Rapport Data Mining 2

May 20, 2021

Étudiants: Adrien Maïtammar, Maxime Le Paumier

Dans ce rapport nous allons analyser la base *wine-quality-red*, à partir du lien suivant :

https://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/

Le rapport est constitué de deux parties: - Analyse des données - Classification

La partie analyse nous permettra de comprendre notre jeu de donnée ainsi que de dégager une problématique à laquelle nous tenterons de répondre dans la partie classificatiion.

1 Analyse du jeu de données

[1]:
```
import os
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import sklearn

url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/
 ↪winequality-red.csv'
data = pd.read_csv(url, delimiter=';')
df = data.copy()
df.sample(8)
```

[1]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	\
154	7.1	0.43	0.42	5.5	0.070	
1179	8.2	0.35	0.33	2.4	0.076	
992	6.5	0.40	0.10	2.0	0.076	
1279	9.8	0.30	0.39	1.7	0.062	
630	8.7	0.54	0.26	2.5	0.097	
411	9.1	0.45	0.35	2.4	0.080	
271	11.5	0.18	0.51	4.0	0.104	
1004	8.2	0.43	0.29	1.6	0.081	

	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	\
154	29.0	129.0	0.99730	3.42	0.72	
1179	11.0	47.0	0.99599	3.27	0.81	

992	30.0	47.0	0.99554	3.36	0.48
1279	3.0	9.0	0.99480	3.14	0.57
630	7.0	31.0	0.99760	3.27	0.60
411	23.0	78.0	0.99870	3.38	0.62
271	4.0	23.0	0.99960	3.28	0.97
1004	27.0	45.0	0.99603	3.25	0.54

	alcohol	quality
154	10.5	5
1179	11.0	6
992	9.4	6
1279	11.5	7
630	9.3	6
411	9.5	5
271	10.1	6
1004	10.3	5

1.1 Description des variables

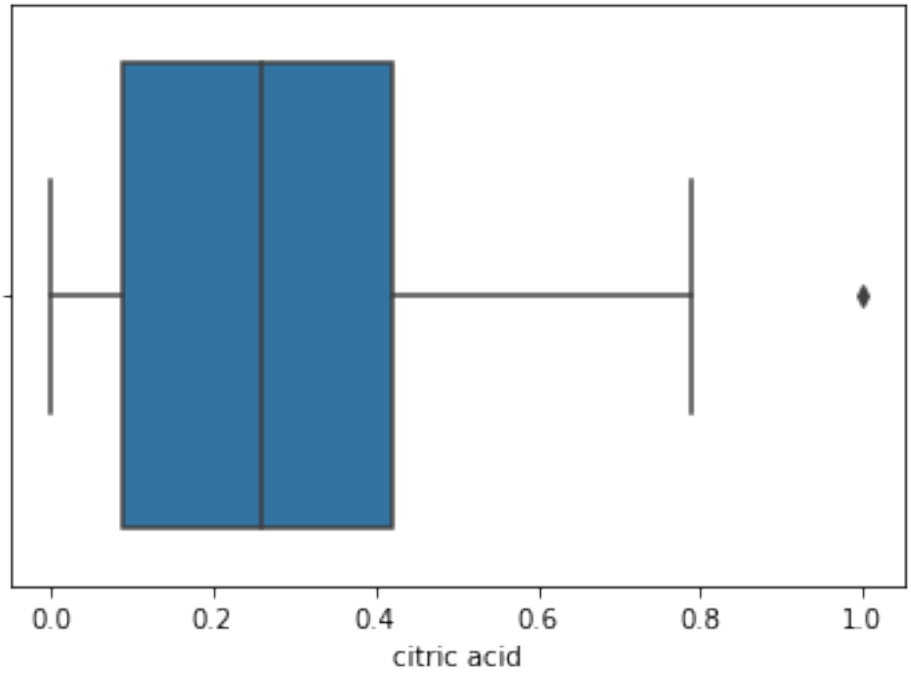
- `fixed acidity` (g(tartaric acid)/dm3): la plupart des acides présents dans le vin sont fixes ou non volatiles (ne s'évaporent pas facilement).
- `volatile acidity` (g(acetic acid)/dm3): la quantité d'acide acétique dans le vin, qui, à des niveaux trop élevés, peut donner un goût désagréable de vinaigre.
- `citric acid` (g/dm3): présent en petites quantités, l'acide citrique peut ajouter de la fraîcheur et de la saveur au vin.
- `residual sugar` (g/dm3): la quantité de sucre restant après l'arrêt de la fermentation. Il est rare de trouver des vins avec moins de 1 gramme/litre. Les vins avec plus de 45 grammes/litre sont considérés comme doux.
- `chlorides` (g(sodium chloride)/dm3): la quantité de sel dans le vin.
- `free sulfur dioxide` (mg/dm3): la forme libre du SO2; elle empêche la croissance microbienne et l'oxydation du vin.
- `total sulfur dioxide` (mg/dm3): quantité de formes libres et liées de S02 ; à faible concentration, le SO2 est généralement indétectable dans le vin, mais à des concentrations de SO2 libre supérieures à 50 ppm, le SO2 devient évident au nez et au goût du vin.
- `density` (g/cm3): la densité du vin est proche de celle de l'eau, selon le pourcentage d'alcool et la teneur en sucre.
- `pH`: décrit l'acidité ou la basicité d'un vin sur une échelle allant de 0 (très acide) à 14 (très basique) ; la plupart des vins se situent entre 3 et 4 sur l'échelle du pH.
- `sulphates` (g(potassium sulphate)/dm3): un additif du vin qui peut contribuer aux niveaux de gaz sulfureux (S02), qui agit comme un antimicrobien et un antioxydant.
- `alcohol` (% vol.): le pourcentage d'alcool contenu dans le vin.
- `quality`: variable de sortie (basée sur des données sensorielles, note entre 0 et 10).

1.2 Généralités

Une première analyse de cette base de donnée peut être effectuée avec le module `ProfileReport` du package `pandas_profiling`. Le code correspondant ainsi que le détail des résultats peuvent être retrouvés en annexe. Les points importants sont les suivants: - On dispose de 1599 observations et

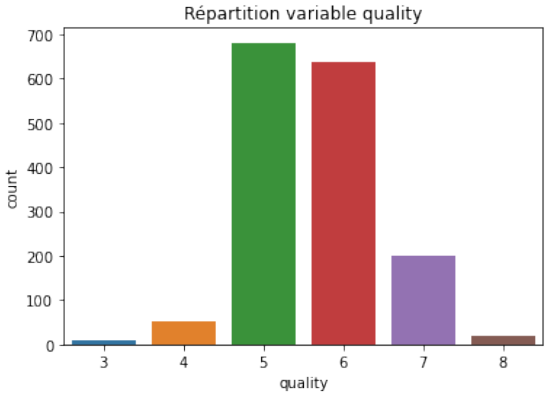
de 11 variables quantitatives continues et d'une variable qualitative ordinale, `quality`, que l'on va tenter de prédire. - Il n'y a pas de valeur manquante. - Il y a 15% de lignes dupliquées. - Il y a 8,3% de zeros pour la variable `citric acid`. - La répartition des valeurs pour la variable `quality` est très inégale.

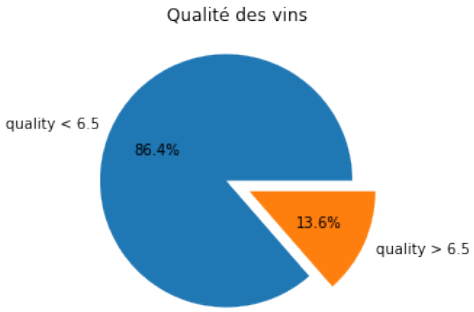
[2]:
```
sns.boxplot(x='citric acid', data=df);
```



Les zeros de la variable `citric acid` ne semblent pas abérrants au vu de la distribution de la variable.

[3]:
```
plt.figure(figsize=(13,4))
plt.subplot(1,2,1)
sns.countplot(x='quality', data=df)
plt.title('Répartition variable quality');
plt.subplot(1,2,2)
bons_vins = df.quality > 6.5
labels = labels=['quality < 6.5', 'quality > 6.5']
plt.pie(x=bons_vins.value_counts(), autopct="%.1f%%", explode=[0.1]*2,␣
 ↪labels=labels)
plt.title('Qualité des vins');
```

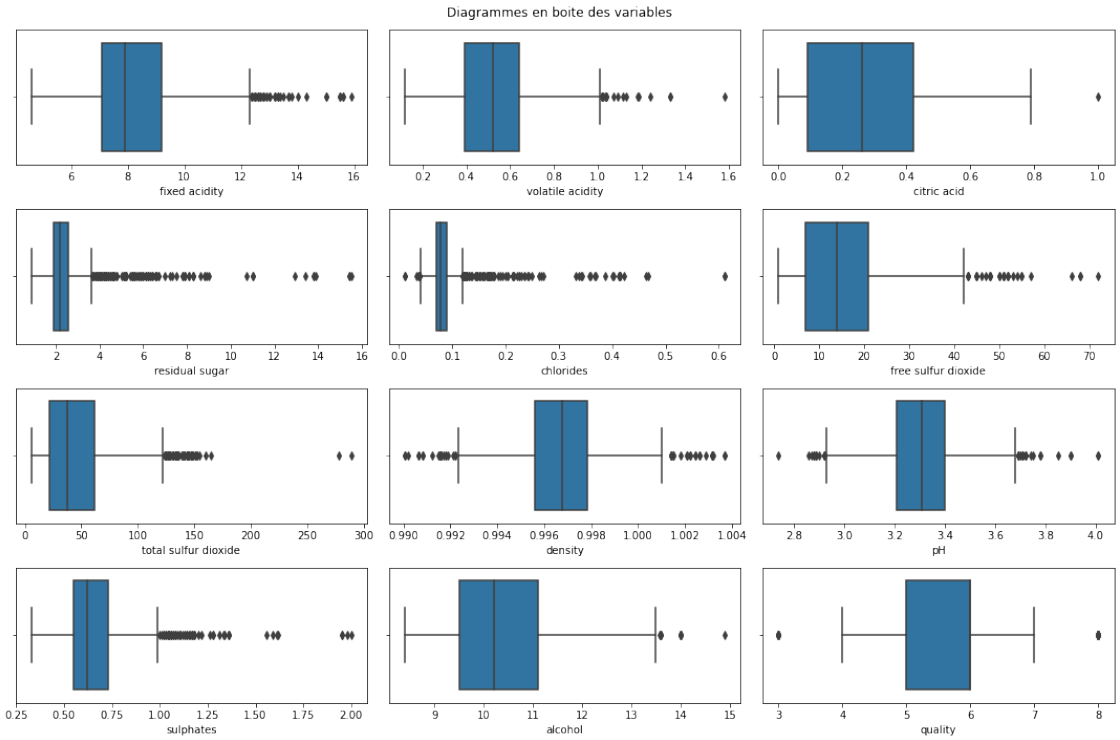




Les classes sont assez déséquilibrées, il y a beaucoup plus de vins avec une note moyenne que de vins avec une mauvaise ou une bonne note.

1.3 Statistiques univariés

[4]:
```
fig = plt.figure(figsize=(15,10))
for i, column in enumerate(df.columns):
    plt.subplot(4,3,i+1)
    fig.tight_layout(pad=2)
    sns.boxplot(x = column, data=df)
fig.suptitle('Diagrammes en boite des variables')
fig.tight_layout()
fig.subplots_adjust(top=0.95)
```

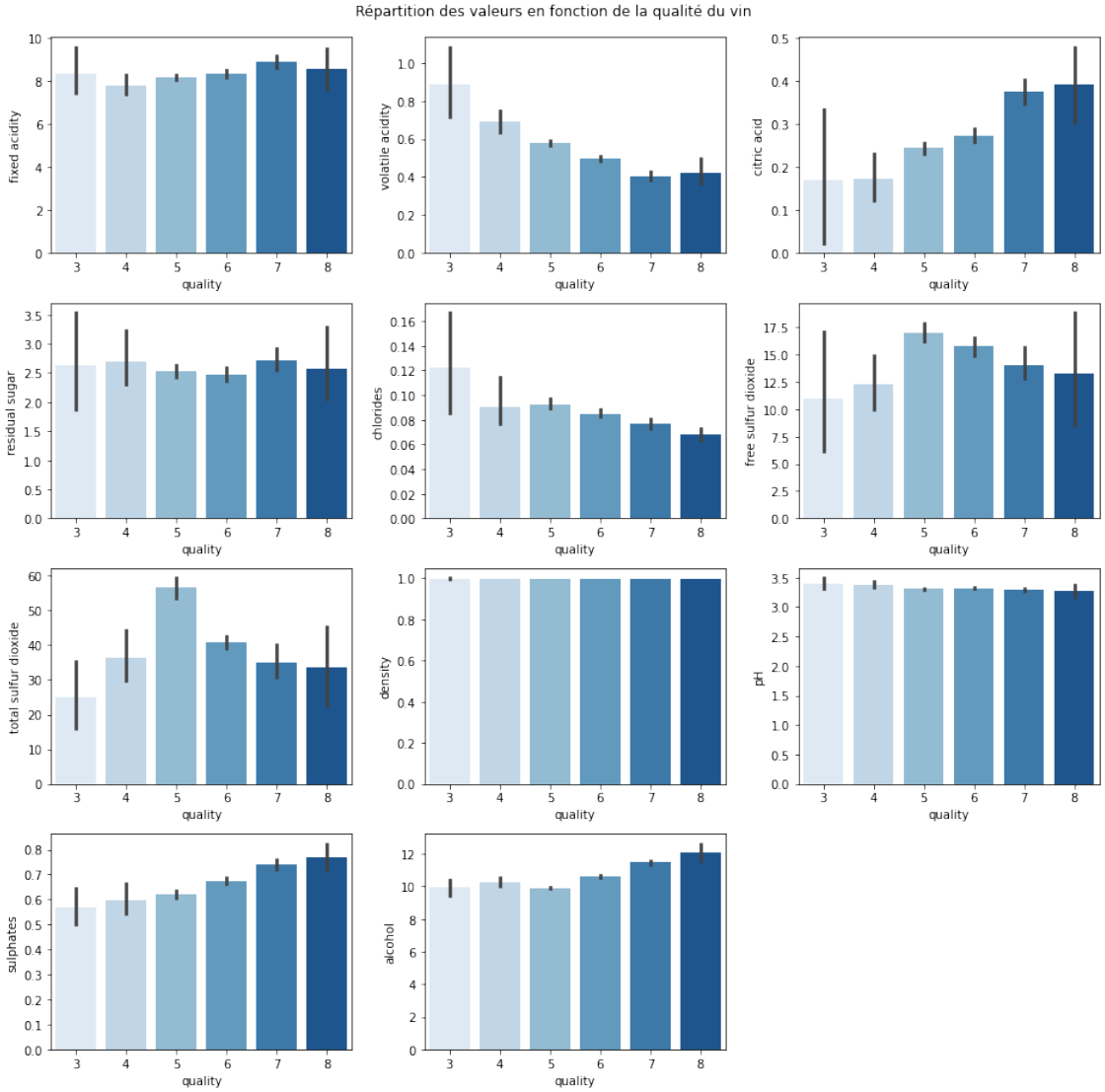


On remarque qu'il y a beaucoup d'outliers pour la majorité des variables.

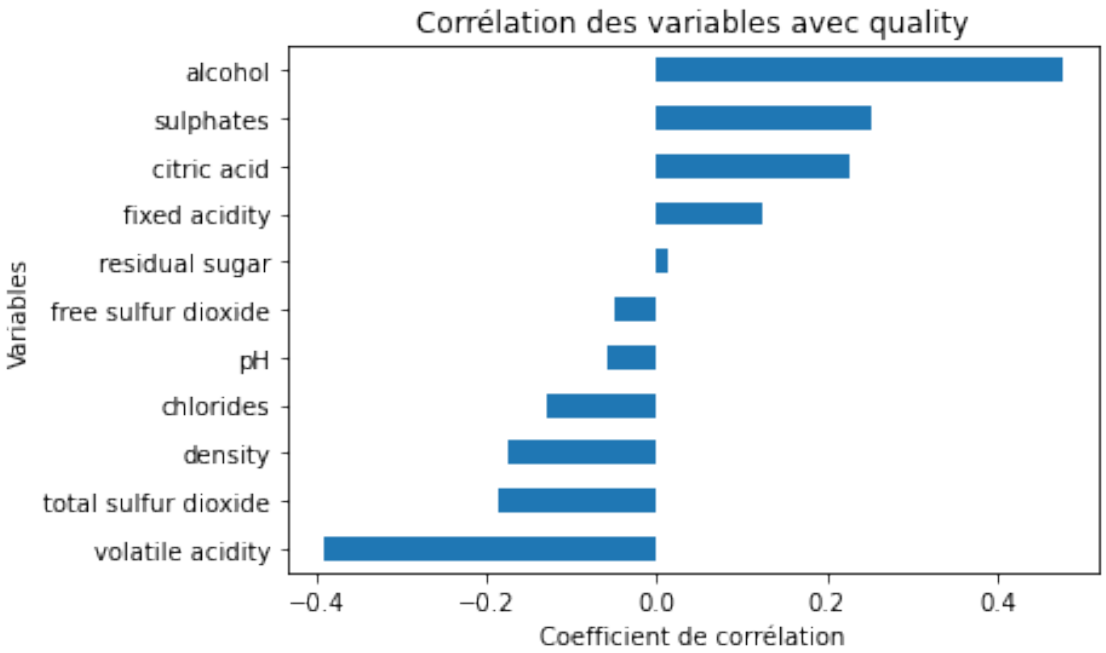
1.4 Statistiques bivariées

Intéressons nous maintenant aux corrélations entre la variable objectif `quality` et les autres variables.

[5]:
```
fig = plt.figure(figsize = (13, 13))
for i, column in enumerate(df.drop('quality', axis=1).columns):
    plt.subplot(4, 3, i+1)
    sns.barplot(x = 'quality', y = f'{column}', data = df, palette='Blues')
fig.suptitle('Répartition des valeurs en fonction de la qualité du vin')
fig.tight_layout()
fig.subplots_adjust(top=0.95)
```

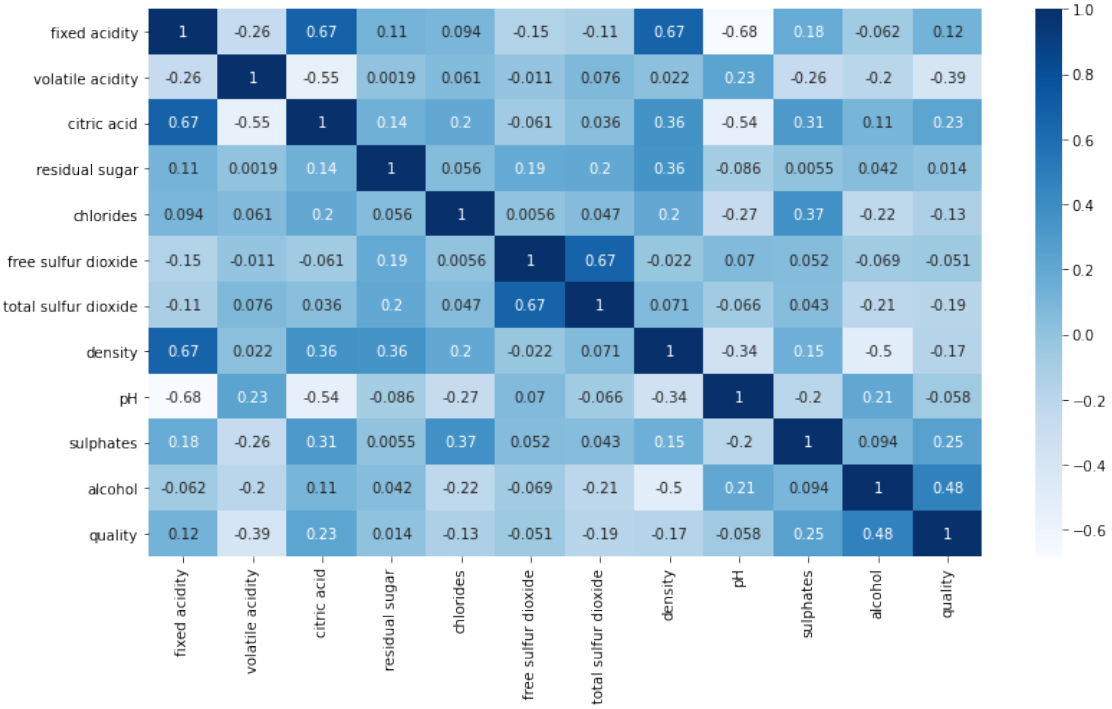


[6]:
```
df.corr().quality.sort_values(ascending=False).iloc[1:12][::-1].
 ↪plot(kind='barh')
plt.title("Corrélation des variables avec quality", size=12)
plt.xlabel("Coefficient de corrélation")
plt.ylabel("Variables");
```



Nous pouvons voir que sur notre échantillon la variable `quality` est principalement corrélée: - positivement avec la variable `alcohol` - négativement avec la variable `volatile acidity`.

[7]:
```
plt.figure(figsize=(13, 7))
sns.heatmap(df.corr(), cmap='Blues', annot=True);
```



Concernant les autres variables, on remarque des corrélations positives entre les variables `fixed acidity`, `citric acid` et `density`. Les variables `pH` et `fixed acidity` sont quant à elles corrélées négativement. Ces conclusions sont logiques si on considère les propriétés des physiques de ces éléments.

1.5 Tests d'hypothèses

1.5.1 Test de Student

Testons les hypothèses nulles suivantes avec un test de Student:

1. les vins bien notés ont un volume d'alcool moyen égaux aux autres vins
2. les vins moins bien notés présentent une acidité volatile moyenne égale aux autre vins.

Le tests de Student se font sur des classes équilibrées. Nous allons donc tirer au hasard dans **medium_rate_df** et **good_rate_df** des échantillons de même taille.

[8]:
```
def prepare_test(df):
    good_mark_df = df[df.quality > 6.5]
    medium_mark_df = df[df.quality < 6.5]
    balanced_medium_mark_df = medium_mark_df.sample(good_mark_df.shape[0])
    return balanced_medium_mark_df, good_mark_df
```

[9]:
```
from scipy.stats import ttest_ind, kstest
def test(col, test):
    medium_rate_df, good_rate_df = prepare_test(df)
    alpha = 0.01
    stat, p = test(medium_rate_df[col], good_rate_df[col])
    if p < alpha:
        return 'H0 Rejetée'
    else :
        return 0
```

[10]:
```
print('H0 : Les vins bien notés ont les mêmes moyennes que les vins avec des␣
 ↪notes moyennes.\n')
print(f'{str( ) :<25} {"ttest":<15} {"kstest"}')
for col in df.columns:
    print(f'{col :<25} {test(col, ttest_ind) :<15} {test(col, kstest) :<5}')
```

```
H0 : Les vins bien notés ont les mêmes moyennes que les vins avec des notes
moyennes.

                         ttest           kstest
fixed acidity            0               H0 Rejetée
volatile acidity         H0 Rejetée      H0 Rejetée
citric acid              H0 Rejetée      H0 Rejetée
residual sugar           0               0
```

```
chlorides                H0 Rejetée      H0 Rejetée
free sulfur dioxide      H0 Rejetée      H0 Rejetée
total sulfur dioxide     H0 Rejetée      H0 Rejetée
density                  H0 Rejetée      H0 Rejetée
pH                       0               0
sulphates                H0 Rejetée      H0 Rejetée
alcohol                  H0 Rejetée      H0 Rejetée
quality                  H0 Rejetée      H0 Rejetée
```

On peut conclure d'après le résultats des tests que les variables `residual sugar` et `pH` ne semblent pas avoir d'impact significatif sur la variable `quality`.

1.5.2 Test du chi2

Le test du chi2 mesure la dépendance entre les variables stochastiques, donc l'utilisation de cette fonction détecte les caractéristiques qui sont les plus susceptibles d'être indépendantes de la variable objectif y et donc non pertinentes pour la classification.

[11]:
```
df.drop(['quality'], axis=1)
```

[11]:
```
      fixed acidity  volatile acidity  citric acid  residual sugar  chlorides  \
0               7.4             0.700         0.00             1.9      0.076
1               7.8             0.880         0.00             2.6      0.098
2               7.8             0.760         0.04             2.3      0.092
3              11.2             0.280         0.56             1.9      0.075
4               7.4             0.700         0.00             1.9      0.076
…               …                 …            …                …        …
1594            6.2             0.600         0.08             2.0      0.090
1595            5.9             0.550         0.10             2.2      0.062
1596            6.3             0.510         0.13             2.3      0.076
1597            5.9             0.645         0.12             2.0      0.075
1598            6.0             0.310         0.47             3.6      0.067

      free sulfur dioxide  total sulfur dioxide  density    pH  sulphates  \
0                    11.0                  34.0  0.99780  3.51       0.56
1                    25.0                  67.0  0.99680  3.20       0.68
2                    15.0                  54.0  0.99700  3.26       0.65
3                    17.0                  60.0  0.99800  3.16       0.58
4                    11.0                  34.0  0.99780  3.51       0.56
…                     …                     …        …     …          …
1594                 32.0                  44.0  0.99490  3.45       0.58
1595                 39.0                  51.0  0.99512  3.52       0.76
1596                 29.0                  40.0  0.99574  3.42       0.75
1597                 32.0                  44.0  0.99547  3.57       0.71
1598                 18.0                  42.0  0.99549  3.39       0.66

      alcohol
0         9.4
```

```
1          9.8
2          9.8
3          9.8
4          9.4
…          …
1594      10.5
1595      11.2
1596      11.0
1597      10.2
1598      11.0

[1599 rows x 11 columns]
```

[12]:
```
from sklearn.feature_selection import chi2
X = df.drop(['quality'], axis=1)
y = df.quality
test_values, p_values = np.round(chi2(X, y),5)
res = pd.DataFrame({'Val. test':test_values, 'p-values':p_values, 'H0 rejected':
 ↪np.round(chi2(X, y),5)[1] < 0.05}, index=X.columns)
res
```

[12]:

	Val. test	p-values	H0 rejected
fixed acidity	11.26065	0.04645	True
volatile acidity	15.58029	0.00815	True
citric acid	13.02567	0.02314	True
residual sugar	4.12329	0.53180	False
chlorides	0.75243	0.97997	False
free sulfur dioxide	161.93604	0.00000	True
total sulfur dioxide	2755.55798	0.00000	True
density	0.00023	1.00000	False
pH	0.15465	0.99953	False
sulphates	4.55849	0.47210	False
alcohol	46.42989	0.00000	True

Seules les variables `fixed acidity`, `volatile acidity`, `citric acid`, `free sulfur dioxide`, `total sulfur dioxide` et `alcohol` semblent avoir une dépendance avec la variable `quality` au seuil de 5%.

1.6 Bilan de l'analyse

Le jeu de données ne présente pas de difficulté particulière pour le nettoyage et la préparation si ce n'est le fait que la proportion de vins jugés comme bons est assez faible. Il serait intéressant de remédier à ce problème pour obtenir de meilleures prédictions. Il est en effet plus dur pour un modèle de Machine Learning d'apprendre les caratéristiques d'une classe minoritaire. De plus, un modèle naïf peut atteindre une précision raisonnable en prédisant simplement la classe majoritaire dans tous les cas.

En outre, certaines variables disposent de nombreux outliers.

Il serait avant tout intéressant de procéder à du feature engineering (ré-échantillonage, sélection de variables).

2 Classification

Dans cette partie, nous allons tenter de construire un modèle permettant de prédire si un vin a obtenu une bonne note ou une mauvaise. Pour cela nous allons évaluer plusieurs modèles. Puis nous procédrons à des transformations sur le jeu de donnée et optimiserons les modèles afin d'afinner les prédictions.

[13]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
warnings.simplefilter(action='ignore')

url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/
 ↪winequality-red.csv'
data = pd.read_csv(url, delimiter=';')
df = data.copy()
df.shape
```

[13]:
```
(1599, 12)
```

Il est important d'effectuer une copie du jeu de donnée afin de le garder tel qu'il est. Toutes les recherches seront effectuées sur sa copie.

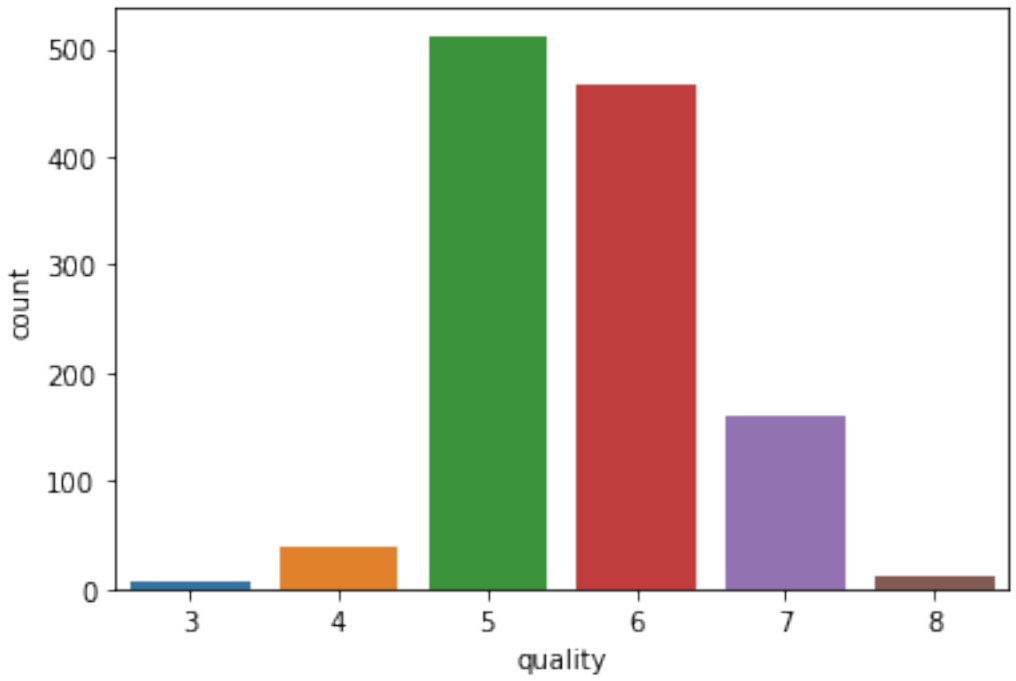
2.1 Preprocessing

Nous allons séparer le jeu de données en deux, soit un jeu d'entrainement et un jeu de test. Le jeu de test ne sera utilisé qu'à la toute fin pour ne pas biaiser les résultats.

[14]:
```
from sklearn.model_selection import train_test_split
trainset, testset = train_test_split(df, test_size=0.25, random_state=0)
```

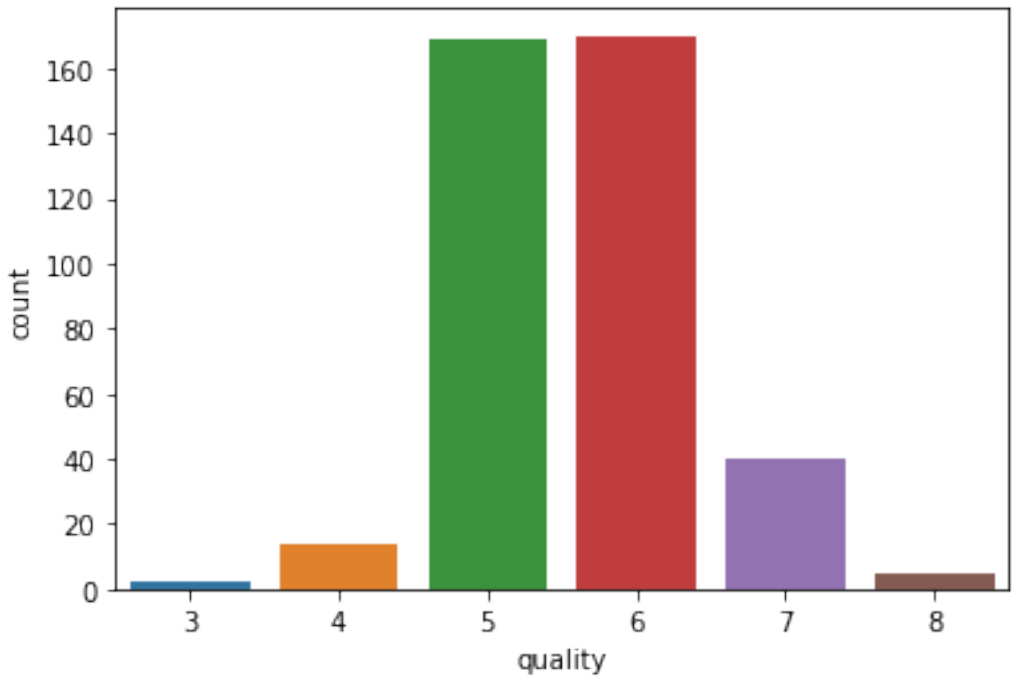
[15]:
```
sns.countplot(x='quality', data=trainset);
trainset.quality.value_counts()
```

[15]:
```
5    512
6    468
7    159
4     39
8     13
3      8
Name: quality, dtype: int64
```



[16]:
```
sns.countplot(x='quality', data=testset);
testset.quality.value_counts()
```

[16]:
```
6    170
5    169
7     40
4     14
8      5
3      2
Name: quality, dtype: int64
```



La répartition des classes au sein du trainset et du testset est bien similaire.

À présent construisons une fonction permettant de traiter le jeu de données afin de préparer la phase d'apprentisage. Le jeu de donnée, étant uniquement constitué de variables numériques et n'ayant pas de valeurs manquantes, ne nécéssite pas beaucoup de transformations. Dans un premier temps, nous affectons à X les colonnes des variables et à y la colonne objectif `df['good_wine']` telle que: `df['good_wine'] = df.quality > 6.5`.

[17]:
```
def preprocessing(df):
    df['good_wine'] = df.quality > 6.5
    X = df.drop(['quality', 'good_wine'], axis=1)
    y = df['good_wine']
    print(y.value_counts())
    return X, y
```

[18]:
```
X_train, y_train = preprocessing(trainset)
```

```
False    1027
True      172
Name: good_wine, dtype: int64
```

[19]:
```
X_test, y_test = preprocessing(testset)
```

```
False    355
True      45
Name: good_wine, dtype: int64
```

2.2 Procédure d'évaluation

Il est important de décider dès à présent les modalités de la procédure d'évaluation des modèles. Cela permettra de les comparer de manière objective. Pour ce faire, on décide de prendre comme métrique le score F_1 tel que

$$F_1 = 2 * \frac{spcificit * sensibilit}{spcificit + sensibilit}$$

avec

$$sensibilit = \frac{TP}{TP + FN}$$

et

$$spcificit = \frac{TN}{TN + FP}$$

On aussi choisit de prendre le score `f_weighted`. Ce score calcule les scores F_1 pour chaque étiquette, et trouve leur moyenne pondérée par le support (le nombre d'instances vraies pour chaque étiquette). Cela modifie le score pour prendre en compte le déséquilibre des étiquettes. Cela peut donner un score F_1 qui ne se situe pas entre la précision et le rappel.

On utilise aussi la courbe d'apprentissage, learning curve en anglais, qui nous donne des informations essentielles sur l'état d'apprentissage du modèle: surapprentissage ou bien en sous-apprentissage.

Nous afficherons aussi à titre indicatif le score ROC AUC mais cet indicateur ne sera pas décisif dans le choix du modèle.

[20]:
```
from sklearn.model_selection import StratifiedKFold, cross_val_score,␣
↪cross_val_predict, cross_validate
from sklearn.metrics import f1_score, roc_auc_score, plot_roc_curve,␣
↪confusion_matrix, ConfusionMatrixDisplay
from sklearn.model_selection import learning_curve

def evaluation(X_train, y_train, model, display_graph=False):

    cv = StratifiedKFold(n_splits=4)

    y_pred = cross_val_predict(model,
                               X_train,
                               y_train,
                               cv = cv)

    score = f1_score(y_train, y_pred)
    score_w = f1_score(y_train, y_pred, average='weighted')
    score_auc = roc_auc_score(y_train, y_pred)

    cm = confusion_matrix(y_train, y_pred)

    N, train_score, val_score = learning_curve(model, X_train, y_train,
                                               cv = cv,
                                               scoring = 'f1_weighted',
```

```
                                                    train_sizes = np.linspace(0.1,␣
↪1, 10))

    if display_graph:
        print(f'{model[-1].__class__.__name__}')
        print(f'F_1: {round(score,3)}, F_1 weighted: {round(score_w,3)}, AUC:␣
↪{round(score_auc,3)}')
        fig, ax = plt.subplots(1, 2, figsize = (11,4))
        fig.subplots_adjust(top=0.80)
        # Confusion matrix
        ax[0].set_title('Confusion matrix')
        cm_display = ConfusionMatrixDisplay(cm).plot(cmap='Blues', ax=ax[0])
        # Learning curve with F1 score
        ax[1].set_title('Learning curve')
        ax[1].plot(N, train_score.mean(axis=1), label='train score')
        ax[1].plot(N, val_score.mean(axis=1), label='validation score')

        plt.suptitle(model[-1].__class__.__name__, fontsize=14)
        plt.legend()
        plt.show()

    return score, score_w, score_auc
```

2.3 Modélisation

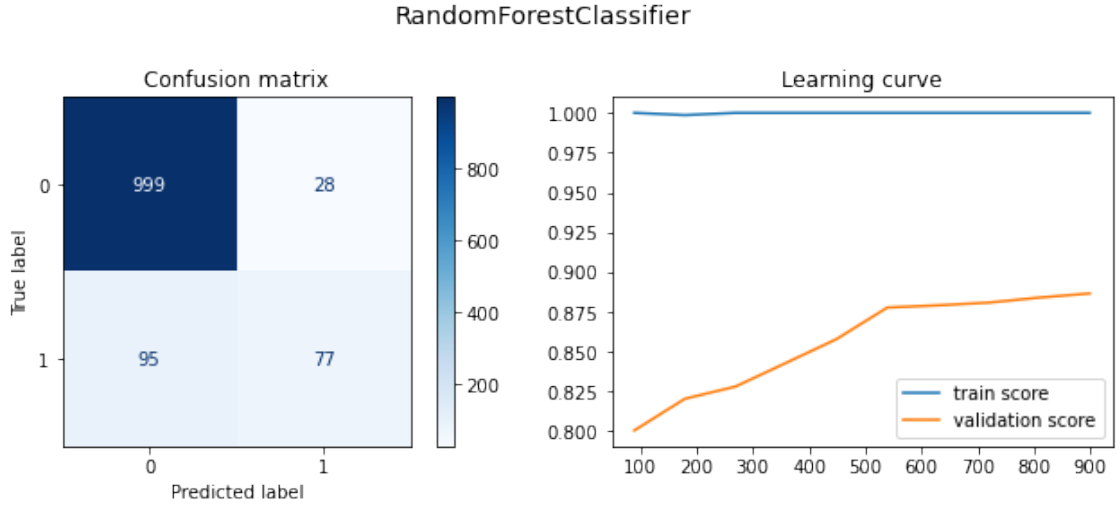
Testons à présent, de manière rapide et sans optimisation, plusieurs modèles afin de retenir quelques-uns des plus prometteurs. Pour se faire, nous allons utiliser des pipelines permettant dans cette partie d'initialiser les modèles. Cet outils sera utile par la suite pour améliorer/tuner les modèles d'apprentissage.

[21]:
```
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
# models
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.neural_network import MLPClassifier

RandomForest = make_pipeline(RandomForestClassifier(random_state=0))
SVM = make_pipeline(StandardScaler(), SVC(random_state=0))
KNN = make_pipeline(StandardScaler(), KNeighborsClassifier())
GBClassifier = make_pipeline(StandardScaler(),␣
 ↪GradientBoostingClassifier(random_state=0))
GaussianNB = make_pipeline(StandardScaler(), GaussianNB())
MLPClassifier = make_pipeline(StandardScaler(),MLPClassifier(random_state=0))
```

```
dict_of_models = {'RandomForest':RandomForest,
                  'GBoost':GBClassifier,
                  'SVM':SVM,
                  'KNN':KNN,
                  'NaiveBayes':GaussianNB,
                  'MLPClassifier':MLPClassifier
                 }
```

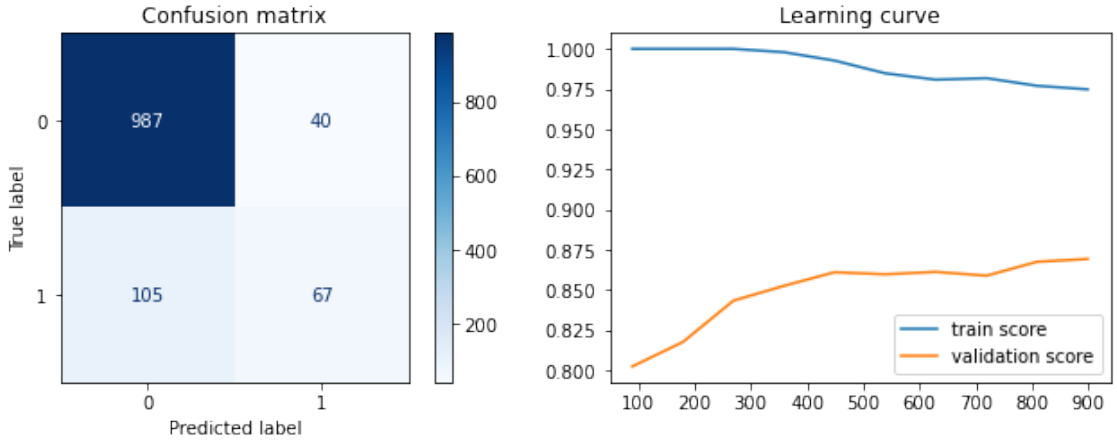
[22]:
```
scores_f1 = []
scores_f1_weighted = []
scores_auc = []
for name, model in dict_of_models.items():
    score, score_w, score_auc = evaluation(X_train, y_train, model,␣
 ↪display_graph=True)
    scores_f1.append(score)
    scores_f1_weighted.append(score_w)
    scores_auc.append(score_auc)

res = pd.DataFrame({'Score F1':np.round(scores_f1,2),
                    'Score F1 weighted':np.round(scores_f1_weighted,2),
                    'Score AUC':scores_auc},
                   index=dict_of_models.keys())
display(res.sort_values(by=['Score F1 weighted','Score F1'], ascending=False))
```

```
RandomForestClassifier
F_1: 0.556, F_1 weighted: 0.887, AUC: 0.71
```



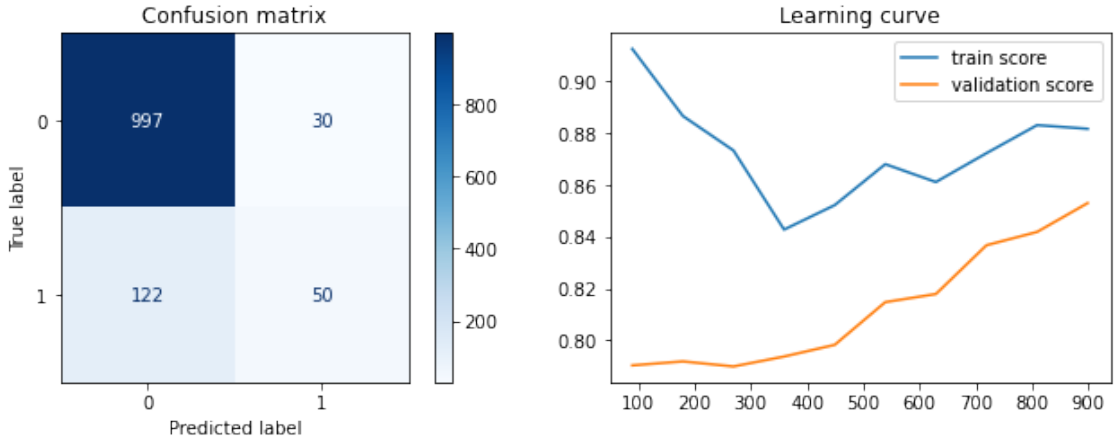

```
GradientBoostingClassifier
F_1: 0.48, F_1 weighted: 0.867, AUC: 0.675
```

GradientBoostingClassifier

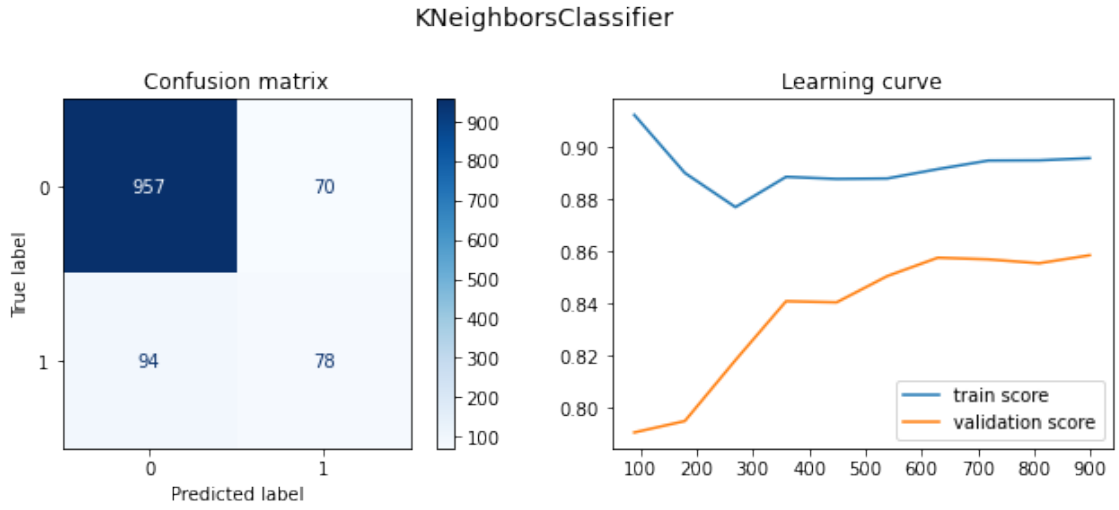


SVC
F_1: 0.397, F_1 weighted: 0.853, AUC: 0.631

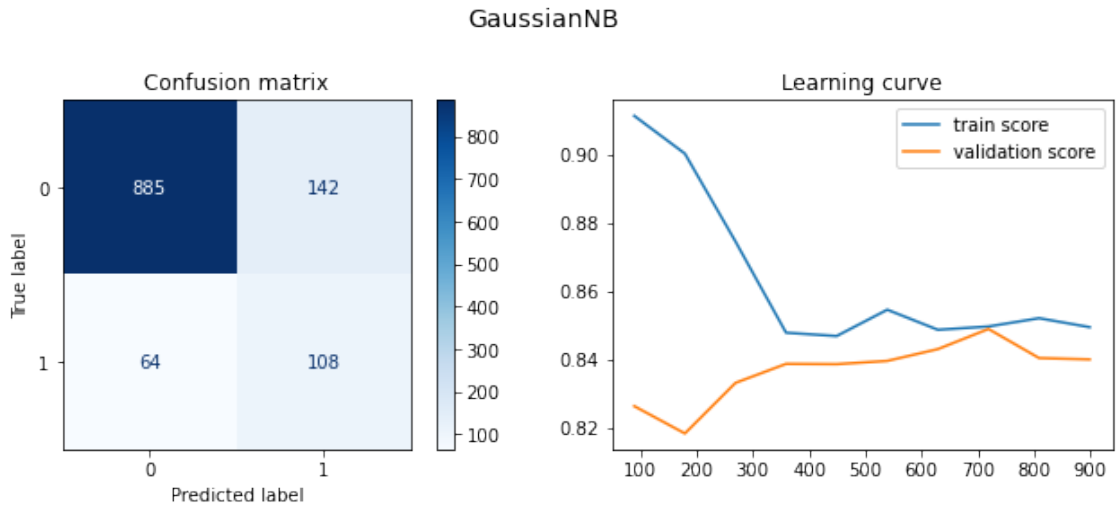
SVC



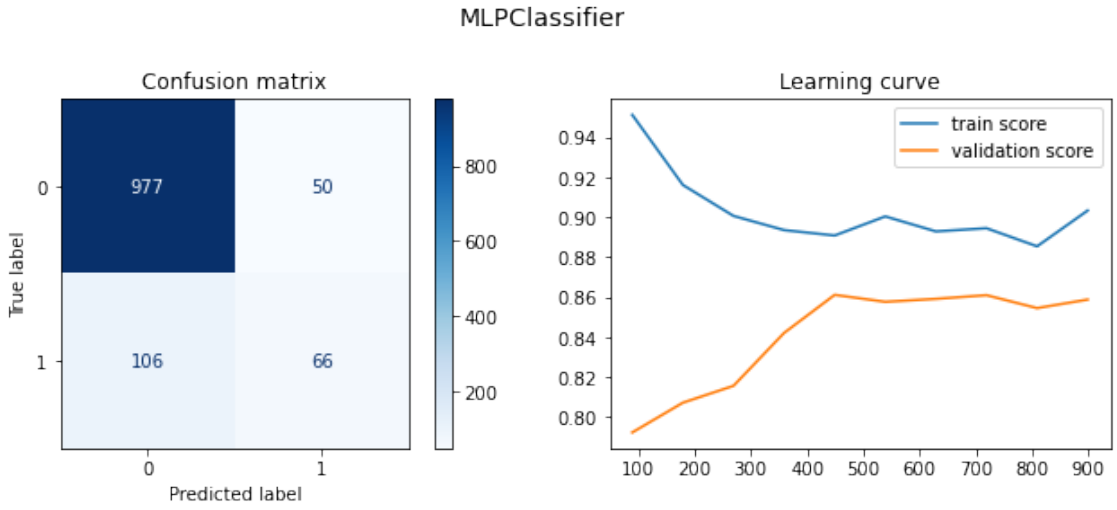
KNeighborsClassifier
F_1: 0.488, F_1 weighted: 0.859, AUC: 0.693



GaussianNB
F_1: 0.512, F_1 weighted: 0.841, AUC: 0.745



MLPClassifier
F_1: 0.458, F_1 weighted: 0.859, AUC: 0.668



	Score F1	Score F1 weighted	Score AUC
RandomForest	0.56	0.89	0.710205
GBoost	0.48	0.87	0.675293
KNN	0.49	0.86	0.692664
MLPClassifier	0.46	0.86	0.667518
SVM	0.40	0.85	0.630743
NaiveBayes	0.51	0.84	0.744820

Nous pouvons voir que le meilleur modèle est le `RandomForest`. En effet, ce modèle bat la très grande majorité des autres modèle sur les trois indicateurs.

2.4 Optimisation

Nous allons dans cette partie essayer de selectionner les meilleurs variables ainsi qu'optimiser les modèles `RandomForest`, `GBClassifier` et `KNN`.

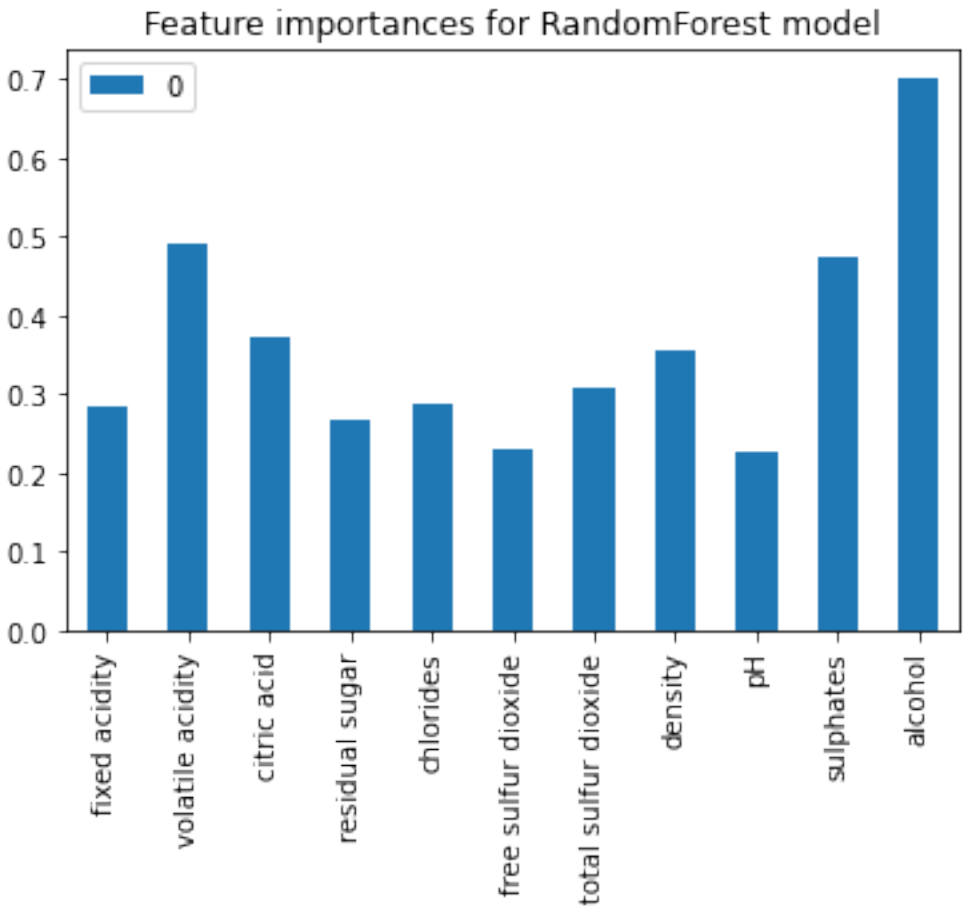
2.4.1 Feature engineering

Observons l'importance des variables au sein du modèle `RandomForest`.

[23]:
```
cross_validate
rf_estimators = cross_validate(RandomForest,
                               X_train,
                               y_train,
                               cv=StratifiedKFold(4),
                               scoring='f1_weighted',
                               return_estimator=True)['estimator']

feature_importances = np.zeros(len(X_train.columns))
for estimator in rf_estimators:
    feature_importances += estimator[-1].feature_importances_
```

```
plt.figure()
pd.DataFrame(feature_importances, index=X_train.columns).plot.bar()
plt.title('Feature importances for RandomForest model');
```

```
<Figure size 432x288 with 0 Axes>
```



Aucune variable ne semble à priori à exclure impérativement. On remarque cependant que la variable `alcohol` est une variable ayant beaucoup d'influence à l'instar de la variable `volatile acidity`. On retrouve donc les variables ayant les plus forte corrélations.

Peu de variables sont présentes dans notre dataset. De plus les courbes d'apprentissage des modèles `RandomForest` et `KNN` ne semblent pas avoir atteint leur plateau sur la courbe du validation set, cela est un signe de sous ajustement (underfitting). Augmenter le degré des variables permettra peut-être de trouver un juste millieu entre underfitting et overfitting pour nos modèles.

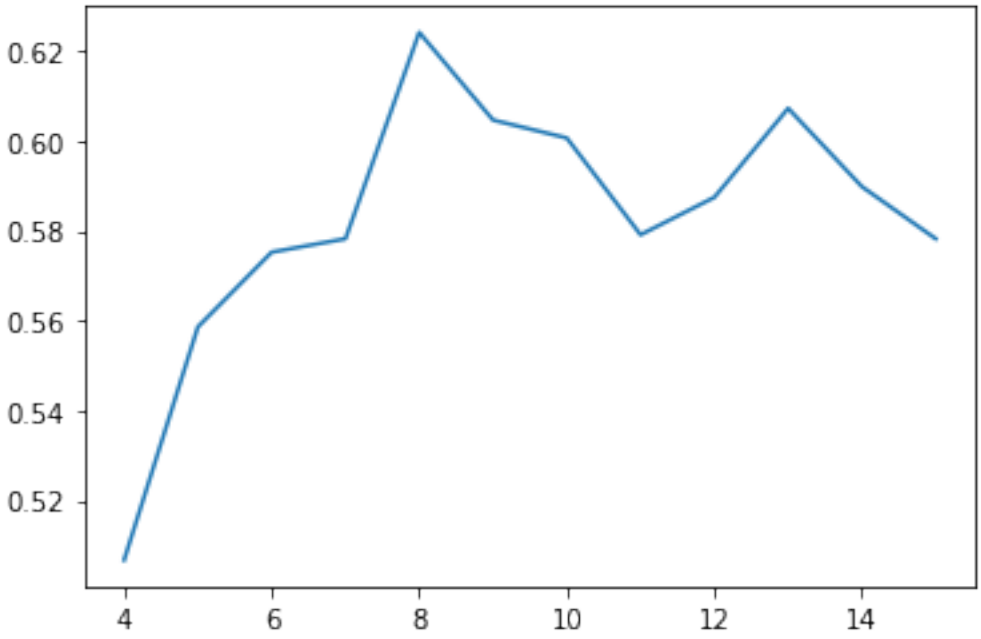
[24]:
```
from sklearn.preprocessing import PolynomialFeatures
poly = PolynomialFeatures(2)
X_train_poly = pd.DataFrame(poly.fit_transform(X_train))
```

```
print(X_train_poly.shape)
```

```
(1199, 78)
```

Select K Best La sélection de caractéristiques univariées consiste à sélectionner les meilleures caractéristiques sur la base de tests statistiques univariés. Le test considéré dans notre cas est l'ANOVA.

[25]:
```
from sklearn.feature_selection import SelectKBest, f_classif
f1_scores = []
for k in range(4,16):
    model = RandomForestClassifier(random_state=0)
    selector_kbest = make_pipeline(SelectKBest(f_classif, k=k), model)
    f1_scores.append(evaluation(X_train_poly, y_train, selector_kbest)[0])
plt.plot(range(4,16), f1_scores);
```

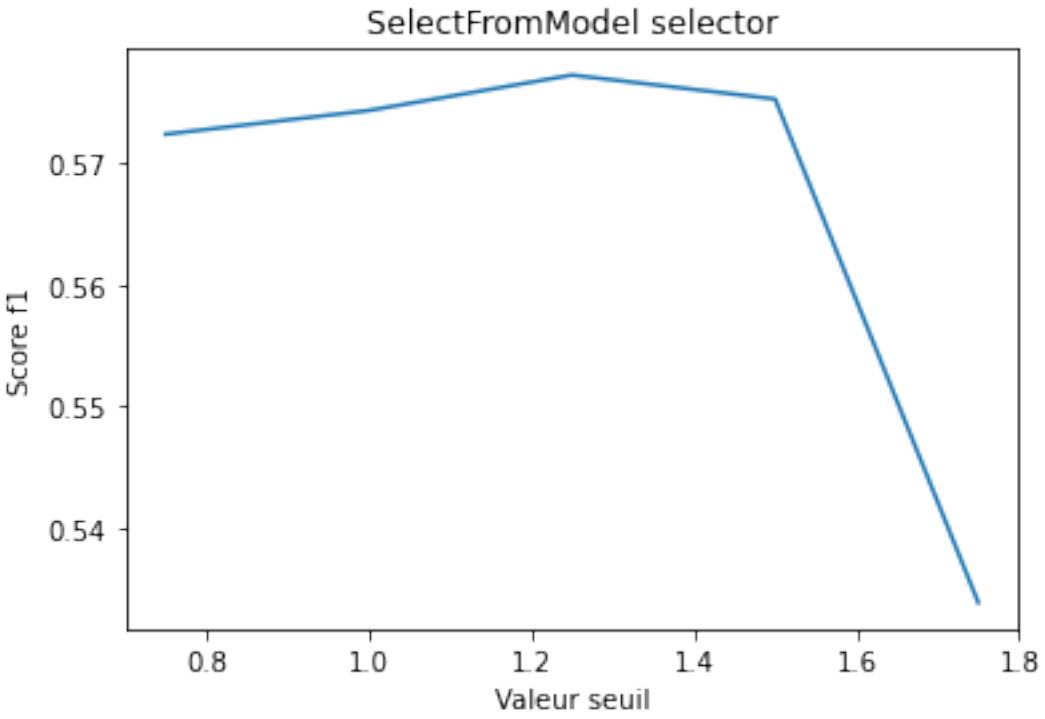


Pour cette méthode, les meilleurs résultats sont obtenus lorsque nous prenons les 8 meilleures variables.

Select From Model SelectFromModel est un méta-transformateur qui peut être utilisé avec n'importe quel estimateur qui attribue une importance à chaque caractéristique par le biais d'un attribut spécifique (tel que coef_, feature_importances_) ou via un importance_getter appelable après l'ajustement. Les caractéristiques sont considérées comme non importantes et supprimées si l'importance correspondante des valeurs des caractéristiques est inférieure au paramètre de seuil fourni. En plus de spécifier le seuil numériquement, il existe des heuristiques intégrées pour trouver un seuil en utilisant un argument de type chaîne. Les heuristiques disponibles sont "moyenne",

“médiane” et des multiples flottants de ceux-ci comme ”0.1*moyenne”.

[26]:
```
from sklearn.feature_selection import SelectFromModel
f1_scores = []
x = np.arange(0.75, 2, 0.25)
for k in x:
    selector_sfm = SelectFromModel(RandomForestClassifier(random_state=0),␣
 ↪threshold=f'{k}*mean')
    selector_sfm.fit_transform(X_train_poly, y_train)
    X_train_sfm = X_train_poly[X_train_poly.columns[selector_sfm.get_support()]]
    f1_scores.append(evaluation(X_train_sfm, y_train, RandomForest)[0])
plt.plot(x, f1_scores)
plt.xlabel('Valeur seuil')
plt.ylabel('Score f1')
plt.title('SelectFromModel selector');
```



Les meilleurs résultat sont obtenus pour une valeur seuil correspondant à 1,25 fois la moyenne.

PCA L’ACP est utilisée pour décomposer un ensemble de données multivariées en un ensemble de composantes orthogonales successives qui expliquent une quantité maximale de la variance. Dans scikit-learn, l’ACP est implémentée comme un objet transformateur qui apprend les composantes dans sa méthode d’ajustement, et peut être utilisée sur de nouvelles données pour les projeter sur ces composantes.

L’ACP centre mais ne met pas à l’échelle les données d’entrée pour chaque caractéristique avant

d'appliquer le SVD. Le paramètre optionnel whiten=True permet de projeter les données sur l'espace singulier tout en mettant à l'échelle chaque composante à la variance unitaire.

[27]:
```
from sklearn.decomposition import PCA
model = RandomForestClassifier(random_state=0)
selector_pca = make_pipeline(PCA(10,whiten=True), model)
evaluation(X_train, y_train, selector_pca)
```

[27]:
```
(0.5421245421245421, 0.8839313232189648, 0.7019711963044315)
```

PLS PLS présente des similitudes avec la régression en composantes principales (PCR), où les échantillons sont d'abord projetés dans un sous-espace de dimension inférieure, et les cibles y sont prédites en utilisant transformé(X). Un problème avec la PCR est que la réduction de la dimensionnalité n'est pas supervisée et peut perdre certaines variables importantes : La PCR conserve les caractéristiques ayant la plus grande variance, mais il est possible que les caractéristiques ayant une faible variance soient pertinentes pour prédire la cible. D'une certaine manière, PLS permet le même type de réduction de la dimensionnalité, mais en prenant en compte les cibles y.

[28]:
```
from sklearn.cross_decomposition import PLSSVD
model = RandomForestClassifier(random_state=0)
pls = PLSSVD(n_components=10).fit(X_train, y_train)
X_pls = pls.transform(X_train)
evaluation(X_pls, y_train, model)
```

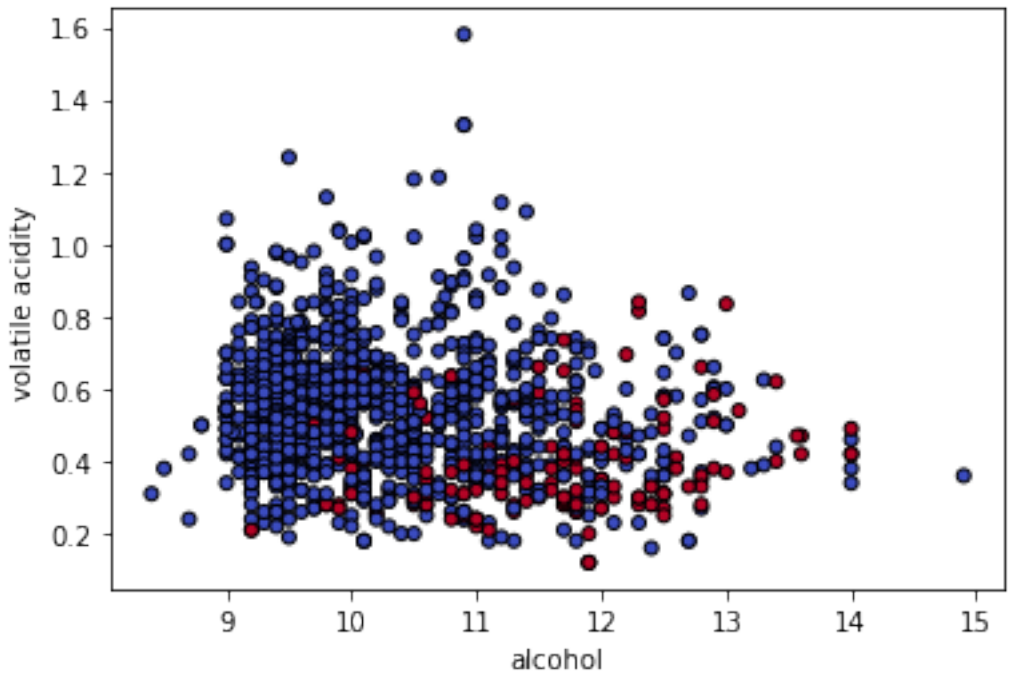
[28]:
```
(0.4307692307692308, 0.8419016787370058, 0.6630794139625461)
```

SMOTE SMOTE est une technique de suréchantillonnage où les échantillons synthétiques sont générés pour la classe minoritaire. Cet algorithme permet de surmonter le problème de surajustement posé par le suréchantillonnage aléatoire. Il se concentre sur l'espace des caractéristiques pour générer de nouvelles instances à l'aide de l'interpolation entre les instances positives qui se trouvent ensemble.

Nous utiliserons la représentation de la variable `volatile acidity` en fonction de la variable `alcohol` pour illustrer l'effet du rééchantionage.

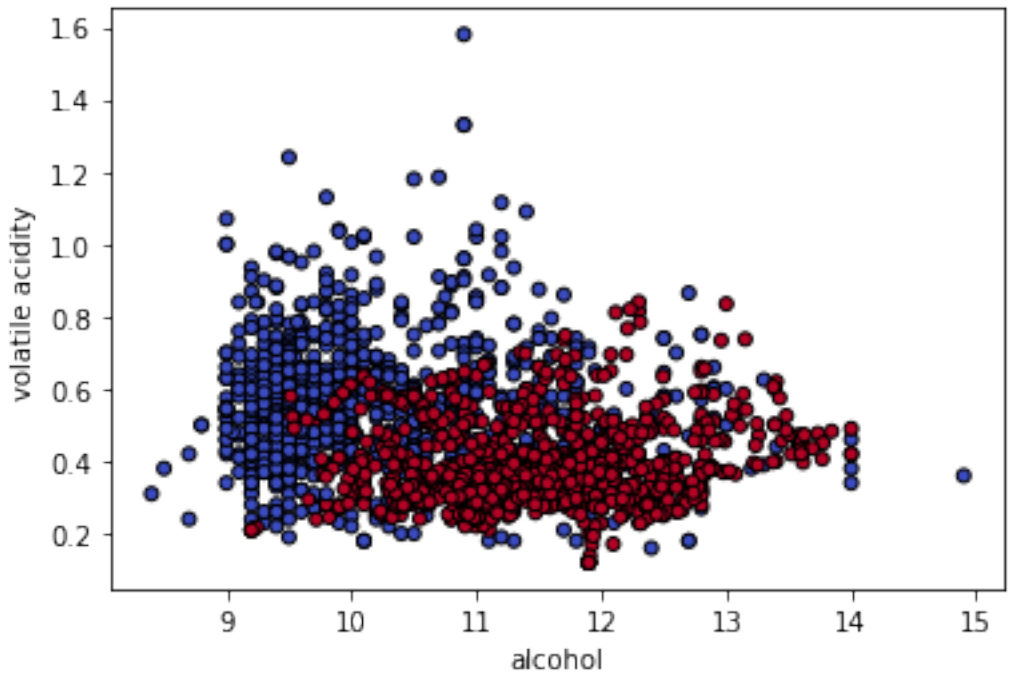
[29]:
```
print(f'Original dataset shape:\n{y_train.value_counts()}')
plt.scatter(X_train['alcohol'],X_train['volatile acidity'], marker='o',␣
 ↪c=y_train, s=25, edgecolor='k', cmap=plt.cm.coolwarm)
plt.xlabel('alcohol')
plt.ylabel('volatile acidity');
```

```
Original dataset shape:
False    1027
True      172
Name: good_wine, dtype: int64
```



[30]:
```
from imblearn.over_sampling import SMOTE
smote = SMOTE(sampling_strategy='auto', random_state=0, k_neighbors=4)
X_smoted, y_smoted = smote.fit_resample(X_train, y_train)
print(f'Resampled dataset shape:\n{y_smoted.value_counts()}')
plt.scatter(X_smoted['alcohol'],X_smoted['volatile acidity'], marker='o',␣
 ↪c=y_smoted, s=25, edgecolor='k', cmap=plt.cm.coolwarm)
plt.xlabel('alcohol')
plt.ylabel('volatile acidity');
```

```
Resampled dataset shape:
True     1027
False    1027
Name: good_wine, dtype: int64
```



[31]:
```
model = RandomForestClassifier(random_state=0)
model.fit(X_smoted, y_smoted)
y_pred = model.predict(X_test)
score_f1 = f1_score(y_test, y_pred)
score_f1_w = f1_score(y_test, y_pred, average='weighted')
score_roc_auc = roc_auc_score(y_test, y_pred)
print(f'F1: {round(score_f1,2)}, F1_weighted: {round(score_f1_w,2)}, ROC AUC:␣
↪{round(score_roc_auc,2)}')
```

```
F1: 0.57, F1_weighted: 0.88, ROC AUC: 0.83
```

2.4.2 Bilan

La méthode ayant obtenu les meilleurs résultats est la méthode SMOTE. Nous allons donc intégrer ces traitements à la nouvelle pipeline.

Il faudra cependant veiller à obtenir un bon résultat sur le testset. Si ce n'est pas le cas, nous serons alors très probablement en cas d'overfitting sur le trainset. Nous pourrons alors ré-essayer avec la méthode PolynomialFeatures associée à la méthode SelectKBest.

[32]:
```
def preprocessing(df, test=False):
    df['good_wine'] = df.quality > 6.5
    X = df.drop(['quality', 'good_wine'], axis=1)
    y = df['good_wine']
    if not test:
        smote = SMOTE(sampling_strategy='auto', random_state=0, k_neighbors=4)
```

```
        X, y = smote.fit_resample(X, y)
    return X, y

X_train, y_train = preprocessing(trainset)
X_test, y_test = preprocessing(testset, test=True)
```

Nous veillons à ne pas appliquer la méthode de ré-échantillonage sur le testset pour ne pas fausser les résultats!

2.5 Hyper-paramètres

2.5.1 RandomForest

Une forêt aléatoire est un méta-estimateur qui ajuste un certain nombre de classificateurs d'arbres de décision sur divers sous-échantillons de l'ensemble de données et utilise la moyenne pour améliorer la précision prédictive et contrôler l'ajustement excessif.

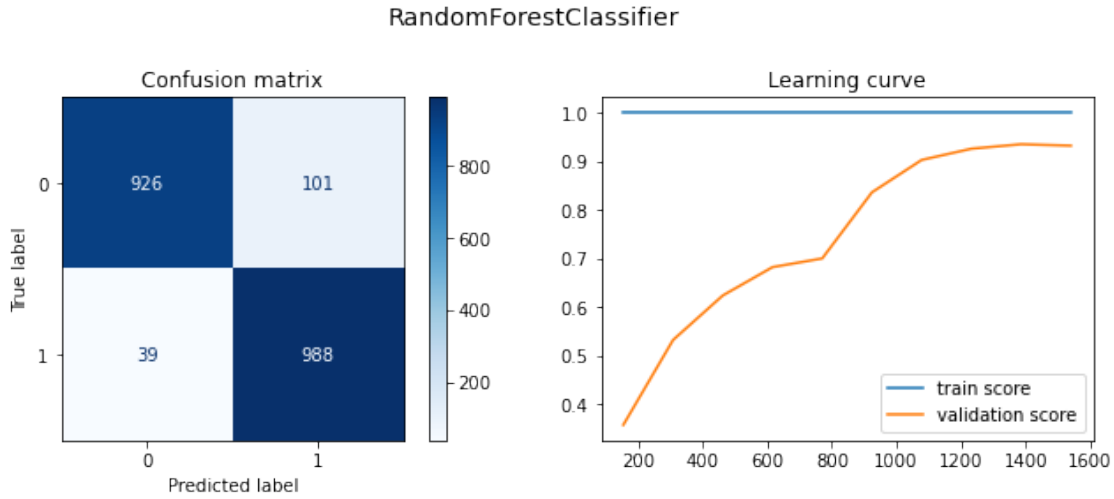
[33]:
```
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV
score = 'f1_weighted'
```

[34]:
```
params_rf = {
    'randomforestclassifier__class_weight':['balanced'],
    'randomforestclassifier__criterion' :['entropy','gini'],
    'randomforestclassifier__n_estimators':[50,100,200,400,600,800],
    'randomforestclassifier__max_depth':[None,3,6,10,14,18]
}
grid_rf = RandomizedSearchCV(RandomForest, params_rf, scoring=score,␣
 ↪cv=StratifiedKFold(4), n_iter=20)
grid_rf.fit(X_train, y_train)
print('Best params_rf:',grid_rf.best_params_)
print('\nBest score:', grid_rf.best_score_)
```

```
Best params_rf: {'randomforestclassifier__n_estimators': 600,
'randomforestclassifier__max_depth': None, 'randomforestclassifier__criterion':
'gini', 'randomforestclassifier__class_weight': 'balanced'}

Best score: 0.9317892864356356
```

[35]:
```
evaluation(X_train, y_train, grid_rf.best_estimator_,display_graph=True)
```

```
RandomForestClassifier
F_1: 0.934, F_1 weighted: 0.932, AUC: 0.932
```



[35]: (0.9338374291115312, 0.9317781523067696, 0.9318403115871471)

2.5.2 GBClassifier

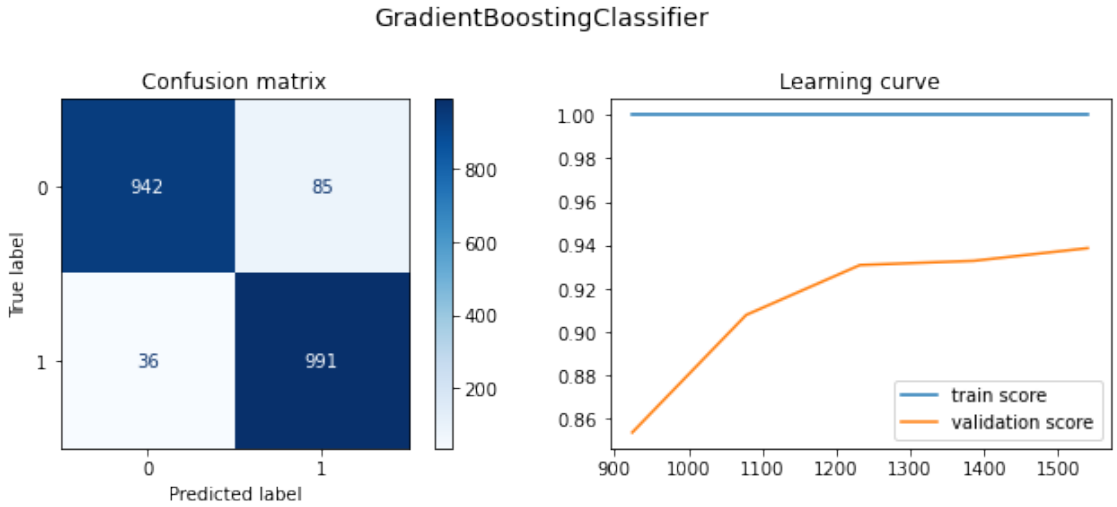
Le boosting de gradient est une technique d'apprentissage automatique pour les problèmes de régression et de classification, qui produit un modèle de prédiction sous la forme d'un ensemble de modèles de prédiction faibles, généralement des arbres de décision. Elle construit le modèle par étapes, comme le font les autres méthodes de boosting, et elle les généralise en permettant l'optimisation d'une fonction de perte différentiable arbitraire.

[36]:
```
params_gb={
    "gradientboostingclassifier__learning_rate"    : [0.05, 0.10, 0.15, 0.20, 0.
 ↪25, 0.30 ] ,
    "gradientboostingclassifier__max_depth"        : [ 3, 4, 5, 6, 8, 10, 12,␣
 ↪15],
}
grid_gb = RandomizedSearchCV(GBClassifier, params_gb, scoring=score, cv=4,␣
 ↪n_iter=100)
grid_gb.fit(X_train, y_train)
print('Best params_xgb:',grid_gb.best_params_)
print('\nBest score:', grid_gb.best_score_)
```

```
Best params_xgb: {'gradientboostingclassifier__max_depth': 8,
'gradientboostingclassifier__learning_rate': 0.3}

Best score: 0.9410662186810168
```

[37]:
```
evaluation(X_train, y_train, grid_gb.best_estimator_,display_graph=True)
```

```
GradientBoostingClassifier
F_1: 0.942, F_1 weighted: 0.941, AUC: 0.941
```



[37]: (0.9424631478839753, 0.9410570103509652, 0.9410905550146056)

2.5.3 KNN

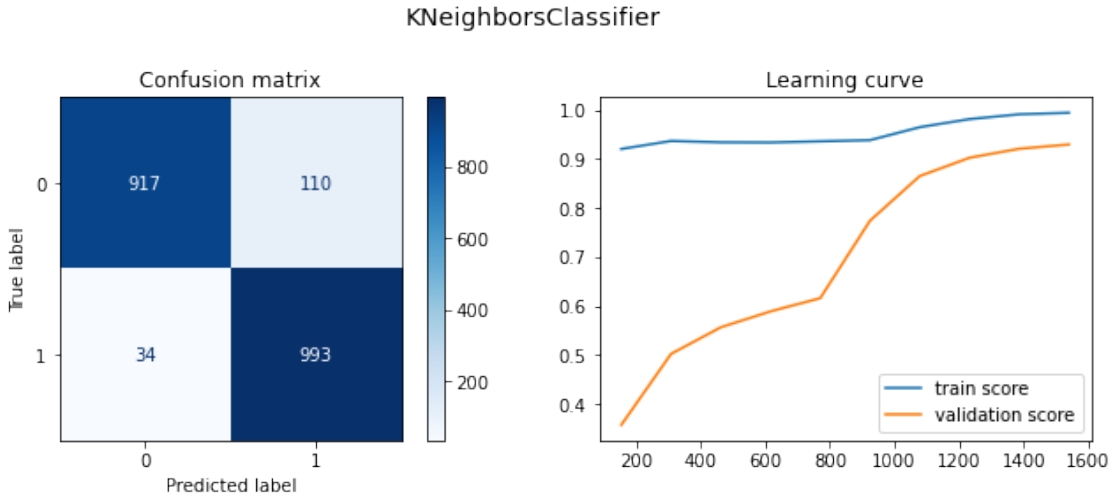
La classification K-Nearest Neighbors est calculée à partir d'un simple vote majoritaire des voisins les plus proches de chaque point : un point d'interrogation se voit attribuer la classe de données qui a le plus de représentants parmi les voisins les plus proches du point.

[38]:
```
params_knn = {
    'kneighborsclassifier__weights':['distance', 'uniform'],
    'kneighborsclassifier__n_neighbors':list(range(1,25))
}
grid_knn = GridSearchCV(KNN, params_knn, scoring=score)
grid_knn.fit(X_train, y_train)
print('Best params_knn:',grid_knn.best_params_)
print('\nBest score:', grid_knn.best_score_)
```

```
Best params_knn: {'kneighborsclassifier__n_neighbors': 2,
'kneighborsclassifier__weights': 'uniform'}

Best score: 0.93610519323039
```

[39]:
```
evaluation(X_train, y_train, grid_knn.best_estimator_,display_graph=True)
```

```
KNeighborsClassifier
F_1: 0.932, F_1 weighted: 0.93, AUC: 0.93
```



[39]: (0.9323943661971831, 0.9297967786496532, 0.9298928919182083)

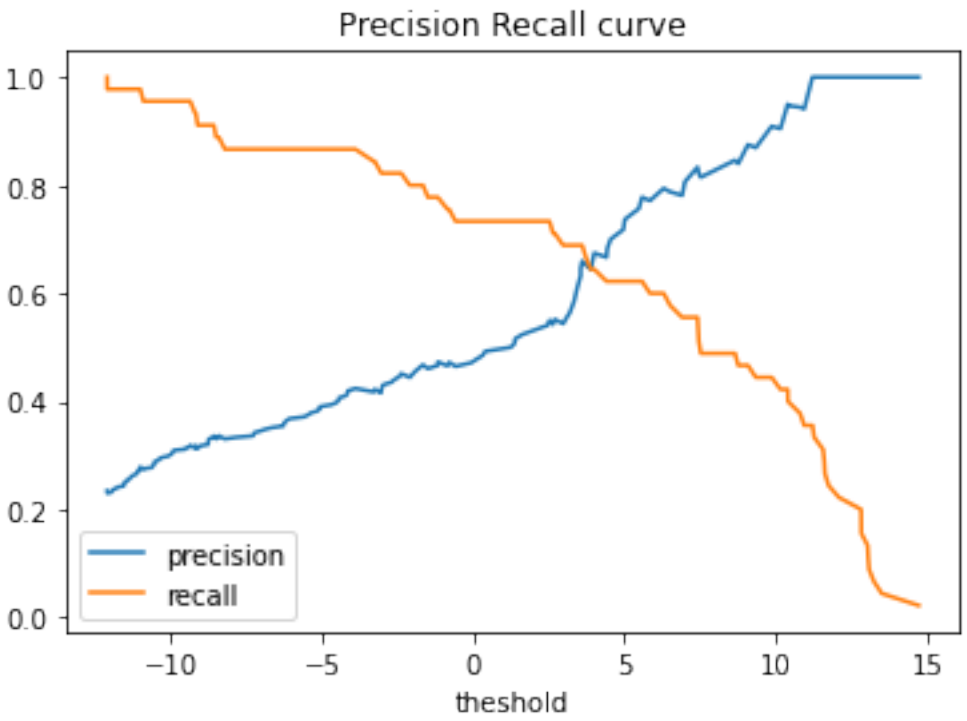
Les meilleurs résultats sont obtenus avec GBClassifier. Ce modèle obtient généralement de bons résultats sur ce genre de dataset car il repose sur l'entrainement de modèles en série: l'idée est que à chaque itération le modèle donne davantage de poids au points ayant reçu une mauvaise prédiction.

2.6 Precision Recall Curve

Il est intéressant de pourvoir avoir un impact sur le modèle concernant ses scores de spécificité et précision. En effet, on pourrait avoir envie de prioriser un indicateur plutôt qu'un autre. La courbe précision-rappel nous permet de faire cela en visualisant .

Dans notre cas, prioriser le rappel (= la sensibilité) revient à vouloir détecter un maximum de vins avec une bonne note quitte à y inclure plus de vins ayant une mauvaise note. À l'inverse prioriser la précision revient à minimiser le taux de faux négatifs. En contrepartie, on détectera moins de vins ayant une bonne note.

[40]:
```
from sklearn.metrics import precision_recall_curve
precision, recall, threshold = precision_recall_curve(y_test, grid_gb.
 ↪best_estimator_.decision_function(X_test))
plt.plot(threshold, precision[:-1], label='precision')
plt.plot(threshold, recall[:-1], label='recall')
plt.xlabel('theshold')
plt.legend()
plt.title('Precision Recall curve');
```



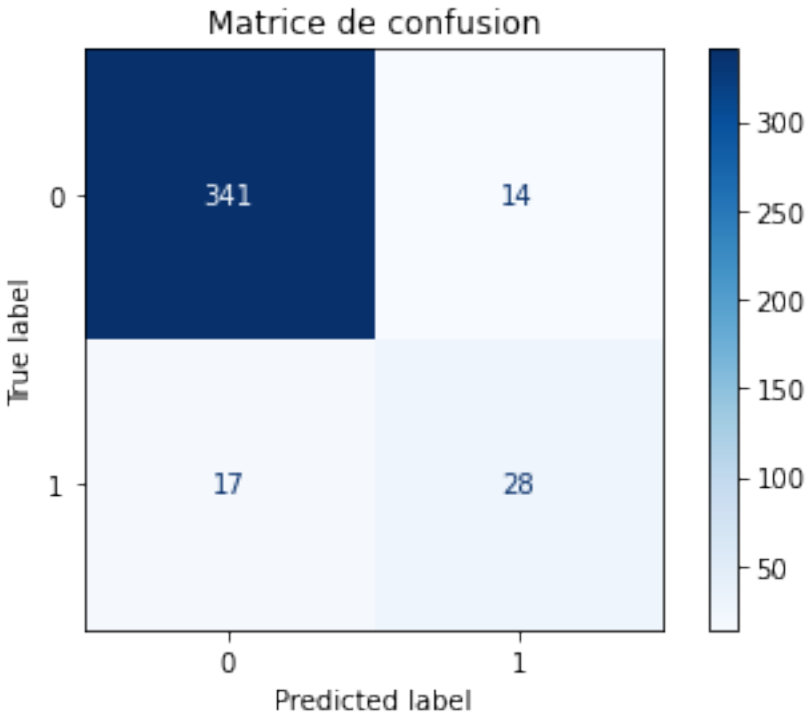
[41]:
```
def model_final(model, X, threshold):
    return model.decision_function(X) > threshold
```

Ainsi, si nous souhaitons un compromis entre la précision et le rappel, il faut choisir la valeur pour laquelle les courbes s'intersectent. Puis nous pouvons construire notre modèle final afin d'obtenir le score final.

[44]:
```
from sklearn.metrics import f1_score
y_pred = model_final(grid_gb.best_estimator_, X_test, 4)
f1_score_val = f1_score(y_test, y_pred)
f1_w_score = f1_score(y_test, y_pred, average='weighted')
roc_auc_score = f1_score(y_test, y_pred, average='weighted')
print("Final scores:")
print(f"F1: {f1_score_val}, F1_weighted: {f1_w_score}, ROC AUC:␣
 ↪{roc_auc_score}")
```

```
Final scores:
F1: 0.6436781609195403, F1_weighted: 0.9213268365817092, ROC AUC:
0.9213268365817092
```

[45]:
```
cm = confusion_matrix(y_test, y_pred)
ConfusionMatrixDisplay(cm).plot(cmap='Blues')
plt.title('Matrice de confusion');
```



2.7 Bilan de la classification

Nous avons donc réussi à obtenir de très bons scores avec notamment une ROC AUC et un score F_1 weighted de 0.92 pour le modèle final sur le testset. Ces résultats ont été possibles grâce à l'utilisation de la méthode de ré-échantillonage SMOTE couplée à l'optimisation des hyper-paramètres du modèle *GBClassifier*.