Christophe coudé, 2017

Parcours Data Scientist, projet n°5, Notebook de data cleaning et d'exploration

Objectifs :
L'entreprise britannique Datazon, leader dans la vente en ligne de toute sorte d'objets, cherche à mieux comprendre les comportements de ses clients pour augmenter la fréquence d'achat et la valeur du panier moyen. Pour cela, il est nécessaire de comprendre les différents types d'utilisateurs grâce à leur comportement dans la durée, afin de détecter les plus susceptibles de passer à l'achat.

Les données sont téléchargeables à partir du lien suivant:
https://archive.ics.uci.edu/ml/machine-learning-databases/00352/Online%20Retail.xlsx
Ces données contiennent toutes les transactions ayant eu lieu entre le 01/12/2010 et le 09/12/2011 pour une société basée en Grande-Bretagne et enregistrée comme commerce de détail en ligne.

Mission :

- Effectuer différentes modifications des séries temporelles fournies afin de trouver les features qui permettent de détecter les catégories dignes d'intérêt.
- Utiliser ces catégories pour classer automatiquement les utilisateurs le plus vite possible après leur premier achat.

Remarque préalable : Le fichier de données utilisé n'est pas le fichier téléchargé Online Retail.xlsx, mais sa version CSV OnlineRetail.csv. Il y a donc exactement les mêmes données, c'est juste le format du fichier qui change.

1- Description des données

Importation des données et des packages

In [186]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings('ignore')
warnings.filterwarnings("ignore", category=DeprecationWarning)
import seaborn as sns
%matplotlib inline
import math
import time
import datetime as dt
```

In [187]:
```
parser = lambda date: pd.datetime.strptime(date, '%d/%m/%Y %H:%M')
dfOriginel = pd.read_csv('OnlineRetail.csv', sep=';', decimal='.', parse_dates=[4], date_parser=parser, converters={'Customer ID': lambda x: str(x)})
```

Description des variables

Variable	Description de la variable	Type de la variable
InvoiceNo	Numéro de la facture pour chaque transaction. Si le code commence par 'c', il s'agit d'une annulation.	Caractères
StockCode	Code de l'article.	Caractères
Description	Nom du produit.	Caractères
Quantity	Quantité de produit acheté par transaction.	Numérique
InvoiceDate	Date et heure de la facture, de la génération de la transaction.	Datetime
UnitPrice	Prix unitaire en livre sterling.	Numérique
CustomerID	Identifiant du client.	Numérique
Country	Pays où réside le client.	Caractères

Volume des données

In [188]:
```
print("Il y a {0} enregistrements et {1} variables".format(dfOriginel.shape[0], dfOriginel.shape[1]))
```

```
Il y a 541909 enregistrements et 8 variables
```

2- Data Cleaning

Suppression des enregistrements non utiles à l'objectif

Puisque l'objectif est d'étudier les habitudes des clients (CustomerID), on supprime les enregistrements sans CustomerID

In [189]:
```
dfCleaned =  dfOriginel[dfOriginel['CustomerID'] <> '']
dfCleaned.reset_index(drop=True, inplace=True)
print("Il reste {0} enregistrements et {1} variables".format(dfCleaned.shape[0], dfCleaned.shape[1]))
```

```
Il reste 406829 enregistrements et 8 variables
```

Suppression des enregistrements qui correspondent à des ajustements (Manual/M, AMAZON FEE/AMAZONFEE, Adjust bad debt/B, POSTAGE/POST, DOTCOM POSTAGE/DOT, Discount/D, CRUK Commission/CRUK, Bank Charges/BANK CHARGES, SAMPLES/S, CARRIAGE/C2) puisqu'on ne sait pas à quels articles ils correspondent.

In [190]:
```
dfCleaned = dfCleaned[dfCleaned["StockCode"] != "M"]
dfCleaned = dfCleaned[dfCleaned["StockCode"] != "AMAZONFEE"]
dfCleaned = dfCleaned[dfCleaned["StockCode"] != "B"]
dfCleaned = dfCleaned[dfCleaned["StockCode"] != "POST"]
dfCleaned = dfCleaned[dfCleaned["StockCode"] != "DOT"]
dfCleaned = dfCleaned[dfCleaned["StockCode"] != "D"]
dfCleaned = dfCleaned[dfCleaned["StockCode"] != "CRUK"]
dfCleaned = dfCleaned[dfCleaned["StockCode"] != "BANK CHARGES"]
dfCleaned = dfCleaned[dfCleaned["StockCode"] != "S"]
dfCleaned = dfCleaned[dfCleaned["StockCode"] != "C2"]
```

Cas des annulations de commandes

- on ne peut se contenter de supprimer les enregistrements qui correspondent à des annulations de commandes puisque ce serait garder les commandes finalement annulées
- on doit donc rechercher les commandes qui correspondent aux commandes annulées, et lorsqu'on les trouve, les supprimer afin de supprimer de l'historique toutes les commandes qui n'en sont en réalité pas puisqu'elles ont été annulées
- mais quand on ne trouve pas les commandes initiales correspondant aux commandes annulées, par exemple parce qu'elles sont antérieures à la date de début des enregistrements, on ne supprime de fait que les annulations

Note : l'application de la méthode deletePayback dure entre 20 et 30 minutes...

In [191]:
```
dfCleaned["concat"] =  dfCleaned["Quantity"].map(str) + dfCleaned["StockCode"].map(str) +dfCleaned["UnitPrice"].map(str) + df
Cleaned["CustomerID"].map(str) + dfCleaned["Country"].map(str)
def deletePayback(row):
    if row['Quantity']<0:
        dfCleaned.drop(row.name, inplace=True)
        achats = dfCleaned[dfCleaned['concat'] == row['concat'][1:]]
        if achats.shape[0] > 0:
            premierAchat = achats.iloc[0]
            dfCleaned.drop(premierAchat.name, inplace=True)
dfCleaned.apply(deletePayback, axis=1)
dfCleaned.drop("concat", inplace=True, axis=1)
dfCleaned.reset_index(drop=True, inplace=True)
```

In [192]:
```
print("Il reste désormais {0} enregistrements et {1} variables".format(dfCleaned.shape[0], dfCleaned.shape[1]))
```

```
Il reste désormais 393368 enregistrements et 8 variables
```

Changement de type de la variable UnitPrice

La variable UnitPrice est normalement de type numérique, mais sa séparation des décimales est une virgule, on la transforme donc en un point.

In [193]:
```
dfCleaned["UnitPrice"] = dfCleaned["UnitPrice"].str.replace(",", ".").astype(float)
```

3- Feature engineering

Pour les besoins de la future classification par segmentation RFM, on crée la variable Amount correspondant au montant total d'une transaction pour un article

In [194]:
```
dfCleaned['Amount'] = dfCleaned.Quantity * dfCleaned.UnitPrice
print(dfCleaned.head(1))
```

```
  InvoiceNo StockCode                         Description  Quantity  \
0    536365    85123A  WHITE HANGING HEART T-LIGHT HOLDER         6

          InvoiceDate  UnitPrice CustomerID         Country  Amount
0 2010-12-01 08:26:00       2.55      17850  United Kingdom    15.3
```

In [195]:
```
dfCleaned['year'] = pd.DatetimeIndex(dfCleaned['InvoiceDate']).year
dfCleaned['month'] = pd.DatetimeIndex(dfCleaned['InvoiceDate']).month
dfCleaned['day'] = pd.DatetimeIndex(dfCleaned['InvoiceDate']).day
dfCleaned['dayofweek'] = pd.DatetimeIndex(dfCleaned['InvoiceDate']).dayofweek
dfCleaned['hour'] = pd.DatetimeIndex(dfCleaned['InvoiceDate']).hour
```

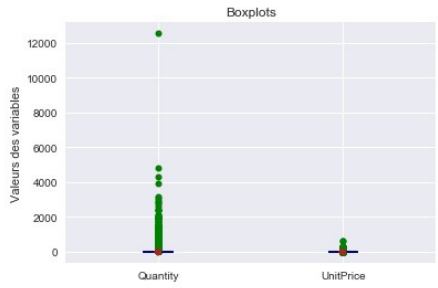
4- Analyse univariée

Utilisation des boxplots afin de repérer d'éventuelles asymétries aberrantes

In [196]:
```
def dfBoxPlot(_df, _title):
    """Dessine le boxplot
    Arguments:
    _df -- dataframe contenant les données initiales
    _title -- titre du boxplot
    """
    plt.figure(figsize=(12,6))
    colors=dict(boxes='DarkGreen', whiskers='DarkOrange', medians='DarkBlue', caps='Gray')
    boxprops = dict(linewidth=2)
    medianprops = dict(linewidth=2)
    meanpointprops = dict(marker='D', markeredgecolor='black', markerfacecolor='firebrick')
    flierprops = dict(marker='o', linestyle='none', markerfacecolor='green', markersize=6)
    ax = _df.plot.box(showmeans=True, showfliers=True, flierprops=flierprops, boxprops=boxprops, medianprops=medianprops, col
or=colors, meanprops=meanpointprops)
    ax.set_ylabel('Valeurs des variables')
    ax.set_title(_title)
```

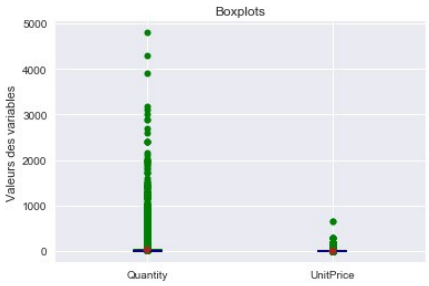
In [197]:
```
dfBoxPlot(dfCleaned[['Quantity', 'UnitPrice']], "Boxplots")
```

```
<matplotlib.figure.Figure at 0x35f85470>
```



In [198]:
```
dfCleaned2 = dfCleaned[dfCleaned['Quantity']<10000]
dfBoxPlot(dfCleaned2[['Quantity', 'UnitPrice']], "Boxplots")
```

```
<matplotlib.figure.Figure at 0x35f89be0>
```



Il ne semble pas y avoir de valeurs aberrantes en ce qui concerne les quantités.

In [199]:
```
# données manquantes
dfNanCleaned = 100*(1-dfCleaned.count()/dfCleaned.shape[0])
print(dfNanCleaned.sort_values())
```

```
InvoiceNo      0.0
StockCode      0.0
Description    0.0
Quantity       0.0
InvoiceDate    0.0
UnitPrice      0.0
CustomerID     0.0
Country        0.0
Amount         0.0
year           0.0
month          0.0
day            0.0
dayofweek      0.0
hour           0.0
dtype: float64
```

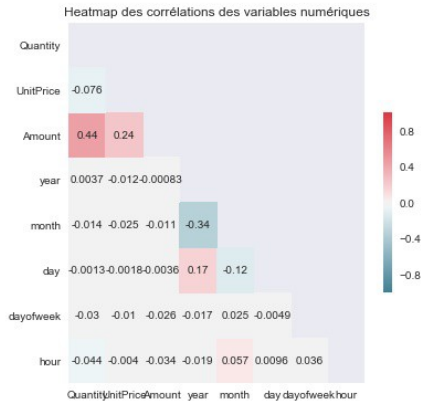
Le dataset ne comporte plus aucune donnée manquante.

5- Analyse multivariée

In [200]:
```
def heatmap(_df, _methode, _title):
    """Affichage de la heatmap de la matrice de corrélation des variables continues
    Arguments:
    _df -- dataframe contenant les variables
    _methode -- méthode statistique pour la corrélation, Pearson, Spearman ou Kendall
    _title -- titre de la heatmap
    """

    # suppression des figures
    plt.clf()
    # on calcule la matrice de corrélation pour le dataframe _df passé en paramètre
    corr = _df.corr(method=_methode)
    # masque d'affichage de la heatmap
    mask = np.zeros_like(corr, dtype=np.bool)
    mask[np.triu_indices_from(mask)] = True
    f, ax = plt.subplots(figsize=(6,6))
    cmap = sns.diverging_palette(220, 10, as_cmap=True)

    # traçage de la heatmap
    sns.heatmap(corr, cmap=cmap, cbar_kws={"shrink": .5}, mask=mask, annot=True)
    plt.title(_title)
    plt.show()
```

In [201]:
```
heatmap(dfCleaned, "pearson", u"Heatmap des corrélations des variables numériques")
```

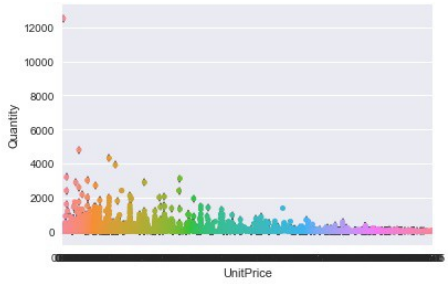
```
<matplotlib.figure.Figure at 0x2c1e36a0>
```



Mis à part la corrélation normale entre Quantity et Amount, il n'y a pas de corrélation particulière entre la quantité d'articles achetés et le moment de l'achat.

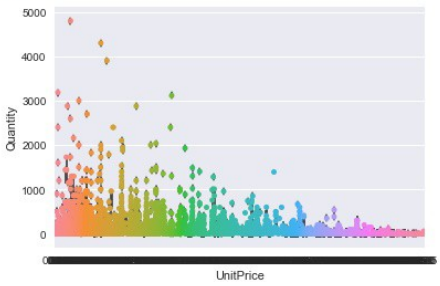
Relation entre le prix unitaire des articles et la quantité:

In [202]:
```
ax=sns.boxplot(x="UnitPrice",y="Quantity",data=dfCleaned)
ax=sns.stripplot(x="UnitPrice",y="Quantity",data=dfCleaned,jitter=True)
```



Pour mieux voir la répartition des données, je supprime la donnée extrême de la variable quantity:

In [203]:
```
dfCleanedSansTop = dfCleaned[dfCleaned['Quantity']<10000]
ax=sns.boxplot(x="UnitPrice",y="Quantity",data=dfCleanedSansTop)
ax=sns.stripplot(x="UnitPrice",y="Quantity",data=dfCleanedSansTop,jitter=True)
```



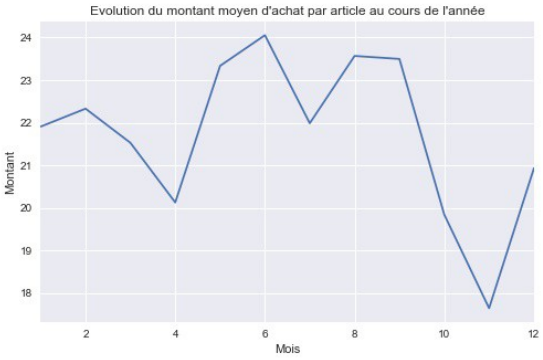
Tendance logique à la diminution du nombre d'articles achetés quand leur prix augmente, mais aucune relation linéaire.

On peut essayer de voir si le montant des achats est corrélé à certaines périodes de l'année ou de la semaine:

Evolution du montant moyen d'achat par article au cours de l'année

In [204]:
```
f, ax = plt.subplots(figsize = (8, 5))
dfCleaned.groupby(['month']).Amount.mean().plot()
ax.set_title(u"Evolution du montant moyen d'achat par article au cours de l'année")
ax.set_xlabel('Mois')
ax.set_ylabel('Montant')
```

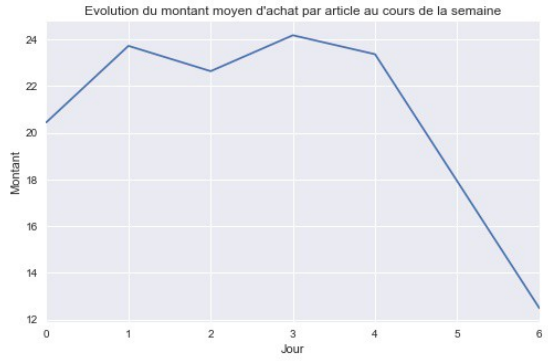
Out[204]:
```
<matplotlib.text.Text at 0x2e43dd30>
```



Evolution du montant moyen d'achat par article au cours de la semaine

In [205]:
```
f, ax = plt.subplots(figsize = (8, 5))
dfCleaned.groupby(['dayofweek']).Amount.mean().plot()
ax.set_title(u"Evolution du montant moyen d'achat par article au cours de la semaine")
ax.set_xlabel('Jour')
ax.set_ylabel('Montant')
```

Out[205]: <matplotlib.text.Text at 0x24bab668>



!!! il n'y a eu aucun achat le jeudi (jour 5), le graphe devrait tomber à 0

Evolution du nombre d'achats d'articles au cours de la semaine

In [206]:
```
f, ax = plt.subplots(figsize = (8, 5))
dfCleaned.groupby(['dayofweek']).Amount.count().plot()
ax.set_title(u"Evolution du nombre d'achats par article au cours de la semaine")
ax.set_xlabel('Jour')
ax.set_ylabel("Nombre d'achat")
```

Out[206]: <matplotlib.text.Text at 0x31867b38>

Les graphes de l'évolution des achats, au niveau de l'article, ne donne pas de résultats intéressants.

Parcours Data Scientist, projet n°5, Notebook des modèles

Objectifs :
L'entreprise britannique Datazon, leader dans la vente en ligne de toute sorte d'objets, cherche à mieux comprendre les comportements de ses clients pour augmenter la fréquence d'achat et la valeur du panier moyen. Pour cela, il est nécessaire de comprendre les différents types d'utilisateurs grâce à leur comportement dans la durée, afin de détecter les plus susceptibles de passer à l'achat.

Les données sont téléchargeables à partir du lien suivant:
https://archive.ics.uci.edu/ml/machine-learning-databases/00352/Online%20Retail.xlsx
Ces données contiennent toutes les transactions ayant eu lieu entre le 01/12/2010 et le 09/12/2011 pour une société basée en Grande-Bretagne et enregistrée comme commerce de détail en ligne.

Mission :

- Effectuer différentes modifications des séries temporelles fournies afin de trouver les features qui permettent de détecter les catégories dignes d'intérêt.
- Utiliser ces catégories pour classer automatiquement les utilisateurs le plus vite possible après leur premier achat.

Remarque préalable : Le fichier de données utilisé n'est pas le fichier téléchargé Online Retail.xlsx, mais sa version CSV OnlineRetail.csv. Il y a donc exactement les mêmes données, c'est juste le format du fichier qui change.

In [324]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings('ignore')
warnings.filterwarnings("ignore", category=DeprecationWarning)
import seaborn as sns
%matplotlib inline
import math
import time
import datetime as dt
from sklearn.cluster import KMeans
from sklearn.decomposition import PCA
from ggplot import *
from lifetimes import BetaGeoFitter
from lifetimes.plotting import plot_frequency_recency_matrix
from lifetimes.plotting import plot_probability_alive_matrix
from lifetimes.plotting import plot_period_transactions
from lifetimes.plotting import plot_history_alive
import scipy.sparse as sparse
import random
import implicit
from sklearn import metrics
from sklearn.preprocessing import MinMaxScaler
```

In [325]:
```
parser = lambda date: pd.datetime.strptime(date, '%d/%m/%Y %H:%M')
dfOriginel = pd.read_csv('OnlineRetail.csv', sep=';', decimal='.', parse_dates=[4], date_parser=parser, converters={'Customer
ID': lambda x: str(x)})
```

Reprise de la préparation des données

In [326]:
```
dfCleaned =  dfOriginel[dfOriginel['CustomerID'] <> '']
dfCleaned.reset_index(drop=True, inplace=True)
dfCleaned = dfCleaned[dfCleaned["StockCode"] != "M"]
dfCleaned = dfCleaned[dfCleaned["StockCode"] != "AMAZONFEE"]
dfCleaned = dfCleaned[dfCleaned["StockCode"] != "B"]
dfCleaned = dfCleaned[dfCleaned["StockCode"] != "POST"]
dfCleaned = dfCleaned[dfCleaned["StockCode"] != "DOT"]
dfCleaned = dfCleaned[dfCleaned["StockCode"] != "D"]
dfCleaned = dfCleaned[dfCleaned["StockCode"] != "CRUK"]
dfCleaned = dfCleaned[dfCleaned["StockCode"] != "BANK CHARGES"]
dfCleaned = dfCleaned[dfCleaned["StockCode"] != "S"]
dfCleaned = dfCleaned[dfCleaned["StockCode"] != "C2"]
dfCleaned["concat"] =  dfCleaned["Quantity"].map(str) + dfCleaned["StockCode"].map(str) +dfCleaned["UnitPrice"].map(str) + df
Cleaned["CustomerID"].map(str) + dfCleaned["Country"].map(str)
def deletePayback(row):
    if row['Quantity']<0:
        dfCleaned.drop(row.name, inplace=True)
        achats = dfCleaned[dfCleaned['concat'] == row['concat'][1:]]
        if achats.shape[0] > 0:
            premierAchat = achats.iloc[0]
            dfCleaned.drop(premierAchat.name, inplace=True)
dfCleaned.apply(deletePayback, axis=1)
dfCleaned.drop("concat", inplace=True, axis=1)
dfCleaned.reset_index(drop=True, inplace=True)
dfCleaned["UnitPrice"] = dfCleaned["UnitPrice"].str.replace(",", ".").astype(float)
dfCleaned['Amount'] = dfCleaned.Quantity * dfCleaned.UnitPrice
dfCleaned['year'] = pd.DatetimeIndex(dfCleaned['InvoiceDate']).year
dfCleaned['month'] = pd.DatetimeIndex(dfCleaned['InvoiceDate']).month
dfCleaned['day'] = pd.DatetimeIndex(dfCleaned['InvoiceDate']).day
dfCleaned['dayofweek'] = pd.DatetimeIndex(dfCleaned['InvoiceDate']).dayofweek
dfCleaned['hour'] = pd.DatetimeIndex(dfCleaned['InvoiceDate']).hour
```

1- Segmentation RFM et KMeans clustering

La segmentation RFM (Récence / Fréquence / Montant) est un concept marketing qui permet d'évaluer le potentiel d'un client afin de discerner les actions qui doivent être envisagées selon la catégorie à laquelle il appartient.
Elle porte sur 3 indicateurs:

- Récence : laps de temps écoulé depuis la dernière visite d'un client ou la dernière commande d'un client.
- Fréquence : nombre de fois où le client est venu sur la période concernée, ou a passé commande. Je choisis ici de ne pas comptabiliser le nombre d'articles achetés, mais le nombre de factures puisque le nombre d'article est lié, même si de façon indirecte, au montant moyen d'une facture.
- Montant : montant moyen d'un "panier", moyenne de la somme des articles par facture.

Segmentation RFM sur la totalité des données

Les données s'étalent sur la période du 01/12/2010 inclus au 09/12/2011 inclus.

La date de référence permettant le calcul de la récence est donc celle du jour d'après la dernière transaction, soit le 10/12/2011.

In [327]:
```
dateRef = dt.datetime(2011,12,10)
```

La récence d'un client est le minimum de l'écart (en nombre de jours) entre la date de référence et les dates d'achat des articles

In [328]:
```
# mise au bon format de la date de la facture correspondant à l'article acheté
dfCleaned['InvoiceDateTemp'] = pd.to_datetime(dfCleaned['InvoiceDate'], format='%d%m%Y')
# création de la variable InvoiceDateDayDiff représentant la différence entre la date de la facture et la date de référence
(en jours)
dfCleaned['InvoiceDateDayDiff'] = dfCleaned['InvoiceDateTemp'].apply(lambda x: (dateRef - x).days)
```

Les trois variables sont donc calculées ainsi pour chaque client (CustomerID):

- récence : minimum de la variable InvoiceDateDayDiff
- fréquence : nombre de factures
- montant (moyen): somme de tous les articles achetés par un client, donc de toutes les factures, divisée par le nombre de factures

In [329]:
```
rfmTable = dfCleaned.groupby('CustomerID').agg({'InvoiceDateDayDiff': lambda x: x.min(), # Recency
                                                'InvoiceNo': lambda x: len(x.unique()),      # Frequency
                                                'Amount': lambda x: x.sum()}) # Monetary Value
rfmTable.rename(columns={'InvoiceDateDayDiff': 'recency', 'InvoiceNo': 'frequency', 'Amount': 'amount'}, inplace=True)
rfmTable['amount'] = rfmTable['amount'] / rfmTable['frequency']
rfmTable.head()
```

Out[329]:

	frequency	recency	amount
CustomerID			
12347	7	2	615.714286
12348	4	75	359.310000
12349	1	18	1457.550000
12350	1	310	294.400000
12352	7	36	180.772857

On obtient ainsi pour chaque client, la valeur des trois variables fréquence / récence / montant.

Mais ceci n'est pas suffisant, il est nécessaire de catégoriser le comportement d'un client en découpant les variables en classes.
Or il y a de nombreuses façons de créer les catégories. On peut trouver par exemple deux familles de procédés, celle utilisant les données du jeu de donnée, et celle ne les utilisant pas.
Par exemple, si on ne veut pas rendre tributaire le découpage des aléas du jeu de donnée, on peut choisir les frontières de façon arbitraire, par exemple en découpant la récence en tranches temporelles telles que:

- < 1 semaine
- entre 1 semaine et 6 mois
- entre 6 mois et 12 mois
- entre ...
- jamais

Cela a comme inconvénient de potentiellement retrouver tous les clients dans une seule catégorie, et de rendre difficile le choix des frontières, par exemple pour le montant.

Dans la seconde famille de procédés, on peut par exemple utiliser la partition des quantiles.
Mais cela a comme inconvénient de faire dépendre la partition du jeu de donnée. On ne peut alors guère comparer les catégories qui en résultent, aux catégories d'un autre jeu de donnée, ou d'une autre période comme nous le verrons plus loin.

En fait, tout dépend de si on veut que la segmentation représente une catégorisation "absolue", ou une catégorisation relative à un ensemble donné de clients.

On pourrait utiliser un score du genre : Potentiel Score = Récence *pr* + *Fréquence* pf + Montant * pm où pr, pf et pm sont des pondérations permettant d'impliquer l'importance relative des différentes variables, ce qui peut être utile dans certains cas où on connaît les catégories d'articles. Cependant, dans notre cas, les articles ne nécessitent pas une pondération particulière.

Dans la première illustration de la segmentation RFM, comme il n'y aura pas de comparaison entre périodes permettant de suivre l'évolution d'un client, son passage d'une catégorie à une autre, on peut utiliser les quartiles (permettant de limiter le nombre de catégories) afin de séparer les différentes catégories.

In [330]:
```
quantiles = rfmTable.quantile(q=[0.25,0.5,0.75])
print(quantiles)
```

```
      frequency  recency      amount
0.25        1.0     17.0  176.877500
0.50        2.0     50.0  289.240000
0.75        5.0    143.0  425.006667
```

Différentiation des catégories :
On applique un nombre de points différents suivant la position du client dans les différentes tranches des variables.

- pour la variable récence, on donne le maximum de points aux clients dont la récence est la plus basse, donc ceux qui sont venus le plus récemment, car, on estime, dans le contexte d'achats "d'articles de première nécessité", que plus un client n'est pas venu depuis longtemps, moins il est probable qu'il ne revienne. Un client présent dans la première tranche, donc dans les 25% les plus récents des clients, prend donc le plus grand nombre de points, 4.
- pour les variables fréquence et montant, c'est l'inverse, des clients venant fréquemment ou achetant beaucoup de choses ont plus de chance de revenir, on met alors plus de points aux valeurs élevées de ces deux variables, ou, puisqu'ici on parle en terme de quartiles, à la tranche correspondant aux 25% des clients venant le plus fréquemment ou achetant le plus de choses.

In [331]:
```
def RCateg(_x,_p,_d):
    """Méthode qui détermine la tranche (entre deux quartiles) où se trouve le client sur sa récence
    Paramètres:
    _x -- valeur de la récence
    _d -- tableau des quartiles
    Retour:
    tranche (quartile) où se trouve le client sur sa récence
    """
    if _x < _d[_p][0.25]:
        return 4
    elif _x < _d[_p][0.5]:
        return 3
    elif _x < _d[_p][0.75]:
        return 2
    else:
        return 1
def FMCateg(_x,_p,_d):
    """Méthode qui détermine la tranche (entre deux quartiles) où se trouve le client sur sa fréquence ou son montant
    Paramètres:
    _x -- valeur de la fréquence ou du montant
    _d -- tableau des quartiles
    Retour:
    tranche (quartile) où se trouve le client sur sa fréquence ou son montant
    """
    if _x <= _d[_p][0.25]:
        return 1
    elif _x <= _d[_p][0.5]:
        return 2
    elif _x <= _d[_p][0.75]:
        return 3
    else:
        return 4
```

Calcul des valeurs de la variable RFMClass donnant le "score RFM" consituté de la concaténation des points de chaque variable

In [332]:
```
rfmSegmentation = rfmTable
rfmSegmentation['R_Quartile'] = rfmSegmentation['recency'].apply(RCateg, args=('recency',quantiles))
rfmSegmentation['F_Quartile'] = rfmSegmentation['frequency'].apply(FMCateg, args=('frequency',quantiles))
rfmSegmentation['M_Quartile'] = rfmSegmentation['amount'].apply(FMCateg, args=('amount',quantiles))
rfmSegmentation['RFMClass'] = rfmSegmentation.R_Quartile.map(str) + rfmSegmentation.F_Quartile.map(str) + rfmSegmentation.M_Q
uartile.map(str)
rfmSegmentation.head()
```

Out[332]:

	frequency	recency	amount	R_Quartile	F_Quartile	M_Quartile	RFMClass
CustomerID							
12347	7	2	615.714286	4	4	4	444
12348	4	75	359.310000	2	3	3	233
12349	1	18	1457.550000	3	1	4	314
12350	1	310	294.400000	1	1	3	113
12352	7	36	180.772857	3	4	2	342

On peut supprimer les variables R_Quartile, F_Quartile, M_Quartile qui ne servent plus.

In [333]:
```
rfmSegmentation.drop(['R_Quartile', 'F_Quartile', 'M_Quartile'], axis=1, inplace=True)
rfmSegmentation.head()
```

Out[333]:

	frequency	recency	amount	RFMClass
CustomerID				
12347	7	2	615.714286	444
12348	4	75	359.310000	233
12349	1	18	1457.550000	314
12350	1	310	294.400000	113
12352	7	36	180.772857	342

Comme pour chaque variable, des points allant de 1 à 4 sont atribués, on obtient donc 4x4x4 = 64 catégories possibles, allant de 111 à 444.
Ainsi, un client appartenant à la catégorie 111 est considéré comment n'étant pas venu depuis très longtemps, peu fréquemment et n'ayant presque rien acheté par rapport aux autres clients. (Puisqu'il s'agit de quantiles, la comparaison ne se fait pas dans l'absolu, mais relativement aux autres clients.)
Le client de la catégorie 444 est ainsi un "très bon client" puisqu'il vient, relativement aux autres clients, de façon fréquente, achète beaucoup, et est venu récemment.

Clustering

Le problème, est qu'on dispose de 64 catégories, il peut alors être fastidieux et délicat de leur attribuer de façon personnelle une stratégie commerciale, surtout que les différences entre certaines catégories peuvent être minimes (par exemple entre la 443 et la 444...)
Il peut alors être utile de tenter de regrouper ces 64 catégories en un ensemble moins élevé de catégories qu'on pourra plus facilement traiter.
Pour cela, on peut essayer un Kmeans sur les scores RFM des articles.

On peut regrouper le code, déjà utilisé pour expliciter chaque partie, dans une fonction.

In [334]:
```
def calc(_dateRef, _x):
    """Méthode qui calcule la différence entre la date de facture d'un client et la date de référence
    Paramètres:
    _dateRef -- date de référence
    _x -- la date de la facture du client
    Retour:
    différence entre la date de facture d'un client et la date de référence
    """
    y = (_dateRef - _x).days
    if y >= 0:
        return y
    else:
        return 10000000000
def getRFMSegmentationCateg(_df,_dateRef):
    """Méthode qui calcule les segments RFM ('111', '141', '342' etc.) pour chaque client
    Paramètres:
    _df -- le dataframe contenant les données nettoyées
    _dateRef -- la date de référence pour la segmentation RFM
    Retour:
    rfmTable2 -- la table RFM avec les segments RFM ('111', '141', '342' etc.)
    """
    df2 = _df.copy()
    # différence de jours entre la date d'une facture et la date de référence
    df2['InvoiceDateDayDiff'] = df2['InvoiceDate'].apply(lambda x: calc(_dateRef, x))
    # calcul des valeurs des variables recency, frequency et amount
    rfmTable2 = df2.groupby('CustomerID').agg({'InvoiceDateDayDiff': lambda x: x.min(), # Recency
                                               'InvoiceNo': lambda x: len(x.unique()),      # Frequency
                                               'Amount': lambda x: x.sum()}) # Amount
    rfmTable2.rename(columns={'InvoiceDateDayDiff': 'recency', 'InvoiceNo': 'frequency', 'Amount': 'amount'}, inplace=True)
    # on divise pour avoir les montant par facture et non par article
    rfmTable2['amount'] = rfmTable2['amount'] / rfmTable2['frequency']
    # les quartiles
    quantiles2 = rfmTable2.quantile(q=[0.25,0.5,0.75])
    # positionnement pour chaque variable R, F, M
    rfmTable2['R_Quartile'] = rfmTable2['recency'].apply(RCateg, args=('recency',quantiles2))
    rfmTable2['F_Quartile'] = rfmTable2['frequency'].apply(FMCateg, args=('frequency',quantiles2))
    rfmTable2['M_Quartile'] = rfmTable2['amount'].apply(FMCateg, args=('amount',quantiles2))
    # classe RFM, concaténation des valeurs pour chaque variable
    rfmTable2['RFMClass'] = rfmTable2.R_Quartile.map(str) + rfmTable2.F_Quartile.map(str) + rfmTable2.M_Quartile.map(str)
    # suppression des variables temporaires
    rfmTable2.drop(['R_Quartile', 'F_Quartile', 'M_Quartile'], axis=1, inplace=True)
    return rfmTable2
```

In [335]:
```
# date de référence du 10/12/2011, jour d'après la dernière transaction de la période
dateRef = dt.datetime(2011, 12, 10)
RFMSegmentationCateg10122011 = getRFMSegmentationCateg(dfCleaned, dateRef)
RFMSegmentationCateg10122011.head()
```

Out[335]:

	frequency	recency	amount	RFMClass
CustomerID				
12347	7	2	615.714286	444
12348	4	75	359.310000	233
12349	1	18	1457.550000	314
12350	1	310	294.400000	113
12352	7	36	180.772857	342

- On voit que le client 12347, est un client parfait (du moins, relativement aux autres clients du dataset), puisque'il se situe pour les 3 variables, dans les 25% des meilleurs clients (444).
- Par contre, le client 12350, de catégorie 113, est un client qui a fait une fois un moyen gros achat il y a longtemps. Il est donc bien représenté pour la variable amount (3 points), mais pas pour les deux autres variables (1 point chaque).

Remarque : on observe ici l'effet d'utiliser les quartiles, les deux clients 12347 et 12349 sont dans la même catégorie concernant l'amount, pourtant la différence de panier moyen est énorme entre les deux (615 et 1457). Avec une partition non basée sur la spécificité d'un dataset, ils auraient certainement été dans des catégories assez éloignées.

KMeans avec 8 clusters

Pourquoi 8 clusters ?

J'ai utilisé l'article suivant: http://iaiest.com/dl/journals/5-%20IAJ%20of%20Accounting%20and%20Financial%20Management/v3-i6-jun2016/paper3.pdf pour définir 8 catégories "marketing" dont voici la description: (cf. p. 26)

Category of customer	Description
Best Customers	It refers to those who bought recently, with a high buying frequency in a definite period of time, with high monetary value in each transaction.
Valuable Customers	It refers to those who bought recently, with a low buying frequency in a definite period of time, but with high monetary value in each transaction.
Shopper Customers	It refers to those who bought recently, with a high buying frequency in a definite period of time, but with low monetary value in each transaction.
First Time Customers	It refers to those who bought recently, but with a low buying frequency in a definite period of time and with low monetary value in each transaction.
Churn Customers	It refers to those with a high buying frequency in a definite period of time and high monetary value in each transaction but they haven't bought recently for some specific reasons.
Frequent Customers	It refers to those who haven't bought recently and their monetary value of their transaction is not that significant, but they buy frequently in a definite period of time.
Spenders Customers	It refers to those who haven't bought recently and they don't buy frequently in a definite period of time but the monetary value of their transactions is very significant.
Uncertain Customers	It refers to those who haven't bought recently, with a low buying frequency in a definite period of time and with low monetary value in each transaction. This segment is the most trivial customer cluster with the lowest buying traits.

Ils définissent ainsi une hiérarchie "marketing" des pattern RFM. (Le numéro des clusters est indicatif.)
La définition de la différence entre ↑ et ↓ est discutable. Dans l'article, il utilise le positionnement de la moyenne d'une variable (par ex. F↑/F↓) pour un cluster relativement à la moyenne sur la totalité du dataset.
Si par exemple, on avait une moyenne générale de 3 pour la fréquence de la totalité des clients du dataset, et pour un cluster, une moyenne de 4 pour les clients de ce cluster, on adjoindrait alors le pattern F↑ pour la variable F de ce cluster.
Toutefois, d'autres choix pourraient être faits que la moyenne. Pour le cas présent, je choisis d'ailleurs non pas la moyenne, mais la médiane afin de rester dans l'optique de comparaison entre les clients du dataset. Plus loin, lors de l'étude de l'évolution des catégories, j'utiliserai les deux fonctions pour montrer la dépendance des catégoriesà ces fonctions.

On a donc 2 possibilités (↑ ou ↓) pour les 3 variables R, F et M, soit 8 catégories différentes:

cluster	RFM Pattern	Customer Type
C1	R↑F↑M↑	Best
C2	R↑F↓M↑	Valuable
C3	R↑F↑M↓	Shopper
C4	R↑F↓M↓	First Time
C5	R↓F↑M↑	Churn
C6	R↓F↑M↓	Frequent
C7	R↓F↓M↑	Spenders
C8	R↓F↓M↓	Uncertain

Bien sûr, il se peut que toutes ces catégories (Best, Valuable, etc.) ne soient pas représentées dans le dataset et donc que certaines apparaissent plusieurs fois.

In [336]:
```
dfRFMClass = pd.DataFrame(RFMSegmentationCateg10122011['RFMClass'])
kmeans = KMeans(init='k-means++', n_clusters=8, n_init=100, random_state=3425)
dfRFMClass['cluster'] = kmeans.fit_predict(dfRFMClass)
print(kmeans.cluster_centers_)
```

```
[[ 439.01140684]
 [ 126.39612188]
 [ 316.75536481]
 [ 234.7551963 ]
 [ 336.6271722 ]
 [ 112.21220527]
 [ 418.57142857]
 [ 216.26246334]]
```

On obtient ainsi 8 clusters avec leurs centres respectifs. Le fait de n'avoir que 8 clusters permet qu'ils ne soient pas trop rapprochés les uns des autres.
Voici par exemple l'attribution des clusters à chacun des 5 premiers clients:

In [337]:
```
dfRFMClass.head()
```

Out[337]:

	RFMClass	cluster
CustomerID		
12347	444	0
12348	233	3
12349	314	2
12350	113	5
12352	342	4

In [338]:
```
RFMCategClustered = pd.merge(RFMSegmentationCateg10122011, dfRFMClass, left_index=True, right_index=True)
RFMCategClustered.drop('RFMClass_y', axis=1, inplace=True)
RFMCategClustered.columns = ['frequency', 'recency', 'amount', 'RFMClass', 'cluster']
RFMCategClustered.head()
```

Out[338]:

	frequency	recency	amount	RFMClass	cluster
CustomerID					
12347	7	2	615.714286	444	0
12348	4	75	359.310000	233	3
12349	1	18	1457.550000	314	2
12350	1	310	294.400000	113	5
12352	7	36	180.772857	342	4

Médiane des variables par cluster

In [339]:
```
RFMCategClusteredGroupedby = RFMCategClustered[['cluster', 'frequency', 'recency', 'amount']].groupby('cluster').median()
print(RFMCategClusteredGroupedby)
```

```
         frequency  recency    amount
cluster
0                7        7  323.8600
1                2      197  230.7400
2                1       30  257.5000
3                4       74  317.1750
4                5       28  303.7225
5                1      263  229.4700
6                2        9  243.4250
7                1       75  305.4100
```

In [340]:
```
quantilesRFMCategClusteredGroupedby = RFMCategClusteredGroupedby.quantile(q=[0.25,0.5,0.75])
print(quantilesRFMCategClusteredGroupedby)
```

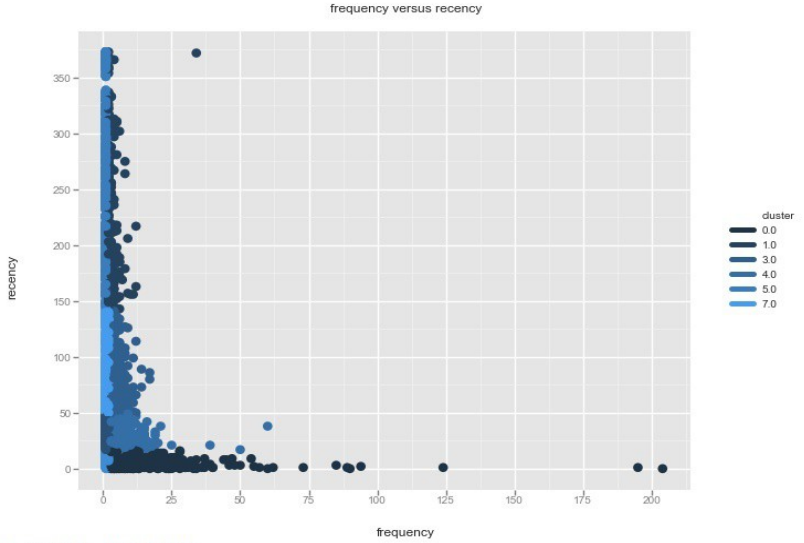
```
      frequency  recency     amount
0.25       1.00    23.25  240.25375
0.50       2.00    52.00  280.61125
0.75       4.25   105.50  308.35125
```

En reprenant la classification décrite ci-dessous qui consiste à séparer les "haut" et les "bas" des médianes des clusters pour chaque variable, et en prenant comme seuil de séparation les médianes de ces variables, on obtient :
F↑ pour une moyenne de la frequency par cluster >= 2.0
F↓ pour une moyenne de la frequency par cluster < 2.0
R↑ pour une moyenne de la recency par cluster < 52.00
R↓ pour une moyenne de la recency par cluster >= 52.00
M↑ pour une moyenne de la monetary_value par cluster >= 280.61125
M↓ pour une moyenne de la monetary_value par cluster < 280.61125

cluster	fréquence	récence	montante	F	R	M	Type de client
0	7	7	323.8600	F↑	R↑	M↑	Best
1	2	197	230.7400	F↑	R↓	M↓	Frequent
2	1	30	257.5000	F↓	R↑	M↓	First Time
3	4	74	317.1750	F↑	R↓	M↑	Churn
4	5	28	303.7225	F↑	R↑	M↑	Best
5	1	263	229.4700	F↓	R↓	M↓	Uncertain
6	2	9	243.4250	F↑	R↑	M↓	Shopper
7	1	75	305.4100	F↓	R↓	M↑	Spenders

On obtient ainsi le graphe permettant de visualiser les 8 clusters :

In [341]:
```
ggplot(RFMCategClustered, aes(x='frequency', y='recency', color='cluster')) + geom_point(size=75) + ggtitle("frequency versus recency")
```



Out[341]: <ggplot: (58217560)>

2- Evolution des segments de la segmentation RFM

On regarde pour chaque client, de quelle catégorie il passe à quelle catégorie, on regarde le nombre pour chaque transition.
Par exemple, le client 12347 a pour FRM 7 / 2 / 615.714286, c'est-à dire qu'il est 444 et R↑F↑M↑, Cluster 0, Best.
On fait les mêmes calculs pour les 6 premiers mois, puis pour les 6 derniers, et on regarde de quel cluster à quel cluster il va, puis on calcule chaque transition pour tous les clusters.

In [342]:
```
deb = dt.datetime(2010, 12, 1) # date de la première transaction
fin = dt.datetime(2011, 12, 9) # date de la dernière transaction
MI = deb + dt.timedelta((1+(fin-deb).days)/2)
periode1 = dfCleaned[dfCleaned['InvoiceDate']<MI]
periode2 = dfCleaned.copy()
```

In [343]:
```
print(MI)
```

```
2011-06-06 00:00:00
```

Période 1 (6 premiers mois)

In [344]:
```
# les enregistrements de la période "periode1" finissent au 05/06/2011, MI, du jour d'après, est la date de référence pour la récence
dfPeriode1 = getRFMSegmentationCateg(periode1, MI)
```

In [345]:
```
RFMCategPeriode1Median = dfPeriode1[['frequency', 'recency', 'amount']].median()
RFMCategPeriode1Mean = dfPeriode1[['frequency', 'recency', 'amount']].mean()
```

In [346]:
```
dfPeriode1RFM = pd.DataFrame(dfPeriode1['RFMClass'])
kmeans1 = KMeans(init='k-means++', n_clusters=8, n_init=100, random_state=3425)
dfPeriode1RFM['cluster'] = kmeans1.fit_predict(dfPeriode1RFM)
print(kmeans1.cluster_centers_)
dfPeriode1RFM.head()
```

```
[[ 317.19070905]
 [ 126.22222222]
 [ 417.575     ]
 [ 215.04513889]
 [ 112.27433628]
 [ 338.96350365]
 [ 236.81355932]
 [ 440.39698492]]
```

Out[346]:

	RFMClass	cluster
CustomerID		
12347	234	6
12348	233	6
12350	113	4
12352	241	6
12353	411	2

In [347]:
```
RFMCategClusteredPeriode1 = pd.merge(dfPeriode1, dfPeriode1RFM, left_index=True, right_index=True)
RFMCategClusteredPeriode1.drop('RFMClass_y', axis=1, inplace=True)
RFMCategClusteredPeriode1.columns = ['frequency', 'recency', 'amount', 'RFMClass', 'cluster']
RFMCategClusteredPeriode1.head()
```

Out[347]:

	frequency	recency	amount	RFMClass	cluster
CustomerID					
12347	3	59	607.810	234	6
12348	3	61	389.080	233	6
12350	1	123	294.400	113	4
12352	4	75	130.295	241	6
12353	1	17	89.000	411	2

In [348]:
```
RFMCategClusteredGroupedbyPeriode1Median = RFMCategClusteredPeriode1[['cluster', 'frequency', 'recency', 'amount']].groupby('cluster').median()
print(RFMCategClusteredGroupedbyPeriode1Median)
```

```
         frequency  recency    amount
cluster
0              1.0     31.0  304.0500
1              2.0    128.0  234.3375
2              1.5     12.0  303.0675
3              1.0     73.0  262.5025
4              1.0    144.0  233.4500
5              4.0     28.0  311.1035
6              3.0     67.0  290.4050
7              5.0     11.0  332.3880
```

In [349]:
```
print(RFMCategPeriode1Median)
```

```
frequency      2.0
recency       52.0
amount       286.8
dtype: float64
```

1ère période : tableau des médianes des variables pour chaque cluster ainsi que les catégories de chaque cluster vis-à-vis de ces médianes:

cluster	fréquence	récence	montant	F	R	M	Type de client
0	1.0	31.0	304.0500	F↓	R↑	M↑	Valuable
1	2.0	128.0	234.3375	F↑	R↓	M↓	Frequent
2	1.5	12.0	303.0675	F↓	R↑	M↑	Valuable
3	1.0	73.0	262.5025	F↓	R↓	M↓	Uncertain
4	1.0	144.0	233.4500	F↓	R↓	M↓	Uncertain
5	4.0	28.0	311.1035	F↑	R↑	M↑	Best
6	3.0	67.0	290.4050	F↑	R↓	M↑	Churn
7	5.0	11.0	332.3880	F↑	R↑	M↑	Best

In [400]:
```
RFMCategClusteredGroupedbyPeriode1Mean = RFMCategClusteredPeriode1[['cluster', 'frequency', 'recency', 'amount']].groupby('cl
uster').mean()
print(RFMCategClusteredGroupedbyPeriode1Mean)
```

```
         frequency     recency      amount
cluster
0         1.459658   33.667482  422.488779
1         2.674603  132.476190  298.366111
2         1.500000   10.760714  366.006143
3         1.267361   73.272569  350.834557
4         1.000000  146.389381  319.429027
5         4.956204   30.894161  370.134525
6         3.830508   68.754237  400.302571
7         7.610553    9.914573  456.467720
```

In [401]:
```
print(RFMCategPeriode1Mean)
```

```
frequency      2.726875
recency       65.154406
amount       373.546461
dtype: float64
```

1ère période : tableau des moyennes des variables pour chaque cluster ainsi que les catégories de chaque cluster vis-à-vis de ces moyennes:

cluster	fréquence	récence	montant	F	R	M	Type de client
0	1.459658	33.667482	422.488779	F↓	R↑	M↑	Valuable
1	2.674603	132.476190	298.366111	F↓	R↓	M↓	Uncertain
2	1.500000	10.760714	366.006143	F↓	R↑	M↓	First time
3	1.267361	73.272569	350.834557	F↓	R↓	M↓	Uncertain
4	1.000000	146.389381	319.429027	F↓	R↓	M↓	Uncertain
5	4.956204	30.894161	370.134525	F↑	R↑	M↓	Shopper
6	3.830508	68.754237	400.302571	F↑	R↓	M↑	Churn
7	7.610553	9.914573	456.467720	F↑	R↑	M↑	Best

Remarque: Si la médiane et la moyenne sont assez proches pour la fréquence et la récence, on peut remarquer qu'il y a une nette différence entre elles pour le montant. On passe par exemple de 286 à 373, ce qui fait qu'uun cluster avec la même valeur de montant pourra être mise dans deux catégories différentes en fonction du choix de la fonction médiane ou moyenne.

Statistiques sur les clusters de la première période

In [351]:
```
RFMCategClusteredGroupedbyPeriode1Stats = RFMCategClusteredPeriode1[['cluster', 'frequency', 'recency', 'amount']].groupby('cluster').min()
RFMCategClusteredGroupedbyPeriode1Stats.rename(columns={'frequency': 'frequency_min', 'recency': 'recency_min', 'amount': 'amount_min'}, inplace=True)
RFMCategClusteredGroupedbyPeriode1StatsMean = RFMCategClusteredPeriode1[['cluster', 'frequency', 'recency', 'amount']].groupby('cluster').mean()
RFMCategClusteredGroupedbyPeriode1StatsMean.rename(columns={'frequency': 'frequency_mean', 'recency': 'recency_mean', 'amount': 'amount_mean'}, inplace=True)
RFMCategClusteredGroupedbyPeriode1Stats = pd.merge(RFMCategClusteredGroupedbyPeriode1Stats, RFMCategClusteredGroupedbyPeriode1StatsMean, left_index=True, right_index=True)
RFMCategClusteredGroupedbyPeriode1StatsMax = RFMCategClusteredPeriode1[['cluster', 'frequency', 'recency', 'amount']].groupby('cluster').max()
RFMCategClusteredGroupedbyPeriode1StatsMax.rename(columns={'frequency': 'frequency_max', 'recency': 'recency_max', 'amount': 'amount_max'}, inplace=True)
RFMCategClusteredGroupedbyPeriode1Stats = pd.merge(RFMCategClusteredGroupedbyPeriode1Stats, RFMCategClusteredGroupedbyPeriode1StatsMax, left_index=True, right_index=True)
RFMCategClusteredGroupedbyPeriode1StatsMedian = RFMCategClusteredPeriode1[['cluster', 'frequency', 'recency', 'amount']].groupby('cluster').median()
RFMCategClusteredGroupedbyPeriode1StatsMedian.rename(columns={'frequency': 'frequency_median', 'recency': 'recency_median', 'amount': 'amount_median'}, inplace=True)
RFMCategClusteredGroupedbyPeriode1Stats = pd.merge(RFMCategClusteredGroupedbyPeriode1Stats, RFMCategClusteredGroupedbyPeriode1StatsMedian, left_index=True, right_index=True)
RFMCategClusteredGroupedbyPeriode1StatsStd = RFMCategClusteredPeriode1[['cluster', 'frequency', 'recency', 'amount']].groupby('cluster').std()
RFMCategClusteredGroupedbyPeriode1StatsStd.rename(columns={'frequency': 'frequency_std', 'recency': 'recency_std', 'amount': 'amount_std'}, inplace=True)
RFMCategClusteredGroupedbyPeriode1Stats = pd.merge(RFMCategClusteredGroupedbyPeriode1Stats, RFMCategClusteredGroupedbyPeriode1StatsStd, left_index=True, right_index=True)
print(RFMCategClusteredGroupedbyPeriode1Stats)
```

```
         frequency_min  recency_min  amount_min  frequency_mean  recency_mean
cluster
0                    1           20   30.600000        1.459658     33.667482
1                    2          100   11.900000        2.674603    132.476190
2                    1            0    2.900000        1.500000     10.760714
3                    1           52   10.655000        1.267361     73.272569
4                    1          100    3.750000        1.000000    146.389381
5                    3           20   12.164000        4.956204     30.894161
6                    3           52   65.416667        3.830508     68.754237
7                    3            0   43.719500        7.610553      9.914573

         amount_mean  frequency_max  recency_max    amount_max  \
cluster
0         422.488779              2           51  12425.450000
1         298.366111             34          186   1893.560000
2         366.006143              2           19   3096.000000
3         350.834557              2           98   3978.990000
4         319.429027              1          186   3202.920000
5         370.134525             29           51   2497.124000
6         400.302571              8           98   3209.627500
7         456.467720             85           19   8341.283333

         frequency_median  recency_median  amount_median  frequency_std  \
cluster
0                     1.0            31.0       304.0500       0.498980
1                     2.0           128.0       234.3375       2.917065
2                     1.5            12.0       303.0675       0.500895
3                     1.0            73.0       262.5025       0.442967
4                     1.0           144.0       233.4500       0.000000
5                     4.0            28.0       311.1035       2.911059
6                     3.0            67.0       290.4050       1.269666
7                     5.0            11.0       332.3880       7.968868

         recency_std  amount_std
cluster
0           9.394509  804.642676
1          24.336216  240.638097
2           6.100992  308.009450
3          13.864701  371.825878
4          29.656674  314.866188
5           8.788923  305.316539
6          12.328784  425.147457
7           6.010716  635.007881
```

Période 2 (12 premiers mois, totalité)

In [352]:
```
print(fin + dt.timedelta(1))
```

```
2011-12-10 00:00:00
```

In [353]:
```
# fin + dt.timedelta(1) sert de date de référence pour la récence de la seconde période
dfPeriode2 = getRFMSegmentationCateg(periode2, fin + dt.timedelta(1))
```

In [354]:
```
RFMCategPeriode2Median = dfPeriode2[['frequency', 'recency', 'amount']].median()
RFMCategPeriode2Mean = dfPeriode2[['frequency', 'recency', 'amount']].mean()
```

In [355]:
```
dfPeriode2RFM = pd.DataFrame(dfPeriode2['RFMClass'])
kmeans2 = KMeans(init='k-means++', n_clusters=8, n_init=100, random_state=3425)
dfPeriode2RFM['cluster'] = kmeans2.fit_predict(dfPeriode2RFM)
print(kmeans2.cluster_centers_)
dfPeriode2RFM.head()
```

```
[[ 439.01140684]
 [ 126.39612188]
 [ 316.75536481]
 [ 234.7551963 ]
 [ 336.6271722 ]
 [ 112.21220527]
 [ 418.57142857]
 [ 216.26246334]]
```

Out[355]:

	RFMClass	cluster
CustomerID		
12347	444	0
12348	233	3
12349	314	2
12350	113	5
12352	342	4

In [356]:
```
RFMCategClusteredPeriode2 = pd.merge(dfPeriode2, dfPeriode2RFM, left_index=True, right_index=True)
RFMCategClusteredPeriode2.drop('RFMClass_y', axis=1, inplace=True)
RFMCategClusteredPeriode2.columns = ['frequency', 'recency', 'amount', 'RFMClass', 'cluster']
RFMCategClusteredPeriode2.head()
```

Out[356]:

	frequency	recency	amount	RFMClass	cluster
CustomerID					
12347	7	2	615.714286	444	0
12348	4	75	359.310000	233	3
12349	1	18	1457.550000	314	2
12350	1	310	294.400000	113	5
12352	7	36	180.772857	342	4

In [357]:
```
RFMCategClusteredGroupedbyPeriode2Median = RFMCategClusteredPeriode2[['cluster', 'frequency', 'recency', 'amount']].groupby('cluster').median()
print(RFMCategClusteredGroupedbyPeriode2Median)
```

```
         frequency  recency    amount
cluster
0                7        7  323.8600
1                2      197  230.7400
2                1       30  257.5000
3                4       74  317.1750
4                5       28  303.7225
5                1      263  229.4700
6                2        9  243.4250
7                1       75  305.4100
```

In [358]:
```
print(RFMCategPeriode2Median)
```

```
frequency      2.00
recency       50.00
amount       289.24
dtype: float64
```

2ème période : tableau des médianes des variables pour chaque cluster ainsi que les catégories de chaque cluster vis-à-vis de ces médianes:

cluster	fréquence	récence	montant	F	R	M	Type de client
0	7.0	7.0	323.8600	F↑	R↑	M↑	Best
1	2.0	197.0	230.7400	F↑	R↓	M↓	Frequent
2	1.0	30.0	257.5000	F↓	R↑	M↓	First Time
3	4.0	74.0	317.1750	F↑	R↓	M↑	Churn
4	5.0	28.0	303.7225	F↑	R↑	M↑	Best
5	1.0	263.0	229.4700	F↓	R↓	M↓	Uncertain
6	2.0	9.0	243.4250	F↑	R↑	M↓	Shopper
7	1.0	75.0	305.4100	F↓	R↓	M↑	Spenders

In [359]:
```
RFMCategClusteredGroupedbyPeriode2Mean = RFMCategClusteredPeriode2[['cluster', 'frequency', 'recency', 'amount']].groupby('cl
uster').mean()
print(RFMCategClusteredGroupedbyPeriode2Mean)
```

```
         frequency     recency      amount
cluster
0        11.006337    6.816223  419.584662
1         2.839335  212.409972  386.541090
2         1.437768   30.515021  337.212897
3         4.568129   81.540416  370.242870
4         6.113744   28.878357  393.726526
5         1.000000  264.352288  324.518752
6         1.626050    8.378151  316.704601
7         1.368035   82.881232  425.031798
```

In [360]:
```
print(RFMCategPeriode2Mean)
```

```
frequency      4.225769
recency       92.292852
amount       378.557552
dtype: float64
```

2ème période : tableau des moyennes des variables pour chaque cluster ainsi que les catégories de chaque cluster vis-à-vis de ces moyennes:

cluster	fréquence	récence	montant	F	R	M	Type de client
0	11.006337	6.816223	419.584662	F↑	R↑	M↑	Best
1	2.839335	212.409972	386.541090	F↓	R↓	M↑	Spenders
2	1.437768	30.515021	337.212897	F↓	R↑	M↓	First Time
3	4.568129	81.540416	370.242870	F↑	R↑	M↓	Shopper
4	6.113744	28.878357	393.726526	F↑	R↑	M↑	Best
5	1.000000	264.352288	324.518752	F↓	R↓	M↓	Uncertain
6	1.626050	8.378151	316.704601	F↓	R↑	M↓	First Time
7	1.368035	82.881232	425.031798	F↓	R↑	M↑	Valuable

Statistiques sur les clusters de la seconde période

In [361]:
```
RFMCategClusteredGroupedbyPeriode2Stats = RFMCategClusteredPeriode2[['cluster', 'frequency', 'recency', 'amount']].groupby('c
luster').min()
RFMCategClusteredGroupedbyPeriode2Stats.rename(columns={'frequency': 'frequency_min', 'recency': 'recency_min', 'amount': 'am
ount_min'}, inplace=True)
RFMCategClusteredGroupedbyPeriode2StatsMean = RFMCategClusteredPeriode2[['cluster', 'frequency', 'recency', 'amount']].groupb
y('cluster').mean()
RFMCategClusteredGroupedbyPeriode2StatsMean.rename(columns={'frequency': 'frequency_mean', 'recency': 'recency_mean', 'amount
': 'amount_mean'}, inplace=True)
RFMCategClusteredGroupedbyPeriode2Stats = pd.merge(RFMCategClusteredGroupedbyPeriode2Stats, RFMCategClusteredGroupedbyPeriode
2StatsMean, left_index=True, right_index=True)
RFMCategClusteredGroupedbyPeriode2StatsMax = RFMCategClusteredPeriode2[['cluster', 'frequency', 'recency', 'amount']].groupby
('cluster').max()
RFMCategClusteredGroupedbyPeriode2StatsMax.rename(columns={'frequency': 'frequency_max', 'recency': 'recency_max', 'amount':
'amount_max'}, inplace=True)
RFMCategClusteredGroupedbyPeriode2Stats = pd.merge(RFMCategClusteredGroupedbyPeriode2Stats, RFMCategClusteredGroupedbyPeriode
2StatsMax, left_index=True, right_index=True)
RFMCategClusteredGroupedbyPeriode2StatsMedian = RFMCategClusteredPeriode2[['cluster', 'frequency', 'recency', 'amount']].grou
pby('cluster').median()
RFMCategClusteredGroupedbyPeriode2StatsMedian.rename(columns={'frequency': 'frequency_median', 'recency': 'recency_median', '
amount': 'amount_median'}, inplace=True)
RFMCategClusteredGroupedbyPeriode2Stats = pd.merge(RFMCategClusteredGroupedbyPeriode2Stats, RFMCategClusteredGroupedbyPeriode
2StatsMedian, left_index=True, right_index=True)
RFMCategClusteredGroupedbyPeriode2StatsStd = RFMCategClusteredPeriode2[['cluster', 'frequency', 'recency', 'amount']].groupby
('cluster').std()
RFMCategClusteredGroupedbyPeriode2StatsStd.rename(columns={'frequency': 'frequency_std', 'recency': 'recency_std', 'amount':
'amount_std'}, inplace=True)
RFMCategClusteredGroupedbyPeriode2Stats = pd.merge(RFMCategClusteredGroupedbyPeriode2Stats, RFMCategClusteredGroupedbyPeriode
2StatsStd, left_index=True, right_index=True)
print(RFMCategClusteredGroupedbyPeriode2Stats)
```

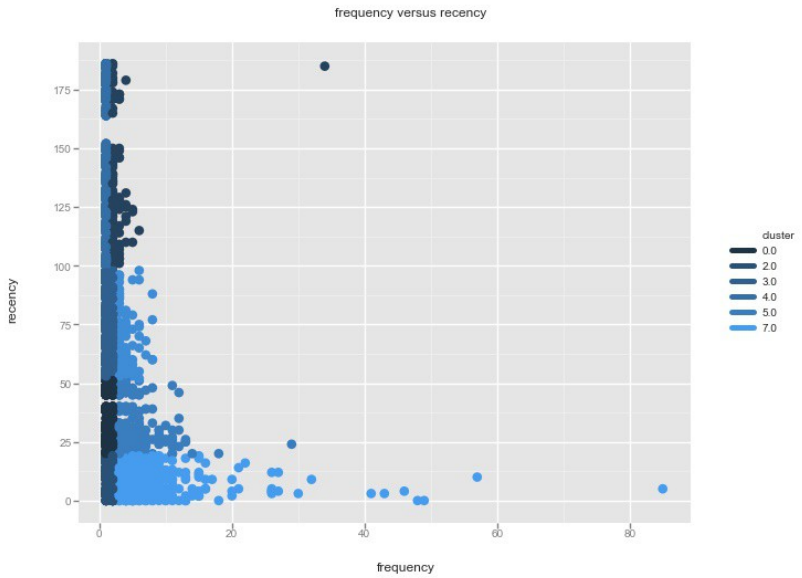
```
         frequency_min  recency_min  amount_min  frequency_mean  recency_mean  \
cluster
0                    3            0   27.310000       11.006337      6.816223
1                    2          143   11.670000        2.839335    212.409972
2                    1           17   20.800000        1.437768     30.515021
3                    3           50   28.216667        4.568129     81.540416
4                    3           17    9.140000        6.113744     28.878357
5                    1          143    2.900000        1.000000    264.352288
6                    1            0    0.000000        1.626050      8.378151
7                    1           50    5.900000        1.368035     82.881232

         amount_mean  frequency_max  recency_max    amount_max  \
cluster
0         419.584662            204           16   4327.621667
1         386.541090             34          373  19809.750000
2         337.212897              2           49   6207.670000
3         370.242870             17          142   2280.097500
4         393.726526             60           49   6512.930526
5         324.518752              1          373   9341.260000
6         316.704601              2           16   3861.000000
7         425.031798              2          142   5533.860000
```

cluster	frequency_median	recency_median	amount_median	frequency_std	recency_std	amount_std
0	7	7	323.8600	14.805977	4.846155	439.221259
1	2	197	230.7400	2.212767	55.350783	1233.007535
2	1	30	257.5000	0.496645	9.282704	378.548999
3	4	74	317.1750	2.122542	24.969961	262.993221
4	5	28	303.7225	4.651087	8.810742	401.448295
5	1	263	229.4700	0.000000	65.398018	466.729553
6	2	9	243.4250	0.484870	4.654278	320.036668
7	1	75	305.4100	0.482625	26.258888	530.230636

Comparaison des deux graphiques représentant le positionnement des différents clusters les uns par rapport aux autres

1ère période (6 premiers mois):

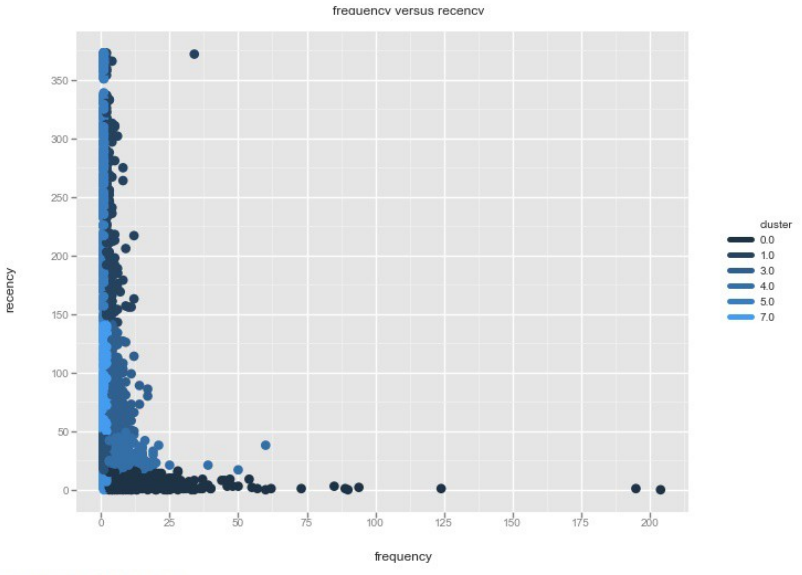
In [362]:
```
ggplot(RFMCategClusteredPeriode1, aes(x='frequency', y='recency', color='cluster')) + geom_point(size=75) + ggtitle("frequenc
y versus recency")
```



Out[362]: <ggplot: (35642621)>

2ème période (les 12 mois):

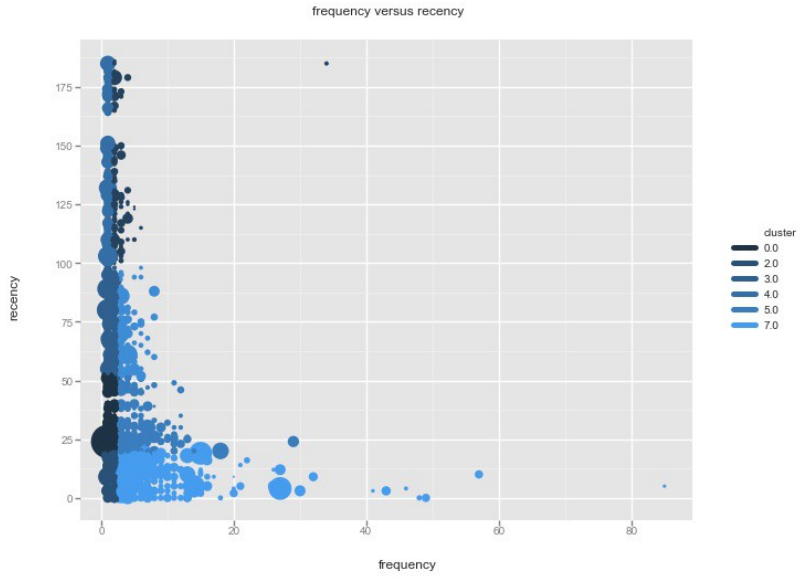
In [363]:
```
ggplot(RFMCategClusteredPeriode2, aes(x='frequency', y='recency', color='cluster')) + geom_point(size=75) + ggtitle("frequenc
y versus recency")
```



Out[363]: <ggplot: (29154831)>

Dans le graphique de la première période ci-dessous, j'ai adapté la taille des points aux rapports des montants (j'ai supprimé l'enregistrement d'un tel montant qu'il empêchait la lisibilité sur les autres données), on peut juste noter l'impression que les clients qui n'achètent pas fréquemment, achètent en plus gros volume.

In [364]:
```
RFMCategClusteredPeriode11 = RFMCategClusteredPeriode1[RFMCategClusteredPeriode1['amount']<10000]
RFMCategClusteredPeriode11['amount'] = RFMCategClusteredPeriode11['amount']/10
ggplot(RFMCategClusteredPeriode11, aes(x='frequency', y='recency', color='cluster')) + geom_point(aes(size='amount')) + ggtit
le("frequency versus recency")
```



Out[364]: <ggplot: (35050198)>

Matrice de transition du passage d'une catégorie à une autre

In [365]:
```
listeClient = dfCleaned.CustomerID.unique()
listeClientPeriode1 = periode1.CustomerID.unique()
listeClientPeriode2 = periode2.CustomerID.unique()
```

In [366]:
```
indexCluster = []
for i in range(9):
    for j in range(9):
        indexCluster.append(str(i)+str(j))
```

In [367]:
```
dfEvolutionCluster = pd.DataFrame(index = indexCluster, columns = ['source', 'target', 'value'])
for i in range(9):
    for j in range(9):
        dfEvolutionCluster.loc[str(i)+str(j)].source = 'Cluster'+str(i)
        dfEvolutionCluster.loc[str(i)+str(j)].target = 'Cluster'+str(j)
        dfEvolutionCluster.loc[str(i)+str(j)].value = 0
```

In [368]:
```
# listeClient = listeClientPeriode2 mais c'est au cas où on voudrait changer la période2
for client in listeClient:
    if client in listeClientPeriode1:
        if client in listeClientPeriode2:
            pos = str(1+RFMCategClusteredPeriode1.loc[client].cluster) + str(1+RFMCategClusteredPeriode2.loc[client].cluster)
        else:
            pos = str(1+RFMCategClusteredPeriode1.loc[client].cluster) + "0"
    else:
        if client in listeClientPeriode2:
            pos = "0" + str(1+RFMCategClusteredPeriode2.loc[client].cluster)
    dfEvolutionCluster.loc[pos].value = dfEvolutionCluster.loc[pos].value + 1
```

On a donc le nombre de clients passant d'un cluster à un autre.
On cherche maintenant à connaître la matrice de transition des catégories. Comme un cluster peut être classé dans deux catégories différentes en fonction de la fonction qui lui est appliquée (moyenne ou médiane), on ne va prendre qu'un exemple, celui de la moyenne.
Attention, même s'ils portent les mêmes noms, ce ne sont pas les mêmes clusters. Ces clusters sont relatifs à une période donnée, relatifs à un ensemble déterminé de clients, puisque la classification RFM est construite à partir des quantiles d'un ensemble de clients. Un client peut donc garder exactement le même comportement d'achat tout au long des différentes périodes, mais changer de cluster et de catégorie car leur définition n'est pas commune aux deux périodes. C'est un problème qui s'amenuise quand on augmente la taille du dataset, mais qui doit être noté pour des datasets pas assez volumineux.
Prenons par exemple le cas du client 12348:

In [369]:
```
print(RFMCategClusteredPeriode1.loc['12348'])
print(RFMCategClusteredPeriode2.loc['12348'])
```

```
frequency         3
recency          61
amount       389.08
RFMClass        233
cluster           6
Name: 12348, dtype: object
frequency         4
recency          75
amount       359.31
RFMClass        233
cluster           3
Name: 12348, dtype: object
```

Son comportement n'a pas trop évolué et il est catégorisé RFM comme 233 pour les deux périodes.
Pourtant, il a changé de cluster. Il est passé du 6 au 3
Si on se réfère à la médiane, cela ne change rien car ces deux clusters sont étiquetés comme "churn".
6 3.0 67.0 290.4050 F↑ R↓ M↑ Churn
3 4.0 74.0 317.1750 F↑ R↓ M↑ Churn
Par contre, si on prend la moyenne,
6 3.830508 68.754237 400.302571 F↑ R↓ M↑ Churn
3 4.568129 81.540416 370.242870 F↑ R↑ M↓ Shopper
ce client au comportement presque constant, passe de l'étiquette Churn à Shopper, donc avec une stratégie marketing différente, alors qu'il n'a quasiment pas changé de comportement d'achat.

On doit donc garder à l'esprit lorsqu'on veut comparer deux périodes, que les frontières des clusters et des catégories dépendent des comportements généraux des clients, et que puisque ceux-ci peuvent changer dans leur ensemble, la classification des individus peut en être modifiée indépendamment de leur comportement spécifique.

In [370]:
```
periode1Categories = ['First Time', 'Spenders', 'First Time', 'Uncertain', 'Best', 'Best', 'Frequent', 'Uncertain']
periode2Categories = ['Spenders', 'Valuable', 'Shopper', 'First Time', 'Best', 'Valuable', 'Shopper', 'Uncertain']
categories = ['Uncertain', 'Spenders', 'Frequent', 'Churn', 'First Time', 'Shopper', 'Valuable', 'Best']
```

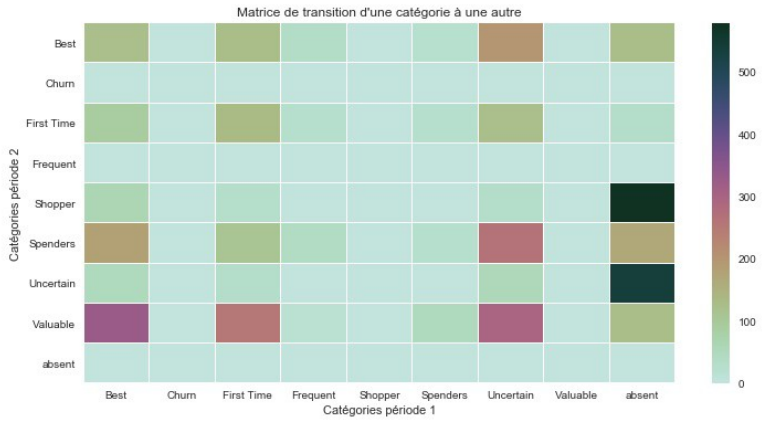
In [371]:
```
indexCategorie = []
for i in range(9):
    for j in range(9):
        indexCategorie.append(str(i)+str(j))
```

In [372]:
```
dfEvolutionCategorie = pd.DataFrame(index = indexCategorie, columns = ['source', 'target', 'value'])
for i in range(9):
    for j in range(9):
        if i > 0:
            dfEvolutionCategorie.loc[str(i)+str(j)].source = categories[i-1]
            if j > 0:
                dfEvolutionCategorie.loc[str(i)+str(j)].target = categories[j-1]
            else:
                dfEvolutionCategorie.loc[str(i)+str(j)].target = 'absent'
        else:
            dfEvolutionCategorie.loc[str(i)+str(j)].source = 'absent'
            if j > 0:
                dfEvolutionCategorie.loc[str(i)+str(j)].target = categories[j-1]
            else:
                dfEvolutionCategorie.loc[str(i)+str(j)].target = 'absent'
        dfEvolutionCategorie.loc[str(i)+str(j)].value = 0
```

In [373]:
```
for i in range(9):
    for j in range(9):
        if i > 0:
            if j > 0:
                dfEvolutionCategorie.loc[str(1+categories.index(periode1Categories[i-1]))+str(1+categories.index(periode2Cat
egories[j-1]))].value += dfEvolutionCluster.loc[str(i)+str(j)].value
            else:
                dfEvolutionCategorie.loc[str(1+categories.index(periode1Categories[i-1]))+"0"].value += dfEvolutionCluster.l
oc[str(i)+str(j)].value
        else:
            if j > 0:
                dfEvolutionCategorie.loc["0"+str(1+categories.index(periode2Categories[j-1]))].value += dfEvolutionCluster.l
oc[str(i)+str(j)].value
```

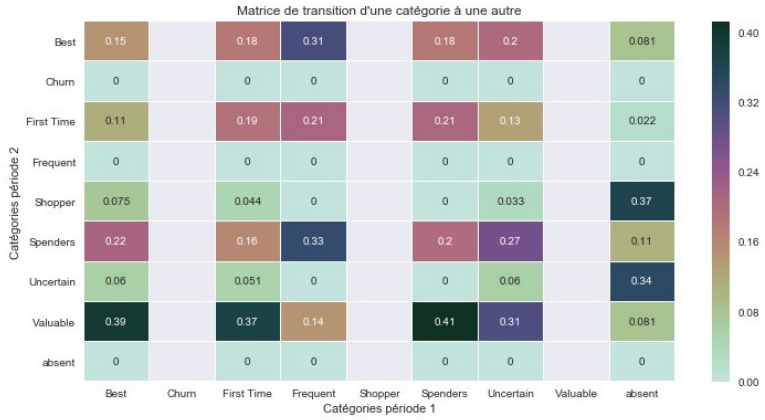
In [408]:
```
dfEvolutionCategorie["value"] = dfEvolutionCategorie["value"].astype(int)
pt = dfEvolutionCategorie.pivot_table(index = 'target', columns = 'source', values = 'value')
f, ax = plt.subplots(figsize = (12, 6))
cmap = sns.cubehelix_palette(start = 1.5, rot = 1.5, as_cmap = True)
sns.heatmap(pt, cmap = cmap, linewidths = 0.05, ax = ax)
ax.set_title(u"Matrice de transition d'une catégorie à une autre")
ax.set_xlabel(u"Catégories période 1")
ax.set_ylabel(u"Catégories période 2")
```

Out[408]:
```
<matplotlib.text.Text at 0x21fd99e8>
```



In [375]:
```
def getPercent(path, row):
    return row / dfEvolutionCategorieSum.loc[path]

dfEvolutionCategorieSum = dfEvolutionCategorie.groupby(['source']).sum()
dfEvolutionCategorie2 = dfEvolutionCategorie.copy()
dfEvolutionCategorie2['pourcentage'] = map(lambda path, row: getPercent(path, row), dfEvolutionCategorie2['source'], dfEvolu
tionCategorie2['value'])

dfEvolutionCategorie2["pourcentage"] = dfEvolutionCategorie2["pourcentage"].astype(float)
pt = dfEvolutionCategorie2.pivot_table(index = 'target', columns = 'source', values = 'pourcentage')
f, ax = plt.subplots(figsize = (12, 6))
cmap = sns.cubehelix_palette(start = 1.5, rot = 1.5, as_cmap = True)
sns.heatmap(pt, cmap = cmap, linewidths = 0.05, ax = ax, annot=True)
ax.set_title(u"Matrice de transition d'une catégorie à une autre")
ax.set_xlabel(u"Catégories période 1")
ax.set_ylabel(u"Catégories période 2")
```

Out[375]: `<matplotlib.text.Text at 0x433a86d8>`



Remarques :

- **On voit ainsi par exemple qu'un client qui n'était pas présent dans les 6 premiers mois (catégorie 'absent''), s'il arrive dans les 6 mois suivant, a le plus de chance (probabilité = 0.37) de passer en client 'Shopper', c'est-à-dire fréquence plus élevée que la moyenne des autres clients, récence plus élevée mais montant moins élevé, c'est-à-dire qu'on peut l'interpréter comme un client n'ayant jamais acheté auparavant dans le magasin, et qui est prudent, qui teste.**
- **La plus grosse transition est celle d'un client 'Spenders' (F↓R↓M↑) passant en client 'Valuable' (F↓R↑M↑), c'est-à-dire que sa récence s'améliore, il revient.**
- **Il n'y a toutefois pas de transition presque nécessaire (le maximum étant de 0.41).**
- **Il n'y avait pas dans la première période, de client 'Churn', 'Shopper' ni 'Valuable'.**
- **Il peut être intéressant de regarder dans le détail quels sont les clients qui passent d'une catégorie donnée à une autre catégorie pour tenter de dégager les raisons de ces transitions.**
- **J'ai restreint le nombre de catégories à 8 (Churn, Shopper, etc.) mais il y a parfois des clients qui sont dans la même catégorie mais qui ont des valeurs quand même assez différentes, il pourrait alors être intéressant d'augmenter le nombre de catégories pour départager ces comportements, par exemple des clients Best et Super Best, ou Uncertain et Super Uncertain.**

3- Customer Lifetime Value

Dans l'optique de continuer à étudier le comportement des clients, on peut s'intéresser à la "Valeur Vie Client", traduction de la Customer Lifetime Value, ou CLV. Il s'agit en marketing, d'étudier les profits attendus lors de la "durée de vie" du client, c'est-à-dire la période pendant laquelle il consomme un bien de l'entreprise. Dans notre cas, il s'agit de la pétiode où un client achète/commande des articles de Datazon.

Tous les clients ne sont pas d'une rentabilité équivalente. Ils ne méritent pas les efforts destinés à les fidéliser. Parfois même un client peut coûter plus cher à l'entreprise que ce qu'il rapporte. Les profits dégagés estimés dans la durée sont en fait supérieurs aux frais dépensés à son égard.
La mesure de la valeur des clients, client par client, est alors une opération essentielle pour mieux cibler et rentabiliser les actions marketing et au final améliorer la rentabilité globale.
On peut ainsi utiliser la Customer Lifetime Value qui est un indicateur estimant, sous la forme d'une espérance mathématique, la somme des profits nets susceptibles d'être générés par un unique client au fil de sa durée de vie.

On peut se baser par exemple sur les deux articles suivants pour comprendre les probabilités utilisées:

http://mktg.uni-svishtov.bg/ivm/resources/Counting_Your_Customers.pdf

https://www.datascience.com/blog/intro-to-predictive-modeling-for-customer-lifetime-value

La CLV peut se baser sur 3 variables dont voici la définition:

- frequency : la fréquence représente le nombre d'achats suivant le premier achat effectué. C'est donc une unité de moins que le nombre total d'achats.
- T : représenten l'âge du client (non pas de la personne elle-même, mais de son historique d'achats), peu importe l'unité de temps choisie (pour nous ce sera le jour). l'âge du client est égal à la durée (donc en jour) entre le premier achat du client, et la date de référence choisie pour la période étudiée.
- recency : la récence représente l'âge du client quand il a fait son achat le plus récent. Il est donc égal à la durée entre son premier achat et son dernier achat. (Ainsi, l'âge / la récence d'un client n'ayant fait qu'un seul achat sera de 0.)

La CLV se fait sur toute la période du dataset, donc la date de référence sera le jour d'après la dernière transaction, soit le 10/12/2011

In [376]:
```
# date de référence
dateRefCLV = dt.datetime(2011,12,10)
dfCLV = dfCleaned.copy()
# calcul de la différence en jours entre la date de référence et la facture d'un client
dfCLV['InvoiceDateDayDiff'] = dfCLV['InvoiceDate'].apply(lambda x: calc(dateRefCLV, x))
# calcul des valeurs des variables T et frequency
rfmTableCLV1 = dfCLV.groupby('CustomerID').agg({'InvoiceDateDayDiff': lambda x: x.max(), # T
                                                'InvoiceNo': lambda x: len(x.unique())-1}) # frequency
rfmTableCLV1.rename(columns={'InvoiceDateDayDiff': 'T', 'InvoiceNo': 'frequency'}, inplace=True)
# calcul des valeurs de la variable recency
rfmTableCLV2 = dfCLV.groupby('CustomerID').agg({'InvoiceDateDayDiff': lambda x: x.max()-x.min()}) # Recency
rfmTableCLV2.rename(columns={'InvoiceDateDayDiff': 'recency'}, inplace=True)
# rassemblement en une seule table
RFTtable = pd.merge(rfmTableCLV1, rfmTableCLV2, left_index=True, right_index=True)
RFTtable.head()
```

Out[376]:

	frequency	T	recency
CustomerID			
12347	6	367	365
12348	3	358	283
12349	0	18	0
12350	0	310	0
12352	6	296	260

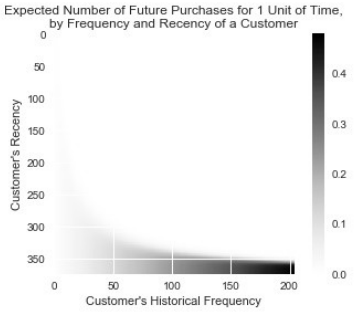
Pour effectuer les calculs, j'utilise la librairie lifetimes et ses méthodes de calcul comme BetaGeoFitter ou de visualisation comme plot_frequency_recency_matrix etc.

In [377]:
```
bgf = BetaGeoFitter(penalizer_coef=0.0)
bgf.fit(RFTtable['frequency'], RFTtable['recency'], RFTtable['T'])
print(bgf)
```

```
<lifetimes.BetaGeoFitter: fitted with 4323 subjects, a: 0.03, alpha: 51.86, b: 3.49, r: 0.72>
```

Suivant les données de nos clients, on peut voir la tendance qu'ils auront à acheter par unité de temps (pour nous c'est par jour) en fonction de sa fréquence et de sa récence (sa longueur de vie d'acheteur):

In [378]:
```
plot_frequency_recency_matrix(bgf)
```

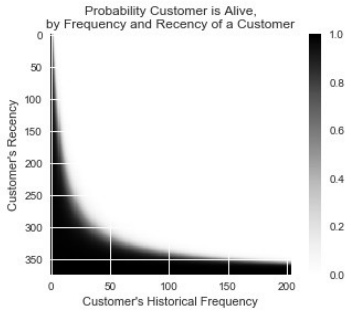


On observe logiquement que plus un client achète fréquemment et que ces achats sont dispersés sur une longue période (donc qu'il s'agit d'un "vieux" client en terme de vie d'acheteur), plus il est attendu que le nombre d'achat qu'il effectuera par jour sera élevé.

On peut également s'intéresser à la probabilité qu'un client soit "en vie", c'est-à-dire qu'il continue à acheter:

In [379]:
```
plot_probability_alive_matrix(bgf)
```

Out[379]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x37143908>
```



On peut ainsi noter que:

- les meilleurs clients, c'est-à-dire ceux qui ont le plus de chance d'être en vie, donc de revenir, se situent en bas à droite, ce sont les clients ayant acheté près de deux cents fois et dont l'âge approche "la durée de vie du dataset", c'est-à-dire que ce sont des clients qui ont acheté de façon constante pendant toute l'année.
- la zone à probabilité proche de 0 comprend par exemple d'éventuels clients qui auraient par exemple acheté près de 150 fois en moins de 50 jours, cela est vraiment peu probable, et d'ailleurs il n'y en a pas dans le dataset.
- il y a également des acheteurs qui ont une bonne probabilité de revenir, ce sont ceux ("queue" à gauche) qui ne sont pas venus fréquemment, mais comme ils viennent régulièrement (longue durée de vie), on a de bonnes chances de les revoir.

Une autre information intéressante peut être d'ordonner les clients en fonction de la prédiction de leurs futurs possibles achats dans le jour suivant (en fonction de leur historique):

In [380]:
```
RFTtable['predicted_purchases'] = bgf.conditional_expected_number_of_purchases_up_to_time(1, RFTtable['frequency'], RFTtable
['recency'], RFTtable['T'])
RFTtable.sort_values(by='predicted_purchases').head()
```

Out[380]:

	frequency	T	recency	predicted_purchases
CustomerID				
17850	33	373	1	2.781819e-29
15332	3	369	3	7.444804e-04
13093	7	373	98	1.650430e-03
14729	0	373	0	1.694840e-03
13065	0	373	0	1.694840e-03

In [381]:
```
RFTtable.sort_values(by='predicted_purchases').tail()
```

Out[381]:

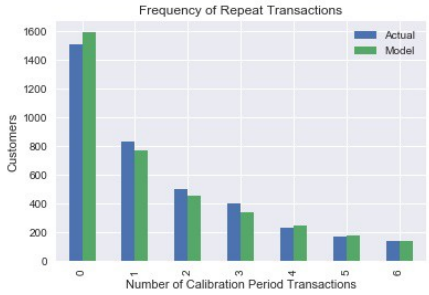
	frequency	T	recency	predicted_purchases
CustomerID				
15311	89	373	373	0.211100
13089	93	369	367	0.222571
17841	123	373	372	0.291100
14911	194	373	372	0.458192
12748	203	373	373	0.479416

Au contraire, le client 12748 a de fortes chances d'acheter à nouveau au prochain jour tant il a acheté de façon répétée, depuis longtemps (depuis le début d'après son T) et très récemment (d'après son recency).

Mais le modèle est-il fiable ?
On peut essayer de confronter les données du dataset aux prédictions du modèle.

In [382]:
```
plot_period_transactions(bgf)
```

Out[382]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x30565908>
```



On observe que le modèle prédit globalement bien les données du dataset.

Pour l'instant, nous avons vu des données globales. On va essayer de descendre au niveau du client en visualisant sa probabilité historique d'être vivant:

In [383]:
```
transaction_data = dfCleaned[['InvoiceDate', 'CustomerID']]
```

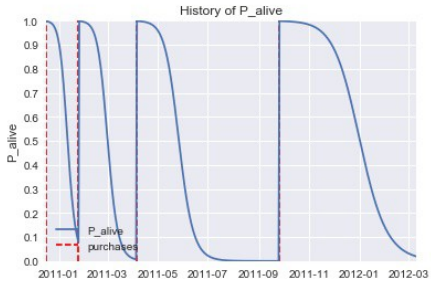
Exemple : prenons le cas du client 12348:

In [384]:
```
print(RFTtable.loc['12348'])
```

```
frequency              3.000000
T                    358.000000
recency              283.000000
predicted_purchases    0.008971
Name: 12348, dtype: float64
```

In [385]:
```
days_since_birth = 450
sp_trans = transaction_data.loc[transaction_data['CustomerID'] == '12348']
plot_history_alive(bgf, days_since_birth, sp_trans, 'InvoiceDate')
```

Out[385]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x30564320>
```



- **On retrouve bien les 3+1 achats (16/12/2010, 25/01/2011, 05/04/2011, 25/09/2011).**
- **On observe également qu'au fur et à mesure qu'il y a des achats répétés, la probabilité d'être vivant met plus de temps à décroître, ce qui semble normal puisque plus un client revient souvent, plus il a de chance de revenir.**

4- Système de recommandation

Dans ce qui suit, j'ai utilisé l'article suivant: https://opendatascience.com/rec-system/?utm_content=52586516&_hsenc=p2ANqtz-9jGizLIpsoa76ETOX2LRnsRKzzER0lleENGuQulvUflclIijdwfT6L6w-md3zQOEiTZp3xaly1l0CsoeDgKVLRhzkPKg&_hsmi=52595398

Une des stratégies qui peut être utilisée pour fidéliser un client ou faire revenir un ancien bon client qui "hésiterait" (catégorie Churn), est d'utiliser un système de recommandation. Par exemple, chez Amazon, 35% des ventes proviennent de recommandations, chez Netflix, 67% des films. On peut alors chercher, en utilisant l'historique des clients, à recommander des articles susceptibles de les intéresser afin de "les faire passer dans des catégories plus intéressantes" pour le magasin. Cela peut se faire quand on a suffisamment de données sur les achats des clients.

Pour créer un système de recommandation, on se base sur la notion de "collaborative filtering" (filtrage collaboratif) utilisant les liens entre les clients et leurs actions (achats d'articles dans notre cas).

Puisque dans les données, on ne sait rien sur le client (âge, profession, goûts) ni sur les produits (catégories, composition), on peut utiliser la méthode ALS (Alternating Least Squares) afin de déterminer, à partir de l'imposante matrice mais très clairsemée des interactions utilisateurs / articles, les caractéristiques latentes qui les relient. C'est ce que fait la méthode ALS à travers la factorisation de matrices. Elle décompose la matrice creuse (sparse matrix) des interactions utilisateurs / articles en un produit de matrices de dimensions inférieures à la matrice creuse et qui correspondent aux caractéristiques respectives des utilisateurs et des articles. (Leur produit n'est toutefois qu'une approximation de la matrice initiale.)

L'auteur de l'article s'est basé sur l'article de Yifan Hu, Yehuda Koren et Chris Volinsky pour produire son code: http://yifanhu.net/PUB/cf.pdf

Toutefois, en écrivant son code directement à partir des formules de l'article, il a rencontré un problème de lenteur d'exécution. Il a alors utilisé une librairie faisant les mêmes calculs mais de façon nettement plus rapide. C'est donc plutôt cette librairie, implicit, que j'ai reprise.

In [386]:
```
dfCleanedSDR = dfCleaned.copy()
# table d'association code article / description article
item_lookup = dfCleanedSDR[['StockCode', 'Description']].drop_duplicates() # Only get unique item/description pairs
item_lookup['StockCode'] = item_lookup.StockCode.astype(str) # Encode as strings for future lookup ease
```

Afin de pouvoir entraîner et tester le modèle, ainsi que pour évaluer sa performance, on va séparer les données initiales en plusieurs datasets, dataset d'entraînement et dataset de test.
Toutefois, comme on a affaire des achats d'articles, il est probable que ces articles ne seront pas achetés à nouveau dans une période proche. Il est alors préférable, plutôt que de faire deux tranches temporelles de 80% et 20%, de choisir aléatoirement les données masquées dans l'ensemble du dataset initial.
C'est ce que fait la méthode suivante make_train:

In [387]:
```
def make_train(ratings, pct_test = 0.2):
    '''
    Fonction qui prend en entrée la matrice originale clients / articles et "masque" un pourcentage donné des
    données initiales où il y a eu une interaction afin d'en faire différents dataset. Un set de test contient toutes les
    données originales alors que un set d'entraînement remplace le pourcentage choisi de données avec un zéro dans
    la matrice originale.
    paramètres:

    ratings - la matrice originelle creuse à partir de laquelle on crée les dataset de test et d'entraînement.
    Le dataset de test est juste une copie du dataset original. Forme d'une matrice creuse (sparse csr_matrix).

    pct_test - Le pourcentage d'interactions clients / articles qu'on veut masker dans le dataset d'entraînement afin
    de comparaison avec le dataset de test contenant toutes les informations initiales.

    retourne:
    training_set - La version modifiée du dataset originel avec un certain pourcentage de données mise à zero.

    test_set - Copie de la matrice originelle, non modifiée, afin qu'elle puisse être utilisée afin de comparaison avec les
    interactions réelles.

    user_inds - A partir des indices clients / articles choisis aléatoirement, quelles lignes clients ont été modifiées
    dans le dataset d'entraînement. Nécessaire ultérieurement afin d'évaluer la performance via l'AUC.
    '''
    test_set = ratings.copy() # Make a copy of the original set to be the test set.
    test_set[test_set != 0] = 1 # Store the test set as a binary preference matrix
    training_set = ratings.copy() # Make a copy of the original data we can alter as our training set.
    nonzero_inds = training_set.nonzero() # Find the indices in the ratings data where an interaction exists
    nonzero_pairs = list(zip(nonzero_inds[0], nonzero_inds[1])) # Zip these pairs together of user,item index into list
    random.seed(0) # Set the random seed to zero for reproducibility
    num_samples = int(np.ceil(pct_test*len(nonzero_pairs))) # Round the number of samples needed to the nearest integer
    samples = random.sample(nonzero_pairs, num_samples) # Sample a random number of user-item pairs without replacement
    user_inds = [index[0] for index in samples] # Get the user row indices
    item_inds = [index[1] for index in samples] # Get the item column indices
    training_set[user_inds, item_inds] = 0 # Assign all of the randomly chosen user-item pairs to zero
    training_set.eliminate_zeros() # Get rid of zeros in sparse array storage after update to save space
    return training_set, test_set, list(set(user_inds)) # Output the unique list of user rows that were altered
```

In [388]:
```
dfCleanedSDR['CustomerID'] = dfCleanedSDR.CustomerID.astype(int) # Convert to int for customer ID
dfCleanedSDR = dfCleanedSDR[['StockCode', 'Quantity', 'CustomerID']] # Get rid of unnecessary info
grouped_cleaned = dfCleanedSDR.groupby(['CustomerID', 'StockCode']).sum().reset_index() # Group together
grouped_cleaned.Quantity.loc[grouped_cleaned.Quantity == 0] = 1 # Replace a sum of zero purchases with a one to
# indicate purchased
grouped_purchased = grouped_cleaned.query('Quantity > 0') # Only get customers where purchase totals were positive

customers = list(np.sort(grouped_purchased.CustomerID.unique())) # Get our unique customers
products = list(grouped_purchased.StockCode.unique()) # Get our unique products that were purchased
quantity = list(grouped_purchased.Quantity) # All of our purchases

rows = grouped_purchased.CustomerID.astype('category', categories = customers).cat.codes
# Get the associated row indices
cols = grouped_purchased.StockCode.astype('category', categories = products).cat.codes
# Get the associated column indices
purchases_sparse = sparse.csr_matrix((quantity, (rows, cols)), shape=(len(customers), len(products)))
training_set = purchases_sparse.copy()
product_train, product_test, product_users_altered = make_train(purchases_sparse, pct_test = 0.2)
```

In [389]:
```
# les paramètres choisis sont ceux qui ont donné les meilleurs résultats
alpha = 15
user_vecs, item_vecs = implicit.alternating_least_squares((product_train*alpha).astype('double'),
                                                          factors=20,
                                                          regularization = 0.1,
                                                          iterations = 50)
```

In [390]:
```
def auc_score(predictions, test):
    '''
    Fonction qui détermine l'aire se trouvant sous la courbure utilisant la métrique de scikit learn.
    paramètres:
    - predictions: le résultat de la prédiction
    - test: la cible réelle à laquelle comparer les résultats du modèle
    retourne:
    - AUC (aire sous la courbe ROC)
    '''
    fpr, tpr, thresholds = metrics.roc_curve(test, predictions)
    return metrics.auc(fpr, tpr)
```

In [391]:
```
def calc_mean_auc(training_set, altered_users, predictions, test_set):
    '''
    Fonction qui calcule l'AUC moyen par client pour tout client qui a eu sa partie de matrice clients / articles modifiée.
    paramètres:
    training_set - Le dataset d'entraînement produit par make_train, où un certain pourcentage des interactions clients / ar
ticles
    sont mises à 0 afin de les cacher du modèle
    predictions - La matrice des données prédites pour chaque paire client/article.
    altered_users - Les indices des clients où au moins une paire client/article a été modifié via la fonction make_train
    test_set - Le dataset de test construit plus tôt à partir de la fonction make_train
    retourne:
    La moyenne AUC du dataset sur uniquement les interactions client/article.
    '''
    store_auc = [] # An empty list to store the AUC for each user that had an item removed from the training set
    popularity_auc = [] # To store popular AUC scores
    pop_items = np.array(test_set.sum(axis = 0)).reshape(-1) # Get sum of item iteractions to find most popular
    item_vecs = predictions[1]
    for user in altered_users: # Iterate through each user that had an item altered
        training_row = training_set[user,:].toarray().reshape(-1) # Get the training set row
        zero_inds = np.where(training_row == 0) # Find where the interaction had not yet occurred
        # Get the predicted values based on our user/item vectors
        user_vec = predictions[0][user,:]
        pred = user_vec.dot(item_vecs).toarray()[0,zero_inds].reshape(-1)
        # Get only the items that were originally zero
        # Select all ratings from the MF prediction for this user that originally had no iteraction
        actual = test_set[user,:].toarray()[0,zero_inds].reshape(-1)
        # Select the binarized yes/no interaction pairs from the original full data
        # that align with the same pairs in training
        pop = pop_items[zero_inds] # Get the item popularity for our chosen items
        store_auc.append(auc_score(pred, actual)) # Calculate AUC for the given user and store
        popularity_auc.append(auc_score(pop, actual)) # Calculate AUC using most popular and score
    # End users iteration

    return float('%.3f'%np.mean(store_auc)), float('%.3f'%np.mean(popularity_auc))
   # Return the mean AUC rounded to three decimal places for both test and popularity benchmark
```

In [392]:
```
calc_mean_auc(product_train, product_users_altered, [sparse.csr_matrix(user_vecs), sparse.csr_matrix(item_vecs.T)], product_
test)
# AUC for our recommender system
```

Out[392]:
```
(0.87, 0.813)
```

On obtient ainsi une moyenne AUC de 0.87, ce qui est un très bon score.

In [393]:
```
customers_arr = np.array(customers) # Array of customer IDs from the ratings matrix
products_arr = np.array(products) # Array of product IDs from the ratings matrix
```

In [394]:
```
def get_items_purchased(customer_id, mf_train, customers_list, products_list, item_lookup):
    '''
    Fontion qui donne quels articles ont déjà été achetés par un client dans le set d'entraînement.
    paramètres:
    customer_id - Id du client dont on veut voir les précédents achats
    mf_train - La matrice initiale d'entraînement
    customers_list - Le tableau des cients utilisés dans la matrice initiale
    products_list - Le tableau des articles utilisés dans la matrice initiale
    item_lookup - dataframe des couples uniques code produit / description article disponibles
    retourne:
    Une liste des codes et descriptions d'articles d'un client particulier qui ont déjà été achetés dans le dataset d'entraî
nement
    '''
    cust_ind = np.where(customers_list == customer_id)[0][0] # Returns the index row of our customer id
    purchased_ind = mf_train[cust_ind,:].nonzero()[1] # Get column indices of purchased items
    prod_codes = products_list[purchased_ind] # Get the stock codes for our purchased items
    return item_lookup.loc[item_lookup.StockCode.isin(prod_codes)]
```

In [395]:
```
def rec_items(customer_id, mf_train, user_vecs, item_vecs, customer_list, item_list, item_lookup, num_items = 10):
    '''
    Fonction qui retourne les articles les plus recommandés aux clients
    paramètres:
    customer_id - L'id du client dont on veut connaître les recommandations
    mf_train - matrice d'entraînement utilisée pour le fitting de la factorisation de matrice
    user_vecs - le vecteur des clients de la fitted matrix factorization
    item_vecs - le vecteur des articles de la fitted matrix factorization
    customer_list - un tableau des id des clients qui font les lignes de la matrice initiale (dans l'ordre des lignes de la
matrice)
    item_list - un tableau des produits qui font les colonnes de la matrice initiale (dans l'ordre de la matrice)
    item_lookup - dataframe des couples uniques code produit / description article disponibles
    num_items - Le nombre d'articles qu'on veut recommander dans l'optique de meilleures recommandations. Par défaut, 10.
    retourne:
    - Les n meilleures recommandations choisies basées sur les vecteurs client/article pour des articles n'ayant jamais
    été achetés
    ...
    cust_ind = np.where(customer_list == customer_id)[0][0] # Returns the index row of our customer id
    pref_vec = mf_train[cust_ind,:].toarray() # Get the ratings from the training set ratings matrix
    pref_vec = pref_vec.reshape(-1) + 1 # Add 1 to everything, so that items not purchased yet become equal to 1
    pref_vec[pref_vec > 1] = 0 # Make everything already purchased zero
    rec_vector = user_vecs[cust_ind,:].dot(item_vecs.T) # Get dot product of user vector and all item vectors
    # Scale this recommendation vector between 0 and 1
    min_max = MinMaxScaler()
    rec_vector_scaled = min_max.fit_transform(rec_vector.reshape(-1,1))[:,0]
    recommend_vector = pref_vec*rec_vector_scaled
```

```
    # Items already purchased have their recommendation multiplied by zero
    product_idx = np.argsort(recommend_vector)[::-1][:num_items] # Sort the indices of the items into order
    # of best recommendations
    rec_list = [] # start empty list to store items
    for index in product_idx:
        code = item_list[index]
        rec_list.append([code, item_lookup.Description.loc[item_lookup.StockCode == code].iloc[0]])
        # Append our descriptions to the list
    codes = [item[0] for item in rec_list]
    descriptions = [item[1] for item in rec_list]
    final_frame = pd.DataFrame({'StockCode': codes, 'Description': descriptions}) # Create a dataframe
    return final_frame[['StockCode', 'Description']] # Switch order of columns around
```

Testons maintenant les résultats pour le client n°12361:

In [396]:
```
get_items_purchased(12361, product_train, customers_arr, products_arr, item_lookup)
```

Out[396]:

	StockCode	Description
33	22326	ROUND SNACK BOXES SET OF4 WOODLAND
34	22629	SPACEBOY LUNCH BOX
89	20725	LUNCH BAG RED RETROSPOT
354	22382	LUNCH BAG SPACEBOY DESIGN
355	20726	LUNCH BAG WOODLAND
531	22328	ROUND SNACK BOXES SET OF 4 FRUITS
533	22630	DOLLY GIRL LUNCH BOX
34786	20725	LUNCH BAG RED SPOTTY

Le client semble avoir déjà acheté pas mal de lunch box...

In [397]:
```
rec_items(12361, product_train, user_vecs, item_vecs, customers_arr, products_arr, item_lookup, num_items = 10)
```

Out[397]:

	StockCode	Description
0	22662	LUNCH BAG DOLLY GIRL DESIGN
1	22383	LUNCH BAG SUKI DESIGN
2	20727	LUNCH BAG BLACK SKULL.
3	20728	LUNCH BAG CARS BLUE
4	20719	WOODLAND CHARLOTTE BAG
5	23209	LUNCH BAG DOILEY PATTERN
6	21731	RED TOADSTOOL LED NIGHT LIGHT
7	23207	LUNCH BAG ALPHABET DESIGN
8	85099B	JUMBO BAG RED RETROSPOT
9	20724	RED RETROSPOT CHARLOTTE BAG

Il lui est alors recommandé d'autres lunch box.

On peut également tester un client non utilisé dans l'article pour illustrer les bons résultats, par exemple le client 17852:

In [406]:
```
get_items_purchased(17852, product_train, customers_arr, products_arr, item_lookup)
```

Out[406]:

	StockCode	Description
107	22774	RED DRAWER KNOB ACRYLIC EDWARDIAN
229	22150	3 STRIPEY MICE FELTCRAFT
230	22619	SET OF 6 SOLDIER SKITTLES
375	22805	BLUE DRAWER KNOB ACRYLIC EDWARDIAN
387	84380	SET OF 3 BUTTERFLY COOKIE CUTTERS
622	21790	VINTAGE SNAP CARDS
633	22775	PURPLE DRAWERKNOB ACRYLIC EDWARDIAN
672	21892	TRADITIONAL WOODEN CATCH CUP GAME
919	84947	ANTIQUE SILVER TEA GLASS ENGRAVED
1322	22800	ANTIQUE TALL SWIRLGLASS TRINKET POT
1841	21888	BINGO SET
7617	22831	WHITE BROCANTE SOAP DISH
155879	23229	VINTAGE DONKEY TAIL GAME
156316	23029	DRAWER KNOB CRACKLE GLAZE GREEN
156386	23229	DONKEY TAIL GAME
167041	23029	DOORKNOB CRACKED GLAZE GREEN
284524	23490	T-LIGHT HOLDER HANGING LOVE BIRD
295859	23570	TRADITIONAL PICK UP STICKS GAME
296360	23569	TRADTIONAL ALPHABET STAMP SET

On peut maintenant comparer les articles recommandés au client 17852 avec ceux qu'il a déjà achetés:

In [407]:
```
rec_items(17852, product_train, user_vecs, item_vecs, customers_arr, products_arr, item_lookup, num_items = 10)
```

Out[407]:

	StockCode	Description
0	84832	ZINC WILLIE WINKIE CANDLE STICK
1	21326	AGED GLASS SILVER T-LIGHT HOLDER
2	22083	PAPER CHAIN KIT RETROSPOT
3	23084	RABBIT NIGHT LIGHT
4	22087	PAPER BUNTING WHITE LACE
5	84949	SILVER HANGING T-LIGHT HOLDER
6	71459	HANGING JAM JAR T-LIGHT HOLDER
7	23194	GYMKHANNA TREASURE BOOK BOX
8	84945	MULTI COLOUR SILVER T-LIGHT HOLDER
9	47566	PARTY BUNTING

On retrouve bien des articles assez proches des achats déjà effectués, comme toute la panoplie des "T-LIGHT HOLDER".

5- API RESTFULL sur AWS et tests

L'API consiste en un service REST / Flask hébergé sur AWS (Amazon Web Service) appelé par HTTP et qui rend le résultat sous format Json.

Comme une requête HTTP tombe rapidement en timeout, je n'ai pas intégré dans le code de l'API tout le nettoyage possible car cela prend beaucoup trop longtemps (plus de 10 minutes). J'ai donc juste supprimé tous les enregistrements (cf. fonction OCRDSP5fonc.initialisation) qui :

- n'avait pas de numéro de client
- qui avaient des code article ne correspondant pas à de véritables articles
- qui avaient des quantités négatives d'article (remboursements)

Pour mieux faire, il aurait fallu supprimer les articles achetés correspondant à ces remboursements, mais cela prend beaucoup de temps d'exécution.

Lors de mes essais, une requête prend 20 secondes.

Remaques importantes:

Le fichier de données utilisé est le même utilisé dans ces notebooks, ce n'est donc pas le fichier téléchargé Online Retail.xlsx, mais sa version CSV OnlineRetail.csv. Il y a donc exactement les mêmes données, c'est juste le format du fichier qui change.

Il faut absolument entrer les données dans le bon ordre, dans le bon format et que le client possède bien des factures dans la période donnée, sinon il sera retourné une erreur:

- identifiant du client
- date du début (incluse) de la séquence temporelle : format DDMMYYYY
- date du fin (incluse) de la séquence temporelle : format DDMMYYYY

Les trois données doivent être concaténées et séparées par un ":".
Exemple :

http://127.0.0.1:5000/projet5/category/17850:01122010:09122011

Avec cette URL, on obtient le résultat:


```
{
    "Date debut": "01-12-2010",
    "id client": "17850",
    "Amount": 158.56500000000003,
    "Date fin": "09-12-2011",
    "Recency": 372,
    "Cluster": "7",
    "Categorie": "Frequent",
    "Frequency": 34,
    "Classe RFM": "141"
}
```


Avec le client 14849 entre le 06/06/2011 et le 01/11/2011, on a un client BEST: http://127.0.0.1:5000/projet5/category/14849:06062011:01112011


```
{
  "Date debut": "06-06-2011",
  "id client": "14849",
  "Amount": 461.88555555555547,
  "Date fin": "01-11-2011",
  "Recency": 26,
  "Cluster": "4",
  "Categorie": "Best",
  "Frequency": 9,
  "Classe RFM": "343"
}
```


Il a une bonne fréquence sur cette courte période, une récence faible et un bon montant, de quoi le classer en 343 parmi les meilleurs des clients.

6- Conclusions

- La segmentation RFM a permis de classer les clients dans différentes catégories afin de mettre en valeur différents types de comportement nécessitant le cas échéant d'avoir recours à des stratégies commerciales ciblées.
- Ces catégories ne sont pas absolues, elles dépendent de l'ensemble des clients ainsi que de la période étudiée.
- Cela a comme inconvéniant de rendre difficile la comparaison et l'évolution de ces catégories dans le temps puisque les frontières définissant ces catégories dépendent de l'ensemble des clients et non du seul client dont on cherche la catégorie. Mais d'un point de vue inverse, cela a par contre l'avantage de relativiser le comportement d'un client à l'ensemble des autres clients. En effet, on peut se demander par exemple pourquoi un client ne change pas de comportement d'achat quand la presque totalité des autres clients en changerait. L'étude du comportement d'achat d'un client doit donc s'accompagner de l'étude du comportement général.
- La définition des catégories dépent aussi de la statistique utilisée (moyenne, médiane etc.). Le choix doit s'effectuer en fonction du contexte du jeu de donnée.
- On a donc obtenu 64 classes RFM (de 111 à 444) qu'une méthode KMean a permis de regrouper en 8 clusters.
- Chaque cluster et chaque client s'est vu attribuer enfin une catégorie définie par rapport à la moyenne générale des clients.
- L'équipe marketing peut alors employer différentes stratégies en fonction de ces catégories afin d'augmenter le chiffre d'affaire de l'entreprise. Par exemple, les clients "churn" qu'on cherche à faire revenir, les clients "First Time" que l'on veut fidéliser et pousser à acheter plus, les clients Best dont on veut entretenir la flamme, etc.
- La matrice de transition d'une catégorie à une autre permet de voir des tendances, qui, quand elles sont négatives pour l'entreprise, nécessitent des actions ciblées. On peut par exemple chercher à comprendre pourquoi presque la moitiée des clients Best de la première période (les 6 premiers mois) est passée dans la catégorie Shopper, c'est-à-dire que, s'ils ont continué à venir aussi fréquemment, ils ont baissé le montant moyen de leurs paniers. De le même façon, on peut s'interroger sur la raison pour laquelle presque 1% de ces clients Best sont passés dans Uncertain, même si cela suggère que ce sont des clients qui ont tout simplement arrêtés de venir (puisque la seconde période contient la première, donc les valeurs ne se sont pas annulées).
- Quand on observe les frontières entre catégories, on s'aperçoit qu'il peut y avoir des différences importantes de valeurs au sein d'une même catégorie. C'est le cas par exmemple pour les "extrêmes", où une différenciation entre "Best" et "Super Best", ou "Uncertain" et "Super Uncertain" pourrait être utile. Il pourrait alors être intéressant d'affiner ces catégories en augmentant leur nombre, même si pour certaines, cela n'a pas d'intérêt particulier. Il s'agit donc de trouver le bon compromis dans le nombre de catégories afin de ne pas en avoir trop et chercher des stratégies inutiles, mais d'en avoir suffisamment pour bien délimiter les comportements remarquables et les traiter.
- La CLV, Custumer Lifetime Value, a permis de commencer à étudier la valeur commerciale des clients, en s'intéressant par exemple à la "duréee de vie" d'un client, la probabilité que le client soit toujours "commercialement" en vie, quelles chances il y a de le voir de nouveau acheter et combien dans une prochaine unité de temps. On a par exemple utilisé un algorithme qui montrait que la probabilité d'achat futur remontait à 1 à chaque nouvel achat puis diminuait avec le temps passant sans achat, mais avec taux de décroissance moins élevé au fur et à mesure que la fréquence d'achat augmentait.
- Enfin, une des stratégies de fidélisation d'un client peut être de lui proposer, de lui recommander des articles correspondant à ses goûts en fonction des achats qu'il a déjà effectués. Cela s'avère une stratégie assez payante pour nombre d'entreprises.