Projet - Exercice 2 :
De la régression à la classification

Prédire une réponse au traitement du cancer

Documents et jupyterNB au Github :
https://github.com/kjousselin/M2_Stats_Gde_dim_projet2_classification_regression.git

Enoncé de l'exercice :
Dans ce second exercice, on s'intéresse à une base de données liée au cancer sein (Attention, les données sous un format pxn, i.e: variable x individu). Les bases de données sont téléchargeables au lien suivant :
https://plmbox.math.cnrs.fr/f/fedcac32b2a949198dce/

Dans cette base de données, l'objectif de prédire la réaction au traitement. La variable à prédire est "treatment_response" à partir de données génétiques et d'autres caractéristiques (âge/ethnie/Stade de la tumeur T/N).

On pourra éventuellement se contenter des données génétiques afin de simplifier le problème et probablement d'éviter de mettre sur un même plan des variables "cliniques" probablement très corrélées avec la réponse et des variables génétiques dont la complexité nécessite des méthodes plus complexes.

On propose ici de comparer plusieurs méthodes : Régression sur composantes principales (non recommandé a priori mais ça peut se tester !), Régression PLS, LASSO, Sparse PCA+ Régression et éventuellement un algorithme de type complètement différent pour terminer (probablement plus performant).

Une approche "active" sera bien sûr appréciée (par "active", on sous-entend que vous pouvez prendre des initiatives pour améliorer les méthodes ci-dessus ou sur la manière de mesurer les résultats, pas uniquement en "accuracy" par exemple).

Voici la démarche que nous avons adopté pour répondre au problème :

I – Comprendre et traiter les données

Dans un premier temps, j'ai crée une fonction **process_data()** qui m'a permis d'importer les données, et d'obtenir un set d'entrainement et de validation. Avec en option la possibilité en autres :

- De centrer-réduire les données,
- De choisir la taille du set d'entrainement,
- de prendre en compte (ou non) les données cliniques (argument *clinique*).

La variable à prédir **'treatment_response'** est également convertie en 0 ou 1. Dans la suite, on appelera « individu malade » ou un individu dont **'treatment_response'** est à 0.

La fonction **process_data()** permet aussi d'afficher quelques informations sur les données (argument **display**).

```
def process_data(file, dir = "./", clinique = True, display = True, test_size = 0.35, centre_red = True):
    """
    file :       str         nom du fichier
    dir :        str         dossier contenant le fichier
    clinique :   bool        Prendre en compte et traiter les données cliniques (ou non)
    display :    bool        Afficher des informations sur le stdout
    test_size : float        pourcentage des données servant à créer un jeu de test.
    centre_red : bool        Centrer et réduire les données

    Sortie :
    X_train, X_test, y_train, y_test
    """
```

Voici un aperçu du df X_train obtenu :

	1007_s_at	1053_at	117_at	121_at	1255_g_at	1294_at	1316_at	1320_at	1405_i_at	1431_at	...	AFFX-r2-Hs28SrRNA-5_at	AFFX-r2-Hs28SrRNA-M_at	AFFX-r2-P1-cre-3_at	AFFX-r2-P1-cre-5_at	AFFX-ThrX-3_at	AFFX-ThrX-5_at	AF Th M
0	-0.375263	0.355374	0.320139	-0.449468	0.033397	-1.054221	-0.167410	0.435017	0.191654	-0.551015	...	0.117428	-0.576495	-0.342313	-0.429296	-0.599425	0.124140	-1.655
1	-1.362700	0.404197	-0.998941	-2.481996	-2.614741	-1.863933	-1.104999	-2.690549	1.236845	-0.739771	...	0.704679	-1.029702	-0.136013	-0.278871	0.117435	0.506715	-1.803
2	-1.166847	0.212096	-3.168177	-2.611532	0.279191	-0.385342	-2.004847	-2.297611	1.349379	-1.697005	...	-0.320620	-0.160792	-0.371105	-0.659696	-0.849123	0.236177	0.310
3	0.634166	0.150025	-0.299050	-0.427094	0.035070	-0.064986	-0.285499	1.006642	0.150935	-2.980045	...	0.165597	0.398648	-0.782548	-1.066343	-0.635399	0.004884	-1.346
4	0.904319	-0.526623	-0.375820	-0.353298	-0.219166	1.079651	0.421965	-0.857601	0.558508	1.004381	...	-0.242217	-0.993866	-0.604912	-0.282256	-0.614077	0.102485	0.362
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
161	0.057456	-0.546986	0.089172	-0.211594	-1.789348	0.318826	0.502669	-1.328859	0.316605	0.038800	...	-0.731167	0.121677	-0.116219	-0.094977	-0.319613	-0.291680	-1.018
162	1.387264	0.472279	0.536146	0.349857	0.290183	-0.246451	-0.025941	-1.517425	0.419015	-0.039568	...	-0.468970	-0.857393	-0.555683	-0.223340	-0.170158	-0.304999	-2.118
163	1.192600	-0.276376	-0.349174	-0.132565	0.309059	-0.672675	-0.301640	0.236177	-1.701012	0.562581	...	-0.507830	0.464135	-0.392314	-0.470662	-0.261104	-0.465329	-0.776
164	-1.806415	1.057660	-0.996866	-2.458183	-2.229244	-1.280202	-1.358267	-2.137575	-1.074076	-0.824617	...	-0.256907	-0.047883	-0.222004	-0.286518	-0.666119	0.433921	-0.496
165	0.624953	-0.433762	-0.188753	-0.083236	0.113444	0.757029	0.012749	-0.940899	1.313192	0.669107	...	-1.478298	0.415044	-0.491158	-0.842838	1.350366	0.000817	-0.710

166 rows × 22283 columns

Remarque : Dans la suite, nos modèles seront basés sur les données génétiques uniquement *(clinique = False)*.

II – De la régression à la classification et mesure de performance

Règle de décision et seuil :

Dans la section suivante (III), nous allons tester différentes méthodes de régression sur les données afin de prédir la variable **'treatment_response'**. La régression renvoie une valeur réelle, alors que notre variable à prédir est une donnée binaire (0 ou 1).

Nous avons donc besoin d'une **règle de décision** dépendant d'un **seuil** α :

$$f_\alpha : \begin{array}{ccc} \mathbb{R} & \to & \{0,1\} \\ x & \mapsto & \begin{cases} 0 & si\ x < \alpha \\ 1 & sinon \end{cases} \end{array}$$

On comprend que lorsque α évolue de 0 vers 1, alors le nombre de valeurs prédites à 1 baisse tandis que le nombre de valeurs prédites à 0 augmente.

Matrice de confusion :

Fixons un $\alpha \in \mathbb{R}$, alors $f_\alpha(x)$ donnera une prédiction y_pred de x à 0 ou à 1. Or y_true vaut 0 ou 1. En regroupant dans un tableau les valeurs prédites à 0 ou 1 en abscisse et réellement égale à 0 ou 1 en ordonnée, on obtient **la matrice de confusion** :

réel			
	0	TN	FP
	1	FN	TP
		0	1
		Prédit / Classé	

On définit : La **sensibilité** : $Se = P(\text{classé "1"} \mid \text{"1"}) = \frac{TP}{TP+FN}$

La **spécificité** : $Sp = P(\text{classé "0"} \mid \text{"0"}) = \frac{TN}{TN+FP}$

T : True / F : False / P : Positive (classé positif) / N : Negative (classé Négatif)

Risque asymétrique/Courbe ROC :

Le but est d'obtenir le maximum de TP (taux de TP proche de 1) et le minimum de FP (taux de FP proche de 0)

Ainsi, une façon d'estimer la qualité de notre modèle est d'observer **l'évolution du taux de TP (« sensibilité ») en fonction du taux de FP (« 1 moins la spécificité »)** : la courbe **ROC (« receiver operating characteristic »)**.

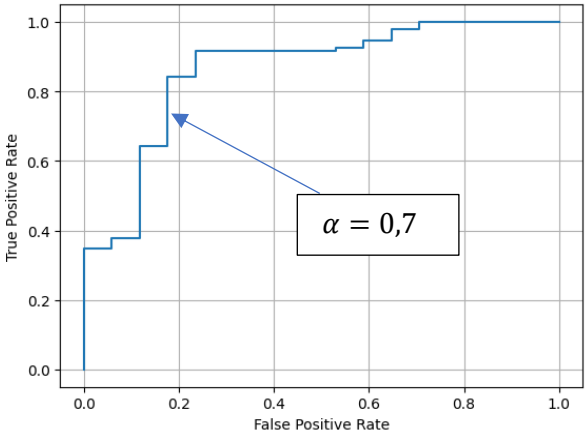
Or, lorsque α augmente, le taux de TP diminue en même temps que celui de FP.

On est donc amené à choisir une valeur de α qui maximise un des taux (taux de FP ou TP) selon nos besoins (**), ou plutôt, **trouver un α qui donne un bon compromis entre les taux**. D'une façon générale, le modèle est intéressant lorsqu'à la fois TP est élevé et FP est faible : **on cherche donc une courbe ROC la plus « en haut à gauche »**, celle qui maximise l'intégrale sur [0, 1].

(**) En général en médecine, il est usuellement plus grave de classer parmi les malades une personne saine plutôt qu'une personne saine parmi les malades, on cherche donc plutôt à maximiser le taux de FN.

Exemple de la prédiction III-a (Régression linéaire simple) :

Voici ci-contre un exemple de courbe ROC : il s'agit de celle obtenue.



On lit le taux de TP en fonction du taux de FP :

Par exemple pour un $\alpha = 0{,}7$ on a les valeurs suivantes :

$TN = 14 \quad TP = 3$

$FN = 23 \quad FP = 72$

Et les taux :

$$sensibilité = \frac{TP}{TP+FN} = \frac{72}{95} \approx 0{,}76$$

$$1 - spécificité = 1 - \frac{TN}{TN+FP} = \frac{FP}{TN+FP} = \frac{3}{17} \approx 0{,}18$$

On retrouve bien le point de coordonnées (0,18, 0,76) sur la courbe ROC.

réel			
	0	14	3
	1	23	72
		0	1
		Prédit / Classé	

Mesure du score et affichage d'une matrice de confusion : Ma fonction « Mon_score_perso() »

Afin de répondre à la problématique, je créé une fonction '**Mon_score_perso()**' qui va permettre de donner le taux de bonnes prédictions ainsi d'afficher éventuellement la matrice de confusion. Ce score sera transformé en score interprétable par la fonction GridSearchCV grâce à la fonction de sklearn 'MakeScorer' :

```
def mon_score_perso(y_true, y_pred, alpha = 0.5, display_MC = True):
    """
    y_pred      list ou np.array    valeurs prédites par un regresseur (reel)
    y_true      list ou np.array    "vraies" valeurs (binaire : 0 ou 1)
    alpha       reel dans [0, 1]    seuil de prédiction
    display_MC  bool                Affichage ou non de la matrice de confusion
    """
```

Données mal balancées :
Lorsque l'une des classes est mal représentée ou mal prédite, nous pouvons ajouter du poids à la classe sous-représentée. Nous avons choisi, pour simplifier l'étude, de ne pas tester ces solutions.

III – Régression

Nous allons tester plusieurs méthodes afin de modéliser les données :

a) Régression linéaire simple,
b) Régression linéaire sur composantes principales (après une ACP),
c) Regression PLS (Partial Least Squares)
d) Regression LASSO
e) Automatisons la démarche pour tester d'autres régressions
f) ElasticNet, Ridge, …

Pour chaque approche, nous allons :
- entrainer notre modèle sur le jeu d'entrainement
- le tester sur le jeu de test.

Nous mesurerons nos performances en comptant le taux général de bonne prédiction : $\frac{\mathrm{TP+TN}}{TP+TN+FN+FP}$ (*).

En général, nous afficherons également la matrice de confusion pour en savoir plus sur la quantité de chaque composante TP, TN, FN, FP, ainsi que la courbe ROC,

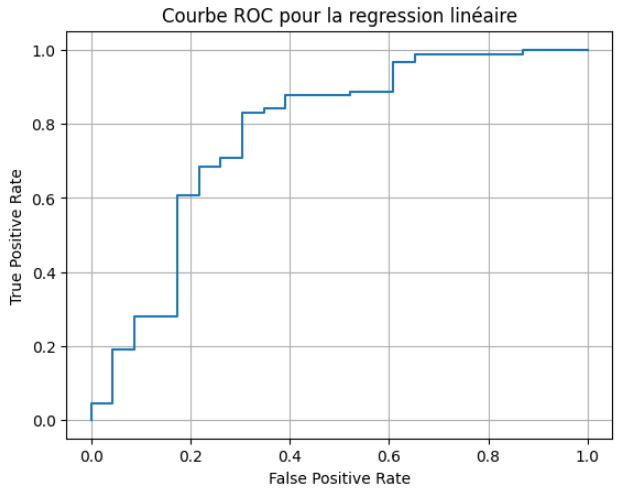
A l'aide de **sklearn**, nous pouvons utiliser les méthodes *.fit()* et *.predict()* pour construire le modèle, ajuster le modèle sur nos données d'entrainement, puis prédir les données de validation.

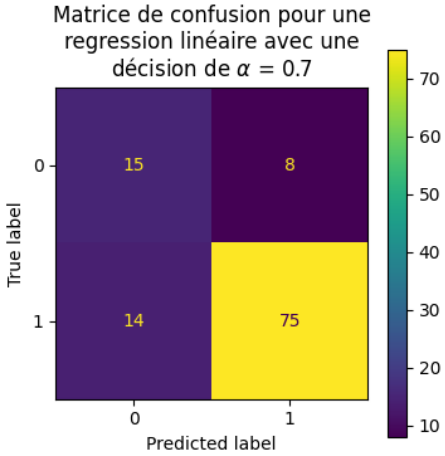
Pour nos premiers regresseurs, nous allons effectuer quelques tests à la main (a, b, c) puis nous allons utiliser **la fonction Grid_Search()** avec **notre propre score** afin de faire une recherche plus automatisée. Enfin, nous allons construire une fonction qui effectue de manière automatisée une série de test avec un retour graphique sur un regresseur quelconque.

a) Regression linéaire simple !

Commençons *pourquoi pas (***)* par tenter une régression linéaire directement ! Pour cela, nous utilisons la classe *sklearn.linear_model.LinearRegression*.

Après avoir obtenue un score d'entrainement de 100%, nous mesurons la qualité du modèle sur les données de validation et nous obtenons les résultats suivants :





« `Le taux de bonne prédiction est 80.4 %.` »

Étonnamment le taux global de bonnes prédictions n'est pas si mauvais (autour de 80 %).

Remarque 1 : Evidemment, à chaque tirage, ce taux varie, mais semble tourner autour de 78-83 %

Remarque 2 : en tenant compte des données cliniques (clinique = True), alors le taux atteint 86 % !

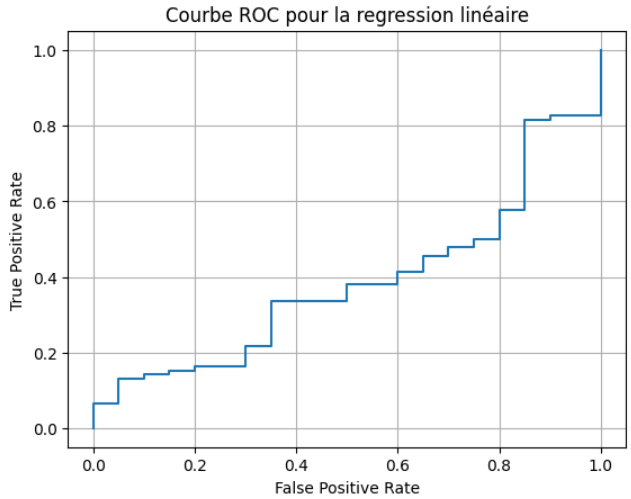
b) Régression linéaire sur composantes principales (PCR)

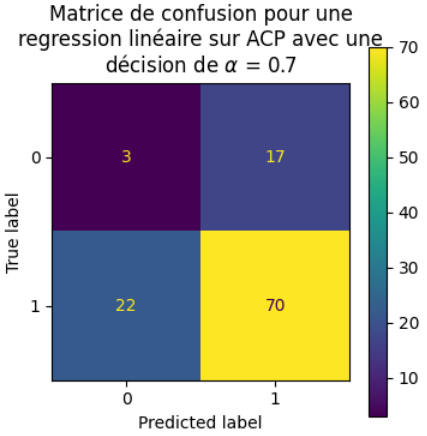
L'idée ici est de faire une régression linéaire, non pas sur l'ensemble des données, mais seulement sur des composantes principales obtenues après une ACP,

On utilise *sklearn.decomposition.PCA* pour faire une ACP sur les données d'entrée df_train.

Dans un premier temps, nous décidons de conserver 20 composantes principales.

Voici la courbe ROC obtenue, ainsi que la matrice de confusion pour un seuil α de 0.7.





Le taux global de bonne prédiction pour $\alpha = 0{,}7$ est assez faible (65%).

Testons les taux de prédictions et affichons les courbes ROC pour différents nombres de composantes principales retenues :

J'obtiens grâce à mon programme, les taux suivants :

Taux de bonnes prédictions pour différents nb de composantes principales et différentes valeurs de alpha :

Pour 1 composante(s) principale(s) :

alpha=0.6 : taux : 82.1 % //alpha=0.7 : taux : 82.1 % //alpha=0.8 : taux : 17.9 % //alpha=0.9 : taux : 17.9 % //
alpha=1.0 : taux : 17.9 % //alpha=1.1 : taux : 17.9 % //

Pour 2 composante(s) principale(s) :

alpha=0.6 : taux : 82.1 % //alpha=0.7 : taux : 82.1 % //alpha=0.8 : taux : 24.1 % //alpha=0.9 : taux : 17.9 % //
alpha=1.0 : taux : 17.9 % //alpha=1.1 : taux : 17.9 % //

Pour 3 composante(s) principale(s) :

alpha=0.6 : taux : 82.1 % //alpha=0.7 : taux : 82.1 % //alpha=0.8 : taux : 36.6 % //alpha=0.9 : taux : 19.6 % //
alpha=1.0 : taux : 17.9 % //alpha=1.1 : taux : 17.9 % //

Pour 5 composante(s) principale(s) :

alpha=0.6 : taux : 82.1 % //alpha=0.7 : taux : 73.2 % //alpha=0.8 : taux : 32.1 % //alpha=0.9 : taux : 20.5 % //
alpha=1.0 : taux : 17.9 % //alpha=1.1 : taux : 17.9 % //

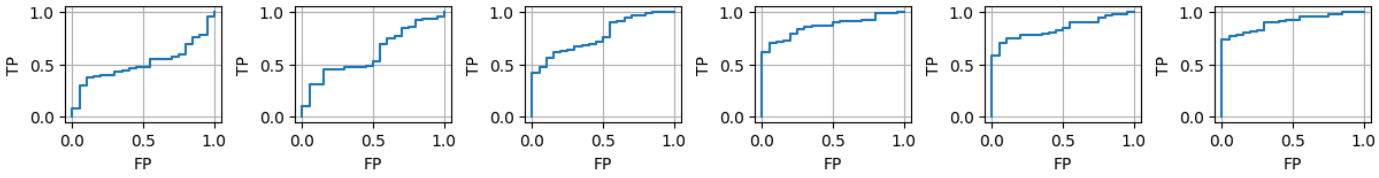
Pour 10 composante(s) principale(s) :

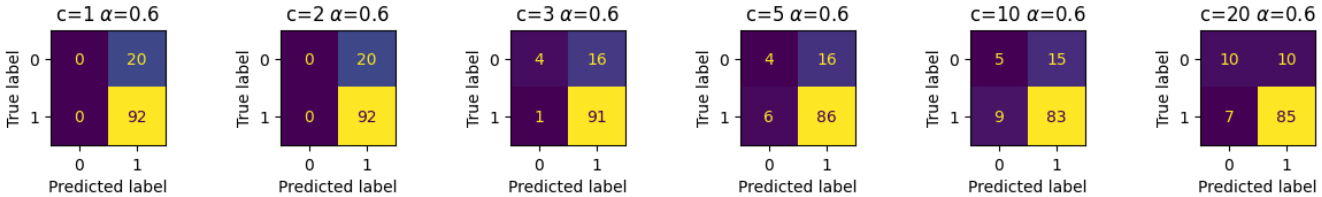
alpha=0.6 : taux : 82.1 % //alpha=0.7 : taux : 71.4 % //alpha=0.8 : taux : 36.6 % //alpha=0.9 : taux : 21.4 % //
alpha=1.0 : taux : 18.8 % //alpha=1.1 : taux : 17.9 % //

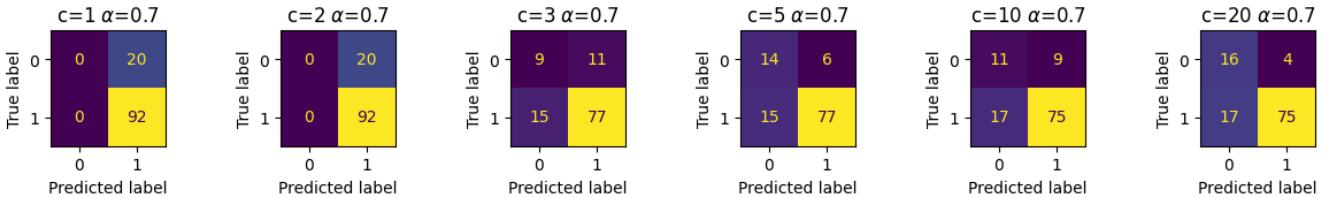
Pour 20 composante(s) principale(s) :

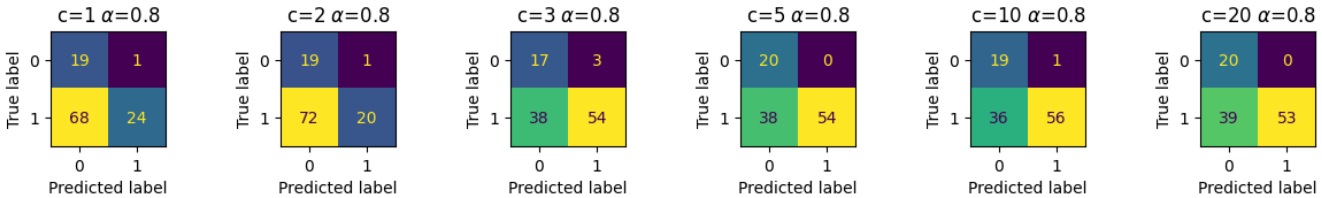
alpha=0.6 : taux : 76.8 % //alpha=0.7 : taux : 65.2 % //alpha=0.8 : taux : 41.1 % //alpha=0.9 : taux : 26.8 % //
alpha=1.0 : taux : 21.4 % //alpha=1.1 : taux : 19.6 % //

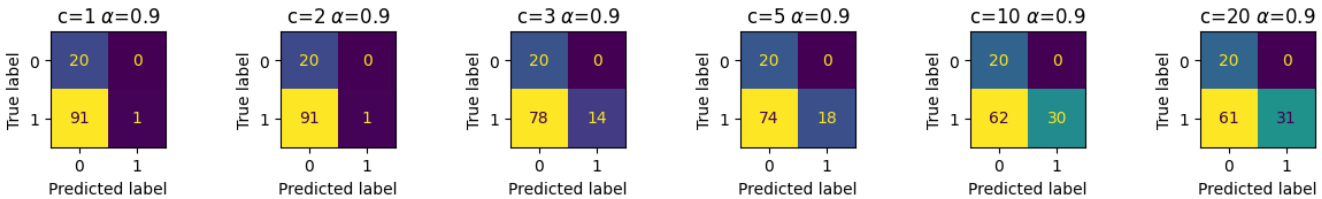
Ainsi que les courbes ROC et matrices de confusions suivantes :

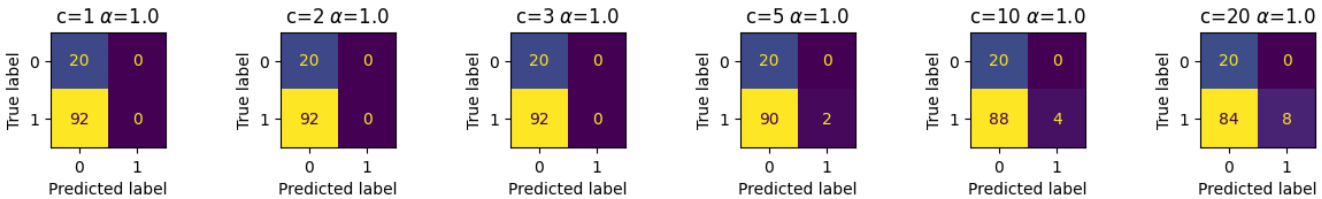


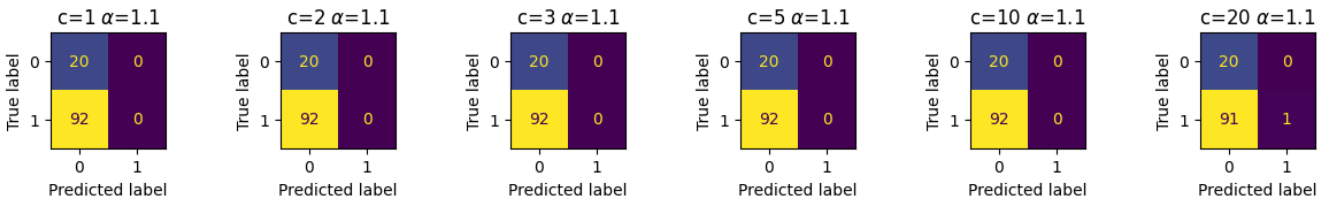






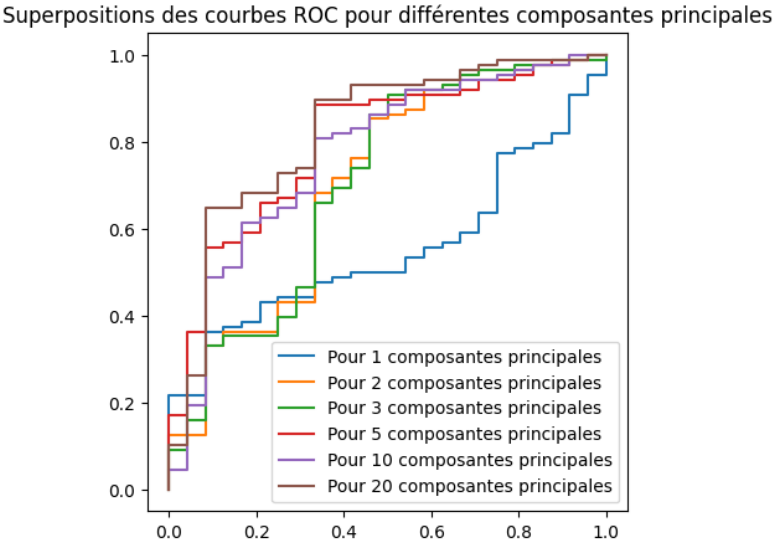






En ne conservant que 1 à 20 composantes principales, les taux de bonnes prédictions peuvent paraitres corrects dans certains cas (autour de 80%), mais l'analyse de la courbe ROC et des matrices de confusions révèle que certains modèles ne prédisent aucune réponse '0' !

Superposons les courbes ROC :



La superposition des courbes ROC confirme que la conservation de 20 composantes principales semble plus intéressante (courbe plus « en haut à gauche »).

Conclusion :

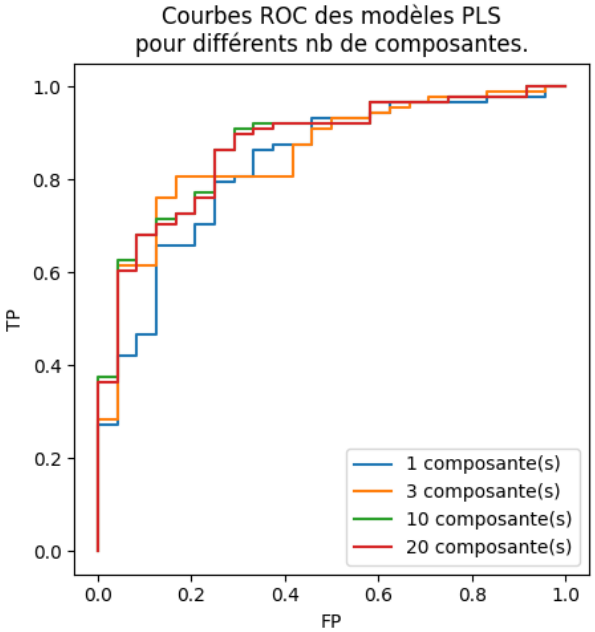
La régression sur PCA n'est pas une méthode efficace ceci est dû sans aucun doute à la très grande dimension du jeu de données. En effet, la PCA n'est pas robuste à la grande dimension.

Finalement, il y a peu de modèles intéressants parmi tous ceux utilisant une PCA dans ce contexte. Testons d'autres méthodes réputées plus efficaces en cas de grande dimensions.

c) Regression PLS (Partial Least Squares)

On retient un certain nombre de composantes,

Après avoir fait un premier test pour un alpha arbitraire, nous décidons d'observer les courbes ROC pour différents nombres de composantes retenues afin de déterminer un alpha intéressant selon nos besoins (voir courbes ci-contre).



A première vue, l'observation de ces différentes courbes ROC révèle que les méthodes ne conservant 10 ou 20 composantes semblent plus intéressantes, sans que la différence soit vraiment significative.

Une étude plus approfondie serait nécessaire pour davantage de précisions en gardant à l'esprit qu'un modèle avec moins de variables (moins de composantes) devrait être privilégié.

d) Regression LASSO

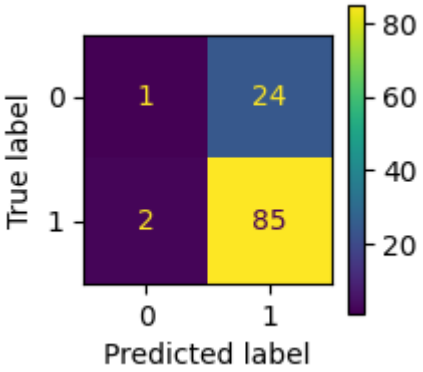
Après un premier test, nous avons voulue utiliser GridSearchCV. Pour cela, nous allons utiliser notre propre score « `mon_score_GSCV` » écrit en partie II : il a fallu écrire notre propre score afin que gridsearchCV mesure le taux de bonne prédiction :

Avec ce score, nous avons pu utiliser GridSearchCV, et obtenu les résultats suivants :

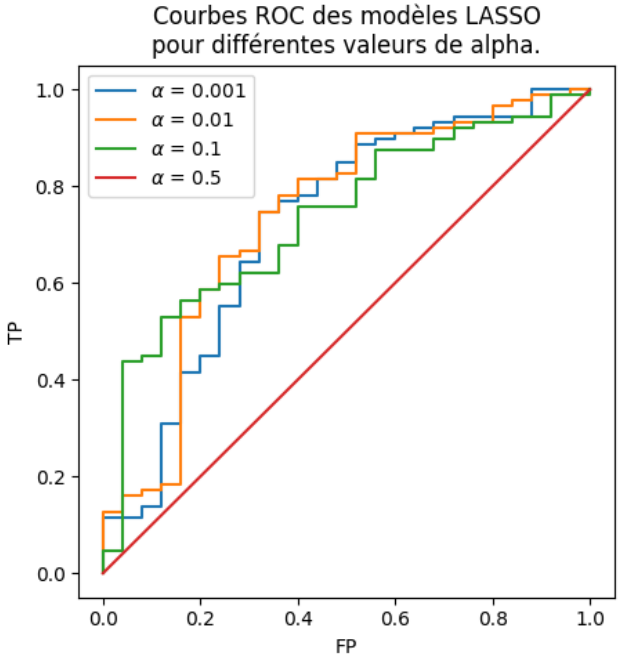
```
Meilleur score :
            0.8614973262032086
Pour une valeur de alpha correspondante de :
            {'alpha': 0.03}
```

Le « alpha » cité est le paramètre du LASSO.

Une mesure sur notre échantillon de test donne un score de test de 77 %, mais l'analyse de la matrice de confusion révèle une très mauvaise prédiction des valeurs négatives (voir ci-dessous).



Terminons l'étude de la méthode Lasso par ses courbes ROC :



e) Automatisons la démarche pour tester d'autres regressions : ma fonction « Explore_regressions() »

Pour en savoir un peu plus sur les résultats suivant les différents régresseurs et différents paramètres, je vais créer une fonction qui va afficher les taux de bonnes prédictions et les matrices de confusion pour les différents paramètres ainsi que les courbes ROC :

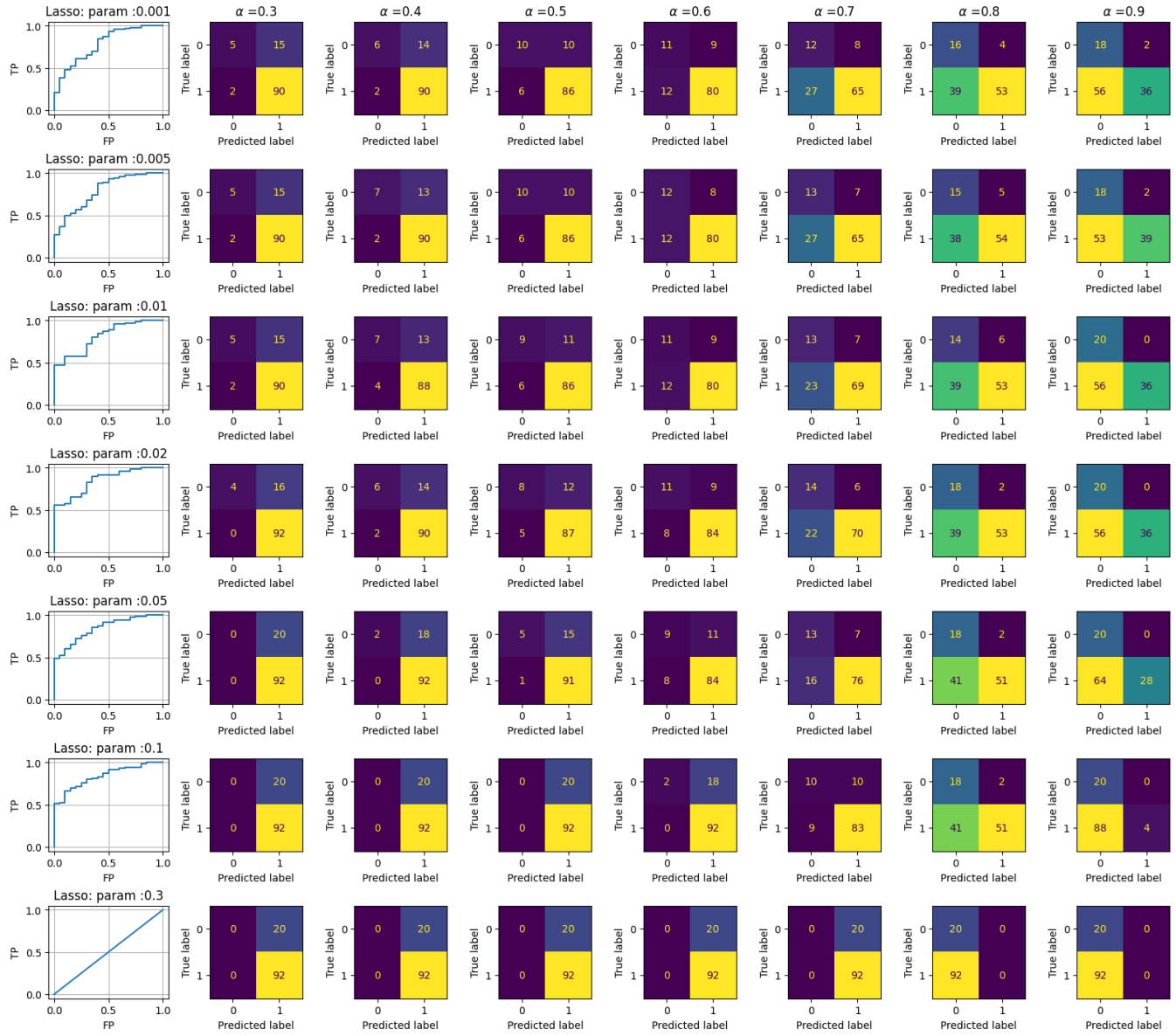
```
def Explore_regressions(clf, params, erreur_valid = True, liste_alpha_seuil = [0.3, 0.4,
0.5, 0.6, 0.7], display_print = True, display_CM=True):
    """
    clf :                       regresseur ou classifieur sklearn
    params :            dict    dictionnaire contenant le paramètre (Pour l'instant un seul
                                   paramètre est géré)
    erreur_valid :      bool    if True : calcul de l'erreur de valid. if False calcul de l'erreur
                                   d'entrainement.
    liste_alpha_seuil : list    liste des seuil alpha à tester et afficher
    display_print :     bool    Affiche sur le std_out les résultats de score
    display_CM :        bool    Affiche toutes les matrices de confusion
    """
```

Puis je l'exécute sur un LASSO avec les paramètres suivants :

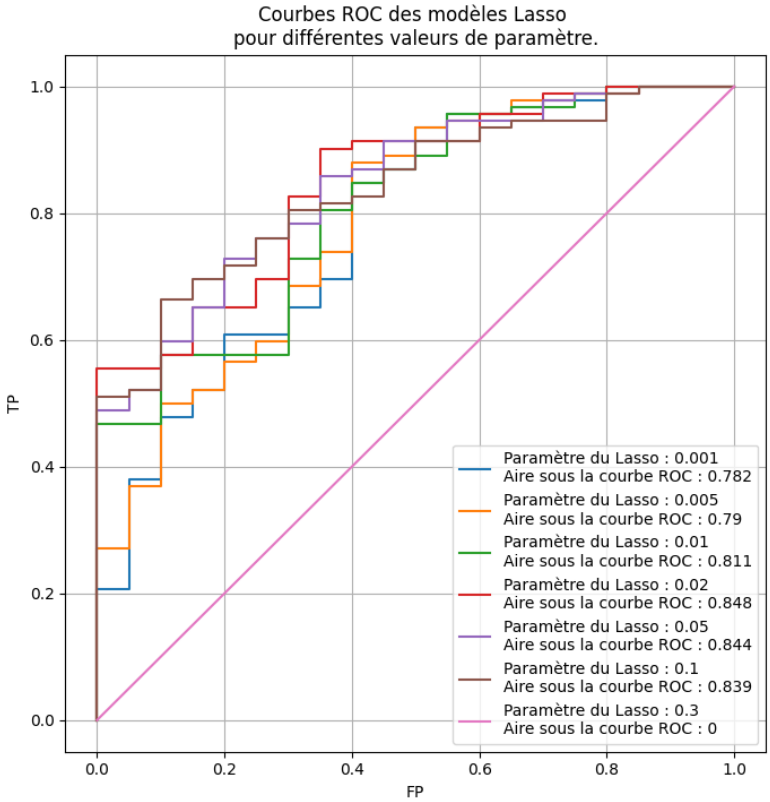
```
ma_list_lasso_param = [0.001, 0.005, 0.01, 0.02, 0.05, 0.1, 0.3]
clf = Lasso
params = {'alpha':ma_list_lasso_param}
liste_alpha_ROC = [0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
meilleur_parametre = Explore_regressions(
                        clf,
                        params,
                        erreur_valid = True,
                        liste_alpha_seuil = liste_alpha_seuil,
                        display_print = True,
                        )
```

J'obtiens les résultats suivants :

```
Taux de bonnes prédictions pour différents paramètres et différentes valeurs de seuil alpha, test sur le jeu de validation :

Pour param = 0.001 :
                    alpha=0.3 : taux : 84.8 % // alpha=0.4 : taux : 85.7 % // alpha=0.5 : taux : 85.7 % // alpha=0.6 : taux : 81.2 % //
alpha=0.7 : taux : 68.8 % // alpha=0.8 : taux : 61.6 % // alpha=0.9 : taux : 48.2 % //

Pour param = 0.005 :
                    alpha=0.3 : taux : 84.8 % // alpha=0.4 : taux : 86.6 % // alpha=0.5 : taux : 85.7 % // alpha=0.6 : taux : 82.1 % //
alpha=0.7 : taux : 69.6 % // alpha=0.8 : taux : 61.6 % // alpha=0.9 : taux : 50.9 % //

Pour param = 0.01 :
                    alpha=0.3 : taux : 84.8 % // alpha=0.4 : taux : 84.8 % // alpha=0.5 : taux : 84.8 % // alpha=0.6 : taux : 81.2 % //
alpha=0.7 : taux : 73.2 % // alpha=0.8 : taux : 59.8 % // alpha=0.9 : taux : 50.0 % //

Pour param = 0.02 :
                    alpha=0.3 : taux : 85.7 % // alpha=0.4 : taux : 85.7 % // alpha=0.5 : taux : 84.8 % // alpha=0.6 : taux : 84.8 % //
alpha=0.7 : taux : 75.0 % // alpha=0.8 : taux : 63.4 % // alpha=0.9 : taux : 50.0 % //

Pour param = 0.05 :
                    alpha=0.3 : taux : 82.1 % // alpha=0.4 : taux : 83.9 % // alpha=0.5 : taux : 85.7 % // alpha=0.6 : taux : 83.0 % //
alpha=0.7 : taux : 79.5 % // alpha=0.8 : taux : 61.6 % // alpha=0.9 : taux : 42.9 % //

Pour param = 0.1 :
                    alpha=0.3 : taux : 82.1 % // alpha=0.4 : taux : 82.1 % // alpha=0.5 : taux : 82.1 % // alpha=0.6 : taux : 83.9 % //
alpha=0.7 : taux : 83.0 % // alpha=0.8 : taux : 61.6 % // alpha=0.9 : taux : 21.4 % //

Pour param = 0.3 :
                    alpha=0.3 : taux : 82.1 % // alpha=0.4 : taux : 82.1 % // alpha=0.5 : taux : 82.1 % // alpha=0.6 : taux : 82.1 % //
alpha=0.7 : taux : 82.1 % // alpha=0.8 : taux : 17.9 % // alpha=0.9 : taux : 17.9 % //
```




```
Au vu de l'aire sous la courbe ROC,
le meilleur paramètre est : 0.02
```



En examinant les résultats et notamment l'aire sous la courbe ROC, le régresseur LASSO semble globalement plus intéressant pour le paramètre 0.1.

Mais la question de la **sensibilité** (taux de TP) et de la **spécificité** (taux de TN : $\frac{TN}{N} = 1 - \frac{FP}{N}$) nous invitera peut-être à choisir un autre paramètre.

Conclusion :

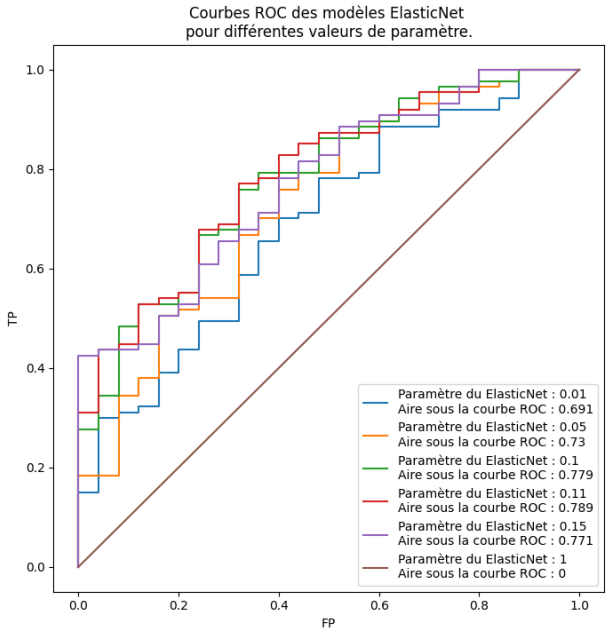
D'une façon générale, les courbes ROC ainsi que les matrices de confusion nous permettent de faire un choix éclairé sur la valeur des paramètres à choisir (paramètres des regresseur et du seuil α) selon nos besoins.

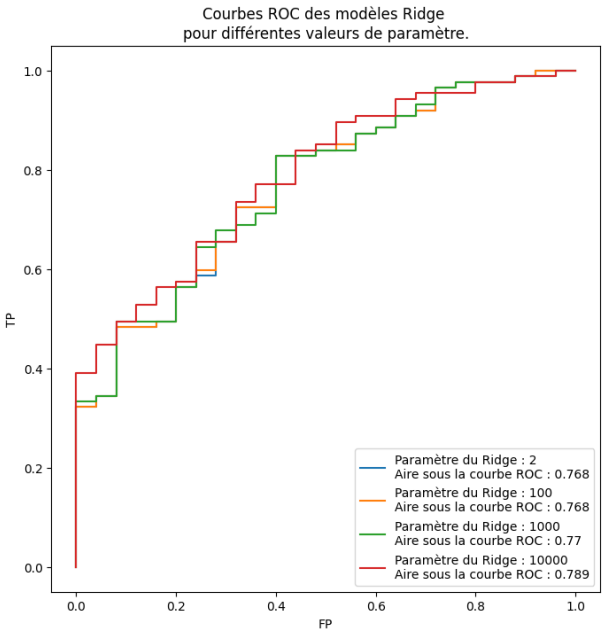
f) ElasticNet, Ridge, …

Grâce à la fonctions précédente, je peux tester une multitude de régresseurs :

```
from sklearn.linear_model import ElasticNet
from sklearn.linear_model import Ridge

clfs = [ElasticNet, Ridge]
params_S = [ {'alpha':[0.01, 0.05, 0.1, 0.11, 0.15, 1]}, {'alpha':[2, 100, 1000, 10000]}]  #[0.0001,
0.01, 1]}]
liste_alpha_seuil = [0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]

liste_scores = []

for k, clf in enumerate(clfs):
    print(f"\n\tMéthode : {clf.__name__} avec les paramètres : {params_S[k]}")
    result = meilleur_parametre = Explore_regressions(
                        clf,
                        params = params_S[k],
                        erreur_valid = True,
                        liste_alpha_seuil = liste_alpha_seuil,
                        display_print = False,
                        display_CM = False
                        )
    liste_scores.append(result)

meilleur = max(liste_scores)
print(f"Selon la courbe ROC, le meilleur classifieur est {meilleur[2].__name__}, avec le paramètre
{meilleur[1]}.")
```

Et en affichant les courbes ROC uniquement (display_print = False, display_CM = False), on obtient :






```
Méthode : ElasticNet avec les paramètres : {'alpha': [0.01, 0.05, 0.1, 0.11, 0.15, 1]}
Méthode : Ridge avec les paramètres : {'alpha': [0.01, 0.5, 2, 100, 1000, 10000]}

Selon la courbe ROC, le meilleur classifieur est Ridge, avec le paramètre 10000.
```

Remarque : Bizarrement, des petites valeurs de pénalisation Ridge ne modifie pas la courbe ROC, ces résultats me paraissent aberrants.

IV – Testons avec des méthodes de classification

Pour tester différents classifieurs, nous allons afficher les matrices de confusions pour différents paramètres. Pour cela, nous allons créer la fonction Explore_classifieurs() :

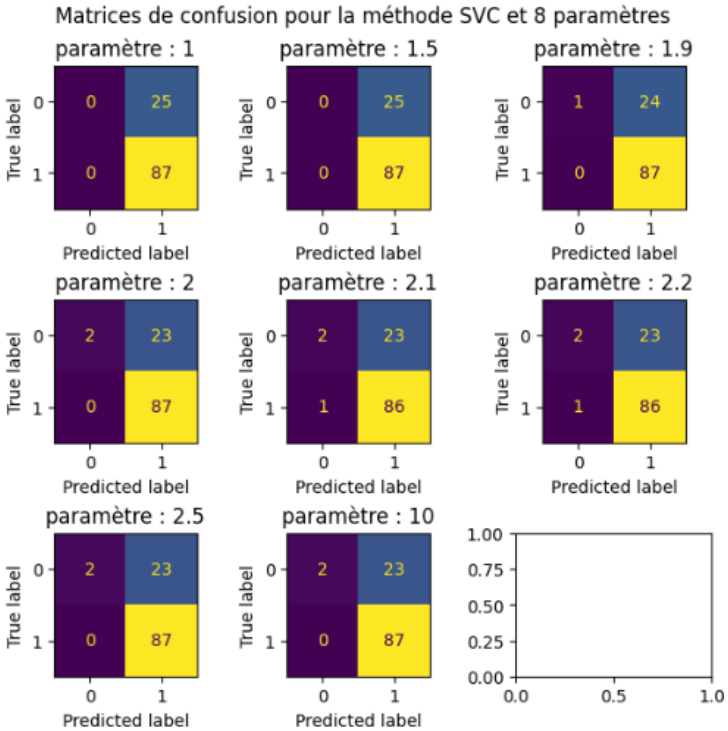
```
def Explore_classifieurs(clf, params, erreur_valid = True, display_print = True):
    """
    clf :      classifieur https://scikit-learn.org/stable/tutorial/machine_learning_map/index.html
    params :        dict     dictionnaire contenant le paramètre (Pour l'instant un seul paramètre
                             est géré)
    erreur_valid :  bool     if True : calcul de l'erreur de validation. if False calcul de l'erreur
                             d'entrainement.
    display_print : bool     Affiche sur le std_out les résultats de score
    """
```

Puis on exécute une boucle sur les classifieurs SVC, SGDCClassifier avec une liste de paramètres :

```
from sklearn.svm import SVC
from sklearn.linear_model import SGDClassifier
from sklearn.naive_bayes import GaussianNB

clfs_classif = [SVC, SGDClassifier]
params_S = [   {'C':[1, 1.5, 1.9, 2, 2.1, 2.2, 2.5, 10]},
               {'alpha':[0.0001, 0.001, 0.01, 0.1, 1, 1.2, 1.4, 1.6, 1.8, 1.9, 2, 2.1, 2.2, 2.4, 2.5,
2.6, 2.8, 10, 100, 1000]}    ]

for k, clf in enumerate(clfs_classif):
    print(f"Méthode : {clf.__name__}")

    result = Explore_classifieurs(         clf,
                                           params = params_S[k],
                                           erreur_valid = True,
                                           display_print = True,        )

    print(f"Selon le taux de bonne prédiction, le meilleur paramètre pour le classifieur
{clf.__name__} est {result[1]}.")
```

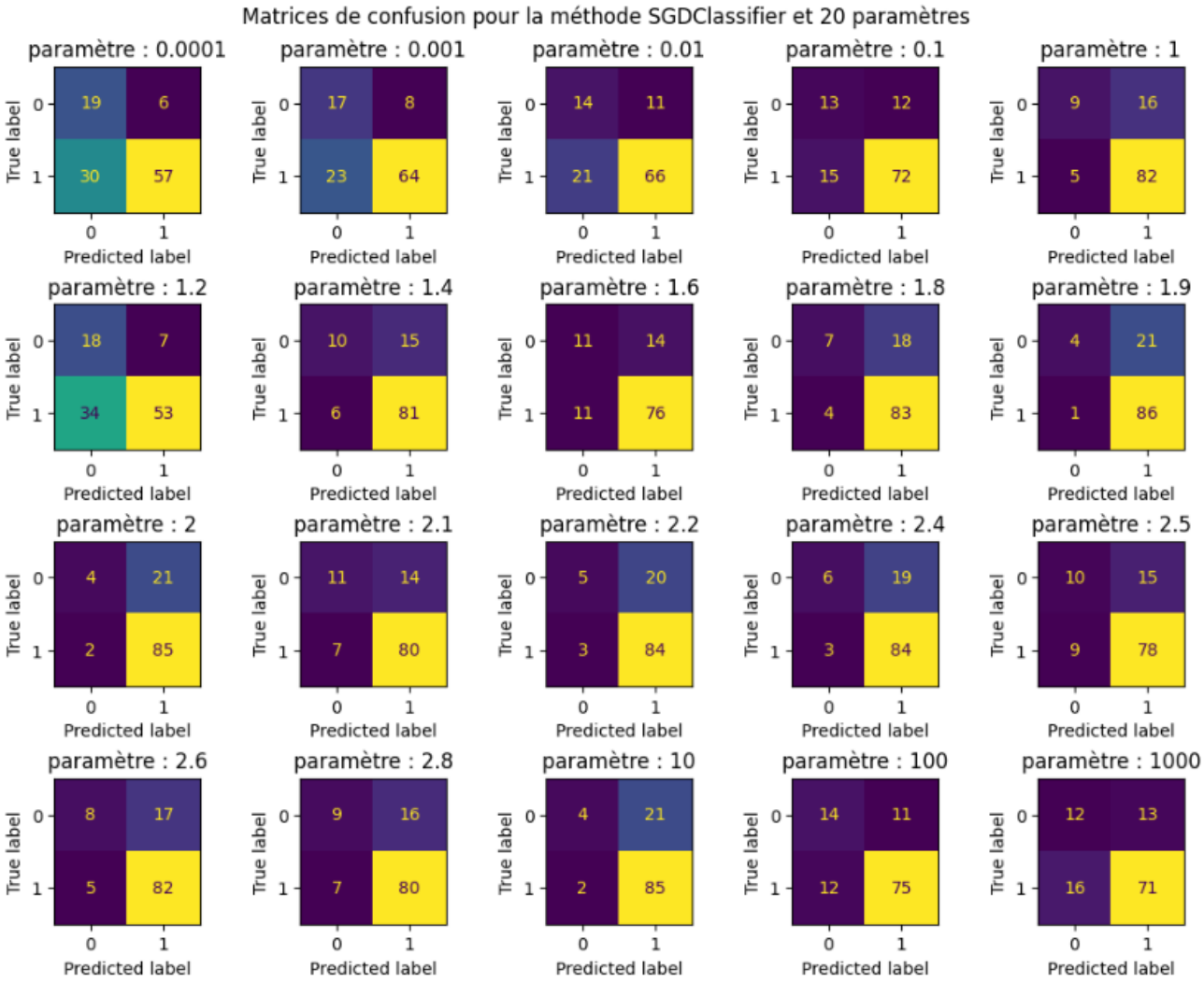
et on obtient les résultats suivants :

```
Méthode : SVC
Erreur de validation :
Le taux de bonne prédiction pour C = 1 est  77.7 %.
Le taux de bonne prédiction pour C = 1.5 est  77.7 %.
Le taux de bonne prédiction pour C = 1.9 est  78.6 %.
Le taux de bonne prédiction pour C = 2 est  79.5 %.
Le taux de bonne prédiction pour C = 2.1 est  78.6 %.
Le taux de bonne prédiction pour C = 2.2 est  78.6 %.
Le taux de bonne prédiction pour C = 2.5 est  79.5 %.
Le taux de bonne prédiction pour C = 10 est  79.5 %.
```




```
Selon le taux de bonne prédiction, le meilleur paramètre pour le classifieur SVC est 10.
```

```
Méthode : SGDClassifier
Erreur de validation :
Le taux de bonne prédiction pour alpha = 0.0 est  67.9 %.
Le taux de bonne prédiction pour alpha = 0.0 est  72.3 %.
Le taux de bonne prédiction pour alpha = 0.01 est  71.4 %.
Le taux de bonne prédiction pour alpha = 0.1 est  75.9 %.
Le taux de bonne prédiction pour alpha = 1 est  81.2 %.
…
Le taux de bonne prédiction pour alpha = 2 est  79.5 %.
Le taux de bonne prédiction pour alpha = 2.1 est  81.2 %.
Le taux de bonne prédiction pour alpha = 2.2 est  79.5 %.
Le taux de bonne prédiction pour alpha = 2.4 est  80.4 %.
Le taux de bonne prédiction pour alpha = 2.5 est  78.6 %.
Le taux de bonne prédiction pour alpha = 2.6 est  80.4 %.
Le taux de bonne prédiction pour alpha = 2.8 est  79.5 %.
Le taux de bonne prédiction pour alpha = 10 est  79.5 %.
…
```




```
Selon le taux de bonne prédiction, le meilleur paramètre pour le classifieur SGDClassifier est 2.1.
```

V – Conclusion

En conclusion, nous avons exploré comment faire des prédictions de classe à partir d'un régresseur. Pour cela, nous avons utilisé le taux de « bonnes prédictions », mais également utilisé les courbes ROC et observé les matrices de confusion. On retiendra que nous pourrons choisir d'autres régresseurs ou classifieurs selon nos besoins (avoir une bonne sensibilité, ou une bonne spécificité, ou encore avoir un bon compromis).

Si nous devions ne garder qu'une seule méthode au vu de nos données, alors on utiliserait **une PLS avec 10 composantes**, ou un **Lasso(0.1) avec seuil $\alpha = 0,6$** ou encore **SGDCalssifier(2.1)** pour leur bon score global (s'approchant des 85 %) mais aussi pour leur capacité à avoir un bon compromis sensibilité/spécificité.

Comme extension, pour une étude supplémentaire, nous pourrions également donner un poids supplémentaire aux classes sous représenté ou mal prédite, ce que nous n'avons pas fait ici.