FabienDaniel

Customer Segmentation

123 voters

last run 5 months ago · IPython Notebook HTML · 23,587 views
using data from E-Commerce Data · Public

Notebook Code Data (1) Comments (22) Log Versions (63) Forks (483) Fork Notebook

Tags tutorial classification clustering forecasting marketing

Notebook

Customer segmentation

F. Daniel (September 2017)

This notebook aims at analyzing the content of an E-commerce database that lists purchases made by $\sim$4000 customers over a period of one year (from 2010/12/01 to 2011/12/09). Based on this analysis, I develop a model that allows to anticipate the purchases that will be made by a new customer, during the following year and this, from its first purchase.

Acknowledgement: many thanks to J. Abécassis (https://www.kaggle.com/judithabk6) for the advices and help provided during the writing of this notebook

1. Data preparation

As a first step, I load all the modules that will be used in this notebook:

In [1]:

Hide

```
import pandas as pd
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
import seaborn as sns
import datetime, nltk, warnings
import matplotlib.cm as cm
import itertools
from pathlib import Path
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_samples, silhouette_score
from sklearn import preprocessing, model_selection, metrics, feature_selection
from sklearn.model_selection import GridSearchCV, learning_curve
from sklearn.svm import SVC
from sklearn.metrics import confusion_matrix
from sklearn import neighbors, linear_model, svm, tree, ensemble
from wordcloud import WordCloud, STOPWORDS
from sklearn.ensemble import AdaBoostClassifier
from sklearn.decomposition import PCA
from IPython.display import display, HTML
import plotly.graph_objs as go
from plotly.offline import init_notebook_mode,iplot
init_notebook_mode(connected=True)
warnings.filterwarnings("ignore")
plt.rcParams["patch.force_edgecolor"] = True
plt.style.use('fivethirtyeight')
mpl.rc('patch', edgecolor = 'dimgray', linewidth=1)
%matplotlib inline
```

Then, I load the data. Once done, I also give some basic informations on the content of the dataframe: the type of the various variables, the number of null values and their percentage with respect to the total number of entries:

In [2]: Hide

```
#__________________
# read the datafile
df_initial = pd.read_csv('../input/data.csv',encoding="ISO-8859-1",
                         dtype={'CustomerID': str,'InvoiceID': str})
print('Dataframe dimensions:', df_initial.shape)
#______
df_initial['InvoiceDate'] = pd.to_datetime(df_initial['InvoiceDate'])
#____________________________________________________________
# gives some infos on columns types and numer of null values
tab_info=pd.DataFrame(df_initial.dtypes).T.rename(index={0:'column type'})
tab_info=tab_info.append(pd.DataFrame(df_initial.isnull().sum()).T.rename(index={
0:'null values (nb)'}))
tab_info=tab_info.append(pd.DataFrame(df_initial.isnull().sum()/df_initial.shape[
0]*100).T.
                         rename(index={0:'null values (%)'}))
display(tab_info)
#__________________
# show first lines
display(df_initial[:5])
```

```
Dataframe dimensions: (541909, 8)
```

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country
column type	object	object	object	int64	datetime64[ns]	float64	object	object
null values (nb)	0	0	1454	0	0	0	135080	0
null values (%)	0	0	0.268311	0	0	0	24.9267	0

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country
0	536365	85123A	WHITE HANGING HEART T-LIGHT HOLDER	6	2010-12-01 08:26:00	2.55	17850	United Kingdom
1	536365	71053	WHITE METAL LANTERN	6	2010-12-01 08:26:00	3.39	17850	United Kingdom
2	536365	84406B	CREAM CUPID HEARTS COAT HANGER	8	2010-12-01 08:26:00	2.75	17850	United Kingdom
3	536365	84029G	KNITTED UNION FLAG HOT WATER BOTTLE	6	2010-12-01 08:26:00	3.39	17850	United Kingdom
4	536365	84029E	RED WOOLLY HOTTIE WHITE HEART.	6	2010-12-01 08:26:00	3.39	17850	United Kingdom

While looking at the number of null values in the dataframe, it is interesting to note that $\sim$25% of the entries are not assigned to a particular customer. With the data available, it is impossible to impute values for the user and these entries are thus useless for the current exercise. So I delete them from the dataframe:

In [3]: Hide

```
df_initial.dropna(axis = 0, subset = ['CustomerID'], inplace = True)
print('Dataframe dimensions:', df_initial.shape)
#____________________________________________________________
# gives some infos on columns types and numer of null values
tab_info=pd.DataFrame(df_initial.dtypes).T.rename(index={0:'column type'})
tab_info=tab_info.append(pd.DataFrame(df_initial.isnull().sum()).T.rename(index={
0:'null values (nb)'}))
tab_info=tab_info.append(pd.DataFrame(df_initial.isnull().sum()/df_initial.shape[
0]*100).T.
                         rename(index={0:'null values (%)'}))
display(tab_info)
```

```
Dataframe dimensions: (406829, 8)
```

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country
column type	object	object	object	int64	datetime64[ns]	float64	object	object
null values (nb)	0	0	0	0	0	0	0	0
null values (%)	0	0	0	0	0	0	0	0

OK, therefore, by removing these entries we end up with a dataframe filled at 100% for all variables! Finally, I check for duplicate entries and delete them:

In [4]:

```
print('Entrées dupliquées: {}'.format(df_initial.duplicated().sum()))
df_initial.drop_duplicates(inplace = True)
```

```
Entrées dupliquées: 5225
```

2. Exploring the content of variables

This dataframe contains 8 variables that correspond to:

InvoiceNo: Invoice number. Nominal, a 6-digit integral number uniquely assigned to each transaction. If this code starts with letter 'c', it indicates a cancellation.
StockCode: Product (item) code. Nominal, a 5-digit integral number uniquely assigned to each distinct product.
Description: Product (item) name. Nominal.
Quantity: The quantities of each product (item) per transaction. Numeric.
InvoiceDate: Invice Date and time. Numeric, the day and time when each transaction was generated.
UnitPrice: Unit price. Numeric, Product price per unit in sterling.
CustomerID: Customer number. Nominal, a 5-digit integral number uniquely assigned to each customer.
Country: Country name. Nominal, the name of the country where each customer resides.

2.1 Countries

Here, I quickly look at the countries from which orders were made:

In [5]:

Hide

```
temp = df_initial[['CustomerID', 'InvoiceNo', 'Country']].groupby(['CustomerID',
'InvoiceNo', 'Country']).count()
temp = temp.reset_index(drop = False)
countries = temp['Country'].value_counts()
print('Nb. de pays dans le dataframe: {}'.format(len(countries)))
```

```
Nb. de pays dans le dataframe: 37
```

and show the result on a chloropleth map:

In [6]:

Hide

```
data = dict(type='choropleth',
locations = countries.index,
locationmode = 'country names', z = countries,
text = countries.index, colorbar = {'title':'Order nb.'},
colorscale=[[0, 'rgb(224,255,255)'],
            [0.01, 'rgb(166,206,227)'], [0.02, 'rgb(31,120,180)'],
            [0.03, 'rgb(178,223,138)'], [0.05, 'rgb(51,160,44)'],
            [0.10, 'rgb(251,154,153)'], [0.20, 'rgb(255,255,0)'],
            [1, 'rgb(227,26,28)']],
reversescale = False)
#_______________________
layout = dict(title='Number of orders per country',
geo = dict(showframe = True, projection={'type':'Mercator'}))
#______________
choromap = go.Figure(data = [data], layout = layout)
iplot(choromap, validate=False)
```

Number of orders per country

Expo

We see that the dataset is largely dominated by orders made from the UK.

2.2 Customers and products

The dataframe contains $\sim$400,000 entries. What are the number of users and products in these entries ?

In [7]:

Hide

```
pd.DataFrame([{'products': len(df_initial['StockCode'].value_counts()),
               'transactions': len(df_initial['InvoiceNo'].value_counts()),
               'customers': len(df_initial['CustomerID'].value_counts()),
              }], columns = ['products', 'transactions', 'customers'], index = [
'quantity'])
```

Out[7]:

	products	transactions	customers
quantity	3684	22190	4372

It can be seen that the data concern 4372 users and that they bought 3684 different products. The total number of transactions carried out is of the order of $\sim$22'000.

Now I will determine the number of products purchased in every transaction:

In [8]:

Hide

```
temp = df_initial.groupby(by=['CustomerID', 'InvoiceNo'], as_index=False)['Invoic
eDate'].count()
nb_products_per_basket = temp.rename(columns = {'InvoiceDate':'Number of product
s'})
nb_products_per_basket[:10].sort_values('CustomerID')
```

Out[8]:

	CustomerID	InvoiceNo	Number of products
0	12346	541431	1
1	12346	C541433	1
2	12347	537626	31
3	12347	542237	29
4	12347	549222	24
5	12347	556201	18
6	12347	562032	22
7	12347	573511	47
8	12347	581180	11
9	12348	539318	17

The first lines of this list shows several things worthy of interest:

- the existence of entries with the prefix C for the **InvoiceNo** variable: this indicates transactions that have been canceled
- the existence of users who only came once and only purchased one product (e.g. nº12346)
- the existence of frequent users that buy a large number of items at each order

2.2.1 Cancelling orders

First of all, I count the number of transactions corresponding to canceled orders:

In [9]:

Hide

```
nb_products_per_basket['order_canceled'] = nb_products_per_basket['InvoiceNo'].ap
ply(lambda x:int('C' in x))
display(nb_products_per_basket[:5])
#________________________________________________________________________________
_____________
n1 = nb_products_per_basket['order_canceled'].sum()
n2 = nb_products_per_basket.shape[0]
print('Number of orders canceled: {}/{} ({:.2f}%) '.format(n1, n2, n1/n2*100))
```

	CustomerID	InvoiceNo	Number of products	order_canceled
0	12346	541431	1	0
1	12346	C541433	1	1
2	12347	537626	31	0
3	12347	542237	29	0
4	12347	549222	24	0

```
Number of orders canceled: 3654/22190 (16.47%)
```

We note that the number of cancellations is quite large ($\sim$16% of the total number of transactions). Now, let's look at the first lines of the dataframe:

In [10]: Hide

```
display(df_initial.sort_values('CustomerID')[:5])
```

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country
61619	541431	23166	MEDIUM CERAMIC TOP STORAGE JAR	74215	2011-01-18 10:01:00	1.04	12346	United Kingdom
61624	C541433	23166	MEDIUM CERAMIC TOP STORAGE JAR	-74215	2011-01-18 10:17:00	1.04	12346	United Kingdom
286623	562032	22375	AIRLINE BAG VINTAGE JET SET BROWN	4	2011-08-02 08:48:00	4.25	12347	Iceland
72260	542237	84991	60 TEATIME FAIRY CAKE CASES	24	2011-01-26 14:30:00	0.55	12347	Iceland
14943	537626	22772	PINK DRAWER KNOB ACRYLIC EDWARDIAN	12	2010-12-07 14:57:00	1.25	12347	Iceland

On these few lines, we see that when an order is canceled, we have another transactions in the dataframe, mostly identical except for the **Quantity** and **InvoiceDate** variables. I decide to check if this is true for all the entries. To do this, I decide to locate the entries that indicate a negative quantity and check if there is *systematically* an order indicating the same quantity (but positive), with the same description (**CustomerID**, **Description** and **UnitPrice**):

In [11]:

```
df_check = df_initial[df_initial['Quantity'] < 0][['CustomerID','Quantity',
                                                   'StockCode','Description','UnitPri
ce']]
for index, col in  df_check.iterrows():
```

```
    if df_initial[(df_initial['CustomerID'] == col[0]) & (df_initial['Quantity'] == -
col[1])
                & (df_initial['Description'] == col[2])].shape[0] == 0:
        print(df_check.loc[index])
        print(15*'-'+'>'+' HYPOTHESIS NOT FULFILLED')
        break
```

```
CustomerID        14527
Quantity             -1
StockCode             D
Description    Discount
UnitPrice          27.5
Name: 141, dtype: object
---------------> HYPOTHESIS NOT FULFILLED
```

We see that the initial hypothesis is not fulfilled because of the existence of a '*Discount*' entry. I check again the hypothesis but this time discarding the '*Discount*' entries:

In [12]:

```
df_check = df_initial[(df_initial['Quantity'] < 0) & (df_initial['Description'] !
= 'Discount')][
                                 ['CustomerID','Quantity','StockCode',
                                  'Description','UnitPrice']]

for index, col in  df_check.iterrows():
    if df_initial[(df_initial['CustomerID'] == col[0]) & (df_initial['Quantity']
== -col[1])
                & (df_initial['Description'] == col[2])].shape[0] == 0:
        print(index, df_check.loc[index])
        print(15*'-'+'>'+' HYPOTHESIS NOT FULFILLED')
        break
```

```
154 CustomerID                               15311
Quantity                                    -1
StockCode                               35004C
Description    SET OF 3 COLOURED  FLYING DUCKS
UnitPrice                                 4.65
Name: 154, dtype: object
---------------> HYPOTHESIS NOT FULFILLED
```

Once more, we find that the initial hypothesis is not verified. Hence, cancellations do not necessarily correspond to orders that would have been made beforehand.

At this point, I decide to create a new variable in the dataframe that indicate if part of the command has been canceled. For the cancellations without counterparts, a few of them are probably due to the fact that the buy orders were performed before December 2010 (the point of entry of the database). Below, I make a census of the cancel orders and check for the existence of counterparts:

In [13]:

Hide

```
df_cleaned = df_initial.copy(deep = True)
df_cleaned['QuantityCanceled'] = 0

entry_to_remove = [] ; doubtfull_entry = []

for index, col in  df_initial.iterrows():
    if (col['Quantity'] > 0) or col['Description'] == 'Discount': continue

    df_test = df_initial[(df_initial['CustomerID'] == col['CustomerID']) &
                         (df_initial['StockCode']  == col['StockCode']) &
                         (df_initial['InvoiceDate'] < col['InvoiceDate']) &
                         (df_initial['Quantity']   > 0)].copy()
    #_________________________________
    # Cancelation WITHOUT counterpart
    if (df_test.shape[0] == 0):
        doubtfull_entry.append(index)
    #________________________________
    # Cancelation WITH a counterpart
    elif (df_test.shape[0] == 1):
        index_order = df_test.index[0]
        df_cleaned.loc[index_order, 'QuantityCanceled'] = -col['Quantity']
        entry_to_remove.append(index)
    #______________________________________________________________
    # Various counterparts exist in orders: we delete the last one
    elif (df_test.shape[0] > 1):
        df_test.sort_index(axis=0 ,ascending=False, inplace = True)
        for ind, val in df_test.iterrows():
            if val['Quantity'] < -col['Quantity']: continue
            df_cleaned.loc[ind, 'QuantityCanceled'] = -col['Quantity']
            entry_to_remove.append(index)
            break
```

In the above function, I checked the two cases:

1. a cancel order exists without counterpart
2. there's at least one counterpart with the exact same quantity

The index of the corresponding cancel order are respectively kept in the `doubtfull_entry` and `entry_to_remove` lists whose sizes are:

In [14]:

Hide

```
print("entry_to_remove: {}".format(len(entry_to_remove)))
print("doubtfull_entry: {}".format(len(doubtfull_entry)))
```

```
entry_to_remove: 7521
```

```
doubtfull_entry: 1226
```

Among these entries, the lines listed in the *doubtfull_entry* list correspond to the entries indicating a cancellation but for which there is no command beforehand. In practice, I decide to delete all of these entries, which count respectively for $\sim$1.4% and 0.2% of the dataframe entries.

Now I check the number of entries that correspond to cancellations and that have not been deleted with the previous filter:

In [15]:

```
df_cleaned.drop(entry_to_remove, axis = 0, inplace = True)
df_cleaned.drop(doubtfull_entry, axis = 0, inplace = True)
remaining_entries = df_cleaned[(df_cleaned['Quantity'] < 0) & (df_cleaned['StockC
ode'] != 'D')]
print("nb of entries to delete: {}".format(remaining_entries.shape[0]))
remaining_entries[:5]
```

Hide

```
nb of entries to delete: 48
```

Out[15]:

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country	QuantityC
77598	C542742	84535B	FAIRY CAKES NOTEBOOK A6 SIZE	-94	2011-01-31 16:26:00	0.65	15358	United Kingdom	0
90444	C544038	22784	LANTERN CREAM GAZEBO	-4	2011-02-15 11:32:00	4.95	14659	United Kingdom	0
111968	C545852	22464	HANGING METAL HEART LANTERN	-5	2011-03-07 13:49:00	1.65	14048	United Kingdom	0
116064	C546191	47566B	TEA TIME PARTY BUNTING	-35	2011-03-10 10:57:00	0.70	16422	United Kingdom	0
132642	C547675	22263	FELT EGG COSY LADYBIRD	-49	2011-03-24 14:07:00	0.66	17754	United Kingdom	0

If one looks, for example, at the purchases of the consumer of one of the above entries and corresponding to the same product as that of the cancellation, one observes:

In [16]:

```
df_cleaned[(df_cleaned['CustomerID'] == 14048) & (df_cleaned['StockCode'] == '22464'
)]
```

```
)]
```

Out[16]:

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country	QuantityCanceled

We see that the quantity canceled is greater than the sum of the previous purchases.

2.2.2 StockCode

Above, it has been seen that some values of the **StockCode** variable indicate a particular transaction (i.e. D for *Discount*). I check the contents of this variable by looking for the set of codes that would contain only letters:

In [17]:

```
list_special_codes = df_cleaned[df_cleaned['StockCode'].str.contains('^[a-zA-Z]+'
, regex=True)]['StockCode'].unique()
list_special_codes
```

Out[17]:

```
array(['POST', 'D', 'C2', 'M', 'BANK CHARGES', 'PADS', 'DOT'], dtype=object)
```

In [18]:

```
for code in list_special_codes:
    print("{:<15} -> {:<30}".format(code, df_cleaned[df_cleaned['StockCode'] == c
ode]['Description'].unique()[0]))
```

```
POST            -> POSTAGE
D               -> Discount
C2              -> CARRIAGE
M               -> Manual
BANK CHARGES    -> Bank Charges
PADS            -> PADS TO MATCH ALL CUSHIONS
DOT             -> DOTCOM POSTAGE
```

We see that there are several types of peculiar transactions, connected e.g. to port charges or bank charges.

2.2.3 Basket Price

I create a new variable that indicates the total price of every purchase:

In [19]:

```
df_cleaned['TotalPrice'] = df_cleaned['UnitPrice'] * (df_cleaned['Quantity'] - df
_cleaned['QuantityCanceled'])
```

```
df_cleaned.sort_values('CustomerID')[:5]
```

Out[19]:

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country	QuantityCa
61619	541431	23166	MEDIUM CERAMIC TOP STORAGE JAR	74215	2011-01-18 10:01:00	1.04	12346	United Kingdom	74215
148288	549222	22375	AIRLINE BAG VINTAGE JET SET BROWN	4	2011-04-07 10:43:00	4.25	12347	Iceland	0
428971	573511	22698	PINK REGENCY TEACUP AND SAUCER	12	2011-10-31 12:25:00	2.95	12347	Iceland	0
428970	573511	47559B	TEA TIME OVEN GLOVE	10	2011-10-31 12:25:00	1.25	12347	Iceland	0
428969	573511	47567B	TEA TIME KITCHEN APRON	6	2011-10-31 12:25:00	5.95	12347	Iceland	0

Each entry of the dataframe indicates prizes for a single kind of product. Hence, orders are split on several lines. I collect all the purchases made during a single order to recover the total order prize:

In [20]: Hide

```
#____________________________________________
# somme des achats / utilisateur & commande
temp = df_cleaned.groupby(by=['CustomerID', 'InvoiceNo'], as_index=False)['TotalP
rice'].sum()
basket_price = temp.rename(columns = {'TotalPrice':'Basket Price'})
#_____________________
# date de la commande
df_cleaned['InvoiceDate_int'] = df_cleaned['InvoiceDate'].astype('int64')
temp = df_cleaned.groupby(by=['CustomerID', 'InvoiceNo'], as_index=False)['Invoic
eDate_int'].mean()
df_cleaned.drop('InvoiceDate_int', axis = 1, inplace = True)
basket_price.loc[:, 'InvoiceDate'] = pd.to_datetime(temp['InvoiceDate_int'])
#______________________________________
# selection des entrées significatives:
basket_price = basket_price[basket_price['Basket Price'] > 0]
basket_price.sort_values('CustomerID')[:6]
```

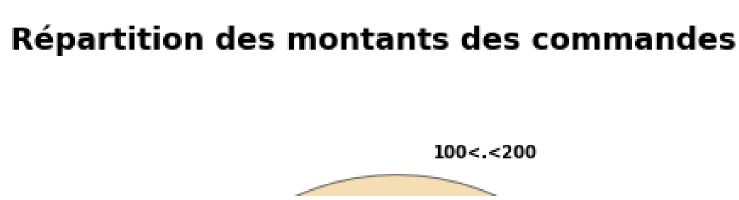
Out[20]:

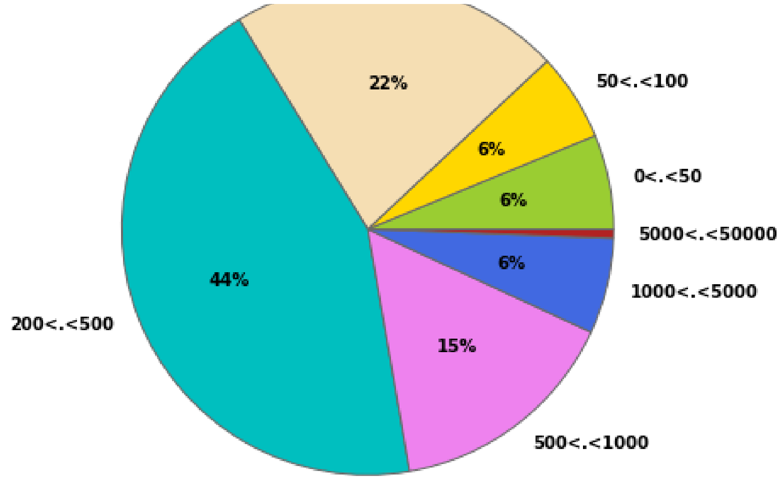
	CustomerID	InvoiceNo	Basket Price	InvoiceDate
1	12347	537626	711.79	2010-12-07 14:57:00.000001024
2	12347	542237	475.39	2011-01-26 14:29:59.999999744
3	12347	549222	636.25	2011-04-07 10:42:59.999999232
4	12347	556201	382.52	2011-06-09 13:01:00.000000256
5	12347	562032	584.91	2011-08-02 08:48:00.000000000
6	12347	573511	1294.32	2011-10-31 12:25:00.000001280

In order to have a global view of the type of order performed in this dataset, I determine how the purchases are divided according to total prizes:

In [21]: Hide

```
#____________________
# Décompte des achats
price_range = [0, 50, 100, 200, 500, 1000, 5000, 50000]
count_price = []
for i, price in enumerate(price_range):
    if i == 0: continue
    val = basket_price[(basket_price['Basket Price'] < price) &
                       (basket_price['Basket Price'] > price_range[i-1])]['Basket
 Price'].count()
    count_price.append(val)
#____________________________________________
# Représentation du nombre d'achats / montant
plt.rc('font', weight='bold')
f, ax = plt.subplots(figsize=(11, 6))
colors = ['yellowgreen', 'gold', 'wheat', 'c', 'violet', 'royalblue','firebrick']
labels = [ '{}<.<{}'.format(price_range[i-1], s) for i,s in enumerate(price_range
) if i != 0]
sizes  = count_price
explode = [0.0 if sizes[i] < 100 else 0.0 for i in range(len(sizes))]
ax.pie(sizes, explode = explode, labels=labels, colors = colors,
       autopct = lambda x:'{:1.0f}%'.format(x) if x > 1 else '',
       shadow = False, startangle=0)
ax.axis('equal')
f.text(0.5, 1.01, "Répartition des montants des commandes", ha='center', fontsize
 = 18);
```





It can be seen that the vast majority of orders concern relatively large purchases given that $\sim$65% of purchases give prizes in excess of £ 200.

3. Insight on product categories

In the dataframe, products are uniquely identified through the **StockCode** variable. A shrort description of the products is given in the **Description** variable. In this section, I intend to use the content of this latter variable in order to group the products into different categories.

3.1 Products Description

As a first step, I extract from the **Description** variable the information that will prove useful. To do this, I use the following function:

In [22]: Hide

```
is_noun = lambda pos: pos[:2] == 'NN'

def keywords_inventory(dataframe, colonne = 'Description'):
    stemmer = nltk.stem.SnowballStemmer("english")
    keywords_roots  = dict()  # collect the words / root
    keywords_select = dict()  # association: root <-> keyword
    category_keys   = []
    count_keywords  = dict()
    icount = 0
    for s in dataframe[colonne]:
        if pd.isnull(s): continue
        lines = s.lower()
        tokenized = nltk.word_tokenize(lines)
        nouns = [word for (word, pos) in nltk.pos_tag(tokenized) if is_noun(pos)]


        for t in nouns:
```

```
            t = t.lower() ; racine = stemmer.stem(t)
            if racine in keywords_roots:
                keywords_roots[racine].add(t)
                count_keywords[racine] += 1
            else:
                keywords_roots[racine] = {t}
                count_keywords[racine] = 1

    for s in keywords_roots.keys():
        if len(keywords_roots[s]) > 1:
            min_length = 1000
            for k in keywords_roots[s]:
                if len(k) < min_length:
                    clef = k ; min_length = len(k)
            category_keys.append(clef)
            keywords_select[s] = clef
        else:
            category_keys.append(list(keywords_roots[s])[0])
            keywords_select[s] = list(keywords_roots[s])[0]

    print("Nb of keywords in variable '{}': {}".format(colonne,len(category_keys
)))
    return category_keys, keywords_roots, keywords_select, count_keywords
```

This function takes as input the dataframe and analyzes the content of the **Description** column by performing the following operations:

- extract the names (proper, common) appearing in the products description
- for each name, I extract the root of the word and aggregate the set of names associated with this particular root
- count the number of times each root appears in the dataframe
- when several words are listed for the same root, I consider that the keyword associated with this root is the shortest name (this systematically selects the singular when there are singular/plural variants)

The first step of the analysis is to retrieve the list of products:

In [23]: Hide

```
df_produits = pd.DataFrame(df_initial['Description'].unique()).rename(columns = {
0:'Description'})
```

Once this list is created, I use the function I previously defined in order to analyze the description of the various products:

In [24]: Hide

```
keywords, keywords_roots, keywords_select, count_keywords = keywords_inventory(df
_produits)
```

```
Nb of keywords in variable 'Description': 1483
```

The execution of this function returns three variables:

- `keywords`: the list of extracted keywords
- `keywords_roots`: a dictionary where the keys are the keywords roots and the values are the lists of words associated with those roots
- `count_keywords`: dictionary listing the number of times every word is used

At this point, I convert the `count_keywords` dictionary into a list, to sort the keywords according to their occurences:

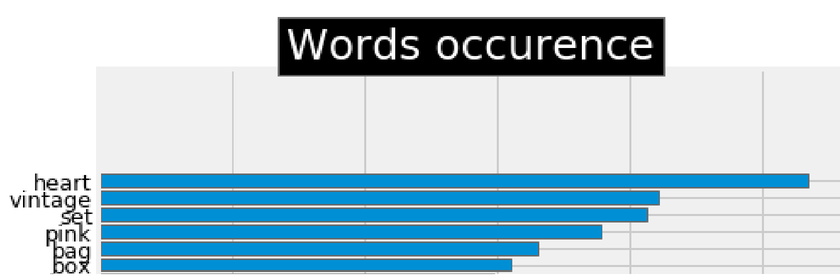
In [25]: Hide

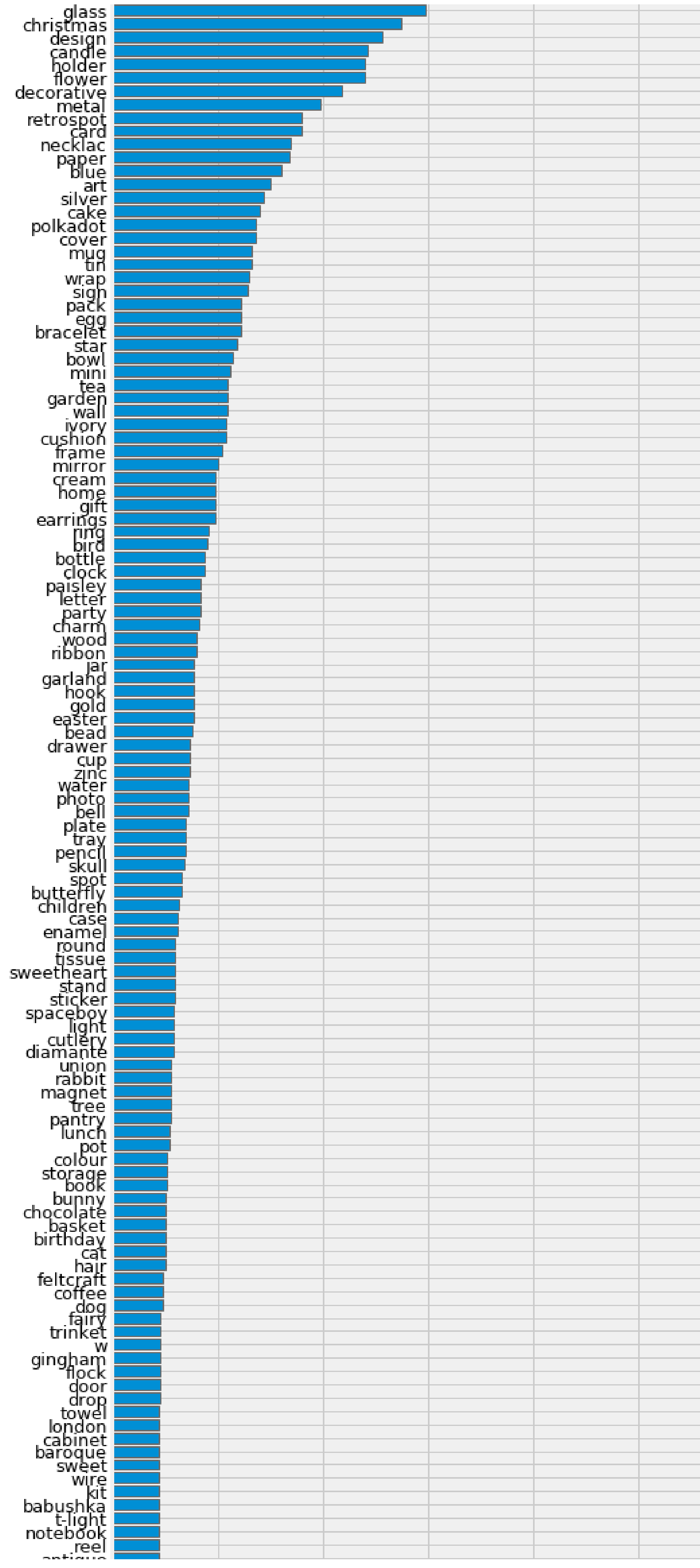
```
list_products = []
for k,v in count_keywords.items():
    list_products.append([keywords_select[k],v])
list_products.sort(key = lambda x:x[1], reverse = True)
```

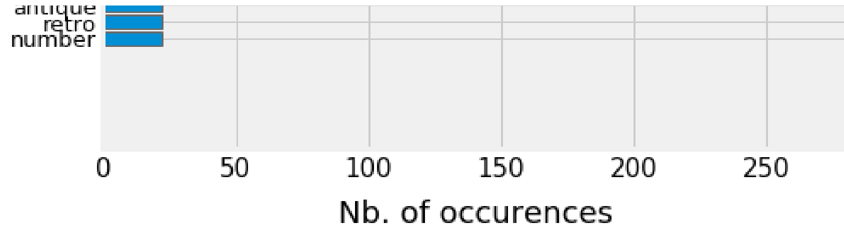
Using it, I create a representation of the most common keywords:

In [26]: Hide

```
liste = sorted(list_products, key = lambda x:x[1], reverse = True)
#_______________________________
plt.rc('font', weight='normal')
fig, ax = plt.subplots(figsize=(7, 25))
y_axis = [i[1] for i in liste[:125]]
x_axis = [k for k,i in enumerate(liste[:125])]
x_label = [i[0] for i in liste[:125]]
plt.xticks(fontsize = 15)
plt.yticks(fontsize = 13)
plt.yticks(x_axis, x_label)
plt.xlabel("Nb. of occurences", fontsize = 18, labelpad = 10)
ax.barh(x_axis, y_axis, align = 'center')
ax = plt.gca()
ax.invert_yaxis()
#_______________________________________________________________________________
______
plt.title("Words occurence",bbox={'facecolor':'k', 'pad':5}, color='w',fontsize =
 25)
plt.show()
```




glass
christmas
design
candle
holder
flower
decorative
metal
retrospot
card
necklac
paper
blue
art
silver
cake
polkadot
cover
mug
tin
wrap
sign
pack
egg
bracelet
star
bowl
mini
tea
garden
wall
ivory
cushion
frame
mirror
cream
home
gift
earrings
ring
bird
bottle
clock
paisley
letter
party
charm
wood
ribbon
jar
garland
hook
gold
easter
bead
drawer
cup
zinc
water
photo
bell
plate
tray
pencil
skull
spot
butterfly
children
case
enamel
round
tissue
sweetheart
stand
sticker
spaceboy
light
cutlery
diamante
union
rabbit
magnet
tree
pantry
lunch
pot
colour
storage
book
bunny
chocolate
basket
birthday
cat
hair
feltcraft
coffee
dog
fairy
trinket
w
gingham
flock
door
drop
towel
london
cabinet
baroque
sweet
wire
kit
babushka
t-light
notebook
reel



3.2 Defining product categories

The list that was obtained contains more than 1400 keywords and the most frequent ones appear in more than 200 products. However, while examinating the content of the list, I note that some names are useless. Others are do not carry information, like colors. Therefore, I discard these words from the analysis that follows and also, I decide to consider only the words that appear more than 13 times.

In [27]:

Hide

```
list_products = []
for k,v in count_keywords.items():
    word = keywords_select[k]
    if word in ['pink', 'blue', 'tag', 'green', 'orange']: continue
    if len(word) < 3 or v < 13: continue
    if ('+' in word) or ('/' in word): continue
    list_products.append([word, v])
#______________________________________________________
list_products.sort(key = lambda x:x[1], reverse = True)
print('mots conservés:', len(list_products))
```

```
mots conservés: 193
```

3.2.1 Data encoding

Now I will use these keywords to create groups of product. Firstly, I define the X matrix as:

	mot 1	...	mot j	...	mot N
produit 1	$a_{1,1}$				$a_{1,N}$
...			...		
produit i	...		$a_{i,j}$		...
...			...		
produit M	$a_{M,1}$				$a_{M,N}$

produit m	$a_{m,1}$				$a_{m,n}$

where the $a_ {i, j}$ coefficient is 1 if the description of the product i contains the word j, and 0 otherwise.

In [28]: Hide

```
liste_produits = df_cleaned['Description'].unique()
X = pd.DataFrame()
for key, occurence in list_products:
    X.loc[:, key] = list(map(lambda x:int(key.upper() in x), liste_produits))
```

The X matrix indicates the words contained in the description of the products using the *one-hot-encoding* principle. In practice, I have found that introducing the price range results in more balanced groups in terms of element numbers. Hence, I add 6 extra columns to this matrix, where I indicate the price range of the products:

In [29]: Hide

```
threshold = [0, 1, 2, 3, 5, 10]
label_col = []
for i in range(len(threshold)):
    if i == len(threshold)-1:
        col = '.>{}'.format(threshold[i])
    else:
        col = '{}<.<{}'.format(threshold[i],threshold[i+1])
    label_col.append(col)
    X.loc[:, col] = 0

for i, prod in enumerate(liste_produits):
    prix = df_cleaned[ df_cleaned['Description'] == prod]['UnitPrice'].mean()
    j = 0
    while prix > threshold[j]:
        j+=1
        if j == len(threshold): break
    X.loc[i, label_col[j-1]] = 1
```

and to choose the appropriate ranges, I check the number of products in the different groups:

In [30]: Hide

```
print("{:<8} {:<20} \n".format('gamme', 'nb. produits') + 20*'-')
for i in range(len(threshold)):
    if i == len(threshold)-1:
        col = '.>{}'.format(threshold[i])
    else:
        col = '{}<.<{}'.format(threshold[i],threshold[i+1])
    print("{:<10}  {:<20}".format(col, X.loc[:, col].sum()))
```

```
gamme    nb. produits
--------------------
0<.<1       964
1<.<2       1009
2<.<3       673
3<.<5       606
5<.<10      470
.>10        156
```

3.2.2 Creating clusters of products

In this section, I will group the products into different classes. In the case of matrices with binary encoding, the most suitable metric for the calculation of distances is the Hamming's metric (https://en.wikipedia.org/wiki/Distance_de_Hamming). Note that the **kmeans** method of sklearn uses a Euclidean distance that can be used, but it is not to the best choice in the case of categorical variables. However, in order to use the Hamming's metric, we need to use the kmodes (https://pypi.python.org/pypi/kmodes/) package which is not available on the current plateform. Hence, I use the **kmeans** method even if this is not the best choice.

In order to define (approximately) the number of clusters that best represents the data, I use the silhouette score:

In [31]: Hide

```
matrix = X.as_matrix()
for n_clusters in range(3,10):
    kmeans = KMeans(init='k-means++', n_clusters = n_clusters, n_init=30)
    kmeans.fit(matrix)
    clusters = kmeans.predict(matrix)
    silhouette_avg = silhouette_score(matrix, clusters)
    print("For n_clusters =", n_clusters, "The average silhouette_score is :", si
lhouette_avg)
```

```
For n_clusters = 3 The average silhouette_score is : 0.100716817581
For n_clusters = 4 The average silhouette_score is : 0.126098937473
For n_clusters = 5 The average silhouette_score is : 0.146313552489
For n_clusters = 6 The average silhouette_score is : 0.14389114354
For n_clusters = 7 The average silhouette_score is : 0.151659629857
For n_clusters = 8 The average silhouette_score is : 0.147108263245
For n_clusters = 9 The average silhouette_score is : 0.122096798066
```

In practice, the scores obtained above can be considered equivalent since, depending on the run, scores of $ 0.1 \pm 0.05 $ will be obtained for all clusters with `n_clusters` $> $ 3 (we obtain slightly lower scores for the first cluster). On the other hand, I found that beyond 5 clusters, some clusters contained very few elements. I therefore choose to separate the dataset into 5 clusters. In order to ensure a good classification at every run of the notebook, I iterate untill we obtain the best possible silhouette score, which is, in the present case, around 0.15:

Hide

In [32]:

```
n_clusters = 5
silhouette_avg = -1
while silhouette_avg < 0.145:
    kmeans = KMeans(init='k-means++', n_clusters = n_clusters, n_init=30)
    kmeans.fit(matrix)
    clusters = kmeans.predict(matrix)
    silhouette_avg = silhouette_score(matrix, clusters)

    #km = kmodes.KModes(n_clusters = n_clusters, init='Huang', n_init=2, verbose=
0)
    #clusters = km.fit_predict(matrix)
    #silhouette_avg = silhouette_score(matrix, clusters)
    print("For n_clusters =", n_clusters, "The average silhouette_score is :", si
lhouette_avg)
```

```
For n_clusters = 5 The average silhouette_score is : 0.125164429401
For n_clusters = 5 The average silhouette_score is : 0.146625760353
```

3.2.3 Characterizing the content of clusters

I check the number of elements in every class:

In [33]: Hide

```
pd.Series(clusters).value_counts()
```

Out[33]:

```
4    1009
2     964
1     762
3     673
0     470
dtype: int64
```

a / *Silhouette intra-cluster score*

In order to have an insight on the quality of the classification, we can represent the silhouette scores of each element of the different clusters. This is the purpose of the next figure which is taken from the sklearn documentation (http://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_silhouette_analysis.html):

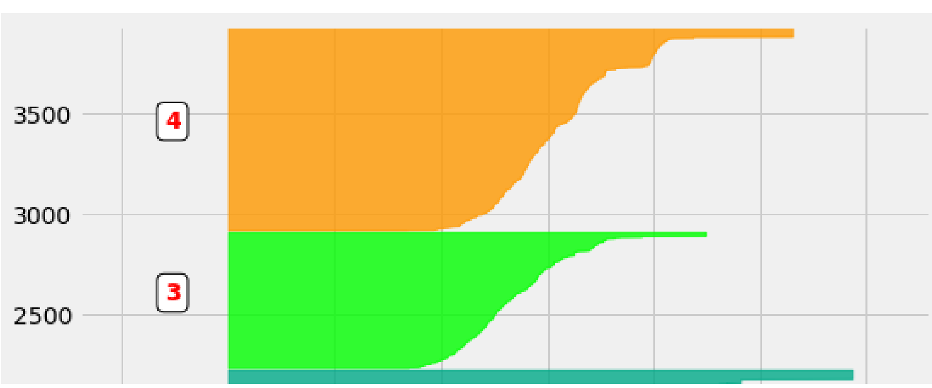
In [34]: Hide

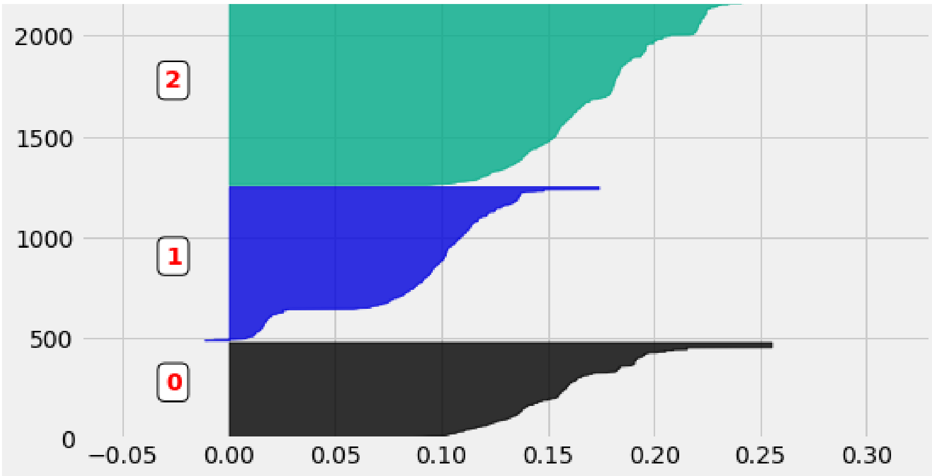
```
def graph_component_silhouette(n_clusters, lim_x, mat_size, sample_silhouette_val
ues, clusters):
    plt.rcParams["patch.force_edgecolor"] = True
```

```
    plt.style.use('fivethirtyeight')
    mpl.rc('patch', edgecolor = 'dimgray', linewidth=1)
    #____________________________
    fig, ax1 = plt.subplots(1, 1)
    fig.set_size_inches(8, 8)
    ax1.set_xlim([lim_x[0], lim_x[1]])
    ax1.set_ylim([0, mat_size + (n_clusters + 1) * 10])
    y_lower = 10
    for i in range(n_clusters):
        #_______________________________________________________________________
__________
        # Aggregate the silhouette scores for samples belonging to cluster i, and
 sort them
        ith_cluster_silhouette_values = sample_silhouette_values[clusters == i]
        ith_cluster_silhouette_values.sort()
        size_cluster_i = ith_cluster_silhouette_values.shape[0]
        y_upper = y_lower + size_cluster_i
        color = cm.spectral(float(i) / n_clusters)
        ax1.fill_betweenx(np.arange(y_lower, y_upper), 0, ith_cluster_silhouette_
values,
                          facecolor=color, edgecolor=color, alpha=0.8)
        #__________________________________________________________________
        # Label the silhouette plots with their cluster numbers at the middle
        ax1.text(-0.03, y_lower + 0.5 * size_cluster_i, str(i), color = 'red', fo
ntweight = 'bold',
                bbox=dict(facecolor='white', edgecolor='black', boxstyle='round,
 pad=0.3'))
        #______________________________________
        # Compute the new y_lower for next plot
        y_lower = y_upper + 10
```

In [35]: Hide

```
#____________________________________
# define individual silouhette scores
sample_silhouette_values = silhouette_samples(matrix, clusters)
#__________________
# and do the graph
graph_component_silhouette(n_clusters, [-0.07, 0.33], len(X), sample_silhouette_v
alues, clusters)
```





b/ *Word Cloud*

Now we can have a look at the type of objects that each cluster represents. In order to obtain a global view of their contents, I determine which keywords are the most frequent in each of them

In [36]:

Hide

```
liste = pd.DataFrame(liste_produits)
liste_words = [word for (word, occurence) in list_products]

occurence = [dict() for _ in range(n_clusters)]

for i in range(n_clusters):
    liste_cluster = liste.loc[clusters == i]
    for word in liste_words:
        if word in ['art', 'set', 'heart', 'pink', 'blue', 'tag']: continue
        occurence[i][word] = sum(liste_cluster.loc[:, 0].str.contains(word.upper
()))
```

and I output the result as wordclouds:

In [37]:

Hide

```
#_______________________________________________________________________
def random_color_func(word=None, font_size=None, position=None,
                      orientation=None, font_path=None, random_state=None):
    h = int(360.0 * tone / 255.0)
    s = int(100.0 * 255.0 / 255.0)
    l = int(100.0 * float(random_state.randint(70, 120)) / 255.0)
    return "hsl({}, {}%, {}%)".format(h, s, l)
#_______________________________________________________________________
def make_wordcloud(liste, increment):
    ax1 = fig.add_subplot(4,2,increment)
    words = dict()
    trunc_occurences = liste[0:150]
```

```
    for s in trunc_occurences:
        words[s[0]] = s[1]
    #________________________________________________________
    wordcloud = WordCloud(width=1000,height=400, background_color='lightgrey',
                          max_words=1628,relative_scaling=1,
                          color_func = random_color_func,
                          normalize_plurals=False)
    wordcloud.generate_from_frequencies(words)
    ax1.imshow(wordcloud, interpolation="bilinear")
    ax1.axis('off')
    plt.title('cluster nº{}'.format(increment-1))
#________________________________________________________________________
fig = plt.figure(1, figsize=(14,14))
color = [0, 160, 130, 95, 280, 40, 330, 110, 25]
for i in range(n_clusters):
    list_cluster_occurences = occurence[i]

    tone = color[i] # define the color of the words
    liste = []
    for key, value in list_cluster_occurences.items():
        liste.append([key, value])
    liste.sort(key = lambda x:x[1], reverse = True)
    make_wordcloud(liste, i+1)
```

From this representation, we can see that for example, one of the clusters contains objects that could be associated with gifts (keywords: Christmas, packaging, card, ...). Another cluster would rather contain luxury items and jewelry (keywords: necklace, bracelet, lace, silver, ...). Nevertheless, it can also be observed that many words appear in various clusters and it is therefore difficult to clearly distinguish them.

c / *Principal Component Analysis*

In order to ensure that these clusters are truly distinct, I look at their composition. Given the large number of variables of the initial matrix, I first perform a PCA:

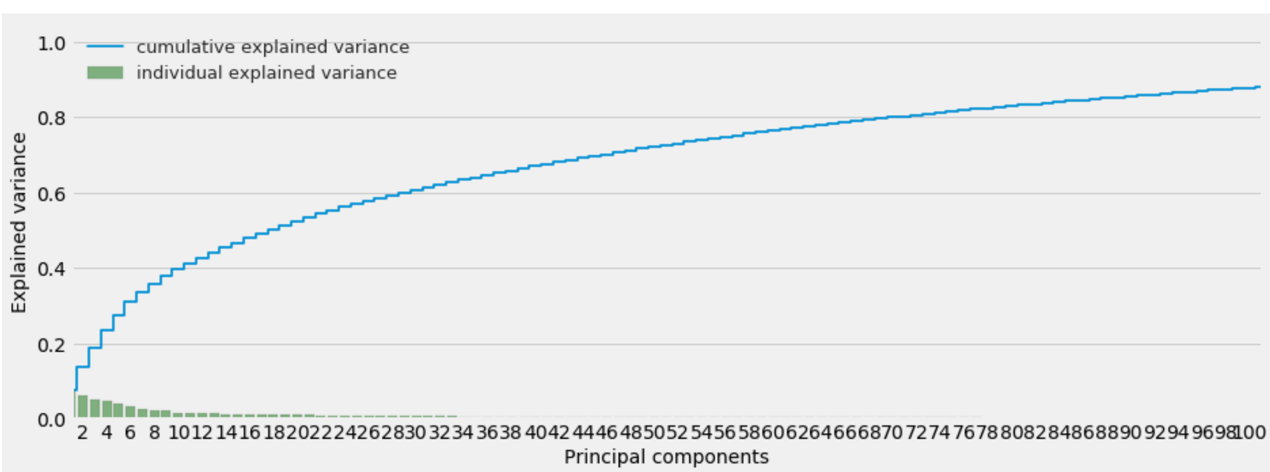
In [38]: Hide

```
pca = PCA()
pca.fit(matrix)
pca_samples = pca.transform(matrix)
```

and then check for the amount of variance explained by each component:

In [39]: Hide

```
fig, ax = plt.subplots(figsize=(14, 5))
sns.set(font_scale=1)
plt.step(range(matrix.shape[1]), pca.explained_variance_ratio_.cumsum(), where='m
id',
         label='cumulative explained variance')
sns.barplot(np.arange(1,matrix.shape[1]+1), pca.explained_variance_ratio_, alpha=
0.5, color = 'g',
            label='individual explained variance')
plt.xlim(0, 100)

ax.set_xticklabels([s if int(s.get_text())%2 == 0 else '' for s in ax.get_xtickla
bels()])

plt.ylabel('Explained variance', fontsize = 14)
plt.xlabel('Principal components', fontsize = 14)
plt.legend(loc='upper left', fontsize = 13);
```



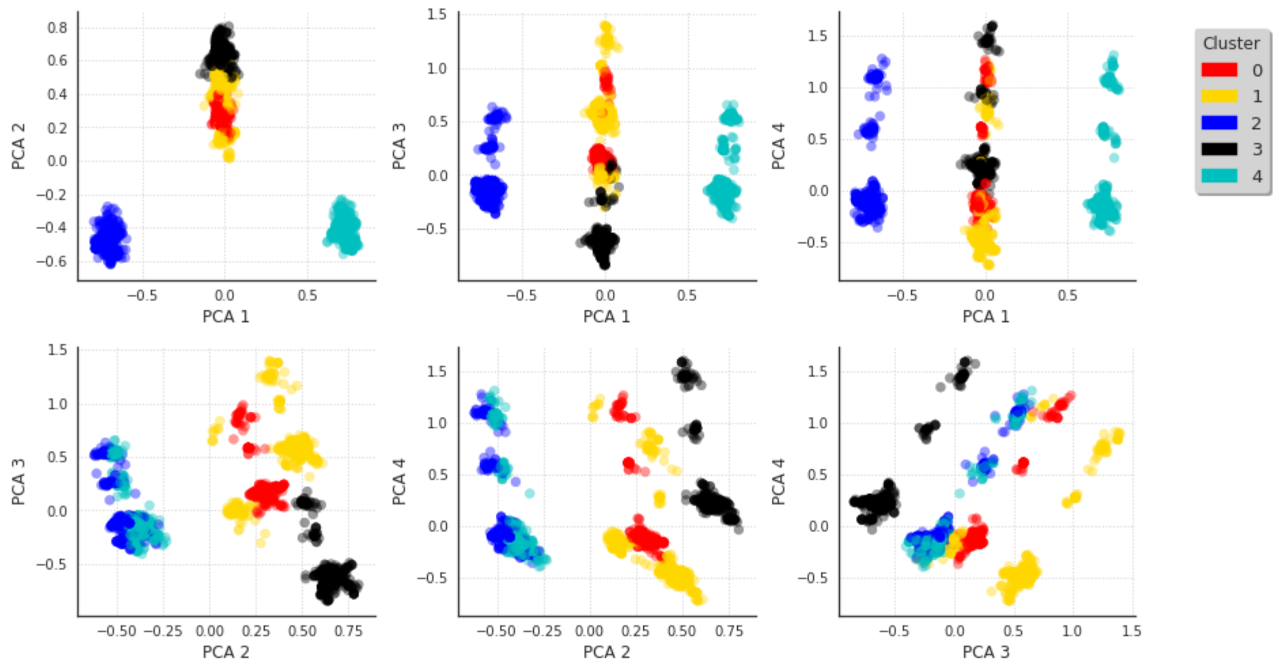
We see that the number of components required to explain the data is extremely important: we need more than 100 components to explain 90% of the variance of the data. In practice, I decide to keep only a limited number of components since this decomposition is only performed to visualize the data:

In [40]: Hide

```
pca = PCA(n_components=50)
matrix_9D = pca.fit_transform(matrix)
mat = pd.DataFrame(matrix_9D)
mat['cluster'] = pd.Series(clusters)
```

In [41]: Hide

```
import matplotlib.patches as mpatches

sns.set_style("white")
sns.set_context("notebook", font_scale=1, rc={"lines.linewidth": 2.5})

LABEL_COLOR_MAP = {0:'r', 1:'gold', 2:'b', 3:'k', 4:'c', 5:'g'}
label_color = [LABEL_COLOR_MAP[l] for l in mat['cluster']]

fig = plt.figure(figsize = (12,10))
increment = 0
for ix in range(4):
    for iy in range(ix+1, 4):
        increment += 1
        ax = fig.add_subplot(3,3,increment)
        ax.scatter(mat[ix], mat[iy], c= label_color, alpha=0.4)
        plt.ylabel('PCA {}'.format(iy+1), fontsize = 12)
        plt.xlabel('PCA {}'.format(ix+1), fontsize = 12)
        ax.yaxis.grid(color='lightgray', linestyle=':')
        ax.xaxis.grid(color='lightgray', linestyle=':')
        ax.spines['right'].set_visible(False)
        ax.spines['top'].set_visible(False)

        if increment == 9: break
    if increment == 9: break

#_______________________________________________
# I set the legend: abreviation -> airline name
comp_handler = []
for i in range(5):
    comp_handler.append(mpatches.Patch(color = LABEL_COLOR_MAP[i], label = i))

plt.legend(handles=comp_handler, bbox_to_anchor=(1.1, 0.97),
           title='Cluster', facecolor = 'lightgrey',
           shadow = True, frameon = True, framealpha = 1,
           fontsize = 13, bbox_transform = plt.gcf().transFigure)

plt.tight_layout()
```



4. Customer categories

4.1 Formatting data

In the previous section, the different products were grouped in five clusters. In order to prepare the rest of the analysis, a first step consists in introducing this information into the dataframe. To do this, I create the categorical variable **categ_product** where I indicate the cluster of each product :

In [42]: Hide

```
corresp = dict()
for key, val in zip (liste_produits, clusters):
    corresp[key] = val
#__________________________________________________________________________
df_cleaned['categ_product'] = df_cleaned.loc[:, 'Description'].map(corresp)
```

4.1.1 Grouping products

In a second step, I decide to create the **categ_N** variables (with $ N \in [0: 4]$) that contains the amount spent in each product category:

In [43]: Hide

```
for i in range(5):
    col = 'categ_{}'.format(i)
```

```
    df_temp = df_cleaned[df_cleaned['categ_product'] == i]
    price_temp = df_temp['UnitPrice'] * (df_temp['Quantity'] - df_temp['QuantityC
anceled'])
    price_temp = price_temp.apply(lambda x:x if x > 0 else 0)
    df_cleaned.loc[:, col] = price_temp
    df_cleaned[col].fillna(0, inplace = True)
#________________________________________________________________________________
_________________
df_cleaned[['InvoiceNo', 'Description', 'categ_product', 'categ_0', 'categ_1', 'c
ateg_2', 'categ_3','categ_4']][:5]
```

Out[43]:

	InvoiceNo	Description	categ_product	categ_0	categ_1	categ_2	categ_3	categ_4
0	536365	WHITE HANGING HEART T-LIGHT HOLDER	3	0.0	0.00	0.0	15.3	0.0
1	536365	WHITE METAL LANTERN	1	0.0	20.34	0.0	0.0	0.0
2	536365	CREAM CUPID HEARTS COAT HANGER	1	0.0	22.00	0.0	0.0	0.0
3	536365	KNITTED UNION FLAG HOT WATER BOTTLE	1	0.0	20.34	0.0	0.0	0.0
4	536365	RED WOOLLY HOTTIE WHITE HEART.	1	0.0	20.34	0.0	0.0	0.0

Up to now, the information related to a single order was split over several lines of the dataframe (one line per product). I decide to collect the information related to a particular order and put in in a single entry. I therefore create a new dataframe that contains, for each order, the amount of the basket, as well as the way it is distributed over the 5 categories of products:

In [44]: Hide

```
#___________________________________________
# somme des achats / utilisateur & commande
temp = df_cleaned.groupby(by=['CustomerID', 'InvoiceNo'], as_index=False)['TotalP
rice'].sum()
basket_price = temp.rename(columns = {'TotalPrice':'Basket Price'})
#____________________________________________________________
# pourcentage du prix de la commande / categorie de produit
for i in range(5):
    col = 'categ_{}'.format(i)
    temp = df_cleaned.groupby(by=['CustomerID', 'InvoiceNo'], as_index=False)[col
].sum()
    basket_price.loc[:, col] = temp
#_____________________
# date de la commande
df_cleaned['InvoiceDate_int'] = df_cleaned['InvoiceDate'].astype('int64')
temp = df_cleaned.groupby(by=['CustomerID', 'InvoiceNo'], as_index=False)['Invoic
eDate_int'].mean()
df_cleaned.drop('InvoiceDate_int', axis = 1, inplace = True)
```

```
basket_price.loc[:, 'InvoiceDate'] = pd.to_datetime(temp['InvoiceDate_int'])
#______________________________________
# selection des entrées significatives:
basket_price = basket_price[basket_price['Basket Price'] > 0]
basket_price.sort_values('CustomerID', ascending = True)[:5]
```

Out[44]:

	CustomerID	InvoiceNo	Basket Price	categ_0	categ_1	categ_2	categ_3	categ_4	InvoiceDate
1	12347	537626	711.79	124.44	293.35	23.40	83.40	187.2	2010-12-07 14:57:00.000001024
2	12347	542237	475.39	0.00	207.45	84.34	53.10	130.5	2011-01-26 14:29:59.999999744
3	12347	549222	636.25	0.00	153.25	81.00	71.10	330.9	2011-04-07 10:42:59.999999232
4	12347	556201	382.52	19.90	168.76	41.40	78.06	74.4	2011-06-09 13:01:00.000000256
5	12347	562032	584.91	97.80	196.41	61.30	119.70	109.7	2011-08-02 08:48:00.000000000

4.1.2 Separation of data over time

The dataframe `basket_price` contains information for a period of 12 months. Later, one of the objectives will be to develop a model capable of characterizing and anticipating the habits of the customers visiting the site and this, from their first visit. In order to be able to test the model in a realistic way, I split the data set by retaining the first 10 months to develop the model and the following two months to test it:

In [45]: Hide

```
print(basket_price['InvoiceDate'].min(), '->',  basket_price['InvoiceDate'].max
())
```

```
2010-12-01 08:26:00 -> 2011-12-09 12:50:00
```

In [46]: Hide

```
set_entrainement = basket_price[basket_price['InvoiceDate'] < datetime.date(2011,
10,1)]
set_test         = basket_price[basket_price['InvoiceDate'] >= datetime.date(2011
,10,1)]
basket_price = set_entrainement.copy(deep = True)
```

4.1.3 Consumer Order Combinations

In a second step, I group together the different entries that correspond to the same user. I thus determine the number of purchases made by the user, as well as the minimum, maximum, average amounts and the total amount spent during all the visits:

In [47]:

Hide

```
#__________________________________________________________________
# nb de visites et stats sur le montant du panier / utilisateurs
transactions_per_user=basket_price.groupby(by=['CustomerID'])['Basket Price'].agg
(['count','min','max','mean','sum'])
for i in range(5):
    col = 'categ_{}'.format(i)
    transactions_per_user.loc[:,col] = basket_price.groupby(by=['CustomerID'])[co
l].sum() /\
                                            transactions_per_user['sum']*100

transactions_per_user.reset_index(drop = False, inplace = True)
basket_price.groupby(by=['CustomerID'])['categ_0'].sum()
transactions_per_user.sort_values('CustomerID', ascending = True)[:5]
```

Out[47]:

	CustomerID	count	min	max	mean	sum	categ_0	categ_1	categ_2	categ_3
0	12347	5	382.52	711.79	558.172000	2790.86	8.676179	36.519926	10.442659	14.524555
1	12348	4	227.44	892.80	449.310000	1797.24	0.000000	20.030714	38.016069	0.000000
2	12350	1	334.40	334.40	334.400000	334.40	0.000000	11.961722	11.692584	27.900718
3	12352	6	144.35	840.30	345.663333	2073.98	14.301006	68.944734	0.491808	3.370331
4	12353	1	89.00	89.00	89.000000	89.00	22.359551	44.719101	0.000000	19.887640

Finally, I define two additional variables that give the number of days elapsed since the first purchase (**FirstPurchase**) and the number of days since the last purchase (**LastPurchase**):

In [48]:

Hide

```
last_date = basket_price['InvoiceDate'].max().date()

first_registration = pd.DataFrame(basket_price.groupby(by=['CustomerID'])['Invoic
eDate'].min())
last_purchase      = pd.DataFrame(basket_price.groupby(by=['CustomerID'])['Invoic
eDate'].max())

test  = first_registration.applymap(lambda x:(last_date - x.date()).days)
test2 = last_purchase.applymap(lambda x:(last_date - x.date()).days)

transactions_per_user.loc[:, 'LastPurchase'] = test2.reset_index(drop = False)['I
nvoiceDate']
transactions_per_user.loc[:, 'FirstPurchase'] = test.reset_index(drop = False)['I
nvoiceDate']
```

```
transactions_per_user[:5]
```

Out[48]:

	CustomerID	count	min	max	mean	sum	categ_0	categ_1	categ_2	categ_3
0	12347	5	382.52	711.79	558.172000	2790.86	8.676179	36.519926	10.442659	14.524555
1	12348	4	227.44	892.80	449.310000	1797.24	0.000000	20.030714	38.016069	0.000000
2	12350	1	334.40	334.40	334.400000	334.40	0.000000	11.961722	11.692584	27.900718
3	12352	6	144.35	840.30	345.663333	2073.98	14.301006	68.944734	0.491808	3.370331
4	12353	1	89.00	89.00	89.000000	89.00	22.359551	44.719101	0.000000	19.887640

A customer category of particular interest is that of customers who make only one purchase. One of the objectives may be, for example, to target these customers in order to retain them. In part, I find that this type of customer represents 1/3 of the customers listed:

In [49]: Hide

```
n1 = transactions_per_user[transactions_per_user['count'] == 1].shape[0]
n2 = transactions_per_user.shape[0]
print("nb. de clients avec achat unique: {:<2}/{:<5} ({:<2.2f}%)".format(n1,n2,n1
/n2*100))
```

```
nb. de clients avec achat unique: 1445/3608  (40.05%)
```

4.2 Creation of customers categories

4.2.1 Data encoding

The dataframe `transactions_per_user` contains a summary of all the commands that were made. Each entry in this dataframe corresponds to a particular client. I use this information to characterize the different types of customers and only keep a subset of variables:

In [50]: Hide

```
list_cols = ['count','min','max','mean','categ_0','categ_1','categ_2','categ_3',
'categ_4']
#____________________________________________________________
selected_customers = transactions_per_user.copy(deep = True)
matrix = selected_customers[list_cols].as_matrix()
```

In practice, the different variables I selected have quite different ranges of variation and before continuing the analysis, I create a matrix where these data are standardized:

In [51]:

Hide

```
scaler = StandardScaler()
scaler.fit(matrix)
print('variables mean values: \n' + 90*'-' + '\n' , scaler.mean_)
scaled_matrix = scaler.transform(matrix)
```

```
variables mean values:
------------------------------------------------------------------------------------------
-----
 [   3.62305987  259.93189634  556.26687999  377.06036244   15.67936332
   23.91238925   13.98907929   21.19884856   25.22916919]
```

In the following, I will create clusters of customers. In practice, before creating these clusters, it is interesting to define a base of smaller dimension allowing to describe the `scaled_matrix` matrix. In this case, I will use this base in order to create a representation of the different clusters and thus verify the quality of the separation of the different groups. I therefore perform a PCA beforehand:

In [52]:

Hide

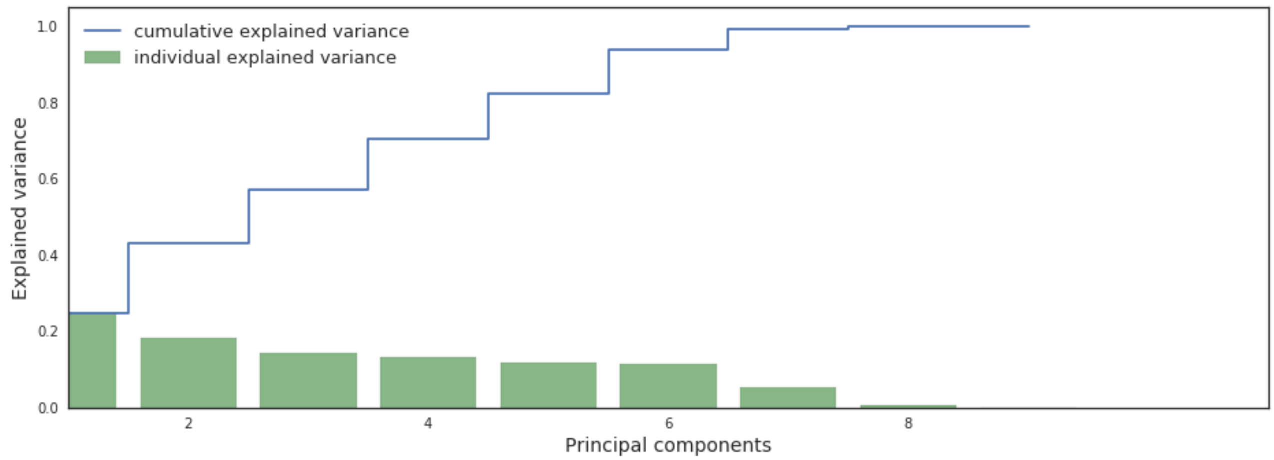
```
pca = PCA()
pca.fit(scaled_matrix)
pca_samples = pca.transform(scaled_matrix)
```

and I represent the amount of variance explained by each of the components:

In [53]:

Hide

```
fig, ax = plt.subplots(figsize=(14, 5))
sns.set(font_scale=1)
plt.step(range(matrix.shape[1]), pca.explained_variance_ratio_.cumsum(), where='m
id',
         label='cumulative explained variance')
sns.barplot(np.arange(1,matrix.shape[1]+1), pca.explained_variance_ratio_, alpha=
0.5, color = 'g',
            label='individual explained variance')
plt.xlim(0, 10)

ax.set_xticklabels([s if int(s.get_text())%2 == 0 else '' for s in ax.get_xtickla
bels()])

plt.ylabel('Explained variance', fontsize = 14)
plt.xlabel('Principal components', fontsize = 14)
plt.legend(loc='best', fontsize = 13);
```



4.2.2 Creation of customer categories

At this point, I define clusters of clients from the standardized matrix that was defined earlier and using the `k-means` algorithm from `scikit-learn`. I choose the number of clusters based on the silhouette score and I find that the best score is obtained with 11 clusters:

In [54]:

```
n_clusters = 11
kmeans = KMeans(init='k-means++', n_clusters = n_clusters, n_init=100)
kmeans.fit(scaled_matrix)
clusters_clients = kmeans.predict(scaled_matrix)
silhouette_avg = silhouette_score(scaled_matrix, clusters_clients)
print('score de silhouette: {:<.3f}'.format(silhouette_avg))
```

Hide

```
score de silhouette: 0.213
```

At first, I look at the number of customers in each cluster:

In [55]:

```
pd.DataFrame(pd.Series(clusters_clients).value_counts(), columns = ['nb. de clien
ts']).T
```

Hide

Out[55]:

	4	8	1	5	10	2	0	7	6	3	9
nb. de clients	1453	476	433	351	293	235	185	153	12	10	7

a / *Report via the PCA*

There is a certain disparity in the sizes of different groups that have been created. Hence I will now try to understand the content of these clusters in order to validate (or not) this particular separation. At first, I use the result of the PCA:

In [56]: Hide

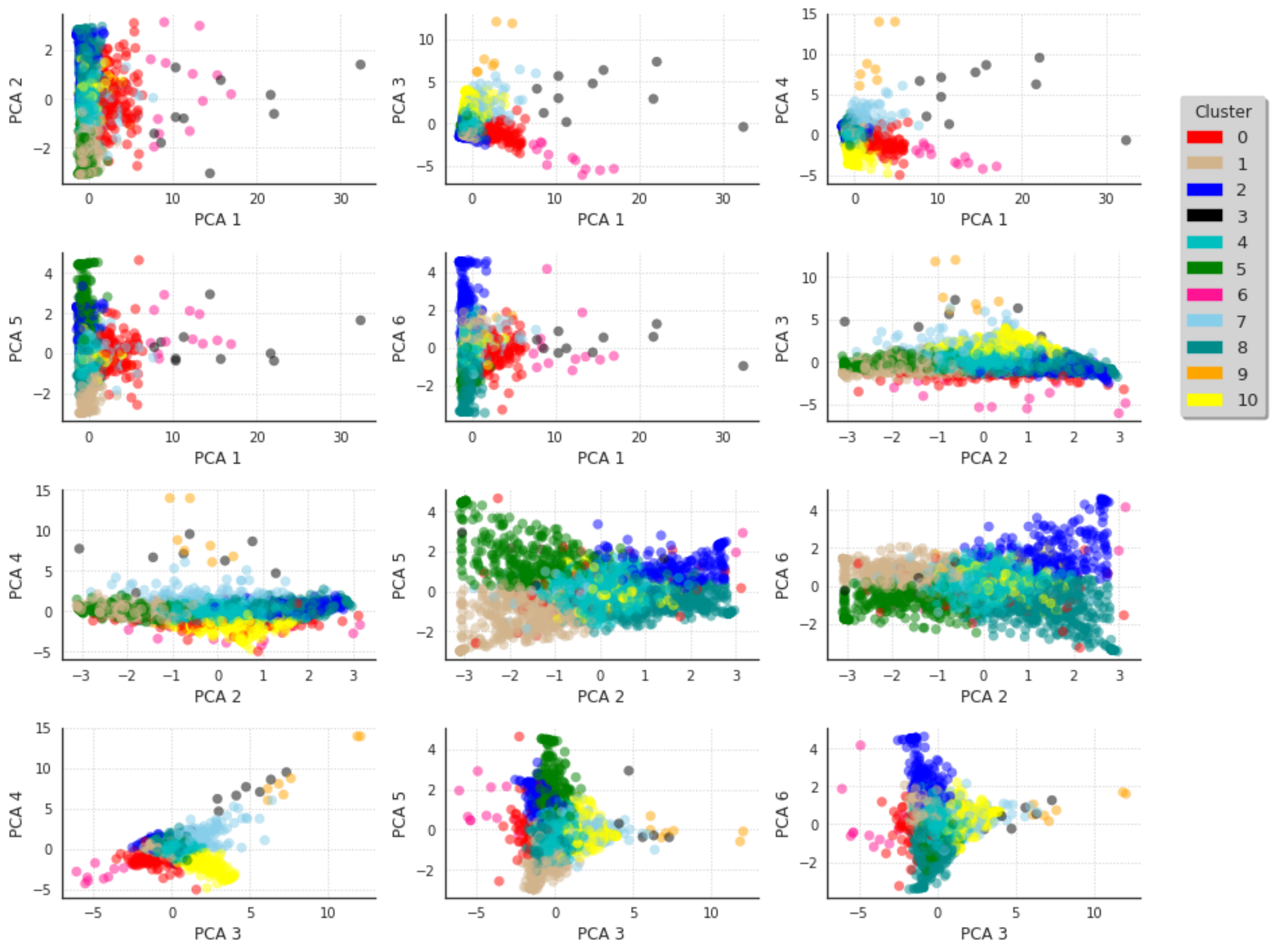
```
pca = PCA(n_components=6)
matrix_3D = pca.fit_transform(scaled_matrix)
mat = pd.DataFrame(matrix_3D)
mat['cluster'] = pd.Series(clusters_clients)
```

in order to create a representation of the various clusters:

In [57]: Hide

```
import matplotlib.patches as mpatches

sns.set_style("white")
sns.set_context("notebook", font_scale=1, rc={"lines.linewidth": 2.5})

LABEL_COLOR_MAP = {0:'r', 1:'tan', 2:'b', 3:'k', 4:'c', 5:'g', 6:'deeppink', 7:'s
kyblue', 8:'darkcyan', 9:'orange',
                   10:'yellow', 11:'tomato', 12:'seagreen'}
label_color = [LABEL_COLOR_MAP[l] for l in mat['cluster']]

fig = plt.figure(figsize = (12,10))
increment = 0
for ix in range(6):
    for iy in range(ix+1, 6):
        increment += 1
        ax = fig.add_subplot(4,3,increment)
        ax.scatter(mat[ix], mat[iy], c= label_color, alpha=0.5)
        plt.ylabel('PCA {}'.format(iy+1), fontsize = 12)
        plt.xlabel('PCA {}'.format(ix+1), fontsize = 12)
        ax.yaxis.grid(color='lightgray', linestyle=':')
        ax.xaxis.grid(color='lightgray', linestyle=':')
        ax.spines['right'].set_visible(False)
        ax.spines['top'].set_visible(False)

        if increment == 12: break
    if increment == 12: break

#_______________________________________________
# I set the legend: abreviation -> airline name
comp_handler = []
for i in range(n_clusters):
    comp_handler.append(mpatches.Patch(color = LABEL_COLOR_MAP[i], label = i))
```

```
plt.legend(handles=comp_handler, bbox_to_anchor=(1.1, 0.9),
           title='Cluster', facecolor = 'lightgrey',
           shadow = True, frameon = True, framealpha = 1,
           fontsize = 13, bbox_transform = plt.gcf().transFigure)

plt.tight_layout()
```



From this representation, it can be seen, for example, that the first principal component allow to separate the tiniest clusters from the rest. More generally, we see that there is always a representation in which two clusters will appear to be distinct.

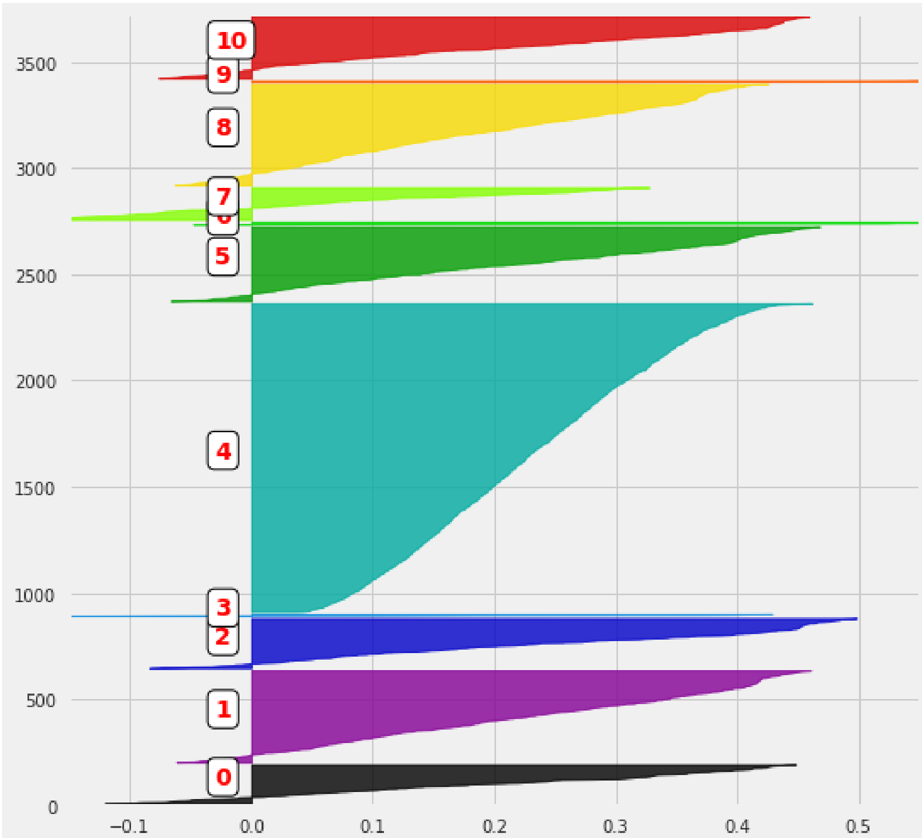
b/ *Score de silhouette intra-cluster*

As with product categories, another way to look at the quality of the separation is to look at silouhette scores within different clusters:

In [58]: Hide

```
sample_silhouette_values = silhouette_samples(scaled_matrix, clusters_clients)
#____________________________________
# define individual silouhette scores
sample_silhouette_values = silhouette_samples(scaled_matrix, clusters_clients)
#__________________
```

```
# and do the graph
graph_component_silhouette(n_clusters, [-0.15, 0.55], len(scaled_matrix), sample_
silhouette_values, clusters_clients)
```



c/ Customers morphotype

At this stage, I have verified that the different clusters are indeed disjoint (at least, in a global way). It remains to understand the habits of the customers in each cluster. To do so, I start by adding to the `selected_customers` dataframe a variable that defines the cluster to which each client belongs:

In [59]:

```
selected_customers.loc[:, 'cluster'] = clusters_clients
```

Then, I average the contents of this dataframe by first selecting the different groups of clients. This gives access to, for example, the average baskets price, the number of visits or the total sums spent by the clients of the different clusters. I also determine the number of clients in each group (variable **size**):

In [60]:

Hide

```
merged_df = pd.DataFrame()
for i in range(n_clusters):
    test = pd.DataFrame(selected_customers[selected_customers['cluster'] == i].me
an())
    test = test.T.set_index('cluster', drop = True)
    test['size'] = selected_customers[selected_customers['cluster'] == i].shape[0
```

```
    test['size'] = selected_customers[selected_customers['cluster'] == i].shape[0
]
    merged_df = pd.concat([merged_df, test])
#_____________________________________________________
merged_df.drop('CustomerID', axis = 1, inplace = True)
print('number of customers:', merged_df['size'].sum())

merged_df = merged_df.sort_values('sum')
```

```
number of customers: 3608
```

Finally, I re-organize the content of the dataframe by ordering the different clusters: first, in relation to the amount wpsent in each product category and then, according to the total amount spent:

In [61]:

Hide

```
liste_index = []
for i in range(5):
    column = 'categ_{}'.format(i)
    liste_index.append(merged_df[merged_df[column] > 45].index.values[0])
#___________________________________
liste_index_reordered = liste_index
liste_index_reordered += [ s for s in merged_df.index if s not in liste_index]
#___________________________________________________________
merged_df = merged_df.reindex(index = liste_index_reordered)
merged_df = merged_df.reset_index(drop = False)
display(merged_df[['cluster', 'count', 'min', 'max', 'mean', 'sum', 'categ_0',
                   'categ_1', 'categ_2', 'categ_3', 'categ_4', 'size']])
```

	cluster	count	min	max	mean	sum	categ_0	categ_1	ca
0	5.0	2.501425	193.559775	313.719972	247.197748	639.210516	52.104009	19.310955	5.283
1	1.0	2.251732	210.875727	360.404688	274.749495	706.889723	12.889648	59.600437	5.221
2	2.0	2.234043	192.611319	320.940596	248.095380	597.051489	5.442219	8.084564	57.20
3	10.0	2.593857	211.443311	381.836143	292.761811	823.467918	7.268753	9.601113	7.030
4	8.0	2.449580	216.744496	334.344750	272.473924	679.057838	6.086111	11.322272	13.14
5	4.0	3.276669	216.697151	455.781398	327.445094	1083.386511	14.375830	23.742257	13.79
6	0.0	1.697297	1047.116541	1373.455465	1196.774113	2084.702059	13.840762	26.321130	12.13
7	6.0	1.666667	3480.920833	3966.812500	3700.139306	5949.600000	18.278470	25.406109	22.89
8	7.0	18.281046	89.030915	1606.934575	576.333843	9965.851242	15.554552	22.948047	12.25
9	9.0	92.000000	10.985714	1858.250000	374.601553	34845.105714	17.721038	20.826842	13.11
10	3.0	23.200000	415.148000	17158.271000	4853.774161	87883.059000	22.390560	25.974481	7.172

d / *Customers morphology*

Finally, I created a representation of the different morphotypes. To do this, I define a class to create "Radar Charts" (which has been adapted from this kernel (https://www.kaggle.com/yassineghouzam/don-t-know-why-employees-leave -read-this)):

In [62]:

Hide

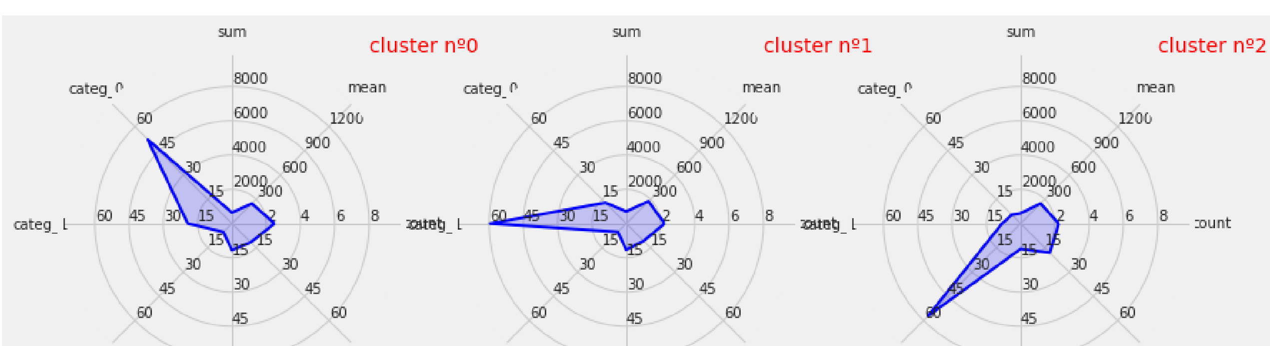
```
def _scale_data(data, ranges):
    (x1, x2) = ranges[0]
    d = data[0]
    return [(d - y1) / (y2 - y1) * (x2 - x1) + x1 for d, (y1, y2) in zip(data, ra
nges)]

class RadarChart():
    def __init__(self, fig, location, sizes, variables, ranges, n_ordinate_levels
 = 6):

        angles = np.arange(0, 360, 360./len(variables))

        ix, iy = location[:] ; size_x, size_y = sizes[:]

        axes = [fig.add_axes([ix, iy, size_x, size_y], polar = True,
        label = "axes{}".format(i)) for i in range(len(variables))]

        _, text = axes[0].set_thetagrids(angles, labels = variables)

        for txt, angle in zip(text, angles):
            if angle > -1 and angle < 181:
                txt.set_rotation(angle - 90)
            else:
                txt.set_rotation(angle - 270)

        for ax in axes[1:]:
            ax.patch.set_visible(False)
            ax.xaxis.set_visible(False)
            ax.grid("off")

        for i, ax in enumerate(axes):
            grid = np.linspace(*ranges[i],num = n_ordinate_levels)
            grid_label = [""]+["{:.0f}".format(x) for x in grid[1:-1]]
            ax.set_rgrids(grid, labels = grid_label, angle = angles[i])
            ax.set_ylim(*ranges[i])

        self.angle = np.deg2rad(np.r_[angles, angles[0]])
        self.ranges = ranges
        self.ax = axes[0]

    def plot(self, data, *args, **kw):
        sdata = _scale_data(data, self.ranges)
```

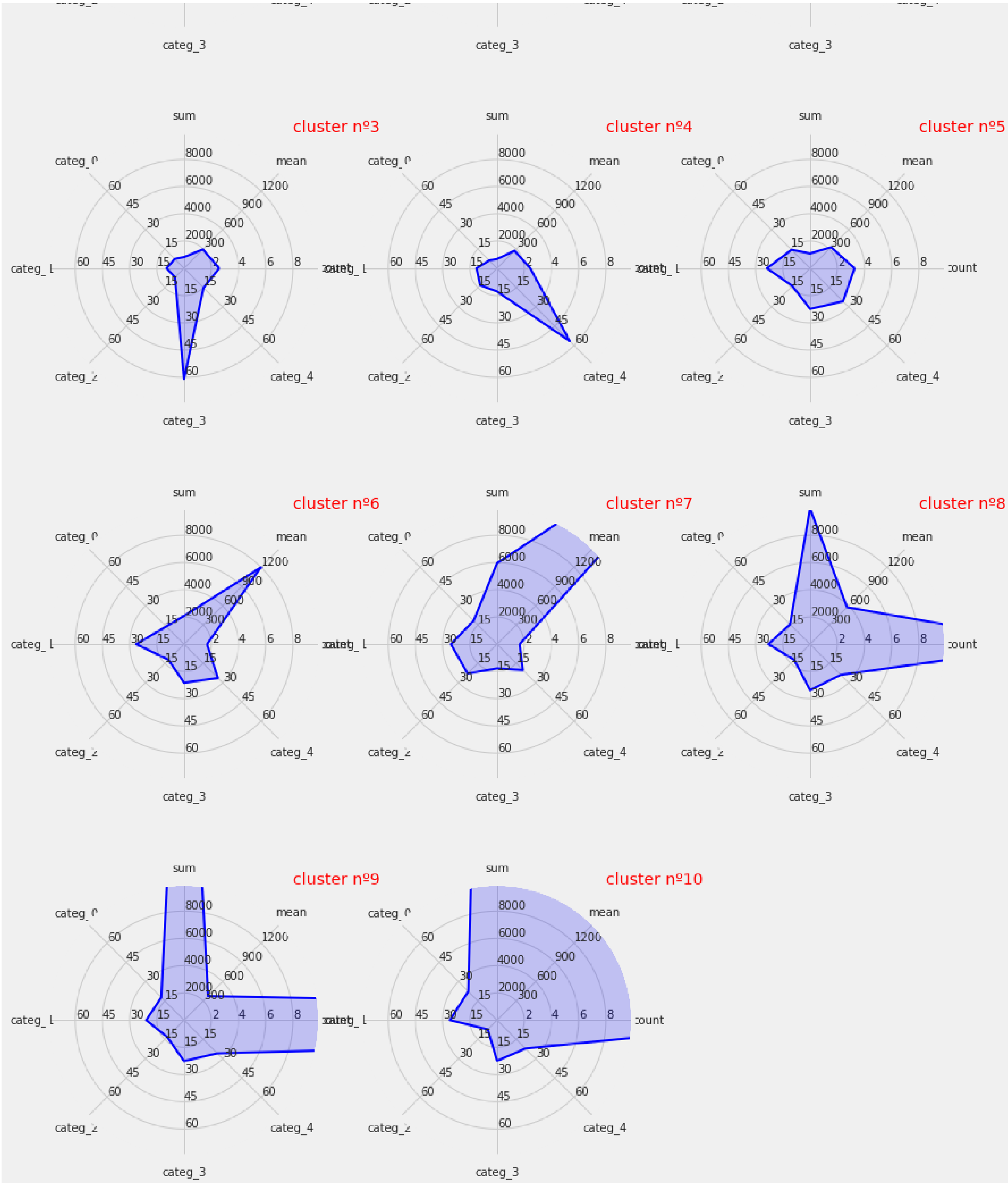
```
        self.ax.plot(self.angle, np.r_[sdata, sdata[0]], *args, **kw)

    def fill(self, data, *args, **kw):
        sdata = _scale_data(data, self.ranges)
        self.ax.fill(self.angle, np.r_[sdata, sdata[0]], *args, **kw)

    def legend(self, *args, **kw):
        self.ax.legend(*args, **kw)

    def title(self, title, *args, **kw):
        self.ax.text(0.9, 1, title, transform = self.ax.transAxes, *args, **kw)
```

This allows to have a global view of the content of each cluster:

In [63]: Hide

```
fig = plt.figure(figsize=(10,12))

attributes = ['count', 'mean', 'sum', 'categ_0', 'categ_1', 'categ_2', 'categ_3',
 'categ_4']
ranges = [[0.01, 10], [0.01, 1500], [0.01, 10000], [0.01, 75], [0.01, 75], [0.01,
 75], [0.01, 75], [0.01, 75]]
index  = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

n_groups = n_clusters ; i_cols = 3
i_rows = n_groups//i_cols
size_x, size_y = (1/i_cols), (1/i_rows)

for ind in range(n_clusters):
    ix = ind%3 ; iy = i_rows - ind//3
    pos_x = ix*(size_x + 0.05) ; pos_y = iy*(size_y + 0.05)
    location = [pos_x, pos_y]  ; sizes = [size_x, size_y]
    #______________________________________________________
    data = np.array(merged_df.loc[index[ind], attributes])
    radar = RadarChart(fig, location, sizes, attributes, ranges)
    radar.plot(data, color = 'b', linewidth=2.0)
    radar.fill(data, alpha = 0.2, color = 'b')
    radar.title(title = 'cluster nº{}'.format(index[ind]), color = 'r')
    ind += 1
```





It can be seen, for example, that the first 5 clusters correspond to a strong preponderance of purchases in a particular category of products. Other clusters will differ from basket averages (**mean**), the total sum spent by the clients (**sum**) or the total number of visits made (**count**).

5. Classification of customers

In this part, the objective will be to adjust a classifier that will classify consumers in the different client categories that were established in the previous section. The objective is to make this classification possible at the first visit. To fulfill this objective, I will test several classifiers implemented in `scikit-learn`. First, in order to simplify their use, I define a class that allows to interface several of the functionalities common to these different classifiers:

In [64]: Hide

```
class Class_Fit(object):
    def __init__(self, clf, params=None):
        if params:
            self.clf = clf(**params)
        else:
            self.clf = clf()

    def train(self, x_train, y_train):
        self.clf.fit(x_train, y_train)

    def predict(self, x):
        return self.clf.predict(x)

    def grid_search(self, parameters, Kfold):
        self.grid = GridSearchCV(estimator = self.clf, param_grid = parameters, c
v = Kfold)

    def grid_fit(self, X, Y):
        self.grid.fit(X, Y)

    def grid_predict(self, X, Y):
        self.predictions = self.grid.predict(X)
        print("Precision: {:.2f} % ".format(100*metrics.accuracy_score(Y, self.pr
edictions)))
```

Since the goal is to define the class to which a client belongs and this, as soon as its first visit, I only keep the variables that describe the content of the basket, and do not take into account the variables related to the frequency of visits or variations of the basket price over time:

In [65]: Hide

```
columns = ['mean', 'categ_0', 'categ_1', 'categ_2', 'categ_3', 'categ_4' ]
X = selected_customers[columns]
Y = selected_customers['cluster']
```

Finally, I split the dataset in train and test sets:

In [66]: Hide

```
X_train, X_test, Y_train, Y_test = model_selection.train_test_split(X, Y, train_s
ize = 0.8)
```

5.1 Support Vector Machine Classifier (SVC)

The first classifier I use is the SVC classifier. In order to use it, I create an instance of the `Class_Fit` class and then call `grid_search()` . When calling this method, I provide as parameters:

- the hyperparameters for which I will seek an optimal value
- the number of folds to be used for cross-validation

In [67]:

```
svc = Class_Fit(clf = svm.LinearSVC)
svc.grid_search(parameters = [{'C':np.logspace(-2,2,10)}], Kfold = 5)
```

Hide

Once this instance is created, I adjust the classifier to the training data:

In [68]:

```
svc.grid_fit(X = X_train, Y = Y_train)
```

Hide

then I can test the quality of the prediction with respect to the test data:

In [69]:

```
svc.grid_predict(X_test, Y_test)
```

Hide

```
Precision: 83.93 %
```

5.1.1 Confusion matrix

The accuracy of the results seems to be correct. Nevertheless, let us remember that when the different classes were defined, there was an imbalance in size between the classes obtained. In particular, one class contains around 40% of the clients. It is therefore interesting to look at how the predictions and real values compare to the breasts of the different classes. This is the subject of the confusion matrices and to represent them, I use the code of the sklearn documentation (http://scikit-learn.org/stable/auto_examples/model_selection/plot_confusion_matrix.html):

In [70]:

```
def plot_confusion_matrix(cm, classes, normalize=False, title='Confusion matrix',
 cmap=plt.cm.Blues):
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')
```

Hide

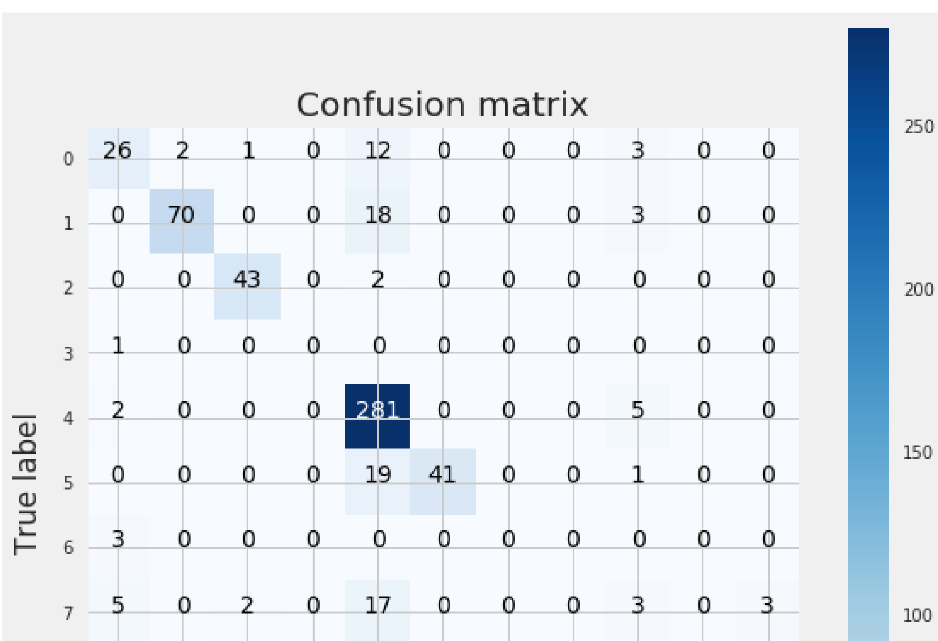
```
    #_________________________________________________
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=0)
    plt.yticks(tick_marks, classes)
    #_________________________________________________
    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")
    #_________________________________________________
    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
```

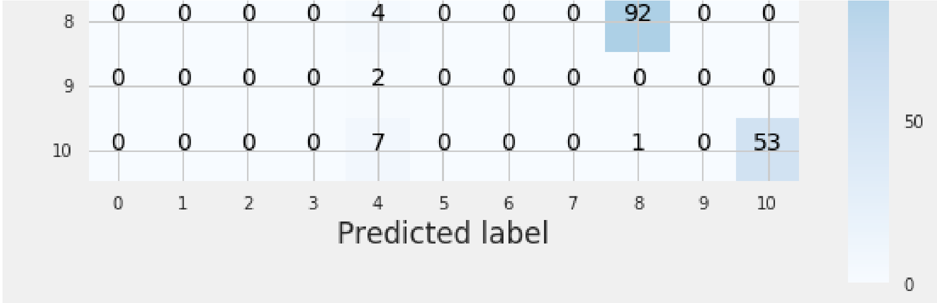
from which I create the following representation:

In [71]: Hide

```
class_names = [i for i in range(11)]
cnf_matrix = confusion_matrix(Y_test, svc.predictions)
np.set_printoptions(precision=2)
plt.figure(figsize = (8,8))
plot_confusion_matrix(cnf_matrix, classes=class_names, normalize = False, title=
'Confusion matrix')
```

```
Confusion matrix, without normalization
```





5.1.2 Learning curve

A typical way to test the quality of a fit is to draw a learning curve. In particular, this type of curves allow to detect possible drawbacks in the model, linked for example to over- or under-fitting. This also shows to which extent the mode could benefit from a larger data sample. In order to draw this curve, I use the scikit-learn documentation code again (http://scikit-learn.org/stable/auto_examples/model_selection/plot_learning_curve.html#sphx-glr- self-examples-model-selection-pad-learning-curve-py)

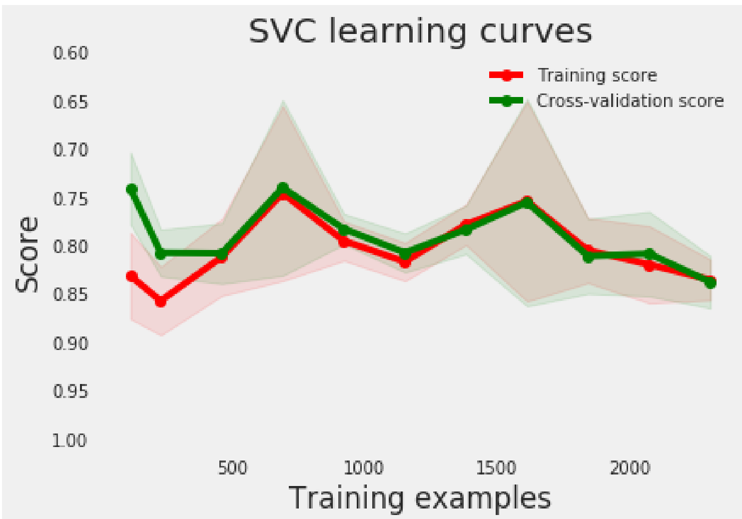
In [72]: Hide

```
def plot_learning_curve(estimator, title, X, y, ylim=None, cv=None,
                        n_jobs=-1, train_sizes=np.linspace(.1, 1.0, 10)):
    """Generate a simple plot of the test and training learning curve"""
    plt.figure()
    plt.title(title)
    if ylim is not None:
        plt.ylim(*ylim)
    plt.xlabel("Training examples")
    plt.ylabel("Score")
    train_sizes, train_scores, test_scores = learning_curve(
        estimator, X, y, cv=cv, n_jobs=n_jobs, train_sizes=train_sizes)
    train_scores_mean = np.mean(train_scores, axis=1)
    train_scores_std = np.std(train_scores, axis=1)
    test_scores_mean = np.mean(test_scores, axis=1)
    test_scores_std = np.std(test_scores, axis=1)
    plt.grid()

    plt.fill_between(train_sizes, train_scores_mean - train_scores_std,
                     train_scores_mean + train_scores_std, alpha=0.1, color="r")
    plt.fill_between(train_sizes, test_scores_mean - test_scores_std,
                     test_scores_mean + test_scores_std, alpha=0.1, color="g")
    plt.plot(train_sizes, train_scores_mean, 'o-', color="r", label="Training sco
re")
    plt.plot(train_sizes, test_scores_mean, 'o-', color="g", label="Cross-validat
ion score")

    plt.legend(loc="best")
    return plt
```

from which I represent the leanring curve of the SVC classifier:

In [73]: Hide

```
g = plot_learning_curve(svc.grid.best_estimator_,
                        "SVC learning curves", X_train, Y_train, ylim = [1.01, 0.6],
                        cv = 5,  train_sizes = [0.05, 0.1, 0.2, 0.3, 0.4, 0.5,
                                                0.6, 0.7, 0.8, 0.9, 1])
```



On this curve, we can see that the train and cross-validation curves converge towards the same limit when the sample size increases. This is typical of modeling with low variance and proves that the model does not suffer from overfitting. Also, we can see that the accuracy of the training curve is correct which is synonymous of a low bias. Hence the model does not underfit the data.

5.2 Logistic Regression

I now consider the logistic regression classifier. As before, I create an instance of the `Class_Fit` class, adjust the model on the training data and see how the predictions compare to the real values:

In [74]: Hide

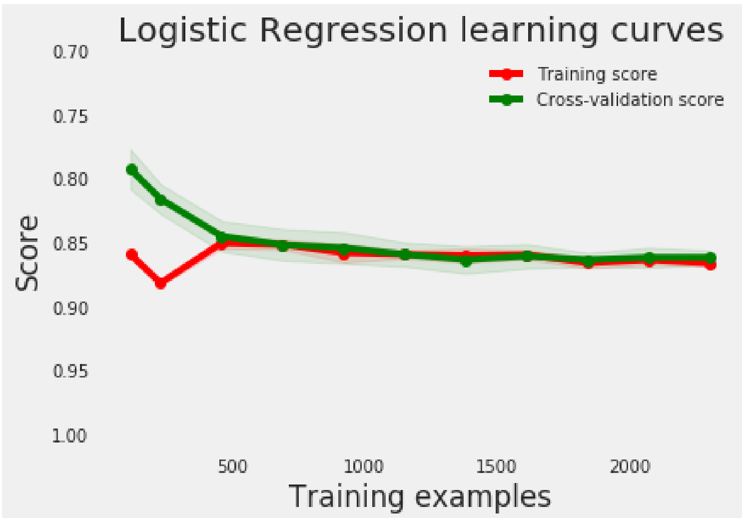
```
lr = Class_Fit(clf = linear_model.LogisticRegression)
lr.grid_search(parameters = [{'C':np.logspace(-2,2,20)}], Kfold = 5)
lr.grid_fit(X = X_train, Y = Y_train)
lr.grid_predict(X_test, Y_test)
```

```
Precision: 85.04 %
```

Then, I plot the learning curve to have a feeling of the quality of the model:

In [75]: Hide

```
g = plot_learning_curve(lr.grid.best_estimator_, "Logistic Regression learning cu
rves", X_train, Y_train,
                        ylim = [1.01, 0.7], cv = 5,
                        train_sizes = [0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7,
0.8, 0.9, 1])
```



5.3 k-Nearest Neighbors

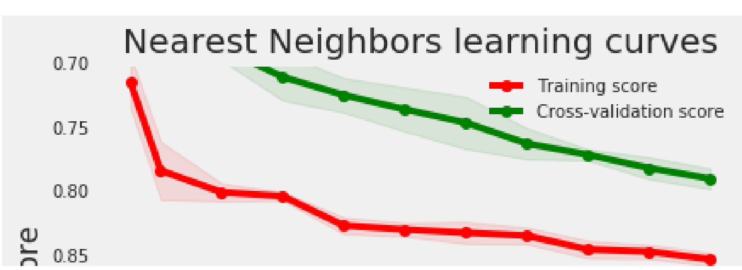
In [76]: Hide

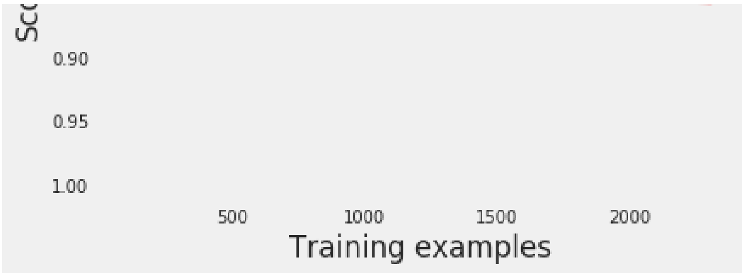
```
knn = Class_Fit(clf = neighbors.KNeighborsClassifier)
knn.grid_search(parameters = [{'n_neighbors': np.arange(1,50,1)}], Kfold = 5)
knn.grid_fit(X = X_train, Y = Y_train)
knn.grid_predict(X_test, Y_test)
```

```
Precision: 80.75 %
```

In [77]: Hide

```
g = plot_learning_curve(knn.grid.best_estimator_, "Nearest Neighbors learning cur
ves", X_train, Y_train,
                        ylim = [1.01, 0.7], cv = 5,
                        train_sizes = [0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7,
0.8, 0.9, 1])
```





5.4 Decision Tree

In [78]:

Hide

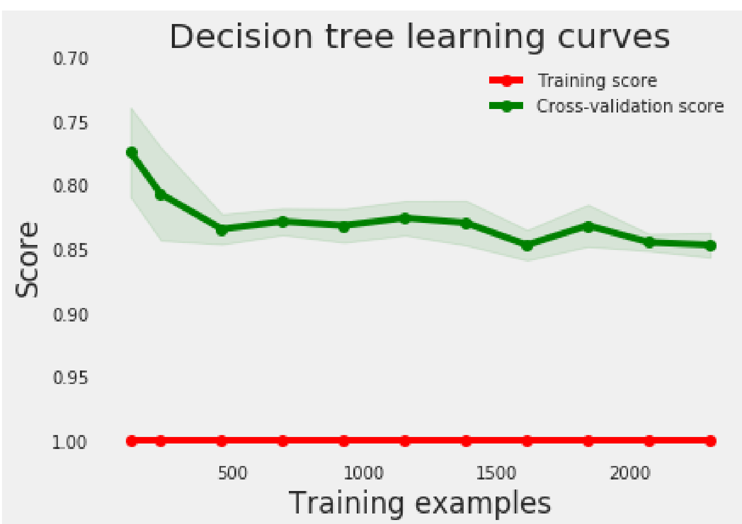
```
tr = Class_Fit(clf = tree.DecisionTreeClassifier)
tr.grid_search(parameters = [{'criterion' : ['entropy', 'gini'], 'max_features' :
['sqrt', 'log2']}], Kfold = 5)
tr.grid_fit(X = X_train, Y = Y_train)
tr.grid_predict(X_test, Y_test)
```

```
Precision: 81.16 %
```

In [79]:

Hide

```
g = plot_learning_curve(tr.grid.best_estimator_, "Decision tree learning curves",
 X_train, Y_train,
                        ylim = [1.01, 0.7], cv = 5,
                        train_sizes = [0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7,
0.8, 0.9, 1])
```



5.5 Random Forest

In [80]:

Hide

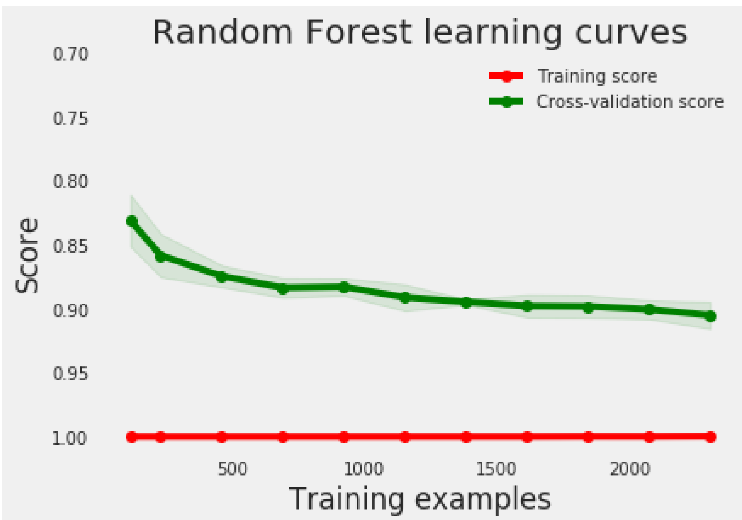
```
rf = Class_Fit(clf = ensemble.RandomForestClassifier)
```

```
param_grid = {'criterion' : ['entropy', 'gini'], 'n_estimators' : [20, 40, 60, 80
, 100],
               'max_features' :['sqrt', 'log2']}
rf.grid_search(parameters = param_grid, Kfold = 5)
rf.grid_fit(X = X_train, Y = Y_train)
rf.grid_predict(X_test, Y_test)
```

```
Precision: 89.47 %
```

In [81]: Hide

```
g = plot_learning_curve(rf.grid.best_estimator_, "Random Forest learning curves",
 X_train, Y_train,
                        ylim = [1.01, 0.7], cv = 5,
                        train_sizes = [0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7,
0.8, 0.9, 1])
```



5.6 AdaBoost Classifier

In [82]: Hide

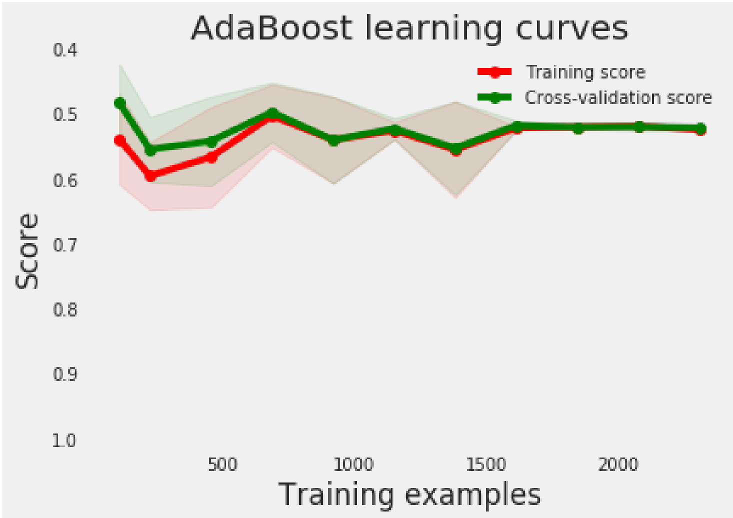
```
ada = Class_Fit(clf = AdaBoostClassifier)
param_grid = {'n_estimators' : [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]}
ada.grid_search(parameters = param_grid, Kfold = 5)
ada.grid_fit(X = X_train, Y = Y_train)
ada.grid_predict(X_test, Y_test)
```

```
Precision: 52.08 %
```

In [83]: Hide

```
g = plot_learning_curve(ada.grid.best_estimator_, "AdaBoost learning curves", X_t
rain, Y_train,
                        ylim = [1.01, 0.4], cv = 5,
```

```
                        train_sizes = [0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7,
0.8, 0.9, 1])
```



5.7 Gradient Boosting Classifier

In [84]:

```
gb = Class_Fit(clf = ensemble.GradientBoostingClassifier)
param_grid = {'n_estimators' : [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]}
gb.grid_search(parameters = param_grid, Kfold = 5)
gb.grid_fit(X = X_train, Y = Y_train)
gb.grid_predict(X_test, Y_test)
```

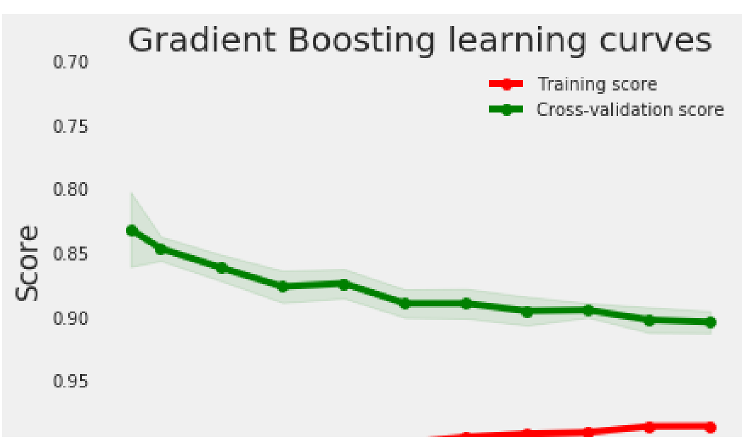
Hide

```
Precision: 89.34 %
```

In [85]:

```
g = plot_learning_curve(gb.grid.best_estimator_, "Gradient Boosting learning curv
es", X_train, Y_train,
                        ylim = [1.01, 0.7], cv = 5,
                        train_sizes = [0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7,
0.8, 0.9, 1])
```

Hide



5.8 Let's vote !

Finally, the results of the different classifiers presented in the previous sections can be combined to improve the classification model. This can be achieved by selecting the customer category as the one indicated by the majority of classifiers. To do this, I use the `VotingClassifier` method of the `sklearn` package. As a first step, I adjust the parameters of the various classifiers using the *best* parameters previously found:

In [86]:

```
rf_best  = ensemble.RandomForestClassifier(**rf.grid.best_params_)
gb_best  = ensemble.GradientBoostingClassifier(**gb.grid.best_params_)
svc_best = svm.LinearSVC(**svc.grid.best_params_)
tr_best  = tree.DecisionTreeClassifier(**tr.grid.best_params_)
knn_best = neighbors.KNeighborsClassifier(**knn.grid.best_params_)
lr_best  = linear_model.LogisticRegression(**lr.grid.best_params_)
```

Hide

Then, I define a classifier that merges the results of the various classifiers:

In [87]:

```
votingC = ensemble.VotingClassifier(estimators=[('rf', rf_best),('gb', gb_best),
                                                ('knn', knn_best)], voting='soft'
)
```

Hide

and train it:

In [88]:

```
votingC = votingC.fit(X_train, Y_train)
```

Hide

Finally, we can create a prediction for this model:

In [89]:

```
predictions = votingC.predict(X_test)
print("Precision: {:.2f} % ".format(100*metrics.accuracy_score(Y_test, prediction
s)))
```

Hide

```
Precision: 89.89 %
```

Note that when defining the `votingC` classifier, I only used a sub-sample of the whole set of classifiers defined above and only retained the *Random Forest*, the *k-Nearest Neighbors* and the *Gradient Boosting* classifiers. In practice, this choice has been done with respect to the performance of the classification carried out in the next section.

6. Testing predictions

In the previous section, a few classifiers were trained in order to categorize customers. Until that point, the whole analysis was based on the data of the first 10 months. In this section, I test the model the last two months of the dataset, that has been stored in the `set_test` dataframe:

In [90]: Hide

```
basket_price = set_test.copy(deep = True)
```

In a first step, I regroup reformattes these data according to the same procedure as used on the training set. However, I am correcting the data to take into account the difference in time between the two datasets and weights the variables **count** and **sum** to obtain an equivalence with the training set:

In [91]: Hide

```
transactions_per_user=basket_price.groupby(by=['CustomerID'])['Basket Price'].agg
(['count','min','max','mean','sum'])
for i in range(5):
    col = 'categ_{}'.format(i)
    transactions_per_user.loc[:,col] = basket_price.groupby(by=['CustomerID'])[co
l].sum() /\
                                            transactions_per_user['sum']*100

transactions_per_user.reset_index(drop = False, inplace = True)
basket_price.groupby(by=['CustomerID'])['categ_0'].sum()

#_______________________
# Correcting time range
transactions_per_user['count'] = 5 * transactions_per_user['count']
transactions_per_user['sum']   = transactions_per_user['count'] * transactions_pe
r_user['mean']

transactions_per_user.sort_values('CustomerID', ascending = True)[:5]
```

Out[91]:

	CustomerID	count	min	max	mean	sum	categ_0	categ_1	categ_2	categ_3
0	12347	10	224.82	1294.32	759.57	7595.70	5.634767	29.307371	12.696657	32.343299
1	12349	5	1757.55	1757.55	1757.55	8787.75	20.389178	36.346050	4.513101	12.245455
2	12352	5	311.73	311.73	311.73	1558.65	17.290604	32.881019	6.672441	8.735123
3	12356	5	58.35	58.35	58.35	291.75	0.000000	100.000000	0.000000	0.000000
4	12357	5	6207.67	6207.67	6207.67	31038.35	25.189000	36.560900	5.089832	14.684737

Then, I convert the dataframe into a matrix and retain only variables that define the category to which consumers belong. At this level, I recall the method of normalization that had been used on the training set:

In [92]:

```
list_cols = ['count','min','max','mean','categ_0','categ_1','categ_2','categ_3',
'categ_4']
#_____________________________________________________________
matrix_test = transactions_per_user[list_cols].as_matrix()
scaled_test_matrix = scaler.transform(matrix_test)
```

Each line in this matrix contains a consumer's buying habits. At this stage, it is a question of using these habits in order to define the category to which the consumer belongs. These categories have been established in Section 4. **At this stage, it is important to bear in mind that this step does not correspond to the classification stage itself**. Here, we prepare the test data by defining the category to which the customers belong. However, this definition uses data obtained over a period of 2 months (via the variables **count** , **min** , **max** and **sum**). The classifier defined in Section 5 uses a more restricted set of variables that will be defined from the first purchase of a client.

Here it is a question of using the available data over a period of two months and using this data to define the category to which the customers belong. Then, the classifier can be tested by comparing its predictions with these categories. In order to define the category to which the clients belong, I recall the instance of the `kmeans` method used in section 4. The `predict` method of this instance calculates the distance of the consumers from the centroids of the 11 client classes and the smallest distance will define the belonging to the different categories:

In [93]:

```
Y = kmeans.predict(scaled_test_matrix)
```

Finally, in order to prepare the execution of the classifier, it is sufficient to select the variables on which it acts:

In [94]:

```
columns = ['mean', 'categ_0', 'categ_1', 'categ_2', 'categ_3', 'categ_4' ]
X = transactions_per_user[columns]
```

It remains only to examine the predictions of the different classifiers that have been trained in section 5:

In [95]:

```
classifiers = [(svc, 'Support Vector Machine'),
                (lr, 'Logostic Regression'),
                (knn, 'k-Nearest Neighbors'),
                (tr, 'Decision Tree'),
```

```
                (rf, 'Random Forest'),
                (gb, 'Gradient Boosting')]
#______________________________
for clf, label in classifiers:
    print(30*'_', '\n{}'.format(label))
    clf.grid_predict(X, Y)
```

```
______________________________
Support Vector Machine
Precision: 71.42 %
______________________________
Logostic Regression
Precision: 71.89 %
______________________________
k-Nearest Neighbors
Precision: 66.68 %
______________________________
Decision Tree
Precision: 72.29 %
______________________________
Random Forest
Precision: 75.19 %
______________________________
Gradient Boosting
Precision: 75.26 %
```

Finally, as anticipated in Section 5.8, it is possible to improve the quality of the classifier by combining their respective predictions. At this level, I chose to mix *Random Forest*, *Gradient Boosting* and *k-Nearest Neighbors* predictions because this leads to a slight improvement in predictions:

In [96]: Hide

```
predictions = votingC.predict(X)
print("Precision: {:.2f} % ".format(100*metrics.accuracy_score(Y, predictions)))
```

```
Precision: 76.17 %
```

7. Conclusion

The work described in this notebook is based on a database providing details on purchases made on an E-commerce platform over a period of one year. Each entry in the dataset describes the purchase of a product, by a particular customer and at a given date. In total, approximately $\sim$4000 clients appear in the database. Given the available information, I decided to develop a classifier that allows to anticipate the type of purchase that a customer will make, as well as the number of visits that he will make during a year, and this from its first visit to the E-commerce site.

Did you find this Kernel useful?
Show your appreciation with an upvote

123

Comments (22)

All Comments

Sort by Hotness

Please sign in to leave a comment.

Nick Brooks • Posted on Version 59 • 6 months ago • Options

1

Don't fix what aint broke :P. Good thing I forked this. Merci Bien!

FabienDaniel Kernel Author • Posted on Version 59 • 6 months ago • Options

1

Indeed ... I just wanted to try the new editor and i just can't commit this notebook anymore ... the version you forked works fine entirely ?

Nick Brooks • Posted on Version 60 • 6 months ago • Options

2

I forked and ran it a month ago. While it also ran over 3600 seconds, and got killed, all the output actually went through..? *Note, I was already beta testing the new editor at that time.*

Here I made it public for reference: https://www.kaggle.com/nicapotato/customer-segmentation?scriptVersionId=2096497/notebook

FabienDaniel Kernel Author • Posted on Version 60 • 6 months ago • Options

1

Thanks a lot for the help and for testing the script. Sincerely, I don't know what to do and I hope people from Kaggle could have a look at it .... I notified this problem there

Bukun • Posted on Version 40 • 10 months ago • Options

1

Awesome! I plan to go through this in detail

DSEverything • Posted on Version 29 • a year ago • Options

1

Hi FabienDaniel, you are pretty thorough here. I am reading through and get lots of insights.

Just one quick notice here, you may want to cross entropy function, which uses the raw predicted probability to evaluate the classifier performance as that wouldn't depend on your cutoff threshold (so potentially avoid the issue that the accuracy values of some classifiers get really close and hard to differentiate).

FabienDaniel Kernel Author • Posted on Version 29 • a year ago • Options 0

Thanks ! I would be happy to have a look at the cross entropy function: any link where I could see an example of what you have in mind ?

DSEverything • Posted on Version 42 • 10 months ago • Options 1

Hi FabienDaniel,

unfortunately there are not many kernels or plots related to the cross entropy as this mostly applies to multi-category classification case. It is especially overlooked as there are AUC, Gini metrics to use in binary classification scenarios.

Essentially the smaller this cross entropy value is, the model is performing better. This link below is a simplified version of explanations, you should have known I believe.
https://pmirla.github.io/2016/10/09/Basics-crossEntropy.html

FabienDaniel Kernel Author • Posted on Version 42 • 10 months ago • Options 0

Many thanks for the link, I'll have a look at it !

GSD • Posted on Version 46 • 10 months ago • Options 2

Excellent coding and visualisation. Pardon my ignorance for this question .I have been creating kernals using R markdown notebooks and I was under the assumption that with python you cannot create excellent reports like this.I would appreciate if you could explain how to go about creating such wonderful reports in notebooks..

FabienDaniel Kernel Author • Posted on Version 46 • 10 months ago • Options 1

Thanks a lot for the compliment :)

In fact, I also believe that R allows a better rendering of the work, mainly because of the graphical facilities. For example I recently looked at this notebook where many plots are embedded in a single graph: each plot is made accessible through a given tab. The text that appear below each plot is also tab dependent. This makes everything much more concise and easier to read and as far as I know, we can't do anything similar in Python.

Well, in Kaggle's kernels, they recently added the possibility to hide cells in Python which allows to make much nice looking notebooks.

GSD • Posted on Version 47 • 10 months ago • Options 0

Thats Awesome ...Thank you ...and did you type the introduction part and index by yourself ?I thought there would be markdown kind of features where you could specify everything prior and knitr will take care of everything else...

Muhammed Buyu... • Posted on Version 32 • 10 months ago • Options 0

GREAT

[Deleted User] • Posted on Version 34 • 10 months ago • Options 0

Wow. Thanks for the detailed explanations.

Michael • Posted on Version 55 • 9 months ago • Options 0

Thanks for this comprehensive analysis. Planning to do the rfm und clv analysis on these data. Genuine transaction data are extremely hard to find, indeed.

Michael • Posted on Version 55 • 9 months ago • Options 0

Goran • Posted on Version 56 • 9 months ago • Options 0

Great!

Charalampos • Posted on Version 62 • 5 months ago • Options 0

Hello, fist of all thank you for this perfect report and then i have a question.. In both 5 and 6 sections, why do we have to use the classifiers and train the model as a supervised one and dont use the kmeans.pedict command to classify the new entries in the store??

AlOtaibi,Intisar • Posted on Latest Version • 5 months ago • Options 0

great work

Pedro Hojas • Posted on Latest Version • 4 months ago • Options 0

Great example. Thanks for all the details in your work.

DGarcia • Posted on Latest Version • 4 months ago • Options

0

Amazing example! Thank you very much! Just wanted to know if it would be possible to see the predicted class against the real one.

Sergi Fernandez • Posted on Latest Version • 4 months ago • Options

0

Excellent job. It's really useful to analyze the data.

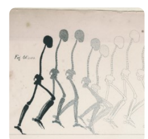
I'm working with R and, as a beginner, I would be interested in knowing the equivalence in R or how to code in R the codes starting in [11] to detect and remove the cancellation orders with the counterparts. Could you tell me a bit, please? Thank you.

Similar Kernels

Sentiment Analysis And Clustering

FIFA 18 Visualisation | Clustering | ML

Predicting R Vs. Python



Machine Learning Tutorial For Beginners

Start Here: A Gentle Introduction

Our Team Terms Privacy Contact/Support