Amazon Products and Reviews Analysis

Final Project

Technical University of Denmark
Department of Applied Mathematics and Computer Science
02807 - Computational Tools For Big Data
December 4, 2016

Contents

1 Introduction

This report is meant to document the work done in regard of the Final Project of the course Computational Tools for Big Data.

First, we will describe the dataset on which we are working. Its format, attributes and data structures will be exposed and a special focus will be on size. Then, we will discuss our developing environment: we will list all of the approaches we experimented to front the big size of the data, listing the problems associated with them, to end up describing the final solution we decided to adopt to carry out the analysis.

The data analysis will mainly revolve around three modules:

1. Querying with Apache Spark to extract insightful statistics
2. Text analysis, hashing and machine learning to predict the rating of the review according to contained words
3. Graph Database of the products to explore the relationships between them

2 The Dataset

The dataset we decided to work with is the **Amazon Reviews and Product Metadata**, which contains more than 41 milion reviews and metadata for 9.4 milion products. The time range of the reviews starts from May 1996 until July 2014. The full description of the dataset is available at the page `http://jmcauley.ucsd.edu/data/amazon/` .
We decided to work on the most structured and interesting part of the dataset: two **JSON** files, one for the reviews and one for the metadata. It's worth saying that the reviews we took in consideration are actually a subset of the whole set of the reviews, since we excluded all those reviews from users who wrote less than 5 reviews (which is called "5-core"). The JSON files contain a list of JSON dictionaries, one per line. Hereby we show a sample of a **review dictionary**:


```
{
  "reviewerID": "A2SUAM1J3GNN3B",
  "asin": "0000013714",
  "reviewerName": "J. McDonald",
  "helpful": [2, 3],
  "reviewText": "I bought this for my husband who plays the piano.
  He is having a wonderful time playing these old hymns. The music is
  at times hard to read because we think the book was published for
  singing from more than playing from.  Great purchase though!",
  "overall": 5.0,
  "summary": "Heavenly Highway Hymns",
  "unixReviewTime": 1252800000,
  "reviewTime": "09 13, 2009"
}
```

And below you can see a sample of the **metadata dictionary**:


```
{
  "asin": "0000031852",
  "title": "Girls Ballet Tutu Zebra Hot Pink",
  "price": 3.17,
  "imUrl":
    "http://ecx.images-amazon.com/images/I/51fAmVkTbyL._SY300_.jpg",
  "related":
  {
    "also_bought": ["B00JHONN1S", "B002BZX8Z6", "0000031909",
     [...]
    "B00D103F8U", "B007R2RM8W"],
    "also_viewed": ["B002BZX8Z6", "B00JHONN1S", "B008F0SU0Y",
     [...]
    "B00AL2569W", "B00B608000", "B008F0SMUC", "B00BFXLZ8M"],
    "bought_together": ["B002BZX8Z6"]
  },
  "salesRank": {"Toys & Games": 211836},
  "brand": "Coxlures",
  "categories": [["Sports & Outdoors", "Other Sports", "Dance"]]
}
```


The size of the files declared on the website is 9.9 GB for the 5-core and 3.1 GB for the metadata. We assumed this to be the dimension of the whole uncompressed dataset, but once downloaded we found out that the dimensions are relative to the compressed `.gz` archives. Unfortunately, there is a bug[1] in the `.gz` format: the size of the original file is stored in an integer of length 32 bits, which means it is able to describe files up to 2^{32} bytes which means 4 GB: we couldn't know in principle the size of our dataset! The bash of Unix saves us once again: running the following command gave us the number of bytes of the uncompressed file:

```
~/Desktop/ctfbd/proj/ctfbd$ zcat kcore_5.json.gz | wc -c
32072979001
~/Desktop/ctfbd/proj/ctfbd$ zcat metadata.json.gz | wc -c
10544467811
```

Here `zcat` decompresses the file which comes piped into `wc` that counts its characters. This means that we are dealing with two files of 29.9 and 9.8 GB: three times what we expected. Instead of the option to count the characters, we can count the lines, and since the there is one record for every line we get the numbers:

```
~/Desktop/ctfbd/proj/ctfbd$ zcat kcore_5.json.gz | wc -l
41135699
~/Desktop/ctfbd/proj/ctfbd$ zcat metadata.json.gz | wc -l
9430088
```

That is, 41 millions reviews and 9 millions metadata records.

[1] `https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=149775`

3 Environment Setup

We first start giving a full explanation of how we achieved a working environment, since we spent most of the first two weeks of the project choosing the proper libraries and the tools we would have worked with. In order to show our efforts for this step, we are going to list all of our attempts: for each of them we will list why we couldn't adopt that solution.

3.1 Loading file lines in memory and generators

Solution The first problem we faced was loading the content of the files into memory: this is the very first approach while working with files, but by the other hand it's not the proper one when very big sized files are taken in consideration. Without any doubts it's not possible to work with memory and simply opening the files and saving the contents in Python data structures like lists or dictionaries.
Obviously we started looking for another solution, and we found out that using **generators** probably could have solved something. This was partially true, but our idea was to start working with some fixed-schema structures like those ones provided by the **Pandas** library, which is widely used for data science and Big Data environments, and this approach couldn't easily fit our intentions. That's why we decided to discard it and try to look for something different.
The main problem was always an **excessive usage of RAM by Python**, thus we decided to focus our attention looking for some environments that could make us work *without affecting the memory.*

3.2 SQLite & Pandas

One solution that was fitting quite nicely our purposes was a combination of the **SQLite library** and **Pandas**. The basic idea was taking the content of the JSON files and copying it into a `.db` file, through SQLite modules, and then working easily with it with Pandas **DataFrames**. The problem we couldn't overcome was that this process was taking *too much time* and we were forced not to continue with it. We tried creating the `.db` file even with the smallest portion of the reviews, for those products in the category *Musical Instruments* (7MB) and the execution time was extremely high. Definitely it was not possible to work with files of up to 10GB.
In the Appendix 9 the code used to attempt the creation of such DB can be visioned.

3.3 Blaze, Dask and Castra

Solution We also tried to setup an environment involving Blaze, Dask and Castra. We took inspiration from the following article: `http://blaze.pydata.org/blog/2015/09/08/reddit-comments/`.
The **Blaze Ecosystem** is a collection of useful tools that are used by data scientist to work with *larger-than-memory* datasets. Since we realized that probably this ecosystem was able to solve our problems, we decided to commit to it and try to properly setup our environment.
Dask and **Castra** are two components of the Blaze ecosystem:

Dask is designed to fit the space between in memory tools like NumPy/Pandas and distributed tools like Spark/Hadoop. By using blocked algorithms and the existing Python ecosystem, it is able to work efficiently on large arrays or dataframes - often in parallel. Castra is:

- A partitioned, on disk column-store: storing data by column allows us to only read in data for columns we care about. Partitioning data along the index allows us to ignore irrelevant rows. Together, they reduce the amount of I/O that needs to be done for each query.
- That uses compression to improve disk access: Castra uses `blosc` to compress data, further reducing the amount of needed I/O. Blosc is a modern compressor that is much faster than bzip/gzip.
- And knows about pandas categoricals: storing repetitive strings as categoricals both reduce I/O (Blosc can achieve better compression ratios), and also improves computational efficiency in Pandas.

In the Appendix 9 it can be seen the full code we used to create Castra files of our dataset and some queries executed on the Dask Data Frame.
Once we've set it up and started doing some queries with it, we realised it was not a reliable environment. The queries were slower than expected, the project was not well documented and of course not supported. We decided to look for yet another more stable solution.

3.4 Spark installation

As last resort, we headed towards Spark. It was our last option because in the lectures this technology was labeled as "very hard to install". In the Appendix 9 it is possible to read the steps followed for the installation for both Windows and Linux/macOS operative systems.
With Spark we found the perfect solution to efficiently query our datasets. Our approach in this regard is accurately described in Section 4.

3.5 Amazon EC2 Instance

We also managed to get a working set up of an Amazon EC2 instance to take advantage of its tools and services:

- First, it's convenient to have a **common remote space** where all the members of our group can work easily and efficiently (e.g. easily avoid problems of annoying path problems between different machines);
- Even if the computational power of the instance is less than our machines, the main advantage is the **very big size of the hard disks that we can attach**, such that we can save space on our own disks and feel free to use as much space as we need (important to notice that the Castra files that we generated on our machines take about 26 GB of *additional* space).

- Since the EC2 instance works with Linux, it's much easier to install tools like Apache Spark and it's as fast as it is on our machines to get a final working environment: we just installed all the needed libraries.
- We installed `jupyter notebook` on the instance, and we set up a **server notebook**, that allows to write the code from our own browsers, and run the code on the instance. It is important to notice that the instance has only 1GB of RAM available, but it has a **SSD hard disk** and in our case this is a great advantage, since our code needs more disk power than memory.
- We installed `git` on the instance in order to make all the files (IPython notebooks and Python scripts) synchronized with our repository.

4 Querying with Apache Spark

4.1 Parsing Reviews and Metadata

We decided to conduct our main analysis with Apache Spark, which was confirmed as one of the most efficient tools for querying huge datasets with reasonable execution times. Spark has been installed both in local *and* on the Amazon EC2 instance, in order to perform parallel executions and optimize as much as possible the queries. The average execution time in local was around 5-7 minutes, while on the instance was never shorter than 10 minutes.
Our input files were the `k_core_no_books.json` and the `metadata_no_books.json`: we got rid of books, not relevant for our analysis. Amazon became a large book shop and more than 50% out of the total reviews available in the original file are written for books. This makes them incomparable with all the other reviews involving different kind of products. This was the main reason that led us to decide not to consider books anymore and focus on reviews for each specific product.
In Appendix 9 we show all the libraries we used and had to import, the **mapping functions** we used to properly **clean and parse** the JSON files, and how we initialized the basic data structures for Spark. Basically, we used **DataFrames**, which can be used with a nice **SQL/Pandas**-like syntax and because of this are easy to query. In order to perform SQL queries, Spark provides an extremely easy-to-use syntax, like `spark.sql(query)`, where `query` is a *string*. In the appendix, we're not showing everytime this statement, such that when we performed SQL queries we just simply show the SQL syntax.
It's worth saying that Spark tries to infer automatically the schema of the DataFrame, if no meta-data information (*data types*) are provided; in our case, we decided to force our own schema to make sure every column was treated as it should have.
It is also mportant to notice that Spark doesn't perform any transformation until an **action** is invoked. The outputs in the listing 9 show the schema of the two datasets. As a first example, we also show the first rows of the DataFrames created by Spark in the listing 9.
In order to perform SQL queries, a **table name** must be registered, as we did in the lising 9.
Our very first query (listing 9 found the exact number of reviews and products we had

to work with, and the results showed that our dataset contained 9.288.692 reviews and 6.984.215 products.

4.2 Example Queries

In order to make sure we could actually work properly with all the columns and their data types, we tried to perform some simple queries, some of them are listed in 9 with relative outputs. Actually, we encountered troubles while trying to perform queries involving `date` objects: we handled them correctly but it seems that extracting information from a `date` object with Spark raises an `AnalysisException`, thus this forced us to treat dates as **strings** while querying; instead, the returned rows always contained the proper object type.
While performing some example queries, we also found out that actually the set of the products included in our dataset has been restricted to exclude those products whose price was higher than 999.99$. We run a simple bash script in order to figure out whether this happened because of some weird behaviours of Spark or because of the dataset. The result clearly showed that the dataset had been really reduced:

```
[ec2-user@ip-X-X-X-X EC2]$ cat metadata_no_books.json | awk '/'\''price'\'':
    [0-9]+.?[0-9][0-9][0-9]/ {print;}'
[ec2-user@ip-X-X-X-X EC2]$
```

Last but not least, it's worth mentioning that we used the `pickle` module of Python in order to save on disk the important results we had to work with, since it was too much time consuming repeating the same queries. We hereby show an example of the standard operations we were performing in order to pickle/unpicle objects.

```
#Example of Pickler:
with open(pickled_objects_folder+'filename','w') as f:
    pickle.dump(object, f)

#Example of Unpickler:
with open(pickled_objects_folder+'filename') as f:
    pickle.load(f)
```

Besides, the type of the object we usually pickled was a list of `Row` objects, which is part of the `spark.sql.types` module. In order to convert it to a more usable data type, like a dictionary, we were executing the following statement:

```
list = [row.asDict() for row in list]
```

4.3 Reviews Helpfulness

Which are the most helpful reviews ever? In order to state this we took in consideration the `helpful` parameter: it is represented as an array of two elements, where the second one indicates how many people voted the review, and the first one tells how many voters out of the total considered this review *helpful*. The difference between these two values, thus, told how many considered this review *not helpful*. Since there are lots of reviews with very few voters, we filtered them and took in consideration only those reviews that were voted a lot of times.
As a first interesting statistic, we calculated the **average number of voters of a**

single review, the **average number of voters who considered the review helpful** and the **average number of voters who considered the review not helpful**, and this is what we obtained:

Average Number of Voters	Average Helpful Voters	Average Not Helpful Voters
8.4	1.8	6.5

The results above show that there are much more reviews with a low number of voters, which is obviously acceptable.
It came natural then to extract the **most voted reviews** (listing 9, and we found out that the top 3 most voted reviews were written for:

- Hutzler 571 Banana Slicer
- Kindle Fire HD 7"
- SimCity: Limited Edition (PC Video Game)

We came up to an interesting result: *Most of the top 20 most voted reviews were positive: their overall score is above 4.0; But there are two outliers, the 3rd and 5th reviews in the list: these reviews have an overall score of 1.0!*

- It's evident that all these reviews are pretty much **recent**, most of them were written in the time range from 2010 to 2013, with a few reviews in 2007, 2008 and 2009;
- People tend to **vote much more good reviews rather than bad reviews**;
- However, there are few examples of **bad reviews** who got **A LOT of voters**, mostly supporting their opinion;
- Since Amazon is showing first the top reviews for each product, it comes quite natural that once a review becomes a "top review", its voters grow exponentially. That's why the number of voters of the first most voted reviews differ by such a high gap;
- Why did those top-voted reviews actually become top? We thought about taking in consideration the review text and analyze it.

After these results, we wondered whether the voters agree with the reviewer or not. Actually if a review is a top-voted doesn't explicitly show how how many voters actually considered that review helpful. From the basic stats we calculated, it's already clear that, **on average, most of the voters agree with the opinion given by the reviewer**.
However, we found out that there are so many products for which people have written reviews that a lot of voters **strongly** didn't agree with (listing 9)!
Interesting considerations could be pointed out: the obtained result clearly showed a **majority of negative reviews** (overall 1.0) than positive reviews (overall 5.0):

- this leads to state that **there are lots of negative reviews that a lot of people disagree with**, and...

- **there are some positive reviews with a relative high number of people who disagree with it**

Concerning the **positive reviews**, We pointed out that **the number of people who disagree is often much smaller than the number of people who agree**, and mostly we're talking about thousands of voters. Thus, we can simply state that this difference is acceptable and they can still be considered top reviews.
By the other hand, the **negative reviews** are quite different than the previous ones: we can divide them in two subsets:

- negative reviews with high number of voters and very small number of people who considered them helpful: possible candidates for no-sense reviews, they don't characterize very well the related product;

- negative reviews with high number of voters and people who considered them helpful: as well as we stated for the positive reviews, these can be considered top reviews, so they strongly characterize the product, even if in a bad way.

Therefore, in order to see **which top-reviews** (in terms of *number of voters*) also gained the **highest number of opinion supporters** (*people who agree with the review and the vote and think that the review is helpful*), we had to check where the difference between `helpful[1]` and `helpful[0]` is minimum. However, this couldn't be easily achieved, because there could have been many reviews with a very small number of voters. A primary idea was to filter the reviews and keep only those ones with at least 3000 number of voters, but we were conscious that this solution also cut off a lot of reviews that could still be considered top-reviews, in terms of helpfulness (results in listing 9).

4.3.1 Simple Text Analysis and Word Clouds

We thought it could be interesting to generate two wordclouds related to the two sets of reviews we previously identified: the **most voted reviews** and the **reviews most people didn't agree with**. We wrote down a simple script to clean, tokenize the text and generate the word clouds (listing 9). Hereby we show what we obtained for the most voted reviews and the worst reviews.
Our curiosity led us to read the text of the *most voted review ever*:

```
Summary: No more winning for you, Mr. Banana!
Review: For decades I have been trying to come up with an ideal way to
slice a banana.  "Use a knife!" they say.  Well...my parole officer won't
allow me to be around knives.  "Shoot it with a gun!"  Background check...
HELLO! I had to resort to carefully attempt to slice those bananas with my
bare hands. 99.9% of the time, I would get so frustrated that I just ended
up squishing the fruit in my hands and throwing it against the wall in
anger. Then, after a fit of banana-induced rage, my parole officer
introduced me to this kitchen marvel and my life was changed.  No longer
consumed by seething anger and animosity towards thick-skinned yellow
fruit, I was able to concentrate on my love of theatre and am writing a
musical play about two lovers from rival gangs that just try to make it in
the world.  I think I'll call it South Side Story.Banana slicer...thanks
to you, I see greatness on the horizon.
```

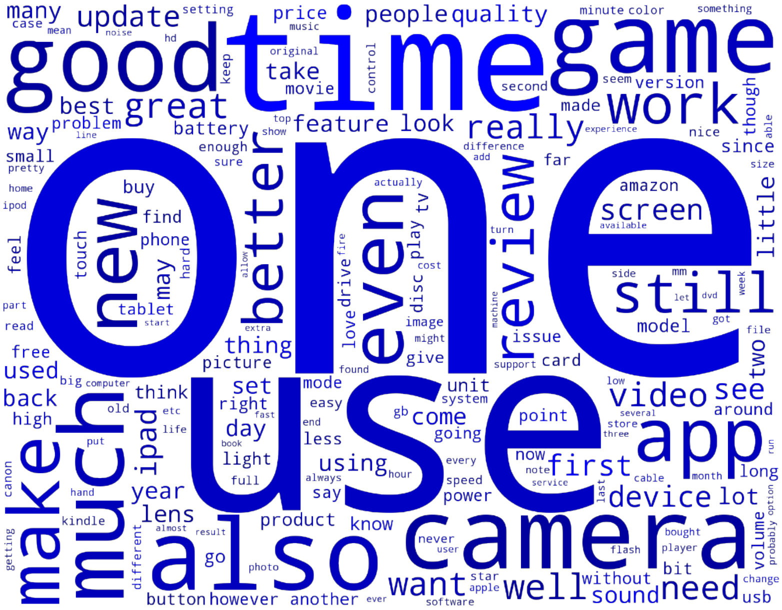


Figure 1: Most voted reviews

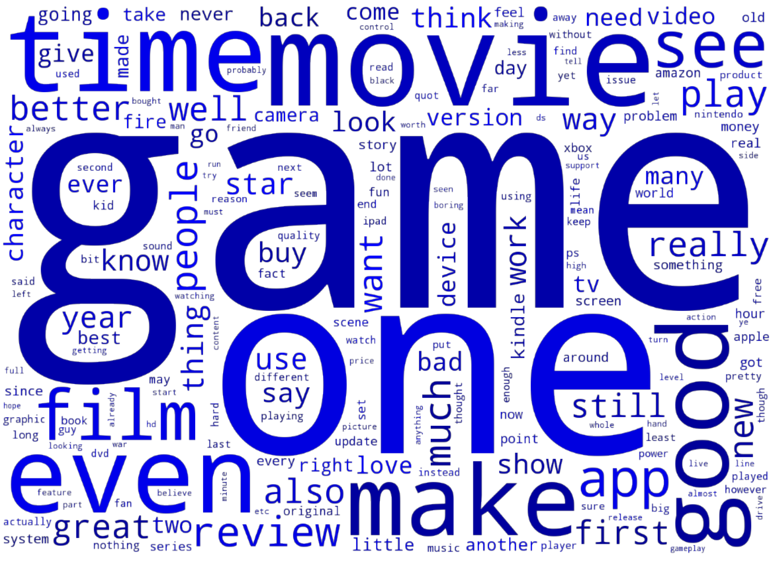


Figure 2: Worst reviews

And the *worst review ever* :

```
Summary: OUT OF STOCK!!!?????
Review: They didnt make enough copy and not enough art books.This is
unacceptable. Even though i love  JRPG, this game deserves 1 out of 5
since it messes with fan's mind.Plus i think call of duty vita is
better than this game
```

We can notice some common frequent words between the two subsets, despite the great difference between them.

5 Products Analysis

We conducted our work trying to provide an answer to these questions:

- What are the products for which the most voted reviews were written?
- Top salesRankValues: which are the top-sold products for each category?
- What are the most appreciated products in terms of average overall?
- What are the main categories of the products for which people have been more satisfied in terms of overall score?

5.1 Top 20 products related to the most voted reviews

Unfortunately, some products were stored in the `metadata.json` file with missing attributes (like title and price). However, filtering the results excluding `null` values, these are the results we obtained (listing 9):

```
Hutzler 571 Banana Slicer
Kindle Fire HD 7&amp;quot;, Dolby Audio, Dual-Band Wi-Fi, 16 GB
Tria&amp;reg; Laser Hair Removal System - Gen 3 Previous Model (Discontinued)
Kleenex Facial Tissue, White (18 boxes)
Google Chromecast HDMI Streaming Media Player
Kindle Fire HD 7&amp;quot;, Dolby Audio, Dual-Band Wi-Fi, 16 GB
Google Chromecast HDMI Streaming Media Player
Apple iPod touch 32GB Black (4th Generation) (Discontinued by Manufacturer)
Eureka Enviro Hard-Surface Floor Steamer, 313A
INSANITY Base Kit - DVD Workout
Breville BJE200XL Compact Juice Fountain 700-Watt Juice Extractor
```

It's quite clear that most of these products are well known worldwide. Still, the *Banana Slicer* holds a surprising position among them.

5.2 Top-sold products for each category

The `salesRankCategories` and `salesRankValues` represent what is the rank for each product in some specific categories. For each main category, we wanted to find out which are the **top-sold products** (listing 9):

Category	Product
Appliances	Danby 0.7 cu.ft. Countertop Microwave, White
Arts, Crafts & Sewing	Crayola Crayons 24 Count - 2 Packs crayon great school
Automotive	2x Aluminium Hand Grips Throttle Assistor [...]
Baby	Carter's Keep Me Dry Waterproof Fitted Quilted Crib Pad
Beauty	HSI PROFESSIONAL IONIC HAIR STRAIGHTENER [...]
Books	Rosencrantz and Guildenstern Are Dead
Camera & Photo	Fujifilm Instax Mini Instant Film, 10 Sheets x 5 packs
Clothing	Disney Frozen Dress Princess Elsa Girls Pajama [...]
Computers & Accessories	ARRIS / Motorola SurfBoard Cable Modem - White [...]
Electronics	Google Chromecast HDMI Streaming Media Player
Grocery & Gourmet Food	Viva Labs Organic Extra Virgin Coconut Oil, 16 Ounce
Health & Personal Care	Pampers Sensitive Wipes 7x Box 448 Count
Home & Kitchen	Honeywell TurboForce Fan, HT-900
Home Improvement	O'Keeffe's Healthy Feet Creme 3.2oz Jar
Industrial & Scientific	Georgia-Pacific Signature Premium Multifold Paper Towel
Jewelry	Alex and Ani Initial Expandable Wire Bangle Bracelet
Kitchen & Dining	Paderno Cuisine Tri-Blade Spiral Vegetable Slicer
Musical Instruments	iMBAPrice Gold Plated 3.5mm Male to Female Cable [...]
Patio, Lawn & Garden	Weber 6492 Original Instant-Read Thermometer
Pet Supplies	Precious Cat Ultra Premium Clumping Cat Litter
Prime Pantry	Nutella Hazelnut Spread
Shoes	Sanuk Women's Yoga Sling 2 Flip Flop
Sports & Outdoors	LifeStraw Personal Water Filter
Toys & Games	Cards Against Humanity
Watches	U.S. Polo Assn. Sport Men's US9061 Watch [...]

These results are actually updated to the time the dataset was generated (2014), thus many changes might happened.

5.3 Most appreciated products according to the Average Overall

We could calculate which products are the **most appreciated** ones according to the **average overall of their reviews** (*we had to filter the results and for instance take only those products having at least 1000 reviews*). The extended result table can be seen in the listing 9

```
Butler Creek Lula Universal Pistol Magazine Loader
|AmazonBasics High-Speed HDMI Cable 2-Pack - 6.5 Feet (2 Meters)
AmazonBasics USB 2.0 A-Male to A-Female Extension Cable - 9.8 Feet (3 Meters)
Mediabridge ULTRA Series HDMI Cable (6 Feet)
AmazonBasics High-Speed HDMI Cable - 15 Feet ( 4.6 Meters) Supports Ethernet
SanDisk 4GB Extreme SDHC Class 10 Memory Card
Samsung Electronics 840 EVO-Series 120GB 2.5-Inch SATA III
BlueRigger High Speed HDMI Cable with Ethernet 6.6 Feet (2m)
Masterpiece Classic: Downton Abbey Season 3
AmazonBasics Digital Optical Audio Toslink Cable, 6 Feet
Garmin Portable Friction Mount - Frustration Free Packaging
Firefly: The Complete Series
eneloop SEC-CSPACER4PK C Size Spacers for use with AA battery cells
```

```
SanDisk 2GB Class 4 SD Flash Memory Card- SDSDB-002G-B35 (Label May Change)
Lodge L8SK3 Pre-Seasoned Cast-Iron Skillet, 10.25-inch
New Trent iCarrier 12000mAh Portable Dual USB Port External Battery Charger
Transcend 8 GB Class 10 SDHC Flash Memory Card
ARRIS / Motorola SB6121 SURFboard DOCSIS 3.0 Cable Modem
Kingston 4 GB microSDHC Class 4 Flash Memory Card SDC4/4GBET
DVI Gear HDMI Cable 2M 6 feet
```

5.4 Which are the most recommended categories to buy from on Amazon?

We were wondering this, such that according to the current insights, we could have a clear overview of **which categories are the most/least reliable according to customer satisfaction**. Since there are basically differences between each category, and the amount of orders of products from the different categories is not the same, we couldn't provide results according only to the **overall** (*how much a customer was satisfied*) or the **reviews frequency** (*how many people bought the products*). Hence, we calculated the average overall for each category but we considered also the **standard deviation**, and the lower this value, the better (listing 9

Categories	Count	Average Overall
Movies,Movies & TV	1291511	4.0185534618
Games,Apps for Android	536135	3.99561677563
TV,Movies & TV	351045	4.4227577661
Baby	160792	4.21411513011
Grocery & Gourmet Food	145829	4.24051457529
Cases,Basic Cases,Cell Phones & Accessories	72493	4.10490668065
Electronics,Cases,Touch Screen Tablet Accessories	72172	4.20832178684
Electronics,Headphones,Audio & Video Accessories	68323	4.01330445092
Cables & Interconnects,HDMI Cables,Video Cables	40824	4.54027042916
Apps for Android,Entertainment	39424	3.56166294643
Amazon Instant Video	37126	4.20952970964
Point & Shoot Digital Cameras,Camera & Photo,Electronics	29373	4.18986824635
Electronics,External Hard Drives,Computers & Accessories	28346	4.16446764976
Electronics,Computers & Accessories,Cables & Accessories	27105	4.13075078399
Xbox 360,Video Games,Games	27003	3.99200088879
Games,Video Games,PlayStation 3	25302	4.09971543751
PC,Video Games,Games	24765	3.75045427014
Cables & Interconnects,Electronics,USB Cables	23890	4.35755546254
Routers,Electronics,Computers & Accessories	23255	3.87103848635
Electronics,Computers & Accessories,Cables & Accessories	23163	4.17394119933

We can see a high majority of electronics categories; actually it's well known they are the "leaders" of Amazon, together with Books, so if we want to see who else has also nice results let's try to see what happens if we drop *Electronics* categories:

Categories	Count	Average Overall
Movies,Movies & TV	1291511	4.0185534618
Games,Apps for Android	536135	3.99561677563
TV,Movies & TV	351045	4.4227577661
Baby	160792	4.21411513011

Grocery & Gourmet Food	145829	4.24051457529
Cases,Basic Cases,Cell Phones & Accessories	72493	4.10490668065
Apps for Android,Entertainment	39424	3.56166294643
Amazon Instant Video	37126	4.20952970964
Xbox 360,Video Games,Games	27003	3.99200088879
Games,Video Games,PlayStation 3	25302	4.09971543751
PC,Video Games,Games	24765	3.75045427014
CDs & Vinyl,Pop,Rock	23146	4.27991877646
Productivity,Apps for Android	17162	3.8216408344
Games,Video Games,Apps for Android	16505	4.19515298394
Screen Protectors,Accessories,Cell Phones & Accessories	16087	4.10132405048
Movies & TV	12843	4.06493809858
Toys & Games,Board Games,Games	12750	4.35050980392
CDs & Vinyl,Metal,Pop Metal	12015	4.26433624636
Kids,Apps for Android	11209	4.0569185476
Education,Apps for Android	10736	4.14232488823

5.5 Which are the most reviewed products?

It was quite easy to extract this information (listing 9):

Title	Price	Reviews Count
SanDisk Ultra 64GB MicroSDXC Class 10 UHS ...	36.79	4915
AmazonBasics High-Speed HDMI Cable - 15 Feet (4.6 Meters)	11.99	4143
Google Chromecast HDMI Streaming Media Player	34.49	3798
Mediabridge ULTRA Series HDMI Cable (6 Feet) ...	8.99	3435
Transcend 8 GB Class 10 SDHC Flash Memory Card (TS8GSDHC10E)	7.52	2813
Panasonic RPHJE120D In-Ear Headphone, Orange	7.04	2652
DVI Gear HDMI Cable 2M 6 feet	2.80	2599
AmazonBasics Apple Certified Lightning to USB Cable - 3 Feet	13.99	2542
Roku 3 Streaming Media Player	99.97	2104
eneloop SEC-CSPACER4PK C Size Spacers for use with AA battery cells	9.99	2082
Kindle Fire HD 7", Dolby Audio, Dual-Band Wi-Fi, 16 GB	199.00	2069
Motorola Vehicle Power Adapter micro-USB Rapid Rate Charger	8.95	2021
Tech Armor Privacy Screen Protector for Apple New iPhone 5...	8.95	1961
Garmin Portable Friction Mount - Frustration Free Packaging	19.16	1960
SanDisk 4GB Extreme SDHC Class 10 Memory Card	16.49	1890
SanDisk Cruzer Fit 4GB USB 2.0 Low-Profile Flash Drive	4.75	1884
WD My Passport 2TB Portable External USB 3.0 Hard Drive...	129.00	1866
Kingston Digital 8GB DataTraveler 101 G2 USB 2.0 Drive...	5.38	1812
PowerGen 2.4Amps / 12W Dual USB Car charger...	8.99	1710
Mohu Leaf Paper-Thin Indoor HDTV Antenna - Made in USA	35.99	1581

It's clearly evident that all these product are related to the **Electronics** category.

6 Temporal Analysis

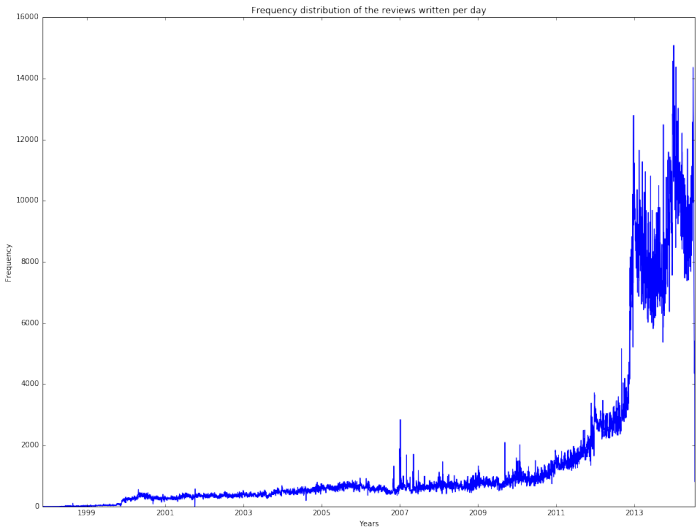
How did the reviews change over time? We had the opportunity to see the evolution of different parameters related to the reviews, so for this part we decided that providing some plots was the best idea. We tried to provide an answer to these questions:

- How many reviews written every day/month/year?
- Distribution of the average overall score: did it change over time?

6.1 Distribution of the reviews over time

We plotted the **frequency distribution** of the reviews, according to different time intervals (daily, monthly, yearly). The main results we inferred are hereby listed:

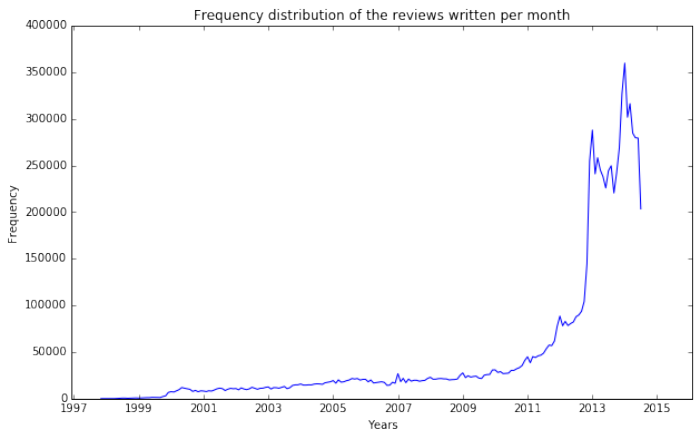
- We can clearly see that, as it could have been said intuitively, the distribution follows a linear trend until 2011, while in 2013 there has been a huge shift upwards;
- If we switch from a daily distribution to a monthly distribution, and from this to a yearly distribution, we can notice that the curve becomes less sharp, since the variation of the number of reviews is much higher if we reduce the time interval;
- Finally, the number of variations in the curve is much higher in the recent years, this is clearly caused by huge number of reviews that have been written, much higher than in the old times (90s and early years of 00s).

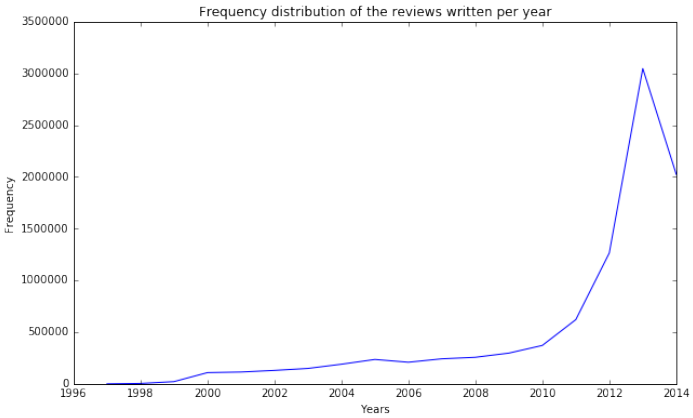


We extracted some basic statistics:

```
The day with the highest number of reviews was: 2014-01-07 with: 15082 reviews
Average reviews per day: 1556.9379819
Standard deviation: 2608.82600895
```

In order to find the peaks, we provide a smoother overview of the growth of the number of reviews per year and per month:





For every year, we extract the month with the highest number of reviews:

Year	Peak_Month	Review_Counter
1997	11	8
1998	12	741
1999	12	6683
2000	5	11936
2001	7	11233
2002	7	12183
2003	12	14845
2004	12	18126
2005	8	21596
2006	1	20700
2007	1	26793
2008	12	24926
2009	12	30721
2010	12	40929
2011	12	77398
2012	12	254728
2013	12	327004
2014	1	359637

As we can see, it's clear that most of the peaks during the years were reached during **december**! Just very few ones in summer...
For a better insight, for every month, we extracted the day with the highest number of reviews, such that it was possible to find the **peaks**, and we came up with these results:

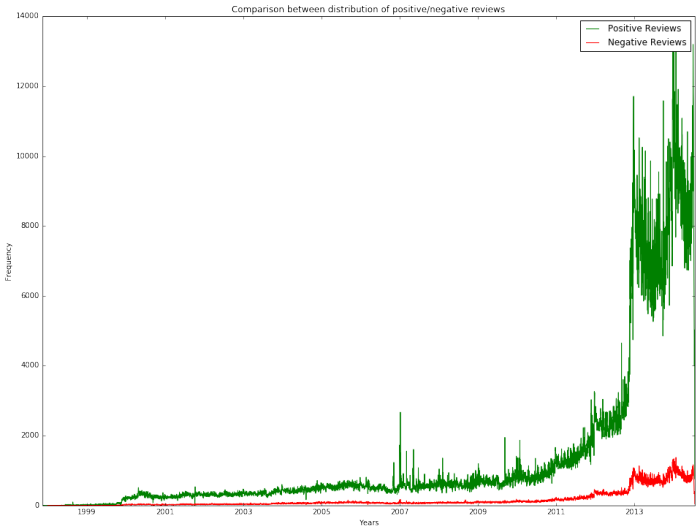
- The big peak in **2007** was reached on the **9th of January**;
- The huge shift in **2012** from an almost linear trend to an exponential and much more vibrant trend started on the **28th of December**.

Summarizing the obtained results:

- We've seen that people has been particularly active in terms of writing reviews in winter months, especially in december/january;
- Lots of peaks were identified close to Christmas period in december: is this maybe a clear signal that people tend to buy a lot of products in that period?

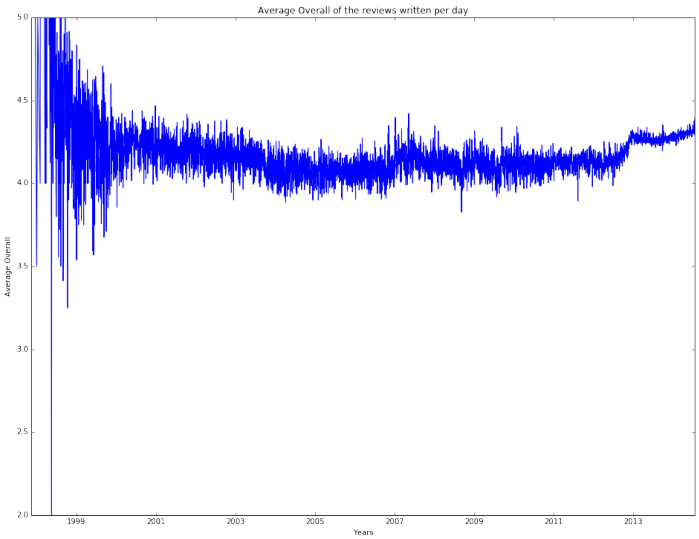
6.2 Distribution of negative/positive reviews and comparison

We can see that, with the cryteria we chose (a review is positive if its overall score is >=3, negative otherwise), there's not much difference: the positive reviews still represent the majority.



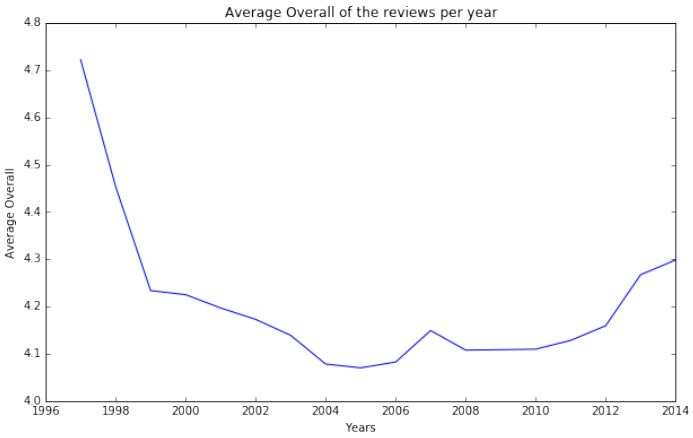
6.3 Average Overall over time

It may be more interesting if we group by `reviewTime` and calculate the average overall of the reviews for each day:




```
The average overall from 1997 to 2014 of all the reviews ever written is:
4.16870308781
```

This result is probably enough to state that people generally are satisfied by Amazon products, so it's really worth it buying something there. For a smoother overview of the average overall over time, let's see the same distribution over years and months: this could be interesting for example in order to figure out what are the periods when people are more likely to give positive reviews/negative reviews.



Now we can go more in depth with the insights, as we already did for the frequency of the reviews: we can extract the peaks for every year/month:

Year	Peak Month	Average Overall
1997	11	4.75
1998	4	4.93154761905
1999	4	4.30570072595
2000	6	4.26382481522
2001	1	4.21575231322
2002	2	4.2033351855
2003	5	4.17172718046
2004	10	4.0985796831
2005	2	4.10352766058
2006	11	4.10570951946
2007	1	4.18142756257
2008	11	4.13426258576
2009	1	4.15186385187
2010	2	4.14377948438
2011	9	4.15204830964
2012	12	4.28339999887
2013	1	4.27669959518

```
| 2014 |     7      |  4.32588527504  |
+------+------------+-----------------+
```

It's not so evident now whether there are some specific months were people's overall scores are high, and the same analysis with the peaks of each month confirmed this.

7 Quantitative text analysis, hashing and machine learning

More than 9 millions of Amazon reviews contain an amount of text information that is as interesting to analyse as hard to handle. We decided to try some machine learning on it, and see if we were able to take something out of such a huge text set.

Each review contains two text fields that we have deemed to be equally important: `summary` and `reviewText`. The concatenation of these two strings gave us the starting point for our data mining.

Bag of words and feature hashing We have decided to build a bag of words model to structure the information of the reviews. Such a matrix would have had 17'583'210 columns; even when saved as a sparse matrix, the data structure took too much space to be handled efficiently.

We decided therefore to apply some feature hashing to the tokens of the bag of word. In particular, we hashed the words in just 5000 different values and made a bag of words, still using the `CountVectorizer` class. We remembered the lesson learned from the course, i.e. that there is still enough information after a feature hashing, and we can save a lot of space and time using this method.

In fact, the feature hashing worked as expected, reducing the processing time a lot and making the overall structure more manageable. But at some point the code of the countvectorizer allocated a big space in memory that caused our laptop to run out of capacity. After digging into the method we found out that part of the problem is that countvectorizer has to save a vocabulary of all the words it sees in order to be able to transform other data on a following round. We didn't need that, being working with hashes. Therefore we found the practical and lightweight `HashingVetorizer` that worked pretty much in the same way but faster and using less memory, still without losing any of the features we needed.

```
1 HashingVectorizer( tokenizer = featuretokenizer, n_features=
    hashbuckets)
```

Generators The resulting data structures, though, were still too big to fit in memory. Our first attempt to solve this was to switch from lists to generators all over the code. We defined a class to be a file handler that yielded the training set incrementally and then, from the same file handler, yielded the rest of the records in another method of the class. The class worked fine and allowed also to split the sets in different chunks with an optional argument.

The class had the following signatures:

```
class FileTrainTestHandler (object):
    def __init__ (self, fname, fraction, requiredvalue):
    def computelen(self, fullname):
    def xtrainG (self, buckets=1):
    def ytrainG (self, buckets=1):
    def xtestG (self, buckets=1):
    def ytestG (self, buckets=1):
```

Our lass improved the performances of the data handling a bit so we decided to go on with the classification

Tree models Starting with a subset of the reviews we have trained a simple random forest classifier to predict the overall vote connected to the review. To make the task easier we have limited the lasses to two: whether or not the rating is above 2.5 stars. The classifier worked fine on the smaller dataset, classifying correctly 91% of the validation records (that were the 20% of the total). We wanted to try a more extreme approach and tried a more fine-grained method, i.e. regression.

We decided to keep the idea of a tree model, and went for the decision tree model. We tested it on a smaller dataset again, and got a validation score of about 0.5 out of 1.

```
dtr = RandomForestClassifier (n_estimators=50)
# and
dtr = DecisionTreeRegressor(max_depth=5, min_samples_leaf=20)
```

Good enough, time to move to the bigger dataset!

Pickle Unfortunately, the classifier required the whole dataset to be fed to it for the training phase. As said, this is something we wouldn't do because of memory constraints. This means that our previous work needed to be changed.

We decided to save the files on disk and fetch them incrementally. The best way of doing this was to use pickle.

The pickle library is a very powerful python tool that is able to save and load any different kind of data structure. As a consequence,it sounded like the solution we needed! We provided to pickle our bag of words yielded by a generator, but again the size turned out to cause a `MemoryError` to be raised. We blamed the old variables to be responsible for that, and planned to save and retrieve the dataset in a fresh kernel.

We tried different pickle sizes to avoid running out of memory, to end up with saving in each different pickle one tenth of the initial reviews. After half of the files the size of a particular chunk turned out to be still too big, so we had to change the file size to $\frac{1}{12}$ of the reviews for the last half of the reviews.

In the same way we pickled the values that we planned to use as classification values i.e. the overall vote and the number of helpful/not helpful mark that the review had got.

SGD models As mentioned, the tree classifiers required the data to be fed at once; it turned out that our optimisations were not enough, and still the data could not fit in RAM.

The final solution we have found is to use an incremental model instead of a tree, that allows us to insert the data in different bunches. An example of such models are the SGD models i.e. stochastic optimization models that follow the gradient descent method.

Using these models we can use the partial fit method that partially trains the classifier/regressor and allows subsequent bunches of data to be used to learn better.

Predicting the ratings Our first exploration at this point was trying to classify the reviews based on the text and the summary and be able to understand if the review was positive or negative.

We have tied to predict the positiveness of the review in terms of being above or below 2.5 (two classes). The result is promising, with an accuracy of 86% on the validation set ($\frac{1}{12}$ of data).

```
Start, pickle loading y, instancing classifier, training,
   validating, re-validating, done!
Classification SGD:
  > using 8359831 values from 9288692
  > validation score: 0.869932
  > 100677 wrong out of 774042 in validation
  > elapsed time: 4104.959 s
```

For the regression the validation score is quite low. We have also counted the number of wrong record (where wrong here means that the value assigned by the regression is ore than 1 away from the true value).

Just about one seventh of the data was misclassified but still the validation score was quite low.

```
Start, pickle loading y, instancing classifier, training,
   validating, re-validating, done!
Regression SGD:
  > using 8359831 values from 9288692
  > validation score: 0.207955
  > 119597 wrong out of 774042 in validation
  > elapsed time: 4790.087 s
```

Predicting the votes We have another important parameter for a review that is the number of users that have voted it (either as very helpful or as useless). We have decided to split the reviews between non-voted (0-1 votes) and voted (ore 2 or ore). Here the classificator struggles more, but seems to still be able to find a common pattern, hitting almost the 70% of records.

```
Start, pickle loading y, instancing classifier, training,
   validating, re-validating, done!
Random Forest C:
  > using 8359831 values from 9288692
  > validation score: 0.685676
  > 150517 wrong out of 774042 in validation
  > elapsed time: 3848.518 s
```

Once again, the regression was way less precise, scoring just 0.03 validation score and misclassifying half of the data

```
Start, pickle loading y, instancing classifier, training,
   validating, re-validating, done!
Regression SGD:
  > using 8359831 values from 9288692
  > validation score: 0.032157
  > 494726 wrong out of 774042 in validation
  > elapsed time: 3956.582 s
```

Predicting the votes What about trying to classify the five different votes each by itself? This task proved to be much harder for the SGD model, but still more than half of the records were corretly labeled. Good enough!

```
Start, pickle loading y, instancing classifier, training,
   validating, re-validating, done!
Classification SGD:
  > using 8359831 values from 9288692
  > validation score: 0.562968
  > 285932 wrong out of 774042 in validation
  > elapsed time: 4511.368 s
```

8 Graph Database

In this section it will be documented how we built a graph database of all the (relevant) Amazon products and the relationships between them. First, we will talk about the schema we implemented: nodes, attributes, relationships. Then, we will talk about the population of the DB: the focus will be on the big size of the metadata file we used as input and the solution we adopted to generate millions of nodes and relationships. Finally, we will show the results of some example queries performed on the huge database.

8.1 Nodes and Relationships

Initially, we were thinking of including in the graph the biggest amount of information available to us, also reviews. Since reviews are deeply analysed in the other sections, we decided to focus only on the products.
In Section 2, we can see what information the `metadata.json` file provides about the single product. Among all the attributes, we decided to include only the *main* ones: title(`name`), asin(`productID`) and `price`. As for the relationships, we decided to include all of them: `ALSO_BOUGHT`, `ALSO_VIEWED`, `BOUGHT_TOGETHER`, `BUY_AFTER_VIEWING`.

8.2 Population

While in week 6 we saw in detail how to query a graph database, we didn't focus on the *creation* of the nodes and relationships: we have just executed a dumb-proof list of commands reading from a .txt file. Looking at this file, we notice that the different categories of nodes(Products, Categories, Suppliers, Customers, Orders) in the Northwind DB have all been loaded from (small) .csv files, while we are dealing with a 9.9GB .json file.
Therefore, the first thing we did was to **filter** our dataset: we removed the books from

the metadata file with a simple script, bringing the file's size to 8GB. Then, we decided not to include in the DB the products without a title and without related products, as they were not meaningful for our analysis. We check this at runtime, when evaluating the line corresponding to the product.
The framework we decided to use is Neo4J. We researched the quite extensive documentation and discovered that an official driver is available for Python: this allows us to connect to the Neo4J shell and execute query directly from an iPython Notebook, for instance. Doing this, we can parse the metadata file, build the parametric query and feed it into the shell. As it is mentioned above, in the defined parsing function, we check whether a product has a title and other related products. If not, we don't include it in the DB.

8.2.1 Products

Our first approach was the most naive: we tried to simply build the string corresponding to the CREATE query concatenating the parsed values. After few thousands of products, both the notebook and the Neo4J shell were struggling, probably due to the high overhead of the Cypher queries.
We jumped back to the documentation and implemented *real* parametric queries, provided by the Python Driver. It is in fact possible to supply a dictionary, with each key corresponding to a parameter that can be referenced in the query statement with curly brackets.

```
product = {"asin": "0000031852",
    "title": "Girls Ballet Tutu Zebra Hot Pink",
    "price": 3.17}
create_query ="""CREATE (p:Product {productID:{asin}})
SET p.name={title}, p.price=toFloat({price})"""
session.run(create_query, product)
```

Doing this, series of queries that share the same fixed schema are efficiently managed by Neo4J. We also researched more about indexes and **constraints**, adding one on the `productID`. This guaranteed the absence of duplicates and also provided indexing of the nodes for faster subsequent queries. With this approach, the performance was of course better, but again we could not get the number of nodes over few hundreds of thousands: single Cypher queries gave still too much of overhead.
The final piece we needed to complete the puzzle was **transactions**, the only part that is not exactly well documented on the web. What we did until this moment was:

1. Instantiate a `driver`
2. Obtain a `session` from the driver
3. *For each product,* run a `CREATE` query on the session
   (a) Implicitly, the session created a `transaction` encapsulating the query
   (b) If successful, the transaction was committed and closed
4. Close the session at the end of the population

After some experiments, we managed to get a reliable, working mechanism. What we did differently was:

1. ... Obtain a session from the driver
2. Gather the parameters for a chunk of 5000 products
3. Begin a transaction in the current session
4. Run 5000 CREATE query inside of this transaction, consuming the result of each of them
5. If the results of all the queries were successfully consumed, commit it
6. Continue with the next chunk of 1000 products
7. ... Close the session at the end of the population

Starting from 6,984,215 products in the `metadata_no_book.json` file, excluding the products with no title or without related products, we populated the graph database with 4,418,851 nodes in 1:16h. The size of the Neo4j DB at this stage was 2.3GB. The full script can be examined in the listing 9.

8.2.2 Relationships

To add the relationships we have followed the same exact logic, only considering chunks of 1000 products, instead of 5000. This is to take into account that each product can have up to four relationships. Each relationship can contain up to 30-40 products, represented by the productID. The process of population of the graph with relationships took much longer, due to the incredibly high number of relationships between the products. It lasted more than 12 hours and the size reached a stellar size of *27GB*! The script can be visioned at the listing 9.

8.3 Querying the Graph

Our initial intent was to carry a true graph analysis of the amazon products, extracting statistics like:

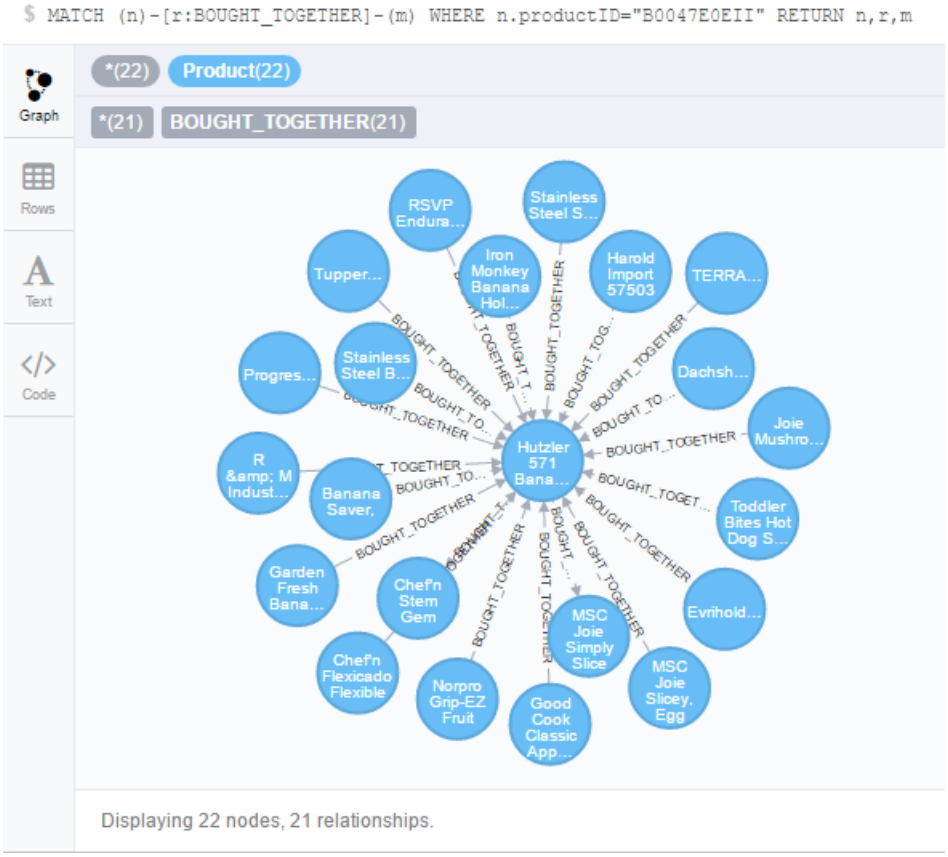
- Diameter of the network(longest *shortest* path)
- Degree centrality(highest number of connections)
- Betweenness centrality(clusters connectors)
- Closeness centrality(highly connected nodes inside a cluster)

Unfortunately, each query that was not traversing the graph through the previously indexed `productID` was taking just **too much time**. So, for example, a query to extract the 5 products with most `ALSO_BOUGHT` connections would look like this:

```
MATCH (p:Product)
RETURN p.name AS product, size( (p)-[:ALSO_BOUGHT]-() ) AS degree
ORDER BY degree DESC LIMIT 5
```

When executed on Neo4J, both via Python driver and Web Shell, the result failed to arrive after few hours of computing, so we eventually gave up. Our suspects are that the problem resides either in the Community Edition of Neo4J, maybe not suitable for such big datasets, or in the limited power of our machines. After all, a graph of 27GB needs to be traversed somehow!

What we have is still a very useful representation of the whole Amazon catalogue, with meaningful attributes to query. We can use the results from our analysis on the review to extract other interesting facts. If we, for example, ask for the products that a customer also bought with the **Hutzler 571 Banana Slicer**, we can visualize this graph:



9 Conclusion

In this report we have discussed the work done for the final project of the course Computational Tools for Big Data. A big dataset sized 30GB has been chosen and different analyses have been carried out on it.

First, we investigated in depth the data we had, choosing what informations were important and which were negligible. We therefore proceeded to filter/clean the data, in different ways according to the different needs required by the different analyses.
Secondly, we evaluated the different tools available to handle the big size of our *larger-than-memory* dataset. We tried to solve the problems every approach provided, and eventually set up a working environment.

The analysis has been carried out on three main unrelated fronts.
Apache Spark represented an extremely useful computational tool to analyze huge clusters, and we had relatively fast execution times. Besides, the combination with Pandas like data structures (DataFrames) and the SQL module gave us the opportunity to perform queries easily and efficiently. We could analyze reviews and metadata of the products and extract relevant and extremely interesting results.
A machine learning part involved different attempts to classify and predict data. We have reached satisfying results with good accuracies, even if we haven't pushed the improvement of the classifier out of the course scope (for example with neural networks). We could hit precisely the rating of almost 60% of the reviews in our test set, with just 5000 hash values as features.
A graph database has been populated with more than 4 millions of nodes and tens of millions of relationships. Queries on the single products have been executed successfully and visual results can easily be consumed.

In this course, and in particular with this project, we had a concrete hands-on experience with big data. It was definitely not easy, we encountered many problems. In the process of overcoming them, we learned how to treat big data and how to efficiently use the tools that this course introduced.

Appendix

3 Environment Setup

SQLite & Pandas

```
import pandas as pd
from sqlalchemy import create_engine # database connection

disk_engine = create_engine(
                'sqlite:///C:\\dtu\\ctbd\\amazon_dataset\\db.db')
counter = 0
index_start = 1
filename = "C:\\dtu\\ctbd\\amazon_dataset\\Musical_Instruments_5.
    json"
with open(filename) as f:
    for line in f:
        df = pd.read_json(line)
        df.index += index_start
        df.to_sql('data', disk_engine, if_exists='append')
        index_start = df.index[-1] + 1
        counter+=1
        if counter%100==0:
            print counter
```

Blaze, Dask and Castra

First of all, all the necessary imports:

```
import ujson
import gzip
import ast
from pandas DataFrame
from toolz import dissoc
from toolz import dissoc, partition_all
from castra import Castra
import time
import datetime
import dask.dataframe as dd
import dask.bag as db
from dask.diagnostics import ProgressBar
```

Initializing variables like **paths, column names and chunk size**:

```
path_to = "C:\\dtu\\ctbd\\amazon_dataset\\"
reviews = "kcore_5.json.gz"
metadata = "metadata.json.gz"
reviews_columns = ['asin', 'reviewerID','reviewerName', 'overall',
    'summary','reviewText','reviewTime','unixReviewTime']
metadata_columns = ['asin','title','price','imUrl','related','
    also_bought','also_viewed','bought_together','salesRank','brand
    ','categories']
chunksize = 5000
```

Implementation of the **user defined functions** we used in the script:

```
#Convert a line of JSON into a cleaned up dict.
def to_json(line):
    return ujson.loads(line)

#Convert a not proper line of JSON (due to single quotes) into a
    cleaned up dict.
def fix_json(line):
    return ast.literal_eval(line)

#Convert a list of JSON strings into a DataFrame
def to_df(batch,filename):
    if filename == 'metadata':
        blobs = map(fix_json,batch)
        df = DataFrame.from_records(blobs, columns=
    metadata_columns)
    else:
        blobs = map(to_json, batch)
        df = DataFrame.from_records(blobs, columns=reviews_columns
    )
    return df

#Create the castra dataset for improved I/O operations with Dask
    DataFrames
#We can work properly on compressed GZ files with gzip library
#The chunk size is 5000, which means that 5000 lines per time will
    be processed
def create_castra(fullpath,chunksize):
    filename = fullpath.split('\\')[-1].split('.')[0] #Used later
    with gzip.open(fullpath,'rb') as f:
        batches = partition_all(chunksize, f)
        castra = None
        for batch in batches:
            df = to_df(batch,filename)
            if castra == None:
                castra = Castra('C:\\dtu\\ctbd\\amazon_dataset\\
    amazon_'+filename+'.castra', template=df)
            castra.extend(df)
```

Finally, the following script is the one that creates the castra files on our disk:

```
print 'Starting the creation of the Castra files...'

#Creating the castra file for metadata
print 'Processing compressed metadata...'
start = time.time()
create_castra(path_to+metadata,chunksize)
end = time.time()
print "Done! Metadata processed in:",datetime.timedelta(seconds=(
    end-start))

#Creating the castra file for the 5_cores
print 'Processing compressed 5_cores...'
start = time.time()
create_castra(path_to+reviews,chunksize)
end = time.time()
print "Done! Reviews data processed in:",datetime.timedelta(
    seconds=(end-start))
```

```
# Start a progress bar for all computations
pbar = ProgressBar(minimum=3.0,dt=0.5)
pbar.register()

# Load data into a dask dataframe:
path_to_castra = 'C:\\dtu\\ctbd\\amazon_dataset\\amazon_kcore_5.
    castra'
df = dd.from_castra(path_to_castra)
df.count().compute()
df.asin.value_counts().nlargest(10).compute()
```

```
[########################################] | 100% Completed |  2min 13.3s

B00FAPF5U0    13550
B0051VVOB2    11981
B0074BW614    10836
030758836X    10552
0439023483    10404
B00DR0PDNE    10139
B007WTAJTO     9771
B006GWO5WK     9008
B005SUHPO6     8963
B0064X7B4A     8808
Name: asin, dtype: int64
```

Spark

Installation on Windows We tried to look at the documentation on the website of Spark, but the steps are not always clear and sometime are given for granted and not even listed. Plus, the official installation guide at `http://spark.apache.org/docs/latest/building-spark.html` requires a huge number of additional software to be run.
We sum up here the steps we followed to have the framework up and running, starting from (but not limited at) the guide available at `http://nishutayaltech.blogspot.dk/2015/04/how-to-run-apache-spark-on-windows7-in.html`.

Python We installed the regular Anaconda Python distribution for Windows on our pc.

Java As a second step, we made sure to have Java Developer Kit (jdk) correctly installed on our Windows machine, and to have the system environment variables `$PATH` and `$JAVA_HOME` correctly set.

Scala The code for Apache Spark is partially written in Scala; we carried out a Scala installation from `http://www.scala-lang.org/download/`. We had to struggle a bit more because of an annoying bug that prevents the program to work correctly if there is any space in the absolute path of Scala's binaries. We also had to manually set the environment variable `$SCALA_HOME` and append `%SCALA_HOME%\bin` in the system `$PATH` variable.

Spark raw install We downloaded the latest Spark prebuilt binaries from `http://spark.apache.org/downloads.html`. Note that we have no Hadoop installed in the windows machine. We extracted them in a folder, making sure once again that no spaces occur in the absolute path to the file.

Hadoop binaries We downloaded the latest Hadoop binaries from `https://github.com/steveloughran/winutils`. In fact, just the executable `bin\winutil.exe` is necessary to run the program. We set the environment variable `$HADOOP_HOME` to be the path of the Hadoop folder containing `bin`.

Logger We made sure to change the logger for Spark to show only warnings; we achieved this modifying in the file `<spark>\conf\log4j.properties` the part: `log4j.rootCategory=INFO` → `log4j.rootCategory=WARN`

Pyspark running options Since on the local machine we plan to use pyspark only in combination with jupyter notebook, we have added two additional environment variables to the system:
`PYSPARK_DRIVER_PYTHON="jupyter"`
`PYSPARK_DRIVER_PYTHON_OPTS="notebook"`

Done this, it is sufficient to fire the command `pyspark` from any windows terminal, and an ipython notebook connected with Spark will open.

Installation on MacOS For MacOS it was a simpler process, we followed the guide available at `https://github.com/mGalarnyk/Installations_Mac_Ubuntu_Windows/blob/master/Spark/Install_Apache_Spark_PySpark_Mac.ipynb` without any particular problems.

Python We installed the regular Python distribution for MacOS on our machine.

Spark We downloaded the 1.6.0 Spark release from `http://spark.apache.org/downloads.html`, the version pre-built for Hadoop 2.6. We extraced the folder in our home folder.

Java We made sure to have Java Developer Kit (jdk) correctly installed on our machine.

Jupyter Notebook We make sure to have downloaded and installed Jupyter Notebook through pip.

bash_profile The next step was to edit our bash_profile. It is an hidden file which contains all the shell environment variables, configurations and aliases. We added the following 4 lines:

```
export SPARK_PATH=~/spark-1.6.0-bin-hadoop2.6
export PYSPARK_DRIVER_PYTHON="jupyter"
export PYSPARK_DRIVER_PYTHON_OPTS="notebook"
alias snotebook='\$SPARK_PATH/bin/pyspark --master local[2]'.
```

These lines serve the purpose of binding the alias 'snotebook' to the execution of a spark jupyter notebook. After having saved the changes in the file, we execute the command 'source bash_profile' and finally 'snotebook'.

4 Apache Spark

Parsing

```
import ujson
from toolz import dissoc
import datetime
import re
import time
from pyspark.sql.types import *
import nltk
from nltk.corpus import stopwords
from wordcloud import WordCloud
stopwords = stopwords.words('english')
from string import punctuation
import re
from matplotlib import pyplot as plt
import random
%matplotlib inline
import pprint
import pickle
from prettytable import PrettyTable
import numpy as np
import operator
from dateutil.rrule import rrule, MONTHLY
import matplotlib.dates as mdates

#Function used to convert a UNIX timestamp to a proper datetime
    object (more human-readable):
def convert_date(date):
    date = datetime.date.fromtimestamp(int(date))
    return date

#Main features:
#Remove reviewerName (not interesting)
#Convert UNIX timestamp in date object
def reviews_parser(line):
    try:
        line = ujson.loads(line)
        line['productID'] = line['asin']
        line['reviewTime'] = convert_date(line['unixReviewTime'])
    except:
        line['reviewTime'] = None
    return dissoc(line,'reviewerName','unixReviewTime','asin')

#Metadata parser, we use this to clean the dictionary and make it
    more usable for a DataFrame
#There are some records that lack some keys, that's why we need to
     set them as None where missing
def metadata_parser(line):
    d = eval(line)
    d['productID'] = d['asin']

    #Categories are a list of lists, for our purposes we can
    flatten the whole list
    if not d.has_key('categories'):
        d['categories']=None
```

```
    else:
        d['categories'] = list(set([category for set_of_categories
    in d['categories']
                                    for category in
    set_of_categories]))

    #SalesRank is a dictionary, but it's more convenient to have
    two separate attributes
    if not d.has_key('salesRank'):
        d['salesRankCategories']=None
        d['salesRankValues'] = None
    else:
        d['salesRankCategories'] = d['salesRank'].keys()
        d['salesRankValues'] = d['salesRank'].values()

    #Setting as None other possible missing keys
    other_keys = ['related','description','brand','price']
    for key in other_keys:
        if not d.has_key(key):
            d[key]=None
    return dissoc(d,'asin','salesRank')

path = '/home/ec2-user/amazon_dataset/'
pickled_objects_folder = 'pickled_objects/'

f_reviews = path+'kcore_5_no_books.json'
f_metadata = path+'metadata_no_books.json'

#Definition of schema for the reviews DataFrame
string_fields = ['productID','reviewText','reviewerID','summary']
fields = [StructField(field_name, StringType(), False) for
    field_name in string_fields]
fields+=[StructField('reviewTime',DateType(),True),
         StructField('overall',FloatType(),False),
         StructField('helpful',ArrayType(IntegerType(),
    containsNull=False),False)]
reviews_schema = StructType(fields)

#Open the file with Spark, apply the parser function and
    initialize the DataFrame
reviews = sc.textFile(f_reviews)
reviews = reviews.map(reviews_parser)
reviews = reviews.toDF(reviews_schema)

#Definition of schema for the metadata DataFrame
fields = [StructField('productID',StringType(),False),
          StructField('brand',StringType(),True),
          StructField('categories',ArrayType(StringType(),True),
    True),
          StructField('description',StringType(),True),
          StructField('imUrl',StringType(),True),
          StructField('price',FloatType(),True),
          StructField('title',StringType(),True),
          StructField('related',StructType([StructField('
    also_bought',ArrayType(StringType(),True),True),
                                    StructField('also_viewed',
    ArrayType(StringType(),True),True),
```

```
                                    StructField('bought_together',
    ArrayType(StringType(),True),True),
                                    StructField('buy_after_viewing',
    ArrayType(StringType(),True),True)]),True),
           StructField('salesRankCategories',ArrayType(StringType()
    ,True),True),
           StructField('salesRankValues',ArrayType(IntegerType(),
    True),True)]
metadata_schema = StructType(fields)

#Open the file with Spark, apply the parser function and
    initialize the DataFrame
metadata = sc.textFile(f_metadata)
metadata = metadata.map(metadata_parser)
metadata = metadata.toDF(metadata_schema)
```

Dataset Schemas

```
Reviews schema:
    root
 |-- productID: string (nullable = false)
 |-- reviewText: string (nullable = false)
 |-- reviewerID: string (nullable = false)
 |-- summary: string (nullable = false)
 |-- reviewTime: date (nullable = true)
 |-- overall: float (nullable = false)
 |-- helpful: array (nullable = false)
 |    |-- element: integer (containsNull = false)

Metadata schema:
root
 |-- productID: string (nullable = false)
 |-- brand: string (nullable = true)
 |-- categories: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- description: string (nullable = true)
 |-- imUrl: string (nullable = true)
 |-- price: float (nullable = true)
 |-- title: string (nullable = true)
 |-- related: struct (nullable = true)
 |    |-- also_bought: array (nullable = true)
 |    |    |-- element: string (containsNull = true)
 |    |-- also_viewed: array (nullable = true)
 |    |    |-- element: string (containsNull = true)
 |    |-- bought_together: array (nullable = true)
 |    |    |-- element: string (containsNull = true)
 |    |-- buy_after_viewing: array (nullable = true)
 |    |    |-- element: string (containsNull = true)
 |-- salesRankCategories: array (nullable = true)
```

```
|    |-- element: string (containsNull = true)
|-- salesRankValues: array (nullable = true)
|    |-- element: integer (containsNull = true)
```

Example Rows

```
reviews.show(10)
```

```
+----------+--------------------+--------------+--------------------+----------+-------+--------+
| productID|          reviewText|    reviewerID|             summary|reviewTime|overall| helpful|
+----------+--------------------+--------------+--------------------+----------+-------+--------+
|1384719342|Not much to write...|A2IBPI20UZIR0U|                good|2014-02-28|    5.0|  [0, 0]|
|1384719342|The product does ...|A14VAT5EAX3D9S|                Jake|2013-03-16|    5.0|[13, 14]|
|1384719342|The primary job o...|A195EZSQDW3E21|It Does The Job Well|2013-08-28|    5.0|  [1, 1]|
|1384719342|Nice windscreen p...|A2C00NNG1ZQQG2|GOOD WINDSCREEN F...|2014-02-14|    5.0|  [0, 0]|
|1384719342|This pop filter i...| A94QU4C90B1AX|No more pops when...|2014-02-21|    5.0|  [0, 0]|
|B00004Y2UT|So good that I bo...|A2A039TZMZHH9Y|      The Best Cable|2012-12-21|    5.0|  [0, 0]|
|B00004Y2UT|I have used monst...|A1UPZM995ZAH90|Monster Standard ...|2014-01-19|    5.0|  [0, 0]|
|B00004Y2UT|I now use this ca...| AJNFQI3YR6XJ5|Didn't fit my 199...|2012-11-16|    3.0|  [0, 0]|
|B00004Y2UT|Perfect for my Ep...|A3M1PLEYNDEYO8|         Great cable|2008-07-06|    5.0|  [0, 0]|
|B00004Y2UT|Monster makes the...| AMNTZU1YQN1TH|Best Instrument C...|2014-01-08|    5.0|  [0, 0]|
+----------+--------------------+--------------+--------------------+----------+-------+--------+
only showing top 10 rows
```

```
metadata.select('productID','price','title','categories').show(10)
```

```
+----------+-----+--------------------+--------------------+
| productID|price|               title|          categories|
+----------+-----+--------------------+--------------------+
|0000143561|12.99|Everyday Italian ...|[Movies, Movies &...|
|0000037214| 6.99|Purple Sequin Tin...|[Novelty, Costume...|
|0000032069| 7.89|Adult Ballet Tutu...|[Other Sports, Da...|
|0000031909|  7.0|Girls Ballet Tutu...|[Dance, Other Spo...|
|0000032034| 7.87|Adult Ballet Tutu...|[Other Sports, Da...|
|0000589012|15.95|Why Don't They Ju...|[Movies, Movies &...|
|0000031852| 3.17|Girls Ballet Tutu...|[Dance, Other Spo...|
|0000032050|12.85|Adult Ballet Tutu...|[Other Sports, Da...|
|0000695009| null|Understanding Sei...|[Movies, Movies &...|
|000107461X| null|Live in Houston [...|[Movies, Movies &...|
+----------+-----+--------------------+--------------------+
only showing top 10 rows
```

Table Names

```
reviews.registerTempTable('reviews')
metadata.registerTempTable('metadata')
```

Counting rows

```
spark.sql('SELECT count(*) AS Total_Reviews FROM reviews').show()
spark.sql('SELECT count(*) AS Number_Of_Products FROM metadata').
    show()
```

Example Queries

```
reviews.filter(reviews.reviewTime.startswith('2013-12-25')).show
    (10)
```

```
+----------+--------------------+--------------+--------------------+----------+-------+-------+
| productID|          reviewText|    reviewerID|             summary|reviewTime|overall|helpful|
+----------+--------------------+--------------+--------------------+----------+-------+-------+
|B000068O1N|Haven't used it y...|A3UPEG3LRWOX16|         Nice Cables|2013-12-25|    4.0| [0, 0]|
|B0002E1G5C|Overall, I like t...|A19Q4B515ENF9C|Does the job but ...|2013-12-25|    4.0| [0, 0]|
|B0002E1I2I|I bought my new g...|A19Q4B515ENF9C|I guess this work...|2013-12-25|    3.0| [0, 0]|
|B0006NDF8A|This is fine for ...|A19Q4B515ENF9C|Works fine and pr...|2013-12-25|    5.0| [0, 0]|
|B002NG7DEK|This strap works ...|A19Q4B515ENF9C|Works great for m...|2013-12-25|    4.0| [0, 0]|
|B005LYIW3W|This pedal is ver...|A2W03JTG0K5OKF|         Great Tone!|2013-12-25|    5.0| [0, 0]|
|B0006IX7Y2|Easy to use, easy...|A2KS03QLEXPO9W|       Great product|2013-12-25|    5.0| [0, 0]|
|B0007LVJ20|Easy to use. Keep...|A1MZ2BJHUSTW0W|    safe and secure.|2013-12-25|    5.0| [0, 0]|
|B0009IQXXG|Does as stated, b...|A1JSY4NTAEFOPL|its okay, that's ...|2013-12-25|    3.0| [0, 1]|
|B000GX9FSY|Purchased these f...| ACX25O6EVKQL6|      BMW E-36 318is|2013-12-25|    5.0| [0, 0]|
+----------+--------------------+--------------+--------------------+----------+-------+-------+
only showing top 10 rows
```

```
reviews.filter(reviews.helpful[1]-reviews.helpful[0]>20).show()
```

```
+----------+--------------------+--------------+--------------------+----------+-------+----------+
| productID|          reviewText|    reviewerID|             summary|reviewTime|overall|   helpful|
+----------+--------------------+--------------+--------------------+----------+-------+----------+
|B0002CZV82|This is a cheap p...|A2B58VXLLOFQKR|         It distorts|2009-11-13|    1.0|   [2, 30]|
|B000T9NKEU|It was Not what I...|A3MJARDJ31M698|       Cost Too Much|2010-06-05|    1.0|   [0, 30]|
|B000ULAP4U|garbage if you li...|A1B3CNORXB1USI|almost zero stars...|2012-06-12|    1.0|  [11, 101]|
|B0042F1L4S|My review(s) for ...|A2OPSPL8LSSJPC|this amp is ooooo...|2012-01-24|    1.0|   [5, 30]|
|B004OU2IQG|allotoday went to...|A1B3CNORXB1USI|so so for  conver...|2012-10-25|    1.0|  [17, 65]|
|B004TE5HBU|Just use a smartp...|A16Z3HTUIYPDH8|    Use a smartphone|2013-03-13|    1.0|   [5, 38]|
|B0002U2V1Y|I have a 1998 Ram...| A4WCZVA328QB4|Claying is more f...|2009-09-24|    3.0|[137, 177]|
|B0002U2V1Y|After reading num...| A1VQHH85U7PX0|       Wwww...wow!!!|2005-07-15|    5.0|[134, 172]|
|B000AAMY86|The installation ...| AN81JUYW2SL24|High Utility Fact...|2008-02-07|    4.0|[171, 195]|
|B000BNYMX2|People need to un...|A2WYCJOJY6QSJI|Learn about oil. ...|2011-10-07|    4.0|[210, 234]|
|B000E8T810|Filter is one of ...|A1E3RTMBHQ2RJE|               Price|2010-08-29|    1.0|   [2, 39]|
|B000P17NXQ|Needed a brake co...|A3P15O8PZOUADD|                Nice|2010-10-14|    3.0|   [6, 29]|
|B000RXNLK6|I actually though...|  AZBLP8S3CHH3|  Not as ultimate...|2009-03-25|    3.0|   [3, 37]|
|B000XECJES|I love microfiber...|A3IOCPLIMYDBCD|Hello My Little C...|2008-09-07|    5.0|[133, 159]|
|B0017K69MA|Walmart sells it ...|A1E3RTMBHQ2RJE|Does what it says...|2009-05-03|    1.0|   [8, 40]|
|B003IS3HV0|In an attempt to ...| A9X2MLWQOALBW|Leaves leather oi...|2012-04-08|    2.0|[270, 296]|
|B003POLA84|Has MAACO repaint...| AY4B9XKR85TTK|           Bad Paint|2013-02-22|    3.0|   [1, 27]|
|B003UM7B98|I ordered this la...|A3GX7EVYDJZTWJ|Jack Review / Rev...|2013-03-12|    4.0|   [2, 34]|
|B00480FIBE|I do not own thes...|A1UVOKK606C6QH|Warning about tin...|2012-10-29|    1.0|   [0, 21]|
|B00FJXKE5E|This Automatic ou...|A100WO06OQR8BQ|              Stinko|2013-10-28|    1.0|   [3, 25]|
+----------+--------------------+--------------+--------------------+----------+-------+----------+
only showing top 20 rows
```

```
reviews.groupBy('reviewerID').count().withColumnRenamed("count", "
    total").sort('total',ascending=False).filter('total%2==0').show
    (10)
```

```
+--------------+-----+
|    reviewerID|total|
+--------------+-----+
| A9Q28YTLYREO7|  578|
| A8IFUOL8S9BZC|  256|
|A1GN8UJIZLCA59|  246|
|A2OOC7YQJ45LRR|  200|
|A1J5KCZC8CMW9I|  182|
| A8SCX6VUTEO5H|  172|
|A2582KMXLK2PO6|  160|
|A1IKOYZVFHO1XP|  144|
|A12R54MKO17TWO|  132|
|A2ETZ7GF5B1712|  130|
+--------------+-----+
only showing top 10 rows
```

Basic Stats

```
/*Number of voters (output not displayed below)*/
SELECT avg(helpful[1]) FROM reviews WHERE helpful[0]>0 AND helpful
    [1]>0
SELECT avg(helpful[1]-helpful[0]) AS Avg_Not_Helpful_Voters FROM
    reviews WHERE helpful[0]>0 AND helpful[1]>0
SELECT avg(helpful[0]) AS Avg_Helpful_Voters FROM reviews WHERE
    helpful[0]>0 AND helpful[1]>0

/*Most voted reviews*/
SELECT * FROM reviews ORDER BY helpful[1] DESC
```

```
+----------+--------------------+--------------+--------------------+----------+-------+--------------+
| productID|          reviewText|    reviewerID|             summary|reviewTime|overall|       helpful|
+----------+--------------------+--------------+--------------------+----------+-------+--------------+
|B0047E0EII|For decades I hav...|A1TTA1UUGY4WY4|No more winning f...|2011-03-03|    5.0|[52176, 52861]|
|B0074BW614|I've been an iPad...|A3QCF8CVINEXB8|You Get What You ...|2012-09-17|    4.0|[30735, 31453]|
|B007FTE2VW|Fundamentally, Si...|A1DQOJ8PLXVPCO|    What a lousy toy|2013-03-06|    1.0|[10279, 10533]|
|B0029OSN4U|The primary diffe...|A3FKPBNI7UWQFW|Salon Laser Hair ...|2010-04-05|    4.0|  [9634, 9717]|
|B000FKBCX4|See those older 5...|A3284KYDZ00BZA|Dumbed down exper...|2008-09-07|    1.0|  [8606, 9403]|
|B00B2V66VS|This game is the ...|A1SIOM22E22TYY|THIS GAME IS AWES...|2013-01-24|    5.0|  [8116, 9258]|
|B008PPGFZG|If you DO NOT hav...| AK7T18HR8HYU2|READ APP DESCRIPT...|2012-11-18|    5.0|  [7574, 8549]|
|B00B71TNWC|I want to start t...|A2Q14JXZX4R807| A mother's struggle|2013-12-08|    5.0|  [8115, 8181]|
|B00DR0PDNE|I actually ordere...|A3OELYCRWQIUE1|An Actual Chromec...|2013-07-27|    5.0|  [7802, 8152]|
|B0074BW614|I have been an Am...| APMAMNLZCQJ76|         Exceptional|2012-09-17|    5.0|  [7273, 7637]|
|B00DR0PDNE|So I have no idea...|A13C2DL0LJ2JBC|Honest Review fro...|2013-07-24|    4.0|  [6225, 6798]|
|B0086700CM|This is a very ad...|A18HE80910BTZI|    Full Five Stars!|2012-05-27|    5.0|  [5898, 6771]|
|B003ZSJ212|This is a genuine...|A2K0Y55A5KQSU7|The Girlfriend Te...|2011-09-19|    4.0|  [6084, 6510]|
|B001FA1O18|Having had a chan...| AENLD33KQ6MJ4|The Lines Between...|2010-09-07|    5.0|  [5971, 6310]|
|B009UX2YAC|this is the best ...|A1RQSEP9GQKURW|    best game ever!!|2012-10-28|    5.0|  [5279, 6186]|
|B001LYFBHG|Note: I have upda...|A17V9XL4CWTQ6G|The top steam mop...|2009-08-07|    5.0|  [6128, 6174]|
|B004SJ3BCI|This game has no ...|A12U3TLNE93CTY|   Best game...ever?|2011-12-29|    5.0|  [5405, 6144]|
|B002QZ1RS6|This is a nuts se...|A17M1HL6U2GS7M|Rrrrrripppp....ye...|2011-02-28|    5.0|  [5696, 5819]|
|B009ZKSPDK|the game has were...| A2O7CSZTJ8PED|         Yes awesome|2012-11-12|    5.0|  [4741, 5494]|
|B000MDHHO6|I juice at least ...|A1IXJK1NYTSMJU|WOW - Way better ...|2007-08-24|    5.0|  [5247, 5307]|
+----------+--------------------+--------------+--------------------+----------+-------+--------------+
only showing top 20 rows
```

Bad Reviews

```
SELECT * FROM reviews ORDER BY helpful[1]-helpful[0] DESC
```

```
+----------+--------------------+--------------+--------------------+----------+-------+--------------+
| productID|          reviewText|    reviewerID|             summary|reviewTime|overall|       helpful|
+----------+--------------------+--------------+--------------------+----------+-------+--------------+
|B00AKIPBNS|They didnt make e...|A169RTOHC2SJPI|OUT OF STOCK!!!?????|2013-02-04|    1.0|  [20, 1853]|
|B00H5RYIBI|Often times, I lo...|A3CPJYAQMCUGNR|Purchased as a gi...|2014-01-02|    1.0|  [21, 1589]|
|B00IWULQQ2|Best look to the ...|A17XJCP9P9VSL0|Fictional fluff w...|2014-05-22|    1.0|  [24, 1250]|
|B00B2V66VS|This game is the ...|A1SIOM22E22TYY|THIS GAME IS AWES...|2013-01-24|    5.0|[8116, 9258]|
|B001KVZ6HK|Storytelling as a...|A3DGVB3T5QJNRE|About as Exciting...|2012-05-11|    1.0|  [33, 1170]|
|B005J6U77Q|I need to augment...|A3GGD7ZWVWIKOU|Flaccid Logos Spe...|2011-09-08|    1.0|  [24, 1057]|
|B0001VL0K2|Come on now New L...|A22VNXHU6IZ5MT|Come on New Line,...|2009-04-16|    5.0|[1938, 2959]|
|B001BYLFFS|**UPDATE: 9/24/08...|A3E68QNSCABRVW|A complete sham, ...|2008-07-14|    1.0|[1988, 3001]|
|B008PPGFZG|If you DO NOT hav...| AK7T18HR8HYU2|READ APP DESCRIPT...|2012-11-18|    5.0|[7574, 8549]|
|B008PPGFZG|why is it that wh...| ATGHKDL0D7F88|          no just no|2012-08-22|    1.0|[3659, 4605]|
|B00DC7G0GG|This game is basi...|A33YHYE1QJ47K8|This is the resul...|2013-11-22|    1.0|   [21, 930]|
|B009UX2YAC|this is the best ...|A1RQSEP9GQKURW|    best game ever!!|2012-10-28|    5.0|[5279, 6186]|
|B00DC7G0GG|Kids of the world...|A21GTH20R33D6B|Mario is FINALLY ...|2013-11-22|    1.0|   [10, 901]|
|B00A2LFXVI|As a small busine...|A3CPJYAQMCUGNR|       Holy Bat Crap|2012-11-19|    1.0|    [6, 880]|
|B0086700CM|This is a very ad...|A18HE80910BTZI|    Full Five Stars!|2012-05-27|    5.0|[5898, 6771]|
|B002I0GEHI|...well now that ...| AM9K263JQJL7L|the most epic exp...|2011-11-15|    1.0|    [9, 880]|
|B003ZSJ212|These latest scen...|A2IU9TYLUCRNOE|So this is how St...|2011-08-31|    1.0|[2745, 3599]|
|B0053BCML6|I bought this gam...|A2JRMCHFSW3SPG|Where is the comb...|2013-06-13|    1.0|   [50, 897]|
```

```
|B004P7VGF2|Here we have an e...|A3BXMKBH8QNHZS|Polyamorous Propa...|2011-07-26|    1.0|   [19, 825]|
|B000FKBCX4|See those older 5...|A3284KYDZ00BZA|Dumbed down exper...|2008-09-07|    1.0|[8606, 9403]|
+----------+--------------------+--------------+--------------------+----------+-------+------------+
only showing top 20 rows
```

Top Reviews

```
reviews.filter(reviews.helpful[1]>3000).sort(reviews.helpful[1]-
    reviews.helpful[0]).show()
```

```
+----------+--------------------+--------------+--------------------+----------+-------+------------+
| productID|          reviewText|    reviewerID|             summary|reviewTime|overall|     helpful|
+----------+--------------------+--------------+--------------------+----------+-------+------------+
|B0015EWMX8|I actually could ...|A27C6W7DLDD7QK|    Garmin Nuvi 255w|2008-06-30|    5.0|[3052, 3067]|
|B0002IQJ8W|This was my fourt...|A3RMAWK4ELOLRZ|          Mean Girls|2006-01-10|    5.0|[3013, 3038]|
|B002HWJT1A|Bose QC15 vs. Sen...|A2QDYGI5OETYV3|Bose QC15 vs. Sen...|2009-12-11|    5.0|[3198, 3225]|
|B0000U1OCI|Since getting att...|  A6FIAB28IS79|Best of Breed: Th...|2004-09-04|    5.0|[3443, 3475]|
|B004GFN2ZA|I thought it migh...|A1ITCVZ46MUB3A|Review from a Bio...|2011-11-27|    4.0|[3410, 3442]|
|B0023B14TK|If I'm going to s...|A17V9XL4CWTQ6G|The camcorder SHO...|2009-11-07|    5.0|[3553, 3585]|
|B000UUBCNO|My wife and I dec...| AC1YLEFC9AN5X|Great but not per...|2007-09-16|    4.0|[4736, 4768]|
|B0002IQJ8W|After an adamant ...|A3GRANSGMYPXO0|I actually caved ...|2004-10-13|    5.0|[4646, 4682]|
|B003DZ167A|                    |A2GP6XJ1V3MM3L|Good Quality K3 C...|2010-08-26|    5.0|[3145, 3184]|
|B005IGVY6K|I've been looking...| AHXGALGGZ9ND4|Exceptional camer...|2011-12-20|    5.0|[4863, 4907]|
|B000RPVHZU|(update: This rev...|A17V9XL4CWTQ6G|The steam mop SHO...|2009-11-02|    1.0|[5081, 5126]|
|B001LYFBHG|Note: I have upda...|A17V9XL4CWTQ6G|The top steam mop...|2009-08-07|    5.0|[6128, 6174]|
|B003ZX8B3W|Having been a Gar...|A1D27BCSYV7VWH|Great Screen - Gr...|2010-12-04|    4.0|[3134, 3183]|
|B000I1X6PM|Transmission of m...|A1AYOVI7H1CP5X|Great cable, but ...|2008-06-23|    2.0|[3969, 4021]|
|B0028MB3HM|It started simple...|A1Y1L56592H2ZX|Accidental 3 stea...|2009-09-17|    5.0|[5034, 5088]|
|B0052YFYFK|An exemplary show...| AL13DNODA3M7K|Highly comfortabl...|2011-09-10|    5.0|[3557, 3611]|
|B000MDHHO6|I juice at least ...|A1IXJK1NYTSMJU|WOW - Way better ...|2007-08-24|    5.0|[5247, 5307]|
|B00B71TNWC|I want to start t...|A2Q14JXZX4R807| A mother's struggle|2013-12-08|    5.0|[8115, 8181]|
|B00007E7RY|In this review, I...|A2OMOA7T8MINR2| Best Value Purifier|2007-03-05|    5.0|[3941, 4009]|
|B001NO7KUE|** This review ha...|A28Q94CXRMP2SY|Great customer se...|2009-05-19|    2.0|[3716, 3787]|
+----------+--------------------+--------------+--------------------+----------+-------+------------+
only showing top 20 rows
```

Word Clouds

```
def generate_string(texts):
    s = ''
    for summary,review in texts:
        clean_summary = re.sub(r'\d+','',summary)
        clean_review = re.sub(r'\d+','',review)
        tokens = [word.strip(punctuation).lower() for word in
    clean_summary.split() if word not in stopwords]
        tokens+=[word.strip(punctuation).lower() for word in
    clean_review.split() if word not in stopwords]
        s+=' '.join(tokens)
        s+=' '
    return s

def blue_color_func(word, font_size, position, orientation,
    random_state=None, **kwargs):
    return "hsl(240, 100%%, %d%%)" % random.randint(30, 50)

def generate_wordcloud(texts):
    wordcloud = WordCloud(mode='RGBA',
                          width=1400,
                          height=1000,
                          background_color=None,
```

```
                            margin=10,
                            color_func=blue_color_func).generate(
    generate_string(texts))
     plt.figure(figsize=(18,14))
     plt.imshow(wordcloud)
     plt.axis("off")
     plt.show()

#Most voted reviews
result = spark.sql("SELECT * FROM reviews ORDER BY helpful[1] DESC
    ").take(1000)
texts = [(row.asDict()['summary'],row.asDict()['reviewText']) for
    row in result]
print 'Summary:',texts[0][0]
print 'Review:',texts[0][1]

generate_wordcloud(texts)

#Worst reviews
result = spark.sql("SELECT * FROM reviews ORDER BY helpful[1]-
    helpful[0] DESC").take(1000)
texts = [(row.asDict()['summary'],row.asDict()['reviewText']) for
    row in result]
print 'Summary:',texts[0][0]
print 'Review:',texts[0][1]

generate_wordcloud(texts)
```

Product Analysis

Top 20 Products related to the most voted reviews

```
top_20 = reviews.select('productID','helpful').sort(reviews.
    helpful[1].desc()).take(20)
top_20_dict = [row.asDict() for row in top_20]
top_20_df = sc.parallelize(top_20_dict).toDF()
cols = ['productID','title','price','brand','description','
    categories','salesRankCategories','salesRankValues']
products = metadata.select(cols).join(top_20_df,metadata.productID
    ==top_20_df.productID,'inner').\
                                    sort(top_20_df.helpful[1],
    ascending=False).collect()
df = sc.parallelize(products).toDF().drop('productID').drop('
    description')
```

Top-sold products for each category

```
top_sold = spark.sql("SELECT title,salesRankCategories[0] FROM
    metadata M1 WHERE title!='null' AND salesRankValues[0] == (
    SELECT min(salesRankValues[0]) FROM metadata M2 WHERE M1.
    salesRankCategories==M2.salesRankCategories)").collect()
top_sold_no_dup = [dict(t) for t in set([tuple(d.items()) for d in
     top_sold])]
```

```
top_sold_no_dup = sorted(top_sold_no_dup,key=lambda x: x['salesRankCategories[0]'])
for d in top_sold_no_dup:
    print d['salesRankCategories[0]'],'-->',d['title']
```

Most appreciated products according to the Average Overall

```
most_appreciated = spark.sql("SELECT productID,avg(overall) as avg_overall FROM reviews GROUP BY productID HAVING count(*) >1000 ORDER BY avg_overall DESC LIMIT 100").collect()
top_100 = [row.asDict() for row in most_appreciated]
most_appreciated_df = sc.parallelize(top_100).toDF()
most_appreciated_ids = [row.asDict()['productID'] for row in most_appreciated]

cols = ['productID','title','price','brand','description','categories','salesRankCategories','salesRankValues']
products = metadata.select(cols).join(most_appreciated_df,metadata.productID==most_appreciated_df.productID,'inner').\
                                    sort(most_appreciated_df.avg_overall,ascending=False).collect()

df = sc.parallelize(products).toDF().drop('productID').drop('description').drop('salesRankCategories').drop('salesRankValues')
df.show()
```

```
+--------------------+------------------+---------------+--------------------+------------------+
|               title|             price|          brand|          categories|       avg_overall|
+--------------------+------------------+---------------+--------------------+------------------+
|                null|21.520000457763672|           null|[Movies, Movies &...| 4.890322580645162|
|Butler Creek Lula...|27.489999771118164|        MagLula|[Hunting & Fishin...| 4.879078694817658|
|AmazonBasics High...| 9.989999771118164|           null|[Cables & Interco...| 4.859760394644115|
|AmazonBasics USB ...| 5.789999961853027|           null|[Cables & Interco...| 4.829289428076256|
|Mediabridge ULTRA...| 8.989999771118164|    Mediabridge|[Cables & Interco...| 4.801164483260553|
|AmazonBasics High...|11.989999771118164|           null|[Cables & Interco...| 4.800386193579532|
|SanDisk 4GB Extre...|16.489999771118164|        SanDisk|[Electronics, Mem...|4.7936507936507935|
|Samsung Electroni...| 82.98999786376953|        Samsung|[Electronics, Int...| 4.792913385826772|
|BlueRigger High S...| 5.989999771118164| BlueRigger(TM)|[Cables & Interco...| 4.776276276276277|
|Masterpiece Class...|24.989999771118164|           null|   [TV, Movies & TV]| 4.775748721694668|
|AmazonBasics Digi...| 4.449999809265137|           null|[Cables & Interco...|            4.7625|
|Garmin Portable F...| 19.15999984741211|           null|[GPS & Navigation...|4.7454081632653065|
|Firefly: The Comp...|12.989999771118164|20TH Century Fox|   [TV, Movies & TV]| 4.740829346092504|
|eneloop SEC-CSPAC...| 9.989999771118164|           null|[Accessories, Cam...| 4.736311239193084|
|                null|178.97999572753906|           null|[Kindle Paperwhit...| 4.703802535023349|
|SanDisk 2GB Class...|24.989999771118164|        SanDisk|[Electronics, Mem...| 4.695965417867435|
|                null|              null|           null|[Calculators, App...| 4.689097103918228|
|                null|              null|           null|[Games, Video Gam...| 4.684656900539707|
|                null|1.9900000095367432|           null|[Games, Apps for ...| 4.675236806495264|
|                null| 66.16999816894531|           null|[Movies, Movies &...| 4.667638483965015|
+--------------------+------------------+---------------+--------------------+------------------+
```

Most recommended categories

```
average_overall_per_category = spark.sql("SELECT categories,avg(overall),std(overall),count(*) FROM metadata JOIN reviews ON metadata.productID==reviews.productID GROUP BY categories").collect()
```

```
average_overall_per_category = [row.asDict() for row in
    average_overall_per_category]
average_overall_per_category = sorted(average_overall_per_category
    ,key=lambda x: (x['count(1)'],x['avg(overall)']), reverse=True)
average_overall_per_category = filter(lambda x: x['count(1)']>100,
    average_overall_per_category)

#Filtering electronics
average_overall_per_category = [row.asDict() for row in
    average_overall_per_category]
average_overall_per_category = sorted(average_overall_per_category
    ,key=lambda x: (x['count(1)'],x['avg(overall)']), reverse=True)
average_overall_per_category = filter(lambda x: x['count(1)']>100,
    average_overall_per_category)
average_overall_per_category = filter(lambda x: 'Electronics' not
    in x['categories'],average_overall_per_category)
```

Most Reviewed Products

```
most_reviewed_products = reviews.join(metadata,on=metadata.
    productID==reviews.productID).filter(metadata.title!='null').
    groupBy(metadata.productID,metadata.title,metadata.price).count
    ().sort('count',ascending=False).select('title','price','count'
    ).take(20)

with open(pickled_objects_folder+'most_reviewed_products','w') as
    f:
     pickle.dump(most_reviewed_products,f)
```

Temporal analysis

Distribution of the reviews over time

```
reviews_per_day = reviews.groupBy(reviews.reviewTime).count().
    select('reviewTime','count').collect()
with open(pickled_objects_folder+'reviews_per_day','w') as f:
     pickle.dump(reviews_per_day,f)

reviews_per_day = [row.asDict() for row in reviews_per_day]
reviews_per_day = sorted(reviews_per_day,key=lambda x: x['
    reviewTime'])
dates = [d['reviewTime'] for d in reviews_per_day]
frequencies = [d['count'] for d in reviews_per_day]

plt.figure(figsize=(16,12))
plt.plot(dates,frequencies)
plt.title('Frequency distribution of the reviews written per day')
plt.xlabel('Years')
plt.ylabel('Frequency')
plt.show()
```

```
day_most_reviews = max(reviews_per_day,key=lambda x: x['count'])
```

```
print 'The day with the highest number of reviews was:',
    day_most_reviews['reviewTime'],'with:',day_most_reviews['count'
    ],'reviews'

print 'Average reviews per day:',np.mean(frequencies)
print 'Standard deviation:',np.std(frequencies)
```

```
The day with the highest number of reviews was: 2014-01-07 with:
    15082 reviews
Average reviews per day: 1556.9379819
Standard deviation: 2608.82600895
```

```
years = range(1997,2015)
reviews_per_year = {}
for year in years:
    reviews_per_year[year] = 0
    for day in reviews_per_day:
        if day['reviewTime'].year==year:
            reviews_per_year[year]+=day['count']

frequencies = reviews_per_year.values()
plt.figure(figsize=(10,6))
plt.plot(years,frequencies)
plt.title('Frequency distribution of the reviews written per year'
    )
plt.xlabel('Years')
plt.ylabel('Frequency')
plt.show()

start_date = reviews_per_day[0]['reviewTime']
start_date = datetime.date(start_date.year,start_date.month,1)
end_date = reviews_per_day[-1]['reviewTime']
end_date = datetime.date(end_date.year,end_date.month,1)
months = [dt.date() for dt in rrule(MONTHLY, dtstart=start_date,
    until=end_date)]

reviews_per_month = []
for month in months:
    count=0
    for day in reviews_per_day:
        if day['reviewTime'].year==month.year and day['reviewTime'
    ].month==month.month:
            count+=day['count']
    reviews_per_month.append((month,count))

frequencies = [elm[1] for elm in reviews_per_month]
mat_dates = [mdates.date2num(elm[0]) for elm in reviews_per_month]
fig, ax = plt.subplots(1,1,figsize=(10,6))
plt.plot(mat_dates,frequencies)
plt.title('Frequency distribution of the reviews written per month
    ')
plt.xlabel('Years')
plt.ylabel('Frequency')
locator = mdates.AutoDateLocator()
ax.xaxis.set_major_locator(locator)
ax.xaxis.set_major_formatter(mdates.AutoDateFormatter(locator))
plt.show()
```

```
most_reviews_per_year = {}
for elm in reviews_per_month:
    try:
        most_reviews_per_year[str(elm[0].year)].append(elm)
    except:
        most_reviews_per_year[str(elm[0].year)]=[elm]

for year in most_reviews_per_year:
    most_reviews_per_year[year] = max(most_reviews_per_year[year],
    key=operator.itemgetter(1))
    most_reviews_per_year[year] = (most_reviews_per_year[year][0].
    month,most_reviews_per_year[year][1])

pt = PrettyTable()
pt.field_names = ['Year','Peak_Month','Review_Counter']
list_most_reviews_per_year = sorted(most_reviews_per_year.
    iteritems(),key=operator.itemgetter(0))
for couple in list_most_reviews_per_year:
    pt.add_row([couple[0],couple[1][0],couple[1][1]])
print pt

#Initializing the dictionary
most_reviews_per_month = {}
for month in months:
    most_reviews_per_month[str(month.year)+'-'+str(month.month)] =
    None

#For each month, we create a list of the (date,count) tuples for
    the reviews that were written in that month
for month in most_reviews_per_month:
    for day in reviews_per_day:
        if str(day['reviewTime'].year)==month[:4] and str(day['
    reviewTime'].month)==month[5:]:
            try:
                most_reviews_per_month[month].append((day['
    reviewTime'],day['count']))
            except:
                most_reviews_per_month[month]=[(day['reviewTime'],
    day['count'])]

#Finding the max for each month (the day with the highest count)
for month in most_reviews_per_month:
    most_reviews_per_month[month]=max(most_reviews_per_month[month
    ],key=operator.itemgetter(1))

list_most_reviews_per_month = sorted(most_reviews_per_month.
    iteritems(),key=operator.itemgetter(1))
pt = PrettyTable()
pt.field_names = ['Year-Month','Peak_Day','Review_Counter']
for couple in list_most_reviews_per_month:
    pt.add_row([couple[0],couple[1][0].day,couple[1][1]])
print pt
```

Distribution of negative/positive reviews and comparison

```
positive_reviews_per_day = reviews.filter(reviews.overall>=3.0).
    groupBy(reviews.reviewTime).\
                            count().select('reviewTime','count').
    collect()
with open(pickled_objects_folder+'positive_reviews_per_day','w')
    as f:
     pickle.dump(positive_reviews_per_day,f)

negative_reviews_per_day = reviews.filter(reviews.overall<3.0).
    groupBy(reviews.reviewTime).\
                            count().select('reviewTime','count').
    collect()
with open(pickled_objects_folder+'negative_reviews_per_day','w')
    as f:
     pickle.dump(negative_reviews_per_day,f)

positive_reviews_per_day = [row.asDict() for row in
    positive_reviews_per_day]
positive_reviews_per_day = sorted(positive_reviews_per_day,key=
    lambda x: x['reviewTime'])
positive_dates = [d['reviewTime'] for d in
    positive_reviews_per_day]
positive_frequencies = [d['count'] for d in
    positive_reviews_per_day]

negative_reviews_per_day = [row.asDict() for row in
    negative_reviews_per_day]
negative_reviews_per_day = sorted(negative_reviews_per_day,key=
    lambda x: x['reviewTime'])
negative_dates = [d['reviewTime'] for d in
    negative_reviews_per_day]
negative_frequencies = [d['count'] for d in
    negative_reviews_per_day]

plt.figure(figsize=(16,12))
plt.plot(positive_dates,positive_frequencies,'g',label='Positive
    Reviews')
plt.plot(negative_dates,negative_frequencies,'r',label='Negative
    Reviews')
plt.title('Comparison between distribution of positive/negative
    reviews')
plt.xlabel('Years')
plt.ylabel('Frequency')
plt.legend()
plt.show()
```

Average overall over time

```
average_overall_per_day = spark.sql("SELECT reviewTime,avg(overall
    ) FROM reviews GROUP BY reviewTime").collect()
with open(pickled_objects_folder+'average_overall_per_day','w') as
     f:
     pickle.dump(average_overall_per_day,f)

```

```
average_overall_per_day = [row.asDict() for row in
    average_overall_per_day]
average_overall_per_day = sorted(average_overall_per_day,key=
    lambda x: x['reviewTime'])

dates = [d['reviewTime'] for d in average_overall_per_day]
avg_overall = [d['avg(overall)'] for d in average_overall_per_day]

plt.figure(figsize=(16,12))
plt.plot(dates,avg_overall)
plt.title('Average Overall of the reviews written per day')
plt.xlabel('Years')
plt.ylabel('Average Overall')
plt.show()

all_average_overall_values = [d['avg(overall)'] for d in
    average_overall_per_day]
total_average_overall = np.mean(all_average_overall_values)
print 'The average overall from 1997 to 2014 of all the reviews
    ever written is:',total_average_overall

years = range(1997,2015)
average_overall_per_year = {}
for year in years:
    average_overall_per_year[year] = []
    for day in average_overall_per_day:
        if day['reviewTime'].year==year:
            average_overall_per_year[year].append(day['avg(overall
    )'])
    average_overall_per_year[year] = np.mean(
    average_overall_per_year[year])

frequencies = average_overall_per_year.values()
plt.figure(figsize=(10,6))
plt.plot(years,frequencies)
plt.title('Average Overall of the reviews per year')
plt.xlabel('Years')
plt.ylabel('Average Overall')
plt.show()

start_date = sorted(average_overall_per_day,key=lambda x: x['
    reviewTime'])[0]['reviewTime']
start_date = datetime.date(start_date.year,start_date.month,1)
end_date = sorted(average_overall_per_day,key=lambda x: x['
    reviewTime'])[-1]['reviewTime']
end_date = datetime.date(end_date.year,end_date.month,1)
months = [dt.date() for dt in rrule(MONTHLY, dtstart=start_date,
    until=end_date)]

average_overall_per_month = []
for month in months:
    values = []
    for day in average_overall_per_day:
        if day['reviewTime'].year==month.year and day['reviewTime'
    ].month==month.month:
            values.append(day['avg(overall)'])
    average_overall_per_month.append((month,np.mean(values)))
```

```
52
overalls = [elm[1] for elm in average_overall_per_month]
54 mat_dates = [mdates.date2num(elm[0]) for elm in
    average_overall_per_month]
fig, ax = plt.subplots(1,1,figsize=(10,6))
56 plt.plot(mat_dates,overalls)
plt.title('Average Overall of the reviews per months')
58 plt.xlabel('Years')
plt.ylabel('Average Overall')
60 locator = mdates.AutoDateLocator()
ax.xaxis.set_major_locator(locator)
62 ax.xaxis.set_major_formatter(mdates.AutoDateFormatter(locator))
plt.show()
```

```
1 average_overall_per_year = {}
for elm in average_overall_per_month:
3     try:
          average_overall_per_year[str(elm[0].year)].append(elm)
5     except:
          average_overall_per_year[str(elm[0].year)]=[elm]
7
for year in average_overall_per_year:
9     average_overall_per_year[year] = max(average_overall_per_year[
    year],key=operator.itemgetter(1))
      average_overall_per_year[year] = (average_overall_per_year[
    year][0].month,average_overall_per_year[year][1])
11
pt = PrettyTable()
13 pt.field_names = ['Year','Peak Month','Average Overall']
list_average_overall_per_year = sorted(average_overall_per_year.
    iteritems(),key=operator.itemgetter(0))
15 for couple in list_average_overall_per_year:
      pt.add_row([couple[0],couple[1][0],couple[1][1]])
17 print pt

19 #Initializing the dictionary
average_overall_per_month = {}
21 for month in months:
      average_overall_per_month[str(month.year)+'-'+str(month.month)
    ] = None
23
#For each month, we create a list of the (date,overall) tuples for
     the reviews that were written in that month
25 for month in average_overall_per_month:
      for day in average_overall_per_day:
27        if str(day['reviewTime'].year)==month[:4] and str(day['
    reviewTime'].month)==month[5:]:
              try:
29                average_overall_per_month[month].append((day['
    reviewTime'],day['avg(overall)']))
              except:
31                average_overall_per_month[month]=[(day['reviewTime
    '],day['avg(overall)'])]

33 #Finding the max for each month (the day with the highest avg(
    overall))
for month in average_overall_per_month:
```

```
    average_overall_per_month[month]=max(average_overall_per_month
    [month],key=operator.itemgetter(1))

list_average_overall_per_month = sorted(average_overall_per_month.
    iteritems(),key=operator.itemgetter(1))
pt = PrettyTable()
pt.field_names = ['Year-Month','Peak_Day','Review_Counter']
for couple in list_average_overall_per_month:
    pt.add_row([couple[0],couple[1][0].day,couple[1][1]])
print pt
```

8 Graph Database

Population of products

```
from neo4j.v1 import GraphDatabase, basic_auth

driver = GraphDatabase.driver("bolt://localhost", auth=basic_auth(
    "neo4j", "amazon"), encryption=False)
session = driver.session()
```

```
import time
import datetime

def parse_line(line):
    product = eval(line)
    params = {}
    if (product.has_key('title') and product.has_key('related')):
        params['asin'] = product['asin']
        params['title'] = product['title']
        if product.has_key('price'):
            params['price'] = product['price']
        else:
            params['price'] = None
    return params

start = time.time()
session.run("CREATE CONSTRAINT ON (p:Product) ASSERT p.productID
    IS UNIQUE")

with open("C:\\dtu\\ctbd\\amazon_dataset\\metadata_no_books.json",
    "r") as data:
    line_count=0
    products = []
    for line in data:
        products.append(parse_line(line))
        line_count = line_count + 1
        if(line_count % 5000 == 0):#every 5000 lines
            tx = session.begin_transaction()
            for product in products:
                if not (product.has_key('title')):
                    continue
                else:
                    create_query="""CREATE (p:Product {productID:{
    asin}})
```

```
                    SET p.name={title}, p.price=toFloat({price})
    """
                    res = tx.run(create_query, product)
                    for r in res:
                        continue
            tx.commit()
            products = []
            if line_count % 60000 == 0:
                print '%.3f %% completed' % (line_count*1.0/60000
    ,)
session.close()
end = time.time()
print "executed in:", datetime.timedelta(seconds = (end - start))
```

Population of Relationships

```
def parse_line(line):
    product = eval(line)
    params = {}
    if (product.has_key('title') and product.has_key('related')):
        params['asin'] = product['asin']
        params['title'] = product['title']
        for key in ['also_bought', 'also_viewed', 'bought_together
    ', 'buy_after_viewing']:
            if not product['related'].has_key(key):
                params[key] = []
            else:
                params[key] = product['related'][key]
    return params

start = time.time()
with open("C:\\dtu\\ctbd\\amazon_dataset\\metadata_no_books.json",
    "r") as data:
    line_count=0
    params_list = []
    for line in data:
        params_list.append(parse_line(line))
        line_count = line_count + 1
        if(line_count % 1000 == 0): #every 1000 lines
            tx = session.begin_transaction()
            for params in params_list:
                if not (params.has_key('title')):
                    continue
                else:
                    also_bought_query = """MATCH (p:Product),(q:
    Product)
                    WHERE p.productID = {asin} AND q.productID IN
    {also_bought}
                    CREATE UNIQUE (p)-[:ALSO_BOUGHT]-(q)"""
                    res = tx.run(also_bought_query, params)
                    for r in res:
                        continue
                    also_viewed_query = """MATCH (p:Product),(q:
    Product)
```

```
                    WHERE p.productID = {asin} AND q.productID IN {also_viewed}
                    CREATE UNIQUE (p)-[:ALSO_VIEWED]-(q)"""
                    res = tx.run(also_viewed_query, params)
                    for r in res:
                        continue
                    bought_together_query = """MATCH (p:Product),(q:Product)
                    WHERE p.productID = {asin} AND q.productID IN {bought_together}
                    CREATE UNIQUE (p)-[:BOUGHT_TOGETHER]-(q)"""
                    res = tx.run(bought_together_query, params)
                    for r in res:
                        continue
                    buy_after_viewing_query = """MATCH (p:Product),(q:Product)
                    WHERE p.productID = {asin} AND q.productID IN {buy_after_viewing}
                    CREATE UNIQUE (p)-[:BUY_AFTER_VIEWING]-(q)"""
                    res = tx.run(buy_after_viewing_query, params)
                    for r in res:
                        continue
            tx.commit()
            params_list = []
            if line_count % 60000 == 0:
                print '%.1f %% completed' % (line_count*1.0/60000,)
end = time.time()
session.close()
print "executed in:", datetime.timedelta(seconds = (end - start))
```

7 Code of the machine learning part

```
# FIRST FAILED ATTEMPT
'''
    TRYING TO KEEP IT ALL IN MEMORY
'''


path = '../../data/'
fil = 'kcore_5_no_books'
recordamount = 9288691
hashbuckets = 1000
bagofwords_number = 2


import json # to load the files
from sklearn.feature_extraction.text import CountVectorizer, HashingVectorizer
from sklearn.ensemble import RandomForestClassifier # classifier
from sklearn.tree import DecisionTreeRegressor
from sklearn.linear_model import SGDRegressor, SGDClassifier
from scipy.sparse import vstack
import pickle
import time # measure execution time
```

```
import random as rnd # choice and permutations
import re # tokenization

def filt(y, t):
    thresholds = {
        'rat': 2.5,
        'vot': 1,
        'pct': .5
    }
    tt = thresholds[t]
    if y > tt:
        return 1
    return 0


class FileTrainTestHandler (object):
    def __init__ (self, fname, fraction, requiredvalue):
        nameparts = [path,'.json']
        self.xfile = open(fname.join(nameparts), 'r')
        self.yfile = open(fname.join(nameparts), 'r')
        self.frac = float(fraction)
        self.filelen = recordamount
        if self.filelen == None:
            self.filelen = self.computelen(fname.join(nameparts))
        self.fortrain = int(self.frac * self.filelen)
        print 'for training %d,' % (self.fortrain,),
        self.reqval = requiredvalue
        self.xtot = 0
        self.ytot = 0

    def nomore(self, xy, traintest):
        a = None
        if xy == 'x' and traintest == 'train':
            a= self.xtot < self.fortrain
        if xy == 'x' and traintest == 'test':
            a= self.xtot < self.filelen
        if xy == 'y' and traintest == 'train':
            a= self.ytot < self.fortrain
        if xy == 'y' and traintest == 'test':
            a= self.ytot < self.filelen
        if a==None:
            raise 'error'

        return not a

    def computelen(self, fullname):
        print 'computelen,',
        i=0
        with open(fullname, 'r') as f:
            for _ in f:
                i += 1
                if (i%10e5 == 0):
                    print i,
        print 'ok %d,' % (i),
        return i

    def xallG(self):
```

```
79          for j in self.xtrainG():
                yield j
81          for j in self.xtestG():
                yield j
83
    def xtrainG (self, buckets=1):
85          i=0
            numrb = 1+int(self.filelen * 1.0 / buckets)
87          if self.nomore('x', 'train'):
                print '---',
89              return

91          for i, l in enumerate(self.xfile):
                self.xtot +=1
93              if (i%1e5 == 1):
                    print '1x%d' % (i,) ,
95              js = json.loads(l)
                ret = js['reviewText'] + ' ' + js['summary']
97
                if self.reqval == 'rat' or self.reqval == 'vot':
99                  yield ret.lower()
                elif self.reqval == 'pct':
101                 if js['helpful'][1] != 0:
                        yield ret.lower()
103                 else:
                        continue
105             else:
                    raise 'not recognized!!!'
107             if (self.xtot+1 > self.fortrain) or (i+1 > numrb):
                    # in total I finished the train or
109                 #   next goes more than 1 out of bucket
                    break
111         print 'xtrn-%d' % (self.xtot,) ,


113
    def ytrainG (self, buckets=1):
115         i=0
            numrb = 1+int(self.filelen * 1.0 / buckets)
117         if self.nomore('y', 'train'):
                print '---',
119             return

121         for i, l in enumerate(self.yfile):
                self.ytot += 1
123             if (i%1e5 == 1):
                    print '1y%d' % (i,) ,
125             js = json.loads(l)
                ret = js['reviewText'] + ' ' + js['summary']
127
                if self.reqval == 'rat':
129                 yield js['overall']
                elif self.reqval == 'vot':
131                 yield js['helpful'][1]
                elif self.reqval == 'pct':
133                 if js['helpful'][1] != 0:
                        yield js['helpful'][0]* 1.0 / js['helpful'][1]
135                 else:
```

```
                    continue
137             else:
                    raise 'not recognized!!!'
139
                if self.ytot+1 > self.fortrain or i+1 > numrb:
141                 # next one reaches the train or
                    #   next goes more than 1 out of bucket
143                 break
            print 'ytrn-%d' % (self.xtot,) ,
145


147     def xtestG (self, buckets=1):
            i=0
149         numrb = 1+int(self.filelen * 1.0 / buckets)

151         for i, l in enumerate(self.xfile):
                if (i%1e5 == 1):
153                 print '2x%d' % (i,) ,
                js = json.loads(l)
155             ret = js['reviewText'] + ' ' + js['summary']

157             if self.reqval == 'rat' or self.reqval == 'vot':
                    yield ret.lower()
159             elif self.reqval == 'pct':
                    if js['helpful'][1] != 0:
161                     yield ret.lower()
                    else:
163                     continue
                else:
165                 raise 'not recognized!!!'
                if i+1 > numrb: # next goes more than 1 out of bucket
167                 break
            if ( (i-1)%1e5 != 0 ):
169             print '2x%d' % (i-1,) ,

171         self.xfile.close()


173
    def ytestG (self, buckets = 1):
175     i=0
        numrb = 1+int(self.filelen * 1.0 / buckets)
177
        for i, l in enumerate(self.yfile):
179         if (i%1e5 == 1):
                print '2y%d' % (i,) ,
181         js = json.loads(l)
            ret = js['reviewText'] + ' ' + js['summary']
183
            if self.reqval == 'rat':
185             yield js['overall']
            elif self.reqval == 'vot':
187             yield js['helpful'][1]
            elif self.reqval == 'pct':
189             if js['helpful'][1] != 0:
                    yield js['helpful'][0]* 1.0 / js['helpful'][1]
191             else:
                    continue
```

```
193             else:
                    raise 'not recognized!!!'
195             if i+1 > numrb: # next goes more than 1 out of bucket
                    break
197         if ( (i-1)%1e5 != 0 ):
                print '2y%d' % (i-1,) ,
199         self.yfile.close()


201
def rehash():
203     return CountVectorizer( tokenizer = featurehashtokenizer,
    vocabulary=range(0,hashbuckets) )

205 def classificationorregression (cORr, whichvalue, f):
        print 'Start,',
207     start_fh = time.time()

209     fra = 0.8
        ftth = FileTrainTestHandler(f, fra, whichvalue)
211
        values = []
213     thresholds = {
            'rat': 1,
215         'vot': 1,
            'pct': .05
217     }
        summary = ''
219     print 'bagofwording,',

221     cv = rehash()
        x = [cv.fit_transform(ftth.xtrainG(bagofwords_number))]
223
        for i in range(1,bagofwords_number):
225         if ftth.nomore('x','train'):
                break
227
            cv = rehash()
229         x.append(
                cv.fit_transform(ftth.xtrainG(bagofwords_number))
231         )
        x = vstack(x)
233


235     cv = rehash()

237     yA = yB = ''

239     print 'instancing classifier,',
        dtr = None
241     if cORr:
            dtr = RandomForestClassifier (n_estimators=50)
243         yA = (filt(y, whichvalue) for y in ftth.ytrainG())
            yB = (filt(y, whichvalue) for y in ftth.ytestG())
245         summary += 'Random Forest C:'
        else:
247         dtr = DecisionTreeRegressor(max_depth=5, min_samples_leaf
    =20)
```

```
        yA = ftth.ytrainG()
        yB = ftth.ytestG()
        summary += 'Decision Tree R:'
    l = int(x.shape[0] * fra)
    summary += '\n  > using %d values from (r, c) = %s' % (l, str(x.shape))

    print 'training,',
    dtr.fit(x,
            list(yA))

    print 'validating,',
    yb = list(yB)
    xt = cv.fit_transform(ftth.xtestG())
    summary += '\n  > validation score: %f' % (
        dtr.score(
            xt, yb
            )
        ,)

    print 're-validating,',
    a = b = 0
    for kkk in xrange(xt.shape[0]):
        if thresholds[whichvalue] <= dtr.predict (xt[kkk]) - yb[kkk]:
            a+=1
        else:
            b+=1

    summary += '\n  > %d wrong out of %d in validation' % (a,b+a)

    end_fh = time.time()
    summary += '\n  > elapsed time: %.3f s' % (end_fh - start_fh,)

    print 'done!'

    return summary



print classificationorregression(True, 'rat', fil)
print classificationorregression(False, 'rat', fil)
```

Succesful attempt

```
# LOADING THE PICKLE FILES AND USING THEM


path = '../../data/'
fil = 'kcore_5_no_books'
recordamount = 9288691
hashbuckets = 1000
bagofwords_number = 2

```

```
import json # to load the files
from sklearn.feature_extraction.text import CountVectorizer,
    HashingVectorizer
from sklearn.ensemble import RandomForestClassifier # classifier
from sklearn.tree import DecisionTreeRegressor
from sklearn.linear_model import SGDRegressor, SGDClassifier
from scipy.sparse import vstack
import pickle
import time # measure execution time
import random as rnd # choice and permutations
import re # tokenization

def filt(y, t):
    thresholds = {
        'rat': 2.5,
        'vot': 1,
        'pct': .5
    }
    tt = thresholds[t]
    if y > tt:
        return 1
    return 0


def classificationorregression (cORr, whichvalue, f):
    print 'Start,', #9288691
    start_fh = time.time()

    fra = 0.8

    values = []
    thresholds = {
        'rat': 1,
        'vot': 1,
        'pct': .05
    }
    summary = ''
    print 'pickle loading y,',

    with open(destpath + fil + whichvalue + '.pickle', 'r') as f:
        y = pickle.load(f)

    print 'instancing classifier,',
    dtr = None
    if cORr:
        dtr = SGDClassifier ()
        summary += 'Random Forest C:'
        classes = [0, 1] #for the 5 classes:          classes =
    range(1,6)

        y = [filt(a, whichvalue) for a in y]
    else:
        dtr = SGDRegressor()
        summary += 'Decision Tree R:'
        classes = None
    summary += '\n  > using %d values from %d' % (0.9*len(y)+9,
    len(y))
```

```
65
    print 'training,',
67  for i in range(10):
        print '%d/10' % (i,),
69      with open(destpath + fil + str(i) + '.pickle', 'r') as f:
            x = pickle.load(f)
71      dtr.partial_fit(x, y[: x.shape[0] ], classes = classes)
        print x.shape[0]
73      y = y[ x.shape[0] :]

75  print 'validating,',
    with open(destpath + fil + '10' + '.pickle', 'r') as f:
77      x = pickle.load(f)
    print '--->', len(y), x.shape[0]
79  summary += '\n  > validation score: %f' % (
            dtr.score(x, y[:x.shape[0]])
81      ,)

83  print 're-validating,',
    a = b = 0
85  for kkk in xrange(x.shape[0]):
        if thresholds[whichvalue] <= dtr.predict (x[kkk]) - y[kkk
   ]:
87          a+=1
        else:
89          b+=1

91  summary += '\n  > %d wrong out of %d in validation' % (a,b+a)

93  end_fh = time.time()
    summary += '\n  > elapsed time: %.3f s' % (end_fh - start_fh,)
95
    print 'done!'
97
    return summary
99



101
print classificationorregression(False, 'rat', fil)
103 print classificationorregression(False, 'vot', fil)
print classificationorregression(True, 'rat', fil)
105 print classificationorregression(True, 'vot', fil)
```