Amazon Products and Reviews Analysis

Final Project – Midterm Handin

Technical University of Denmark
Department of Applied Mathematics and Computer Science
02807 - Computational Tools For Big Data
November 19, 2016

1 Introduction

This report is meant to document the progress made in the first two weeks of the Final Project of the course Computational Tools for Big Data of our group.

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. We will describe in details the final environment we managed to set up.

Finally, an overview of the current situation will be outlined. It will give a summary of the different technologies we intend to use and it will hint at the different modules the final prototype may include.

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", "B00D2K1M3O", "0000031909",
     [...]
    "B00D103F8U", "B007R2RM8W"],
    "also_viewed": ["B002BZX8Z6", "B00JHONN1S", "B008F0SU0Y", "B00D23MC6W",
     [...]
    "B007ZN5Y56", "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 our time choosing the proper libraries and the tools we will work with. In order to show our effort 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.
You can find below the code that was meant to create the `.db` file:

```
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
```

The sole purpose of the output is to test the execution time.

3.3 Blaze, Dask and Castra

Solution Finally, we found an environment that was perfectly fitting our needs. 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.
However, we first had to struggle with the `metadata.json` file, which, differently than the dataset of the reviews, had some formatting issues that took us time to fix. Below you can find an example of line that the normal `json.loads()` couldn't properly parse:

```
{
  'asin': '0001048791',
  'salesRank': {'Books': 6334800},
  'imUrl': 'http://ecx.images-amazon.com/images/I/51MKP0T4DBL.jpg',
  'categories': [['Books']],
  'title': 'The Crucible: Performed by Stuart Pankin, Jerome Dempsey
      &amp;amp; Cast'
}
```

As you can see, the structure of the JSON dictionary is not correct, since the keys and the values are not surrounded by double quotes (as they should be), instead there are single quotes. This was causing problems also with those texts containing apostrophes. Fortunately, we figured out how to solve this, simply replacing the standard `json.loads()` with the `ast.literal_eval()`. This meant a slower encoding, however it worked fine.
This cell shows the fix for the `metadata.json` file:

```
import gzip
import ast

metadata_gz = "C:\\dtu\\ctbd\\amazon_dataset\\metadata.json.gz"
fivecore = 'C:\\dtu\\ctbd\\amazon_dataset\\kcore_5.json.gz'

with gzip.open(metadata_gz, 'rb') as f:
    for i,l in enumerate(f):
        try:
            k = ast.literal_eval(l)
        except ValueError, e:
            print '\nERR\n',e,'\n' , l
            break
        if i % 1e5 == 0 and i>0:
            print i,
```

```
        #Just not to process the whole file, we stop the
    computation after 1.000.000 lines
        if i % 1e6 == 0 and i>0:
            break

print i
print "Done"
```

```
100000 200000 300000 400000 500000 600000 700000 800000
900000 1000000 1000000
Done
```

Before showing the code we used to setup our environment, we will give a basic explanation of the structures and the libraries we used, explained in greater detail in the above linked article.
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.

Now, we show step by step how we made our environment to work:
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))
```

```
Starting the creation of the Castra files...
Processing compressed metadata...
Done! Metadata processed in: 0:28:07.318000
Processing compressed 5_cores...
Done! Reviews data processed in: 0:34:50.926000
```

This final output shows that the process of the creation of the castra files on our disk finished successfully. This means also that, at least for the part of the analysis we want to conduct on our dataset involving DataFrames, we don't need anything but these castra files. Thus, the starting point is now creating a **Dask DataFrame** which is *linked* to the castra files.
Below, we show some basic queries that we run, just as examples and in order to see that everything was working fine. It's worth reminding that the problem of RAM usage that we were struggling with has now been solved, and we are able to work with a dataset whose size is more than 20GB and perform queries that take just few seconds/minutes to be run.

```
# 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.head(5)
```

	asin	reviewerID	reviewerName	overall	summary	reviewText	reviewTime	unixReviewTime
0	0000013714	ACNGUPJ3A3TM9	GCM	4.0	Nice Hymnal	We use this type of hymnal at church. I was l...	12 3, 2013	1386028800
1	0000013714	A2SUAM1J3GNN3B	J. McDonald	5.0	Heavenly Highway Hymns	I bought this for my husband who plays the pia...	09 13, 2009	1252800000
2	0000013714	APOZ15IEYQRRR	maewest64	5.0	Awesome Hymn Book	This is a large size hymn book which is great ...	03 9, 2013	1362787200
3	0000013714	AYEDW3BFK53XK	Missb	5.0	Hand Clapping Toe Tapping Oldies	We use this hymn book at the mission. It has ...	01 2, 2012	1325462400
4	0000013714	A1KLCGLCXYP1U1	Paul L "Paul Lytle"	3.0	Misleading	One review advised this book was large print, ...	08 10, 2013	1376092800

```
df.count().compute()
```

```
[########################################] | 100% Completed | 28min 50.0s

asin              41135700
overall           41135700
reviewText        41135700
reviewTime        41135700
reviewerID        41135700
reviewerName      40203149
summary           41135700
unixReviewTime    41135696
dtype: int64
```

```
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
```

Last but not least, Dask can be used also to handle non-structured data, which could be useful for building bag of words, for example.

```
import ujson
import dask.dataframe as dd
import dask.bag as db
from dask.diagnostics import ProgressBar

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

js = db.read_text("C:\\dtu\\ctbd\\amazon_dataset\\Books_5.json").
    map(ujson.loads)
js.count().compute()
```

```
[########################################] | 100% Completed |  2min 46.9s
[########################################] | 100% Completed |  2min 47.4s

8898041
```

3.4 Spark installation

One of the aims we wanted to achieve in the first part of this project was to install and use Apache Spark both on Windows and Mac, and have it running in the Jupiter Notebook interface. In this section we will list the steps followed to achieve this, for both operative systems.

3.4.1 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.

3.4.2 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'.

3.5 Amazon EC2 Instance

We 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 Recap

Up to now, what we've done with our dataset does not involve anything about the analysis that we want to conduct, yet. For this midway hand-in, we tried to put more stress on the effort we put into choosing the components for our ideal environment. We list here a recap of our current situation, in order to explain what we're planning to do:

- The tools we'll be working with are **Pandas**, **Blaze** (**Dask** and **Castra**), **Apache Spark**;
- We'll use **Pandas** because it provides a convenient interface to work with and it's widely used in the data science field; plus, its fixed-schema structure well fits our dataset and it will be easy to run queries with this module;
- **Dask** and **Castra** are the tools from the **Blaze** ecosystem that we used to build a resident copy of our dataset and to work with *larger-than-memory* datasets. Indeed, this allows us to avoid affecting the memory and work with our hard disks;
- We intend to use **Spark** in conjunction with **Neo4J** to build a graph of the products and see how they are related to each other;
- We're looking forward to analyse the dataset in depth. We will run queries to extract the most relevant statistics about the reviews and have a look at the most interesting facts.