Projet 2 OC : concevoir une application au service de la santé publique

Présenté par : Guy Anthony Nama Nyam

Mentor : Julien Hendrick

5 novembre 2019

OPENCLASSROOMS

Sommaire

Contexte

- La population et les besoins alimentaires mondiaux s'accroissent fortement au fil des années.
- Toutes organisations(particuliers, entreprises) tirent profit pour proposer de nombreux produits alimentaires.
- La quantité d'information de part le nombre de produits ainsi que leurs compositions devient rapidement une difficulté pour les consommateurs finaux.
- L'objectif pour nous est de pourvoir proposer un système de recommandation de recherche de produit par l'utilisateur avec des choix plus informés de santé.

Source de données

- La source utilisée est la base ***Open Food Facts*** disponible sous licence *Open Database License*.
- Les contenus individuels de la base de données sont disponibles sous la licence *Database Contents License*.
- Les photos de produits sont disponibles sous la licence *Creative Commons Attribution Partage à l'identique*.
- *Open Food Facts* répertorie les produits alimentaires du monde entier, tout le monde peut les utiliser pour tout usage, et contribuer en y ajoutant des produits.
- Dans un premier temps, la suite de notre travail consistera par décrire la source de données, suivi du nettoyage de la donnée.
- Dans un second temps, de réaliser une analyse exploratoire, de proposer une application et enfin nous conclurons.

Sommaire

Description du fichier de données

- le type du fichier de données est un fichier .csv
- Le fichier utilisé est ***en.openfoodfacts.org.products.csv*** de taille ***1 922 868 Ko*** téléchargeable sur le site de *Open food facts*.
- Les produits ajoutés vont ***31/01/2012T14 :43 :58*** au ***09/09/2019T00 :22 :22***.
- L'encodage du fichier est l'Unicode UTF-8. Le caractère de séparation des champs est ***tabulation (\t)***.

Description des variables

Généralités sur les variables se terminant par :

- _t pour des dates au format UNIX timestamp, _datetime pour des dates au format iso8601
- _100g pour le nombre de constituant en g dans 100g du produit, _{fr|en|..} le code de la langue sur 2 lettres
- _tags pour la liste des tags sur une variable pris par un produit donnée.

Les variables sont séparés en 4 sections :

- Les informations générales sur la fiche du produit : nom, date de modification, nom générique, etc..
- Un ensemble de tags : catégorie du produit, origine, etc.
- Les ingrédients composant les produits et leurs additifs éventuels.
- Des informations nutritionnelles : quantité en grammes d'un nutriment pour 100 grammes du produit.

Importation des données

- Importation de la bibliothèque **pandas**.
- Lecture du fichier source de données :
  df = pd.read_csv("en.openfoodfacts.org.products.csv", delimiter="\t")
- Afficher :
  les 5 premières lignes - print(df.head()) ou print(df[:5])
  les 5 dernières lignes - print(df.tail()) ou print(df[-5 :])
- Informations sur le nombre de lignes et de colonnes :
  print(df.shape). soient 951409 lignes ou entrés et 175 colonnes ou variables
- Afficher :
  l'ensemble des variables - print(list(df.columns))
  plusieurs colonnes - print(df[['creator','product_name']])

Importation des données

- Le typage des colonnes s'obtient par : print(df.dtypes). 2 entiers(created_t, last_modified_t) qui sont des variables quantitatives continues,
117 flottants dont la plupart sont les variables se terminant par _100g, et 56 textes pour les autres.

- La fonction **df.isna()** permet d'obtenir les valeurs manquantes(None ou Numpy.NaN). 8 variables entières renseignées dominées par les méta-données et 14 variables sans informations.

In [6]:
```
def sans_info(data=df):
    sans_info =[]
    for f in data.columns:
        if data[f].isna().all():
            sans_info.append(f)
    print("Les variables sans informations : {} \n".format(sans_info))
```

In [7]:
```
def full_info(data=df):
    full_info = []
    for f in data.columns:
        if data[f].notna().all():
            full_info.append(f)
    print("Les variables complètes : {}".format(full_info))
```

In [8]:
```
sans_info()
full_info()
```

```
Les variables sans informations : ['cities', 'allergens_en', 'no_nutriments', 'ingredients_from_palm_oil', 'ingredients_that_may_be_from_palm_oil', '-caproic-acid_100g', '-lignoceric-acid_100g', '-melissic-acid_100g', '-elaidic-acid_100g', '-gondoic-acid_100g', '-mead-acid_100g', '-erucic-acid_100g', '-nervonic-acid_100g', 'water-hardness_100g']

Les variables complètes : ['code', 'url', 'created_t', 'last_modified_t', 'last_modified_datetime', 'states', 'states_tags', 'states_en']
```

Importation des données

- Le taux de valeurs manquantes de la donnée : 78.996%

```
In [42]: #function pourcentage de données manquantes dans les données
         def percent_missed_values(data=df):
             somme = 0
             for f in list(data.columns):
                 somme += df[f].isna().sum()
             return (somme / (data.shape[0]*data.shape[1]))*100
```

```
In [43]: percent_missed_values()

Out[43]: 78.99605982885835
```

- Remplacer les valeurs manquantes par zéro.

```
In [6]: #Selectionner les variables se terminant par _100g
        def endswith_100g(data = df):
            features_100g = []
            for f in data.columns:
                if f.endswith('_100g'):
                    features_100g.append(f)
            return features_100g
```

```
In [7]: #Conserver les variables qui ont un sens à remplacer à zéro
        features_zero_100g = endswith_100g()
        features_zero_100g.remove("ph_100g")
        features_zero_100g.remove("nutrition-score-fr_100g")
        features_zero_100g.remove("nutrition-score-uk_100g")
```

```
In [8]: #Remplacer les valeurs manquantes par des zeros pour les variables ayant un sens
        def miss_by_0(features, data=df):
            for f in features:
                data[f].fillna(0, inplace=True)
```

```
In [9]: miss_by_0(features_zero_100g)
```

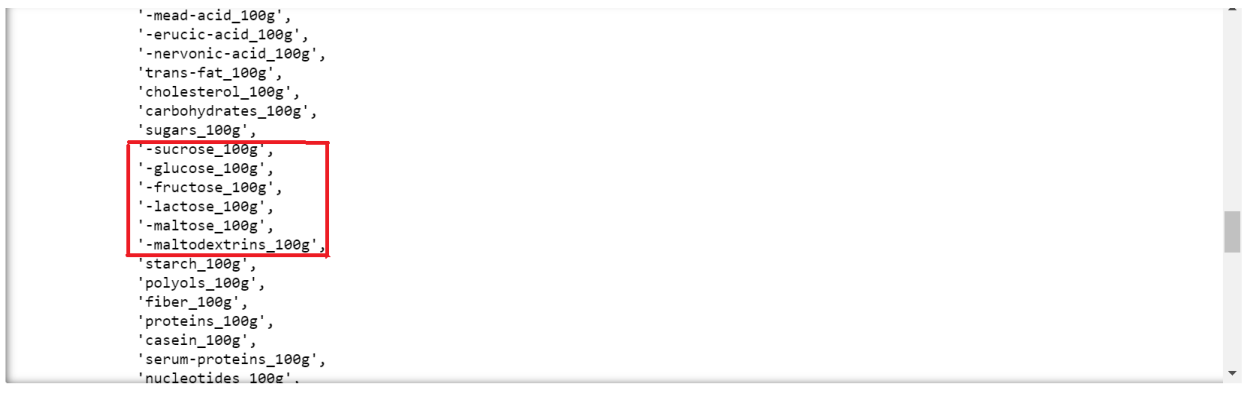
Importation des données I

Règles de suppression des lignes de somme supérieure à 100g.

1. Ne pas prendre en compte les champs suivants :
   - energy_100g, energy-from-fat_100g qui sont des énergies.
   - nutrition-score-fr_100g, nutrition-score-uk_100g qui sont des valeurs calculées sur les nutriments.
   - ph_100g qui mesure l'acidité.
2. Ignorer les sous éléments :

Hypothèse 1

Il y'a corrélation entre un élément principal et ses sous éléments



Importation des données II

3. Ne pas prendre en compte la variable "sodium_100g"

Hypothèse 2

Il y'a duplication d'information avec la variable "salt_100g"

4. Ramener à 100 les éléments supérieur à 100g car n'influence pas dans le calcul du nutriscore qui est connexe à notre sujet d'application.

5. Suppression des lignes de somme supérieure à 100g

```
df.drop(df[df[features_sum_100g].sum(axis=1) > 100].index, inplace=True)
```

Importation des données

- Le nombre de lignes après suppression est de 708960 soit 74.52% de la donnée initiale.
- Suppression des méta-données :

df.drop(['created_t', 'created_datetime', 'last_modified_t', 'last_modified_datetime'], axis=1, inplace=True)

- Garder les variables connexes au sujet d'application : 16 variables.

```
In [17]: #features retenues
features_ret = ['code','product_name', 'brands','ingredients_text', 'allergens','additives',
                'nutrition_grade_fr', 'energy_100g','fat_100g','saturated-fat_100g','sugars_100g',
                'fiber_100g','proteins_100g','salt_100g','fruits-vegetables-nuts_100g','nova_group']
```

```
In [18]: df = df[features_ret]
```

Sommaire

Détection d'erreurs

- Détection des variables inutiles et suppression.

fonction	del_var_useless(data, seuil=1)
	Suppression des variables. inutiles
paramètre(s)	data : DataFrame
	seuil : seuil de suppression. compris entre 0 et 1
retour(s)	Aucun

```
In [4]: #supprimer les variables inutiles
        def del_var_useless(data, seuil=1):
            var_useless = []
            for f in list(data.columns):
                if (data[f].isna().sum()/len(data)) > seuil:
                    data.drop(f, axis=1, inplace=True)
                    var_useless.append(f)
            print("Colonnes supprimées {}".format(var_useless))
```

```
In [5]: del_var_useless(df, 0.9)

        Colonnes supprimées ['allergens', 'additives']
```

- Suppression des lignes dont élément(100g) négatif

fonction	del_rows_val_neg(data=df)
	Supprimer les lignes de valeurs _100g négatives
paramètre(s)	data : DataFrame
retour(s)	Aucun

Détection d'erreurs


```
In [6]: #supprimer des valeurs nutritionnelles négatives
        def del_rows_val_neg(data=df):
            size = data.shape[0]
            fs_pos = ['energy_100g', 'fat_100g', 'saturated-fat_100g', 'sugars_100g', 'fiber_100g',
                      'proteins_100g', 'salt_100g','fruits-vegetables-nuts_100g']
            for f in fs_pos:
                data.drop(data.loc[data[f] < 0].index, inplace=True)
            dif = size -len(data)
            print("Nombre de ligne supprimer: {}, nombre de lignes restantes : {}".format(dif, data.shape[0]))

In [7]: del_rows_val_neg(df)

        Nombre de ligne supprimer: 24, nombre de lignes restantes : 708936
```

- Aucune valeur aberrante pour nutriscore et nova groupe

```
In [160]: df['nutrition_grade_fr'].unique()

Out[160]: array([nan, 'a', 'c', 'e', 'b', 'd'], dtype=object)

In [161]: df['nova_group'].unique()

Out[161]: array([nan, 4., 3., 2., 1.])
```

- Suppression de doublon pour l'identifiant "code"

fonction	delete_doublon_better(data=df) Suppression des doublons en fonction le nutri-grade
paramètre(s)	data : DataFrame
retour(s)	Aucun

Détection d'erreurs

In [25]:
```
#conserver dans le meilleur des cas celui qui possède son nutriscore
def delete_doublon_better(data=df):
    size = len(data)
    index = list(data['code'].value_counts().index)
    valeurs = list(data['code'].value_counts())

    list_sup_one = []#liste des index de code supérieur à 1
    for i in range(0, len(index)):
        if valeurs[i] > 1:
            list_sup_one.append(index[i])
    nb_nutri_sauv = 0
    for index in list_sup_one:#parcours de chaque index pour sauvegarder ceux qui possède l'info nutrigrade
        count = 0
        nb_with_na = len(df[(df['code'] == index) & (df['nutrition_grade_fr'].isna())])
        nb_code = len(df[df['code'] == index])
        if nb_code != nb_with_na:
            df.drop(labels=df[(df['code'] == index) & (df['nutrition_grade_fr'].isna())].index, axis=0)
            nb_nutri_sauv += 1
    data.drop_duplicates(subset=['code'], inplace=True)
    dif = size -len(data)
    print("Nombres de doublons supprimés : {}, nombres de lignes restantes : {}".format(dif, data.shape[0]))
    print("Nombres de nutriscores sauvegardés : {}".format(nb_nutri_sauv))
```

In [27]:
```
delete_doublon_better()
```

```
Nombres de doublons supprimés : 339, nombres de lignes restantes : 708597
Nombres de nutriscores sauvegardés : 66
```

Traiter les valeurs manquantes

- Remplacer les valeurs manquantes par le caractère espace(" ") pour les données textes suivantes : "product_name", "brands", "ingredients_text".

In [46]:
```
features_to_empty = ['product_name', 'brands', 'ingredients_text', "product"]
def miss_by_empty(features, data=df):
    for f in features:
        data[f].fillna(" ", inplace=True)

miss_by_empty(features_to_empty)
```

Traiter les données textes

- Suppression des ponctuations, chiffres et minuscules.

In [39]:
```
#supprimer les ponctuations, les accents et les chiffres
def del_ponct(val):
    if type(val) == str:# éviter les nan
        val = val.lower()
        val = re.compile('[éèëë]+').sub("e", val)
        val = re.compile('[àâä]+').sub("a", val)
        val = re.compile('[ùûü]+').sub("u", val)
        val = re.compile('[îï]+').sub("i", val)
        val = re.compile('[ôö]+').sub("o", val)
        return re.compile('[^A-Za-z" "]+').sub("", val)
    return val

def data_text_del_ponct(data=df):
    listCol = list(data.columns)
    listCol.remove("code")
    listCol.remove("product")
    for f in listCol:
        if (data[f].dtype == "object"):
            data[f] = data[f].apply(del_ponct)
```

In [40]:
```
data_text_del_ponct(df)
```

- Lemmatisation de la donnée texte : racinisation de **Snowball**

In [42]:
```
#Lemmatisation snowball
stemmer = FrenchStemmer(ignore_stopwords=True)
def stem(expr):
    words_stems = []
    if type(expr) == str:#eviter les nan
        expr_words = nltk.word_tokenize(expr)
        for word in expr_words:
            words_stems.append(stemmer.stem(word))
        return " ".join(words_stems)
    return expr
```

In [43]:
```
def data_text_lemma(data=df):
    listCol = list(data.columns)
    listCol.remove("code")
    listCol.remove("product")
    for f in listCol:
        if (data[f].dtype == "object"):
            data[f] = data[f].apply(stem)
```

In [44]:
```
data_text_lemma()
```

Sommaire

Description

- **df.describe(include='all')** pour les statistiques basique
- df.describe() stats sur les variables quantitatives en image

```
In [3]: df.describe()
```

Out[3]:

	energy_100g	fat_100g	saturated-fat_100g	sugars_100g	fiber_100g	proteins_100g	salt_100g	fruits-vegetables-nuts_100g	nova_group	predicted_gra
count	7.085970e+05	708597.000000	708597.000000	708597.000000	708597.000000	708597.000000	708597.000000	708597.000000	204079.000000	708597.0000
mean	9.406699e+36	6.970962	2.535322	3.741359	0.514068	6.323541	1.101110	0.109821	3.522709	0.0202
std	7.918390e+39	11.762543	4.960635	6.819252	2.159911	9.401696	4.833964	2.260011	0.877548	0.1408
min	0.000000e+00	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	1.000000	0.0000
25%	0.000000e+00	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	3.000000	0.0000
50%	3.800000e+02	1.000000	0.100000	0.700000	0.000000	1.900000	0.100000	0.000000	4.000000	0.0000
75%	1.046000e+03	9.380000	2.500000	4.100000	0.000000	9.860000	1.153160	0.000000	4.000000	0.0000
max	6.665559e+42	100.000000	93.100000	100.000000	100.000000	100.000000	100.000000	100.000000	4.000000	1.0000

- La distribution d'une variable

fonction	distribution(feature, data=df) Distribution de la variable feat
paramètre(s)	data : DataFrame feat : le nom de la colonne
retour(s)	Dataframe de distribution

Description variable quantitative


```
In [4]: def distribution(feature, data=df):
            dist = data[feature].value_counts()
            d = pd.DataFrame(dist.index, columns=[feature])
            d["effectifs"] = dist.values
            d["frequences"] = dist.values/len(data)
            d["frequences cumulees"] = d["frequences"].cumsum()
            return d
```

```
In [5]: distribution('brands')
```

Out[5]:

	brands	effectifs	frequences	frequences cumulees
0		300758	0.424442	0.424442
1	carrefour	7566	0.010677	0.435119
2	auchan	6362	0.008978	0.444097
3	u	3901	0.005505	0.449603
4	delhaiz	3114	0.004395	0.453997
5	lead pric	2996	0.004228	0.458225
6	casino	2717	0.003834	0.462060
7	cor	2190	0.003091	0.465150
8	nestl	2188	0.003088	0.468238
9	hacendado	2038	0.002876	0.471114

Fig 1. Distribution de probabilté pour la variable "brands"

- Distribution des observations

fonction dist_graph(feature, data=df)
Distribution graphique d'une variable

paramètre(s) feature : nom de la variable
data : DataFrame

retour(s) aucun

```
In [6]: import seaborn as sns
        sns.set()
        #Affiche le distribution graphique d'une variable quantitative
        def dist_graph(feature, data=df):
            if data[feature].dtype == "float":
                sns.distplot(data[data[feature] != 0][feature])#On ignore la valeur 0
```

```
In [7]: #d_nova = df[df['nova_group'].notna()]
        dist_graph('proteins_100g')
```

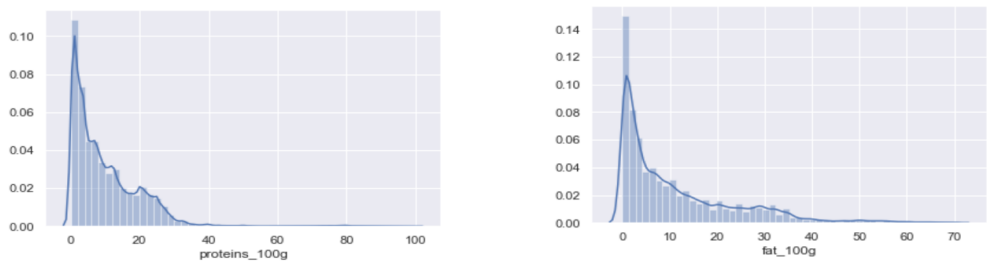


Fig 2. graphe distribution des variables "proteins" et "fat"

Description variable quantitative

- Boites à moustaches des variables.

fonction	data_pai_plot(list_feature, data=df) Boites à moustaches des variables
paramètre(s)	list_feature : listes des variables data : DataFrame
retour(s)	aucun

```
In [8]: #boites à moustaches variable quantitative
sns.set()
def data_pai_plot(list_feature, data=df):
    d = pd.DataFrame()
    for f in list_feature:
        if data[f].dtype == "float":
            d[f] = data[data[f] != 0][f] #ignorer la valeur 0 dominante
    sns.boxplot(data=d, palette="colorblind", orient="h")
```

```
In [9]: fs = ['fat_100g','saturated-fat_100g','sugars_100g','fiber_100g',
       'proteins_100g', 'salt_100g','fruits-vegetables-nuts_100g']

data_pai_plot(fs)
```

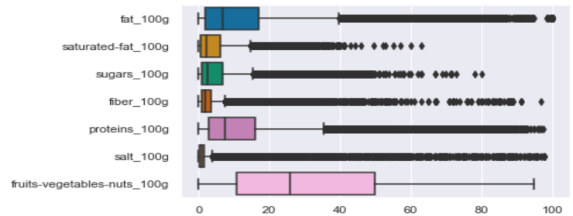


Fig 3. Boîtes à moustaches des éléments nutritionnels du nutriscore

Hypothèse 3

On note la présence de valeurs aberrantes sur les variables quantitatives précédentes

Description variable quantitative

- Test de Grubb sur les valeurs aberrantes visualisées.

fonction	test_grubb(feature, data=df, confiance = 0.05)
	test de Grubb sur les valeurs aberrantes Q3 + IQ
paramètre(s)	feature : la variable à tester
	data : DataFrame
	confiance : intervalle de confiance
retour(s)	retourne la valeur aberrante minimale

In [148]:
```
#Test de Grubb pour confirmer la valeur min anormale
def test_grubb(feature, data=df, confiance = 0.05):
    obs = []
    liste = list_sup_aberante(feature, data)
    size = len(data)
    Gcritical_value = calculate_critical_value(size, confiance)
    for i, row in data.iterrows():
        obs.append(data.loc[i,feature])
    obs.sort()
    mean = np.mean(obs)
    s = np.std(obs)
    val = 0
    for i in range(0, len(liste)):
        abs_val = abs(liste[i] - mean)
        Gcalculated = abs_val / s
        if Gcalculated > Gcritical_value:
            print("la valeur minimale abérrante : {}".format(liste[i]))
            val = liste[i]
            break
    #data.drop(data[data[feature] >= val].index, inplace=True)
    for i, row in data.iterrows():
        if data.loc[i,feature] > val:
            data.loc[i,feature] = mean#imputation par la moyenne
    return val
```

In [114]:
```
#renvoyer une liste de valeurs supposées aberantes
def list_sup_aberante(feature, data=df):
    q = df[feature].quantile([0.25,0.5,0.75])
    IQ = q[0.75] - q[0.25]
    liste = []
    for i, row in data.iterrows():
        if (data.loc[i,feature] < (q[0.25] - 1.5*IQ)) or (data.loc[i,feature] > (q[0.75] + 1.5*IQ)):
            liste.append(data.loc[i,feature])
    return sorted(set(liste))
```

In [149]:
```
def calculate_critical_value(size, alpha):
    t_dist = stats.t.ppf(1 - alpha / (2 * size), size - 2)
    numerateur = (size - 1) * np.sqrt(np.square(t_dist))
    denominateur = np.sqrt(size) * np.sqrt(size - 2 + np.square(t_dist))
    critical_value = numerateur / denominateur
    print("La valeur critique de Grubb : {}". format(critical_value))
    return critical_value
```

- ***L'hypothèse 3*** est confirmée par le test de Grubb, néanmoins les valeurs considérées "aberrantes" sont conservées car présentes dans certains produits.

Description variable qualitative

- Distribution des observations pour les variables qualitatives pertinentes de nutriscore et de nova groupe.

fonction	pie_chart_qual(features, data=df) Pie chart nutri-grade et nova groupe
paramètre(s)	features : les variables nutrigrade et nova groupe data : DataFrame
retour(s)	aucun

```
In [10]: #variable qualitative
def pie_chart_qual(features, data=df):
    fig, axes = plt.subplots(nrows=1, ncols=len(features), figsize=(15, 4))
    i = 0
    for f in features:
        labels = []
        sizes = list(df[f].value_counts().sort_index())
        explode = [0] * len(sizes)
        explode[sizes.index(max(sizes))] = 0.1
        if f == "nutrition_grade_fr":
            cs = ["green", "lime", "yellow", "orange", "red"]
            labels = ["a", "b", "c", "d", "e"]
        if f == "nova_group":
            cs = ["lime", "yellow", "orange", "red"]
            labels = ["1", "2", "3", "4"]
        axes[i].pie(sizes, explode=explode, labels=labels, autopct='%1.1f%%', shadow=True, startangle=90, colors=cs)
        i = i + 1
    j = 0
    for ax in axes:
        ax.axis('equal')
        ax.set_xlabel(features[j])
        j = j + 1
    plt.show()
```

```
In [11]: features_qual_pert_vis = ["nutrition_grade_fr", "nova_group"]
pie_chart_qual(features_qual_pert_vis)
```

Fig 4. Camembert des variables "nutrition_grade" et "nova_group"

- Le nutriscore dominant est D.
- 70% des produits sont de Nova 4.

Corrélation de variables quantitatives

- Corrélation linéaire

fonction	data_pair_plot(data=df) corrélation des variables quantitatives
paramètre(s)	data : DataFrame
retour(s)	aucun

In [10]:
```
#corrélation linéaire variables quantitatives, ignorer nova_group
#pairplot
sns.set(style="ticks", color_codes=True)
def data_pair_plot(data=df):
    vars_quantitative = []
    for f in list(data.columns):
        if data[f].dtype == "float" and f != "nova_group":
            vars_quantitative.append(f)
    data["color"] = np.zeros(len(data))
    g = sns.pairplot(data, vars=vars_quantitative, hue="color")
    data.drop(['color'], axis=1, inplace=True)
    g.map_lower(corrfunc)
```

In [32]:
```
data_pair_plot()
```

Corrélation de variables quantitatives

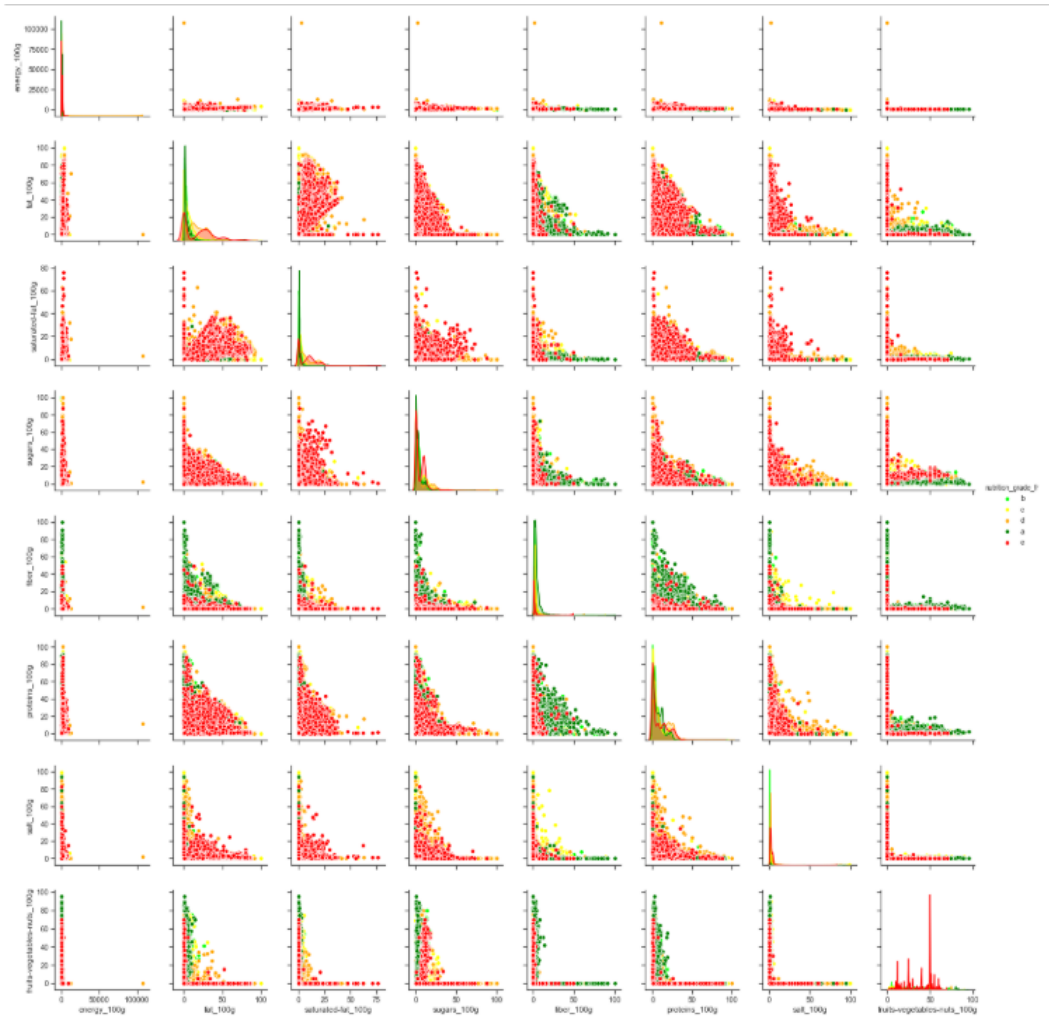


Fig 5. Nuage des points des variables nutritionnelles du nutriscore

Corrélation entre variable qualitative et quantitative

- Corrélation linéaire(ANOVA)

In [18]:
```
#variables quantitatives, variable qualitative=nutri-grade
def anova(Y, quants, data=df):
    fig, axes = plt.subplots(nrows=1, ncols=len(quants), figsize=(15, 4))
    sous_echan = data[data[Y].notnull()]
    modalites = list(sous_echan[Y].unique())
    modalites.sort(reverse=True)
    cs = ["red", "orange", "yellow", "lime", "green"]
    i = 0
    for X in quants:
        if X == "nova_group":
            sous_echan = sous_echan[sous_echan[X].notnull()]
        else:
            sous_echan = sous_echan[sous_echan[X] != 0]
        groupes = []
        for m in modalites:
            groupes.append(sous_echan[sous_echan[Y] == m][X])
        b = axes[i].boxplot(groupes, labels=modalites, vert=False, patch_artist=True)
        for patch, color in zip(b['boxes'], cs):
            patch.set_facecolor(color)
        i = i + 1
    j = 0
    for ax in axes:
        ax.set_xlabel(quants[j])
        ax.set_ylabel(Y)
        j = j + 1
    plt.show()
```

In [15]:
```
anova("nutrition_grade_fr", ["sugars_100g", "proteins_100g", "fiber_100g"])
```

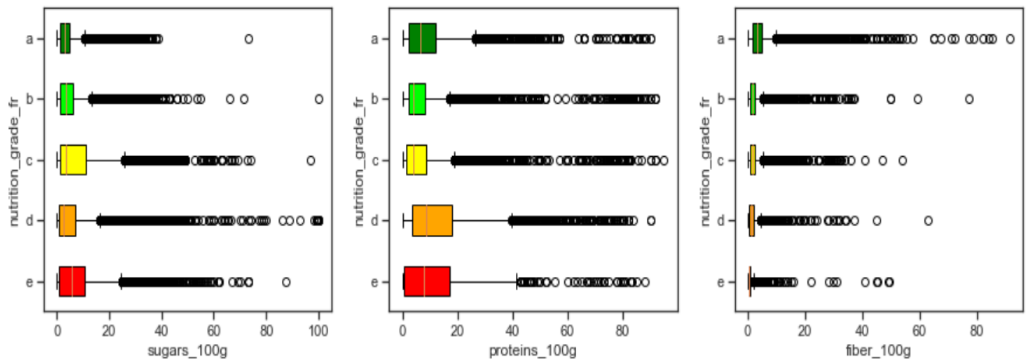


Fig 6. Corrélation entre nutriscore et ses éléments nutritionnels

Matrice de corrélation(heatmap)

- Corrélation entre les variables se terminant par "_100g"

fonction	matrice_correlation(features, data=df) heatmap des variables quantitatives
paramètre(s)	features : listes de variables data : DataFrame
retour(s)	aucun

In [20]:
```
def matrice_correlation(features, data=df):
    X = []
    for f in features:
        X.append(data[f].values)

    Mx = np.corrcoef(X)

    mask = np.zeros_like(Mx)
    n = mask.shape[0]
    for i in range(0, n):
        for j in range(0, n):
            if j > i:
                mask[i, j] = True

    with sns.axes_style("white"):
        ax = sns.heatmap(Mx, vmin=0, vmax=1, linewidths=.5,mask=mask, square=True, xticklabels=features, yticklabels=features)
```

In [21]:
```
feats_sugars = ['sugars_100g', '-sucrose_100g', '-glucose_100g', '-fructose_100g',
                '-lactose_100g', '-maltose_100g', '-maltodextrins_100g']

feats_saturated_fat = ['saturated-fat_100g', '-butyric-acid_100g', '-caproic-acid_100g', '-caprylic-acid_100g',
                       '-capric-acid_100g', '-lauric-acid_100g', '-myristic-acid_100g', '-palmitic-acid_100g',
                       '-stearic-acid_100g', '-arachidic-acid_100g', '-behenic-acid_100g', '-lignoceric-acid_100g',
                       '-cerotic-acid_100g', '-montanic-acid_100g', '-melissic-acid_100g']

feats_omega_3_fat = ['omega-3-fat_100g', '-alpha-linolenic-acid_100g', '-eicosapentaenoic-acid_100g',
                     '-docosahexaenoic-acid_100g']

feats_omega_6_fat = ['omega-6-fat_100g', '-linoleic-acid_100g', '-arachidonic-acid_100g', '-gamma-linolenic-acid_100g',
                     '-dihomo-gamma-linolenic-acid_100g']

feats_omega_9_fat = ['omega-9-fat_100g', '-oleic-acid_100g', '-elaidic-acid_100g', '-gondoic-acid_100g', '-mead-acid_100g',
                     '-erucic-acid_100g', '-nervonic-acid_100g']
```

Hypothèse 1

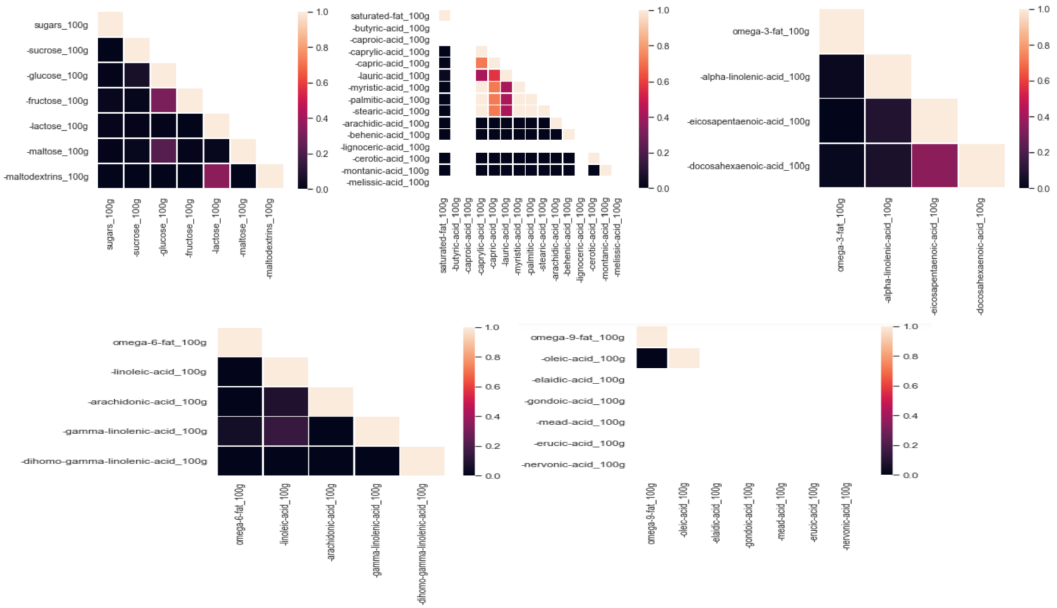


Fig 7. Analyse de corrélation entre un élement principal et ses sous élements

- Pas de corrélation entre un élément principal et ses sous éléments : donc infirmation de ***l'hypothèse 1***

Hypothèse 2

- Corrélation entre le sel et le sodium donc confirmation de ***l'hypothèse 2***

```
In [28]: matrice_correlation(["salt_100g", "sodium_100g"], dc)
```

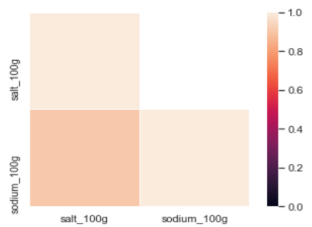


Fig 8. Analyse de corrélation entre les variables "salt" et "sodium"

Sommaire

Interprétation des résultats

- La médiane des valeurs nutritionnelles des éléments du nutriscore est inférieure à 10g, et les valeurs varient pour la plupart entre 0 et 40g.
- La plupart des produits à limiter(nutriscore e) à la consommation sont influencés principalement par le sucre et les matières grasses.
- On note tout de même la présence de l'élément nutritionnel protéine dans la plupart de ces produits qui doit être en faible quantité vu que ça n'influence pas beaucoup le nutriscore.
- Les produits à favoriser (nutriscore a et b) le doivent à la présence des fibres et dans une moindre mesure aux légumes et fruits.
- La faible relation entre le nutriscore et le nova groupe est due au faible nombre d'observation commun.

Application sur le jeu de données

- Exemple de WordCloud sur la variable enseigne(brands).

Fig 9. WordCloud de la variable "brands"

- Ajout au "stopwords" de nltk pour le français les mots de taille 1 du corpus

```
In [96]: res = list(result.index)
         len_one = []
         for r in res:
             if len(r) == 1:
                 len_one.append(r)
         len_one
         my_stop_words = list(set(nltk.corpus.stopwords.words('french') + len_one))
```

Application sur le jeu de données

- Vectorisation des séquences (nom produit, enseigne, ingrédients).

```
In [99]: vectorizer = CountVectorizer(stop_words = my_stop_words)
         X = vectorizer.fit_transform(corpus)
```

- Algorithme de recommandation :

fonction	recommandation(product, nb = 10) recommandation nb premiers produits
paramètre(s)	product : nom du produit nb : nombre de produits de recommandation
retour(s)	aucun

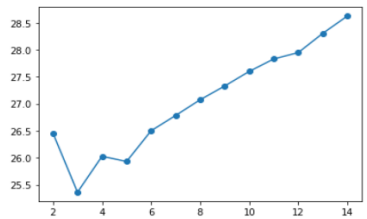
Modèle K-NN

- Hyper-paramètre du modèle

In [8]:
```
#hyperparamètre de Knn
def hyperparametre(dat=df):
    data = dat[dat['nutrition_grade_fr'].notna()]
    target = data['nutrition_grade_fr'].values
    data = data[['energy_100g','saturated-fat_100g','sugars_100g','fiber_100g',
            'proteins_100g','salt_100g','fruits-vegetables-nuts_100g']].values
    xtrain, xtest, ytrain, ytest = train_test_split(data, target, train_size=0.8)
    errors = []
    hyperparam = -1
    min_error = 100
    for k in range(2,15):
        knn = neighbors.KNeighborsClassifier(k)
        error = 100*(1 - knn.fit(xtrain, ytrain).score(xtest, ytest))
        errors.append(error)
        if min_error > error:
            min_error = error
            hyperparam = k
    print("l'hyper-paramètre est de : {}".format(hyperparam))
    plt.plot(range(2,15), errors, 'o-')
    plt.show()
    return hyperparam
```

In [9]:
```
hp = hyperparametre()
```

```
l'hyper-paramètre est de : 3
```



- Classification nutrigrade d'un produit

```
#modèle Knn de remplissage de Nutriscore
def nutriGrade(index, hyperparam, dat=df):
    data = dat[dat['nutrition_grade_fr'].notna()]
    target = data['nutrition_grade_fr'].values
    data = data[['energy_100g','saturated-fat_100g','sugars_100g','fiber_100g',
           'proteins_100g','salt_100g','fruits-vegetables-nuts_100g']].values
    #classification
    knn = neighbors.KNeighborsClassifier(n_neighbors=hyperparam)
    knn.fit(data, target)
    d = dat[dat['nutrition_grade_fr'].isna()]
    elt_to_predict = list(d[d.index == index][['energy_100g','saturated-fat_100g','sugars_100g','fiber_100g',
                                               'proteins_100g','salt_100g','fruits-vegetables-nuts_100g']].values)
    predicted = knn.predict(elt_to_predict)[0]
    return predicted
```

Exemples de recommandations


```
In [41]: recommandation("beignets chocolat", 5)

         Recommandation 1 : Beignet chocolat
         Nutri-grade:  d
         Nova groupe : inconnu
         Proteine : 7.5g
         Fibre : 0.0g
         Fruit et légume : 0.0g
         Energie : 1623.0kJ
         Matière grasse : 17.7g
         Sel : 0.73g
         Sucre : 17.6g
         Code Open Foods Fact BD: 3456580910456

         Recommandation 2 : Beignet au chocolat
         Nutri-grade:  d (predicted)
         Nova groupe : inconnu
         Proteine : 7.5g
         Fibre : 0.0g
         Fruit et légume : 0.0g
         Energie : 1628.0kJ
         Matière grasse : 18.0g
         Sel : 0.7g
         Sucre : 20.0g
         Code Open Foods Fact BD: 0202303000002

         Recommandation 3 : Beignets Chocolat
         Nutri-grade:  d
         Nova groupe : inconnu
         Proteine : 7.5g
         Fibre : 0.0g
         Fruit et légume : 0.0g
         Energie : 1623.0kJ
         Matière grasse : 18.0g
         Sel : 0.7g
         Sucre : 18.0g
         Code Open Foods Fact BD: 0202659019024

In [67]: recommandation("lait de noix de coco", 5)

         Recommandation 1 : Lait de Coco
         Nutri-grade:  d
         Nova groupe :  4.0
         Proteine : 2.0g
         Fibre : 0.2g
         Fruit et légume : 0.0g
         Energie : 707.0kJ
         Matière grasse : 16.9g
         Sel : 0.04g
         Sucre : 0.6g
         Code Open Foods Fact BD: 4002359648618

         Recommandation 2 : Lait De Noix De Coco En Poudre
         Nutri-grade:  d (predicted)
         Nova groupe : inconnu
         Proteine : 1.0g
         Fibre : 0.0g
         Fruit et légume : 0.0g
         Energie : 795.0kJ
         Matière grasse : 14.0g
         Sel : 0.15g
         Sucre : 5.0g
         Code Open Foods Fact BD: 8852114532265

         Recommandation 3 : Lait de riz et noix de coco
         Nutri-grade:  b (predicted)
         Nova groupe : inconnu
         Proteine : 0.3g
         Fibre : 0.0g
         Fruit et légume : 0.0g
         Energie : 268.0kJ
         Matière grasse : 1.8g
         Sel : 0.09g
         Sucre : 5.9g
         Code Open Foods Fact BD: 5603722501128
```

Perspectives

- Appliquer un algorithme de classification plus performant.
- Analyse des valeurs aberrantes des boites à moustaches avec d'autres tests avancés à l'exemple des algorithmes de Isolation Forest
- Comparé l'algorithme de recommandation basé sur la matrice de comptage et celui basé sur la matrice tf-idf.