!!! N.B. : Toutes les informations ci-dessous sont extraites à partir du fichier fourni dans le livrable et datant du 09/03/2017.
Aujourd'hui, le fichier a plus que doublé de volume, on est passé d'un peu moins de 140 000 enregistrements à plus de 326 000 grâce à l'apport d'enregistrements principalement américains.

Projet n°2, rapport d'exploration

(basé sur les notebooks Projet2DataCleaning et Projet2Exploration)

Ce projet consiste à analyser un jeu de données regroupant des informations sur des articles alimentaires vendus dans le monde. Il provient d'une base de données de http://fr.openfoodfacts.org/data. Les variables présentes sont de différentes natures, on y trouve :

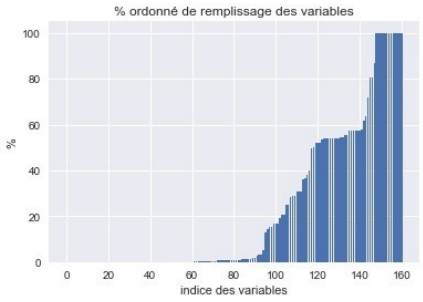
- des textes comme le nom du créateur de l'enregistrement ou le nom de l'article, le packaging
- d'autres textes qui correspondent à des catégories qui indiquent plus globalement à quoi correspond un article
- des url où on peut trouver la description d'un article, son image
- des dates de création et de mise à jour des enregistrements
- des données numériques (qu'on appelera variables continues) représentant principalement la composition chimique des articles

Remarque préalable : on peut noter dès la gestion des données problématiques que le feature engineering peut se faire "à tout moment", dès qu'une technique statistique permet de repérer des possibilités de choix. Cela peut se faire dès l'étude et la description des données remarquables, et comme nous le verrons plus tard, au moment de l'analyse multivariée. Il conviendra à la fin de l'étude du jeu de reprendre les informations collectées par le feature engineering et de s'en servir comme base pour le futur algorithme.

1- Gestion des données manquantes et aberrantes

Nous avons vu dans le notebook de data cleaning que le jeu de donnée contenait énormément de données manquantes surtout parmi les variables des ingrédients. Par exemple, 87 des 98 variables des ingrédients sont remplies à moins de 4%. Certaines variables ne sont même pas du tout remplies.

In [82]:
```
grapheRemplissageVariables(dfOriginel)
```

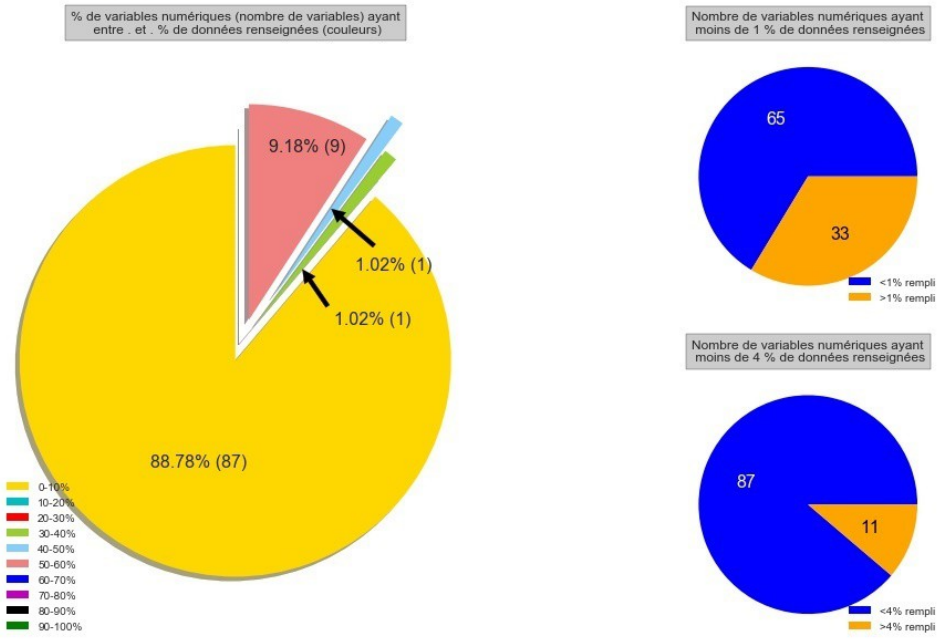


Comme les variables vides n'apportent aucune information, on peut ne pas en tenir compte afin de diminuer la complexité du jeu. On peut s'interroger sur la pertinence des variables ne possédant que très peu de données. Mais en l'absence de modèle possédant une métrique de performance permettant de comparer l'influence des différents seuils, il semble arbitraire de choisir un seuil de suppression de variable à ce point d'étude du jeu de données.

On peut voir que parmi les variables numériques, le remplissage est très limité :

In [83]:
```
grapheRemplissageVariablesNumeriques(dfOriginel)
```

```
<matplotlib.figure.Figure at 0xdda58f28>
```

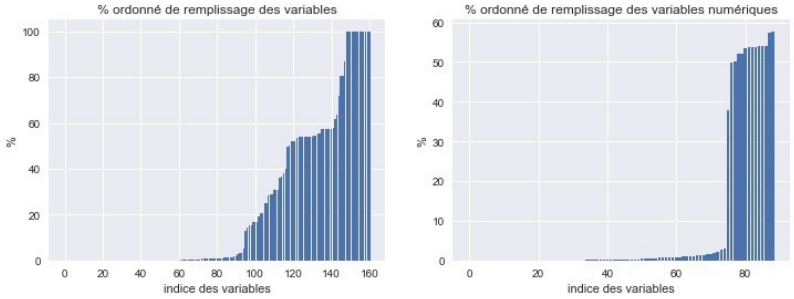


Ainsi, près de 89% des variables numériques sont remplies à moins de 10%. Et sur les 98 variables, 87 sont remplies à moins de 4% et même 65 ne rassemblent même pas 1% de données.

La comparaison des graphes de remplissage montre bien que ce sont surtout les variables numériques qui sont peu remplies, celles-ci ne dépassant pas les 60% :

In [84]:
```
grapheComparaisonNanVariables(dfOriginel)
```

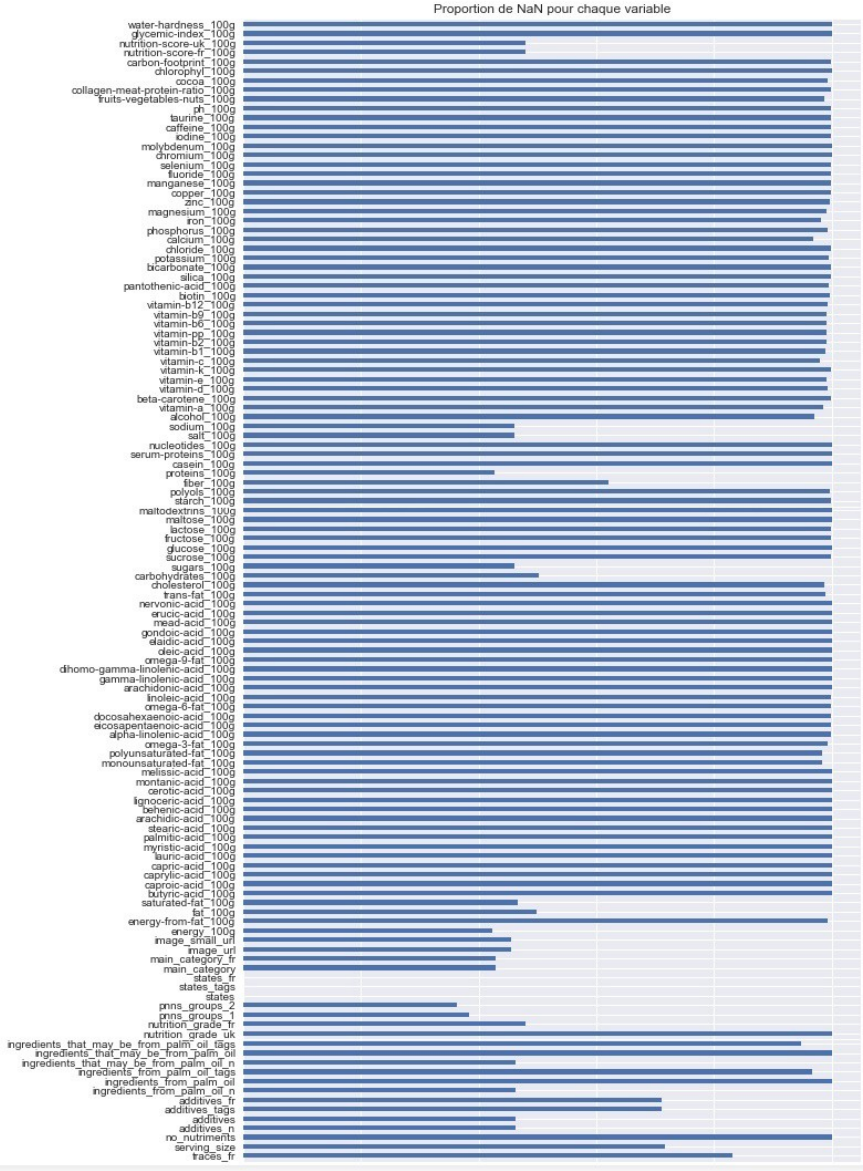
```
<matplotlib.figure.Figure at 0x12ea454e0>
```



Ce graphique permet de mieux voir le remplissage de chaque variable :

In [85]:
```
plt.figure(figsize=(10, 25))
dfOriginel.isnull().mean(axis=0).plot.barh()
plt.title("Proportion de NaN pour chaque variable")
```

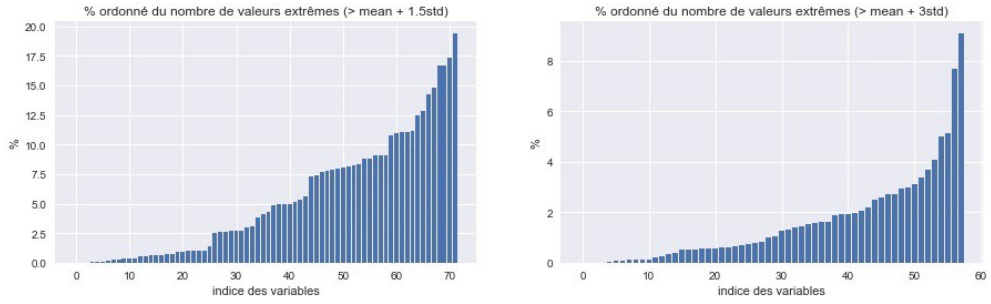
Out[85]:
```
<matplotlib.text.Text at 0xbb82fe80>
```



Valeurs aberrantes : Dans les valeurs aberrantes, on a pu surtout repérer grâce aux statistiques univariées les valeurs surprenantes et sans réel sens pour trois variables de l'enregistrement de code 15666666666 (voir le boxplot "Boxplot de la variable proteins_100g avant annulation de la valeur aberrante" ci-dessous). Puisque les autres variables de cet enregistrement n'étaient pas touchées par ses aberrations, il n'était pas nécessaire de supprimer l'enregistrement dans son ensemble afin de préserver l'intérêt de ses autres informations. Les valeurs aberrantes ont donc été remplacées par la valeur choisie pour remplacer également les données manquantes, c'est-à-dire 0. Nous avons également vu grâce à l'ACP ou aux boxplots que certains enregistrements avaient des valeurs très fortes pour certaines variables particulières. Toutefois, si ces valeurs pouvaient être remarquées, elles ne semblaient pas modifier le comportement général du jeu (par exemple, en annulant les 3 variables du 15666666666, les valeurs marquantes du 3401528535864 ne rendaient pas le clustering inutile).
Valeurs manquantes : Comme beaucoup des implémentations d'algorithme de Machine Learning ne fonctionnent pas avec des datasets contenant des données manquantes (par exemple l'ACP), il est nécessaire de remplacer les valeurs manquantes. Il y a plusieurs possibilités pour le faire. Par exemple, une combinaison linéaire pondérée des observations, comme la moyenne. Ou utiliser la méthode des plus proches voisins, ou des méthodes aléatoires. Mais on peut se demander si affecter une valeur non nulle, par exemple aux ingrédients d'un article en se basant sur d'autres articles qui, a priori, n'ont rien à voir, ne semble pas supprimer toutes les spécificités de ces articles. En outre, il faut tenir compte de la façon dont sont renseignées les informations. Il s'agit d'un site non professionnel ou n'importe qui peut renseigner les variables pour n'importe quel article. Sans connaissance préalable de la façon d'entrer les informations, on peut supposer qu'un utilisateur ne renseignera généralement que les informations non nulles qu'il possède, et omettra d'inscrire 0 pour les informations spécifiées à 0. Il est en effet difficile de distinguer dans un article, un composant dont la valeur est explicitement renseignée comme étant 0 (par exemple "pas d'OGM") d'un composant dont la valeur est implicitement 0 du fait qu'il n'a pas lieu d'être dans cet article. On peut donc dans ce cas particulier de renseignement des ingrédients, remplacer les valeurs manquantes par 0.
Valeurs en double : Même si des codes sont apparus comme manquant ou en double, les informations des enregistrements concernés n'étaient pas en doublon, rien ne permettant de choisir non arbitrairement quels enregistrements garder aux détriments des autres.
Valeurs extrêmes Comme on peut le voir dans les graphes ci-dessous, que ce soit à mean + 1.5std ou à mean + 3std, le pourcentage de valeurs dites "extrêmes" est assez important. Il ne s'agit donc pas de valeurs potentiellement erronées, le jeu de donnée ne regroupe pas que des catégories spécifiques d'articles mais la totalité de ce qui peut exister dans toutes les catégories d'aliments possibles, d'où la possibilité de grandes différences entre les données. Il n'y a pas de raison de les supprimer.

In [86]:
```
comparaisonGraphesExtremes(dfOriginel)
```

```
<matplotlib.figure.Figure at 0xc7342668>
```



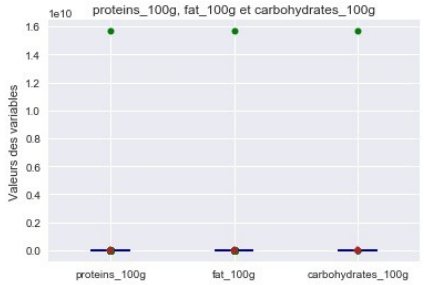
2- Analyse univariée

(La description et l'analyse univariée des différentes variables importantes avec les visualisations associées)

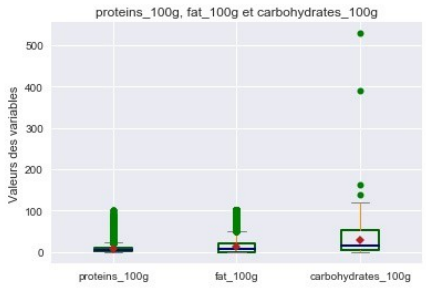
Les différentes statistiques univariées (quantiles, écart-types etc.) ont pu permettre de déceler des valeurs aberrantes (comme décrit précédemment). Le boxplot permet de visualiser l'évolution.

In [87]:
```
showChangeMeanProteins(dfOriginel)
```

```
<matplotlib.figure.Figure at 0x86a44b70>
```



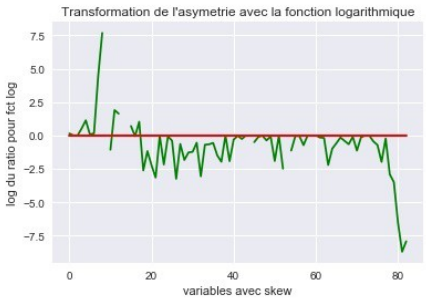

```
Moyenne d'avant annulation de la valeur aberrante : proteins_100g         196029.545926
fat_100g              223819.805820
carbohydrates_100g    225392.673923
dtype: float64

<matplotlib.figure.Figure at 0xc15a6a58>
```

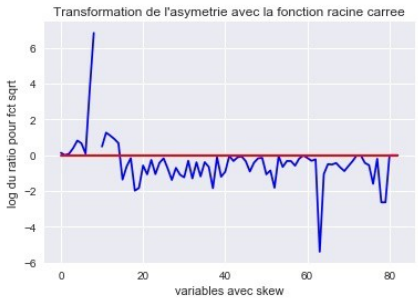



```
Moyenne d'apres annulation de la valeur aberrante : proteins_100g          7.549755
fat_100g              13.488768
carbohydrates_100g    28.144384
dtype: float64
```

Dans les modèles de régression linéaire, il est préférable pour une variable continue de tendre vers une loi normale. On peut alors tenter de diminuer autant que possible la valeur d'asymétrie (skewness) qui lorsqu'elle est trop forte, pose des problèmes. Nous avons vu que pour la plupart des variables d'asymétrie marquée, utiliser une fonction comme le logarithme ou la racine carrée permettait de diminuer, parfois de façon importante, cette asymétrie, comme en témoigne les graphiques suivants :

```
In [88]: dfContinuesSkewed = graphesDiminutionAsymetrie(dfOriginel)
```

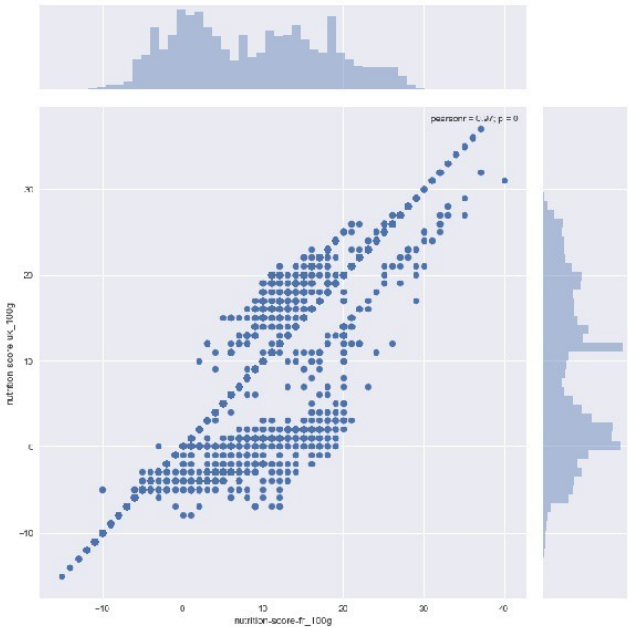




3- Analyse multivariée

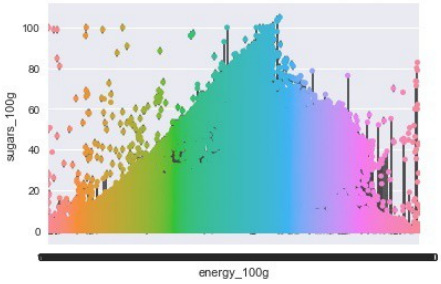
Avant de voir les corrélations de toutes les variables entre elles visualisable grâce à une heatmap, on peut pour deux variables particulières voir comment elles se présentent l'une vis-à-vis de l'autre. On peut prendre par exemple les variables nutrition-score-fr_100g et nutrition-score-uk_100g qui sont très proches l'une de l'autre, nutrition-score-fr_100g étant une adaptation de nutrition-score-uk_100g au marché français :

```
In [89]: sns.jointplot(x="nutrition-score-fr_100g", y="nutrition-score-uk_100g", data=dfOriginel, size=10)
Out[89]: <seaborn.axisgrid.JointGrid at 0xb952cc50>
```



La corrélation est presque parfaite (droite y=x), les autres points dénotant les adaptations au marché français. On peut également visualiser la relation entre les variables energy_100g et sugars_100g :

In [90]:
```
ax=sns.boxplot(x="energy_100g",y="sugars_100g",data=dfOriginel)
ax=sns.stripplot(x="energy_100g",y="sugars_100g",data=dfOriginel,jitter=True)
```

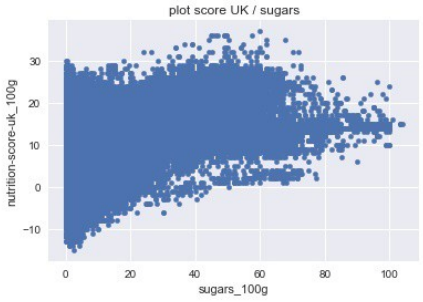


Globalement, l'augmentation du taux de sucre s'accompagne de celle de l'énergie. Bien sûr, l'énergie totale de l'article n'étant pas entièrement comprise dans son ingrédient sucres, on voit que d'autres facteurs (ingrédients) entrent en jeu, qui, pour un même taux de sucre, font varier l'énergie de l'article.

On peut voir pour les 7 variables composant le score, l'influence de chacune sur le score:

In [91]:
```
dfOriginel.plot(kind="scatter",x="sugars_100g", y="nutrition-score-uk_100g",title="plot score UK / sugars", legend=True)
```

Out[91]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x304213c8>
```

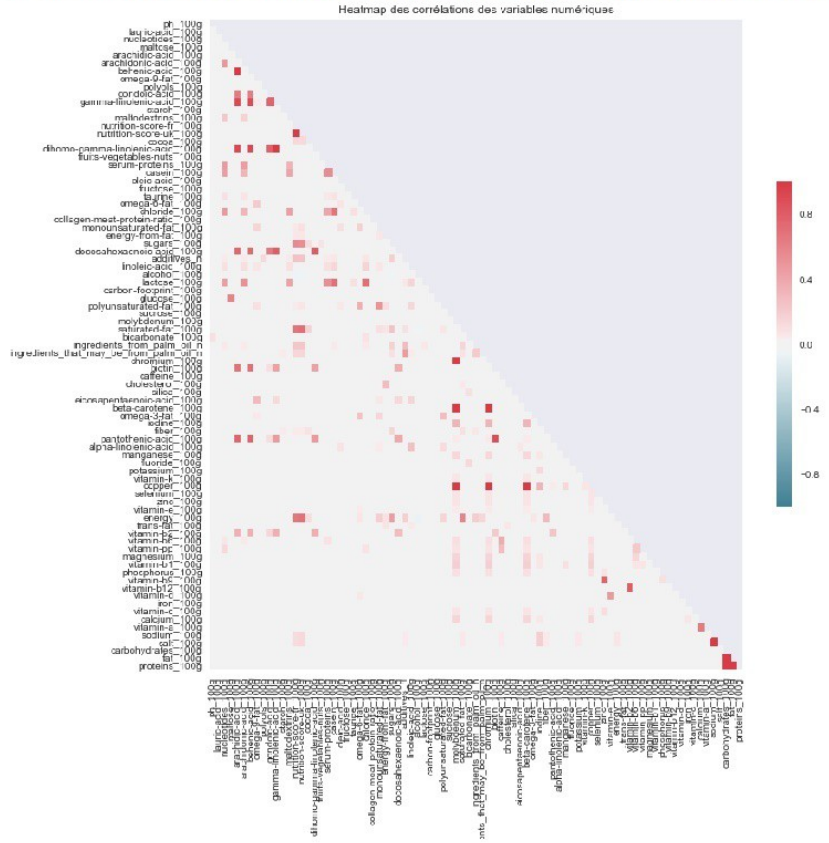


Par exemple, on voit ici nettement qu'à des taux de sucres élevés sont associés des scores élevés, même si d'autres facteurs jouent (on peut avoir des scores élevés même avec des taux en sucres faibles.

Heatmap

Afin de voir visuellement les corrélations entre les variables, on peut utiliser une heatmap.

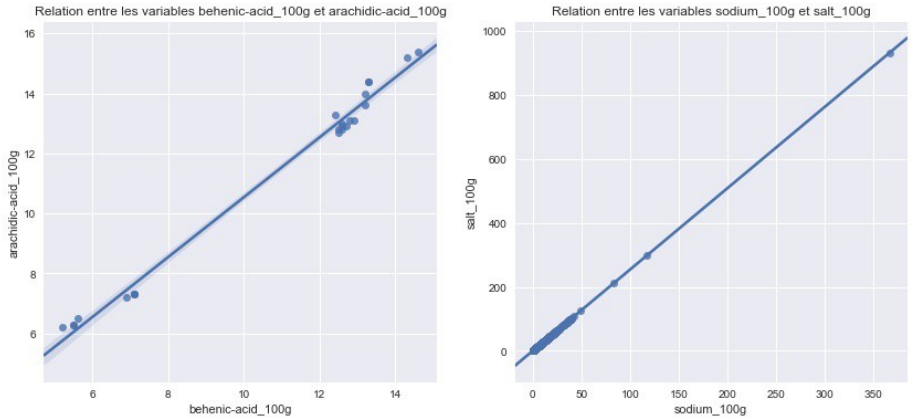
In [92]:
```
dfContinuesSkewedSansNaN = dfContinuesSkewed.fillna(0)
heatmap(dfContinuesSkewedSansNaN, "pearson", u"Heatmap des corrélations des variables numériques")
```



La heatmap de la matrice de corrélation permet de mettre en exergue les fortes corrélations, ce qui peut être utile pour le feature engineering. En effet, dans ce jeu de données, on a des corrélations très proches de 1 et même égales à 1 pour le cas entre les variables salt_100g et sodium_100g (carré rouge vif en bas à droite de la heatmap).

In [93]:
```
graphesComparatifCorrelations(dfContinuesSkewedSansNaN, dfOriginel)
```

```
                     behenic-acid_100g
arachidic-acid_100g           0.999485
                sodium_100g
sodium_100g             1.0

<matplotlib.figure.Figure at 0x191c2390>
```



On voit ainsi que même avec une corrélation très proche de 1 comme entre l'acide béhénique et l'acide arachidique (0.999485), on n'a pas une relation linéaire parfaite. Supprimer une des deux variables reviendrait à supprimer de l'information non redondante. Par coontre, la corrélation à 1 entre les variables sodium_100g et salt_100g est le résultat de l'existence d'un coefficient de multiplication entre les deux (environ 2.4). Ces deux variables sont donc redondantes (le sel n'est finalement que du chlorure de sodium) et en supprimer une permet de diminuer la complexité du dataset, son nombre de dimensions. Comme la variable sodium_100g fait partie des variables utilisées pour le calcul des scores nutritionnels (nutrition-score-uk_100g), il est préférable de la garder.

ACP

On retrouve dans cette heatmap les variables des premières composantes principales d'une ACP. On voit clairement que :

- les variables de la première composante, arachidic-acid_100g, behenic-acid_100g, gamma-linolenic-acid_100g etc. sont fortement corrélées entre elles
- les variables de la deuxième composante, beta-carotene_100g, copper_100g, chromium_100g et molybdenum_100g le sont également, toutefois, comme on l'a vu lors de l'exploration, les corrélations de ces variables dans la deuxième composante sont essentiellement dues à un seul enregistrement (de code 3401528535864) dont les valeurs sont très élevées pour ces variables. Si on supprimme cet enregistrement, ce groupe de corrélations perd en vigueur et les variables "retombent" dans des composantes de rang moins elevé (visualisable plus loin dans la comparaison des heatmaps avec et sans cet enregistrement).
- les variables de la troisième composante, nutrition-score-fr_100g, energy_100g, nutrition-score-uk_100g etc. sont également très corrélées entre elles, même si dans une moindre mesure que pour la première composante. Avec la suppression de l'enregistrement mentionné pour la deuxième composante, ce groupe de variables montent en deuxième composante.
- les variables des deux composantes principales ne sont quasiment pas corrélées entre elles (indépendance des composantes principales)

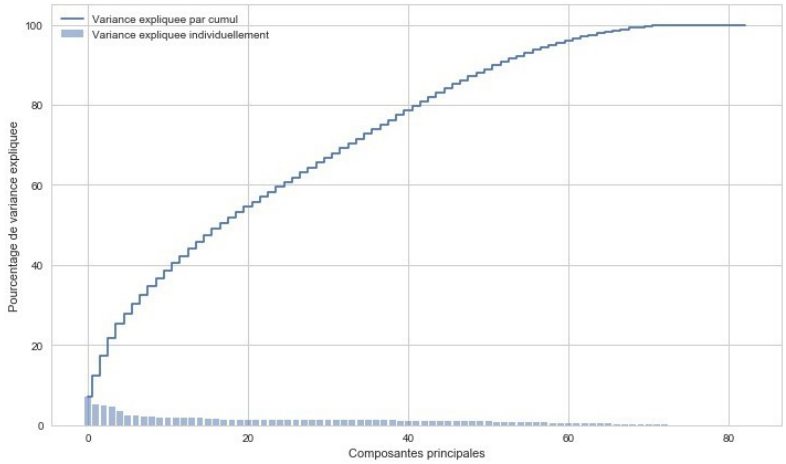
In [94]:
```
dfContinuesSkewedSansNaNNormalise = ACP(dfContinuesSkewedSansNaN)
```

```
Variables de la première composante principale de l'ACP :
                          features     coeff  importances
4                arachidic-acid_100g  0.402990           13
6                  behenic-acid_100g  0.402106           13
10         gamma-linolenic-acid_100g  0.384934           12
16  dihomo-gamma-linolenic-acid_100g  0.386360           12
29         docosahexaenoic-acid_100g  0.335398           11
44                       biotin_100g  0.268317            9
9                  gondoic-acid_100g  0.283268            9
53             pantothenic-acid_100g  0.287494            9
65                   vitamin-b2_100g  0.170795            5
55                    manganese_100g  0.001447            0


Variables de la deuxième composante principale de l'ACP :
               features     coeff  importances
49  beta-carotene_100g  0.464612           12
59         copper_100g  0.464822           12
43       chromium_100g  0.464907           12
38     molybdenum_100g  0.465019           12
51         iodine_100g  0.201465            5
69     vitamin-b1_100g  0.121249            3
66     vitamin-b6_100g  0.080126            2
76        calcium_100g  0.096750            2
68      magnesium_100g  0.100301            2
55      manganese_100g  0.087591            2
```

```
Variables de la troisième composante principale de l'ACP :
                    features     coeff  importances
14  nutrition-score-uk_100g  0.322105            8
63              energy_100g  0.284031            7
13  nutrition-score-fr_100g  0.319537            7
19              casein_100g  0.291174            7
33             lactose_100g  0.307142            7
24            chloride_100g  0.286812            7
2           nucleotides_100g  0.243560            6
18      serum-proteins_100g  0.254630            6
39       saturated-fat_100g  0.255698            6
28              sugars_100g  0.222154            5
```

Comme nous pouvons le voir dans les graphiques ci-dessous, aucune des composantes principales n'est réellement majoritaire. Les quatre premières composantes permettent toutefois d'expliquer un peu plus de 20% de la variance du jeu de données.

In [95]:
```
eig_pairs = getPropre(dfContinuesSkewedSansNaNNormalise, True)
eig_pairsSorted = sorted(eig_pairs, key=lambda x:x[0], reverse=True)
for i in range(7):
    print("{0}\t{1}\t{2}\t{3}\t{4}".format(eig_pairsSorted[5*i][0], eig_pairsSorted[5*i+1][0], eig_pairsSorted[5*i+2][0], eig
_pairsSorted[5*i+3][0], eig_pairsSorted[5*i+4][0]))
```



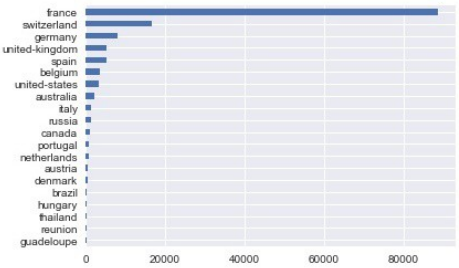

```
5.98281087009   4.35315750769   4.03915124085   3.72989994022   2.99995531598
2.08681279089   2.01946044607   1.78972783488   1.72789260142   1.62935869474
1.60018349291   1.58580057712   1.51730075127   1.51116965865   1.47762098041
1.34442842099   1.30401372271   1.19482814775   1.18684946878   1.12140038239
1.04192409428   1.03502703245   1.02542195427   1.02003541953   1.01492292693
1.01176013706   1.00942021661   1.00353029966   1.00342040861   1.00073854454
0.998730050211  0.997717999594  0.996414846694  0.994744301399  0.994227269508
```

La règle de Kaiser invite à retenir toutes les composantes principales dont la valeur propre est supérieure à 1 puisqu'elles contiennent alors plus d'information que les variables initiales. Il y a cependant dans le cas présent, 30 valeurs propres supérieures à 1.
La "règle du coude" prône de ne prendre que les composantes situées avant le décrochement dans le graphique. Dans notre cas, il n'y a pas de décrochement net. La différence entre les composantes 4 / 5 et 5 / 6 sont assez proches. Toutefois, au-delà de la 5ème composante, les valeurs propres sont assez proches. On peut donc penser qu'il faille s'intéresser principalement aux 5 premières composantes principales.

Le graphe de l'ACP a été couplé avec une variable prédictive, une variable construite à partir de la variable existante "countries_tags" qui indique dans quels pays l'article est vendu. Il peut être ainsi intéressant de voir si certains articles avec leurs compositions spécifiques sont plus vendus en France ou ailleurs (on pourrait créer une variable pour chaque pays). Cela peut renseigner sur les habitudes alimentaires des différents pays, dont la France.

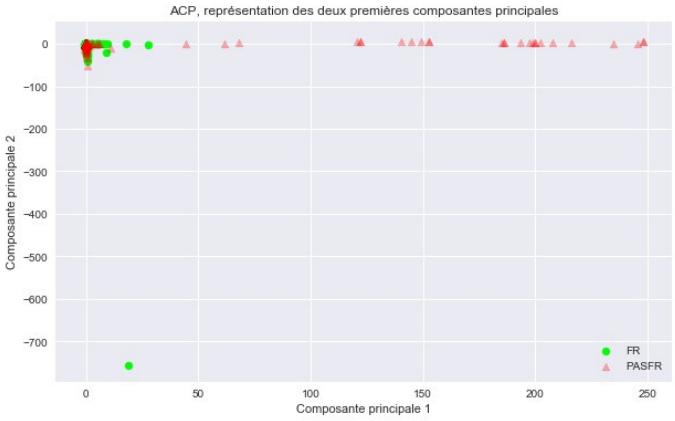
On peut déjà voir qu'une grande majorité des articles renseignés dans ce dataset est vendue en France. (Ce n'est plus le cas dans un dataset récent avec un afflux très important de données américaines.)

In [96]:
```
graphePaysRepresentation(dfOriginel)
```



In [97]:
```
grapheACP(dfContinuesSkewedSansNaNNormalise, dfOriginel)
```

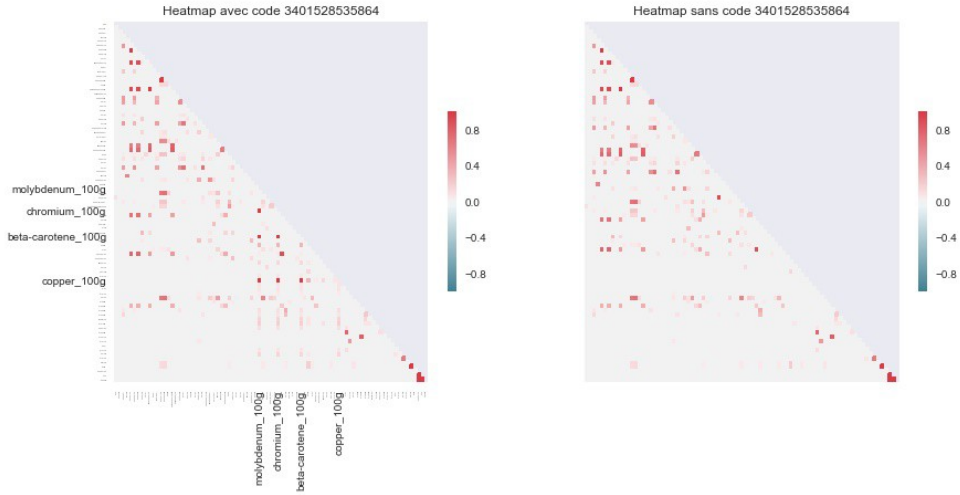
```
<matplotlib.figure.Figure at 0x116e1c9b0>
```



Afin de mieux visualiser la distribution des articles suivant leur lieu de vente, on va supprimer l'unique enregistrement qui est à l'origine de la forte valeur de la composante principale 2. Il s'agit de l'enregistrement de code 3401528535864 qui possède des valeurs très éloignées des moyennes.

In [98]:
```
grapheComparatifHeatmap(dfOriginel)
```

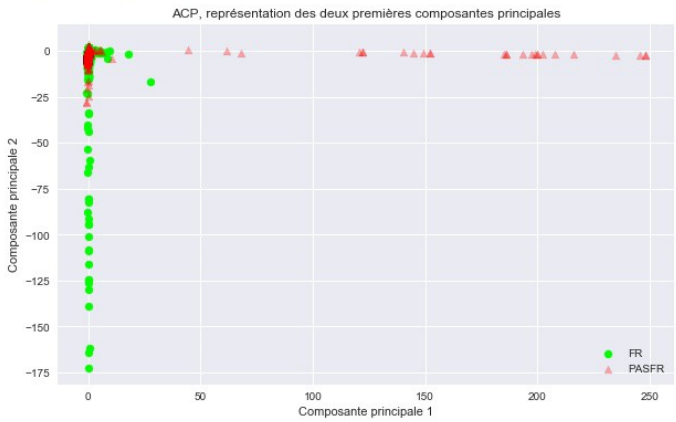
```
<matplotlib.figure.Figure at 0x74fb9198>
```



On observe bien que le groupe des fortes corrélations au centre de la heatmap correspondant à la composante principale 2, a disparu.

In [99]:
```
dfOriginelSans3401528535864 = dfOriginel[dfOriginel.code <> 3401528535864]
dfContinuesSkewedSans3401528535864, dfContinuesSans3401528535864, dfNonContinuesSans3401528535864 = getDfContinuesSkewed(dfOr
iginelSans3401528535864)
dfContinuesSkewedSansNaNSans3401528535864 = dfContinuesSkewedSans3401528535864.fillna(0)
dfContinuesSkewedSansNaNSans3401528535864Normalise = prep.StandardScaler().fit_transform(dfContinuesSkewedSansNaNSans34015285
35864)
grapheACP(dfContinuesSkewedSansNaNSans3401528535864Normalise, dfOriginelSans3401528535864)
```

```
<matplotlib.figure.Figure at 0xce0e2d68>
```



On peut ainsi remarquer la présence de 3 groupes d'enregistrements :

- le premier groupe est constitué presque exclusivement de produits vendus en France et se trouve le long de l'axe de la composante principale 2, elle correspond à un groupe de variables ayant des valeurs élevées sur les variables nutrition-score-fr_100g, energy_100g, fat_100g etc. donc des produits assez riches énergétiquement et peu sains selon le critère nutritif (score élevé). (A été supprimé l'enregistrement qui à lui seul mettait les oligo-éléments cuivre, molybdène etc. dans la deuxième composante.)
- le deuxième groupe est constitué presque exclusivement de produits vendus hors de France et se trouve à peu près sur l'axe de la composante principale 1. Il s'agit de produits contenant certains acides gras essentiels dont quelques-uns ne sont pas synthétisés par le corps humain et doivent provenir d'une alimentation extérieure.
- le troisième groupe est consitué de produits vendus à la fois en France et hors de France, ils sont moins riches et la série des acides gras n'est pas leur constitution principale. Une étude sur les compososantes principales 3, 4 etc. permettrait de voir si d'autres groupes d'éléments en ressortent.

Clustering

Puisque l'ACP a permis une réduction du nombre de dimensions, et qu'elle a mis en évidence 3 groupes d'enregistrements, on peut utiliser une méthode de clustering afin de séparer au mieux ces groupes. Voici la liste des possibilités :

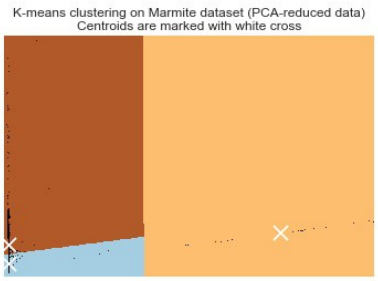
Method name	Parameters	Scalability	Usecase	Geometry (metric used)
K-Means	number of clusters	Very large n_samples, medium n_clusters with MiniBatch code	General-purpose, even cluster size, flat geometry, not too many clusters	Distances between points
Affinity propagation	damping, sample preference	Not scalable with n_samples	Many clusters, uneven cluster size, non-flat geometry	Graph distance (e.g. nearest-neighbor graph)
Mean-shift	bandwidth	Not scalable with n_samples	Many clusters, uneven cluster size, non-flat geometry	Distances between points
Spectral clustering	number of clusters	Medium n_samples, small n_clusters	Few clusters, even cluster size, non-flat geometry	Graph distance (e.g. nearest-neighbor graph)
Ward hierarchical clustering	number of clusters	Large n_samples and n_clusters	Many clusters, possibly connectivity constraints	Distances between points
Agglomerative clustering	number of clusters, linkage type, distance	Large n_samples and n_clusters	Many clusters, possibly connectivity constraints, non Euclidean distances	Any pairwise distance
DBSCAN	neighborhood size	Very large n_samples, medium n_clusters	Non-flat geometry, uneven cluster sizes	Distances between nearest points
Gaussian mixtures	many	Not scalable	Flat geometry, good for density estimation	Mahalanobis distances to centers
Birch	branching factor, threshold, optional global clusterer.	Large n_clusters and n_samples	Large dataset, outlier removal, data reduction.	Euclidean distance between points

http://scikit-learn.org/stable/modules/clustering.html#k-means

Voici ce que donne le k-means sur les données de l'ACP :

In [100]:
```
dfContinuesSkewedSansNaNSansAberrNormalise = getDfContinuesSkewedSansNaNSansAberrNormalise(dfOriginel)
predict_lieu_vente = np.where(dfOriginel["countries_tags"].str.contains("en:france|en:french|en:new-caledonia|en:guadeloupe|en:martinique|en:reunion|en:saint-pierre-and-miquelon")==True, "FR", "PASFR")
graphe_KMeans(dfContinuesSkewedSansNaNSansAberrNormalise, predict_lieu_vente, 3)
```

```
('centroids = ', array([[  1.80868974e-02,  -1.13402889e+00],
       [  1.84789463e+02,   5.91695059e+00],
       [ -1.70019470e-01,   3.25669978e+00]]))
```




```
Temps d'exécution du KMeans : 1.25417105843
Temps d'exécution graphe : 0.432231559718
Inertie du K-means : 215472.693379
```

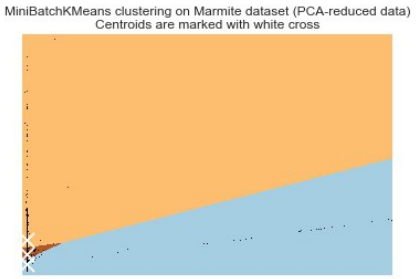
On voit bien la séparation des trois groupes avec les trois centroïdes.

Le désavantage du k-means est qu'il utilise la totalité des enregistrements, ce qui peut faire baisser la performance en temps avec les grosses bases de données. On peut alors utiliser une méthode dérivée du k-means, le MiniBatchKMeans qui utilise des petits groupes aléatoires d'enregistrements qui changent à chaque itération jusqu'à convergence. Mais s'il y a gain de temps, il y a perte de qualité des clusters.

Voici ce que cela donne avec la méthode de clustering MiniBatchKMeans :

In [101]:
```
graphe_MiniBatchKMeans(dfContinuesSkewedSansNaNSansAberrNormalise, 3)
```

```
('centroids = ', array([[ 0.25165978, -1.46277847],
       [-0.20217865,  3.88166705],
       [-0.06449442,  0.64246895]]))
```




```
Temps d'exécution du MiniBatchKMeans : 0.737500651851
Temps d'exécution graphe : 1.46079845073
Inertie du miniBatchKmeans : 931369.778835
```

On voit donc que la méthode MiniBatchKMeans est effectivement plus rapide que le KMeans, mais qu'il y a bien une perte de qualité des clusters, l'inertie du MiniBatchKMeans est bien plus élevée que celle du k-means.

4- Feature engineering

Pour résumer, nous avons donc :

- supprimé (ou du moins non tenu compte) des variables aucunement renseignées (partie 1)
- séparé variables numériques (ou continues) et variables non numériques (ou non continues) (partie 1)
- supprimé la variable sodium_100g du fait de sa corrélation parfaite avec la variable salt_100g (partie 3, heatmap)
- diminué la dimension du jeu de donnée grâce à l'ACP
- créé la variable à prédire predict_lieu_vente (partie 3, heatmap, clustering)
- créé la variable à prédire predict_score (partie 5, clustering)

Idées possibles :

- On pourrait utiliser une méthode comme sklearn.feature_selection.VarianceThreshold pour supprimer les variables dont la variance est tellement faible qu'elles n'apportent pas d'information spéciale utile. Il faudrait alors comparer l'influence et l'intérêt de ces variables en fonction des modèles et de leurs métriques de comparaison de performance.
- On pourrait aussi créer une variable utilisant par exemple traces_tags et déterminant quels sont les risques allergènes d'un article. Le score nutritif n'est pas le seul critère qui peut caractériser une recette saine, sa composition en ingrédients allergènes est également importante à une époque où il y a de plus en plus d'allergies ou de contre-indications alimentaires (gluten, sel, arachide, oeufs, lait, crustacés etc.)
- De même, créer une variable basée sur les additifs qui peuvent être nocifs pour la santée et finalement, produire un score général qui prenne en compte les scores nutritif, allergène et chimique.
- Une autre possibilité serait de prendre des recettes existantes, de les décomposer en ses aliments de base (tomates, céleri, viande de boeuf etc.) et d'inclure dans la base de données les compositions des articles utilisés, et de créer une variable supplémentaire "recette" qui indiquerait si un enregistrement est un simple article ou la totalité des articles d'une recette.

Exploration des données à l'aide des variables nutrition-score-uk_100g et nutrition-score-fr_100g

Selon la Food Standards Agency, la variable nutrition-score-uk_100g est un système simple de score où des points sont alloués sur la base des contenus nutritifs de 100g de nourriture ou de boisson. Les points sont accordés pour des nutriments de classe 'A' que sont l'énergie, les graisses saturées, le sucre et le sodium, et les nutriments de classe 'C' comme les fruits, légumes, noix, fibres, protéines. Dans la majorité des cas, les points des nutriments de la classe 'C' sont soustraits à ceux de la classe 'A' (mais il y a des exceptions). Les nourritures marquant plus de 4 points, ou les boissons marquant plus de 1 point, sont classifiées comme "moins saines" (less healthy), et sont par exemple sujettes à des contrôles de l'Ofcom sur la publicité faite aux enfants sur ces nourritures. On peut donc se baser sur cette variable pour avoir un critère pour déterminer si un article est globalement sain ou non.

Dans le cadre de la Stratégie Nationale de Santé en France, il a été créé un ensemble de catégories, A, B, C, D et E basées sur le score nutritionnel français nutrition-score-fr_100g adapté du score britannique. On peut ainsi voir, comme le montrent les graphiques suivants, la répartition des articles de ce dataset vis-à-vis de ces catégories. Les articles sont également répartis en catégories d'aliment dans la variable pnns_groups_1 comme les boissons ou les produits laitiers :

In [102]:
```
count_by_category = dfOriginel.groupby('pnns_groups_1').nutrition_grade_fr.value_counts().sort_index()
count_by_category.unstack().fillna(0)
```

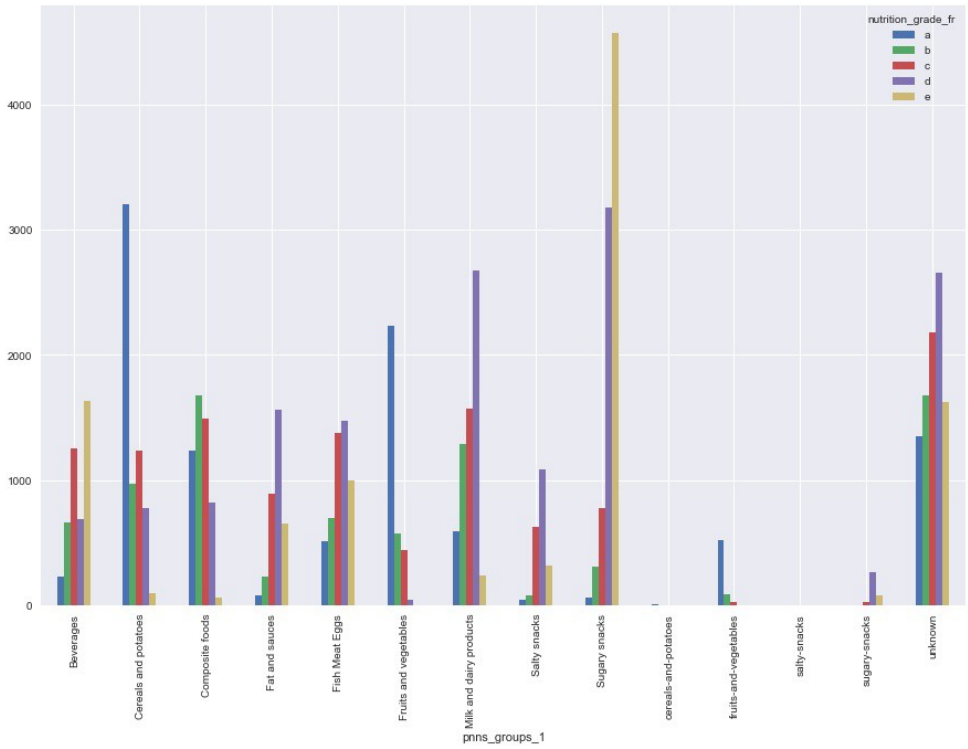
Out[102]:

nutrition_grade_fr	a	b	c	d	e
pnns_groups_1					
Beverages	233.0	666.0	1253.0	690.0	1634.0
Cereals and potatoes	3200.0	972.0	1239.0	780.0	102.0
Composite foods	1235.0	1676.0	1489.0	818.0	63.0
Fat and sauces	85.0	231.0	892.0	1559.0	653.0
Fish Meat Eggs	515.0	702.0	1378.0	1473.0	997.0
Fruits and vegetables	2236.0	578.0	440.0	44.0	3.0
Milk and dairy products	591.0	1286.0	1571.0	2675.0	238.0

nutrition_grade_fr	a	b	c	d	e
pnns_groups_1					
Salty snacks	45.0	78.0	626.0	1086.0	323.0
Sugary snacks	61.0	308.0	780.0	3174.0	4567.0
cereals-and-potatoes	11.0	1.0	0.0	0.0	0.0
fruits-and-vegetables	526.0	89.0	24.0	0.0	0.0
salty-snacks	1.0	0.0	0.0	0.0	0.0
sugary-snacks	0.0	5.0	25.0	263.0	80.0
unknown	1347.0	1676.0	2180.0	2658.0	1627.0

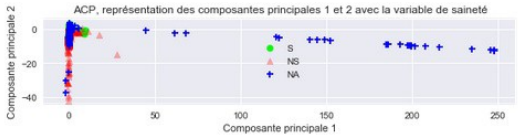
In [103]:
```
count_by_category.unstack().plot(kind='bar', figsize=(15,10))
```

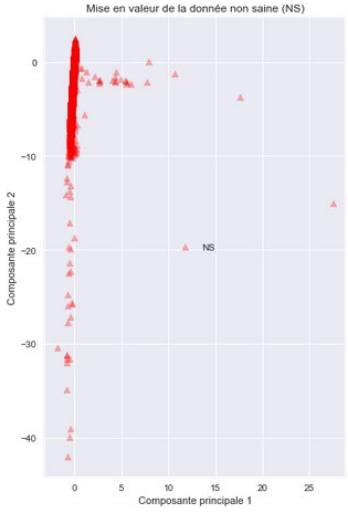
Out[103]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x45463c18>
```

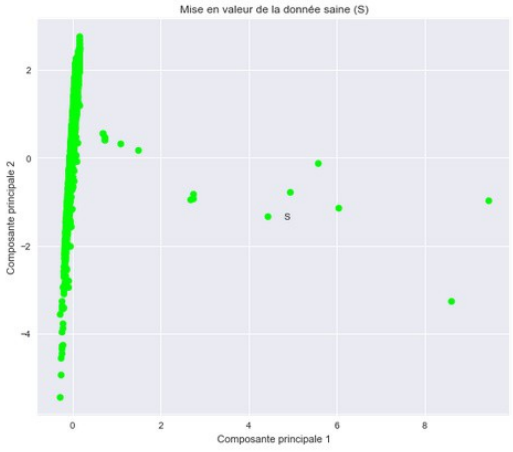


On peut ainsi voir, que pour ce jeu de données, les catégories d'articles déclarées comme étant les plus saines (catégorie A) sont les "céréales et pommes de terre" puis les "fruits et légumes", et que la pire (caégorie E) est celle des snacks sucrés, suivie des boissons.

In [104]:
```
dfOriginelS, dfContinuesSkewedSansNaNNormaliseSain = getDfContinuesSkewedSansNaNNormaliseSain(dfOriginel)
predict_score_sain = np.where(np.logical_or(np.logical_and(dfOriginelS["solide"] == 1, dfOriginelS["nutrition-score-uk_100g"]
< 4), np.logical_and(dfOriginelS["liquide"] == 1, dfOriginelS["nutrition-score-uk_100g"] < 1)), "S",
                              np.where(np.isnan(dfOriginelS["nutrition-score-uk_100g"]) == True, "NA", "NS"))
eig_pairsSain = getPropre(dfContinuesSkewedSansNaNNormaliseSain, False)
grapheACP2Composantes2(eig_pairsSain, dfContinuesSkewedSansNaNNormaliseSain, predict_score_sain, 0, 1)
```





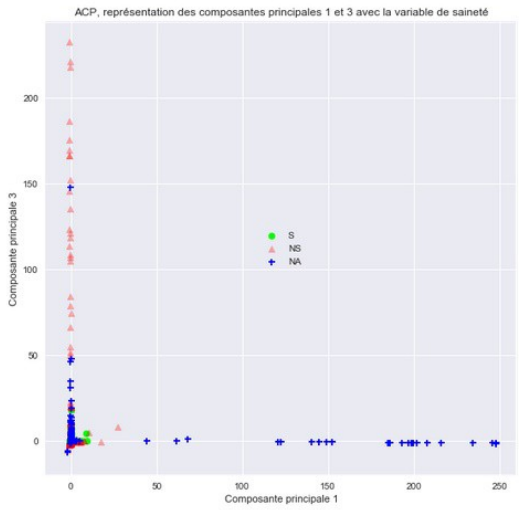


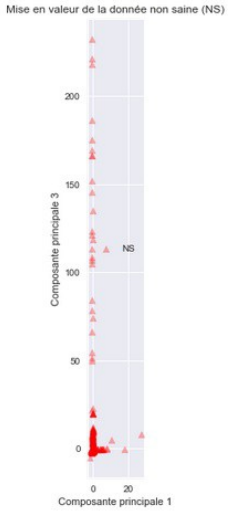
Commentaires :

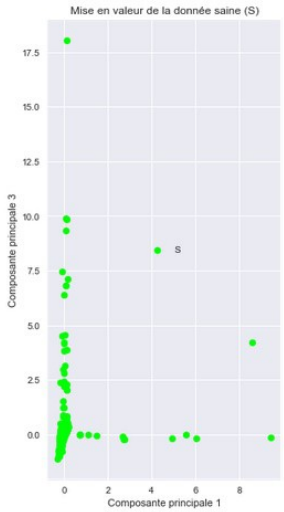
- L'axe 1 représente les acides gras (acides arachidique, béhénique, gamme-linolénique etc.)
- L'axe 2 représente les scores et certaines variables à haut potentiel "énergétique" (energy, fat, saturated-fat, sugars)
- On observe dans ces graphiques que les articles situés sur la première composante sont des articles pour lesquels il n'y a pas de score nutritionnel.
- Les articles non sains se retrouvent sur les deux composantes, mais principalement sur la composante 2, ce qui semble logique puisque la composante 2 est composée de variables dont les fortes valeurs influent très négativement sur le score nutritionnel.
- Les articles sains se retrouvent un peu sur la composante 1, mais surtout sur la composante 2 pour de faibles valeurs.
- La présence d'acides gras ne semble pas être déterminante pour dire si un article est sain ou non, cela dépend de la spécificité de chaque acide, et des conditions ("posologie").
- La composante contenant les scores, c'est donc elle qui permet de mieux discriminer des articles sains ou non.
- Toutefois, il s'agit d'observations sur la "sanité" des articles et non sur celle d'éventuelles recettes dont le score spécifique est à établir.

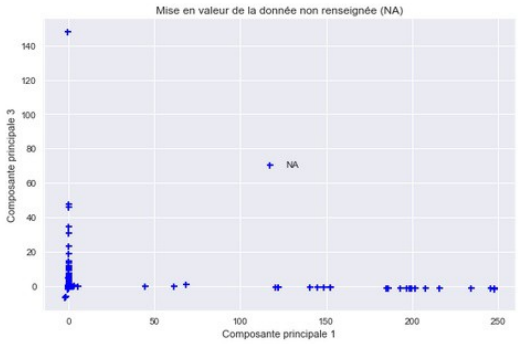
In [105]:
```
grapheACP2Composantes2(eig_pairsSain, dfContinuesSkewedSansNaNNormaliseSain, predict_score_sain, 0, 2)
```

```
<matplotlib.figure.Figure at 0x29bf66a0>
```









Commentaires :

- L'axe 1 représente les acides gras (acides arachidique, béhénique, gamme-linolénique etc.)
- L'axe 3 représente certains oligo-éléments (molybdène, chrome, bêta-carotène, cuivre etc.)
- De la même façon que pour les deux premières composantes principales, les articles sains se trouvent sur des valeurs peu élevées des composantes 1 et 3, les articles non sains possèdent des valeurs élevées sur la composante 3 de ces oligo-éléments.

  On retrouve globalement la même analyse pour les autres composantes principales.
- Les articles possédant des valeurs élevées des différents composants ne sont pas sains.
- Pour les valeurs moins élevées, certains articles sont sains, d'autres non.

L'ACP montre la représentation de la saineté des articles suivant les différentes composantes principales, mais elle ne permet pas de prédire un score de recette.

Calcul déterministe d'un score de recette basé sur le modèle du score d'un article

Un article est l'agrégat de ses constituants (ou ingrédients). Si on suppose qu'une recette est l'agrégat d'articles (le cahier des charges du projet ne précise pas ce qu'est une recette), alors moyennant certaines hypothèses, on peut construire le score d'une recette en ce basant sur le score des articles. Puisque le score d'un article est construit en fonction des valeurs de sept variables le décrivant, que ces variables sont quantitatives et correspondent à des cumuls ramenés à une masse de référence (100g de produit de l'article), on peut faire exactement de même en additionnant les valeurs des différents articles de la recette, pour ces sept variables, que l'ont ramène à des valeurs références pour 100g de recette. On peut alors de la même façon que pour un article, calculer le score pour une recette. Il faut toutefois remarquer que le calcul des scores étant différent pour les articles solides et les articles liquides, une recette mélangeant articles solides et liquides, on doit séparer les articles de la recette en deux catégories, et donc établir deux scores, un score solide et un score liquide pour la recette.

Prenons par exemple une recette factice qui contiendrait :

- un liquide (3499603010436), avec 50cl de jus d'orange de score -3
- un liquide (5410188030266), avec 200cl de soupe de score -4
- un solide (3597620005722), avec 100g de cruesli 4 noix de score 7
- un solide (3245414091221), avec 100g de colin de score 1
- un solide (3299511291268), avec 50g de lasagnes de score 3

et calculons ce score de recette :

In [106]:
```
cr = csv.reader(open("recette.csv", "rb"))
scoreSolide, scoreLiquide = getScoreRecette(dfOriginel, cr)
print("Score de la recette solide : {0}".format(scoreSolide))
print("Score de la recette liquide : {0}".format(scoreLiquide))
```

```
(139578, 161)
(72599, 161)
('liste des codes', [3499603010436L, 5410188030266L, 3597620005722L, 3245414091221L, 3299511291268L])
('liste des portions', [50, 200, 100, 100, 50])
Score de la recette solide : 0
Score de la recette liquide : -4
```

On obtient ainsi une recette donc le score liquide est de -4 et le score solide est de 0.
Que ce soit pour sa composition liquide ou solide, cette recette peut être consédérée (dans l'hypothèse que les consitions initiales sont réalistes et ont un sens) comme saine.

Il s'agit toutefois d'un score déterministe, construit suivant un certain nombre d'hypothèses qui peuvent rendre sa légitimité problématique, comme cela sera expliqué dans la synthèse.
On peut essayer de voir si, d'un point de vue statistique, sans aller jusqu'à prévoir un score de recette, on peut déjà prévoir les scores des articles.

Essai d'un modèle prédisant le score d'un article

Puisqu'un article comprend des ingrédients (variable ingredients_text), l'idée est de voir si on peut relier ces ingrédients au score nutritionnel en fonction de leur présence ou non dans les articles.

- La première étape consiste donc à faire une liste de tous les ingrédients existant dans le jeu de données.
- La deuxième crée les variables indiquant si un ingrédient est présent ou non dans les articles.
- La troisième consiste à tester deux modèles possibles et voir si cela donne des résultats.

In [111]:
```
modelesCalculScore(dfOriginel)
```

```
RandomForest r2_score : -28.9766928762
RandomForest mean_squared_error : 85.7252401449
Régression linéaire r2_score : -40.9371055897
```

Aucun des scores r2 pour ces deux modèles n'est bon.

5- Synthèse

Puisqu'il s'agit d'aider le site Lamarmite à construire un générateur de recettes saines, il faut s'entendre sur la définition de plusieurs choses :

- qu'appelle-t-on exactement une recette ?
- quel est le critère de "sanité" pour une recette ?

Le problème est qu'aucune de ces deux informations, essentielles pour construire un modèle, ne sont présentes.

Première difficulté : portions d'article
Dans une recette, on prend rarement la totalité d'un article, mais seulement une portion. Si l'on veut calculer un score pour la recette entière, on doit donc supposer que la composition des articles est homogène (1).

Deuxième difficulté : score de recette
Puisqu'il n'y a pas de définition du score d'une recette et qu'il n'y a pas de variable "score d'une recette" dans le dataset, on ne peut construire ce score qu'à partir des articles du dataset et sans aucune garantie que ce score sera un réel score de sanité puisqu'il ne pourra être comparé à aucun indicateur existant et officiel de santé pour une recette. Par exemple, il n'y a pas dans ce dataset de scores spécifiques pour les allergènes (gluten, iode, lactose, arachide etc.), et plus généralement pour l'influence des divers ingrédients chimiques sur la santé. Le score qui a été construit "à la main" utilise donc un postulat (2) qui permet de simplifier les choses mais qui le rend moins réaliste. Si une recette était considérée comme un article comme un autre, le passage d'un score d'article à un score de recette ne poserait pas de problème. Mais rien ne permet d'affirmer cette équivalence. Pour illustrer le problème, le fait qu'il existe une borne supérieure (10) aux nombres de points attribués aux variables pour un article, pose un problème quand on ajoute plusieurs articles dans une recette. Si on prend un article isolé, le fait qu'il ait par exemple une valeur de 1g ou de 10g de sel pour 100g d'article ne fait pas de différence, il lui sera ajouté 10 points A, pas un de plus. Mais par contre, si on met des articles ensemble, cet écrêtage est problématique car en fonction de la proportion de cet article très salé dans la recette, cela peut énormément jouer sur le nombre final de points pour le sodium de la recette entière.
Petit calcul pour illustrer :

4 articles sans sodium + 1 article avec 1g de sodium, l'article représentant 10% de la masse totale de la recette.
On a alors pour la recette entière, ramené à la masse de référence, 10% de 1g, soit 0.1g, c'est-à-dire un score de 1 points A.
Mais si l'article avait en réalité 10g de sodium, on aurait alors pour la recette entière, 10% de 10g, soit 1g de sodium, c'est-à-dire un score de 10 points A.
Pourtant, il s'agit du même article, qui, qu'il ait 1g ou 10g de sodium, possède toujours le même score (10 points A) quand il est isolé. Mais ce même article peut considérablement faire varier le score de la recette entière (de 1 à 10 points A) quand il est ajouté à d'autres articles.
Le passage d'un score d'article à un score de recette n'est donc pas si anodin que cela et sans définition précise du score de recette, il devient difficile de donner une réelle valeur à un score produit à partir des seules données de ce dataset.

Troisième difficulté : mixité des recettes
La sanité du score d'un article dépend de s'il est liquide ou solide, les deux scores étant indépendants l'un de l'autre. Or une recette comprend souvent des éléments solides et liquides. On ne peut donc pas mélanger les scores des parties solides et liquides de la recette. On doit donc établir un score pour chaque état de la recette.

Postulats :
(1) la composition des articles est homogène. Pour une variable donnée, un pourcentage d'article (par ex. 50%) donnera le même pourcentage de la valeur de la variable pour la portion utilisée dans la recette.
(2) le score de recette est une fonction (linéaire) des scores des articles.

Les informations contenues dans ce dataset ont permis grâce aux méthodes d'analyse multivariée de montrer certaines corrélations entre les différents composants des articles. Cela peut permettre au générateur de recette de savoir que si certains articles possèdent certains ingrédients, ils s'accompagneront également probablement de certains autres bien déterminés, ce qui peut éventuellement poser des problèmes au niveau de la santé.
Relier le dataset à des variables de critère de santé permet alors de cibler les articles considérés comme sains et d'éventuellement poser des scores sur les recettes elles-mêmes. Un utilisateur pourrait ainsi demander tel ou tel niveau d'influence sur la santé pour divers critères et récupérer des recettes pour lesquelles certains articles lui seraient proposés.
En outre, un critère important qui entre en jeu dans la possibilité de faire des recettes saines est celui de la localisation des articles. Il peut permettre, comme nous l'avons fait avec l'utilisation de la variable countries_tags, de déterminer les habitudes alimentaires des différents pays mais également de voir ce qui est disponible ou non. La recherche d'articles sains peut-être délicate quand ceux-ci ne sont produits que dans des contrées lointaines.

Outre les axes d'améliorations possibles listées ci-après, puisque ce qui est cherché est un score de recette et non plus un score d'article, il semble nécessaire que le dataset contienne comme variable un véritable score officiel de recette, la deuxième difficulté décrite ci-dessus montrant le problème de construire un score de recette à partir des scores d'articles.
Il semblerait ainsi préférable d'inclure dans la base de donnée, non plus seulement les ingrédients et les articles, mais également les recettes elles-mêmes dont on aurait un score de recette.
Cela permettrait ainsi de prédire des scores de recette pour des ingrédients donnés en décomposant les recettes existantes en ingrédients, puis en remontant vers les nouvelles recettes puisque les ingrédients (finalement les composants moléculaires, acides gras etc.) seraient en relation directe avec les scores de recette et non plus uniquement avec les scores d'article dont le passage à un score de recette semble a priori plutôt arbitraire.

6- Améliorations possibles

(Des propositions de traitement potentiel des données qui peuvent aider l'entreprise à développer son générateur de recettes.)

- La variable catégorie est intéressante, mais elle n'est pas canalisée, il pourrait être utile sur le site de renseignement des informations d'intégrer des listes déroulantes afin de pouvoir choisir des catégories déjà existantes (quitte à pouvoir en ajouter quand celles-ci n'existent pas encore). Cela éviterait d'avoir des catégories différentes qui sont en réalité la même. Faire des statistiques sur les catégories plutôt que sur la totalité des articles pourrait être un bénéfice important quand on veut faire une recette où on sait par exemple qu'on souhaite y intégrer une catégorie particulière.
- La variable created_datetime pourrait être nettement améliorée si au lieu de renseigner la date de création de l'enregistrement, elle indiquait la date de création de l'article lui-même. Elle permettrait une meilleure précision si on voulait par exemple étudier l'évolution dans le temps (voire dans les différents pays) des compositions chimiques d'une catégorie d'article (par exemple vis-à-vis d'un critère de bonne santé comme le score). Même si cette information n'est pas forcément facile à trouver, elle donnerait une meilleure indication que celle du renseignement d'un article qui peut être en vente depuis par exemple 15 ans. En fait, il faudrait la date de création de l'article, et une mise à jour de sa date de diffusion (comme la variable last-modified) mais ne correspondant pas à la modification d'un enregistrement mais de la réelle présence à un instant donné de l'article à la vente.
- Les variables dont le nom se termine par "tags" sont intéressantes mais devaient également être canalisées par des listes de propositions afin d'éviter qu'elles ne servent à rien du fait d'une trop grande dispersion des termes (langues, fautes de frappe etc.)
- Cela peut sembler extrême mais certaines informations devraient être rendues obligatoires afin de pouvoir enregistrer un article, comme par exemple la marque, la catégorie, l'origine, du moins toutes les variables que l'ont peut trouver dans tous les articles. Cela permettrait d'être sûr qu'un enregistrement ne sera pas rempli n'importe comment et cela donnerait une meilleure cohérence au jeu de données.
- Il pourrait être intéressant d'indiquer le prix des articles, car il peut parfois y avoir une corrélation forte entre le prix et le caractère sain ou non d'une recette (par exemple le bio est considéré comme plus sain et tend à être en moyenne plus cher que le non bio).
- Enfin, même si la confiance n'exclut pas le contrôle, puisqu'il y a une variable où on peut trouver une photopgrahie du produit, il pourrait de la même façon y avoir une variable où il y aurait une photographie des informations nutritionnelles de l'article, afin de vérifier les informations quand on tombe sur des données aberrantes ou des valeurs extrêmes.

Code:

In [1]:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.decomposition import PCA
import sklearn.preprocessing as prep
from sklearn.cluster import KMeans
from sklearn.cluster import MiniBatchKMeans
from matplotlib.gridspec import GridSpec
import time
%matplotlib inline
import warnings
warnings.filterwarnings('ignore')
import csv
import re

from subprocess import check_output
from nltk.corpus import stopwords
from nltk.tokenize import wordpunct_tokenize, RegexpTokenizer
from nltk.stem.wordnet import WordNetLemmatizer
from nltk.stem import LancasterStemmer
from gensim import models
import nltk
nltk.download('stopwords')
nltk.download('wordnet')
from string import punctuation
from nltk.corpus import stopwords
from sklearn.model_selection import cross_val_predict
from sklearn import linear_model
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import r2_score, mean_squared_error

# fichier des données, adresse à adapter
dfOriginel = pd.read_csv('fr.openfoodfacts.org.productsCopie.csv', sep='\t')

def getDfContinuesSkewed(_df):
    """Sépare les variables continues et non continues puis crée un dataframe des variables continues ordonnées suivant leur
skew croissant
    Arguments:
    _df -- dataframe contenant les données initiales
    Retour:
    dfContinuesSkewed -- dataframe contenant les variables continues ordonnées suivant leur skew croissant
    dfContinues -- dataframe contenant les variables continues
    dfNonContinues -- dataframe contenant les variables non continues
    """

    # séparation variables continues / non continues
    dfContinues = pd.DataFrame()
    dfNonContinues = pd.DataFrame()
    dfSans100NaN = pd.DataFrame()
    co = _df.count()
    i = 0
    while i < _df.shape[1]:
        if co[i] > 0:
            dfSans100NaN.insert(dfSans100NaN.shape[1], _df.columns[i], _df[_df.columns[i]])
        i = i + 1

    i = 0
    while i < dfSans100NaN.shape[1]:
        if dfSans100NaN.dtypes[i] == np.float64 or dfSans100NaN.dtypes[i] == np.int32:
            dfContinues.insert(dfContinues.shape[1], dfSans100NaN.columns[i], dfSans100NaN[dfSans100NaN.columns[i]])
        else:
            dfNonContinues.insert(dfNonContinues.shape[1], dfSans100NaN.columns[i], dfSans100NaN[dfSans100NaN.columns[i]])
        i = i + 1

    dfNanContinues = 100*(1-dfContinues.count()/dfContinues.shape[0])

    # normalisation
    dfContinues = (dfContinues-dfContinues.mean()) / dfContinues.std()


    # tableau des valeurs d'asymétrie afin de ne garder que les variables avec un skew (élimine les variables vides ou quasim
ent vides
    tabSkewness = []
    i = 0
    while i < dfContinues.shape[1]:
        if ~np.isnan(dfContinues[dfContinues.columns[i]].skew()):
            tabSkewness.append((dfContinues.columns[i], dfContinues[dfContinues.columns[i]].skew(), dfNanContinues[dfContinue
s.columns[i]], co[i]))
        i = i + 1

    # tableau ordonné des valeurs d'asymétrie
    tabSkewnessSorted = sorted(tabSkewness, key=lambda x:x[1])

    # dataframe des variables continues ayant une valeur de skewness
    dfContinuesSkewed = pd.DataFrame()
    i = 0
    while i < len(tabSkewnessSorted):
        dfContinuesSkewed.insert(dfContinuesSkewed.shape[1], tabSkewnessSorted[i][0], _df[tabSkewnessSorted[i][0]])
        i = i + 1

    return dfContinuesSkewed, dfContinues, dfNonContinues
```

```
def grapheRemplissageVariables(_df):
    """Dessine le graphe du remplissage des variables du dataset
    Arguments:
    _df -- dataframe contenant les données initiales
    """
    dfNanOriginel = 100*(1-_df.count()/_df.shape[0])
    X = (100-dfNanOriginel).sort_values()
    plt.clf()
    plt.bar(range(len(X)), X)
    plt.xlabel("indice des variables")
    plt.ylabel("%")
    plt.title(u'% ordonné de remplissage des variables')
    plt.show()

def getDfNanOriginel(_df):
    dfNanOriginel = 100*(1-_df.count()/_df.shape[0])
    return dfNanOriginel

def getDfContinues(_df):
    dfContinues = pd.DataFrame()
    dfOriginelSans100NaN = pd.DataFrame()
    co = _df.count()
    i = 0
    while i < _df.shape[1]:
        if co[i] > 0:
            dfOriginelSans100NaN.insert(dfOriginelSans100NaN.shape[1], _df.columns[i], _df[_df.columns[i]])
        i = i + 1
    i = 0
    while i < dfOriginelSans100NaN.shape[1]:
        if dfOriginelSans100NaN.dtypes[i] == np.float64:
            dfContinues.insert(dfContinues.shape[1], dfOriginelSans100NaN.columns[i], dfOriginelSans100NaN[dfOriginelSans100N
aN.columns[i]])
        i = i + 1

    return dfContinues

def getDfNanContinues(_df):
    dfContinues = getDfContinues(_df)
    dfNanContinues = 100*(1-dfContinues.count()/dfContinues.shape[0])
    return dfNanContinues

def make_autopct(values):
    def my_autopct(pct):
        """Customisation de l'affichage des valeurs {p:.2f}% ({v:d})
        Arguments:
        values -- nombres de variables pour chaque section
        pct -- pourcentages de variables pour chaque section
        """
        total = sum(values)
        val = int(round(pct*total/100.0))
        if val == 0:
            return ''
        else:
            return '{p:.2f}% ({v:d})'.format(p=pct, v=val)
    return my_autopct

def make_autopct2(values):
    def my_autopct2(pct):
        """Customisation de l'affichage des valeurs {v:d}
        Arguments:
        values -- nombres de variables pour chaque section
        pct -- pourcentages de variables pour chaque section
        """
        total = sum(values)
        val = int(round(pct*total/100.0))
        return '{v:d}'.format(v=val)
    return my_autopct2


def getComposants(_df):
    # dataframe des variables décrivant des composants (se terminent par _100g)
    dfComposant = pd.DataFrame()
    i = 0
    while i < _df.shape[1]:
        if (_df.dtypes[i] == np.float64) & ("_100g" in _df.columns[i]):
            dfComposant.insert(dfComposant.shape[1], _df.columns[i], _df[_df.columns[i]])
        i = i + 1
    return dfComposant

def grapheRemplissageVariablesNumeriques(_df):
    nb_enreg = _df.shape[0]
    classement=[]
    for i in range(10):
        classement.append(0)
    dfComposant = getComposants(_df)
    for i in range (dfComposant.count().shape[0]):
        classement[dfComposant.count()[i]/(nb_enreg/10)] += 1

    explode = (0.10, 0, 0, 0.1, 0.25, 0.1, 0, 0, 0, 0)

    the_grid = GridSpec(2,2)
    plt.clf()
    plt.figure(figsize=(17,10))
    plt.subplot(the_grid[:,0], aspect=1)
```

```
    _, _, autotexts = plt.pie(classement, colors = ('gold', 'c', 'r', 'yellowgreen', 'lightskyblue', 'lightcoral', 'b', 'm',
'k', 'g'), explode=explode, autopct=make_autopct(classement), startangle=90, shadow=True, pctdistance=0.5) #'%1.1f%%'
    plt.title(u"% de variables numériques (nombre de variables) ayant\n entre . et . % de données renseignées (couleurs)", bb
ox={'facecolor':'0.8', 'pad':5})
    autotexts[0].set_fontsize(16)
    autotexts[3].set_fontsize(16)
    autotexts[3].set_position((0.60, 0.10))
    autotexts[4].set_fontsize(16)
    autotexts[4].set_position((0.70, 0.35))
    autotexts[5].set_fontsize(16)
    autotexts[5].set_position((0.30, 0.90))
    plt.axis('equal')
    labels = ['0-10%', '10-20%', '20-30%', '30-40%', '40-50%', '50-60%', '60-70%', '70-80%', '80-90%', '90-100%']
    plt.legend(labels, loc='lower left')
    plt.annotate('', xy=(0.27, 0.34), xytext=(0.40, 0.15), arrowprops={'facecolor':'black', 'shrink':0.05})
    plt.annotate('', xy=(0.40, 0.62), xytext=(0.62, 0.43), arrowprops={'facecolor':'black', 'shrink':0.05})

    values = [65, 33]
    plt.subplot(the_grid[0,1], aspect=1)
    colors = ['blue', 'orange']
    _, _, autotexts = plt.pie(values, autopct=make_autopct2(values), colors=colors)
    autotexts[0].set_color('white')
    autotexts[1].set_color('black')
    for i in range(2):
        autotexts[i].set_fontsize(16)
    plt.title(u"Nombre de variables numériques ayant\n moins de 1 % de données renseignées", bbox={'facecolor':'0.8', 'pad':
5})
    plt.legend(['<1% rempli', '>1% rempli'], loc='lower right')
    plt.subplot(the_grid[1,1], aspect=1)
    values = [87, 11]
    _, _, autotexts = plt.pie(values, autopct=make_autopct2(values), colors=colors)
    autotexts[0].set_color('white')
    autotexts[1].set_color('black')
    for i in range(2):
        autotexts[i].set_fontsize(16)
    plt.title(u"Nombre de variables numériques ayant\n moins de 4 % de données renseignées", bbox={'facecolor':'0.8', 'pad':
5})
    plt.legend(['<4% rempli', '>4% rempli'], loc='lower right')
    plt.show()

def grapheComparaisonNanVariables(_df):
    plt.clf()
    plt.figure(figsize=(12,4))

    plt.subplot(121)

    X1 = (100-getDfNanOriginel(_df)).sort_values()
    plt.bar(range(len(X1)), X1)
    plt.xlabel("indice des variables")
    plt.ylabel("%")
    plt.title(u'% ordonné de remplissage des variables')

    plt.subplot(122)

    X2 = (100-getDfNanContinues(_df)).sort_values()
    plt.bar(range(len(X2)), X2)
    plt.xlabel("indice des variables")
    plt.ylabel("%")
    plt.title(u'% ordonné de remplissage des variables numériques')
    plt.show()

def dfBoxPlot(_df, _title):
    """Dessine le boxplot
    Arguments:
    _df -- dataframe contenant les données initiales
    _title -- titre du boxplot
    """
    plt.clf()
    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)
    plt.show()

def showChangeMeanProteins(_df):

    dfContinues = getDfContinues(_df)
    # ajout de la variable code comme index afin de pouvoir jouer sur les code et non sur les index initiaux (n° de ligne)
    dfBoxPlot(dfContinues[["proteins_100g", "fat_100g", "carbohydrates_100g"]], "proteins_100g, fat_100g et carbohydrates_100
g")
    print("Moyenne d'avant annulation de la valeur aberrante : {0}".format(dfContinues[["proteins_100g", "fat_100g", "carbohy
drates_100g"]].mean()))
    dfContinues = pd.concat([dfContinues, dfOriginel[['code']]], axis = 1)
    dfContinues = dfContinues.set_index('code')
    dfContinues['proteins_100g'][15666666666] = 0
    dfContinues['fat_100g'][15666666666] = 0
    dfContinues['carbohydrates_100g'][15666666666] = 0
    dfBoxPlot(dfContinues[["proteins_100g", "fat_100g", "carbohydrates_100g"]], "proteins_100g, fat_100g et carbohydrates_100
g")
    print("Moyenne d'apres annulation de la valeur aberrante : {0}".format(dfContinues[["proteins_100g", "fat_100g", "carbohy
drates_100g"]].mean()))
```

```
def graphesDiminutionAsymetrie(_df):
    dfOriginelSans100NaN = pd.DataFrame()
    co = _df.count()
    i = 0
    while i < _df.shape[1]:
        if co[i] > 0:
            dfOriginelSans100NaN.insert(dfOriginelSans100NaN.shape[1], _df.columns[i], _df[_df.columns[i]])
        i = i + 1
    dfContinues = pd.DataFrame()
    dfNonContinues = pd.DataFrame()

    i = 0
    while i < dfOriginelSans100NaN.shape[1]:
        if dfOriginelSans100NaN.dtypes[i] == np.float64:
            dfContinues.insert(dfContinues.shape[1], dfOriginelSans100NaN.columns[i], dfOriginelSans100NaN[dfOriginelSans100N
aN.columns[i]])
        else:
            dfNonContinues.insert(dfNonContinues.shape[1], dfOriginelSans100NaN.columns[i], dfOriginelSans100NaN[dfOriginelSa
ns100NaN.columns[i]])
        i = i + 1
    dfNanContinues = 100*(1-dfContinues.count()/dfContinues.shape[0])

    tabSkewness = []
    co = dfContinues.count()
    X = []
    Y = []
    i = 0
    while i < dfContinues.shape[1]:
        if ~np.isnan(dfContinues[dfContinues.columns[i]].skew()):
            tabSkewness.append((dfContinues.columns[i], dfContinues[dfContinues.columns[i]].skew(), dfNanContinues[dfContinue
s.columns[i]], co[i]))
            Y.append(dfContinues[dfContinues.columns[i]].skew())
            X.append(co[i])
        i = i + 1
    tabSkewnessSorted = sorted(tabSkewness, key=lambda x:x[1])
    dfContinuesSkewed = pd.DataFrame()
    i = 0
    while i < len(tabSkewnessSorted):
        dfContinuesSkewed.insert(dfContinuesSkewed.shape[1], tabSkewnessSorted[i][0], _df[tabSkewnessSorted[i][0]])
        i = i + 1

    dfContinuesSkewed['proteins_100g'][15666666666] = 0
    dfContinuesSkewed['fat_100g'][15666666666] = 0
    dfContinuesSkewed['carbohydrates_100g'][15666666666] = 0

    tabRatLog = []
    tabRatRac = []

    for i in range(dfContinuesSkewed.shape[1]):
        skewNorm = dfContinuesSkewed[dfContinuesSkewed.columns[i]].skew()   # skew variable originelle X
        logcol = np.log(1+dfContinuesSkewed[dfContinuesSkewed.columns[i]])  # variable log(1 + X)
        skewLog = logcol.skew()                                             # skew variable log(1 + X)
        sqrtcol = np.sqrt(dfContinuesSkewed[dfContinuesSkewed.columns[i]])  # variable sqrt(X)
        skewRac = sqrtcol.skew()                                            # skew variable sqrt(X)
        ratLog = np.abs(skewLog/skewNorm)                                   # ratio skewLog/skewNorm
        tabRatLog.append(np.log(ratLog))
        ratRac = np.abs(skewRac/skewNorm)                                   # ratio skewRac/skewNorm
        tabRatRac.append(np.log(ratRac))
    plt.clf()
    plt.figure(1)
    x = np.arange(dfContinuesSkewed.shape[1])
    y = tabRatLog
    plt.title("Transformation de l'asymetrie avec la fonction logarithmique")
    plt.xlabel('variables avec skew')
    plt.ylabel('log du ratio pour fct log')
    plt.plot(x,y,'g')
    plt.plot([0, dfContinuesSkewed.shape[1]-1], [0,0], 'r-', lw=2)
    plt.show()
    plt.figure(2)
    x = np.arange(dfContinuesSkewed.shape[1])
    y = tabRatRac
    plt.xlabel('variables avec skew')
    plt.ylabel('log du ratio pour fct sqrt')
    plt.plot(x,y,'b')
    plt.plot([0, dfContinuesSkewed.shape[1]-1], [0,0], 'r-', lw=2)
    plt.title("Transformation de l'asymetrie avec la fonction racine carree")
    plt.show()

    return dfContinuesSkewed

def grapheExtremes(_coeff, _dfComposant):
    """Graphes des % de valeurs extrêmes
    Arguments:
    _coeff -- coefficient multiplicateur pour déterminer le seuil (moyenne + coeff * écart-type)
    _dfComposant -- DataFrame des variables composant
    """
    valExtr = _dfComposant[_dfComposant>_dfComposant.mean()+_coeff*_dfComposant.std()].count()
    liste_nom_val_extreme = []
    valCo = _dfComposant.count()
    for nom in _dfComposant.columns:
        if valExtr[nom] > 0:
            liste_nom_val_extreme.append((nom, 100.0/valCo[nom]*valExtr[nom]))
    liste_nom_val_extreme_sorted = sorted(liste_nom_val_extreme, key=lambda x:x[1])
    liste_val_extreme_sorted = []
    for i in range(len(liste_nom_val_extreme_sorted)):
        liste_val_extreme_sorted.append(liste_nom_val_extreme_sorted[i][1])

    return liste_val_extreme_sorted
```

```
def comparaisonGraphesExtremes(_df):

    dfComposant = getComposants(_df)
    liste_val_extreme_sorted15 = grapheExtremes(1.5, dfComposant)
    liste_val_extreme_sorted3 = grapheExtremes(3, dfComposant)

    plt.clf()
    plt.figure(figsize=(15,4))
    plt.subplot(121)
    plt.bar(range(len(liste_val_extreme_sorted15)), liste_val_extreme_sorted15)
    plt.xlabel("indice des variables")
    plt.ylabel("%")
    plt.title(u'% ordonné du nombre de valeurs extrêmes (> mean + 1.5std)')
    plt.subplot(122)
    plt.xlabel("indice des variables")
    plt.ylabel("%")
    plt.bar(range(len(liste_val_extreme_sorted3)), liste_val_extreme_sorted3)
    plt.title(u'% ordonné du nombre de valeurs extrêmes (> mean + 3std)')
    plt.show()

def graphesComparatifCorrelations(_dfContinuesSkewedSansNaN, _dfOriginel):
    print(_dfContinuesSkewedSansNaN.corr().iloc[4:5,6:7])
    print(_dfContinuesSkewedSansNaN.corr().iloc[78:79,78:79])

    plt.clf()
    plt.figure(figsize=(14,6))
    plt.subplot(121)
    ax = sns.regplot(x="behenic-acid_100g", y="arachidic-acid_100g", data=dfOriginel)
    plt.title(u'Relation entre les variables behenic-acid_100g et arachidic-acid_100g')
    plt.subplot(122)
    ax = sns.regplot(x="sodium_100g", y="salt_100g", data=dfOriginel)
    plt.title(u'Relation entre les variables sodium_100g et salt_100g')
    plt.show()
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=(12,12))
    cmap = sns.diverging_palette(220, 10, as_cmap=True)

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

def composantesPrincipale(dfNormalise, dfNonNormalise, nombre_composantes, numero_composante):
    """Affichage des dix premières variables importantes d'une composante principale de l'ACP
    Arguments:
    dfNormalise -- dataframe contenant les variables normalisées pour l'ACP
    dfNonNormalise -- dataframe contenant les variables non normalisées
    composante -- numéro de la composante principale à afficher
    """
    pca = PCA(n_components=nombre_composantes).fit(prep.scale(dfNormalise))

    pca_df = pd.DataFrame(pca.components_[numero_composante])
    pca df.columns = ['coeff']
    feat_pca = pd.DataFrame(dfNonNormalise.columns.values)
    feat_pca.columns = ['features']

    pca_feat_importances = pd.merge(feat_pca,pca_df,left_index=True,right_index=True)
    pca_feat_importances['importances']= pca_feat_importances['coeff']/pca_feat_importances['coeff'].sum()*100
    pca_feat_importances['importances'] = pca_feat_importances['importances'].apply(lambda x:abs(int(x)))
    print(pca_feat_importances.sort(['importances'], ascending=[0]).head(10))

def ACP(dfContinuesSkewedSansNaN):
    dfContinuesSkewedSansNaNNormalise = prep.StandardScaler().fit_transform(dfContinuesSkewedSansNaN)
    print("Variables de la première composante principale de l'ACP :")
    composantesPrincipale(dfContinuesSkewedSansNaNNormalise, dfContinuesSkewedSansNaN, 3, 0)
    print("Variables de la deuxième composante principale de l'ACP :")
    composantesPrincipale(dfContinuesSkewedSansNaNNormalise, dfContinuesSkewedSansNaN, 3, 1)
    print("Variables de la troisième composante principale de l'ACP :")
    composantesPrincipale(dfContinuesSkewedSansNaNNormalise, dfContinuesSkewedSansNaN, 3, 2)
    return dfContinuesSkewedSansNaNNormalise
```

```
def getPropre(df, graphe):
    """Méthode permettant d'obtenir les valeurs propres et les vecteurs propres de la matrice de corrélation, affiche égaleme
nt le graphe de la participation de chaque variable à l'explication de la variance
    Arguments:
    df -- dataframe contenant les variables
    graphe -- booléen indiquant si on veut que le graphe soit affiché ou non
    Retour: la liste des tuples (valeurs propres, vecteurs propres)
    """
    cor_mat1 = np.corrcoef(df.T)
    eig_vals, eig_vecs = np.linalg.eig(cor_mat1)
    tot = sum(eig_vals)
    var_exp = [(i / tot)*100 for i in sorted(eig_vals, reverse=True)]
    cum_var_exp = np.cumsum(var_exp)
    if graphe:
        with plt.style.context('seaborn-whitegrid'):
            plt.clf()
            plt.figure(figsize=(10, 6))
            plt.bar(range(df.shape[1]), var_exp, alpha=0.5, align='center',
                    label='Variance expliquee individuellement')
            plt.step(range(df.shape[1]), cum_var_exp, where='mid',
                     label='Variance expliquee par cumul')
            plt.ylabel('Pourcentage de variance expliquee')
            plt.xlabel('Composantes principales')
            plt.legend(loc='best')
            plt.tight_layout()
            plt.show()

    # Make a list of (eigenvalue, eigenvector) tuples
    eig_pairs = [(np.abs(eig_vals[i]), eig_vecs[:,i]) for i in range(len(eig_vals))]

    return eig_pairs

def grapheACP2Composantes(_val_vect_propres, _df, _var_prediction):
    """Représentation graphique des deux composantes principales de l'ACP
    Arguments:
    _val_vect_propres -- tuples des valeurs propres et des vecteurs propres de la matrice de corrélation
    _df -- dataframe des variables continues dont les valeurs manquantes ont été mises à 0
    _var_prediction -- variable à prédire
    """

    # tri des tuples (eigenvalue, eigenvector) par ordre décroissant
    _val_vect_propres.sort(key=lambda x: x[0], reverse=True)

    matrix_w = np.hstack((_val_vect_propres[0][1].reshape(_df.shape[1],1), _val_vect_propres[1][1].reshape(_df.shape[1],1)))

    Y = _df.dot(matrix_w)

    plt.clf()
    plt.figure(figsize=(10, 6))
    for lab, col, m in zip(('FR', 'PASFR'), ('lime', (1,0,0,0.3)), ('o', '^')):
        plt.scatter(Y[_var_prediction==lab, 0], Y[_var_prediction==lab, 1], label=lab, c=col, marker=m)
    plt.xlabel('Composante principale 1')
    plt.ylabel('Composante principale 2')
    plt.title(u'ACP, représentation des deux premières composantes principales')
    plt.legend(loc='lower right')
    plt.show()

def grapheACP(_dfContinuesSkewedSansNaNNormalise, _df):
    eig_pairs = getPropre(_dfContinuesSkewedSansNaNNormalise, False)
    predict_lieu_vente = np.where(_df["countries_tags"].str.contains("en:france|en:french|en:new-caledonia|en:guadeloupe|en:m
artinique|en:reunion|en:saint-pierre-and-miquelon")==True, "FR", "PASFR")
    grapheACP2Composantes(eig_pairs, _dfContinuesSkewedSansNaNNormalise, predict_lieu_vente)

def grapheComparatifHeatmap(_df):
    df = _df.copy()
    plt.clf()
    plt.figure(figsize=(14,6))
    plt.subplot(121)

    dfContinuesSkewed, dfContinues, dfNonContinues = getDfContinuesSkewed(df)
    dfContinuesSkewedSansNaN = dfContinuesSkewed.fillna(0)

    corr = dfContinuesSkewedSansNaN.corr(method="pearson")
    # 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=(12,12))
    cmap = sns.diverging_palette(220, 10, as_cmap=True)
    # traçage de la heatmap
    xticks = False
    ax = sns.heatmap(corr, cmap=cmap, cbar_kws={"shrink": .5}, mask=mask)
    for label in ax.get_yticklabels():
        if label.get_text() != "chromium_100g" and label.get_text() != "beta-carotene_100g" and label.get_text() != "copper_1
00g" and label.get_text() != "molybdenum_100g":
            label.set_size(0.1)
    for label in ax.get_xticklabels():
        if label.get_text() != "chromium_100g" and label.get_text() != "beta-carotene_100g" and label.get_text() != "copper_1
00g" and label.get_text() != "molybdenum_100g":
            label.set_size(0.1)
    plt.title(u"Heatmap avec code 3401528535864")


    plt.subplot(122)
    -                -
```

```
    dfOriginelSans3401528535864 = df[df.code <> 3401528535864]
    dfContinuesSkewedSans3401528535864, dfContinuesSans3401528535864, dfNonContinuesSans3401528535864 = getDfContinuesSkewed
(dfOriginelSans3401528535864)
    dfContinuesSkewedSansNaNSans3401528535864 = dfContinuesSkewedSans3401528535864.fillna(0)

    corr = dfContinuesSkewedSansNaNSans3401528535864.corr(method="pearson")
    # 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=(12,12))
    cmap = sns.diverging_palette(220, 10, as_cmap=True)
    # traçage de la heatmap
    sns.heatmap(corr, cmap=cmap, cbar_kws={"shrink": .5}, mask=mask, xticklabels=False, yticklabels=False)
    plt.title(u"Heatmap sans code 3401528535864")

    plt.show()

def graphe_KMeans(_df, _predict, _nb_clusters): #dfContinuesSkewedSansNaN

    deb = time.clock()

    reduced_data = PCA(n_components=2).fit_transform(_df)
    kmeans = KMeans(init='k-means++', n_clusters=_nb_clusters, n_init=10)
    kmeans.fit(reduced_data)
    fin1 = time.clock()
    h = 0.1     # point in the mesh [x_min, x_max]x[y_min, y_max].

    x_min, x_max = reduced_data[:, 0].min() - 1, reduced_data[:, 0].max() + 1
    y_min, y_max = reduced_data[:, 1].min() - 1, reduced_data[:, 1].max() + 1

    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
    Z = kmeans.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    plt.figure(1)
    plt.clf()
    plt.imshow(Z, interpolation='nearest',
           extent=(xx.min(), xx.max(), yy.min(), yy.max()),
           cmap=plt.cm.Paired,
           aspect='auto', origin='lower')
    plt.plot(reduced_data[:, 0], reduced_data[:, 1], 'k.', markersize=2)
    centroids = kmeans.cluster_centers_
    print("centroids = ",centroids)
    plt.scatter(centroids[:, 0], centroids[:, 1],
            marker='x', s=169, linewidths=3,
            color='w', zorder=10)
    plt.title('K-means clustering on Marmite dataset (PCA-reduced data)\n'
              'Centroids are marked with white cross')
    plt.xlim(x_min, x_max)
    plt.ylim(y_min, y_max)
    plt.xticks(())
    plt.yticks(())
    plt.show()
    fin2 = time.clock()
    print("Temps d'exécution du KMeans : {0}".format(fin1-deb))
    print("Temps d'exécution graphe : {0}".format(fin2-fin1))
    print("Inertie du K-means : {0}".format(kmeans.inertia_))

def getDfContinuesSkewedSansNaNSansAberrNormalise(_df):
    dfSansAberr = _df.copy()
    dfSansAberr = dfSansAberr.set_index('code', drop=False)
    dfSansAberr['proteins_100g'][15666666666] = 0
    dfSansAberr['fat_100g'][15666666666] = 0
    dfSansAberr['carbohydrates_100g'][15666666666] = 0
    dfContinuesSkewedSansAberr, dfContinuesSansAberr, dfNonContinuesSansAberr = getDfContinuesSkewed(dfSansAberr)
    dfContinuesSkewedSansNaNSansAberr = dfContinuesSkewedSansAberr.fillna(0)
    dfContinuesSkewedSansNaNSansAberrNormalise = prep.StandardScaler().fit_transform(dfContinuesSkewedSansNaNSansAberr)
    return dfContinuesSkewedSansNaNSansAberrNormalise

def graphe_MiniBatchKMeans(_df, _nb_clusters): #dfContinuesSkewedSansNaN
    """Représentation graphique des clusters d'après la méthode des MiniBatchKMeans
    Arguments:
    _df -- dataframe des variables continues
    _nb_clusters -- nombre de clusters
    """

    deb = time.clock()
    reduced_data = PCA(n_components=2).fit_transform(_df)
    miniBatchKmeans = MiniBatchKMeans(n_clusters=_nb_clusters, n_init=10)
    miniBatchKmeans.fit(reduced_data)
    fin1 = time.clock()
    h = 0.1     # point in the mesh [x_min, x_max]x[y_min, y_max].

    x_min, x_max = reduced_data[:, 0].min() - 1, reduced_data[:, 0].max() + 1
    y_min, y_max = reduced_data[:, 1].min() - 1, reduced_data[:, 1].max() + 1

    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
    Z = miniBatchKmeans.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    plt.figure(1)
    plt.clf()
    plt.imshow(Z, interpolation='nearest',
           extent=(xx.min(), xx.max(), yy.min(), yy.max()),
           cmap=plt.cm.Paired,
           aspect='auto', origin='lower')
    plt.plot(reduced_data[:, 0], reduced_data[:, 1], 'k.', markersize=2)
    centroids = miniBatchKmeans.cluster_centers_
```

```
    print("centroids = ",centroids)
    plt.scatter(centroids[:, 0], centroids[:, 1],
            marker='x', s=169, linewidths=3,
            color='w', zorder=10)
    plt.title('MiniBatchKMeans clustering on Marmite dataset (PCA-reduced data)\n'
              'Centroids are marked with white cross')
    plt.xlim(x_min, x_max)
    plt.ylim(y_min, y_max)
    plt.xticks(())
    plt.yticks(())
    plt.show()
    fin2 = time.clock()
    print("Temps d'exécution du MiniBatchKMeans : {0}".format(fin1-deb))
    print("Temps d'exécution graphe : {0}".format(fin2-fin1))
    print("Inertie du miniBatchKmeans : {0}".format(miniBatchKmeans.inertia_))

def grapheACP2Composantes2(_val_vect_propres, _df, _var_prediction, _num_comp_inf, _num_comp_sup):
    """Représentation graphique des deux composantes principales de l'ACP pour la variable à prédire sain (S) / pas sain (NS)
    Arguments:
    _val_vect_propres -- tuples des valeurs propres et des vecteurs propres de la matrice de corrélation
    _df -- dataframe des variables continues dont les valeurs manquantes ont été mises à 0
    _var_prediction -- variable à prédire
    _num_comp_inf -- numéro d'une des deux composantes principales à afficher
    _num_comp_inf -- numéro de l'autre des composantes principales à afficher
    """

    # tri des tuples (eigenvalue, eigenvector) par ordre décroissant
    _val_vect_propres.sort(key=lambda x: x[0], reverse=True)
    matrix_w = np.hstack((_val_vect_propres[_num_comp_inf][1].reshape(_df.shape[1],1), _val_vect_propres[_num_comp_sup][1].re
shape(_df.shape[1],1)))

    Y = _df.dot(matrix_w)

    the_grid = GridSpec(2,2)
    plt.clf()
    plt.figure(figsize=(20,20))

    plt.subplot(the_grid[0,0], aspect=1)

    for lab, col, m in zip(('S', 'NS', 'NA'), ('lime', (1,0,0,0.3), 'blue'), ('o', '^', '+')):
        plt.scatter(Y[_var_prediction==lab, 0], Y[_var_prediction==lab, 1], label=lab, c=col, marker=m)
    plt.xlabel('Composante principale '+str(1+_num_comp_inf))
    plt.ylabel('Composante principale '+str(1+_num_comp_sup))
    plt.title(u"ACP, représentation des composantes principales "+str(1+_num_comp_inf)+" et "+str(1+_num_comp_sup)+u" avec la
variable de saineté")
    plt.legend(loc='center')

    plt.subplot(the_grid[0,1], aspect=1)

    for lab, col, m in zip(('', 'NS'), ('lime', (1,0,0,0.3)), ('', '^')):
        plt.scatter(Y[_var_prediction==lab, 0], Y[_var_prediction==lab, 1], label=lab, c=col, marker=m)
    plt.xlabel('Composante principale '+str(1+_num_comp_inf))
    plt.ylabel('Composante principale '+str(1+_num_comp_sup))
    plt.title(u'Mise en valeur de la donnée non saine (NS)')
    plt.legend(loc='center')

    plt.subplot(the_grid[1,0], aspect=1)


    for lab, col, m in zip(('S', ''), ('lime', 'red'), ('o', '')):
        plt.scatter(Y[_var_prediction==lab, 0], Y[_var_prediction==lab, 1], label=lab, c=col, marker=m)
    plt.xlabel('Composante principale '+str(1+_num_comp_inf))
    plt.ylabel('Composante principale '+str(1+_num_comp_sup))
    plt.title(u'Mise en valeur de la donnée saine (S)')
    plt.legend(loc='center')

    plt.subplot(the_grid[0,1], aspect=1)

    for lab, col, m in zip(('', 'NS'), ('lime', (1,0,0,0.3)), ('', '^')):
        plt.scatter(Y[_var_prediction==lab, 0], Y[_var_prediction==lab, 1], label=lab, c=col, marker=m)
    plt.xlabel('Composante principale '+str(1+_num_comp_inf))
    plt.ylabel('Composante principale '+str(1+_num_comp_sup))
    plt.title(u'Mise en valeur de la donnée non saine (NS)')
    plt.legend(loc='center')

    plt.subplot(the_grid[1,0], aspect=1)


    for lab, col, m in zip(('S', ''), ('lime', 'red'), ('o', '')):
        plt.scatter(Y[_var_prediction==lab, 0], Y[_var_prediction==lab, 1], label=lab, c=col, marker=m)
    plt.xlabel('Composante principale '+str(1+_num_comp_inf))
    plt.ylabel('Composante principale '+str(1+_num_comp_sup))
    plt.title(u'Mise en valeur de la donnée saine (S)')
    plt.legend(loc='center')

    plt.subplot(the_grid[1,1], aspect=1)

    for lab, col, m in zip(('', 'NA'), ('lime', 'blue'), ('', '+')):
        plt.scatter(Y[_var_prediction==lab, 0], Y[_var_prediction==lab, 1], label=lab, c=col, marker=m)
    plt.xlabel('Composante principale '+str(1+_num_comp_inf))
    plt.ylabel('Composante principale '+str(1+_num_comp_sup))
    plt.title(u'Mise en valeur de la donnée non renseignée (NA)')
    plt.legend(loc='center')

    plt.show()
```

```
def getDfContinuesSkewedSansNaNNormaliseSain(_df):
    dfOriginelSansAberr = _df.copy() #15666666666
    solide = np.where(_df["quantity"].str.contains("l|L")==False, 1, 0)
    liquide = np.where(_df["quantity"].str.contains("l|L")==True, 1, 0)
    dsl = {'solide' : solide, 'liquide' : liquide}
    dfsl = pd.DataFrame(dsl)
    dfOriginelS = pd.concat([dfOriginelSansAberr, dfsl], axis = 1)
    dfOriginelS = dfOriginelS.set_index('code', drop=False)
    dfOriginelS['proteins_100g'][15666666666] = 0
    dfOriginelS['fat_100g'][15666666666] = 0
    dfOriginelS['carbohydrates_100g'][15666666666] = 0
    dfOriginelS = dfOriginelS[dfOriginelS.code <> 3401528535864]
    dfContinuesSkewed, dfContinues, dfNonContinues = getDfContinuesSkewed(dfOriginelS)
    dfContinuesSkewedSansNaN = dfContinuesSkewed.fillna(0)
    dfContinuesSkewedSansNaNNormaliseSain = prep.StandardScaler().fit_transform(dfContinuesSkewedSansNaN)
    return dfOriginelS, dfContinuesSkewedSansNaNNormaliseSain

def no_null_objects(data, columns=None):
    """
    selects rows with no NaNs
    """
    if columns is None:
        columns = data.columns
    return data[np.logical_not(np.any(data[columns].isnull().values, axis=1))]

def splitDataFrameList(df, target_column, separator):
    ''' df = dataframe to split,
    target_column = the column containing the values to split
    separator = the symbol used to perform the split

    returns: a dataframe with each entry for the target column separated, with each element moved into a new row.
    The values in the other columns are duplicated across the newly divided rows.
    '''
    def splitListToRows(row, row_accumulator, target_column, separator):
        split_row = row[target_column].split(separator)
        for s in split_row:
            new_row = row.to_dict()
            new_row[target_column] = s[3:]
            row_accumulator.append(new_row)
    new_rows = []
    df.apply(splitListToRows,axis=1,args = (new_rows,target_column,separator))
    new_df = pd.DataFrame(new_rows)
    return new_df

def graphePaysRepresentation(_df):
    pays_representes = splitDataFrameList(no_null_objects(_df, ["countries_tags"]), "countries_tags", ",")
    pays_nombre = pays_representes["countries_tags"].value_counts()
    pays_nombre[:20][::-1].plot.barh()


# fonctions du calcul des points A et C :
# elles sont basées sur les données du site https://fr.openfoodfacts.org/score-nutritionnel-experimental-france
def getAenergyPoints(val):
    if val <= 335:    return 0
    elif val <= 670:  return 1
    elif val <= 1005: return 2
    elif val <= 1340: return 3
    elif val <= 1675: return 4
    elif val <= 2010: return 5
    elif val <= 2345: return 6
    elif val <= 2680: return 7
    elif val <= 3015: return 8
    elif val <= 3350: return 9
    else:             return 10
def getAsaturatedfatPoints(val):
    if val <= 1:   return 0
    elif val <= 2: return 1
    elif val <= 3: return 2
    elif val <= 4: return 3
    elif val <= 5: return 4
    elif val <= 6: return 5
    elif val <= 7: return 6
    elif val <= 8: return 7
    elif val <= 9: return 8
    elif val <= 9: return 9
    else:          return 10
def getAsugarsPoints(val):
    if val <= 4.5:    return 0
    elif val <= 9:    return 1
    elif val <= 13.5: return 2
    elif val <= 18:   return 3
    elif val <= 22.5: return 4
    elif val <= 27:   return 5
    elif val <= 31:   return 6
    elif val <= 36:   return 7
    elif val <= 40:   return 8
    elif val <= 45:   return 9
    else:             return 10
def getCfruitsPoints(val):
    if   val <= 40: return 0
    elif val <= 60: return 1
    elif val <= 80: return 2
    else:           return 5
```

```
def getAsodiumPoints(val):
    if   val <= 0.090:  return 0
    elif val <= 0.180:  return 1
    elif val <= 0.270:  return 2
    elif val <= 0.360:  return 3
    elif val <= 0.450:  return 4
    elif val <= 0.540:  return 5
    elif val <= 0.630:  return 6
    elif val <= 0.720:  return 7
    elif val <= 0.810:  return 8
    elif val <= 0.900:  return 9
    else:               return 10
def getCfibresPoints(val):
    if   val <= 0.7: return 0
    elif val <= 1.4: return 1
    elif val <= 2.1: return 2
    elif val <= 2.8: return 3
    elif val <= 3.5: return 4
    else:            return 5
def getCproteinsPoints(val):
    if   val <= 1.6: return 0
    elif val <= 3.2: return 1
    elif val <= 4.8: return 2
    elif val <= 6.4: return 3
    elif val <= 8:   return 4
    else:            return 5

def getScoreRecette(_df, _csvReader):
    """
    Méthode qui calcule le score d'une recette
    Arguments:
    _df -- le dataframe contenant les valeurs
    _csvReader -- le reader de csv permettant de lire les proportions des articles de la recette
    Retour: les deux scores de la recette, solide et liquide
    """
    print(_df.shape)
    # on supprime tous les articles qui n'ont pas de score
    dfOriginelScore = _df[np.isnan(_df['nutrition-score-uk_100g']) == False]
    print(dfOriginelScore.shape)
    # on ne garde que les 7 variables utiles + code + quantity + score
    dfVarScore = dfOriginelScore[['code', 'energy_100g', 'saturated-fat_100g', 'sugars_100g', 'sodium_100g', 'fiber_100g', 'p
roteins_100g', 'fruits-vegetables-nuts_100g', 'quantity', 'nutrition-score-uk_100g']]
    # on détermine si l'article est solide ou liquide (ou aucun des deux)
    solide = np.where(dfOriginelScore["quantity"].str.contains("l|L")==False, 1, 0)
    liquide = np.where(dfOriginelScore["quantity"].str.contains("l|L")==True, 1, 0)
    dsl = {'solide' : solide, 'liquide' : liquide}
    dfsl = pd.DataFrame(dsl)
    dfVarScoreQuality = pd.concat([dfVarScore, dfsl], axis = 1)
    # on passe code en index afin d'utiliser les codes
    dfVarScoreQuality = dfVarScoreQuality.set_index('code', drop=False)

    # Le fichier recette.csv est fourni avec les livrables, mais on peut choisir ce qu'on veut du moment que les articles cho
isis
    # possèdent toutes les valeurs nécessaires (en premier lieu le score, normalement si le score est présent alors toutes le
s autres variables sont également présentes)
    # Contenu du fichier fourni :
    # 3560070464791\t100\n3329770044654\t100\n4008471483083\t200\n3451790397509\t50\n3173610005028\t100\n3300060420105\t50
    # une colonne avec les 5 codes d'article, l'autre colonne avec les 5 portions de ces articles (en grammes)

    listeCodes = []
    listeQuantities = []
    for row in _csvReader:
        listeCodes.append(int((row[0].split('\t'))[0]))
        listeQuantities.append(int((row[0].split('\t'))[1]))
    print("liste des codes", listeCodes)
    print("liste des portions", listeQuantities)

    # calcul des moyennes des variables utilisées dans le score britannique pour 100g de recette
    moyennes_solide = dict()
    moyennes_liquide = dict()

    for varName in ['fruits-vegetables-nuts_100g', 'energy_100g', 'saturated-fat_100g', 'sugars_100g', 'sodium_100g', 'fiber_
100g', 'proteins_100g']:
        cumul_solide = cumul_liquide = 0
        nb_solide = nb_liquide = 0
        moy_solide = moy_liquide = 0
        cumul_portion_solide = cumul_portion_liquide = 0
        for code, portion in zip(listeCodes, listeQuantities):
            if not re.search("L|l", dfVarScoreQuality.ix[code,'quantity']):
                nb_solide = nb_solide + 1
                cumul_portion_solide = cumul_portion_solide + portion
                if varName == 'fruits-vegetables-nuts_100g':
                    cumul_solide = cumul_solide + dfVarScoreQuality.ix[code,varName] * portion
                else:
                    cumul_solide = cumul_solide + dfVarScoreQuality.ix[code,varName] * portion / 100.0
            elif re.search("L|l", dfVarScoreQuality.ix[code,'quantity']):
                nb_liquide = nb_liquide + 1
                cumul_portion_liquide = cumul_portion_liquide + portion
                if varName == 'fruits-vegetables-nuts_100g':
                    cumul_liquide = cumul_liquide + dfVarScoreQuality.ix[code,varName] * portion
                else:
                    cumul_liquide = cumul_liquide + dfVarScoreQuality.ix[code,varName] * portion / 100.0

        if nb_solide > 0:
            if varName == 'fruits-vegetables-nuts_100g':
                moy_solide = cumul_solide / cumul_portion_solide
```

```
            else:
                moy_solide = cumul_solide * 100.0 / cumul_portion_solide
        if nb_liquide > 0:
            if varName == 'fruits-vegetables-nuts_100g':
                moy_liquide = cumul_liquide / cumul_portion_liquide
            else:
                moy_liquide = cumul_liquide * 100.0 / cumul_portion_liquide

        moyennes_solide[varName]= moy_solide
        moyennes_liquide[varName] = moy_liquide

    # SOLIDE
    scoreSolideA = getAenergyPoints(moyennes_solide["energy_100g"]) + getAsaturatedfatPoints(moyennes_solide["saturated-fat_1
00g"]) + getAsugarsPoints(moyennes_solide["sugars_100g"]) + getAsodiumPoints(moyennes_solide["sodium_100g"])
    scoreSolideC = getCfruitsPoints(moyennes_solide["fruits-vegetables-nuts_100g"]) + getCfibresPoints(moyennes_solide["fiber
_100g"]) + getCproteinsPoints(moyennes_solide["proteins_100g"])

    scoreSolide = 0
    if scoreSolideA < 11:
        scoreSolide = scoreSolideA - scoreSolideC
    else:
        if getCfruitsPoints(moyennes_solide["fruits-vegetables-nuts_100g"]) == 5:
            scoreSolide = scoreSolideA - scoreSolideC
        else:
            scoreSolide = scoreSolideA - (moyennes_solide["fruits-vegetables-nuts_100g"]+moyennes_solide["fiber_100g"])

    # LIQUIDE
    scoreLiquideA = getAenergyPoints(moyennes_liquide["energy_100g"]) + getAsaturatedfatPoints(moyennes_liquide["saturated-fa
t_100g"]) + getAsugarsPoints(moyennes_liquide["sugars_100g"]) + getAsodiumPoints(moyennes_liquide["sodium_100g"])
    scoreLiquideC = getCfruitsPoints(moyennes_liquide["fruits-vegetables-nuts_100g"]) + getCfibresPoints(moyennes_liquide["fi
ber_100g"]) + getCproteinsPoints(moyennes_liquide["proteins_100g"])
    scoreLiquide = 0
    if scoreLiquideA < 11:
        scoreLiquide = scoreLiquideA - scoreLiquideC
    else:
        if getCfruitsPoints(moyennes_liquide["fruits-vegetables-nuts_100g"]) == 5:
            scoreLiquide = scoreLiquideA - scoreLiquideC
        else:
            scoreLiquide = scoreLiquideA - (moyennes_liquide["fruits-vegetables-nuts_100g"]+moyennes_liquide["fiber_100g"])

    return scoreSolide, scoreLiquide

def text_to_wordlist(text, remove_stop_words=False, stem_words=False):
    # Clean the text, with the option to remove stop_words and to stem words.

    # Clean the text
    text = re.sub(r"[^A-Za-z0-9]", " ", text)

    # Remove punctuation from text
    text = ''.join([c for c in text if c not in punctuation])

    # Optionally, remove stop words
    if remove_stop_words:
        text = text.split()
        text = [w for w in text if not w in stop_words]
        text = " ".join(text)

    # Optionally, shorten words to their stems
    if stem_words:
        text = text.split()
        stemmer = SnowballStemmer('english')
        stemmed_words = [stemmer.stem(word) for word in text]
        text = " ".join(stemmed_words)

    # Return a list of words
    return(text)
def process_sent(words, stemmer, lemmer, alpha_tokenizer, lemmatize=True):
    words = words.lower()
    words = text_to_wordlist(words)
    tokens = alpha_tokenizer.tokenize(words)
    for index, word in enumerate(tokens):
        if lemmatize:
            tokens[index] = lemmer.lemmatize(word)
        else:
            tokens[index] = stemmer.stem(word)
    return tokens

def modelesCalculScore(_df):

    # liste des ingrédients
    _df = _df.iloc[_df['nutrition-score-uk_100g'].dropna().index.values]
    ingedients = _df.select_dtypes(include=['object'])['ingredients_text']
    ingedients = pd.DataFrame(ingedients.dropna()).reset_index(drop=True)

    stemmer =  LancasterStemmer()
    lemmer = WordNetLemmatizer()
    stop = stopwords.words('english')
    alpha_tokenizer = RegexpTokenizer('[A-Za-z]\w+')

    texts = np.concatenate([ingedients.ingredients_text.values])
    corpus = [process_sent(sent, stemmer, lemmer, alpha_tokenizer) for sent in texts]
```

```
    VECTOR_SIZE = 100
    min_count = 5
    size = VECTOR_SIZE
    window = 100

    model_lemmatized = models.Word2Vec(corpus, min_count=min_count, size=size, window=window)

    # variables indiquant si les articles contiennent ou non les ingrédients
    clefs = list(model_lemmatized.wv.vocab.keys())
    list_var = []
    for i in range(len(clefs)):
        _df["pr_"+str(clefs[i])] = np.where(_df["ingredients_text"].str.contains(clefs[i])==True, 1, 0)
        list_var.append("pr_"+str(clefs[i]))

    # modèles de Random Forest et de régression linéaire
    data_input = _df[list_var]

    # Entraînement avec tous les enregistrements sauf les 1000 derniers
    X_train = data_input[:-1000]
    Y_train = _df['nutrition-score-uk_100g'][:-1000]
    # Test des 1000 derniers enregistrements
    X_test = data_input[-1000:]
    Y_test = _df['nutrition-score-uk_100g'][-1000:]

    # Modèle de l'algorithm Random Forest
    mod = RandomForestRegressor(n_estimators = 10, n_jobs = -1, min_samples_split=3)
    mod.fit(X_train, Y_train)
    Y_pred = mod.predict(X_test)
    print("RandomForest r2_score : {0}".format(r2_score(Y_pred, Y_test)))
    print("RandomForest mean_squared_error : {0}".format(mean_squared_error(Y_pred, Y_test)))

    # Modèle de régression linéaire
    lr = linear_model.LinearRegression()
    lr.fit(X_train, Y_train)
    print("Régression linéaire r2_score : {0}".format(r2_score(lr.predict(X_test), Y_test)))

def getDfOriginelSain(_df):

    dfOriginelSansAberr = _df.copy() #15666666666
    solide = np.where(_df["quantity"].str.contains("l|L")==False, 1, 0)
    liquide = np.where(_df["quantity"].str.contains("l|L")==True, 1, 0)
    dsl = {'solide' : solide, 'liquide' : liquide}
    dfsl = pd.DataFrame(dsl)
    dfOriginelS = pd.concat([dfOriginelSansAberr, dfsl], axis = 1)

    dfOriginelS = dfOriginelS.set_index('code', drop=False)
    dfOriginelS['proteins_100g'][15666666666] = 0
    dfOriginelS['fat_100g'][15666666666] = 0
    dfOriginelS['carbohydrates_100g'][15666666666] = 0
    dfOriginelS = dfOriginelS[dfOriginelS.code <> 3401528535864]
    return dfOriginelS
```