UNIVERSITÉ PARIS 1
PANTHÉON SORBONNE

Master 2 - Traitement de l'Information et Data-science en Entreprise (TIDE)

Apprentissage Statistique

Prédiction de la catégorie de prix d'un téléphone mobile

Amandine ABENA
Hugo COÏC
Oceane TALEB

Année 2022 - 2023

La collaboration sur ce projet a été très enrichissante et positive. Chacun d'entre nous a apporté ses propres compétences, et ensemble, nous avons travaillé en harmonie pour mener à bout notre projet. Les échanges entre nous ont été fructueux, permettant à chacun d'apprendre de l'autre et de se développer dans le respect et la coopération. Nous avons chacun apporté notre contribution de manière équitable. Ainsi, plus précisément, Amandine s'est occupée de la visualisation et du modèle LASSO, Hugo s'est occupé des modèles SVM et KNN et Oceane s'est occupé du modèle XGBoost et de la rédaction du rapport.

Table des matières

1 Introduction

Bob est un fabriquant de téléphones mobiles, qui a récolté des données sur ses produits, et qui souhaite en fonction des caractéristiques communes des téléphones obtenir une fourchette de prix, donc avoir des gammes de téléphones. C'est une problématique de classification, car notre objectif ici est de prédire la classe de prix la plus probable pour un téléphone.

Il était donc question, dans ce projet, de mettre en oeuvre certaines stratégies que nous avons vu en cours d'apprentissage statistique afin d'aider Bob à estimer au mieux les catégories de prix des téléphones qu'il produit. Nous avons donc testé de multiples algorithmes dans le but de déterminer celui qui prédit le mieux les catégories de prix.

Afin d'évaluer la performance de chacun, nous avons choisi une métrique commune qui est l'accuracy_score. Cette dernière permet de mesurer la proportion de bonnes prédictions sur les données de la base de validation. Nous avons commencé ce travail par une rapide analyse des données, puis nous avons appliqué les différents modèles et enfin nous avons sélectionné celui qui nous semblait le plus pertinent afin de prédire les catégories de prix.

2 Présentation et analyse descriptive des données

Nous avons importé nos données dans deux dataframe distincts, train et test. Dans cette analyse descriptive nous nous sommes focalisé sur la base train afin de dégager les informations nécessaires pour la modélisation. Ainsi, notre base train contient 21 variables et 2000 observations. Les variables correspondent aux caractéristiques du téléphone, comme par exemple la taille de l'écran en pixels, la puissance de la batterie, la ram etc. mais également les catégories de prix. Nous n'avons pas détecté de variables manquantes.

[6]:
```
round(train.describe().T,2)
```

[6]:

	count	mean	std	min	25%	50%	75% ␣ ↪\
battery_power	2000.0	1238.52	439.42	501.0	851.75	1226.0	1615.25
blue	2000.0	0.50	0.50	0.0	0.00	0.0	1.00
clock_speed	2000.0	1.52	0.82	0.5	0.70	1.5	2.20
dual_sim	2000.0	0.51	0.50	0.0	0.00	1.0	1.00
fc	2000.0	4.31	4.34	0.0	1.00	3.0	7.00
four_g	2000.0	0.52	0.50	0.0	0.00	1.0	1.00

int_memory	2000.0	32.05	18.15	2.0	16.00	32.0	48.00
m_dep	2000.0	0.50	0.29	0.1	0.20	0.5	0.80
mobile_wt	2000.0	140.25	35.40	80.0	109.00	141.0	170.00
n_cores	2000.0	4.52	2.29	1.0	3.00	4.0	7.00
pc	2000.0	9.92	6.06	0.0	5.00	10.0	15.00
px_height	2000.0	645.11	443.78	0.0	282.75	564.0	947.25
px_width	2000.0	1251.52	432.20	500.0	874.75	1247.0	1633.00
ram	2000.0	2124.21	1084.73	256.0	1207.50	2146.5	3064.50
sc_h	2000.0	12.31	4.21	5.0	9.00	12.0	16.00
sc_w	2000.0	5.77	4.36	0.0	2.00	5.0	9.00
talk_time	2000.0	11.01	5.46	2.0	6.00	11.0	16.00
three_g	2000.0	0.76	0.43	0.0	1.00	1.0	1.00
touch_screen	2000.0	0.50	0.50	0.0	0.00	1.0	1.00
wifi	2000.0	0.51	0.50	0.0	0.00	1.0	1.00
price_range	2000.0	1.50	1.12	0.0	0.75	1.5	2.25

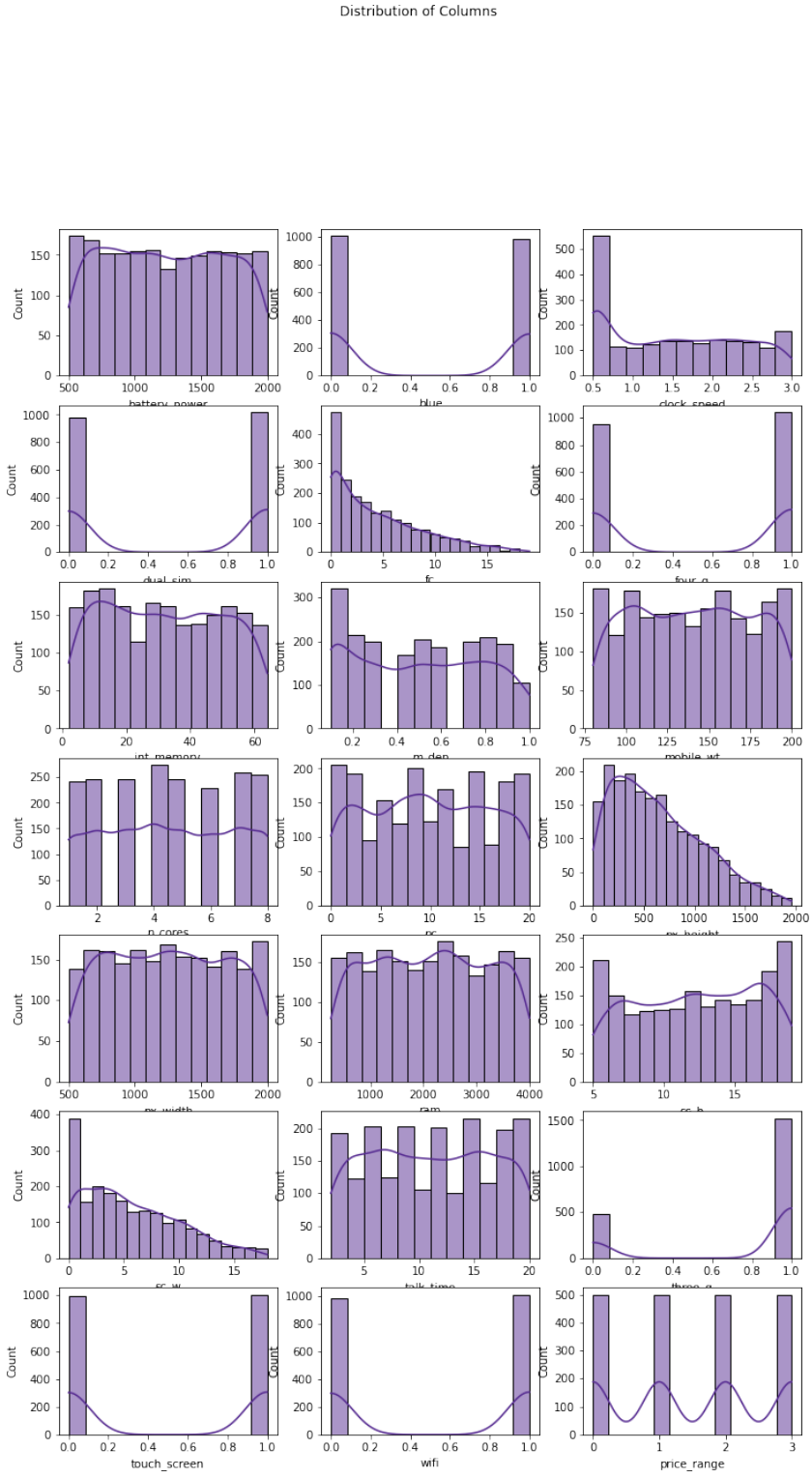
	max
battery_power	1998.0
blue	1.0
clock_speed	3.0
dual_sim	1.0
fc	19.0
four_g	1.0
int_memory	64.0
m_dep	1.0
mobile_wt	200.0
n_cores	8.0
pc	20.0
px_height	1960.0
px_width	1998.0
ram	3998.0
sc_h	19.0
sc_w	18.0
talk_time	20.0
three_g	1.0

```
touch_screen      1.0
wifi              1.0
price_range       3.0
```

Nous avons effectué un résumé statistique des données de train, qui est assez intéressant, car nous observons que les variables ne sont pas à la même échelle. Ce qui signifierait que certaines variables pourraient avoir un impact plus important sur notre variable d'intérêt, qui est la catégorie de prix, que d'autres.

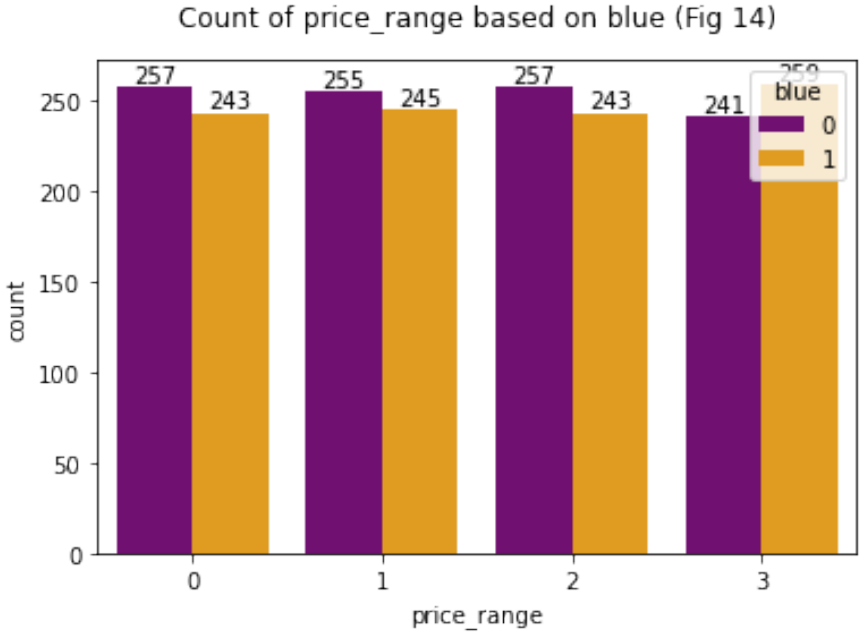
Analyse univariée

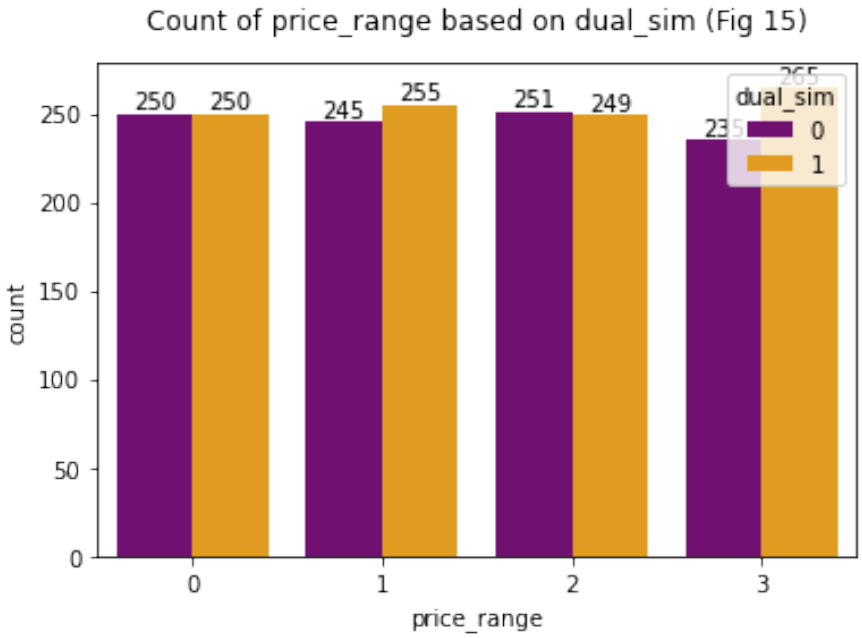
[7]:
```
#Distribution des colonnes
sns.set_palette('Purples_r')
fig, ax = plt.subplots(7,3,figsize=(12,20))
for i, col in enumerate(train):
    sns.histplot(train[col], kde=True, ax=ax[i//3, i%3])
fig.suptitle('Distribution of Columns', y=1.02)
plt.show()
```

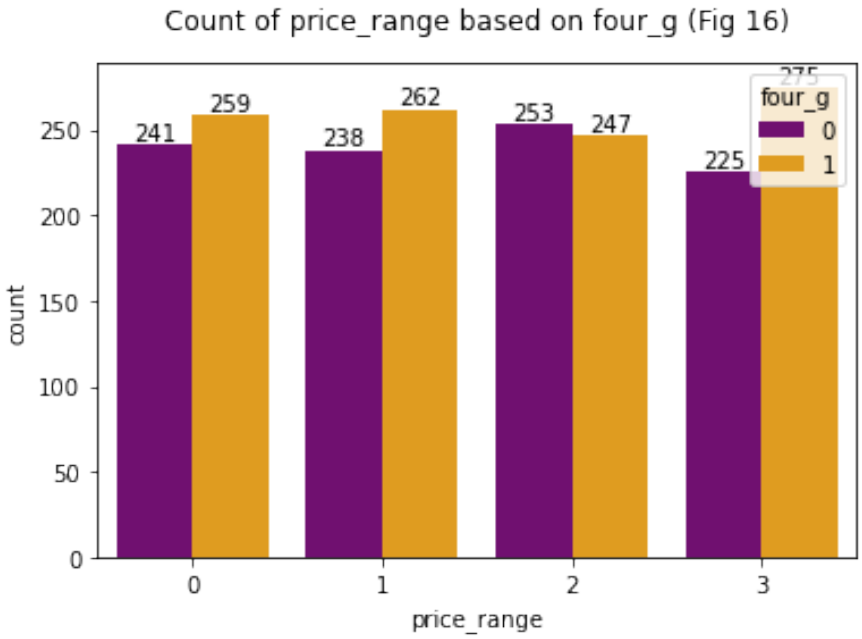


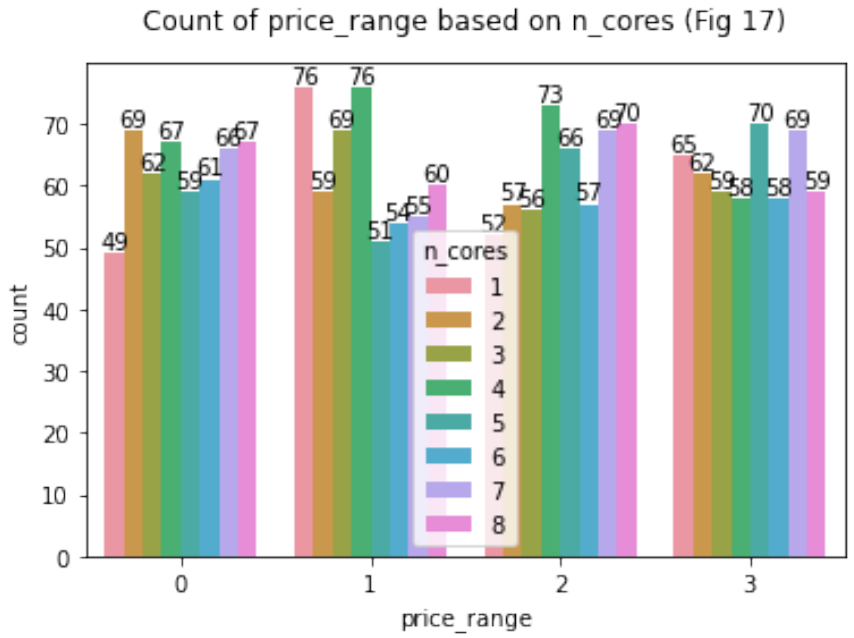
Nous avons représenté graphiquement les distributions de chaque variable avec une courbe de densité dans un premier temps. On observe une homogénéité dans la répartition de notre variable d'intérêt. Les 4 catégories possèdent 500 observations chacune.

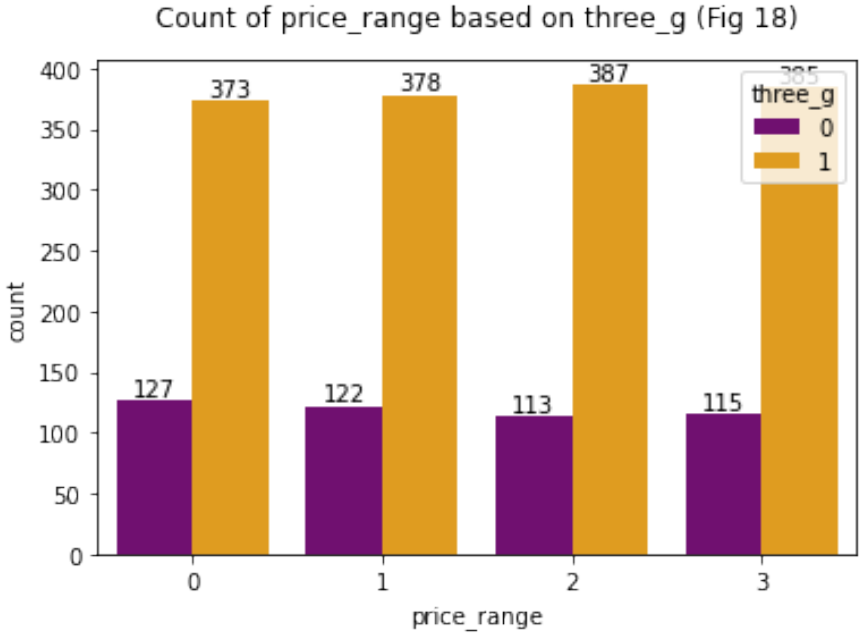
[8]:
```
# Diagrammes à barres de la variable d'intérêt et features catégorielles
cat_features = ['blue', 'dual_sim', 'four_g', 'n_cores', 'three_g',␣
 ↪'touch_screen', 'wifi', 'price_range']
%matplotlib inline
sns.set_palette(['purple','orange','blue','pink'])
for i, col in enumerate(cat_features):
    ax = sns.countplot(data=train, x='price_range', hue=col)
    for container in ax.containers:
        ax.bar_label(container)
    plt.title(f'Count of price_range based on {col} (Fig {i+14})', pad=15)
    plt.show()
```

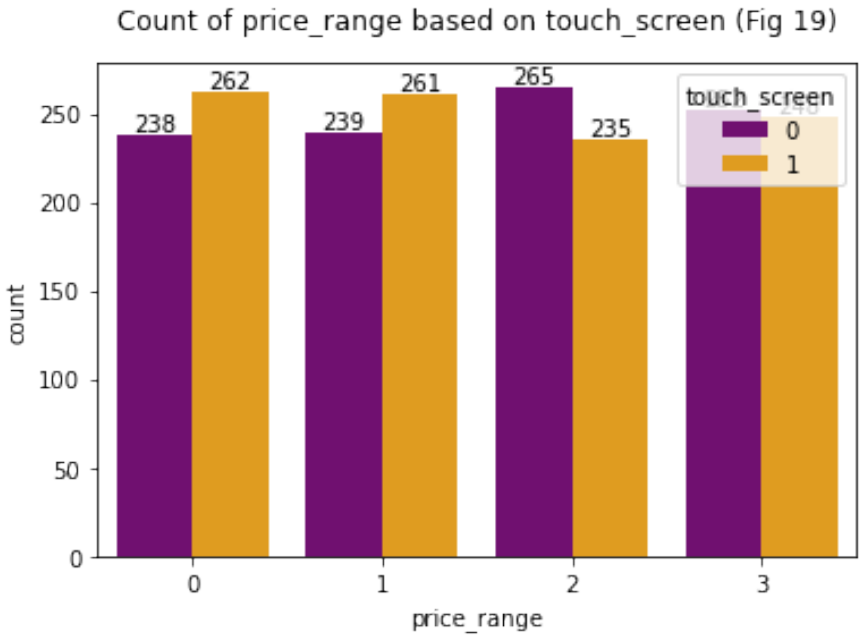



Count of price_range based on dual_sim (Fig 15)
dual_sim
0
1
250
250
245
255
251
249
235
count
price_range
0
1
2
3
0
50
100
150
200
250

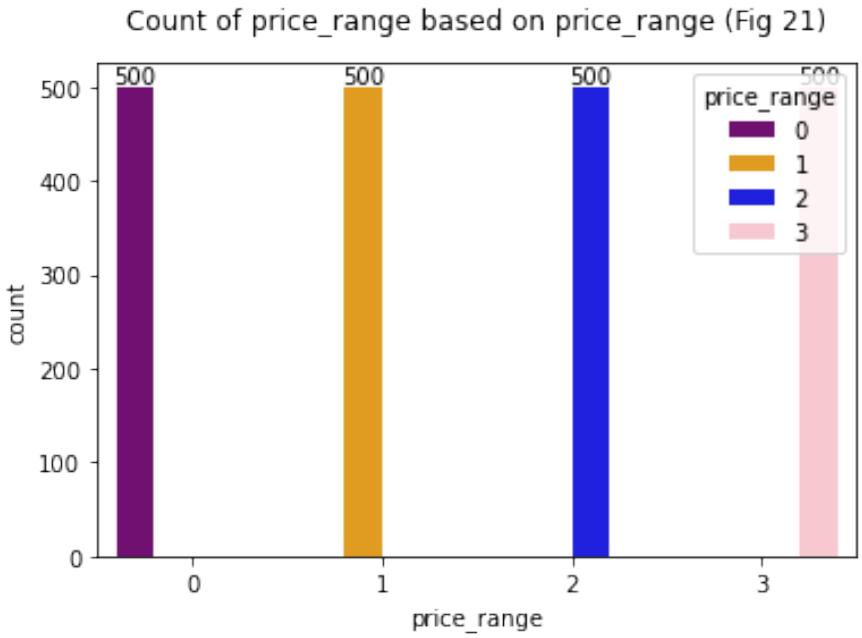

Count of price_range based on four_g (Fig 16)
four_g
0
1
241
259
238
262
253
247
225
count
price_range
0
1
2
3
0
50
100
150
200
250


Count of price_range based on n_cores (Fig 17)
count
price_range
n_cores
1
2
3
4
5
6
7
8
0
1
2
3
49
69
62
67
59
61
66
67
76
59
69
76
51
54
55
60
52
57
56
73
66
57
69
70
65
62
59
58
70
58
69
59


Count of price_range based on three_g (Fig 18)
count
price_range
three_g
0
1
373
378
387
127
122
113
115

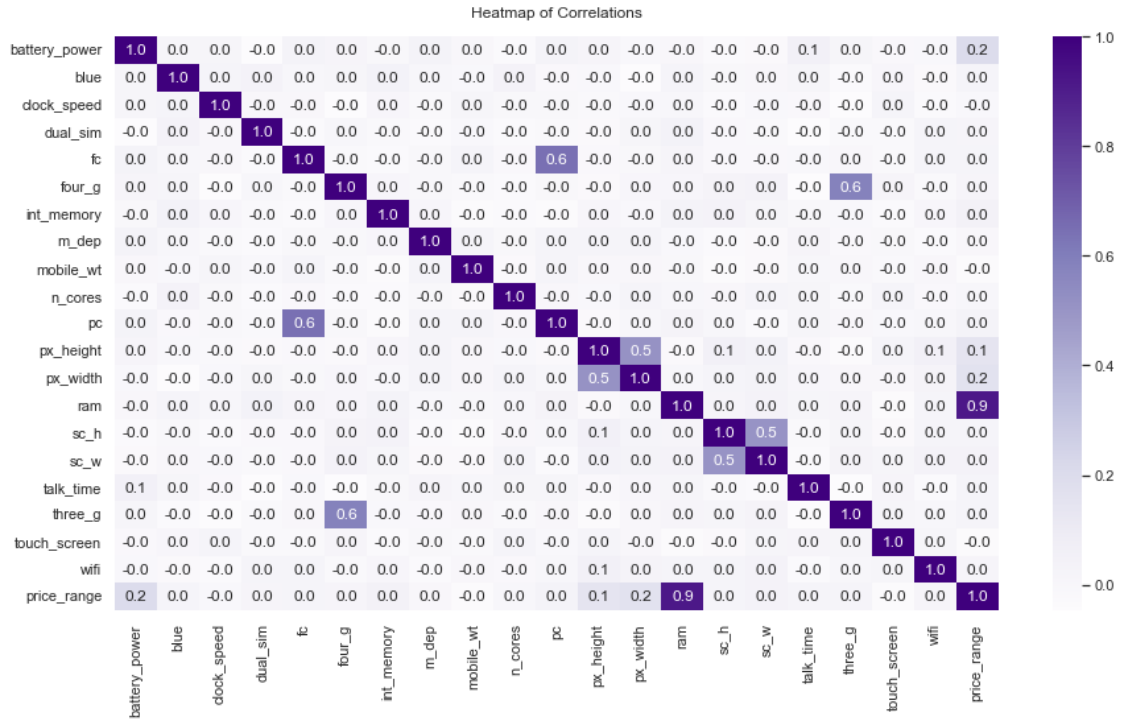

Count of price_range based on touch_screen (Fig 19)
touch_screen
0
1
238
262
239
261
265
235
count
price_range
0
1
2
3
0
50
100
150
200
250


Count of price_range based on wifi (Fig 20)
wifi
0
1
252
248
248
252
248
252
238
count
price_range
0
1
2
3
0
50
100
150
200
250



Puis nous avons représenté les variables catégorielles en fonction de notre variable d'intérêt, en diagramme en barre. On remarque que celles ci sont également homogènes entre les différentes gammes de prix. On a à peu près le même nombre d'observations par modalité dans les 4 gammes de prix.

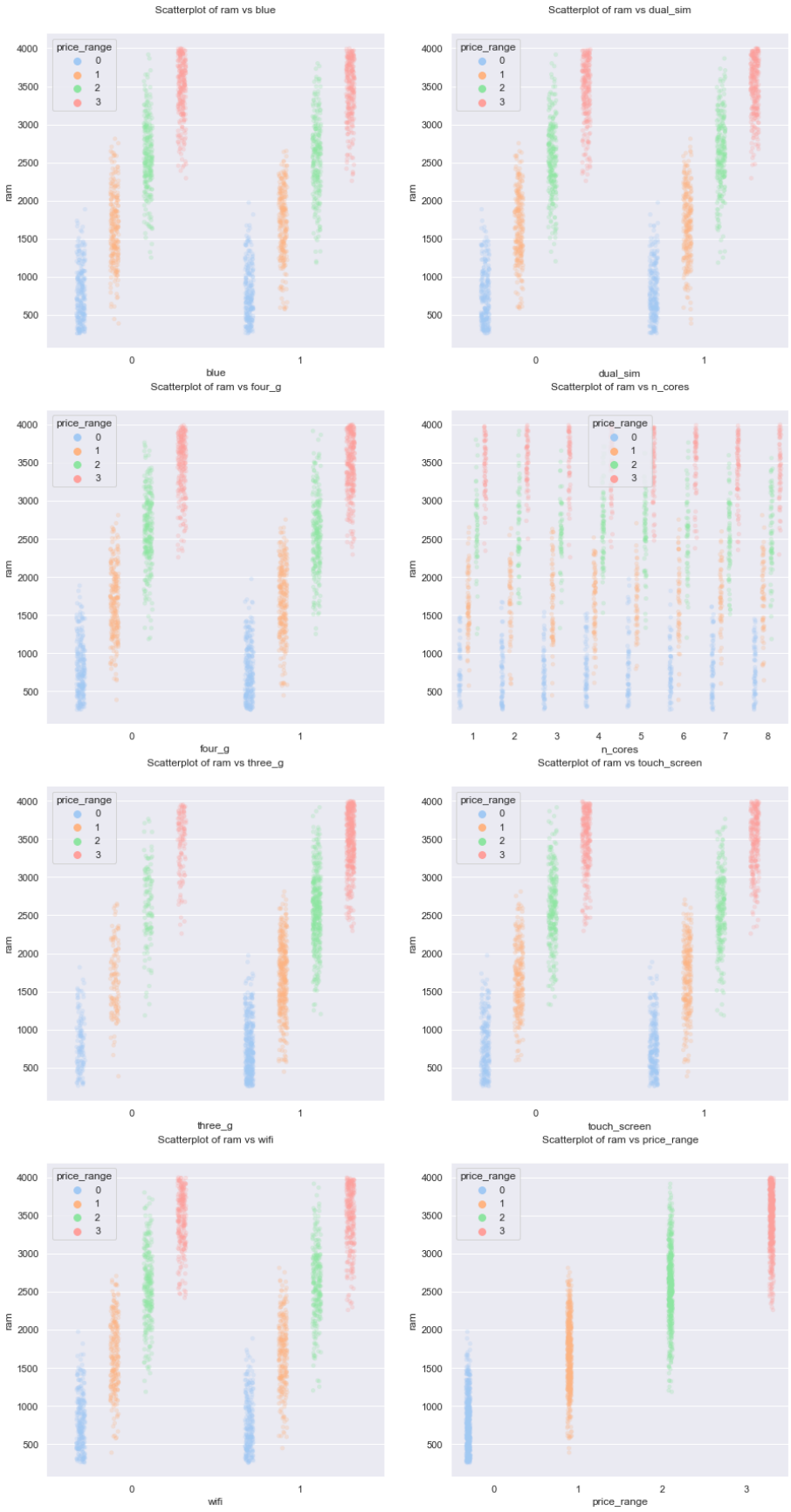
[9]:
```
#Matrice des corrélations
sns.set(rc = {'figure.figsize':(15,8)})
sns.heatmap(train.astype(float).corr(), annot=True, fmt=".1f",␣
 ↪cmap="Purples")
plt.title('Heatmap of Correlations', y=1.02)
plt.show()
```



Enfin, nous avons étudié la corrélation des variables de notre base train. On remarque qu'il y a globalement très peu de corrélation. La variable de prix semble fortement dépendre de la RAM et peu de la taille de l'écran et de la batterie. Les autres variables ne semblent pas avoir d'impact direct sur le prix. Néanmoins il est possible qu'il y en ait un si on combine plusieurs de ces variables. On observe également des corrélations logiques entre certaines variables, par exemple entre la longueur et la largeur de l'écran ou encore la 3G et la 4G.

Analyse multivariée

[10]:
```
f, ax = plt.subplots(4,2,figsize=(15,30))
sns.despine(bottom=True, left=True)
for i, col in enumerate(cat_features):
    sns.stripplot(data=train, x=col, y='ram', hue='price_range',␣
 ↪dodge=True, jitter=True, alpha=.25, zorder=1, ax=ax[i//2,␣
 ↪i%2],palette='pastel')
    ax[i//2, i%2].set_title(f'Scatterplot of ram vs {col}',y=1.05)
plt.show()
```


Scatterplot of ram vs blue
Scatterplot of ram vs dual_sim
Scatterplot of ram vs four_g
Scatterplot of ram vs n_cores
Scatterplot of ram vs three_g
Scatterplot of ram vs touch_screen
Scatterplot of ram vs wifi
Scatterplot of ram vs price_range
price_range
0
1
2
3
ram
blue
dual_sim
four_g
n_cores
three_g
touch_screen
wifi
price_range

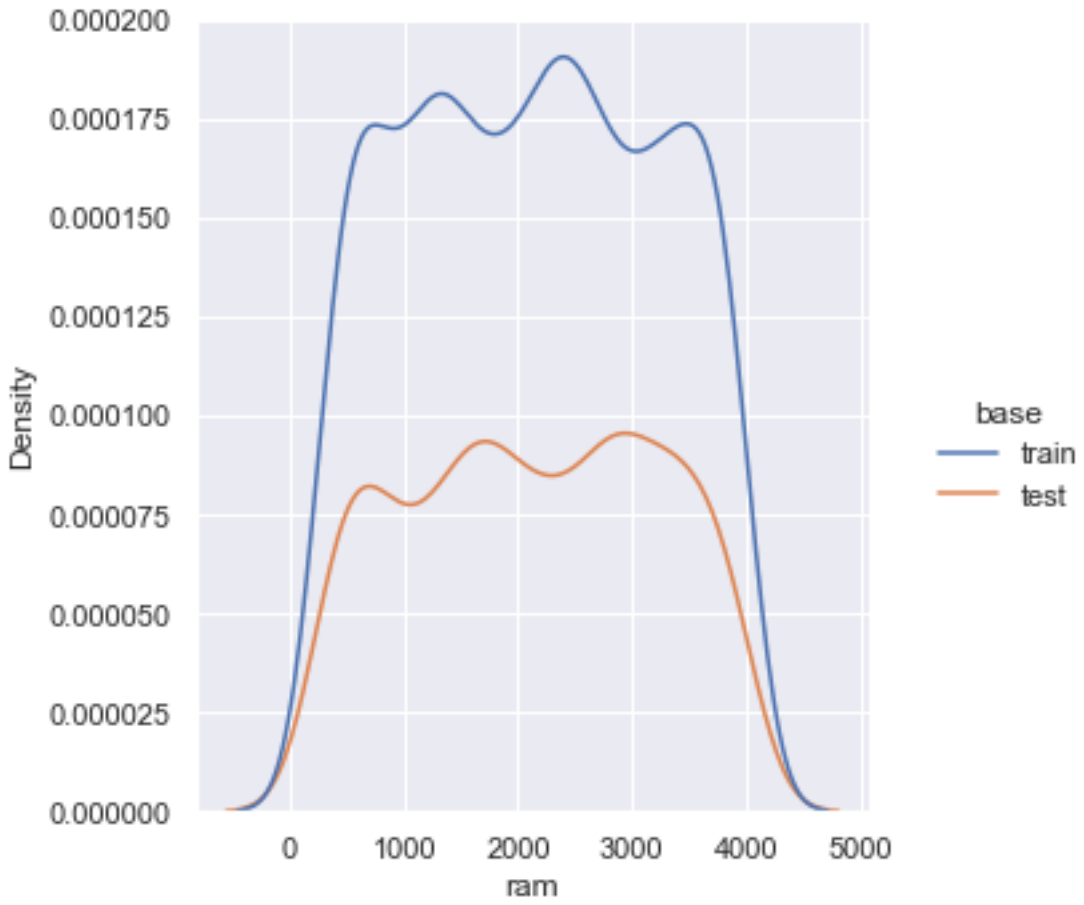
La variable RAM étant la plus corrélée au prix, nous avons analysé la distribution du croisement de cette dernière et des variables catégorielles en fonction de la gamme de prix. On observe des dispersions plus ou moins prononcées au sein de nos classes de prix, ce qui indiquerait une faible corrélation dans nos données, et donc que ces variables n'affectent pas significativement le prix.

Analyse de la base test

Avant de passer à la modélisation, nous avons observé notre base test. Cette dernière contient 21 variables et 1000 observations. Cependant, on remarque que la variable que nous souhaitons prédire n'y figure pas. Notre solution a été de diviser notre base train en données d'entraînement et données de validation, comme nous le verrons plus tard. Nous avons comparé les distributions des variables dans train et dans test afin de nous assurer de la pertinence de notre solution. Pour cela nous nous sommes concentrés uniquement sur les 4 variables corrélées au prix.

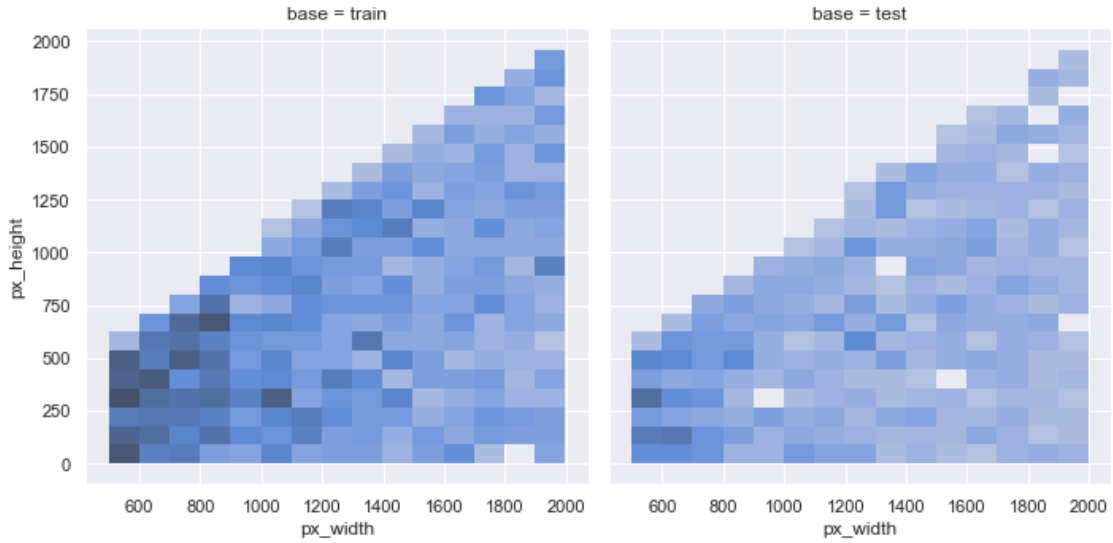
[13]:
```
sns.displot(data=var,x="ram",hue="base",kind="kde")
```

[13]:
```
<seaborn.axisgrid.FacetGrid at 0x7ff77df5ffa0>
```



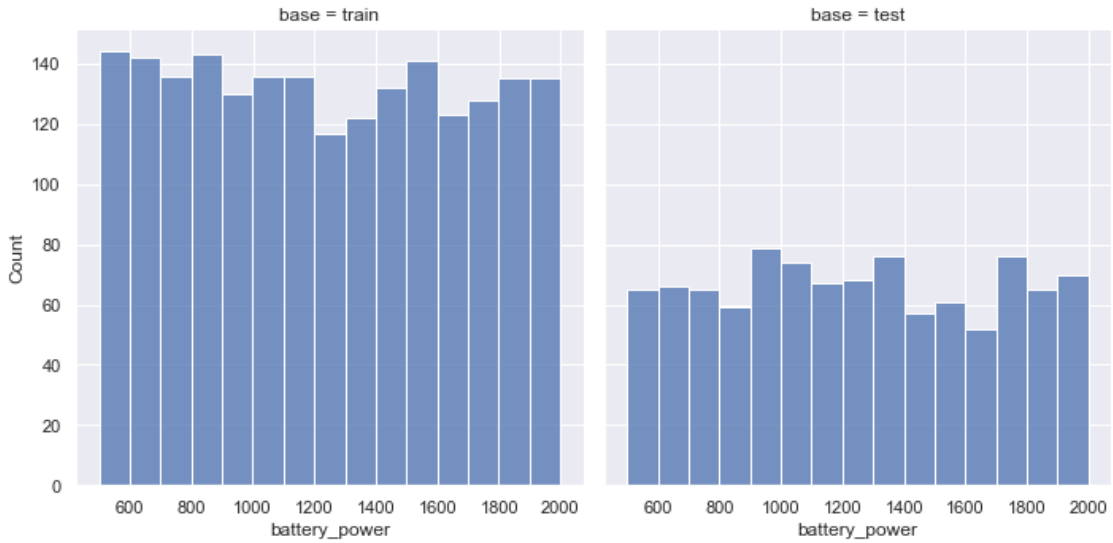
[14]:
```
sns.displot(data=var,x="px_width",y="px_height",col="base")
```

[14]:
```
<seaborn.axisgrid.FacetGrid at 0x7ff77e06ceb0>
```



[15]:
```
sns.displot(data=var,x="battery_power",col='base')
```

[15]:
```
<seaborn.axisgrid.FacetGrid at 0x7ff7985f0730>
```



Les résultats semblent montrer des distributions très similaires entre les deux bases. Nous procédons alors à la division de notre base train.

Standardisation des données

Pour ce projet, nous travaillons sur une problématique de classification, nous allons utiliser des algorithmes qui calculent les distances mais aussi des modèles qui nécessitent des données centrées et réduites. Nous avons alors effectué le centrage et la réduction de nos bases pour la suite de notre travail.

3 Régression LASSO

Bien que cela puisse paraître surprenant dans un cadre de classification, nous avons utilisé en premier lieu une régression LASSO qui normalement est utilisée pour prédire une variable continue et non pas catégorielle. Nous avons fait ce choix car les fourchettes de prix semblent être rangées dans un ordre logique.

Cependant il faut rester vigilants avec cette approche principalement pour deux raisons :

— Rien ne nous indique que les fourchettes de prix sont réellement rangées suivant un ordre logique, même si, à priori, ça semble être le cas.

— Mais surtout, il sera compliqué de prédire la fourchette de prix d'un téléphone qui possède des caractéristiques exceptionnelles. Car pour faire nos prédictions, il faut arrondir à l'entier près. Donc, si un téléphone a des caractéristiques exceptionnelles, il serait possible d'obtenir un résultat de prédiction à l'entier supérieur, et donc obtenir une catégorie de prix supérieure à la catégorie maximale.

[19]:
```
from sklearn.linear_model import Lasso
from sklearn.model_selection import train_test_split, cross_val_score

# Définir une liste de valeurs d'alpha à tester
alphas = np.logspace(-4, -0.5, 30)

# Entraîner un modèle Lasso pour chaque alpha et évaluer la performance␣
 ↪en utilisant la validation croisée
scores = []
for alpha in alphas:
    reg = Lasso(alpha=alpha)
    score = np.mean(cross_val_score(reg, X_train, y_train, cv=5,␣
 ↪scoring='neg_mean_squared_error'))
```

```
    scores.append(score)

# Trouver le meilleur alpha en utilisant la validation croisée
best_alpha = alphas[np.argmax(scores)]

# Entraîner un modèle Lasso sur les données d'entraînement avec le␣
 ↪meilleur alpha
reg = Lasso(alpha=best_alpha)
reg.fit(X_train, y_train)

# Sélectionner les variables en utilisant le modèle Lasso
selected_features = X_train.columns[reg.coef_ != 0]

# Afficher les variables sélectionnées
print("Le meilleur alpha est :",best_alpha )
print("Variables sélectionnées :", selected_features)
```

```
Le meilleur alpha est : 0.006461670787466976
Variables sélectionnées : Index(['battery_power', 'blue', 'dual_sim',
'int_memory', 'mobile_wt',
       'n_cores', 'px_height', 'px_width', 'ram', 'three_g', 'touch_screen',
       'wifi'],
      dtype='object')
```

[20]:
```
from sklearn.metrics import mean_squared_error, r2_score

# Prédire les cibles sur l'ensemble de test
y_pred = reg.predict(X_valid)

print( " R SQUARED is ",round(( r2_score(y_valid,y_pred))*100,2),"%")
print(" Mean squared  error is " , mean_squared_error(y_valid,y_pred))
accuracy_score(y_valid,np.around(y_pred))
```

```
 R SQUARED is  92.2 %
 Mean squared  error is  0.10056966702699618
```

Les résultats obtenus montrent que le modèle explique environ 92% de la variance des don-

nées. L'erreur quadratique moyenne qui lui est associé est aussi relativement basse. On pourrait donc penser que les résultats sont bons, néanmoins ils sont à prendre avec précaution car la régression LASSO n'est pas un modèle de classification.

4 Classification : Régression Logistique

Le premier algorithme de classification que nous avons essayé est celui de la régression logistique. Il modélise la probabilité qu'une observation appartienne à une classe donnée, en fonction de variables explicatives. Cet algorithme est intéressant dans le cadre de la classification, car il peut prédire une variable catégorielle à partir de nombreuses variables, notamment continues.

[21]:
```
modele_regression_logistique = linear_model.LogisticRegression ()
modele_regression_logistique.fit(X_train, y_train)
train_pred=modele_regression_logistique.predict(X_train)
valid_pred_reg_log=modele_regression_logistique.predict(X_valid)
(accuracy_score(y_train, train_pred),accuracy_score(y_valid,␣
 ↪valid_pred_reg_log))
```

[21]:
```
(0.978125, 0.955)
```

Dans cette première partie, nous avons pris en compte toutes les variables de notre base de données. Nous avons ensuite mesuré la performance de l'algorithme grâce à l'accuracy score, qui nous indique la proportion de prédictions correctes du modèle. On observe un score de 0.978125 sur la base d'entraînement et de 0.955 sur la base de validation. C'est un score assez élevé, la régression logistique parvient donc à prédire les classes de prix avec beaucoup de précision.

[22]:
```
accur=[]
for i in (['lbfgs', 'liblinear', 'newton-cg', 'sag', 'saga']):
    modele_regression_logistique2 = linear_model.LogisticRegression␣
 ↪(solver=i)
    modele_regression_logistique2.fit(X_train, y_train)
    predictions=modele_regression_logistique2.predict(X_valid)
    accur.append(accuracy_score(y_valid, predictions))
```

[23]:
```
accur ### pas de gains de performance selon l'algo d'optimisation
```

[23]:
```
[0.955, 0.8375, 0.955, 0.955, 0.955]
```

Nous avons également comparé les différents "solvers" de la régression logistique, qui correspondent aux différents algorithmes d'optimisation du modèle. Nous avons remarqué que nous n'avons pas de gain de performance, même que le solver "liblinear" fait diminuer le score. Les résultats étant pratiquement tous les mêmes, il n'est pas nécessaire de préciser ce paramètre. Nous avons donc conservé nos résultats utilisant le solver par défaut.

5 Classification : SVM

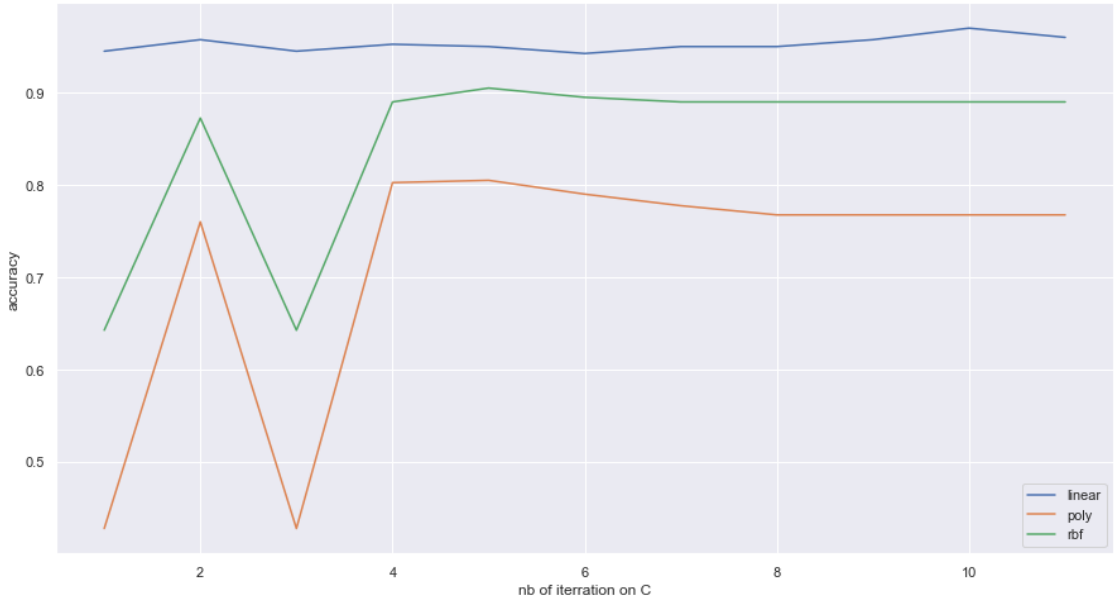
Les classes semblent pouvoir être linéairement séparables en fonction de la RAM, qui est la variable qui semble être la plus discriminante pour le label comme nous l'avons vu à travers la matrice de corrélation. Le modèle SVM devrait donc bien fonctionner, particulièrement avec une séparation linéaire.

Le SVM (Support Vector Machine) est un algorithme de classification qui permet de séparer les données en plusieurs classes. Ce dernier retrouve une ligne appelée hyperplan qui sépare les différents groupes entre eux de manière à ce que les marges entre les classes soient les plus grandes possibles, autrement dit, que les classes formées soient les plus distinctes possibles. Le SVM est régulièrement utilisé dans des travaux de classification, notamment complexes, car il est d'une grande précision et il qu'il resiste à l'overfitting.

Nous avons donc, dans un premier temps représenté l'évolution de l'accuracy score des SVM en fonction du nombre d'itération, pour 3 fonctions de kernel (linear, poly et rbf), dans le but de sélectionner les paramètres (kernel et nombre d'itérations) qui correspondent le mieux à notre objectif de classification et à notre jeu de données.

[27]:
```
plt.plot(l,accur, label='linear')
plt.plot(l,accur_poly, label='poly')
plt.plot(l,accur_rbf, label='rbf')
plt.xlabel('nb of iterration on C')
plt.ylabel('accuracy')
plt.legend()
```

[27]:
```
<matplotlib.legend.Legend at 0x7ff77dfc9280>
```



On observe que la courbe du 'linear' est tout le temps au dessus des deux autres, ce qui veut dire que le choix du kernel 'linear' nous garanti un accuracy score le plus élevé quelque soit le nombre d'itérations du modèle.

[28]:
```
model_svm = SVC(kernel='linear',C=500)
model_svm.fit(X_train,y_train)
```

[28]:
```
SVC(C=500, kernel='linear')
```

[29]:
```
train_pred=model_svm.predict(X_train)
test_pred_svm=model_svm.predict(X_valid)
(accuracy_score(y_train, train_pred),accuracy_score(y_valid,␣
 ↪test_pred_svm))
```

[29]:
```
(0.995, 0.97)
```

Nous avons donc choisi ce kernel pour un nombre d'itération de 500. Nous obtenons un score de 0.995 pour la base d'entraînement et de 0.97 pour la base de validation, ce qui veut dire que le SVM est très performant en termes de prédiction de catégories.

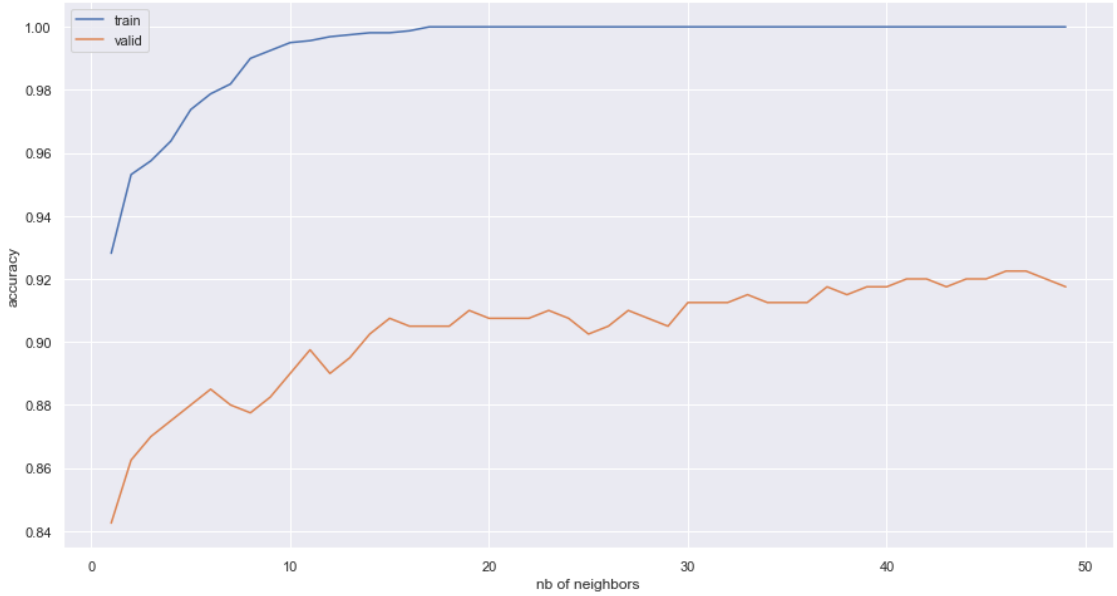
6 Classification : XGBoost

Ensuite, nous avons testé l'algorithme XGBoost. Il s'agit d'un algorithme qui prédit des résultats en utilisant des arbres de décision, c'est un algorithme de gradient boosting. Il

combine plusieurs petits arbres afin de créer un arbre plus performant, qui minimise les erreurs commises sur les arbres avant agrégation. C'est un modèle intéressant car il permet de trouver des relations complexes entre les données, mais aussi car il est capable de prendre en charge les données manquantes et les variables catégorielles.

[51]:
```
plt.plot(range(1,50),accur_train, label='train')
plt.plot(range(1,50),accur_valid, label='valid')
plt.xlabel('nb of estimators')
plt.ylabel('accuracy')
plt.legend()
```

[51]:
```
<matplotlib.legend.Legend at 0x7f85ab0526d0>
```



Nous avons commencé par représenter l'évolution de l'accuracy score en fonction de n_estimators, afin de déterminer le paramètre optimal. Ce paramètre correspond au nombre d'arbres de décision utilisé dans le modèle de gradient boosting. Lorsqu'il est élevé, on améliore la performance de notre algorithme, mais il ne faut pas qu'il soit trop élevé, par risque de surapprentissage. Il est donc important de bien le sélectionner. Ici, on observe que le score maximal pour la base de validation est obtenu pour un n_estimators de 46.

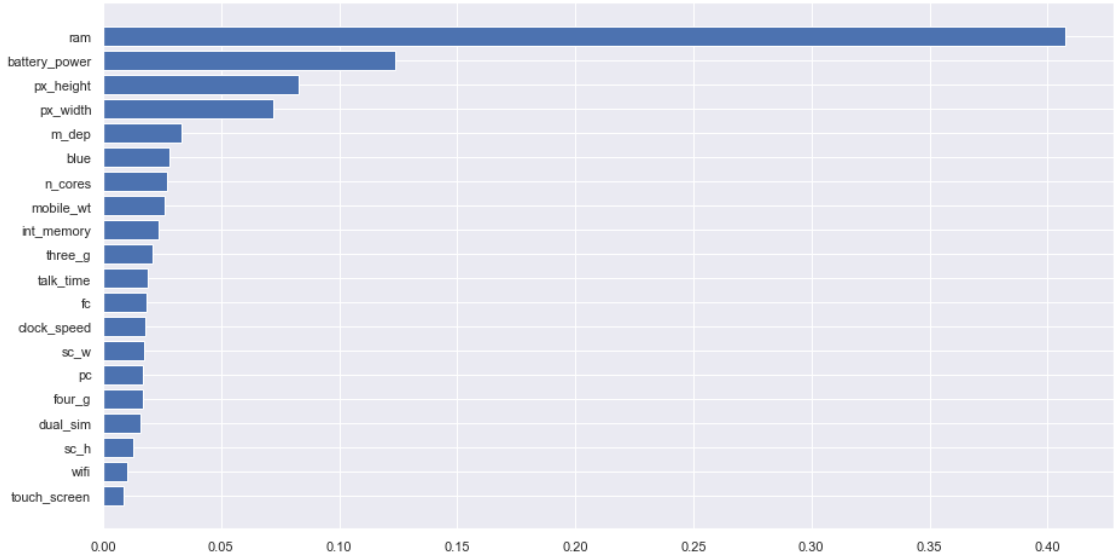
[77]:
```
xgb = XGBClassifier(n_estimators=46)
xgb.fit(X_train, y_train)

pred_train = xgb.predict(X_train)
```

```
pred_valid = xgb.predict(X_valid)

(accuracy_score(y_train, pred_train), accuracy_score(y_valid, pred_valid))
```

[77]: (1.0, 0.9225)

En utilisant ce paramètre, on obtient un score de 1 pour la base d'entraînement et de 0.9225 pour la base de validation. C'est un bon score. Le modèle fait que peu d'erreur sur la prédiction des différentes classes.

[78]:
```
importance=xgb.feature_importances_
indices = np.argsort(importance)

fig, ax = plt.subplots()
ax.barh(range(len(indices)), importance[indices])
ax.set_yticks(range(len(indices)))
_ = ax.set_yticklabels(np.array(X_train.columns)[indices])
```



Par la suite, nous nous sommes intéressés au poids de chaque varibles dans ce modèle. Nous avons alors représenté un diagramme de l'importance des features. On remarque que les 4 variables les plus importantes sont celles qui sont le splus corrélées au prix, à savoir la ram, la battery_power, px_height et pw_width comme nous l'avons vu précédemment. Il serait donc intéressant de voir les résultats de ce modèle avec une sélection de variable.

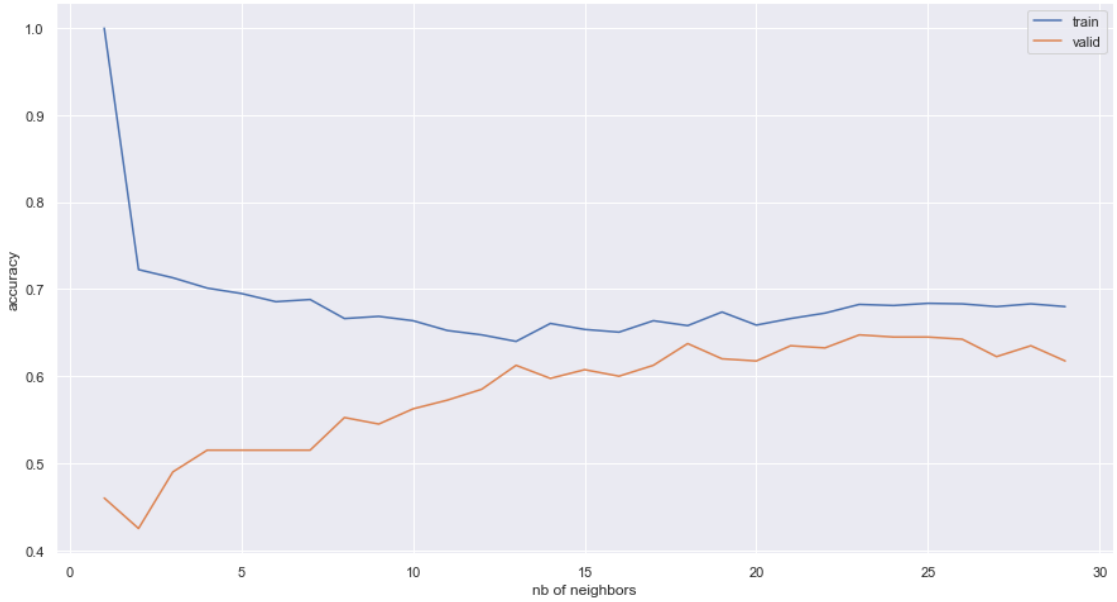
7 Classification : KNN

Pour les mêmes raisons que le SVM le KNN devrait bien fonctionner. C'est ainsi le dernier algorithme de classification que nous allons testé dans ce projet.

L'algorithme des k plus proches voisins (KNN) est un algorithme de classification supervisée basé sur le calcul de distances. Il calcule la distances entre une nouvelle observation spécifique et les autres observations d'entraînement, puis détermine ses k plus proches voisins et détermine la classe à laquelle l'obervation appartient. Le nombre de proches voisins est un paramètre d'entrée de l'algorithme. Il est important de bien choisir ce nombre pour optimiser la performance de l'algorithme.

[30]:
```
accur_train=[]
accur_valid=[]
l=[]
for i in range(1,30):
    model_knn= KNeighborsClassifier(n_neighbors=i)
    model_knn.fit(X_train,y_train)
    train_pred=model_knn.predict(X_train)
    test_pred=model_knn.predict(X_valid)
    score_train=accuracy_score(y_train, train_pred)
    score_valid=accuracy_score(y_valid, test_pred)
    accur_train.append(score_train)
    accur_valid.append(score_valid)
    l.append(i)


plt.plot(l,accur_train, label='train')
plt.plot(l,accur_valid, label='valid')
plt.xlabel('nb of neighbors')
plt.ylabel('accuracy')
plt.legend()
```

[30]:
```
<matplotlib.legend.Legend at 0x7ff77dd6a9d0>
```



Nous avons représenté graphiquement l'évolution de l'accuracy score du modèle en fonction du nombre de voisins que l'algorithme prend en entrée, afin de déterminer le nombre de voisins optimal. On remarque que ce dernier est maximisé entre 20 et 25, et se situe aux alentours de 23 environ.

[31]:
```
train_pred=model_knn.predict(X_train)
test_pred=model_knn.predict(X_valid)
(accuracy_score(y_train, train_pred),accuracy_score(y_valid, test_pred))
```

[31]:
```
(0.68, 0.6175)
```

On observe aussi un résultat de 0.68 pour la base d'entraînement et de 0.6175 pour la base de validation. Il s'agit du score le moins élevé, et donc l'algorithme le moins performant que nous ayant testé dans ce projet.

Cependant il serait intéressant de voir comment se comportent les KNN et SVM lorsqu'on opère une sélection de variables les plus discriminantes. En effet nous avons jusque la utilisé des modèles qui calculent des distances sur toutes les variables, or seules certaines d'entre elles semblent être discriminantes.

8 Modèles avec sélection de variables

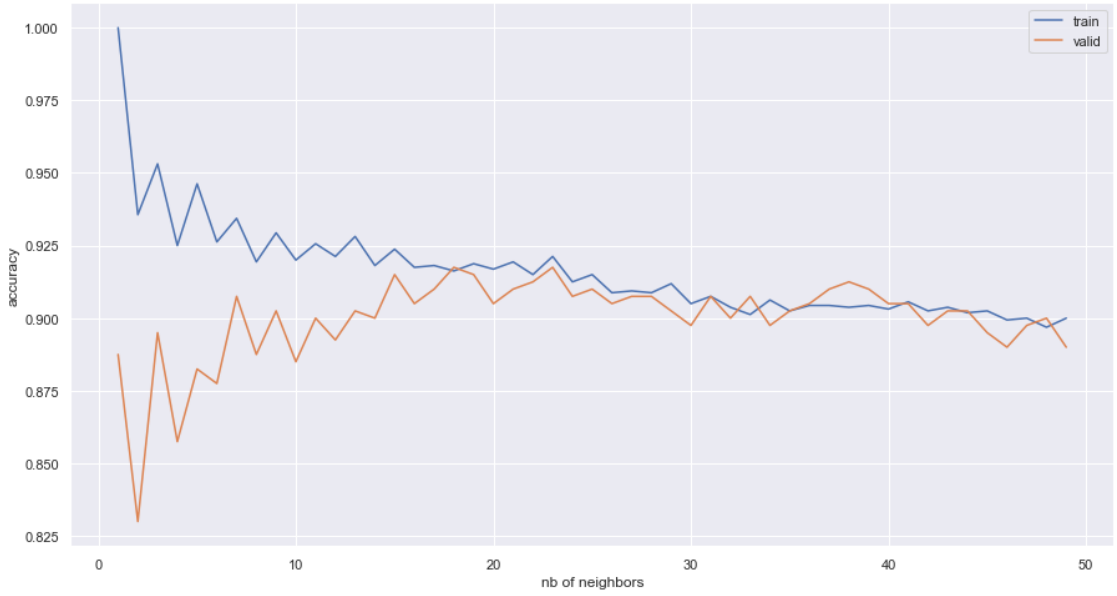
Nous avons, pour la suite de ce travail, jugé utile d'effectuer une sélection de variables afin d'améliorer la performance de nos algorithmes. Pour ce faire, nous avons choisi comme critère

de sélection la corrélation avec la variable d'intérêt. Nous avons observé dans la partie analyse descriptive, à travers la matrice de corrélation, qu'il y avait 4 variables corrélées au prix : ram, battery_power, px_height et px_width. Nous avons donc retenues ces variables et avons relancé nos algorithmes.

8.1 KNN

[33]:
```
accur_train=[]
accur_valid=[]
l=[]
for i in range(1,50):
    model_knn= KNeighborsClassifier(n_neighbors=i)
    model_knn.fit(X_train_r,y_train)
    train_pred=model_knn.predict(X_train_r)
    test_pred=model_knn.predict(X_valid_r)
    score_train=accuracy_score(y_train, train_pred)
    score_valid=accuracy_score(y_valid, test_pred)
    accur_train.append(score_train)
    accur_valid.append(score_valid)
    l.append(i)


plt.plot(l,accur_train, label='train')
plt.plot(l,accur_valid, label='valid')
plt.xlabel('nb of neighbors')
plt.ylabel('accuracy')
plt.legend()
```

[33]:
```
<matplotlib.legend.Legend at 0x7ff77e11de20>
```



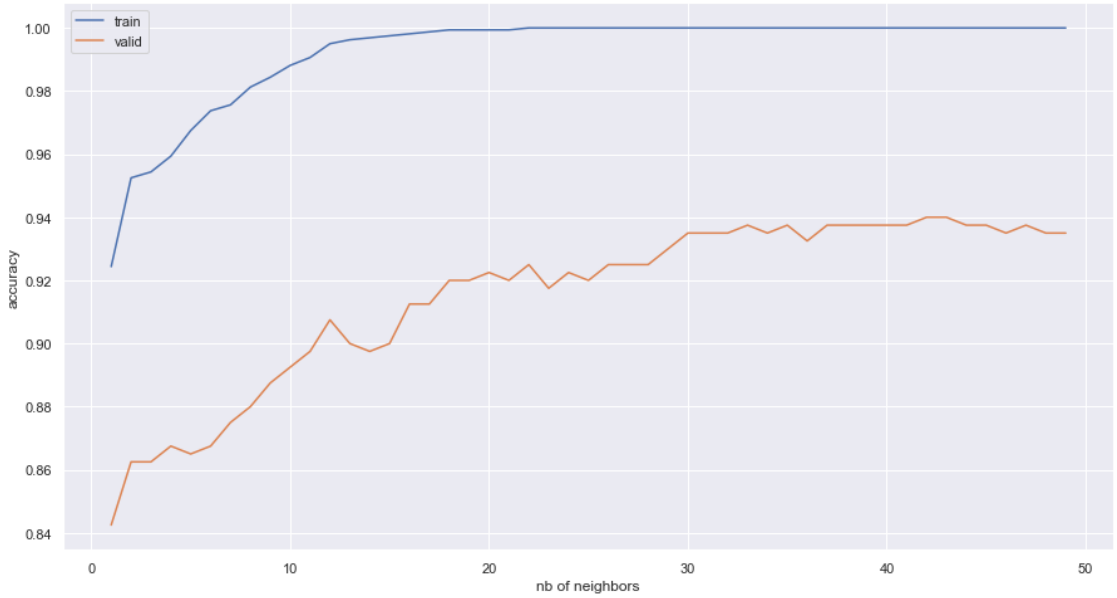
Nous avons à nouveau représenté graphiquement l'évolution de l'accuracy score en fonction du nombre de voisins, mais cette fois avec un modèle de KNN qui prend uniquement 4 variables.

[34]:
```
model_knn= KNeighborsClassifier(n_neighbors=23)
model_knn.fit(X_train_r,y_train)
train_pred=model_knn.predict(X_train_r)
test_pred_knn=model_knn.predict(X_valid_r)
(accuracy_score(y_train, train_pred),accuracy_score(y_valid,␣
 ↪test_pred_knn))
```

On observe que l'échelle du score a changé et que celui ci a augmenté. Le k optimal est quant à lui toujours le même, à voir 23.

Nous avons alors calculé l'accuracy score des KNN avec k = 23 et 4 variables. Nous avons obtenu un résultat de 0.92125 pour la base d'entraînement et de 0.9175 pour la base de validation. Ce qui est un très bon score. La sélection de variable a nettement amélioré la performance du KNN. Nous allons voir si c'est également le cas pour le XGBoost et le SVM.

8.2 XGBoost



Nous opérons encore une fois une sélection des paramètres optimaux, après sélection de variables. Nous représentons l'évolution de l'accuracy score en fonction du nombre d'estimateurs, et observons la valeur la plus haute pour le score, afin de déterminer le nombre d'estimateurs optimal. Ici, on peut voir que le score est maximisé pour 43 estimateurs.

[75]:
```
xgb = XGBClassifier(n_estimators=43)
xgb.fit(X_train_r, y_train)

pred_train = xgb.predict(X_train_r)
pred_valid = xgb.predict(X_valid_r)

(accuracy_score(y_train, pred_train), accuracy_score(y_valid, pred_valid))
```

[75]: (1.0, 0.94)

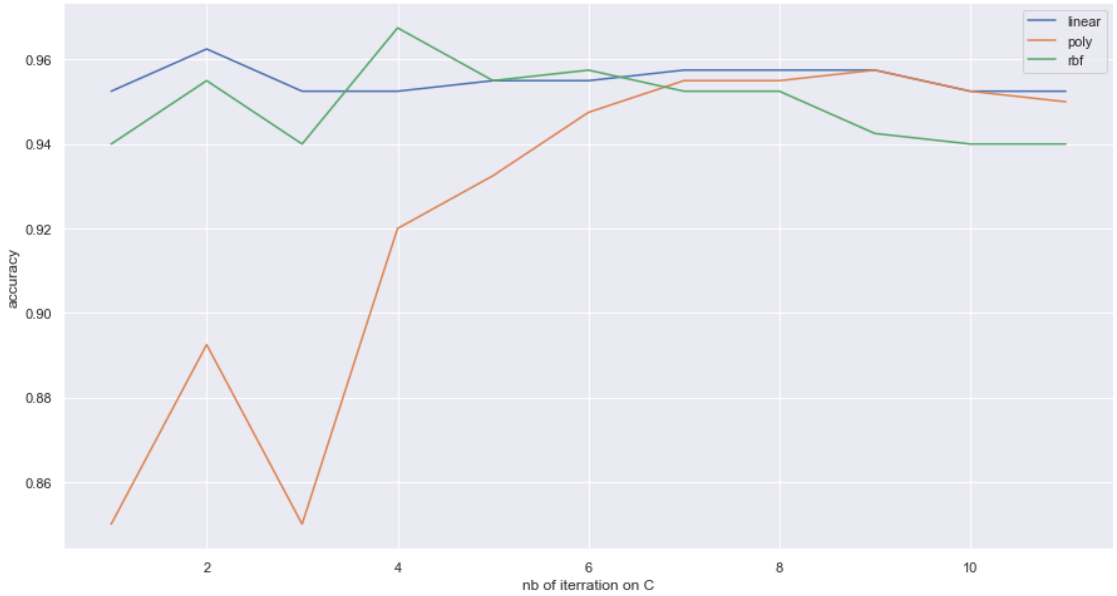
Nous avons donc relancé le XGBoost avec nos 4 variables et 43 estimateurs. On obtient un score de 1 pour la base d'entraînement et de 0.94 pour la base de validation. Encore une fois, la sélection de variable nous a permis d'améliorer la performance de notre algorithme.

8.3 SVM

Pour le SVM avec sélection de variables, nous avons égalemeent représenté l'évolution de l'accuracy score en fonction du nombre d'itération, pour 3 fonctions de kernel (linear, poly et rbf).

[38]:
```
plt.plot(l,accur, label='linear')
plt.plot(l,accur_poly, label='poly')
plt.plot(l,accur_rbf, label='rbf')
plt.xlabel('nb of iterration on C')
plt.ylabel('accuracy')
plt.legend()
```

[38]:
```
<matplotlib.legend.Legend at 0x7ff77e4a2790>
```



On remarque ici que les courbes sont nettement moins distinctes que lorsqu'on n'avait pas sélectionné les variables. Les courbes ont tendance à se confondre par endroit. C'est notamment le cas des courbes représentant les fonctions linear et poly qui sont très proches lorsque le nombre d'itération est élevé. Néanmoins, la plus grande différence entre SVM avec et sans sélection de variable, est le kernel. En effet, lorsque le modèle ne considère que les 4 variables retenues, le kernel qui maximise le score est le rbf.

[39]:
```
model_svm_r=SVC(kernel='rbf',C=1)
model_svm_r.fit(X_train_r,y_train)
model_svm_r.predict(X_valid_r)
train_pred=model_svm_r.predict(X_train_r)
test_pred_svm_r=model_svm_r.predict(X_valid_r)
(accuracy_score(y_train, train_pred),accuracy_score(y_valid,␣
 ↪test_pred_svm_r))
```

[39]: (0.9575, 0.9675)

Ainsi, en sélectionnant ce dernier, pour une seule itération, on obtient un score de 0.9575 pour la base d'entraînement et de 0.9675 pour la base de validation.

Nous avons vu dans cette partie que la sélection de variable est un bon moyen d'améliorer nos algorithmes de classification. Les scores des modèles de kNN et XGBoost ont augmenté. Néanmoins, cela ne fonctionne pas toujours, c'est ce que nous avons observé avec le SVM qui, après sélection de variables, a un score plus faible.

9 Modèle sélectionné et prédictions sur la base de validation

Synthèse des accuracy score des différents modèles :

	Lasso	Logistique	KNN	SVM	XBGoost	KNN*	SVM*	XGBoost*
train		0.978125	0.68	0.995	1	0.92125	0.9575	1
test	0.905	0.955	0.6175	0.97	0.9225	0.9175	0.9675	0.94

$*$: modèles avec sélection de variables.

Nous avons testé différents modèles sous différentes approches, à savoir avec et sans sélection de variables, afin de déterminer l'algorithme qui répond le mieux à notre problématique de classification et qui décrit le mieux nos données. Nous avons conclu à l'algorithme SVM sans sélection de variables, qui est le plus performant. Nous avons alors effectué nos prédictions sur la vraie table de validation que nous avons importé au début de ce projet et nommé test.

Notre table test contient 1000 observations et 20 variables et ne possède pas de données manquantes. Nous avons centré et réduit les données car nous utilisons un algorithme qui maximise la marge entre les données et l'hyperplan. Nous avons donc obtenu :

[45]:
```
test_pred=model_svm.predict(X_test)
```

[47]:
```
Predictions=pd.concat([test.id,pd.DataFrame(test_pred)],axis=1)
Predictions.rename(columns={0: "Classe_prédite"})
```

[47]:

	id	Classe_prédite
0	1	2
1	2	3
2	3	2
3	4	3
4	5	1
..	...	...
995	996	2
996	997	1
997	998	0
998	999	2
999	1000	2

[1000 rows x 2 columns]