Project_Exo2_GeneticOnly_Final

January 22, 2021

1 Exo 2 - Avec Variables Génétiques Uniquement - BreastCancer

1.0.1 Elvina Eury

[78]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import KFold
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import LabelEncoder
import category_encoders as ce
from sklearn.compose import make_column_transformer
from sklearn.preprocessing import StandardScaler
from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV
from sklearn.multiclass import OneVsRestClassifier
from sklearn.metrics import classification_report
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
from sklearn.metrics import f1_score
from sklearn.decomposition import PCA
from sklearn.model_selection import RandomizedSearchCV, train_test_split
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegression
from sklearn.linear_model import LassoLarsCV
from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier
from sklearn.metrics import plot_roc_curve
import itertools
from pandas_profiling import ProfileReport
from sklearn.model_selection import RandomizedSearchCV
from sklearn.model_selection import GridSearchCV
from pprint import pprint
from sklearn.pipeline import Pipeline
from sklearn import linear_model
from sklearn.feature_selection import SelectFromModel
```

```
from sklearn.ensemble import BaggingClassifier
#from imblearn.over_sampling import SMOTE


plt.style.use('bmh')
from scipy.stats import iqr
import warnings
warnings.filterwarnings('ignore')
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
<ipython-input-78-02ce5210c12e> in <module>
     34 from sklearn.feature_selection import SelectFromModel
     35 from sklearn.ensemble import BaggingClassifier
---> 36 from imblearn.over_sampling import SMOTE
     37
     38

ModuleNotFoundError: No module named 'imblearn'
```

[7]:
```
cs=pd.read_csv('~/Projet_ML/BreastCancers.csv').T
```

1.1 Data Pre Processing

[8]:
```
new_header = cs.iloc[0]
cs = cs[1:] # data sans header
cs.columns = new_header
cs.head(5)
```

[8]:
```
Sample_geo_accession Sample_title              tissue age ethnicity  \
GSM505327             BR_FNA_M157  breast cancer cells  57     white
GSM505328             BR_FNA_M196  breast cancer cells  69     asian
GSM505329             BR_FNA_M176  breast cancer cells  77     mixed
GSM505330             BR_FNA_M214  breast cancer cells  54     white
GSM505331             BR_FNA_M113  breast cancer cells  75     black

Sample_geo_accession treatment_response T (tumor) N (Node) bmn_grade  \
GSM505327                            RD         2        0         2
GSM505328                            RD         2        1         2
GSM505329                            RD         4        1         2
GSM505330                            RD         2        1         2
GSM505331                            RD         2        0         3

Sample_geo_accession PR_status:  ER_status:   … AFFX-r2-Hs28SrRNA-5_at  \
GSM505327                      P           P  …                 7.4678
GSM505328                      P           P  …                 9.6656
```

```
GSM505329                       N           P  …                7.6012
GSM505330                       N           P  …                7.6331
GSM505331                       N           N  …                8.0249

Sample_geo_accession AFFX-r2-Hs28SrRNA-M_at AFFX-r2-P1-cre-3_at  \
GSM505327                             9.3738             15.6236
GSM505328                               8.85             15.3234
GSM505329                             8.2567             15.4604
GSM505330                             9.0089             15.5185
GSM505331                             9.2004             15.3143

Sample_geo_accession AFFX-r2-P1-cre-5_at AFFX-ThrX-3_at AFFX-ThrX-5_at  \
GSM505327                        15.2785         3.2915         3.6526
GSM505328                        15.1286         3.3811          2.588
GSM505329                        15.2674         3.1665         3.9743
GSM505330                        15.1655         4.0045         3.8503
GSM505331                        14.9506         3.0514         3.2946

Sample_geo_accession AFFX-ThrX-M_at AFFX-TrpnX-3_at AFFX-TrpnX-5_at  \
GSM505327                   2.6412          1.2652           3.069
GSM505328                   4.4798          4.8098          3.1637
GSM505329                   5.2597          4.3815          2.8034
GSM505330                   5.9114          0.7882          3.1831
GSM505331                   5.1537          3.9179          3.1881

Sample_geo_accession AFFX-TrpnX-M_at
GSM505327                    2.0271
GSM505328                    2.4758
GSM505329                    2.4669
GSM505330                     3.482
GSM505331                    2.9769

[5 rows x 22298 columns]
```

Il y a 279 observations et 22298 variables

Nous sommes dans un cas où le nombre de variables > que le nombre d'observations

1.1.1 Analyse des données

Analyse de la variable réponse La variable treatment_response est notre variable Y: Elle est de type binaire: RD ou pCR

[71]:
```
set(cs['treatment_response'])
```

[71]:
```
{'RD', 'pCR'}
```

La variable réponse a 5 catégories, dont 3 : - 'her2 status: N' - 'pr_status: N' - 'pr_status: P'

semblent être des erreurs. Je vais les enlever de la base de données. Enfin, la variable treatment response n'aura que les 2 modalités: pCR et RD.

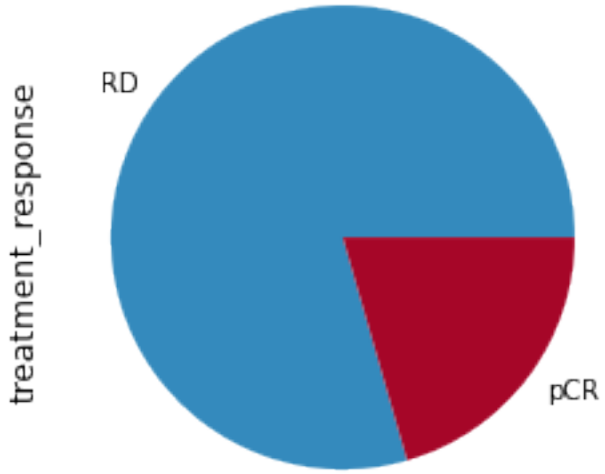
[72]:
```
cs=cs[(cs.treatment_response != 'her2 status: N') & (cs.treatment_response !=␣
↪'pr_status: N')& (cs.treatment_response != 'pr_status: P')]
```

[73]:
```
set(cs['treatment_response'])
```

[73]:
```
{'RD', 'pCR'}
```

[77]:
```
cs.treatment_response.value_counts().plot(kind='pie')
```

[77]:
```
<AxesSubplot:ylabel='treatment_response'>
```



La variable réponse, treatment_response a bien 2 modalités. Le traitement étant efficace dans le cas PCR, non efficace dans le cas RD, il n'est pas étonnant de voir qu'il y a beaucoup de RD que de pCR. Toutefois, nous voyons en avance qu'il y a un soucis de déséquilibre. Nous penserons à adapter les algorithmes pour prendre en compte ce déséquilibre.

1.1.2 Analyse des valeurs manquantes

[12]:
```
missing_data = pd.DataFrame({'total_missing': cs.isnull().sum(axis = 0),␣
↪'perc_missing': (cs.isnull().sum()/len(cs))*100})

missing_data
```

[12]:
```
                      total_missing  perc_missing
Sample_geo_accession
Sample_title                      0           0.0
tissue                            0           0.0
age                               0           0.0
ethnicity                         0           0.0
treatment_response                0           0.0
…                               …             …
AFFX-ThrX-5_at                    0           0.0
AFFX-ThrX-M_at                    0           0.0
AFFX-TrpnX-3_at                   0           0.0
AFFX-TrpnX-5_at                   0           0.0
AFFX-TrpnX-M_at                   0           0.0

[22298 rows x 2 columns]
```

[13]:
```
columns_with_missing_values = missing_data.loc[missing_data['perc_missing']>0]
columns_with_missing_values
```

[13]:
```
                      total_missing  perc_missing
Sample_geo_accession
treatment code                   11      4.104478
```

Les 10 observations manquantes dans la variable 11, her2_status, sont également manquantes dans la variable 12,histology. Cette dernière Comme nous n'avons pas beaucoup d'observations comparés aux variables nous n'allons pas enlever les observations de notre base de données. De plus, comme il y a beaucoup de valeurs manquantes surtout pour les variables 11 et 12, nous allons utiliser une technique d'imputation, soit nous allons assigner à chacune des valeurs manquantes, une valeur calculée. Il existe différentes techniques d'imputation ou d'assignation des valeurs manquantes. Voici les plus communes: - l'utilisation de la valeur moyenne d'une variable, aussi appelé 'mean imputation'. - l'utilisation de la valeur moyenne des k plus proches voisins associés à des données entrainés, aussi appelé knn imputation - l'utilisation de la fréquence, une technique généralement utilisée lorsque les variables sont qualitatives.

Comme la variable 11, her2_status et la variable histology sont de type qualitatives, nous utiliserons la fréquence comme technique d'assignation.

[14]:
```
# Je crée une liste des variables ayant des données manquantes
missing_variables=(columns_with_missing_values.index).tolist()
missing_variables
```

[14]:
```
['treatment code']
```

[15]:
```
# Je remplace les valeurs manquantes par les valeurs les plus communes des␣
 ↪variables.
quali = cs.loc[:,missing_variables].apply(lambda x: x.fillna(x.value_counts().
 ↪index[0]))
# Je crée un dataframe SANS les variables de la liste missing_variables
```

```
cs_without_quali = cs.drop(missing_variables, axis=1) # ok, il manque bien les␣
↪5 colonnes manquantes
```

[16]:
```
# Je crée un nouveau dataframe joignant toutes les variables.
cs_imputed=pd.concat([quali,cs_without_quali],axis=1)
cs_imputed.head(5) # il y a bien toutes les colonnes et lignes (278 x 22298)
```

[16]:
```
Sample_geo_accession treatment code Sample_title              tissue age  \
GSM505327                      TFAC  BR_FNA_M157  breast cancer cells  57
GSM505328                      TFAC  BR_FNA_M196  breast cancer cells  69
GSM505329                      TFAC  BR_FNA_M176  breast cancer cells  77
GSM505330                      TFAC  BR_FNA_M214  breast cancer cells  54
GSM505331                      TFAC  BR_FNA_M113  breast cancer cells  75

Sample_geo_accession ethnicity treatment_response T (tumor) N (Node)  \
GSM505327                white                  RD         2        0
GSM505328                asian                  RD         2        1
GSM505329                mixed                  RD         4        1
GSM505330                white                  RD         2        1
GSM505331                black                  RD         2        0

Sample_geo_accession bmn_grade PR_status:   … AFFX-r2-Hs28SrRNA-5_at  \
GSM505327                    2           P  …                 7.4678
GSM505328                    2           P  …                 9.6656
GSM505329                    2           N  …                 7.6012
GSM505330                    2           N  …                 7.6331
GSM505331                    3           N  …                 8.0249

Sample_geo_accession AFFX-r2-Hs28SrRNA-M_at AFFX-r2-P1-cre-3_at  \
GSM505327                            9.3738             15.6236
GSM505328                              8.85             15.3234
GSM505329                            8.2567             15.4604
GSM505330                            9.0089             15.5185
GSM505331                            9.2004             15.3143

Sample_geo_accession AFFX-r2-P1-cre-5_at AFFX-ThrX-3_at AFFX-ThrX-5_at  \
GSM505327                        15.2785         3.2915         3.6526
GSM505328                        15.1286         3.3811          2.588
GSM505329                        15.2674         3.1665         3.9743
GSM505330                        15.1655         4.0045         3.8503
GSM505331                        14.9506         3.0514         3.2946

Sample_geo_accession AFFX-ThrX-M_at AFFX-TrpnX-3_at AFFX-TrpnX-5_at  \
GSM505327                   2.6412          1.2652          3.069
GSM505328                   4.4798          4.8098         3.1637
GSM505329                   5.2597          4.3815         2.8034
GSM505330                   5.9114          0.7882         3.1831
```

```
GSM505331                    5.1537          3.9179          3.1881

Sample_geo_accession AFFX-TrpnX-M_at
GSM505327                     2.0271
GSM505328                     2.4758
GSM505329                     2.4669
GSM505330                      3.482
GSM505331                     2.9769

[5 rows x 22298 columns]
```

[17]:
```
numerical_columns = list(cs.columns[15:])+['treatment_response'] # colonnes génétiques + variable réponse
categorical_columns = list(cs_imputed.columns[:14]).remove('treatment_response') # colonnes catégorielles sans la variable réponse
```

[18]:
```
cs_imputed_genetic=cs_imputed[numerical_columns]
```

Nous standardisons les données quantitatives StandardScaler follows Standard Normal Distribution (SND). Therefore, it makes mean = 0 and scales the data to unit variance. MinMaxScaler scales all the data features in the range [0, 1] or else in the range [-1, 1] if there are negative values in the dataset. This scaling compresses all the inliers in the narrow range [0, 0.005]. In the presence of outliers, StandardScaler does not guarantee balanced feature scales, due to the influence of the outliers while computing the empirical mean and standard deviation. This leads to the shrinkage in the range of the feature values.

By using RobustScaler(), we can remove the outliers and then use either StandardScaler or MinMaxScaler for preprocessing the dataset.

Treating outliers

[19]:
```
Q1 = cs_imputed_genetic.iloc[:,:-1].quantile(0.25)
Q3 = cs_imputed_genetic.iloc[:,:-1].quantile(0.75)
IQR = Q3 - Q1
print(IQR)
cs_imputed_genetic.iloc[:,14:] = cs_imputed_genetic[~((cs_imputed_genetic.iloc[:,14:] < (Q1 - 1.5 * IQR)) |(cs_imputed_genetic.iloc[:,14:] > (Q3 + 1.5 *IQR))).any(axis=1)]
cs_imputed_genetic.shape
```

```
Series([], dtype: float64)
```

[19]:
```
(268, 22284)
```

Comme le shape n'a pas changé on voit qu'il n'y a pas eu de outliers parmi les variables quantitatives (génétiques). Ainsi le StandardScaler pourra être utilisé comme méthode de normalisation.

[20]:
```
scaler = StandardScaler()
scaler.fit_transform(cs_imputed_genetic.iloc[:,:-1].values)
```

[20]:
```
array([[ 0.93465939,  0.3463953 , -1.14600441, …, -1.43328322,
        -1.03473567, -1.05792573],
       [ 0.56078945, -0.32232983,  0.30009164, …,  1.02318208,
        -0.96192125, -0.55005759],
       [ 1.28304169,  0.73198979, -0.3790667 , …,  0.72636323,
        -1.23895437, -0.56013119],
       …,
       [ 0.12719244, -0.92911206,  0.52114106, …,  0.76579586,
        -1.60994655, -0.40540515],
       [-0.20583586, -0.47938087,  0.85481307, …, -0.43208269,
         0.30383488,  0.6476827 ],
       [ 0.22146601, -0.15289021, -0.43427385, …,  1.3998364 ,
         0.43377826,  0.97909297]])
```

LabelEncoding

[21]:
```
# On converti les strings en nombres - on commence donc par faire le label␣
↪encoding
le = LabelEncoder()

cs_imputed_genetic['treatment_response']= le.
↪fit_transform(cs_imputed['treatment_response'])
```

[22]:
```
# Puis je procède avec one hot encoding
# Cette étape est importante pour permettre l'utilisation du ACP en autre
```

[23]:
```
cs_imputed_encoded = pd.get_dummies(cs_imputed_genetic,␣
↪columns=['treatment_response'], drop_first=True)
cs_imputed_encoded.head(5)
```

[23]:

	1007_s_at	1053_at	117_at	121_at	1255_g_at	1294_at	1316_at	\
GSM505327	12.444	8.3774	6.7866	10.2851	5.9064	8.3767	8.0356	
GSM505328	12.2005	7.8592	8.0963	10.4624	4.9582	9.2973	7.0581	
GSM505329	12.6709	8.6762	7.4812	10.1887	5.2332	9.1721	8.6061	
GSM505330	11.6619	8.2557	7.9923	10.7705	6.3296	9.3777	8.4776	
GSM505331	11.8397	8.7971	7.8321	10.2869	5.8389	7.0841	7.3419	

	1320_at	1405_i_at	1431_at	…	AFFX-r2-Hs28SrRNA-M_at	\
GSM505327	6.6745	6.2325	6.845	…	9.3738	
GSM505328	6.4607	6.9047	5.8878	…	8.85	
GSM505329	7.0932	6.594	5.6843	…	8.2567	
GSM505330	6.5878	6.0877	6.5169	…	9.0089	
GSM505331	7.3167	6.3456	6.1708	…	9.2004	

```
          AFFX-r2-P1-cre-3_at AFFX-r2-P1-cre-5_at AFFX-ThrX-3_at  \
GSM505327             15.6236             15.2785         3.2915
GSM505328             15.3234             15.1286         3.3811
GSM505329             15.4604             15.2674         3.1665
GSM505330             15.5185             15.1655         4.0045
GSM505331             15.3143             14.9506         3.0514

          AFFX-ThrX-5_at AFFX-ThrX-M_at AFFX-TrpnX-3_at AFFX-TrpnX-5_at  \
GSM505327         3.6526         2.6412          1.2652           3.069
GSM505328          2.588         4.4798          4.8098          3.1637
GSM505329         3.9743         5.2597          4.3815          2.8034
GSM505330         3.8503         5.9114          0.7882          3.1831
GSM505331         3.2946         5.1537          3.9179          3.1881

          AFFX-TrpnX-M_at treatment_response_1
GSM505327          2.0271                    0
GSM505328          2.4758                    0
GSM505329          2.4669                    0
GSM505330           3.482                    0
GSM505331          2.9769                    0

[5 rows x 22284 columns]
```

1.2 Model selection

[24]:
```
X=cs_imputed_encoded.iloc[:, cs_imputed_encoded.columns !=␣
 ↪'treatment_response_1']
y=cs_imputed_encoded.treatment_response_1
```

Je commence par 'split' les données, ici je choisi un split de 35% (un choix qui permet de garder des données tests assez conséquente pour mieux prédire).

[25]:
```
Xf_train, Xf_test, yf_train, yf_test = train_test_split(X, y, test_size=0.
 ↪35,random_state=42)
print('Xf_train: ', Xf_train.shape)
print('Xf_test: ',Xf_test.shape)
```

```
Xf_train:  (174, 22283)
Xf_test:  (94, 22283)
```

Reduction de dimension Nous cherchons maintenant à réduire le nombre de dimensions. Tel que vu précédemment nous avons plus de 22000 variables. Il existe plusieurs techniques de réduction de dimension, tel que l'utilisation du Lasso (qui élimine les variables moins significatives) ou le Ridge (qui réduit ces dites variables), ou encore l'Elastic-Net qui lui combine à la fois le Lasso et le Ridge.

Nous choisissons ici d'utiliser une autre technique, soit l'ACP, afin de réduire les dimensions.

[26]:
```
pca1=PCA()
pca1.fit(Xf_train)
cum_sum=np.cumsum(pca1.explained_variance_ratio_)
d=np.argmax(cum_sum>=0.98)+1 # ici nous calcuons la valeur d qui maximise la␣
 ↪variance expliquée à 95%.
```

[27]:
```
print('Le nombre de variables initiales : ', Xf_train.shape[1])
print("Le nombre de variables après l'ACP en choisissant 95% d'inertie : ", d)
```

```
Le nombre de variables initiales :  22283
Le nombre de variables après l'ACP en choisissant 95% d'inertie :  163
```

On remarque qu'avec 98% de variance expliquée on passe de 22701 dimensions à 163 dimensions ce qui représente une nette réduction de dimension. Une ACP à 98% sera ainsi utilisé plus tard dans les pipelines.

Maintenant je fais la sélection de variables (réduction de dimensions) à l'aide du lasso.

[28]:
```
sel_ = SelectFromModel(LogisticRegression(C=3.
 ↪3,penalty='l1',solver='liblinear'))
sel_.fit(scaler.transform(Xf_train), yf_train)
```

[28]:
```
SelectFromModel(estimator=LogisticRegression(C=3.3, penalty='l1',
                                             solver='liblinear'))
```

[29]:
```
selected_feat = Xf_train.columns[(sel_.get_support())]
print('total features: {}'.format((Xf_train.shape[1])))
print('selected features: {}'.format(len(selected_feat)))
print('features with coefficients shrank to zero: {}'.format(
      np.sum(sel_.estimator_.coef_ == 0)))
```

```
total features: 22283
selected features: 194
features with coefficients shrank to zero: 22089
```

[30]:
```
sel_1 = SelectFromModel(LogisticRegression(C=1,penalty='l1',solver='liblinear'))
sel_1.fit(scaler.transform(Xf_train), yf_train)
selected_feat = Xf_train.columns[(sel_1.get_support())]
print('total features: {}'.format((Xf_train.shape[1])))
print('selected features: {}'.format(len(selected_feat)))
print('features with coefficients shrank to zero: {}'.format(
      np.sum(sel_.estimator_.coef_ == 0)))
```

```
total features: 22283
selected features: 108
features with coefficients shrank to zero: 22089
```

En ce qui concerne le Lasso, on va faire varier la pénalisation afin d'obtenir les meilleurs résultats (meilleur performance - comme le accuracy).

Creation de Pipelines Dans cette section je vais construire plusieurs pipelines. La meilleure approche est de créer un seul pipeline qui nous sortirait le meilleur modèle, la meilleur démarche automatiquement. Toutefois, afin de mieux analyser les différents résultats, j'ai opté d'utiliser des pipelines séparés.

Il aurait été également possible d'incorporer le StandardScaler des variables quantitatives et le OneHotEncoding(ou LabelEncoding) des variables qualitatives directement dans les pipelines mais comme cela a déjà été fait au début lors du prétraitement de données, je n'utiliserai pas ces options ici. Toutefois, cela fait partie des améliorations qui pourraient être fait dans le future afin d'uniformiser la structure et rendre le programme plus performant.

On construit les pipelines

[48]:
```
########################## SANS Réduction de dimension␣
↪#########################################

# SVM sans ACP
pipe_svm = Pipeline([('clf', SVC(random_state=42))])


########################## AVEC Réduction de dimension (ACP)␣
↪#####################################

# SVM avec ACP
pipe_svm_pca = Pipeline([('pca', PCA(0.98)),
                         ('clf', SVC(random_state=42))])

# XGBoost avec ACP
pipe_xgb_pca = Pipeline([('pca', PCA(0.98)),
                         ('clf', XGBClassifier(learning_rate=0.02,␣
↪n_estimators=600, objective='binary:logistic', nthread=1))])

# Random Forest avec ACP
pipe_rf_pca = Pipeline([('pca', PCA(0.98)),
                         ('clf', RandomForestClassifier())])

########################## AVEC Réduction de dimension (LASSO)␣
↪#####################################

# SVM avec Lasso
pipe_svm_lasso = Pipeline([('feature_selection',␣
↪SelectFromModel(LogisticRegression(C=1,penalty='l1',solver='liblinear'))),
                         ('clf', SVC())])

# XGBoost avec Lasso
pipe_xgb_lasso = Pipeline([('feature_selection',␣
↪SelectFromModel(LogisticRegression(C=1,penalty='l1',solver='liblinear'))),
```

```
                        ('clf', XGBClassifier(learning_rate=0.02,␣
 ↪n_estimators=600, objective='binary:logistic', nthread=1))])

# Random Forest avec Lasso
pipe_rf_lasso = Pipeline([('feature_selection',␣
 ↪SelectFromModel(LogisticRegression(C=1,penalty='l1',solver='liblinear'))),
                        ('clf', RandomForestClassifier())])




########################## Régression Logistique pénalisée␣
 ↪#####################################

# Régression Logistique pénalisée
pipe_lr = Pipeline([('clf', LogisticRegression(random_state=42))])
```

On crée le grid des paramètres

[49]:
```
# grid_params_lr_initial = '{'clf__penalty': ['l1', 'l2','elasticnet'],'clf__C':
 ↪ [1.0, 0.5, 0.1], 'clf__solver': ['liblinear','saga']}]
# 'penalty' compare le lasso (l1), le ridge(l2) et l'elasticnet
# 'solver' est l'algorithme pour l'optimisation, saga est le seul utilisé pour␣
 ↪l'elastic net.

grid_params_lr = [{'clf__penalty': ['l1','l2','elasticnet'],
'clf__C': [0.05,0.1,0.15], # le degré de pénalisation
'clf__solver': ['liblinear','saga']}]

# grid_params_svm_initial = ''clf__C': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],␣
 ↪clf__gamma':[0.00005,0.0001,0.001]}].
# Après avoir utilisé grid_params_svm_initial, je réduit les choix des␣
 ↪paramètres afin de raccourcir le temps de calcul
grid_params_svm = [{'clf__kernel': ['linear', 'rbf'],
'clf__C': [1,2],
'clf__gamma':[0.00005]}]

# ATTENTION TRÈS LONG
# grid_params_xgb_initial = [ { 'clf__min_child_weight': [1, 5, 10],␣
 ↪'clf__gamma': [0.5, 1, 1.5, 2, 5], 'clf__subsample': [0.6, 0.8, 1.
 ↪0],'clf__colsample_bytree': [0.6, 0.8, 1.0],'clf__max_depth': [3, 4, 5]}]
# Après avoir utilisé grid_params_xgb_initial, je réduit les choix des␣
 ↪paramètres afin de raccourcir le temps de calcul:
grid_params_xgb=[{'clf__colsample_bytree': [0.8,1.0],
                  'clf__gamma': [2,2.5],
                  'clf__max_depth': [2,3]}]
```

```
# grid_params_rf_initial = '[{ 'clf__bootstrap': [True],'clf__max_depth':␣
 ↪[60,80, 90, 100, 110],'clf__max_features':␣
 ↪[10,12,14],'clf__min_samples_leaf': [2,3,4],'clf__min_samples_split':␣
 ↪[3,4,5],'clf__n_estimators': [20,100,200]}]
# Après avoir utilisé grid_params_rf_initial, je réduit les choix des␣
 ↪paramètres afin de raccourcir le temps de calcul
grid_params_rf = [{
    'clf__bootstrap': [False],
    'clf__max_depth': [600],
    'clf__max_features': [40],
    'clf__min_samples_leaf': [5,6,8,10,15],
    'clf__min_samples_split': [5],
    'clf__n_estimators': [22]}]
```

On crée les gridSearchCV

[50]:
```
# Sans réduction de dimension
jobs = -1
cv = KFold(n_splits=3, shuffle=True, random_state=1)
# je choisi de divisé en 3 et pas plus car la taille des données est petite.



gs_svm = GridSearchCV(estimator=pipe_svm,
param_grid=grid_params_svm,
scoring='accuracy',
cv=cv,
n_jobs=jobs)


# Avec Réduction de dimension

gs_lr = GridSearchCV(estimator=pipe_lr,
param_grid=grid_params_lr,
scoring='accuracy',
cv=cv,n_jobs=jobs)

gs_svm_pca = GridSearchCV(estimator=pipe_svm_pca,
param_grid=grid_params_svm,
scoring='accuracy',
cv=cv,
n_jobs=jobs)

gs_xgb_pca=GridSearchCV(estimator=pipe_xgb_pca,
param_grid=grid_params_xgb,
scoring='accuracy',cv=cv, n_jobs=jobs)
```

```
gs_rf_pca=GridSearchCV(estimator=pipe_rf_pca,
param_grid=grid_params_rf,
scoring='accuracy',
cv=cv,n_jobs=jobs)

gs_svm_lasso = GridSearchCV(estimator=pipe_svm_lasso,
param_grid=grid_params_svm,
scoring='accuracy',
cv=cv,
n_jobs=jobs)

gs_xgb_lasso=GridSearchCV(estimator=pipe_xgb_lasso,
param_grid=grid_params_xgb,
scoring='accuracy',cv=cv, n_jobs=jobs)

gs_rf_lasso=GridSearchCV(estimator=pipe_rf_lasso,
param_grid=grid_params_rf,
scoring='accuracy',
cv=cv,n_jobs=jobs)
```

[51]:
```
grids = [gs_svm, gs_lr,␣
↪gs_svm_pca,gs_xgb_pca,gs_rf_pca,gs_svm_lasso,gs_xgb_lasso,gs_rf_lasso]
```

[52]:
```
# Dictionary of pipelines and classifier types for ease of reference
grid_dict = {0: 'Support Vector Machine sans réduction de dimensions',
             1: 'Logistic Regression pénalisé',
             2: 'Support Vector Machine avec Réduction: PCA',
             3: 'XGBoost avec Réduction: PCA',
             4: 'Random Forest avec Réduction: PCA',
             5: 'Support Vector Machine avec Réduction: Lasso Logistique',
             6: 'XGBoost avec Réduction: Lasso Logistique',
             7: 'Random Forest avec Réduction: Lasso Logistique'
             }
```

[53]:
```
#Plotting the confusion matrix
class_names=['RD','pCR']

def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):


    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
```

```
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)



    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, cm[i, j],
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")


    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
```

[54]:
```
# ATTENTION LONG À ROULER CAR IL ROULE TOUS LES CLASSIFIEURS ET PARAMÈTRES
```

2 TENTATIVE 1

[55]:
```
# Fit the grid search objects
print('On débute...')
print('Nous utilisons les mesures de performances suivantes: ')
print("   - Accuracy score (f1 score), calculé sur les données test.")
print("   - Le cross validation score, calculé à partir des données␣
 ↪d'entraînement. On aurait pu utiliser le accuracy score sur données train␣
 ↪sans validation croisée, mais c'est plus biasé que celui calculé par␣
 ↪validation croisée.")
print("   - Les matrices de confusion qui affichent la répartition des classes.
 ↪")
print("Nous cherchons de plus à trouver le bon équilibre entre la précision et␣
 ↪la sensibilité (recall ou taux de vrai positif). Pour nous aider nous nous␣
 ↪baserons sur le f1 score qui est la moyenne harmonique des scores de␣
 ↪précision et de recall. Nous nous baserons également sur les matrices de␣
 ↪confusion afin d'établir l'équilibre des classes.")

best_acc = 0.0
best_clf = 0
best_gs = ''

for idx, gs in enumerate(grids):
    print('\nEstimateur: %s' % grid_dict[idx])
    # Fit sur les données train, permet de trouver l'estimateur (le classifieur␣
 ↪estimé)
    # La validation croisée est faite sur les données train et nous commençons␣
 ↪par calculer le cross validation score à partir des données train.
```

```
    # On s'assure de ne pas 'toucher' aux données tests
    gs.fit(Xf_train, yf_train)
    print('Les meilleurs paramètres sont : %s' % gs.best_params_)
    print('Le cross validation score sur données train: %.3f' % gs.best_score_)

    # Predict sur les données test
    # On ne fait que des prédictions à partir de Xf_test
    y_pred = gs.predict(Xf_test)

    # Le accuracy score est obtenu à partir des données test.
    print('Accuracy score sur données test pour les meilleurs paramètres: %.3f␣
↪' % accuracy_score(yf_test, y_pred))
    print(confusion_matrix(yf_test,y_pred))

    print(classification_report(yf_test, y_pred))
    cm=confusion_matrix(yf_test,y_pred)
    # On veut sortir le classifieur ayant le meilleur accuracy score (1-erreur␣
↪de classification minimum)
    if accuracy_score(yf_test, y_pred) > best_acc:
        best_acc = accuracy_score(yf_test, y_pred)
        best_gs = gs
        best_clf = idx
        best_yf=y_pred
```

```
On débute…
Nous utilisons les mesures de performances suivantes:
   - Accuracy score (f1 score), calculé sur les données test.
   - Le cross validation score, calculé à partir des données d'entraînement. On
aurait pu utiliser le accuracy score sur données train sans validation croisée,
mais c'est plus biasé que celui calculé par validation croisée.
   - Les matrices de confusion qui affichent la répartition des classes.
Nous cherchons de plus à trouver le bon équilibre entre la précision et la
sensibilité (recall ou taux de vrai positif). Pour nous aider nous nous baserons
sur le f1 score qui est la moyenne harmonique des scores de précision et de
recall. Nous nous baserons également sur les matrices de confusion afin
d'établir l'équilibre des classes.

Estimateur: Support Vector Machine sans réduction de dimensions
Les meilleurs paramètres sont : {'clf__C': 1, 'clf__gamma': 5e-05,
'clf__kernel': 'linear'}
Le cross validation score sur données train: 0.816
Accuracy score sur données test pour les meilleurs paramètres: 0.883
[[75  3]
 [ 8  8]]
              precision    recall  f1-score   support
```

```
              precision    recall  f1-score   support

           0       0.90      0.96      0.93        78
           1       0.73      0.50      0.59        16

    accuracy                           0.88        94
   macro avg       0.82      0.73      0.76        94
weighted avg       0.87      0.88      0.87        94
```

```
Estimateur: Logistic Regression pénalisé
Les meilleurs paramètres sont : {'clf__C': 0.1, 'clf__penalty': 'l1',
'clf__solver': 'liblinear'}
Le cross validation score sur données train: 0.828
Accuracy score sur données test pour les meilleurs paramètres: 0.830
[[73  5]
 [11  5]]
              precision    recall  f1-score   support

           0       0.87      0.94      0.90        78
           1       0.50      0.31      0.38        16

    accuracy                           0.83        94
   macro avg       0.68      0.62      0.64        94
weighted avg       0.81      0.83      0.81        94
```

```
Estimateur: Support Vector Machine avec Réduction: PCA
Les meilleurs paramètres sont : {'clf__C': 2, 'clf__gamma': 5e-05,
'clf__kernel': 'rbf'}
Le cross validation score sur données train: 0.833
Accuracy score sur données test pour les meilleurs paramètres: 0.840
[[70  8]
 [ 7  9]]
              precision    recall  f1-score   support

           0       0.91      0.90      0.90        78
           1       0.53      0.56      0.55        16

    accuracy                           0.84        94
   macro avg       0.72      0.73      0.72        94
weighted avg       0.84      0.84      0.84        94
```

```
Estimateur: XGBoost avec Réduction: PCA
Les meilleurs paramètres sont : {'clf__colsample_bytree': 1.0, 'clf__gamma': 2,
'clf__max_depth': 2}
Le cross validation score sur données train: 0.782
Accuracy score sur données test pour les meilleurs paramètres: 0.840
```

```
[[78  0]
 [15  1]]
              precision    recall  f1-score   support

           0       0.84      1.00      0.91        78
           1       1.00      0.06      0.12        16

    accuracy                           0.84        94
   macro avg       0.92      0.53      0.51        94
weighted avg       0.87      0.84      0.78        94
```

```
Estimateur: Random Forest avec Réduction: PCA
Les meilleurs paramètres sont : {'clf__bootstrap': False, 'clf__max_depth': 600,
'clf__max_features': 40, 'clf__min_samples_leaf': 6, 'clf__min_samples_split':
5, 'clf__n_estimators': 22}
Le cross validation score sur données train: 0.776
Accuracy score sur données test pour les meilleurs paramètres: 0.840
[[78  0]
 [15  1]]
              precision    recall  f1-score   support

           0       0.84      1.00      0.91        78
           1       1.00      0.06      0.12        16

    accuracy                           0.84        94
   macro avg       0.92      0.53      0.51        94
weighted avg       0.87      0.84      0.78        94
```

```
Estimateur: Support Vector Machine avec Réduction: Lasso Logistique
Les meilleurs paramètres sont : {'clf__C': 1, 'clf__gamma': 5e-05,
'clf__kernel': 'linear'}
Le cross validation score sur données train: 0.822
Accuracy score sur données test pour les meilleurs paramètres: 0.872
[[74  4]
 [ 8  8]]
              precision    recall  f1-score   support

           0       0.90      0.95      0.92        78
           1       0.67      0.50      0.57        16

    accuracy                           0.87        94
   macro avg       0.78      0.72      0.75        94
weighted avg       0.86      0.87      0.86        94
```

```
Estimateur: XGBoost avec Réduction: Lasso Logistique
```

```
Les meilleurs paramètres sont : {'clf__colsample_bytree': 1.0, 'clf__gamma': 2,
'clf__max_depth': 3}
Le cross validation score sur données train: 0.787
Accuracy score sur données test pour les meilleurs paramètres: 0.819
[[74  4]
 [13  3]]
              precision    recall  f1-score   support

           0       0.85      0.95      0.90        78
           1       0.43      0.19      0.26        16

    accuracy                           0.82        94
   macro avg       0.64      0.57      0.58        94
weighted avg       0.78      0.82      0.79        94


Estimateur: Random Forest avec Réduction: Lasso Logistique
Les meilleurs paramètres sont : {'clf__bootstrap': False, 'clf__max_depth': 600,
'clf__max_features': 40, 'clf__min_samples_leaf': 10, 'clf__min_samples_split':
5, 'clf__n_estimators': 22}
Le cross validation score sur données train: 0.799
Accuracy score sur données test pour les meilleurs paramètres: 0.787
[[72  6]
 [14  2]]
              precision    recall  f1-score   support

           0       0.84      0.92      0.88        78
           1       0.25      0.12      0.17        16

    accuracy                           0.79        94
   macro avg       0.54      0.52      0.52        94
weighted avg       0.74      0.79      0.76        94
```

[56]:
```
print('Meilleur classifieur en se basant sur le accuracy score:␣
 ↪',grid_dict[best_clf])
print('Meilleur accuracy score : ',round(best_acc,3))
print('Erreur de classification : ',round(1-best_acc,3))
```

```
Meilleur classifieur en se basant sur le accuracy score:  Support Vector Machine
sans réduction de dimensions
Meilleur accuracy score :  0.883
Erreur de classification :  0.117
```

Tel que mentionné ci-dessus nous nous basons sur les mesures suivantes afin de mieux sélectionner notre modèle:

-> Accuracy score (f1 score), calculé sur les données test
-> Le cross validation score, calculé à partir des données d'entraînement. On aurait pu utilis

`-> Les matrices de confusion qui affichent la répartition des classes.`

Le classifieur le plus robuste dans notre cas est le SVM. En général le SVM sans méthode de réduction de dimension est celui qui donne le meilleur f1-score. Tel que mentionné ci-dessus, ce score est une moyenne harmonique des scores de précision et de recall. Ainsi il fera parti de nos critères de sélection de modèle. Nous prenons comme mesure le score, soit le pourcentage d'être bien classé, au lieu de l'erreur de classification, le score étant 100% - pourcentage d'erreur de classification. Nous utilisons cette mesure car elle est plus souvent utilisée avec sklearn.

Un autre critère de sélection de modèle est la comparaison du f1-score (obtenu des données tests) avec le score moyen des validations croisée qui lui est calculé en faisant la moyenne des scores obtenus des 3 folds, pris des données d'entraînement.

Si le score obtenu des données d'entraînement (cross validation score ici) est beaucoup plus grand que celui obtenu des données test (f1-score ici), alors il y aurait possiblement un soucis de surapprentissage. Si le cross-validation score est par contre juste un peu plus élevé ou proche du f1-score alors, on aurait tendance à penser que le modèle est de 'good fit', et donc serait un bon modèle à considérer.

Dans notre cas, le SVM sans réduction de dimension est celui ayant le meilleur f1-score, et nous remarquons que le score obtenu des données train est plus bas que celui obtenu des données test, ce qui ne pousse pas à penser à un problème de surapprentissage. Toutefois on se rappelle qu'il n'y a eu aucune réduction de dimension c'est à dire qu'on travaille dans plus de 22000 dimensions. Ainsi cela nous pousse à considérer un autre modèle.

Nous optons pour le SVM avec méthode de réduction de dimension, le lasso (sur régression logistique) car il apporte une réduction de dimension (on a uniquement 107 variables) tout en ayant un f1 score élevé, soit autour de 87.2% (cette valeur fluctue légèrement à chaque qu'on roule à nouveau), et un f1-score plus élevé que le cross validation score. De plus, les scores de précision et de recall sont tous les deux 87%, et ainsi aucun compromis de précision et recall ne doit être fait.

Ainsi le modèle choisi est le SVM avec une réduction de dimension apportée par le Lasso.

Regardons rapidement d'autres modèles. La méthode SVM avec une réduction de dimension apportée par l'ACP a un accuracy score moins élevé, soit de 84%. Toutefois, on voit que ce résultat est très proche du score obtenu avec les données train.

Dans notre cas, les Forêts Aléatoires et XGBoost sont sans surprise des classifieurs plus biaisés car les classes ne sont pas très équilibrés. De plus, on remarque que les Forêts Aléatoires ainsi que le XGBoost sont sensibles aux méthodes de réductions, surtout au niveau des données tests(fluctuation du f1-score). Ainsi ce sont des classifieurs moins robustes dans notre cas.

Finalement nous notons que les résultats obtenus en ne gardant que les variables génétiques et en incluant toutes les variables sont très proches. Il est alors préférable de ne garder que les variables génétiques (car moins de dimensions). Les résultats obtenus à partir de toutes les données sont présentés dans un second rapport.

Analysons maintenant les résultats obtenus dans les matrices de confusion.

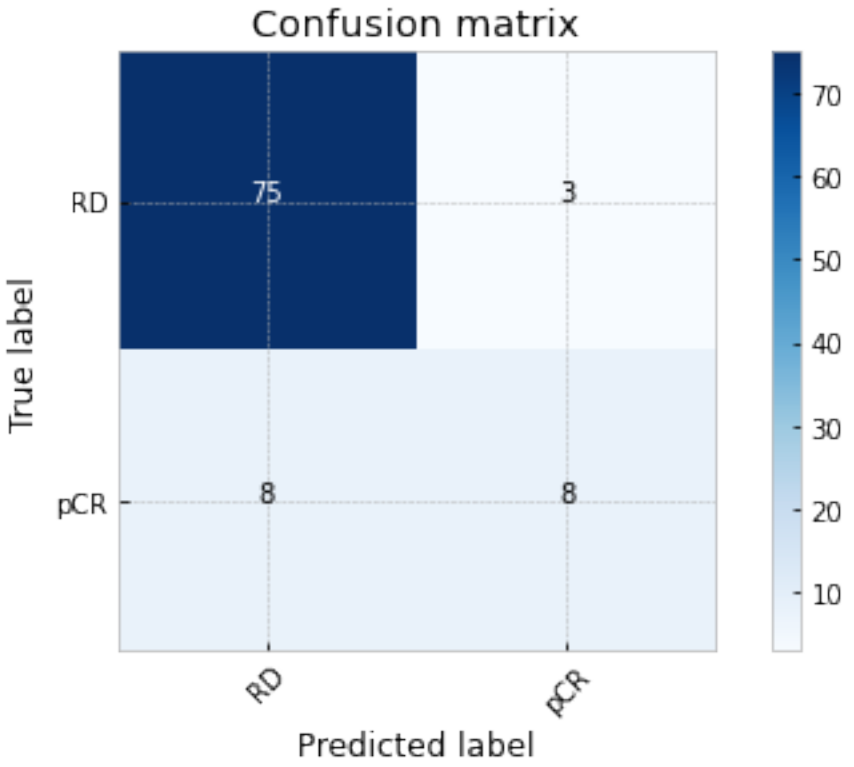
Nous voyons ci-dessous la matrice de confusion du meilleur classifieur, soit SVM avec réduction de dimension apporté le lasso.

La première chose que nous remarquons est le fait que très peu d'observations sont représentées dans la classe pCR, contrairement à la classe RD. De plus les observations sont classés de manière

50/50 dans la vrai classe pCR.

Notre prochaine étape sera d'essayer de mieux équilibrer les résultats. Pour cela nous commençons par ajouter l'option 'class_weights' proposés par sklearn. Cette fonction permet de gérer les déséquilibres dans les classes. Si cette méthode ne fonctionne pas, nous tenterons d'utiliser le bagging classifier en plus (afin de randomniser les données).

[57]:
```
# SVM
cm=confusion_matrix(yf_test,best_yf)
plot_confusion_matrix(cm, classes=class_names, title='Confusion matrix')
```



2.0.1 TENTATIVE 2

[58]:
```
# grid_params_lr_initial = '{'clf__penalty': ['l1', 'l2','elasticnet'],'clf__C':
 ↪ [1.0, 0.5, 0.1], 'clf__solver': ['liblinear','saga']}]
# 'penalty' compare le lasso (l1), le ridge(l2) et l'elasticnet
# 'solver' est l'algorithme pour l'optimisation, saga est le seul utilisé pour␣
 ↪l'elastic net.

grid_params_lr = [{'clf__penalty': ['l1','l2','elasticnet'],
'clf__C': [0.05,0.1,0.15], # le degré de pénalisation
'clf__solver': ['liblinear','saga'],
'clf__class_weight':['balanced']}]
```

```
# grid_params_svm_initial = ''clf__C': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],␣
 ↪clf__gamma':[0.00005,0.0001,0.001]}].
# Après avoir utilisé grid_params_svm_initial, je réduit les choix des␣
 ↪paramètres afin de raccourcir le temps de calcul
grid_params_svm = [{'clf__kernel': ['linear', 'rbf'],
'clf__C': [1,2],
'clf__gamma':[0.00005],
'clf__class_weight':['balanced']}]

# ATTENTION TRÈS LONG
# grid_params_xgb_initial = [ { 'clf__min_child_weight': [1, 5, 10],␣
 ↪'clf__gamma': [0.5, 1, 1.5, 2, 5], 'clf__subsample': [0.6, 0.8, 1.
 ↪0],'clf__colsample_bytree': [0.6, 0.8, 1.0],'clf__max_depth': [3, 4, 5]}]
# Après avoir utilisé grid_params_xgb_initial, je réduit les choix des␣
 ↪paramètres afin de raccourcir le temps de calcul:
grid_params_xgb=[{'clf__colsample_bytree': [0.8,1.0],
                  'clf__gamma': [2,2.5],
                  'clf__max_depth': [2,3],
                  'clf__class_weight':['balanced']}]

# grid_params_rf_initial = '[{ 'clf__bootstrap': [True],'clf__max_depth':␣
 ↪[60,80, 90, 100, 110],'clf__max_features':␣
 ↪[10,12,14],'clf__min_samples_leaf': [2,3,4],'clf__min_samples_split':␣
 ↪[3,4,5],'clf__n_estimators': [20,100,200]}]
# Après avoir utilisé grid_params_rf_initial, je réduit les choix des␣
 ↪paramètres afin de raccourcir le temps de calcul
grid_params_rf = [{
    'clf__bootstrap': [False],
    'clf__max_depth': [600],
    'clf__max_features': [40],
    'clf__min_samples_leaf': [5,6,8,10,15],
    'clf__min_samples_split': [5],
    'clf__n_estimators': [22],
    'clf__class_weight':['balanced','balanced_subsample']}]
```

[59]:
```
# Fit the grid search objects
print('On débute...')
print('Nous utilisons les mesures de performances suivantes: ')
print("   - Accuracy score (f1 score), calculé sur les données test.")
print("   - Le cross validation score, calculé à partir des données␣
 ↪d'entraînement. On aurait pu utiliser le accuracy score sur données train␣
 ↪sans validation croisée, mais c'est plus biasé que celui calculé par␣
 ↪validation croisée.")
print("   - Les matrices de confusion qui affichent la répartition des classes.
 ↪")
```

```
print("Nous cherchons de plus à trouver le bon équilibre entre la précision et␣
 ↪la sensibilité (recall ou taux de vrai positif). Pour nous aider nous nous␣
 ↪baserons sur le f1 score qui est la moyenne harmonique des scores de␣
 ↪précision et de recall. Nous nous baserons également sur les matrices de␣
 ↪confusion afin d'établir l'équilibre des classes.")

best_acc = 0.0
best_clf = 0
best_gs = ''

for idx, gs in enumerate(grids):
    print('\nEstimateur: %s' % grid_dict[idx])
    # Fit sur les données train, permet de trouver l'estimateur (le classifieur␣
 ↪estimé)
    # La validation croisée est faite sur les données train et nous commençons␣
 ↪par calculer le cross validation score à partir des données train.
    # On s'assure de ne pas 'toucher' aux données tests
    gs.fit(Xf_train, yf_train)
    print('Les meilleurs paramètres sont : %s' % gs.best_params_)
    print('Le cross validation score sur données train: %.3f' % gs.best_score_)

    # Predict sur les données test
    # On ne fait que des prédictions à partir de Xf_test
    y_pred = gs.predict(Xf_test)

    # Le accuracy score est obtenu à partir des données test.
    print('Accuracy score sur données test pour les meilleurs paramètres: %.3f␣
 ↪' % accuracy_score(yf_test, y_pred))
    print(confusion_matrix(yf_test,y_pred))

    print(classification_report(yf_test, y_pred))
    cm=confusion_matrix(yf_test,y_pred)
    # On veut sortir le classifieur ayant le meilleur accuracy score (1-erreur␣
 ↪de classification minimum)
    if accuracy_score(yf_test, y_pred) > best_acc:
        best_acc = accuracy_score(yf_test, y_pred)
        best_gs = gs
        best_clf = idx
        best_yf=y_pred
```

On débute…
Nous utilisons les mesures de performances suivantes:
- Accuracy score (f1 score), calculé sur les données test.
- Le cross validation score, calculé à partir des données d'entraînement. On aurait pu utiliser le accuracy score sur données train sans validation croisée, mais c'est plus biasé que celui calculé par validation croisée.

- Les matrices de confusion qui affichent la répartition des classes.

Nous cherchons de plus à trouver le bon équilibre entre la précision et la sensibilité (recall ou taux de vrai positif). Pour nous aider nous nous baserons sur le f1 score qui est la moyenne harmonique des scores de précision et de recall. Nous nous baserons également sur les matrices de confusion afin d'établir l'équilibre des classes.

Estimateur: Support Vector Machine sans réduction de dimensions
Les meilleurs paramètres sont : {'clf__C': 1, 'clf__gamma': 5e-05, 'clf__kernel': 'linear'}
Le cross validation score sur données train: 0.816
Accuracy score sur données test pour les meilleurs paramètres: 0.883
[[75 3]
 [8 8]]

	precision	recall	f1-score	support
0	0.90	0.96	0.93	78
1	0.73	0.50	0.59	16
accuracy			0.88	94
macro avg	0.82	0.73	0.76	94
weighted avg	0.87	0.88	0.87	94

Estimateur: Logistic Regression pénalisé
Les meilleurs paramètres sont : {'clf__C': 0.1, 'clf__penalty': 'l1', 'clf__solver': 'liblinear'}
Le cross validation score sur données train: 0.828
Accuracy score sur données test pour les meilleurs paramètres: 0.830
[[73 5]
 [11 5]]

	precision	recall	f1-score	support
0	0.87	0.94	0.90	78
1	0.50	0.31	0.38	16
accuracy			0.83	94
macro avg	0.68	0.62	0.64	94
weighted avg	0.81	0.83	0.81	94

Estimateur: Support Vector Machine avec Réduction: PCA
Les meilleurs paramètres sont : {'clf__C': 2, 'clf__gamma': 5e-05, 'clf__kernel': 'rbf'}
Le cross validation score sur données train: 0.833
Accuracy score sur données test pour les meilleurs paramètres: 0.840
[[70 8]
 [7 9]]

```
              precision    recall  f1-score   support

           0       0.91      0.90      0.90        78
           1       0.53      0.56      0.55        16

    accuracy                           0.84        94
   macro avg       0.72      0.73      0.72        94
weighted avg       0.84      0.84      0.84        94
```

```
Estimateur: XGBoost avec Réduction: PCA
Les meilleurs paramètres sont : {'clf__colsample_bytree': 1.0, 'clf__gamma': 2,
'clf__max_depth': 2}
Le cross validation score sur données train: 0.782
Accuracy score sur données test pour les meilleurs paramètres: 0.840
[[78  0]
 [15  1]]
              precision    recall  f1-score   support

           0       0.84      1.00      0.91        78
           1       1.00      0.06      0.12        16

    accuracy                           0.84        94
   macro avg       0.92      0.53      0.51        94
weighted avg       0.87      0.84      0.78        94
```

```
Estimateur: Random Forest avec Réduction: PCA
Les meilleurs paramètres sont : {'clf__bootstrap': False, 'clf__max_depth': 600,
'clf__max_features': 40, 'clf__min_samples_leaf': 6, 'clf__min_samples_split':
5, 'clf__n_estimators': 22}
Le cross validation score sur données train: 0.787
Accuracy score sur données test pour les meilleurs paramètres: 0.830
[[77  1]
 [15  1]]
              precision    recall  f1-score   support

           0       0.84      0.99      0.91        78
           1       0.50      0.06      0.11        16

    accuracy                           0.83        94
   macro avg       0.67      0.52      0.51        94
weighted avg       0.78      0.83      0.77        94
```

```
Estimateur: Support Vector Machine avec Réduction: Lasso Logistique
Les meilleurs paramètres sont : {'clf__C': 1, 'clf__gamma': 5e-05,
'clf__kernel': 'linear'}
```

```
Le cross validation score sur données train: 0.816
Accuracy score sur données test pour les meilleurs paramètres: 0.872
[[74  4]
 [ 8  8]]
              precision    recall  f1-score   support

           0       0.90      0.95      0.92        78
           1       0.67      0.50      0.57        16

    accuracy                           0.87        94
   macro avg       0.78      0.72      0.75        94
weighted avg       0.86      0.87      0.86        94


Estimateur: XGBoost avec Réduction: Lasso Logistique
Les meilleurs paramètres sont : {'clf__colsample_bytree': 0.8, 'clf__gamma': 2,
'clf__max_depth': 3}
Le cross validation score sur données train: 0.782
Accuracy score sur données test pour les meilleurs paramètres: 0.809
[[73  5]
 [13  3]]
              precision    recall  f1-score   support

           0       0.85      0.94      0.89        78
           1       0.38      0.19      0.25        16

    accuracy                           0.81        94
   macro avg       0.61      0.56      0.57        94
weighted avg       0.77      0.81      0.78        94


Estimateur: Random Forest avec Réduction: Lasso Logistique
Les meilleurs paramètres sont : {'clf__bootstrap': False, 'clf__max_depth': 600,
'clf__max_features': 40, 'clf__min_samples_leaf': 5, 'clf__min_samples_split':
5, 'clf__n_estimators': 22}
Le cross validation score sur données train: 0.776
Accuracy score sur données test pour les meilleurs paramètres: 0.798
[[74  4]
 [15  1]]
              precision    recall  f1-score   support

           0       0.83      0.95      0.89        78
           1       0.20      0.06      0.10        16

    accuracy                           0.80        94
   macro avg       0.52      0.51      0.49        94
weighted avg       0.72      0.80      0.75        94
```

Malgré l'utilisation de l'option 'class_weight=balanced', il existe encore un soucis de déséquilibre dans les classes. Nous tentons maintenant d'utiliser le bagging afin de réduire le biais apporté par les données elles-même.

2.0.2 TENTATIVE 3

[65]:
```
########################## SANS Réduction de dimension␣
↪#########################################

# SVM sans ACP
pipe_svm = Pipeline([('clf', BaggingClassifier(SVC(random_state=42)))])


########################## AVEC Réduction de dimension (ACP)␣
↪#####################################

# SVM avec ACP
pipe_svm_pca = Pipeline([('pca', PCA(0.98)),
                         ('clf', BaggingClassifier(SVC(random_state=42)))])

# XGBoost avec ACP
pipe_xgb_pca = Pipeline([('pca', PCA(0.98)),
                         ('clf',␣
↪BaggingClassifier(XGBClassifier(learning_rate=0.02, n_estimators=600,␣
↪objective='binary:logistic', nthread=1)))])

# Random Forest avec ACP
pipe_rf_pca = Pipeline([('pca', PCA(0.98)),
                         ('clf', BaggingClassifier(RandomForestClassifier()))])

########################## AVEC Réduction de dimension (LASSO)␣
↪#####################################

# SVM avec Lasso
pipe_svm_lasso = Pipeline([('feature_selection',␣
↪SelectFromModel(LogisticRegression(C=1,penalty='l1',solver='liblinear'))),
                         ('clf', BaggingClassifier(SVC(random_state=42)))])

# XGBoost avec Lasso
pipe_xgb_lasso = Pipeline([('feature_selection',␣
↪SelectFromModel(LogisticRegression(C=1,penalty='l1',solver='liblinear'))),
                         ('clf',␣
↪BaggingClassifier(XGBClassifier(learning_rate=0.02, n_estimators=600,␣
↪objective='binary:logistic', nthread=1)))])

# Random Forest avec Lasso
```

```
pipe_rf_lasso = Pipeline([('feature_selection',␣
 ↪SelectFromModel(LogisticRegression(C=1,penalty='l1',solver='liblinear'))),
                          ('clf', BaggingClassifier(RandomForestClassifier()))])




########################## Régression Logistique pénalisée␣
 ↪#####################################

# Régression Logistique pénalisée
pipe_lr = Pipeline([('clf',␣
 ↪BaggingClassifier(LogisticRegression(random_state=42)))])
```

[66]:
```
# grid_params_lr_initial = '{'clf__penalty': ['l1', 'l2','elasticnet'],'clf__C':
 ↪ [1.0, 0.5, 0.1], 'clf__solver': ['liblinear','saga']}]
# 'penalty' compare le lasso (l1), le ridge(l2) et l'elasticnet
# 'solver' est l'algorithme pour l'optimisation, saga est le seul utilisé pour␣
 ↪l'elastic net.

grid_params_lr = [{'clf__penalty': ['l1','l2','elasticnet'],
'clf__C': [0.05,0.1,0.15], # le degré de pénalisation
'clf__solver': ['liblinear','saga'],
'clf__class_weight':['balanced']}]

# grid_params_svm_initial = ''clf__C': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],␣
 ↪clf__gamma':[0.00005,0.0001,0.001]}].
# Après avoir utilisé grid_params_svm_initial, je réduit les choix des␣
 ↪paramètres afin de raccourcir le temps de calcul
grid_params_svm = [{'clf__kernel': ['linear', 'rbf'],
'clf__C': [1,2],
'clf__gamma':[0.00005],
'clf__class_weight':['balanced']}]

# ATTENTION TRÈS LONG
# grid_params_xgb_initial = [ { 'clf__min_child_weight': [1, 5, 10],␣
 ↪'clf__gamma': [0.5, 1, 1.5, 2, 5], 'clf__subsample': [0.6, 0.8, 1.
 ↪0],'clf__colsample_bytree': [0.6, 0.8, 1.0],'clf__max_depth': [3, 4, 5]}]
# Après avoir utilisé grid_params_xgb_initial, je réduit les choix des␣
 ↪paramètres afin de raccourcir le temps de calcul:
grid_params_xgb=[{'clf__colsample_bytree': [0.8,1.0],
                  'clf__gamma': [2,2.5],
                  'clf__max_depth': [2,3],
                  'clf__class_weight':['balanced']}]
```

```
# grid_params_rf_initial = '[{ 'clf__bootstrap': [True],'clf__max_depth':␣
 ↪[60,80, 90, 100, 110],'clf__max_features':␣
 ↪[10,12,14],'clf__min_samples_leaf': [2,3,4],'clf__min_samples_split':␣
 ↪[3,4,5],'clf__n_estimators': [20,100,200]}]
# Après avoir utilisé grid_params_rf_initial, je réduit les choix des␣
 ↪paramètres afin de raccourcir le temps de calcul
grid_params_rf = [{
    'clf__bootstrap': [False],
    'clf__max_depth': [600],
    'clf__max_features': [40],
    'clf__min_samples_leaf': [5,6,8,10,15],
    'clf__min_samples_split': [5],
    'clf__n_estimators': [22],
    'clf__class_weight':['balanced','balanced_subsample']}]
```

[67]:
```
# Fit the grid search objects
print('On débute...')
print('Nous utilisons les mesures de performances suivantes: ')
print("   - Accuracy score (f1 score), calculé sur les données test.")
print("   - Le cross validation score, calculé à partir des données␣
 ↪d'entraînement. On aurait pu utiliser le accuracy score sur données train␣
 ↪sans validation croisée, mais c'est plus biasé que celui calculé par␣
 ↪validation croisée.")
print("   - Les matrices de confusion qui affichent la répartition des classes.
 ↪")
print("Nous cherchons de plus à trouver le bon équilibre entre la précision et␣
 ↪la sensibilité (recall ou taux de vrai positif). Pour nous aider nous nous␣
 ↪baserons sur le f1 score qui est la moyenne harmonique des scores de␣
 ↪précision et de recall. Nous nous baserons également sur les matrices de␣
 ↪confusion afin d'établir l'équilibre des classes.")

best_acc = 0.0
best_clf = 0
best_gs = ''

for idx, gs in enumerate(grids):
    print('\nEstimateur: %s' % grid_dict[idx])
    # Fit sur les données train, permet de trouver l'estimateur (le classifieur␣
 ↪estimé)
    # La validation croisée est faite sur les données train et nous commençons␣
 ↪par calculer le cross validation score à partir des données train.
    # On s'assure de ne pas 'toucher' aux données tests
    gs.fit(Xf_train, yf_train)
    print('Les meilleurs paramètres sont : %s' % gs.best_params_)
    print('Le cross validation score sur données train: %.3f' % gs.best_score_)
```

```
    # Predict sur les données test
    # On ne fait que des prédictions à partir de Xf_test
    y_pred = gs.predict(Xf_test)

    # Le accuracy score est obtenu à partir des données test.
    print('Accuracy score sur données test pour les meilleurs paramètres: %.3f␣
↪' % accuracy_score(yf_test, y_pred))
    print(confusion_matrix(yf_test,y_pred))

    print(classification_report(yf_test, y_pred))
    cm=confusion_matrix(yf_test,y_pred)
    # On veut sortir le classifieur ayant le meilleur accuracy score (1-erreur␣
↪de classification minimum)
    if accuracy_score(yf_test, y_pred) > best_acc:
        best_acc = accuracy_score(yf_test, y_pred)
        best_gs = gs
        best_clf = idx
        best_yf=y_pred
```

On débute…
Nous utilisons les mesures de performances suivantes:
- Accuracy score (f1 score), calculé sur les données test.
- Le cross validation score, calculé à partir des données d'entraînement. On aurait pu utiliser le accuracy score sur données train sans validation croisée, mais c'est plus biasé que celui calculé par validation croisée.
- Les matrices de confusion qui affichent la répartition des classes.

Nous cherchons de plus à trouver le bon équilibre entre la précision et la sensibilité (recall ou taux de vrai positif). Pour nous aider nous nous baserons sur le f1 score qui est la moyenne harmonique des scores de précision et de recall. Nous nous baserons également sur les matrices de confusion afin d'établir l'équilibre des classes.

```
Estimateur: Support Vector Machine sans réduction de dimensions
Les meilleurs paramètres sont : {'clf__C': 1, 'clf__gamma': 5e-05,
'clf__kernel': 'linear'}
Le cross validation score sur données train: 0.816
Accuracy score sur données test pour les meilleurs paramètres: 0.883
[[75  3]
 [ 8  8]]
              precision    recall  f1-score   support

           0       0.90      0.96      0.93        78
           1       0.73      0.50      0.59        16

    accuracy                           0.88        94
   macro avg       0.82      0.73      0.76        94
```

```
weighted avg       0.87      0.88      0.87        94


Estimateur: Logistic Regression pénalisé
Les meilleurs paramètres sont : {'clf__C': 0.1, 'clf__penalty': 'l1',
'clf__solver': 'liblinear'}
Le cross validation score sur données train: 0.828
Accuracy score sur données test pour les meilleurs paramètres: 0.830
[[73  5]
 [11  5]]
              precision    recall  f1-score   support

           0       0.87      0.94      0.90        78
           1       0.50      0.31      0.38        16

    accuracy                           0.83        94
   macro avg       0.68      0.62      0.64        94
weighted avg       0.81      0.83      0.81        94


Estimateur: Support Vector Machine avec Réduction: PCA
Les meilleurs paramètres sont : {'clf__C': 2, 'clf__gamma': 5e-05,
'clf__kernel': 'rbf'}
Le cross validation score sur données train: 0.833
Accuracy score sur données test pour les meilleurs paramètres: 0.840
[[70  8]
 [ 7  9]]
              precision    recall  f1-score   support

           0       0.91      0.90      0.90        78
           1       0.53      0.56      0.55        16

    accuracy                           0.84        94
   macro avg       0.72      0.73      0.72        94
weighted avg       0.84      0.84      0.84        94


Estimateur: XGBoost avec Réduction: PCA
Les meilleurs paramètres sont : {'clf__colsample_bytree': 1.0, 'clf__gamma': 2,
'clf__max_depth': 2}
Le cross validation score sur données train: 0.782
Accuracy score sur données test pour les meilleurs paramètres: 0.840
[[78  0]
 [15  1]]
              precision    recall  f1-score   support

           0       0.84      1.00      0.91        78
           1       1.00      0.06      0.12        16
```

```
    accuracy                           0.84        94
   macro avg       0.92      0.53      0.51        94
weighted avg       0.87      0.84      0.78        94


Estimateur: Random Forest avec Réduction: PCA
Les meilleurs paramètres sont : {'clf__bootstrap': False, 'clf__max_depth': 600,
'clf__max_features': 40, 'clf__min_samples_leaf': 6, 'clf__min_samples_split':
5, 'clf__n_estimators': 22}
Le cross validation score sur données train: 0.805
Accuracy score sur données test pour les meilleurs paramètres: 0.830
[[77  1]
 [15  1]]
              precision    recall  f1-score   support

           0       0.84      0.99      0.91        78
           1       0.50      0.06      0.11        16

    accuracy                           0.83        94
   macro avg       0.67      0.52      0.51        94
weighted avg       0.78      0.83      0.77        94


Estimateur: Support Vector Machine avec Réduction: Lasso Logistique
Les meilleurs paramètres sont : {'clf__C': 2, 'clf__gamma': 5e-05,
'clf__kernel': 'linear'}
Le cross validation score sur données train: 0.822
Accuracy score sur données test pour les meilleurs paramètres: 0.872
[[73  5]
 [ 7  9]]
              precision    recall  f1-score   support

           0       0.91      0.94      0.92        78
           1       0.64      0.56      0.60        16

    accuracy                           0.87        94
   macro avg       0.78      0.75      0.76        94
weighted avg       0.87      0.87      0.87        94


Estimateur: XGBoost avec Réduction: Lasso Logistique
Les meilleurs paramètres sont : {'clf__colsample_bytree': 0.8, 'clf__gamma':
2.5, 'clf__max_depth': 3}
Le cross validation score sur données train: 0.787
Accuracy score sur données test pour les meilleurs paramètres: 0.809
[[74  4]
 [14  2]]
```

```
              precision    recall  f1-score   support

           0       0.84      0.95      0.89        78
           1       0.33      0.12      0.18        16

    accuracy                           0.81        94
   macro avg       0.59      0.54      0.54        94
weighted avg       0.75      0.81      0.77        94
```

```
Estimateur: Random Forest avec Réduction: Lasso Logistique
Les meilleurs paramètres sont : {'clf__bootstrap': False, 'clf__max_depth': 600,
'clf__max_features': 40, 'clf__min_samples_leaf': 15, 'clf__min_samples_split':
5, 'clf__n_estimators': 22}
Le cross validation score sur données train: 0.782
Accuracy score sur données test pour les meilleurs paramètres: 0.819
[[75  3]
 [14  2]]
              precision    recall  f1-score   support

           0       0.84      0.96      0.90        78
           1       0.40      0.12      0.19        16

    accuracy                           0.82        94
   macro avg       0.62      0.54      0.54        94
weighted avg       0.77      0.82      0.78        94
```

Nous remarquons une légère amélioration en ajoutant le bagging. En effet, si on regarde le SVM avec Lasso, on voit que la classe est un peu mieux réparti. On a un peu moins de faux négatifs. Toutefois il existe encore un déséquilibre dans les classes. Il serait intéressant de tester des techniques comme le SMOT afin d'améliorer cet équilibre.

3 FIN