DATA MINING PROJECT

Pierre Arbelet, Arthur Dupont, Anthony Hayot

December 11, 2014

1 Introduction

The product offer of supermarkets is large and complex and it can be very challenging for consumers to navigate into it. This complexity is an obstacle for conducting basic actions such as:

- comparing price between supermarkets
- find similar products in order to reach a specific budget
- switch products to match a specific behaviour (organic, hallal, gluten-free...)

The objective of this project is to provide consumers with a navigation tool that will transform his shopping experience and make it efficient and powerful.

The following steps have been followed:

- Step 1: Crawl product inventories from online supermarkets
- Step 2: Data cleaning (avoid redundancy, completion, formating)
- Step 3: Build Distance Functions between products to allow efficient substitutions
- Step 4: Find the best candidate for a given shopping cart

2 Crawling inventories

For this project, we have focused our efforts on 3 online supermakets: Auchan, Monoprix and Simply. But the approach that we will detail below can be easily applied to any supermarket website, as they all share a similar architecture:

- a welcome page with links to different categories (beverage, meat, fruit...)
- inside each category, you will have different level a sub-categories (alcohol, juice, soda...)
- after a couple of categorization levels (usually between 1 and 5), you reach the product offer
- by clicking on each product, you reach a "product page" where detailed information (description, energetic values, price...) is displayed

Our crawlers are built in two steps:

- navigate through the categories to collect "product page" urls
- extract relevant information for each "product page"

The Python code for the first step is described below:

```
In []: import requests
       from bs4 import BeautifulSoup
       import pandas as pd
       import re

       #different levels of categorization
       rayonsA=[]
       rayonsB=[]
       rayonsC=[]
       rayonsD=[]
       rayonsE=[]
       productLink=[]

       #start at welcome page and get first level categories
       url='http://www.livraison.simplymarket.fr/'
       r = requests.get(url)
       soup = BeautifulSoup(r.text,'html.parser')
       balises_a=soup.find_all("a",class_='linkMenu')

       for balise_a in balises_a:
               rayonsA.append('http://www.livraison.simplymarket.fr/'+balise_a.get('href'))

       #navigate in each Level-1 category
       for rayonA in rayonsA:
               r = requests.get(rayonA)
               soup = BeautifulSoup(r.text,'html.parser')
               balises_a=soup.find_all("a",class_='lienProduit')
               balises_tr=soup.find_all("tr",class_='trLibelle')

       #find product pages, if any, and add it to the list
               for balise_a in balises_a:
                       productLink.append('http://www.livraison.simplymarket.fr/'+balise_a.get('href'))

       #if no product pages can be found, find Level-2 categories
               if balises_a==[]:
                       for balise_tr in balises_tr:
                               rayonsB.append('http://www.livraison.simplymarket.fr/'+balise_tr.find("a
```

This algorithm is repeated over each level of categorization until no new category is discovered. At the end, the variable productLink is a list that contains all the "product page" urls and we can move to the second step.

The second step consist in scrapping the relevant part of the html code of each "product page" that contains the following fields:

- product_name
- brand
- quantity (eg 6 for 6x33cL)
- weight/volume (eg 33 for 6x33cL)
- weight/volume_total (eg 200 for 6x33cL)
- unit (eg cL)
- description

- ingredients
- preservation
- nutritional details
- origin
- price
- price per unit

3 Data Cleaning

Once the html code containing these data has been scrapped, it needs to be parsed and cleaned in order to fill the table with the correct values. Ideally, we would have stored the code in a text database, such as MongoDB, and then build a parser to transfer information from MongoDB to a standard relationnal database. For this project, we have scrapped and cleaned the data at the same time and built a csv file gathering the data.

First, the parsing/cleaning stage consists in understanding the structure of the html code. For example:

In []:

```
<div class="redactionnel">
    <div class="texteProduit">
        <h1>FOIE GRAS DE CANARD ENTIER SUD OUEST GASTRONOMIQUE MONTFORT 300G - Montfort</h1>
    </div>
            <div class="texteProduit">
        <label>Prix quantité</label><br>
        99,67 eur/Kg
    </div>
            <div class="texteProduit">
        <label>Descriptif</label><br>
        Foie gras de canard entier du Sud Ouest.
    </div>
            <div class="texteProduit">
        <label>Avantages</label><br>
        <div>La recette du Gastronomique est inspirée des préparations des plus grands Chefs
    </div>
            <div class="texteProduit">
        <label>Ingrédients</label><br>
        Foie gras de canard, sel, Armagnac, Porto, poivre, sucre, antioxydant : ascorbate de
    </div>
            <div class="texteProduit">
        <label>Conservation</label><br>
        A conserver au réfrigérateur entre 0°C et +4°C. A consommer rapidement après ouvertu
    </div>
            <div class="texteProduit">
        <label>Renseignements pratiques</label><br>
        <p>Sortir le produit du réfrigérateur quelques minutes avant de le déguster.</p><p>E
    </div>
            <div class="texteProduit">
    <table width="100%" cellspacing="0" cellpadding="0">
        <tbody><tr>
```

For this example, we can observe that each piece of information is preceded by its label (eg 'Conservation'). The parser will try to find these labels and if they exist, it will take the information that comes right after and store it in the relevant variable.

```
In []: if 'Conservation' in reduced_prod_info:
                                index_conservation=reduced_prod_info.index('Conservation')
                                conservation=reduced_prod_info[index_conservation+1]
```

Some data are stored in a different format than expected in our database. To solve this, we have used regular expressions in order to recognize patterns and extract exactly what we were looking for.
Example:
FOIE GRAS DE CANARD ENTIER SUD OUEST GASTRONOMIQUE MONTFORT 300G - Montfort

We want to have:

- name = FOIE GRAS DE CANARD ENTIER SUD OUEST GASTRONOMIQUE MONTFORT
- quantity = 1
- weight/volume = 300
- unit=G

In Python, this will translate:

```
In []: title=re.match(r'(.*)\s(\d+\.?\d+?)(KG|G|ML|CL|L)',nom)
                        if title!=None:
                                nom=title.group(1)
                                quantite=1
                                poids_volume_total=title.group(2)
                                unite=title.group(3)
```

By iterating this operation on most observed formats, we managed to build a clean dataset available for analysis.

4 Distance between products

Now that we have a clean dataset, we need to build a distance function between products in order to find, for each product, the most similar items in our inventory. To build this distance function, we started with distance between identical fields. This could be very simple for numeric fields such as weight/volume but is more complex for Strings. We will focus on String distance for this project.

4.1 Jaccard Distance

The naive of performing distance between sets of Strings is by using Jaccard Distance defined by:

Jaccard Distance = $\frac{card(A \cup B) - card(A \cap B)}{card(A \cup B)}$

where $card(A \cup B)$ represents the number of distinct words in String A and B
and $card(A \cap B)$ represents the number of common words in String A and B.

In Python, it translates as below:
code Jaccard
This method gives us a first hint of the distance between our two strings but has two weaknesses:

- irrelevant words (article, punctuations...) should not be counted
- variations on the same word (singular/plural, conjugation) should be counted as common words

4.2 Stopwords

Those irrelevant words are known as "Stopwords" and should be removed before performing the Jaccard Distance. To do so, we use a list of French Stopwords from https://code.google.com/p/stop-words/. We search for every words of A in the list. If it is in the list, the word is removed.
For example, the sentence:
taboule oriental - salade de semoule de ble dur, aux legumes, aux raisins secs et a la menthe
becomes:
taboule oriental salade semoule ble dur legumes raisins secs menthe
In Python, we have:
code Stopwords

4.3 Lemmatization

The idea here is to be able to count as common words with the same root (eg chocolat, chocolats et chocolatée). To do so, we extract the root using FrenchStemmer from nltk.stem.snowball library. We, then, replace the word by its root.
Our example becomes:
taboule orient salade semoule ble dur legume raisin sec menthe

By removing Stopwords and lemmatizing, we get a significant improvement that we will present in the next sections.

5 Results

To analyse the performance of our algorithm, we have manually define a "Gold Standard". We chose 6 initial products from the Simply inventory but in different product category.
For each product, we picked:

- the 2 most similar products at Simply
- the 3 most similar products at Monoprix
- the 3 most similar products at Auchan

By running the algorithm on the initial products, we can see if it performs as expected.