient in order to create a smooth plot. We came up with the precision of $10^{\lfloor log_{10}(length)-1\rfloor}$ with as minimum bound 1 (such that 1 would have precision 1 and not 0). Because of this there are 100 data points between every 2 consecutive powers of 10 except between 10^0 and 10^1. The length of the review, rounded by using the precision calculated by the previous equation, was set as `outputKey`. Its corresponding `outputValue` though wasn't 1 any more, but the score of the review given corresponding with the computed length. This one is simply taken from the constructed `AmazonReview`-object.

Reducer The Reducer is again very similar to the previous job, but this time the `inputValues` aren't summed, instead, their average is calculated and outputted if there are at least 5 `inputValues` for that specific length which we got in as `intputKey` and output unchanged.

The code can be found in the package `amazon.scorelength`, the resulting text-files as `results/all/ScoreLengthWord.txt` and `results/all/ScoreLengthChar.txt` the corresponding plots, which are shown in Figures 7 and 8, can found as `plots/all/ScoreLengthWord.png` and `plots/all/ScoreLengthChar.png`.

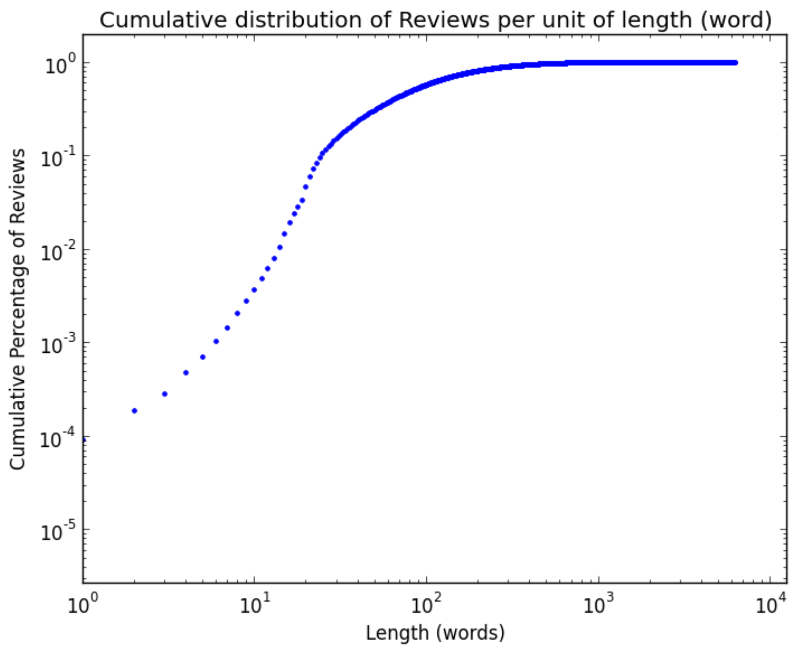


Figure 5: The cumulative distribution of the amount of reviews in function of its length (words)

What we can see on these graphs is that initially the average score is the same for very short reviews. However, once the length reaches +/− 15 words or +/− 100 characters its average score starts decreasing while in the very end, before becoming too messy to interpret, it slightly increases again. This somewhat weird behaviour makes us think that short reviews are often more radical and as such easier rated as a 5, while longer reviews contain more nuance and as such are more moderate in their review score. In the end, when reviews tend to get extremely long this more extreme behaviour can show up again: very long rants or eulogies on a certain product tend to appear we think but it is hard to say as at that point the number of data-points became negligible small..

3.4 Length vs. Helpfulness

We now have an idea about the relationship between the length of a review and its score, but what about its helpfulness? Is there some kind of TL;DR (Too Long Didn't Read) or KISS Keep It Short & Simple) phenomenon? How does the helpfulness of a review change when reviews get longer and longer?

The helpfulness of a review is stored as a combination of two numbers: the total amount of users who found the review helpful and the total amount of users who found the review helpful OR unhelpful. This means that the lower the ratio between those two numbers the less helpful the review was judged and vice versa. The absolute values only tell us how popular this particular product is and as such we shouldn't give too much attention to it. As such, the metric we use to define the helpfulness of the review is simply the ratio of the two aforementioned integers.

Mapper The Mapper of this job is completely the same as the previous one except that this time

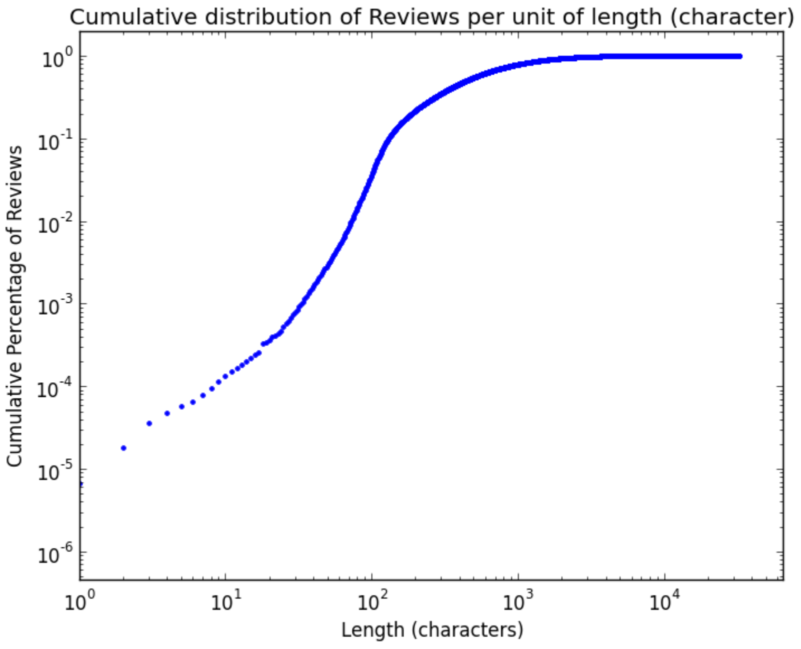


Figure 6: The cumulative distribution of the amount of reviews in function of its length (characters)

not the score is used as `outputKey`, but the helpfulness-ratio as explained above.

Reducer The logic behind the Reducer is completely the same as in the previous job.

The code can be found in the package `amazon.lengthhelp`, the resulting text-files as `results/all/LengthHelpWord.txt` and `results/all/LengthHelpChar.txt` the corresponding plots, which are shown in Figures 9 and 10, can found as `plots/all/LengthHelpWord.png` and `plots/all/ScoreLengthChar.png`.

What we see in these plots is that short reviews tend to be seen as very unhelpful (only $1/5^{th}$ of the users find reviews of 1 word helpful) while longer reviews keep getting more and more helpful until a certain threshold. It seems that users tend to find reviews less helpful when they have to read more than a $+/-$ 900 words or $+/-$ 4000 characters. We could conclude from this that users find longer reviews generally more useful until the review becomes just too long.

3.5 Score over Time

After having studied the correlation between the length of reviews, its helpfulness, etc. it might be an idea to also start holding the time-aspect into account; how do the reviews change over time? A first simple correlation that we could determine is the one between the score of the review and the moment on which it has been written. We decided that a good way of visualising this would be putting a data-point per month of a certain year (12 per year that is) and calculating the average of all scores of reviews given in that month. We used the following MapReduce-job for this:

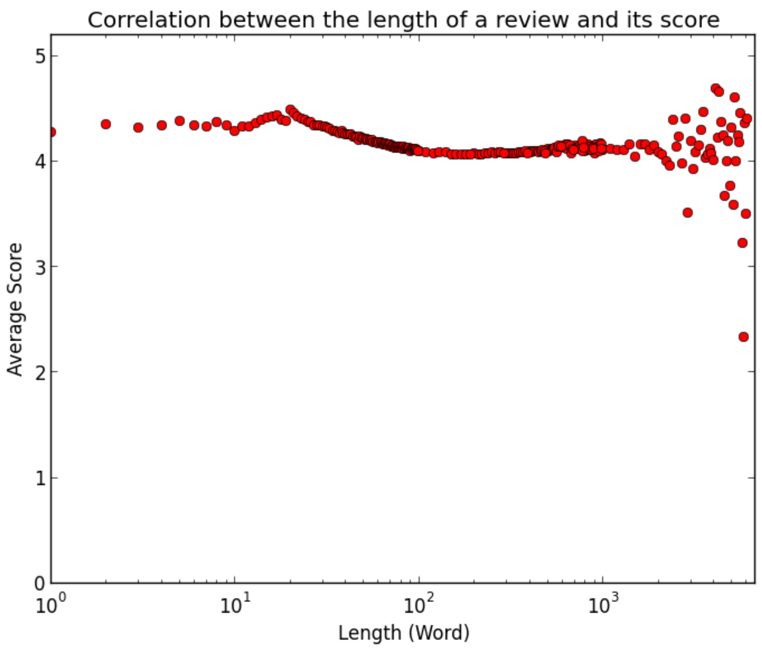


Figure 7: The distribution of the scores of reviews in function of their length (words)

Mapper Firstly we parse the review into an `AmazonReview`-object after which we set the date in the format of yyyy-MM as `outputKey` and the score of the review as `outputValue`.

Reducer The Reducer then computed the average score AND the standard deviation for a certain month/year-combination and outputs these as `outputValue` in combination with the `inputKey` which is set as `outputKey` without any changes.

Afterwards we saw that it would be hard to visualise the standard deviation in a proper manner so we decided to drop this in the visualisation-step.
The corresponding code can be found in the package `amazon.scoretime`, the resulting text-file as `results/all/ScoreTime.txt` and the corresponding plot, which is shown in Figure 11, can found as `plots/all/ScoreTime.png`.

It looks like the average score is stable, however it seems that users got more and more critical for products the longer Amazon existed. From the year of 2005 though it seems that the average score stayed stable. The sudden peak in the end is very probably a glitch as there are also much less reviews in the end in the dataset than in the middle (they still should be added to it).

3.6 Reviews over Time

The next thing that we could study was the relationship between the amount of reviews written over time, and this for every month of the year (so averaged over all the years), for every year (so averaged over all months) and for every month/year-combination. This might tell us something about the popularity of Amazon or about the activity on Amazon during certain months. In order to choose between these three possibilities we allowed an argument

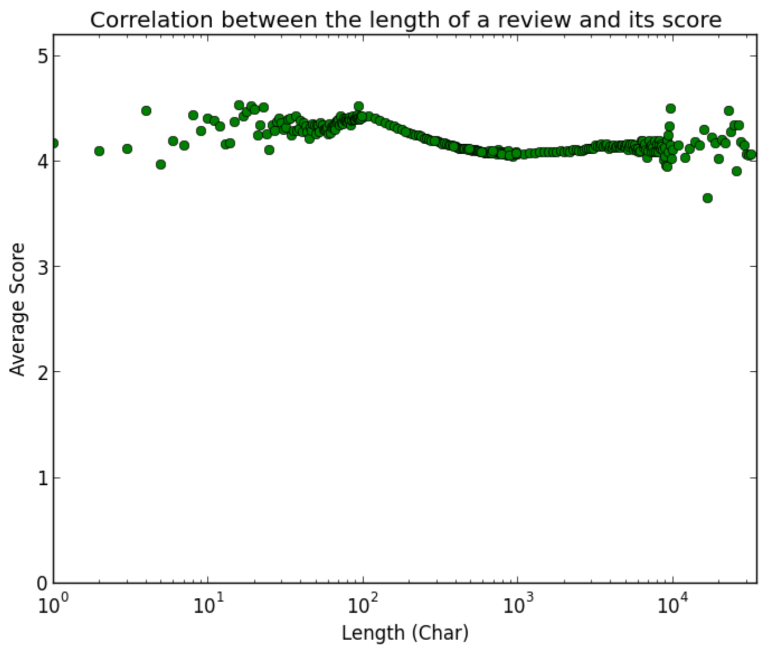


Figure 8: The distribution of the scores of reviews in function of their length (characters)

to be passed on to to the program by using the command-line. This argument should be *year*, *month* or *year-month*. Once the argument is passed to the Mapper the rest is straightforward:

Mapper After parsing the review into an `AmazonReview`-object and obtaining the command-line argument we set as value for out `outputKey` the date of the review, but in a textual format depending on the command-line argument. The `outputValue` is a constant: 1.

Reducer The reducer then sums up all `inputValues` and outputs this as `outputValue` with the key remaining the same.

The code can be found in the package `amazon.reviewtime`, the resulting text-files as `results/all/ReviewsTi` `results/all/ReviewsTimeYear.txt` and `results/all/ReviewsTimeMonthYear.txt` the corresponding plots, which are shown in Figures 12, 13 and 14, can found as `plots/all/ReviewsTimeMonth.png`, `plots/all/ReviewsTimeYear.png` and `plots/all/ReviewsTimeMonthYear.png`.

We can nicely identify the holiday season in Figure 12; people tend to buy much more stuff around the holidays as presents, which we can clearly see in the high number of reviews around the first and last month of the year. Also during the school holidays we see an increase of amount of reviews, which makes sense as well.

These plots in Figures 13 and 14 clearly indicate that something is wrong with the distribution of the reviews over all the years. It is very hard to explain why the amount of reviews has stayed as good as constant the last decade while the internet revolution was gaining more and more momentum. Many people did their very first only shopping during that decade but while looking at the plots we can't see anything like that: it stayed constant. Also the huge peak of reviews in the last months/years of the dataset is very strange to stay the least.

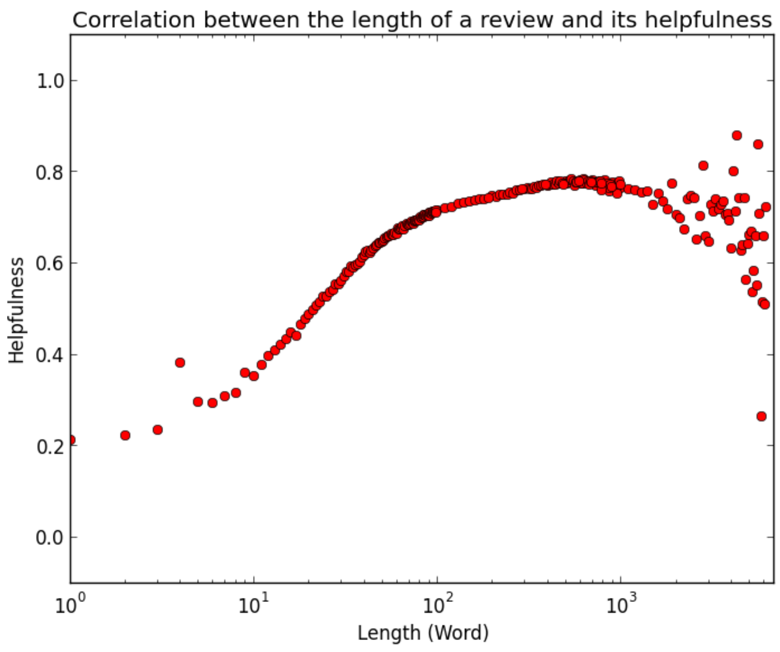


Figure 9: The distribution of the helpfulness of reviews in function of their length (words)

Probably this weird behaviour is because of the way in which this dataset was constructed but this is only guessing. Because of this fact we decided to stop looking for time-correlations as the distribution of reviews over time isn't what it should be it seems.

3.7 Reviews per User

How active are users? How many reviews do they write? That's what we're trying to research here. In order to do this we created a two-phased MapReduce program:

Mapper 1 After parsing the review into an `AmazonReview`-object we set as value for out `outputKey` the `userId`. The `outputValue` is a constant: 1.

Reducer 1 The reducer then sums up all `inputValues` and outputs this as `outputValue` with the key remaining the same.

Now our file exists out of the number of reviews a certain user has given, what we want in the end is total number of users with a certain amount of reviews so we need another step, the 'cumulation'.

Mapper 2 After reading a line from the output-file of the first phase we set the number of reviews of that specific user as `outputKey`. The `outputValue` is a constant: 1. Note that we don't use the `userId` here.

Reducer 2 The reducer then sums up all `inputValues` and outputs this as `outputValue` with the key remaining the same. This is identical to the Reducer of the first phase.

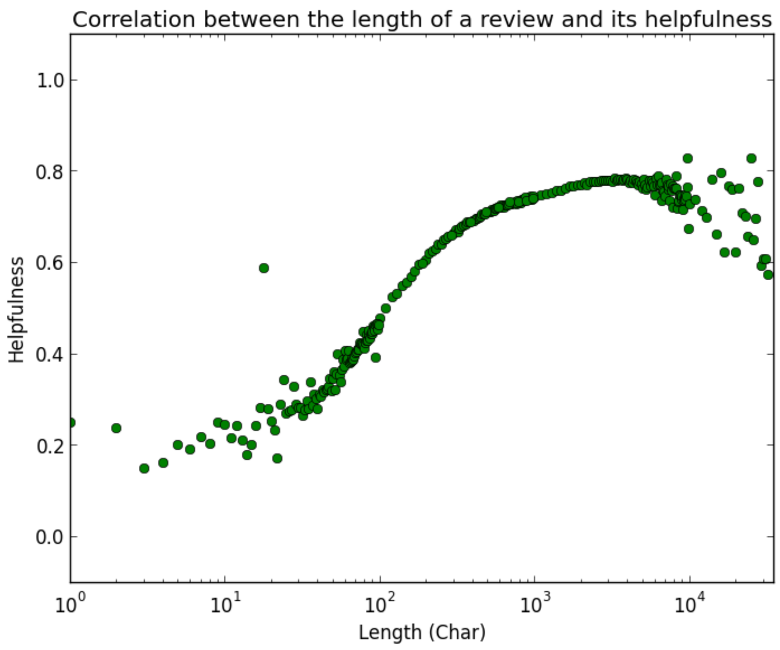


Figure 10: The distribution of the helpfulness of reviews in function of their length (characters)

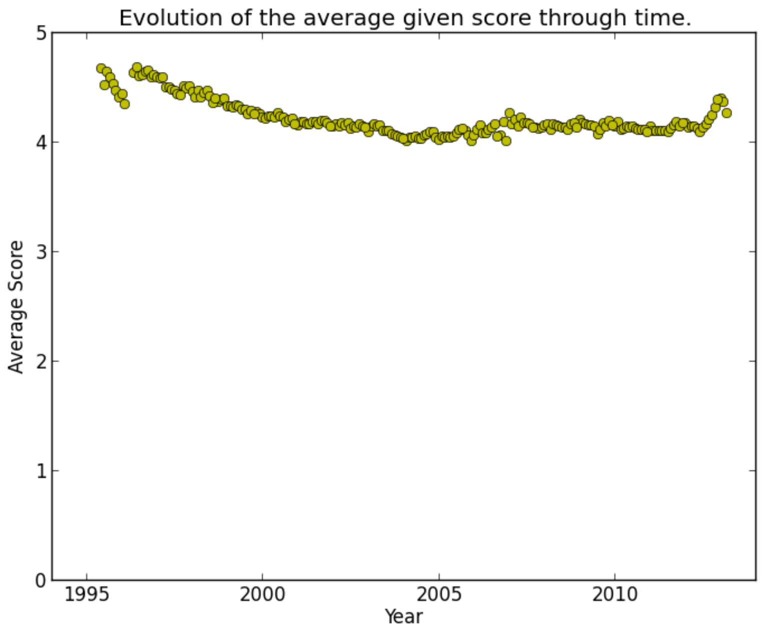


Figure 11: The average score of all reviews in a certain month of a certain year.

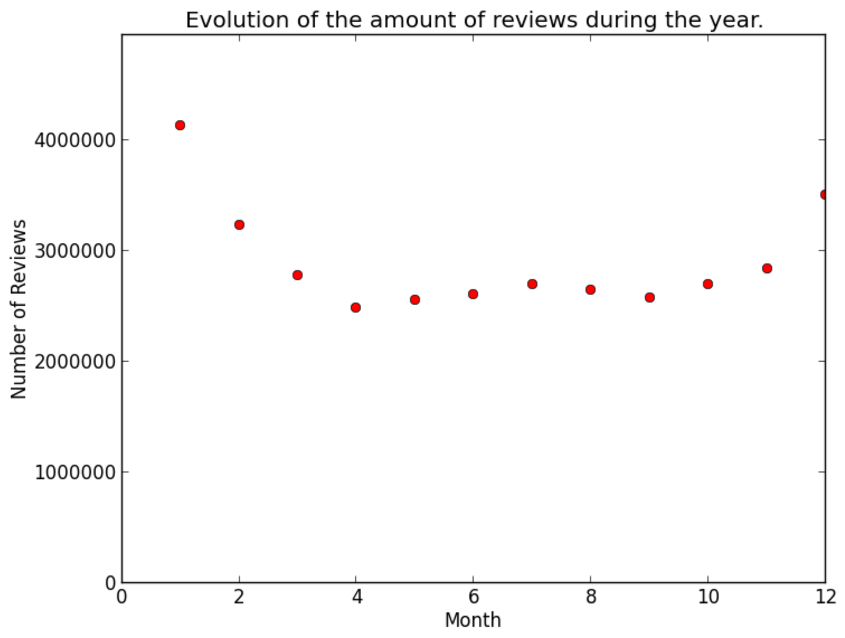


Figure 12: The amount of reviews during the months of the year

The corresponding code can be found in the package `amazon.userreview`, the resulting text-file as `results/all/UserReviews.txt` and the corresponding plot, which is shown in Figure 15 can found as `plots/all/UserReviews.png`.

We can clearly identify a power law in the plot in Figure 15 and even more in is cumulative version in Figure 16. Many users gave only one review, few users gave many reviews. One user for example wrote even more than 10,000 reviews.

3.8 Reviews per Product

This section was done in completely the same manner as the last section. It had the same result, used the same code, and so forth. The only change was that instead of using the `userId` the `productId` was used. The corresponding code can be found in the package `amazon.productreview`, the resulting text-file as `results/all/ProductReviews.txt` and the corresponding plot, which is shown in Figure 17, can found as `plots/all/ProductReviews.png`.

To finalise this part of the project we decided to use a small script in order to compute the cumulative distribution the amount of reviews a certain product has. With this script (`plotcum.py`) we transformed our result and after a small change to the plotting script we obtained the plot given in Figure 18 which can also be found in `plots/all/ProductReviews_cum.png`. This plot gives us an idea of what percentage of products we would keep if we would keep only reviews higher than a certain threshold. For example, if we only keep the products with more than 200 reviews, we keep about 1% of all the products. This given fact will be used later on in the next part of the project. Also note that in Figure 18 that if there is any power law in this relation, it is not a clean one as the data points get somewhat 'messy' in the end.

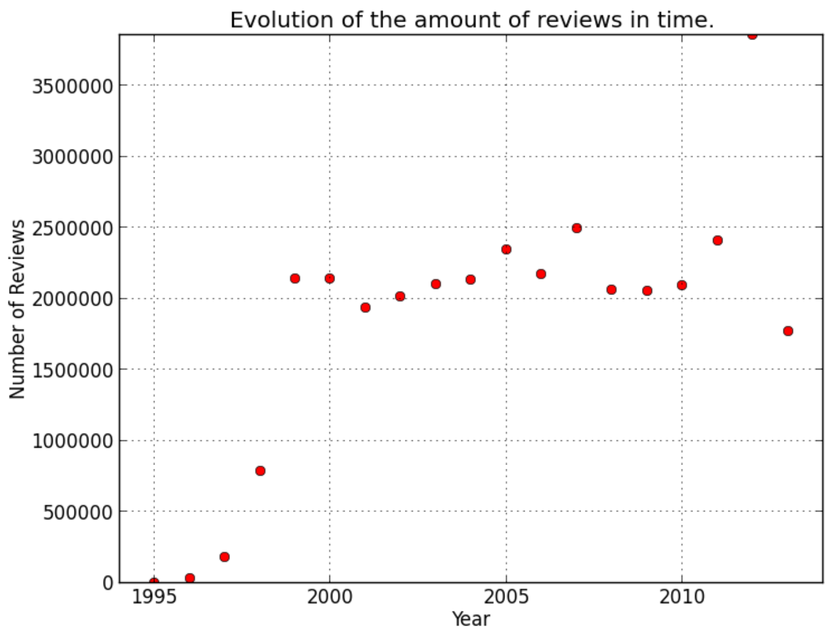


Figure 13: The amount of reviews per year.

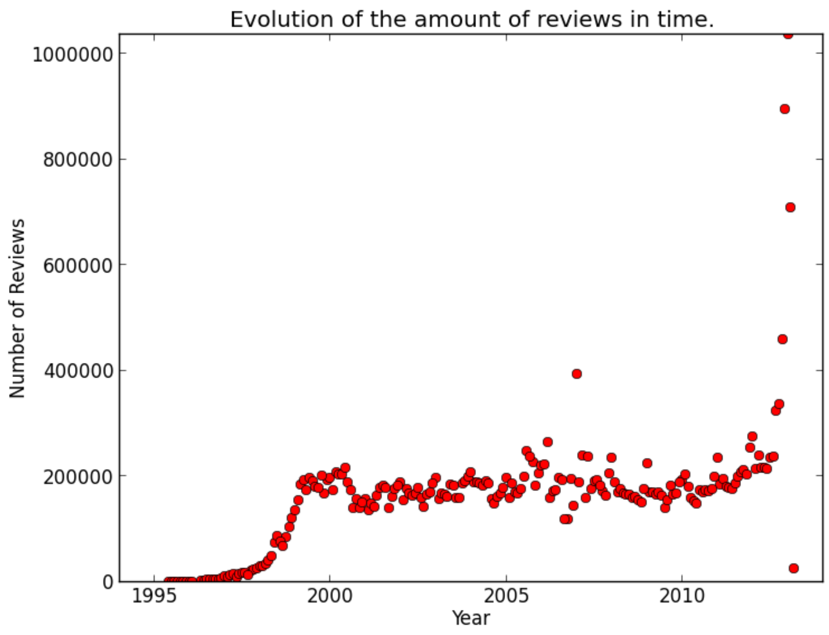


Figure 14: The amount of reviews per year, per month.

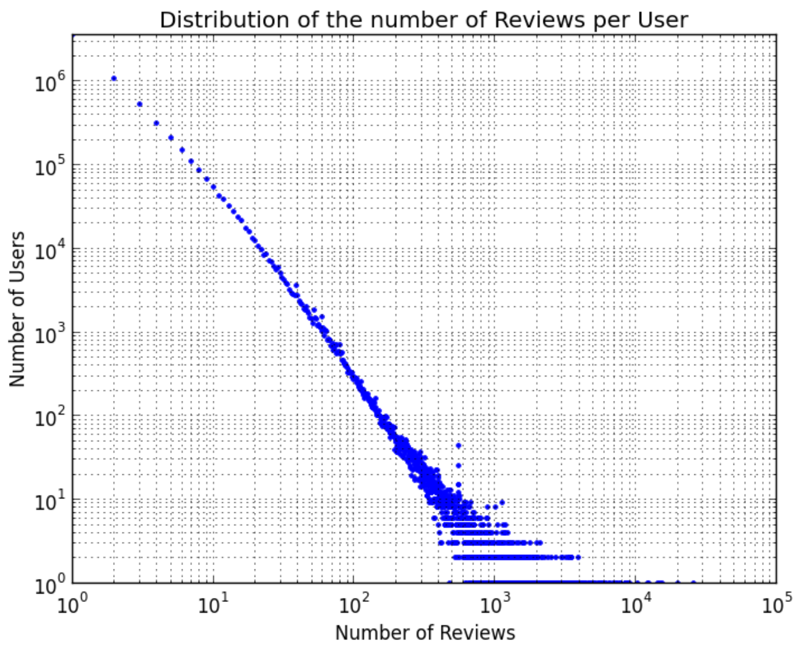


Figure 15: The amount of users who have given a certain amount of reviews.

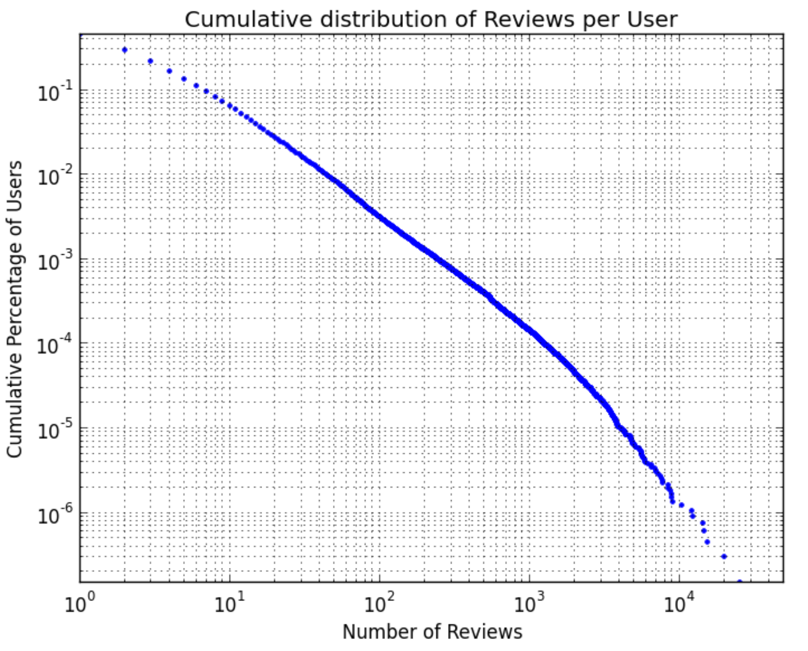


Figure 16: The cumulative distribution of users who have given a certain amount of reviews.

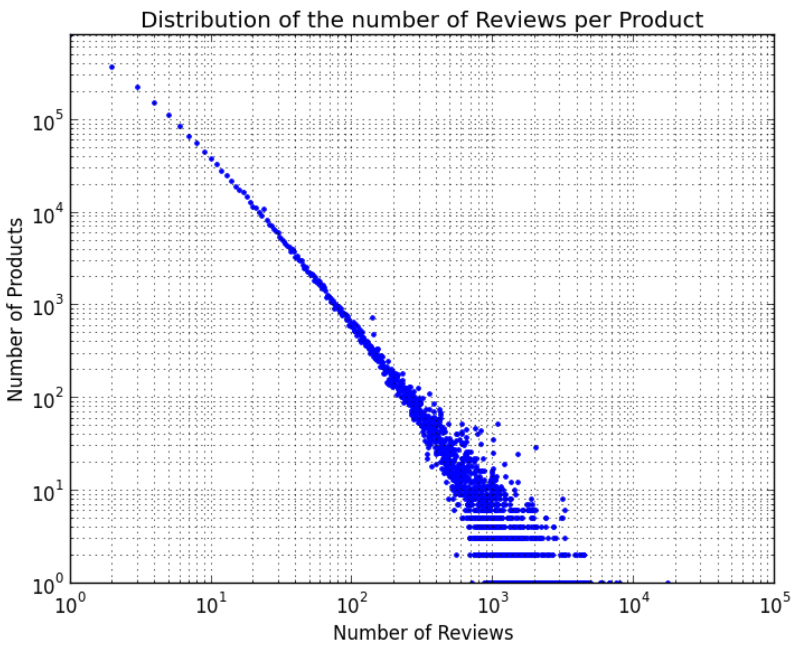


Figure 17: The cumulative percentage of users in function of its given number of reviews.

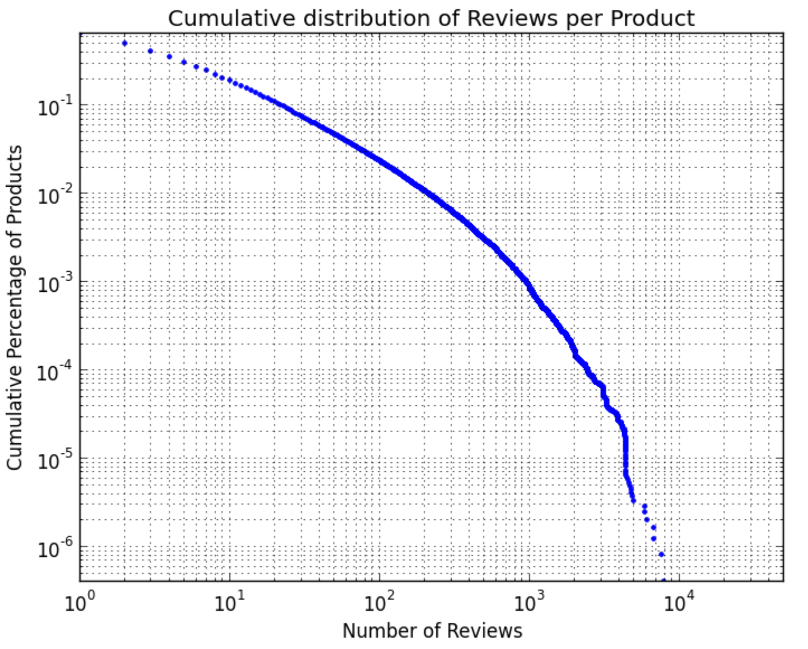


Figure 18: The cumulative percentage of products in function of its number of reviews.

4 Competing Products

These were all relatively simple analytics. The next step would be more advanced. After some thinking and asking input from outsiders we decided to go looking for competing products. This meant that some groups of users would strongly prefer one product above another. Examples of this would be Star Wars vs. Star Trek, PlayStation vs XBox and so forth.

4.1 Methodology

Of course, this information is not easily extractable from the dataset. Everything which we could need for this is in the reviews-dataset but there would be a need of quite some pre-processing and/or transforming from the dataset before getting this information would be straightforward. We decided that one of the best ways to extract this data would be transforming this long list of reviews into a co-rating graph as mentioned before in the Goals of this Project.
The nodes of this graph would be products and an edge would exist between two nodes if there was a user who rated them both ("co-rated"). This wouldn't be the only information though. In the nodes we would want to save the amount of reviews that that certain product has received and in every edge we would like to save three different values. The first value would be the 'strength' of this edge, which is the amount of users that rated both the products. **The** key though of making this method succeed in finding competing products is the so called *counterStrength*. This is a term which actually contains two values, counterStrengthA and counterStrenghB who are defined as follows:

"The counterStrengthA of an edge, existing between the nodes A and B, is the number of users who have given product A a rating which is at least **2** *higher than the rating it gave to B. The counterStrengthB of an edge, existing between the nodes A and B, is the number of users who have given product B a rating which is at least* **2** *higher than the rating it gave to A."*

It is self-explanatory that this creates a directed graph with a bidirectional edge between all nodes, containing exactly the same values, only with the counterStrengths mirrored. Why 2? This value was decided in the end once the graph had been built to be the parameter with the best output. With 1 the counterStrengths would be way too high and information would be lost just like in the case of 3, but then because the counterStrengths were too low.

Because the way how we defined our counterStrength this leads to a high counterStrengthA meaning that many users preferred product A over B. A high counterStrengthB in contrary means that many users preferred product B over A. If only one of these values is high it probably means that the one product is just much more popular than the other one and does not necessarily implies that they are competing. If they are both reasonably high in one edge though, then that would give us much reason to think that these are indeed competing products.
After this graph has been constructed we have to come with some kind of metric which ranks one edge above an other. It is thanks to this that we could then sort our list of edges based on this metric and come up with a ranking of top-competing products.

A graph of course has many different representations but given the fact how we would use this graph later, a simple list of edges seems to be the best idea. With this list of edges we would like to save their strength, counterStrength, total, the productId's and if possible the titles of the products the edge links as well.

4.2 Building the Graph

Given the fact that we start from a simple list of individual reviews it could be quickly seen that a lot of steps would be necessary to transform this data into the graph as described above. After carefully designing the algorithm we came up with a MapReduce-program consisting out of 3 phases:

1. The first MapReduce-phase is used to calculate the total amount of reviews per product and creating a list of products, each of them associated to a list of pairs of score and userId. Or in other words the very essence of all reviews written on that specific product.
2. The second MapReduce-phase is used to create per user the list of co-rated pairs of products. So for a set of products a certain user has rated, all possible pairs between distinct products are created and listed, these are already an early stage of our final edges. With these 'early edges' there is already a counterStrength associated as we simply compare the scores provided by the first phase for every pair of products. This value can for now only take values of 0 and 1.
3. The third MapReduce-phase is used to combine all 'early edges' with the same ordered product-pair, all counterStrengths are summed up and the total strength is calculated for every edge. Afterwards, at the very end, the final representation of the edge is written to the output-file.

This is the main idea behind building the graph. Unfortunately, the implementation of this algorithm proved to be very tricky and challenging, mainly due to nasty properties of the dataset which will be described in more detail in the next section. For more details on the final implementation please see the code in the package `amazon.graphbuilder` and the additional documentation in the repository.

4.3 Refining the Graph

Already in the very beginning we noticed that this graph was very dense and heavy. As every user is handled separately on a single reducer in the second MapReduce-phase and compare all products they reviewed pair-on-pair users with a high amount of reviewed products tend to make things very slowly. As seen in the first part of the project some users have more then 10,000 written reviews and make things very slowly. The first steps we made in making this more bearable were creating filters on the strengths of edges and on the total number of reviews per product before creating the final graph.
More concretely:

- In the first Reducer, before writing our output to the temporary output-file, we first checked whether the total number of reviews of that product was higher than a user-provided parameter (through the command line). If not, we deleted that product and all its corresponding edges. For determining this parameter we could base ourselves on Figure 18.

- In the third and final Reducer, before writing to the final output, we checked whether the strength of the edge was higher than the user-provided parameter (again through the command line). If it was lower we simply did not add the edge to the final graph.

These fixes already solved the problems of computational feasibility but when looking at the results we obtained from constructing the graph we still encountered quite some problems. The most obvious one was that sometimes the strength of an edge was *higher* than the total number of reviews of a product the edge was connected to, something which is of course impossible. Due to a small but tricky bug-fix this issue was quickly solved but not the number of reviews was *as high* as the strength which is still weird, seen to the fact that the productId's were different.

If we printed some additional information such as the titles next to the productId's we saw that many products who had the same title had the same productId's and had identical reviews, as such these two nodes in our initial graph are essentially the same and we could drop one of them. This extra feature was added in our second Reducer.
Unfortunately this wasn't enough. We saw that there were still products who treated the very same product but had a (slightly) different title and had a similar amount of reviews and a very high strength in between them. Also the counterStrength was mostly zero except some few additional cases. In order to solve this as best as we could (too harsh filtering would drop unnecessary nodes and edges) we added the following features:

- When creating the pairs in the Reducer of the second MapReduce-phase we considered two products equal if the title of the product with the longest title starts with the first half of the title of the product with the shortest title. These titles weren't the original ones but were all in lower case and underwent some preprocessing to remove characters like semicolons, comma's, etc.
- In the Reducer of the third and final MapReduce-phase we did not output the edge if its strength was higher than 0.9 times the total amount of reviews of one of the products it is connected to.
- In the same Reducer we added the additional rule that the sum of counterStrengthA and counterStrengthB of an edge **multiplied by 50** has to be **higher** than the normal strength of that edge. This is because very low counterStrengths very often indicated extremely similar products. And these products wouldn't have been considered for top competing products anyway.

After all these steps we have decent graphs who contained a reasonably low amount of duplicate products. The size of the graph was controllable by entering the right parameters during its construction. Based on Figure 18 and after some experimenting we decided to settle for a **minimum of number reviews per product of 100** and a **minimum strength of 50**. The total amount of edges in this graph (size) is 20,740 (note that there is for every edge a mirrored version between the same 2 nodes), a total number of nodes (order) is 6,153, an average in-degree and out-degree of 3.371 and had a size of a small 2 MB. This graph was small enough to allow very easy further processing but still contained all necessary information for further analytics. As the creation of this specific graph took about 7-8 minutes it is not too hard to create other graphs with other parameters which is why we also saved the 150-75 and 80-40 graph.

4.4 Top Competitors

After successfully having built the graph there was only one job that remained: finding the top competing products. This basically meant finding a metric based on the available numbers (total amount of reviews per product, strength and counterStrength) which expresses the competition between two products and outputting this in combination with the information necessary for its presentation (such as its titles).
For this purpose we created a MapReduce-job which combines the file containing the graph from the previous section and a text-file linking all productIds to their corresponding title. In the end we have the very same records as in the graph but with an added title, a rearrangement of the elements on the line, but most of all three different metrics in the beginning of the line. These metrics are defined as follows, with $maxCounterStrength$ being the biggest of $counterStrengthA$ and $counterStrengthB$:

- $metric1 = \frac{counterStrengthA \cdot counterStrengthB}{strength}$
- $metric2 = \frac{maxCounterStrength}{strength}$
- $metric3 = \frac{maxCounterStrength^2 \cdot minCounterStrength}{strength}$

The first metric is based on the idea that if productA is hated by groupB and loved by groupB while the opposite is valid for productB then we very probably have competitors. The second metric is based on the idea that if one product has a very high counterStrength for another one but not vice versa (one product is much more popular) then they are still competitors in a kind, only non-symmetric ones, but watching the results from this could lead us to new insights. Finally, the last metric is based on the two previous metrics.
Note that because the order of counterStrengths doesn't matter we essentially have 2 the same edges which we want to send through our MapReduce-job causing doubles in our final output. This is avoided by only allowing lines to be written to the output if the productId of the first product in the line precedes the productId of the second productId in the line alphabetically and as such removing all eventual doubles.
If we then let our new MapReduce program on the graph we obtain in a minute a text-file containing the ranking with all kinds of additional information. What we are interested in the most though is the ranking of product pairs and its metric to sort upon. The 10 most competing products can be obtained by using the following unix-command: `cat ranking100-50.txt | cut-f1,7 | sort -r | head` which generates the following output:

```
9.891   Who's Looking Out for You?-----Lies and the Lying Liars Who
    Tell Them: A Fair and Balanced Look at the Right
9.891   Political issues and visual rhetoric: Television
   advertising in Hungary's first free election-----Who's Looking
   Out for You?
8.831   Bridge Over Troubled Water/This Is The Night-----Flying
   Without Wings/Superstar
7.207   Duel at Red Table-----Battaglia
6.679   Dreams / Don't Cry Out Loud / I Believe-----I Believe /
   Chain of Fools / Summertime
```

```
5.921    The Real Thing———Some Hearts
5.860    PlayStation Portable (PSP) Value Pack———Nintendo DS
   Titanium
3.992    Eat, Pray, Love: One Woman's Search for Everything Across
   Italy, India and Indonesia———Water for Elephants: A Novel
3.714    Prometheus (2012)———Avatar (Two-Disc Original Theatrical
   Edition Blu-ray/DVD Combo) (2009)
3.714    Prometheus (2012)———Avatar (Mandarin Chinese Edition) [
   Limited Edition] (2009)
```

The first two pairs are a combination of books which are strongly Republican or Democratic. The pair in the third entry contain two singles by finalists of American Idol in 2003. The 4th entry pair contains two competitors for a famous beginner film-maker contest. A bit further we see the obvious competition between PSP and Nintendo DS. Much further we can find many similar, known, findings proving that our algorithm works and makes sense.
The option is also provided to add the links to their corresponding Amazon articles by adding 2 numbers to the command allowing easy further research:

```
cat ranking100-50.txt | cut-f1,7,8,9 | sort -r | head
```

In the end, the searching through this rank system can be done by a python-script which creates a shell command from its arguments. This script, to be found as `rank.py` in the `scripts`-folder can be configured in terms of the amount lines you want to display from the ranking(default is 10), the metric you want to sort upon (default: the first) and the parameters of the graph you want to sort upon (default: 100-50). This script is still very basic but can easily be extended to handle more customisation combined with making changes in the ranking and so on. The filtering on total amount of reviews and strength of edges could for example be postponed to this script and as such allowing much more freedom.

5 Future Work

Like any other data mining project this can always keep on going. By trying new and innovative combinations of review-properties we might always end up discovering new and fresh stuff. We also didn't give any attention to the possibility of NLP on this project but this was dropped as this would quickly get machine-learning intensive which was not the primary goal of this project but this could definitely be investigated later on by someone who wants to go more into depth on machine learning and in the end might find results which we can combine in the mining of these reviews. Building a recommender system from these reviews might also be a possibility.

With our constructed graph there still exists a multiple of opportunities. The graph could further be refined so we never have any more double reviews, we could try some community detection algorithms on this graph or one of many other graph algorithms.
The most concrete thing to do, which is unrelated to data-mining itself, would be creating a nice visualisation of the graph or a nicer, gentler, way of querying the ranking as using scripts and the commando line isn't the most user-friendly.

The most promising continuation of this project would be to use our ranking in order to find the competitors for one certain product, something which can easily be implemented. This might prove to be very useful for marketing purposes in order to know where to focus upon in its marketing strategy, why they're better than others. We might also give the option to study rankings in which an edge is only allowed to be put when two products are from different categories as this might lead to very remarkable findings.

6 Bibliography

[1] `http://lca.epfl.ch/student-projects/projects/2014-spring/hadoop-reviews.html`