TP de Machine Learning : H4243 :

ALPOU Yannick BAILLEUX Henri

MOUSSET Maxime TABAKH Khalil

Nous avons pour ce TP mis en place des méthodes de Machine Learning pour prédire la qualité du vin grâce à un dataset de vins avec leur caractéristiques chimiques. Nous avons mis en place nous-même certaines méthodes et utilisé des bibliothèques pour d'autres.

1) La Régression Linéaire

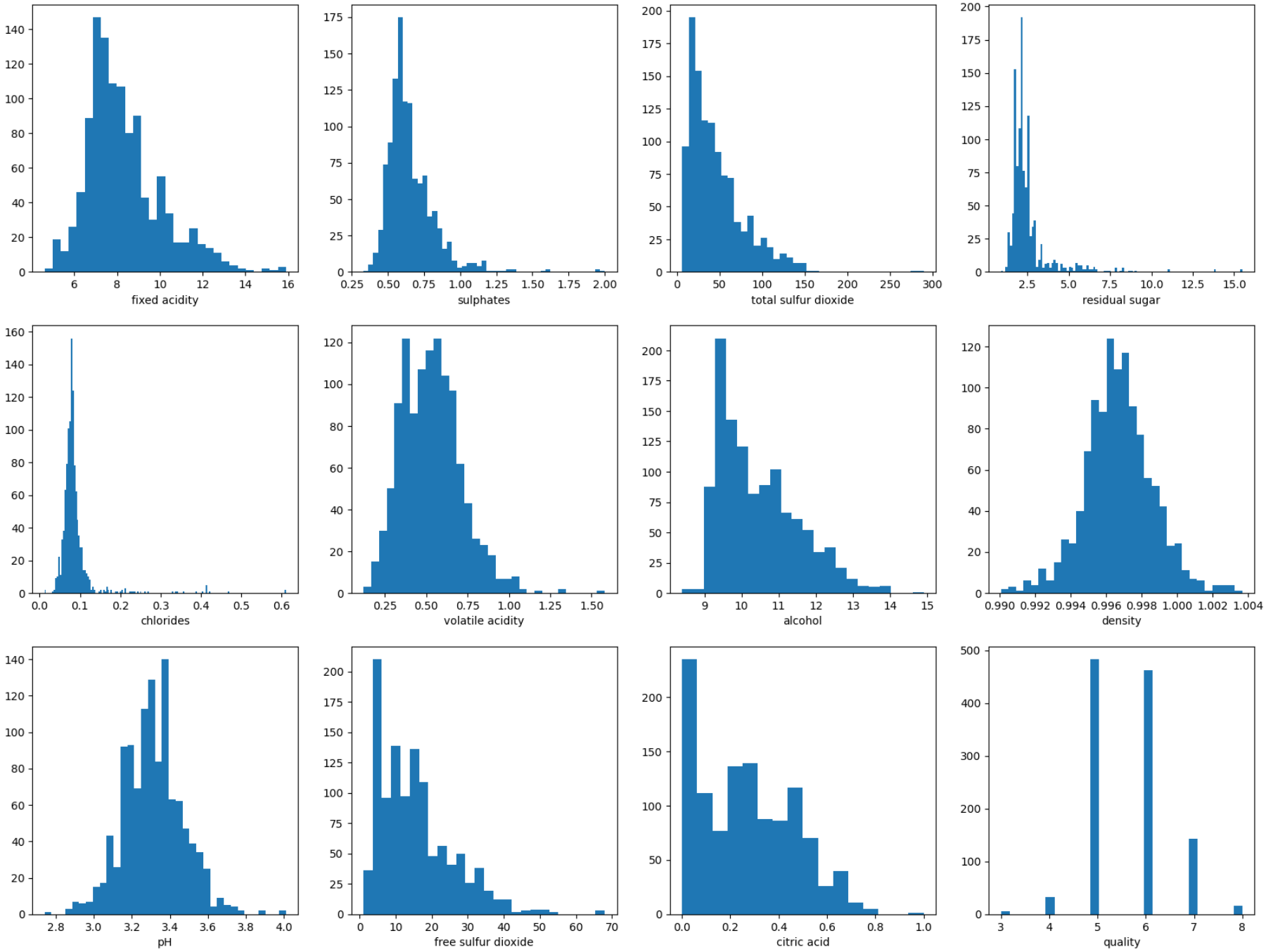
In [ ]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
df = pd.read_csv('WineQT.csv')
```

a) Visualisation et préparation des données

Nous commençons par une étape de visualisation des données. Cette étape nous permet de voir la distribution des différentes caractéristiques des vins.

In [ ]:
```
def plot_hists(df):
    fig, ax = plt.subplots(nrows=3, ncols=4, figsize=(20, 15))
    for n in range(12):
        i = n % 3
        j = n % 4
        ax[i, j].hist(df.iloc[:, n], bins='auto')
        ax[i, j].set_xlabel(df.columns[n])
```

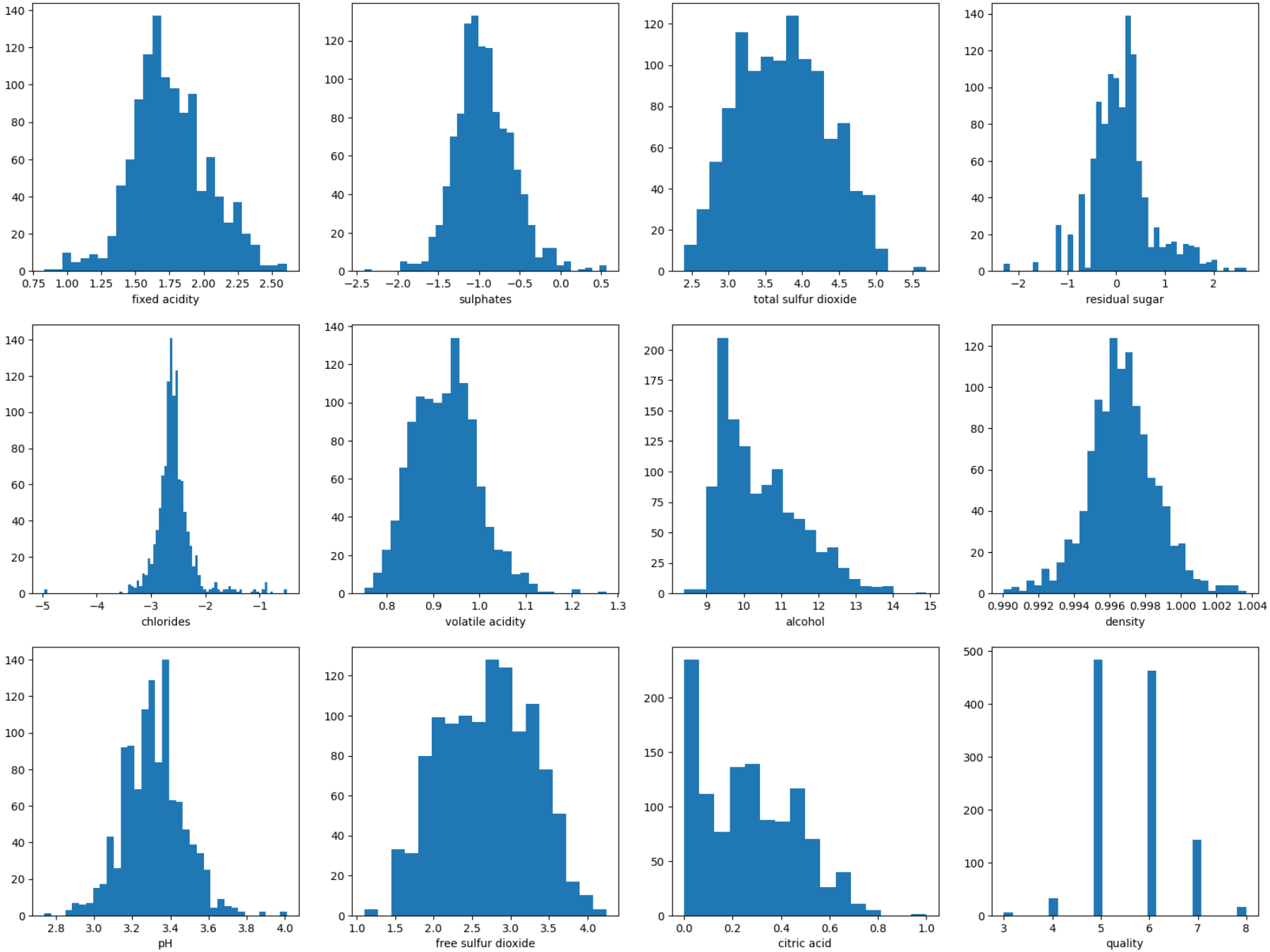
In [ ]:
```
plot_hists(df)
```

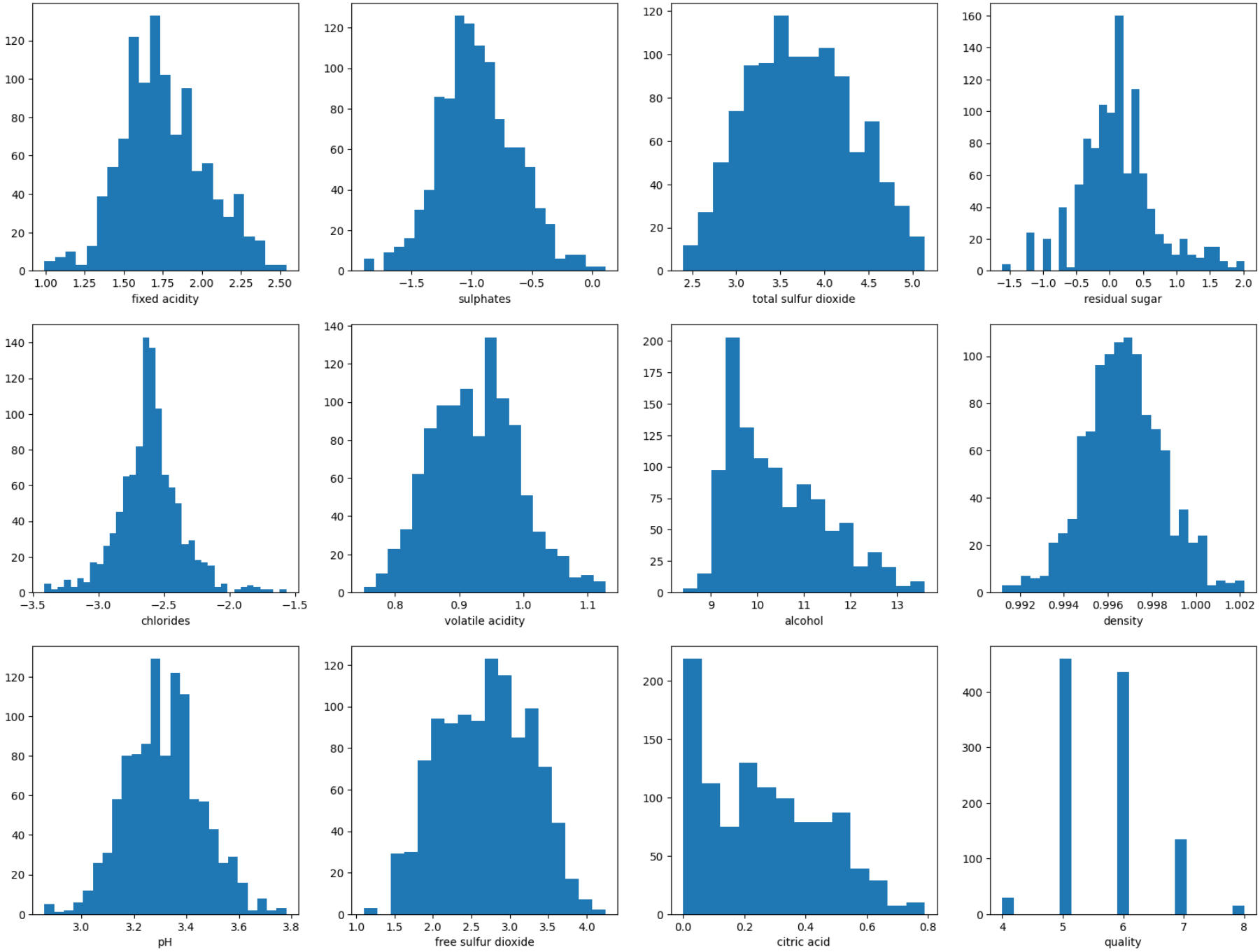

fixed acidity
sulphates
total sulfur dioxide
residual sugar
chlorides
volatile acidity
alcohol
density
pH
free sulfur dioxide
citric acid
quality

Grâce à cette visualisation, nous modifions les colonnes qui ne semblent pas avoir une distribution normale. On normalise avec un paramètre trouvé à la main pour chaque caractéristique et on standardise. Ci-dessous le résultat avant/après.

In [ ]:
```
#mettre entre 0 et 1
def normalize(df, property, parameter):
    df[property] = np.log(df[property] + parameter)

normalize(df, "fixed acidity", -2.3)
normalize(df, "sulphates", -0.24)
normalize(df, "total sulfur dioxide", 5)
normalize(df, "residual sugar", -1.1)
normalize(df, "chlorides", -0.005)
normalize(df, "volatile acidity", 2)
normalize(df, "free sulfur dioxide", 2)
plot_hists(df)

#On enlève les observations anormales
standardized = (df - df.mean()) / df.std()
standardized = standardized[(np.abs(standardized) < 3).all(axis=1)]
rows = np.setdiff1d(list(df.index), list(standardized.index))
df.drop(index=rows, inplace=True)
plot_hists(df)
```

```
/home/henri/.local/lib/python3.8/site-packages/pandas/core/series.py:726: RuntimeWarning: invalid value encountere
d in log
  result = getattr(ufunc, method)(*inputs, **kwargs)
```

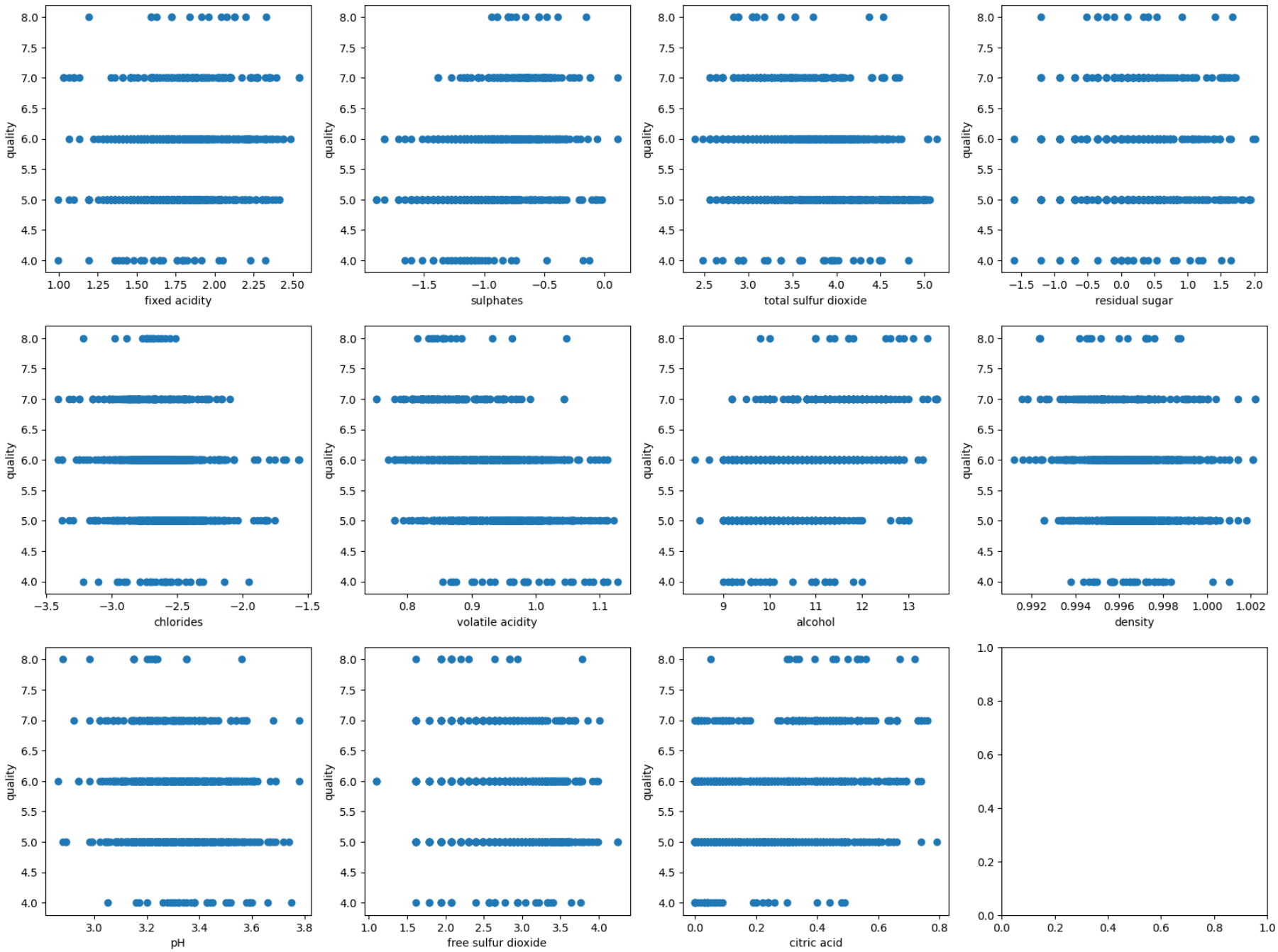

fixed acidity
sulphates
total sulfur dioxide
residual sugar
chlorides
volatile acidity
alcohol
density
pH
free sulfur dioxide
citric acid
quality


fixed acidity
sulphates
total sulfur dioxide
residual sugar
chlorides
volatile acidity
alcohol
density
pH
free sulfur dioxide
citric acid
quality

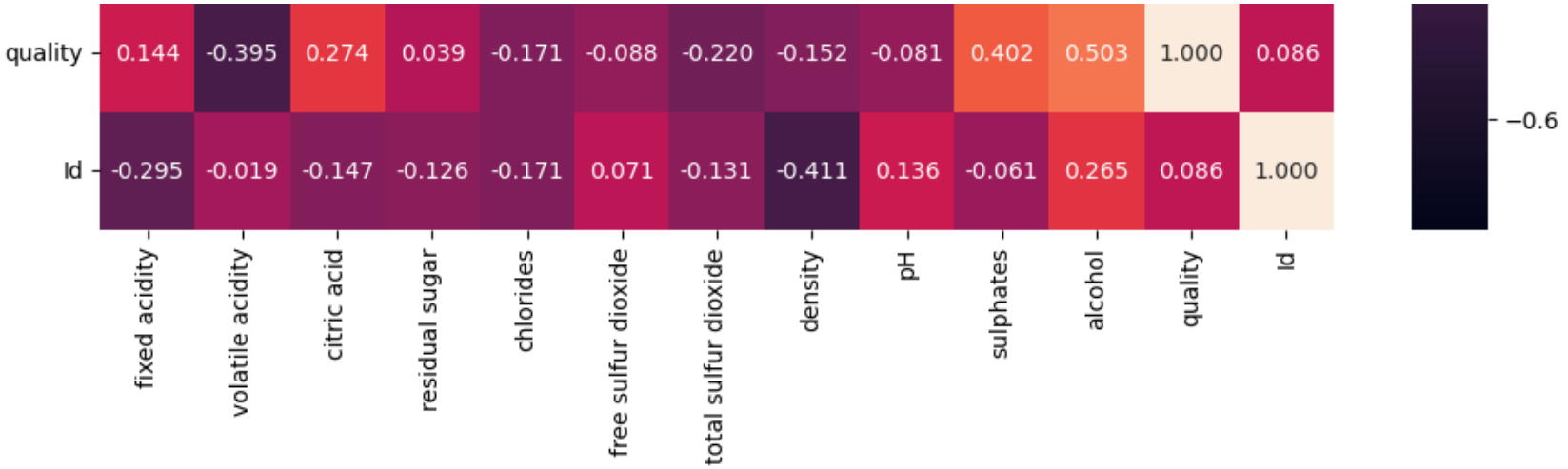
Souhaitant prédire la qualité, nous allons afficher les graphes des différentes caractéristiques en fonction de la qualité, ainsi qu'une matrice de corrélation linéaire.

In [ ]:
```
fig, ax = plt.subplots(nrows=3, ncols=4, figsize=(20, 15))
for n in range(11):
    i = n % 3
    j = n % 4
    ax[i, j].scatter(df.iloc[:, n], df['quality'])
    ax[i, j].set_xlabel(df.columns[n])
    ax[i, j].set_ylabel('quality')

import seaborn as sns

fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(12, 12))
correlation = df.corr()
sns.heatmap(correlation, annot=True, fmt='.3f')
```

Out[ ]:
```
<Axes: >
```


quality
fixed acidity
sulphates
total sulfur dioxide
residual sugar
chlorides
volatile acidity
alcohol
density
pH
free sulfur dioxide
citric acid


fixed acidity 1.000 -0.297 0.685 0.205 0.244 -0.195 -0.119 0.676 -0.724 0.193 -0.080 0.144 -0.295
volatile acidity -0.297 1.000 -0.586 0.043 0.113 0.063 0.136 -0.009 0.236 -0.363 -0.208 -0.395 -0.019
citric acid 0.685 -0.586 1.000 0.201 0.102 -0.115 -0.040 0.386 -0.519 0.318 0.157 0.274 -0.147
residual sugar 0.205 0.043 0.201 1.000 0.189 0.055 0.136 0.403 -0.067 0.047 0.151 0.039 -0.126
chlorides 0.244 0.113 0.102 0.189 1.000 0.005 0.102 0.396 -0.205 -0.012 -0.291 -0.171 -0.171
free sulfur dioxide -0.195 0.063 -0.115 0.055 0.005 1.000 0.774 -0.060 0.142 0.014 -0.077 -0.088 0.071
total sulfur dioxide -0.119 0.136 -0.040 0.136 0.102 0.774 1.000 0.117 0.038 -0.017 -0.250 -0.220 -0.131
density 0.676 -0.009 0.386 0.403 0.396 -0.060 0.117 1.000 -0.320 0.159 -0.464 -0.152 -0.411
pH -0.724 0.236 -0.519 -0.067 -0.205 0.142 0.038 -0.320 1.000 -0.029 0.167 -0.081 0.136
sulphates 0.193 -0.363 0.318 0.047 -0.012 0.014 -0.017 0.159 -0.029 1.000 0.226 0.402 -0.061
alcohol -0.080 -0.208 0.157 0.151 -0.291 -0.077 -0.250 -0.464 0.167 0.226 1.000 0.503 0.265
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4



Certaines caractéristiques sont particulièrement corrélées. Mais on ne détecte rien de très intéressant pour la qualité. Le coefficient R ne dépassant jamais 0.5, on va garder toutes les données et continuer.

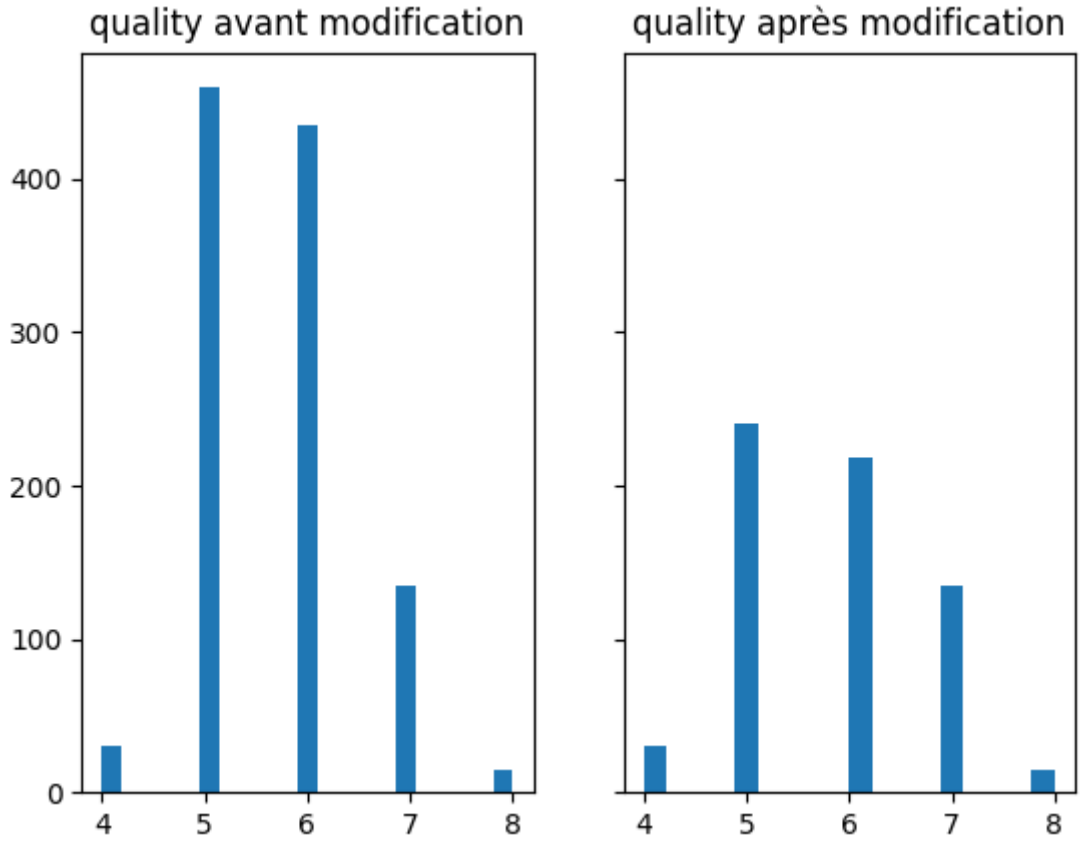
In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
# load the dataset

y = df['quality']
X= [df['fixed acidity'],  df['volatile acidity']  ,df['citric acid']  ,df['residual sugar'],  df['chlorides'],df['f
X=np.transpose(np.array(X))
y=np.asarray(y)
print(X.shape)
print(y.shape)
X_features = ['fixed acidity',  'volatile acidity'  ,'citric acid'  ,'residual sugar',  'chlorides','free sulfur di
nb_feature=len(X_features)

print("nombre d'observations X: ",len(X))
print("nombre d'observations y: ",len(y))
```

```
(1074, 10)
(1074,)
nombre d'observations X:  1074
nombre d'observations y:  1074
```

Nouvelle Modification du jeu de données: on a trop de vins de qualité égale à 5 ou 6... le modèle pourrait se concentrer sur des notes de 5 et 6 pour que la somme des erreures soit minimale. On va donc modifier le jeu de données pour avoir une répartition plus homogène.

In [ ]:
```
import random
supp=[]
for i in range(len(y)):
    if y[i]==5 or y[i]==6:
        rand=random.random()
        if(rand>0.5):
            supp.append(i)
y2=np.delete(y,supp)

X2=np.delete(X,supp,0)

#Plot des modifications
fig,ax=plt.subplots(1,2,sharey=True)
ax[0].hist(y, bins='auto',label="quality")
ax[0].set_title("quality avant modification")

ax[1].hist(y2, bins='auto',label="quality")
ax[1].set_title("quality après modification")
```

Out[ ]:
```
Text(0.5, 1.0, 'quality après modification')
```



Création des jeux de tests: Nous allons pour différents modèles faire de la validation croisée. Nous avons pour cela besoin d'un jeu de données d'entrainement, d'un jeu de validation pour le choix des modèles/hyperparamètres et d'un jeu de test du modèle choisi.

In [ ]:
```
#on créé les jeux de données
from sklearn.model_selection import train_test_split
X_train, X_tmp, y_train, y_tmp = train_test_split(X2, y2, test_size=0.4, random_state=42)
X_cv, X_test, y_cv, y_test = train_test_split(X_tmp, y_tmp, test_size=0.5, random_state=42)

print("nombre d'observations X: ",len(X_train))
print("nombre d'observations y: ",len(y_train))
```

```
nombre d'observations X:  382
nombre d'observations y:  382
```

Dernière étape : on normalise tous les jeux

In [ ]:
```
def zscore_normalize_features(X):
    mu     = np.mean(X, axis=0)                 # mu will have shape (n,)
    # find the standard deviation of each column/feature
    sigma  = np.std(X, axis=0)                  # sigma will have shape (n,)
    # element-wise, subtract mu for that column from each example, divide by std for that column
    X_norm = (X - mu) / sigma

    return (X_norm, mu, sigma)

 # normalize the original features
X_trainN, X_mu, X_sigma = zscore_normalize_features(X_train)
X_cvN, X_mu, X_sigma = zscore_normalize_features(X_cv)
X_testN, X_mu, X_sigma = zscore_normalize_features(X_test)
```

b) Première Méthode : Regression Linéaire par moindre carrés avec régularisation:

Nous allons pour cette première méthode traiter la question comme une régression linéaire. Avec quality = b1*fixe_acidity + b2* volatile_acidity+ ... + b10*sulphates. On va calculer les coefficients avec cette méthodes pour plusieurs Lambda avec le jeu X_train, puis comparer les résultats en fonction de l'hyperparamètre Lambda sur le jeu X_cv et évaluer finalement notre modèle avec le meilleur Lambda sur le jeu X_test.

In [ ]:
```
from numpy.linalg import inv

def moindre_carrees_regularise(X,y,Lambda):
    X=np.array(X)
    y=np.array(y)
    m=len(X[0])
    rterm=np.matmul(np.transpose(X), y)

    lterm=np.matmul(np.transpose(X),X) + Lambda * np.identity(m)

    beta=np.matmul(inv(lterm),rterm)

    return beta
```

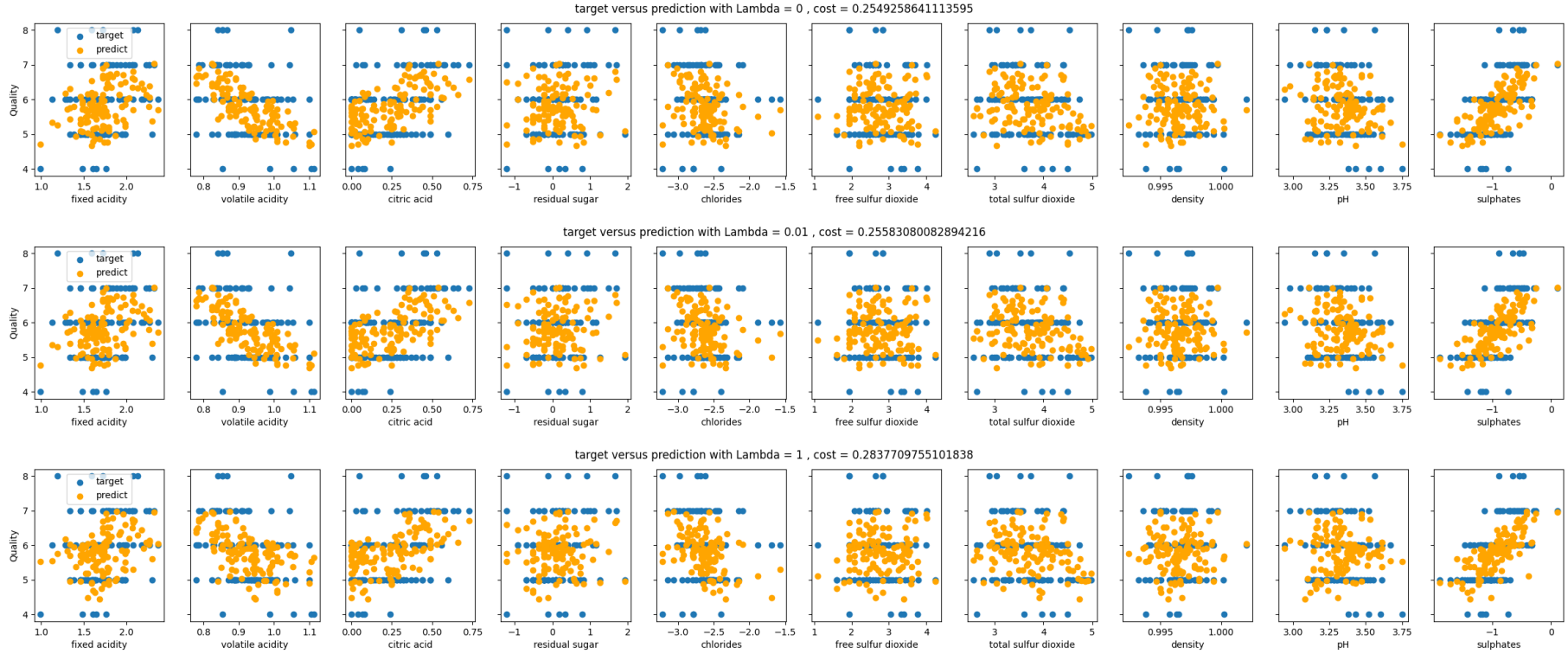
In [ ]:
```
Lambdas=[0,0.01,1,100]
betas=[]
for loop in range(len(Lambdas)):
    betas.append(moindre_carrees_regularise(X_train,y_train,Lambdas[loop]))
    print("Lambda = ",Lambdas[loop]," et Coefficients = ",betas[loop])
```

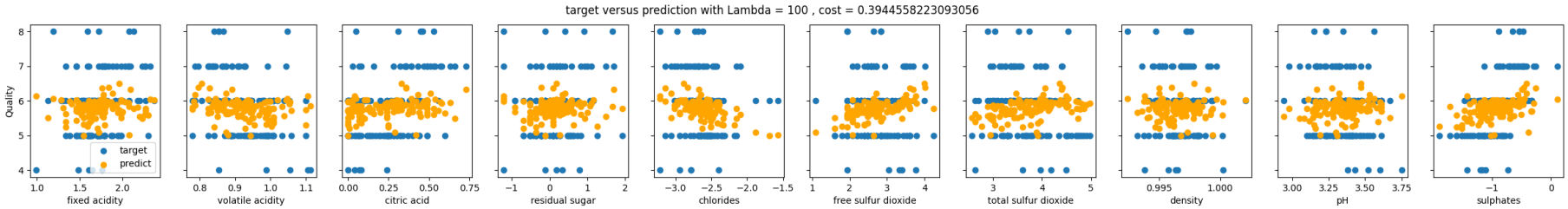
```
Lambda =  0  et Coefficients =  [-0.60640657 -1.91618945  0.82561506  0.10429171 -0.4463142   0.35798358
 -0.61654138 12.45406683 -0.87769952  1.02876821]
Lambda =  0.01  et Coefficients =  [-0.53126569 -1.83135111  0.84393815  0.10338238 -0.46782707  0.35736888
 -0.60829219 11.67425935 -0.73664043  1.01920231]
Lambda =  1  et Coefficients =  [ 0.43306223 -0.52626748  1.01827944  0.09313845 -0.75516322  0.34491482
 -0.49694017  1.83193726  0.95842441  0.90061192]
Lambda =  100  et Coefficients =  [ 0.51049957  0.10703829  0.23570422  0.04631569 -0.6378667   0.16637771
  0.07941815  0.22239853  0.66924061  0.20537541]
```

On a trouvé nos coefficients Beta, on va tester nos résultats:

In [ ]:
```
def compute_cost(X, y, beta):

    m = X.shape[0]

    cost = 0.0
    for i in range(m):
        f_wb_i = np.dot(X[i], beta)           #(n,)(n,) = scalar (see np.dot)
        cost = cost + (f_wb_i - y[i])**2      #scalar
    cost = cost / (2 * m)                     #scalar
    return cost
```

In [ ]:
```
#predict target using normalized features
for loop in range(len(Lambdas)):
    m = X_cv.shape[0]
    yp = np.zeros(m)
    for i in range(m):
        yp[i] = np.dot(X_cv[i], betas[loop])

        # plot predictions and targets versus original features
    fig,ax=plt.subplots(1,nb_feature,figsize=(30, 3),sharey=True)
    for i in range(len(ax)):
        ax[i].scatter([X_cv[:,i]],y_cv, label = 'target')
        ax[i].set_xlabel(X_features[i])
        ax[i].scatter([X_cv[:,i]],yp,color="orange", label = 'predict')
    ax[0].set_ylabel("Quality"); ax[0].legend()
    title="target versus prediction with Lambda = "+str(Lambdas[loop])+ " , cost = "+ str(compute_cost(X_cv,y_cv,be
    fig.suptitle(title)
    plt.show()
```





D'après ces tests, on garde lambda = 0, notre modèle n'est pas assez complexe pour nécessiter une régularisation. évaluons maintenant le modèle sur le jeu de test.

In [ ]:
```
print("cost final = ", str(compute_cost(X_test,y_test,betas[0])))
```

```
cost final =  0.26605026731051534
```

On a une erreur quadratique moyenne de 0,266. Visuellement le résultat ne parait pas très bon. La régression linéaire n'est pas adaptée car on doit trouver des valeurs discrètes.

c) Deuxième Méthode: la Descente de Gradient

Pour trouver de nouveau les coefficients de la régression linéaire, nous allons utiliser la méthode de la descente de gradient. Avec cette fois quality = w1*fixe_acidity + w2* volatile_acidity+ ... + w10*sulphates + b0

Cette méthode actualise de manière itérative les coefficients W et b0 pour minimiser le cout (= l'erreur quadratique moyenne de prédiction).

Elle est moins rapide que la regression précédente mais a l'avantage de pouvoir être utilisée pour n'importe quelle quality = g(W,b) (pas seulement un problème linéaire) Testons cette méthode d'abord pour une regression linéaire.

Codes :

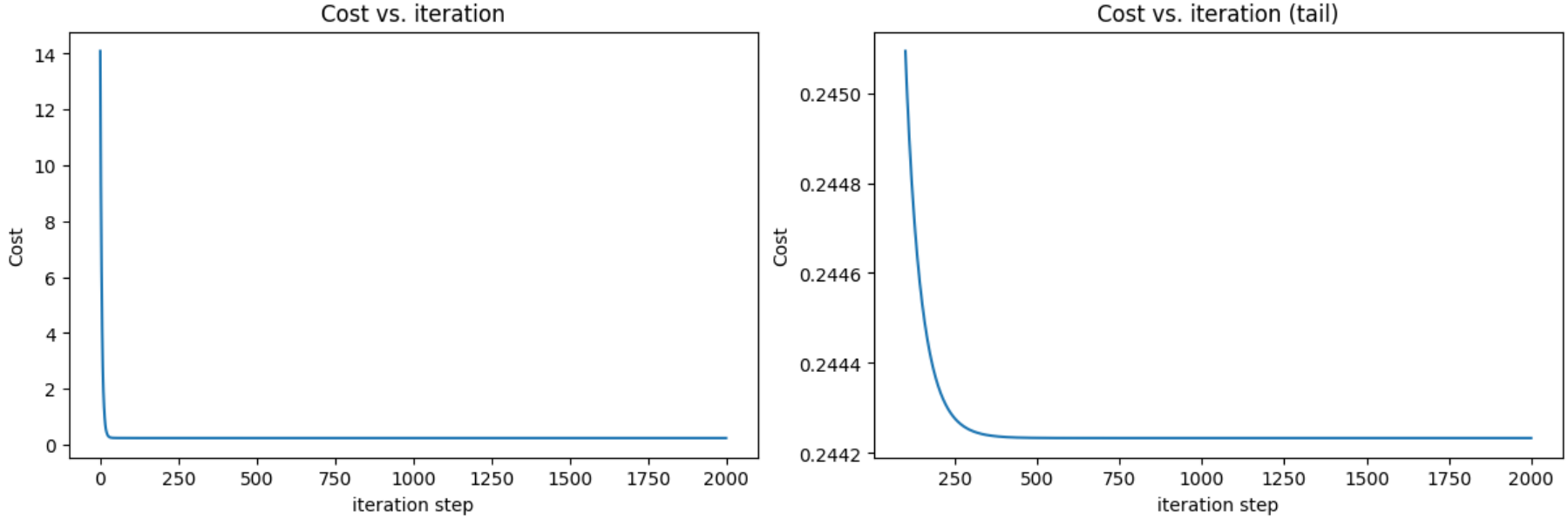
In [ ]:
```
import copy
import math


def compute_cost(X, y, w, b):

    m = X.shape[0]

    cost = 0.0
    for i in range(m):
        f_wb_i = np.dot(X[i], w) + b           #(n,)(n,) = scalar (see np.dot)
        cost = cost + (f_wb_i - y[i])**2       #scalar
    cost = cost / (2 * m)                      #scalar
    return cost

def compute_gradient(X, y, w, b):

    m,n = X.shape           #(number of examples, number of features)
    dj_dw = np.zeros((n,))
    dj_db = 0.

    for i in range(m):
        err = (np.dot(X[i], w) + b) - y[i]
        for j in range(n):
            dj_dw[j] = dj_dw[j] + err * X[i, j]
        dj_db = dj_db + err
    dj_dw = dj_dw / m
    dj_db = dj_db / m

    return dj_db, dj_dw

def gradient_descent(X, y, w_in, b_in, alpha, num_iters):

    # An array to store cost J and w's at each iteration primarily for graphing later
    J_history = []
    w = copy.deepcopy(w_in)  #avoid modifying global w within function
    b = b_in

    for i in range(num_iters):

        # Calculate the gradient and update the parameters
```

```
        # calculate the gradient and update the parameters
        dj_db,dj_dw = compute_gradient(X, y, w, b)   ##None

        # Update Parameters using w, b, alpha and gradient
        w = w - alpha * dj_dw               ##None
        b = b - alpha * dj_db               ##None

        # Save cost J at each iteration
        if i<100000:      # prevent resource exhaustion
            J_history.append( compute_cost(X, y, w, b))

        # Print cost every at intervals 10 times or as many iterations if < 10
        if i% math.ceil(num_iters / 10) == 0:

            print("Iteration", i,": Cost ", J_history[-1])

    return w, b, np.squeeze(J_history) #return final w,b and J history for graphing
```

Application:

In [ ]:
```
#Paramètres:
b_init = 0
w_init = np.array([ 0,0,0,0,0,0,0,0,0,0])
initial_w = np.zeros_like(w_init)
initial_b = 0.
print(X_trainN.shape)
print(y_train.shape)
#test calcul d'un coût
print(f"w_init shape: {w_init.shape}, b_init type: {type(b_init)}")
print("test cout : ",compute_cost(X_trainN, y_train, initial_w, initial_b))

# gradient descent settings
iterations = 2000
alpha =  0.1

# run gradient descent
w_final, b_final, J_hist = gradient_descent(X_trainN, y_train, initial_w, initial_b,alpha, iterations)
print(f" Coefficients b,w trouvés: {b_final},{w_final} ")
m,_ = X_train.shape

# plot cost versus iteration
fig, (ax1, ax2) = plt.subplots(1, 2, constrained_layout=True, figsize=(12, 4))
ax1.plot(J_hist)
ax2.plot(100 + np.arange(len(J_hist[100:])), J_hist[100:])
ax1.set_title("Cost vs. iteration");  ax2.set_title("Cost vs. iteration (tail)")
ax1.set_ylabel('Cost')             ;  ax2.set_ylabel('Cost')
ax1.set_xlabel('iteration step')   ;  ax2.set_xlabel('iteration step')
plt.show()
```

```
(374, 10)
(374,)
w_init shape: (10,), b_init type: <class 'int'>
test cout :  17.362299465240643
Iteration 0 : Cost  14.08996743622551
Iteration 200 : Cost  0.2443481295063563
Iteration 400 : Cost  0.24423591691349456
Iteration 600 : Cost  0.24423350061681215
Iteration 800 : Cost  0.24423344808600853
Iteration 1000 : Cost  0.24423344694386442
Iteration 1200 : Cost  0.24423344691903137
Iteration 1400 : Cost  0.24423344691849116
Iteration 1600 : Cost  0.24423344691847965
Iteration 1800 : Cost  0.24423344691847904
 Coefficients b,w trouvés: 5.820855614973543,[ 0.28087615 -0.1889504   0.05920374  0.14462724 -0.02675343  0.06247
141
 -0.15266377 -0.40236769  0.08360496  0.27332941]
```



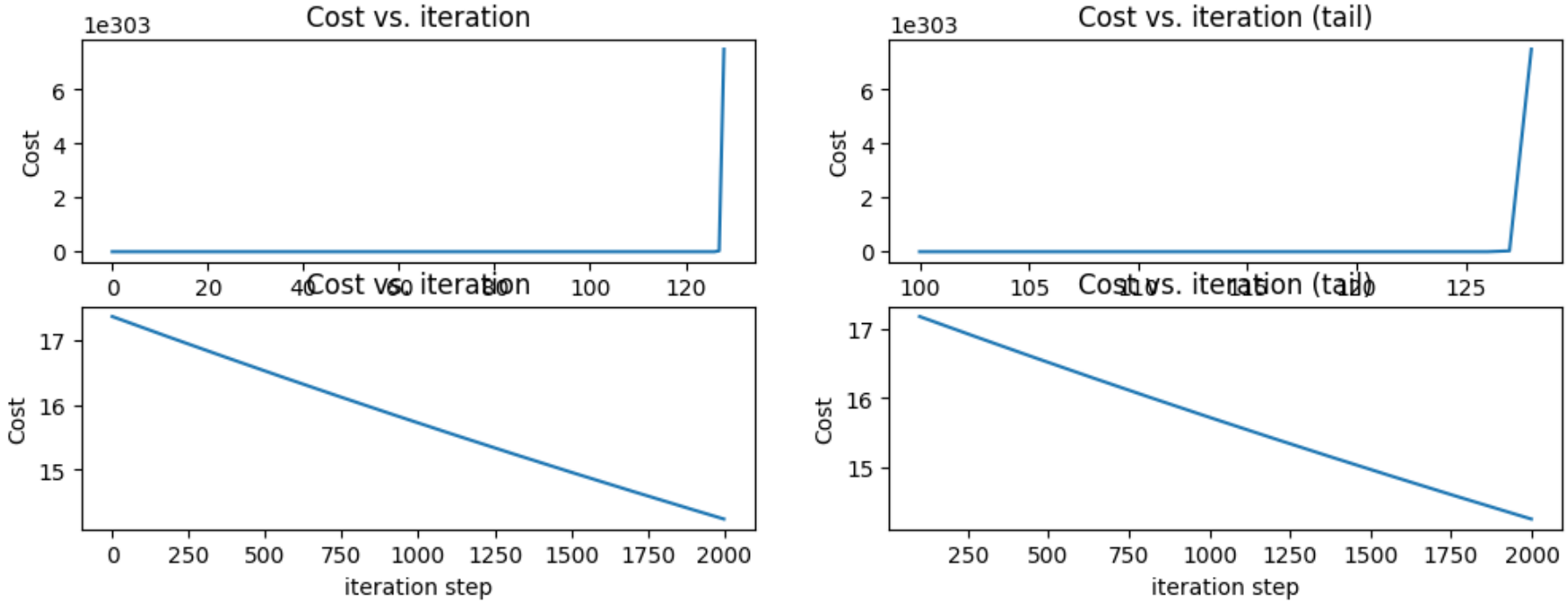
La descente de gradient nous donne les coefficients de la régression linéaire qui minimisent l'erreur. On voit ci-dessus l'évolution de l'erreur moyenne en fonction de l'itération.

Voyons l'influence de l'Hyperparamètre Alpha : Alpha est l'hyperparamètre de la descente de gradient qui met à jour les coefficients de la regression à chaque itération. Plus alpha sera grand, plus l'ajustement à chaque itération sera important.

In [ ]:
```
alpha1 = 5.0
alpha2 = 5.0e-5
fig, [[ax1,ax2],[ax3,ax4]]= plt.subplots(nrows=2, ncols=2, figsize=(12, 4))
w_final1, b_final1, J_hist1 = gradient_descent(X_trainN, y_train, initial_w, initial_b,alpha1, iterations)
w_final2, b_final2, J_hist2 = gradient_descent(X_trainN, y_train, initial_w, initial_b,alpha2, iterations)



ax1.plot(J_hist1)
ax2.plot(100 + np.arange(len(J_hist1[100:])), J_hist1[100:])
ax1.set_title("Cost vs. iteration");  ax2.set_title("Cost vs. iteration (tail)")
ax1.set_ylabel('Cost')             ;  ax2.set_ylabel('Cost')
ax1.set_xlabel('iteration step')   ;  ax2.set_xlabel('iteration step')

ax3.plot(J_hist2)
ax4.plot(100 + np.arange(len(J_hist2[100:])), J_hist2[100:])
ax3.set_title("Cost vs. iteration");  ax4.set_title("Cost vs. iteration (tail)")
ax3.set_ylabel('Cost')             ;  ax4.set_ylabel('Cost')
ax3.set_xlabel('iteration step')   ;  ax4.set_xlabel('iteration step')
plt.show()
```

```
Iteration 0 : Cost  284.0365019780978
```

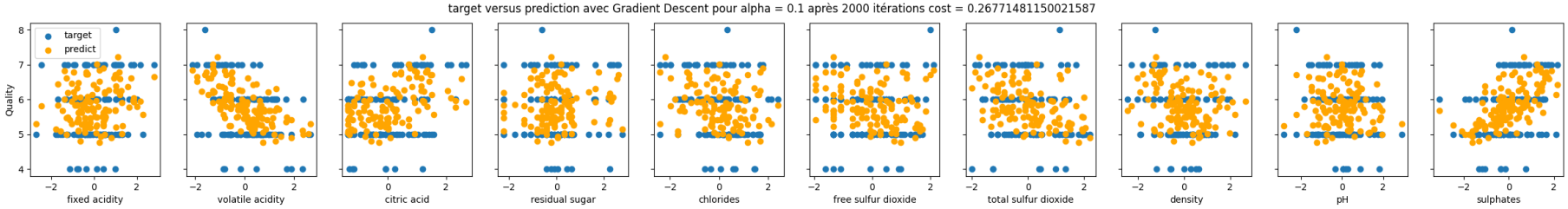
```
/tmp/ipykernel_35587/1263849773.py:22: RuntimeWarning: overflow encountered in double_scalars
  cost = cost + (f_wb_i - y[i])**2       #scalar
Iteration 200 : Cost  inf
/tmp/ipykernel_35587/1263849773.py:46: RuntimeWarning: overflow encountered in double_scalars
  dj_dw[j] = dj_dw[j] + err * X[i, j]
Iteration 400 : Cost  nan
Iteration 600 : Cost  nan
Iteration 800 : Cost  nan
Iteration 1000 : Cost  nan
Iteration 1200 : Cost  nan
Iteration 1400 : Cost  nan
Iteration 1600 : Cost  nan
Iteration 1800 : Cost  nan
Iteration 0 : Cost  17.360575528683576
Iteration 200 : Cost  17.01929376799513
Iteration 400 : Cost  16.6848914508912
Iteration 600 : Cost  16.35722619303301
Iteration 800 : Cost  16.036158762632734
Iteration 1000 : Cost  15.72155299899876
Iteration 1200 : Cost  15.41327573382292
Iteration 1400 : Cost  15.111196715086663
Iteration 1600 : Cost  14.815188533470177
Iteration 1800 : Cost  14.525126551154262
```



Le premier alpha est trop grand, on saute l'optimum. Le second est trop petit, on ne l'atteint qu'après un trop grand nombre d'itérations...

Test du Résultat: Le premier alpha testé est le meilleur trouvé. On a donc nos coefficients permettant de minimiser l'erreur, visualisons les prédictions calculées.

In [ ]:
```
#predict target using normalized features
m = X_cvN.shape[0]
yp = np.zeros(m)
for i in range(m):
    yp[i] = np.dot(X_testN[i], w_final) + b_final

    # plot predictions and targets versus original features
fig,ax=plt.subplots(1,nb_feature,figsize=(30, 3),sharey=True)
for i in range(len(ax)):
    ax[i].scatter([X_testN[:,i]],y_test, label = 'target')
    ax[i].set_xlabel(X_features[i])
    ax[i].scatter([X_testN[:,i]],yp,color="orange", label = 'predict')
ax[0].set_ylabel("Quality"); ax[0].legend()
title="target versus prediction avec Gradient Descent pour alpha = 0.1 après 2000 itérations cost = "+ str(compute_
fig.suptitle(title)
plt.show()
```



Le coût est similaire, un petit peu plus élevé. Nous allons maintenant utiliser la méthode de la descente de gradient pour un nouveau type de fonctions qualité = g(W,B) avec g : un ensemble de fonctions g_y(W,B,x) donnant la probabilité d'appertenance d'un vin x à une qualité y. C'est l'algorithme Softmax.

2) Multi-Classification avec Softmax

a) Préparation des données (comme précédemment)

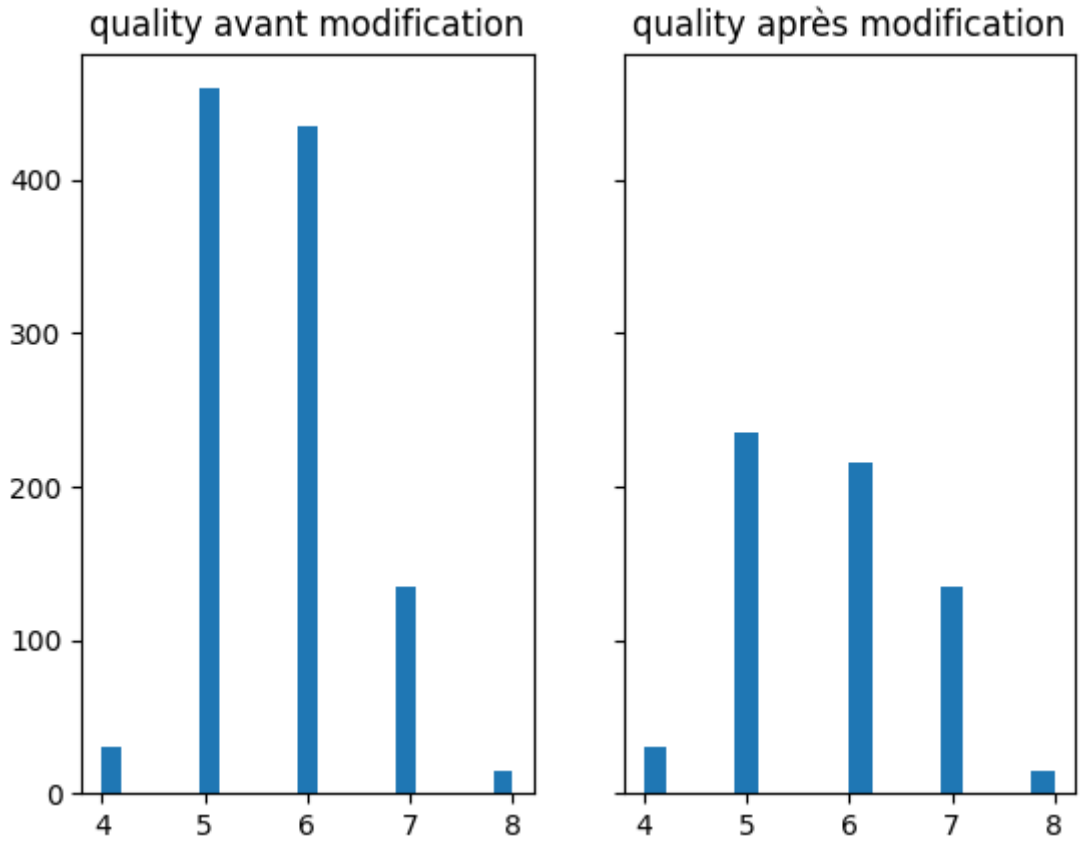
In [ ]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import random
df = pd.read_csv('WineQT.csv')
import numpy as np
import matplotlib.pyplot as plt

def plot_hists(df):
    fig, ax = plt.subplots(nrows=3, ncols=4, figsize=(20, 15))
    for n in range(12):
        i = n % 3
        j = n % 4
        ax[i, j].hist(df.iloc[:, n], bins='auto')
        ax[i, j].set_xlabel(df.columns[n])

#On normalise : mettre entre 0 et 1
def normalize(df, property, parameter):
    df[property] = np.log(df[property] + parameter)


normalize(df, "fixed acidity", -2.3)
normalize(df, "sulphates", -0.24)
normalize(df, "total sulfur dioxide", 5)
normalize(df, "residual sugar", -1.1)
normalize(df, "chlorides", -0.005)
normalize(df, "volatile acidity", 2)
normalize(df, "free sulfur dioxide", 2)
#plot_hists(df)

standardized = (df - df.mean()) / df.std()
standardized = standardized[(np.abs(standardized) < 3).all(axis=1)]
rows = np.setdiff1d(list(df.index), list(standardized.index))
df.drop(index=rows, inplace=True)
#plot_hists(df)

import numpy as np
import matplotlib.pyplot as plt

#Préparation des données
y = df['quality']
```

```
y = df[ 'quality' ]
X= [df['fixed acidity'],  df['volatile acidity']  ,df['citric acid']  ,df['residual sugar'],  df['chlorides'],df['f
X=np.transpose(np.array(X))
y=np.asarray(y)
print(X.shape)
print(y.shape)
X_features = ['fixed acidity',  'volatile acidity'  ,'citric acid'  ,'residual sugar',  'chlorides','free sulfur di
nb_feature=len(X_features)


#on supprime aleatoirement des valeurs de notes 5 et 6 (diviser par 2)
supp=[]
for i in range(len(y)):
    if y[i]==5 or y[i]==6:
        rand=random.random()
        if(rand>0.5):
            supp.append(i)
y2=np.delete(y,supp)

X2=np.delete(X,supp,0)

#Plot des modifications
fig,ax=plt.subplots(1,2,sharey=True)
ax[0].hist(y, bins='auto',label="quality")
ax[0].set_title("quality avant modification")

ax[1].hist(y2, bins='auto',label="quality")
ax[1].set_title("quality après modification")

#on créé les jeux de données
from sklearn.model_selection import train_test_split
X_train, X_tmp, y_train, y_tmp = train_test_split(X2, y2, test_size=0.4, random_state=42)
X_cv, X_test, y_cv, y_test = train_test_split(X_tmp, y_tmp, test_size=0.5, random_state=42)

def zscore_normalize_features(X):
    mu     = np.mean(X, axis=0)                 # mu will have shape (n,)
    # find the standard deviation of each column/feature
    sigma  = np.std(X, axis=0)                  # sigma will have shape (n,)
    # element-wise, subtract mu for that column from each example, divide by std for that column
    X_norm = (X - mu) / sigma

    return (X_norm, mu, sigma)
```

```
    return (X_norm, mu, sigma)

 # normalize the original features
X_train, X_mu, X_sigma = zscore_normalize_features(X_train)
X_cv, X_mu, X_sigma = zscore_normalize_features(X_cv)
X_test, X_mu, X_sigma = zscore_normalize_features(X_test)
```

```
/home/henri/.local/lib/python3.8/site-packages/pandas/core/series.py:726: RuntimeWarning: invalid value encountere
d in log
  result = getattr(ufunc, method)(*inputs, **kwargs)
(1074, 10)
(1074,)
```



b) Méthode Softmax :

Nous allons cette fois utiliser une descente de gradient, mais à la place de trouver des coefficients d'une régression linéaire prédisant une valeur de qualité continue, nous allons trouver les coefficients de 9 fonctions pour chaque valeur de qualité (entre 0 et 8). Ces fonctions donnent pour une observation une probabilité d'appartenir à cette qualité. Une prédiction de qualité d'un vin X est donc la probabilité maximale d'appartenance à une classe de qualité. L'erreur correspond à la proportion d'observations X mal classées.

Codes :

In [ ]:

```
import copy
import math

def sigmoid(z):

    g = 1/(1+np.exp(-z))

    return g

def compute_cost_softmax(X, y, W, B):
    m, n = X.shape
    nb_f=W.shape[0]

    loss_sum = 0

  # on calcule tous les zj=fwb
   # Loop over each training example
    for i in range(m):
      f_WB=np.zeros(nb_f)
      # Loop over each class
      for loop in range(nb_f):
        z_wb = 0
       # Loop over each feature
        for j in range(n):

             z_wb_ij = W[loop,j]*X[i,j]
             z_wb += z_wb_ij
        z_wb += B[loop]

        f_WB[loop] = np.exp(z_wb)#=e(Zij), #à diviser par sumezi pour avoir probabilité que y==loop
      sumEzi=np.sum(f_WB)
      loss_sum += np.log(f_WB[y[i]]/sumEzi) # on ajoute log(a_i) if y=i

    total_cost = -(1 / m) * loss_sum  #cost = -1/m(sum(sum{y==j}log(ezj/sum(ezk))))

    return total_cost


def compute_gradient_softmax(X, y, W, B):
```

```
    m, n = X.shape
    nb_f=W.shape[0]

    dJ_DW = np.zeros((nb_f,n))                          #(n,)
    dJ_DB = np.zeros((nb_f))

    for i in range(m):
      f_WB=np.zeros(nb_f)
      # Loop over each class
      for loop in range(nb_f):
        z_wb = 0
       # Loop over each feature
        for j in range(n):

             z_wb_ij = W[loop,j]*X[i,j]
             z_wb += z_wb_ij
        z_wb += B[loop]
        f_WB[loop] = np.exp(z_wb)#=e(Zij)

      sumEzi=np.sum(f_WB)
      f_WB=f_WB/sumEzi # tableau des probabilité que y==loop

      #on calcule la dérivé
      for loop in range(nb_f):
        err_loop  = f_WB[loop]  - (loop==y[i])         #scalar, proba que y = loop  - (1 ou 0)(si y ==loop)
        for j in range(n):
            dJ_DW[loop,j] = dJ_DW[loop,j] + err_loop * X[i,j]      #scalar
        dJ_DB[loop] = dJ_DB[loop] + err_loop
    dJ_DW = dJ_DW/m                                    #(n,)
    dJ_DB = dJ_DB/m                                    #scalar

    return dJ_DB, dJ_DW

def gradient_descent_softmax(X, y, W_in, B_in, alpha, num_iters):

    # An array to store cost J and w's at each iteration primarily for graphing later
    J_history = []
    W = copy.deepcopy(W_in)  #avoid modifying global w within function
    B = B_in
    m, n = X.shape
    nb_f=W.shape[0]
```

```
    nb_f=W.shape[0]

    for i in range(num_iters):
        # Calculate the gradient and update the parameters
        dJ_DB, dJ_DW = compute_gradient_softmax(X, y, W, B)

        # Update Parameters using w, b, alpha and gradient
        W = W - alpha * dJ_DW
        B = B - alpha * dJ_DB

        # Save cost J at each iteration
        if i<100000:      # prevent resource exhaustion
            J_history.append( compute_cost_softmax(X, y, W, B) )

        # Print cost every at intervals 10 times or as many iterations if < 10
        if i% math.ceil(num_iters / 5) == 0:

            print(f"Iteration {i:4d}: Cost {J_history[-1]}   ")

    return W, B, J_history         #return final w,b and J history for graphing
```

Application :

Choix de l'hyperparametre alpha entre alpha = {0.01;0.1;1}. On va utiliser pour cela la cross validation en entraînant sur X_train et en testant sur X_cv. On lance la descente de gradient et on calcule le pourcentage de prédictions correctes par Alpha sur les différents jeux.

In [ ]:
```
nbClasses=9
np.random.seed(1)
initial_W = np.random.rand(nbClasses,X_train.shape[1]) - 0.5
initial_B = np.ones(nbClasses)*0.5

# Gradient descent settings
iterations = 2000
alpha1 = 0.01 ; alpha2=0.1 ; alpha3=1

W,B, J_history = gradient_descent_softmax(X_train, y_train, initial_W, initial_B, alpha1, iterations)
W2,B2, J_history2 = gradient_descent_softmax(X_train, y_train, initial_W, initial_B, alpha2, iterations)
W3,B3, J_history3 = gradient_descent_softmax(X_train, y_train, initial_W, initial_B, alpha3, iterations)

print("W_alpha1 = ",W, " B_alpha1 = ",B)
print("W_alpha2 = ",W2, " B_alpha2 = ",B2)
print("W_alpha3 = ",W3, " B_alpha3 = ",B3)
```

```
Iteration    0: Cost 2.4843163758563644
Iteration  400: Cost 1.482174029511714
Iteration  800: Cost 1.2462442035892285
Iteration 1200: Cost 1.1548402069288954
Iteration 1600: Cost 1.1062944170077342
Iteration    0: Cost 2.4424552264414383
Iteration  400: Cost 1.0132889702077512
Iteration  800: Cost 0.9799682121423916
Iteration 1200: Cost 0.9677841634503018
Iteration 1600: Cost 0.9610437941954085
Iteration    0: Cost 2.058860649278224
Iteration  400: Cost 0.9469709960508422
Iteration  800: Cost 0.9413999775927384
Iteration 1200: Cost 0.939231105787295
Iteration 1600: Cost 0.938045379404537
W_alpha1 =  [[ 0.02998063  0.10615956 -0.27880275  0.01191312 -0.20539767 -0.13033644
  -0.00589727 -0.00680416 -0.13958837  0.11790825]
 [-0.0752136   0.08986021 -0.20726222  0.23527053 -0.36334896  0.12962421
  -0.06058096  0.04655931 -0.28206452 -0.08989226]
 [ 0.21215108  0.34631873 -0.17588541  0.16862195  0.075031    0.33727115
  -0.31604683 -0.42120519 -0.18556885  0.23705742]
 [-0.56942101  0.05038511  0.18223207 -0.01428231 -0.01487042 -0.15372565
   0.16076516  0.16375792 -0.2578703   0.09667047]
 [ 0.37534696  0.62931389 -0.45612083  0.08019775 -0.22403765 -0.15982986
   0.05808658 -0.15949401  0.22062599 -0.24626982]
 [-0.3684185   0.21206631 -0.14708188 -0.14942879  0.15328915 -0.39395392
   0.7804783   0.48367137 -0.15798598 -0.79083042]
 [-0.08081785 -0.0630514  -0.14401434  0.01260092 -0.12410903 -0.01777083
  -0.06192403  0.19103638  0.00387618  0.22932694]
 [ 0.44649759 -0.53941087  0.07600616  0.45218083 -0.30879808 -0.14384291
  -0.02700534 -0.77456891  0.24567032  0.86485245]
 [ 0.1624731   0.06547575  0.20350059 -0.26584119 -0.43299122  0.10388554
  -0.15065792  0.02281605  0.01410786  0.21041769]]  B_alpha1 =  [-0.36748835 -0.35140557 -0.39928249 -0.3608933
0.29936278  2.18958284
  2.38823494  1.25979759 -0.15790843]
W_alpha2 =  [[ 0.01510583  0.06097955 -0.2369113   0.10630067 -0.16159161 -0.05445503
   0.11292355  0.01668015 -0.09922444  0.04643061]
 [-0.09378544  0.05457945 -0.19433084  0.19653329 -0.25153438  0.10289609
   0.00427415  0.06406208 -0.20450693 -0.04607336]
 [ 0.17485552  0.24716067 -0.16020414  0.21222004 -0.03512559  0.2765797
  -0.19422548 -0.30834113 -0.11048566  0.1304683 ]
```

```
 [-0.51549836  0.07327962  0.10011824  0.05009513 -0.06221172 -0.12360759
   0.17936366  0.18403602 -0.2212413  -0.00592128]
 [ 0.67097739  0.75108231 -0.56895399 -0.14873161 -0.00094841  0.16430225
  -0.44677964  0.12966539  0.72357938 -0.29466421]
 [-0.68222409 -0.00648584 -0.30689758 -0.19025547  0.15631307 -0.53000152
   0.88017965  0.90805858 -0.16840504 -0.92400869]
 [ 0.00969503 -0.18201188 -0.42952734  0.11494872 -0.08194413  0.00162115
  -0.00639572  0.22466186 -0.00288281  0.10743068]
 [ 0.56728021 -0.17619186  0.13699939  0.55291493 -0.33454617  0.29816396
  -0.45710155 -0.85833694  0.32227907  0.84714251]
 [-0.0138277   0.07472525  0.71227894 -0.36279291 -0.67364398 -0.56417772
   0.30497906 -0.81471725 -0.77790993  0.76843616]]  B_alpha2 =  [-1.31244763 -1.31202985 -1.37668858 -1.3193119
1.10125174  3.36285142
  3.63804677  2.47330418 -0.75497615]
W_alpha3 =  [[-0.0103913   0.07123477 -0.29026112  0.16582515 -0.12683126 -0.01926057
   0.1418642   0.07888062 -0.0250145  -0.03040879]
 [-0.11549964  0.07369954 -0.26080427  0.20193258 -0.17466495  0.09778729
   0.03444067  0.13027729 -0.10748591 -0.06442457]
 [ 0.13168631  0.22294922 -0.23163413  0.24511116 -0.07038447  0.24446732
  -0.11166054 -0.1654924  -0.01630578  0.04247053]
 [-0.46572684  0.10048862 -0.0225807   0.10928403 -0.07350257 -0.07960552
   0.16703763  0.25101324 -0.1521749  -0.08019175]
 [ 0.72770037  0.67610124 -0.84561544 -0.05967186 -0.0432796   0.60447819
  -0.87621714  0.36857034  0.79706862 -0.33697228]
 [-0.73856126 -0.02452797 -0.55662509 -0.11660403  0.12650055 -0.24612763
   0.60328084  1.16179792 -0.14763159 -0.95095395]
 [-0.05337179 -0.18890036 -0.66932189  0.18494277 -0.10381407  0.26930492
  -0.26483673  0.47838094  0.01602179  0.06465259]
 [ 0.47180776 -0.16837761 -0.09514937  0.60752761 -0.35020233  0.54615706
  -0.69790061 -0.58939086  0.31499444  0.79124431]
 [ 0.18493478  0.13444982  2.02456339 -0.80711461 -0.62905418 -1.84587977
   1.38120935 -2.16826833 -1.21826983  1.19382463]]  B_alpha3 =  [-2.2987201  -2.30067213 -2.35730711 -2.32077955
2.58999208  4.80910958
  5.0768996   3.9462791  -2.64480147]
```

In [ ]:
```
def locateMax(tab):
    max=0
    for loop in range(len(tab)):
        if tab[loop]>tab[max]:
            max=loop
    return max
```

In [ ]:
```
def predict_accuracy(X,y,W,B):
    yp = np.zeros(len(X))
    for loop in range(len(X)):
        tabProbas=np.dot(W,X[loop])+B
        tabProbas=sigmoid(tabProbas)
        tabProbas=tabProbas/np.sum(tabProbas)
        yp[loop]=locateMax(tabProbas)

    sum=0
    for loop in range(len(yp)):
        if(yp[loop] == y[loop]):
            sum+=1
    print('Train Accuracy (%) : ',(sum/len(yp))*100)
```

In [ ]:
```
print("predictions sur le jeu d'entrainemnt")
predict_accuracy(X_train,y_train,W,B)
predict_accuracy(X_train,y_train,W2,B2)
predict_accuracy(X_train,y_train,W3,B3)

print("predictions sur le jeu cv")
predict_accuracy(X_cv,y_cv,W,B)
predict_accuracy(X_cv,y_cv,W2,B2)
predict_accuracy(X_cv,y_cv,W3,B3)

print("predictions sur le jeu de test")
predict_accuracy(X_test,y_test,W,B)
predict_accuracy(X_test,y_test,W2,B2)
predict_accuracy(X_test,y_test,W3,B3)
```

```
predictions sur le jeu d'entrainemnt
Train Accuracy (%) :  57.67195767195767
Train Accuracy (%) :  56.87830687830689
Train Accuracy (%) :  57.14285714285714
predictions sur le jeu cv
Train Accuracy (%) :  50.79365079365079
Train Accuracy (%) :  50.79365079365079
Train Accuracy (%) :  50.79365079365079
predictions sur le jeu de test
Train Accuracy (%) :  53.96825396825397
Train Accuracy (%) :  53.96825396825397
Train Accuracy (%) :  52.38095238095239
```

Après 2000 itérations, le premier alpha = 0.01 est le meilleur. On ne prédit parfaitement la note que pour 54% des observations pour notre meilleur alpha. Calculons l'erreur du modèle sur la note.

In [ ]:
```
def compute_cost(X,y,W,B):
    m=len(X)
    yp = np.zeros(m)
    for loop in range(m):
        tabProbas=np.dot(W,X[loop])+B
        tabProbas=sigmoid(tabProbas)
        tabProbas=tabProbas/np.sum(tabProbas)
        yp[loop]=locateMax(tabProbas)

    cost = 0.0
    for i in range(m):
        cost = cost + (y[i] - yp[i])**2        #scalar
    cost = cost / (2 * len(X))                 #scalar
    return cost
```

In [ ]:
```
compute_cost(X_test, y_test, W, B)
```

Out[ ]:
```
0.2896825396825397
```

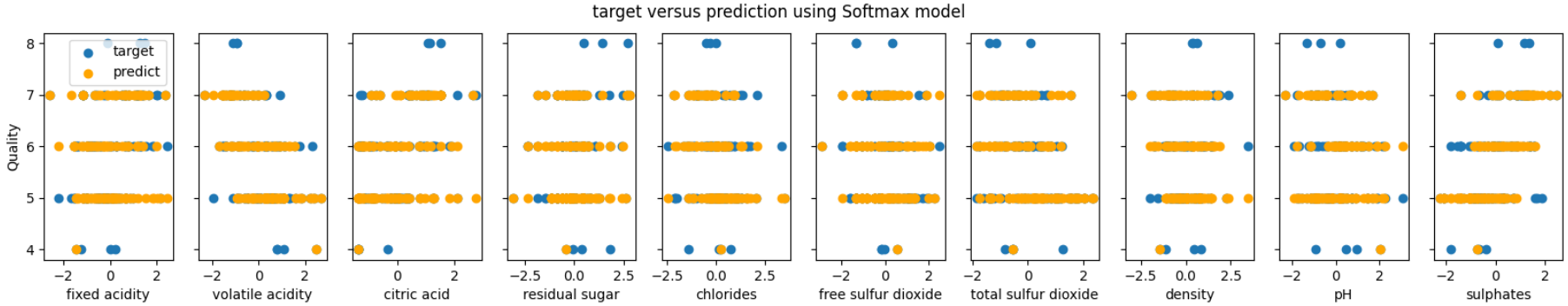
Le côut = 0,29. Il est un peu plus élevé que pour la Regression linéaire. Un problème avec la méthode softmax est qu'elle ne fait que de la multi-classification. Pour la qualité du vin, l'importance de l'erreur sur la note doit être prise en compte. C'est le cas dans la regression polynômiale mais pas dans le Softmax (il faudrait changer la méthode de calcul du coût et du gradient).

Visualisation du résultat sur le jeu de test:

In [ ]:
```
x_test = X_test[50]
print(x_test)

tabProbas=np.dot(W,x_test)+B
tabProbas=sigmoid(tabProbas)
tabProbas=tabProbas/np.sum(tabProbas)
print(np.sum(tabProbas))
print(tabProbas)

print("largest value", np.max(tabProbas), "smallest value", np.min(tabProbas))
print("position du max : ",locateMax(tabProbas))
print("valeur réelle: ",y_test[50])
```

```
[-0.26543644  0.22331012 -0.91343632  0.37202347  1.61392296  0.21672234
  0.44556875  0.28921786 -0.62551511  0.17755786]
1.0
[0.08282363 0.07814757 0.09684344 0.09359955 0.10951211 0.19090459
 0.18345746 0.11763434 0.04707732]
largest value 0.19090458944201769 smallest value 0.047077316978753544
position du max :  5
valeur réelle:  6
```

In [ ]:
```
#predict target using normalized features
m = X_test.shape[0]
yp = np.zeros(m)
for loop in range(len(X_test)):
    tabProbas=np.dot(W,X_test[loop])+B
    tabProbas=sigmoid(tabProbas)
    tabProbas=tabProbas/np.sum(tabProbas)
    yp[loop]=locateMax(tabProbas)

    # plot predictions and targets versus original features
fig,ax=plt.subplots(1,nb_feature,figsize=(20, 3),sharey=True)
for i in range(len(ax)):
    ax[i].scatter([X_test[:,i]],y_test, label = 'target')
    ax[i].set_xlabel(X_features[i])
    ax[i].scatter([X_test[:,i]],yp,color="orange", label = 'predict')
ax[0].set_ylabel("Quality"); ax[0].legend()
fig.suptitle("target versus prediction using Softmax model")
plt.show()
print(yp)
print(y_test)
```



```
[5. 7. 5. 6. 6. 6. 7. 5. 5. 5. 5. 6. 6. 7. 5. 6. 5. 5. 5. 7. 5. 7. 6. 7.
 5. 5. 7. 7. 5. 6. 5. 5. 5. 6. 5. 6. 7. 6. 6. 7. 7. 5. 5. 6. 6. 4. 5. 5.
 5. 5. 5. 7. 5. 6. 6. 6. 6. 7. 7. 5. 7. 5. 5. 5. 7. 6. 5. 7. 5. 5. 7. 5.
 6. 7. 7. 6. 6. 6. 6. 6. 6. 5. 7. 7. 5. 5. 5. 5. 5. 6. 5. 5. 5. 5. 6. 7.
 6. 7. 5. 7. 7. 5. 5. 5. 5. 5. 5. 6. 6. 6. 5. 7. 7. 6. 5. 6. 7. 7. 6. 5.
 5. 5. 7. 5. 6. 5.]
[5 5 5 5 5 6 6 5 6 6 6 7 7 7 6 5 5 6 5 7 5 7 6 7 5 6 7 6 6 6 6 5 6 7 7 5 7
 6 7 7 6 6 5 5 6 4 5 5 5 6 6 7 6 5 5 6 6 7 7 4 7 5 5 5 7 6 5 6 5 4 7 5 6 5
 6 5 7 6 5 7 5 5 6 7 6 5 5 5 5 5 5 5 5 6 6 5 7 5 8 8 6 5 6 5 6 5 6 6 7 7
 6 7 5 5 6 8 7 4 5 5 5 6 5 7 6]
```

On a aussi du mal à prédire les notes hautes et basses. Pour avoir une autre approche que la régression ou que cette multi-classification, nous allons utiliser un réseau de Neurones. Notre réseau de neurones va utiliser Softmax pour une multi-classification avec une probabilité d'appartenance à une qualité en fonction de l'observation X, mais après plusieurs couches de transformation des données à l'aide de fonctions linéaires.

3) Réseau de Neurones

Le réseau de neurones va prédire la note de la même manière que le Softmax, mais après des étapes de transformations des données. On va créer nos modèles grâce à TensorFlow.

a) Préparation des données (comme précédemment)

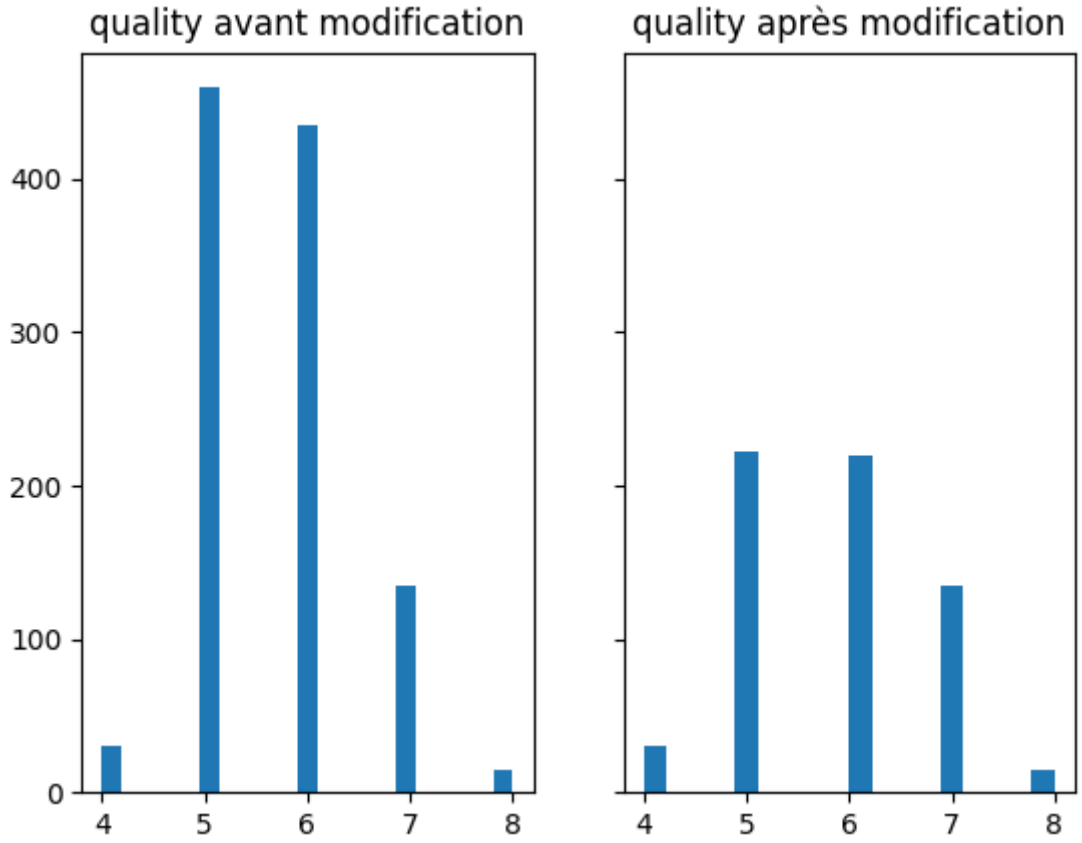
In [ ]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import random
df = pd.read_csv('WineQT.csv')
import numpy as np
import matplotlib.pyplot as plt

def plot_hists(df):
    fig, ax = plt.subplots(nrows=3, ncols=4, figsize=(20, 15))
    for n in range(12):
        i = n % 3
        j = n % 4
        ax[i, j].hist(df.iloc[:, n], bins='auto')
        ax[i, j].set_xlabel(df.columns[n])

#On normalise : mettre entre 0 et 1
def normalize(df, property, parameter):
    df[property] = np.log(df[property] + parameter)

normalize(df, "fixed acidity", -2.3)
normalize(df, "sulphates", -0.24)
normalize(df, "total sulfur dioxide", 5)
normalize(df, "residual sugar", -1.1)
normalize(df, "chlorides", -0.005)
normalize(df, "volatile acidity", 2)
normalize(df, "free sulfur dioxide", 2)
#plot_hists(df)

standardized = (df - df.mean()) / df.std()
standardized = standardized[(np.abs(standardized) < 3).all(axis=1)]
rows = np.setdiff1d(list(df.index), list(standardized.index))
df.drop(index=rows, inplace=True)
#plot_hists(df)

import numpy as np
import matplotlib.pyplot as plt

#Préparation des données
y = df['quality']
X= [df['fixed acidity'],  df['volatile acidity'],  df['citric acid'],  df['residual sugar'],  df['chlorides'] df['f
```

```
X= [df['fixed acidity'],  df['volatile acidity']  ,df['citric acid']  ,df['residual sugar'],  df['chlorides'],df['
X=np.transpose(np.array(X))
y=np.asarray(y)
print(X.shape)
print(y.shape)
X_features = ['fixed acidity',  'volatile acidity'  ,'citric acid'  ,'residual sugar',  'chlorides','free sulfur di
nb_feature=len(X_features)


#on supprime aleatoirement des valeurs de notes 5 et 6 (diviser par 2)
supp=[]
for i in range(len(y)):
    if y[i]==5 or y[i]==6:
        rand=random.random()
        if(rand>0.5):
            supp.append(i)
y2=np.delete(y,supp)

X2=np.delete(X,supp,0)

#Plot des modifications
fig,ax=plt.subplots(1,2,sharey=True)
ax[0].hist(y, bins='auto',label="quality")
ax[0].set_title("quality avant modification")

ax[1].hist(y2, bins='auto',label="quality")
ax[1].set_title("quality après modification")

#on créé les jeux de données
from sklearn.model_selection import train_test_split
X_train, X_tmp, y_train, y_tmp = train_test_split(X2, y2, test_size=0.4, random_state=42)
X_cv, X_test, y_cv, y_test = train_test_split(X_tmp, y_tmp, test_size=0.5, random_state=42)

def zscore_normalize_features(X):
    mu     = np.mean(X, axis=0)                 # mu will have shape (n,)
    # find the standard deviation of each column/feature
    sigma  = np.std(X, axis=0)                  # sigma will have shape (n,)
    # element-wise, subtract mu for that column from each example, divide by std for that column
    X_norm = (X - mu) / sigma

    return (X_norm, mu, sigma)
```

```
 # normalize the original features
X_train, X_mu, X_sigma = zscore_normalize_features(X_train)
X_cv, X_mu, X_sigma = zscore_normalize_features(X_cv)
X_test, X_mu, X_sigma = zscore_normalize_features(X_test)
```

```
/home/henri/.local/lib/python3.8/site-packages/pandas/core/series.py:726: RuntimeWarning: invalid value encountere
d in log
  result = getattr(ufunc, method)(*inputs, **kwargs)
(1074, 10)
(1074,)
```



b) Création de 3 modèles de réseau de neurones avec des architectures différentes :

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from IPython.display import display, Markdown, Latex
from matplotlib.widgets import Slider
import logging
logging.getLogger("tensorflow").setLevel(logging.ERROR)
tf.autograph.set_verbosity(0)
```

In [ ]:
```
tf.random.set_seed(1234)
```

In [ ]:
```
def build_models():

    model_1 = Sequential(
        [
            tf.keras.Input(shape=(nb_feature,)),
            Dense(25, activation = 'relu'),
            Dense(15, activation = 'relu'),
            Dense(9, activation = 'softmax')
        ],
        name='model_1'
    )
    model_2 = Sequential(
        [
            tf.keras.Input(shape=(nb_feature,)),
            Dense(20, activation = 'relu'),
            Dense(12, activation = 'relu'),
            Dense(12, activation = 'relu'),
            Dense(20, activation = 'relu'),
            Dense(9, activation = 'softmax')
        ],
        name='model_2'
    )
    model_3 = Sequential(
        [
            tf.keras.Input(shape=(nb_feature,)),
            Dense(32, activation = 'relu'),
            Dense(16, activation = 'relu'),
            Dense(8, activation = 'relu'),
            Dense(4, activation = 'relu'),
            Dense(12, activation = 'relu'),
            Dense(9, activation = 'softmax')
        ],
        name='model_3'
    )

    model_list = [model_1, model_2, model_3]

    return model_list
```

In [ ]:
```
def locateMax(tab):
    max=0
    for loop in range(len(tab)):
        if tab[loop]>tab[max]:
            max=loop
    return max
```

In [ ]:
```
def sigmoid(z):
    g = 1/(1+np.exp(-z))
    return g

def calcul_error(yhat,y_test):

    cost = 0.0
    m=len(y_test)
    for i in range(m):
        cost = cost + (y_test[i] - yhat[i])**2   #scalar
    cost = cost / (2 * m)                  #scalar
    return cost
```

In [ ]:
```
X_train = np.tile(X_train,(100,1))
y_train= np.transpose(np.tile(y_train,(1,100))   )
print(X_train.shape, y_train.shape)
```

```
(37300, 10) (37300, 1)
```

c) Entraînement et évaluation de nos modèles

In [ ]:
```
# Setup the loss and optimizer
def try_model(model):

    model.compile(
    loss=tf.keras.losses.SparseCategoricalCrossentropy(),
    optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
    )

    print(f"Training {model.name}...")


    # Train the model
    model.fit(
        X_train, y_train,
        epochs=10,
        verbose=0
    )
    print("Done!\n")

    # Record the fraction of misclassified examples for the training set
    predictions = model.predict(X_train)
    yhat= np.zeros(len(predictions))

    for loop in range(len(predictions)):
        yhat[loop]=locateMax(predictions[loop])
    #print(yhat)

    train_error = calcul_error(yhat,y_train)

    # Record the fraction of misclassified examples for the cross validation set
    predictions = model.predict(X_cv)
    yhat= np.zeros(len(predictions))

    for loop in range(len(predictions)):
        yhat[loop]=locateMax(predictions[loop])
    #print(yhat)

    cv_error = calcul_error(yhat,y_cv)
    return(train_error,cv_error)
```

In [ ]:
```
from sklearn.metrics import mean_squared_error
# Initialize lists that will contain the errors for each model
nn_train_error = []
nn_cv_error = []


# Build the models
nn_models = build_models()

# Loop over the the models
for model in nn_models:

    train_error,cv_error=try_model(model)
    nn_train_error.append(train_error)
    nn_cv_error.append(cv_error)


# Print the result
for model_num in range(len(nn_train_error)):
    print("Model ",model_num,": Training Set Classification Error:",
          nn_train_error[model_num],"CV Set Classification Error:",nn_cv_error[model_num])
```

```
Training model_1...
Done!

1166/1166 [==============================] - 1s 886us/step
4/4 [==============================] - 0s 2ms/step
Training model_2...
Done!

1166/1166 [==============================] - 1s 876us/step
4/4 [==============================] - 0s 1ms/step
Training model_3...
Done!

1166/1166 [==============================] - 1s 863us/step
4/4 [==============================] - 0s 1ms/step
Model  0 : Training Set Classification Error: [0.] CV Set Classification Error: 0.4879032258064516
Model  1 : Training Set Classification Error: [0.] CV Set Classification Error: 0.5080645161290323
Model  2 : Training Set Classification Error: [0.00134048] CV Set Classification Error: 0.5080645161290323
```

Nos trois réseaux de neurones ne commettent aucune erreur sur le jeu d'entraînement. On choisit le premier modèle car il a l'erreur la plus faible sur le jeu de validation croisée. Visualisons et testons le modèle choisi : d'abord sur le jeu d'entrainement puis sur le jeu de test:

In [ ]:
```
def predict_model(model,X,y):

    predictions = model.predict(X)
    yhat= np.zeros(len(predictions))

    for loop in range(len(predictions)):
        yhat[loop]=locateMax(predictions[loop])
    print(yhat)
    print(y)
    print(f"Selected Model: {model_num}")
    sum=0
    for loop in range(len(yhat)):
        if(y[loop] == yhat[loop]):
            sum+=1
    print('Train Accuracy (%): ',(sum/len(yhat))*100)

    nn_test_error =calcul_error(yhat,y)

    print(f"Set Classification Error: {nn_test_error:.4f}")
```

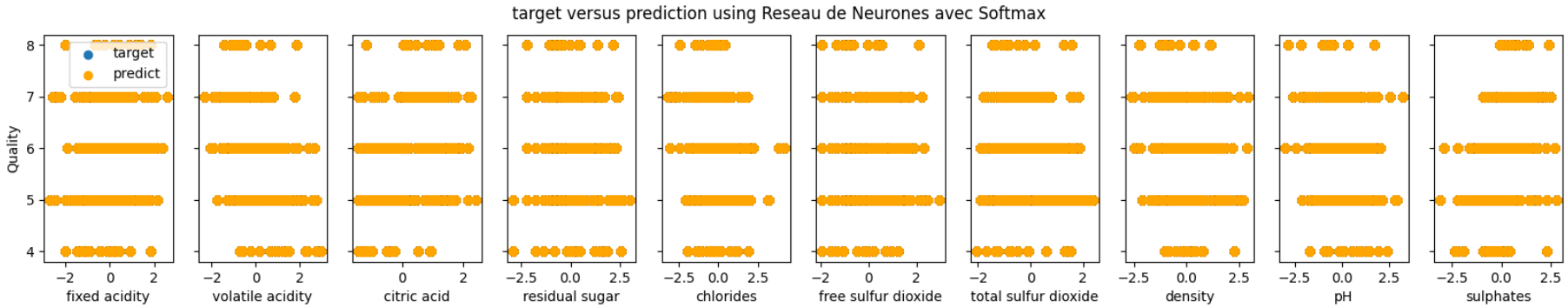
In [ ]:
```
model_num = 1
model=nn_models[model_num-1]
predict_model(model,X_test,y_test)
```

```
4/4 [==============================] - 0s 1ms/step
[5. 6. 7. 6. 5. 5. 5. 5. 6. 6. 6. 6. 7. 5. 5. 5. 7. 7. 6. 7. 5. 7. 6. 7.
 6. 5. 6. 7. 5. 7. 5. 5. 6. 6. 5. 8. 7. 6. 5. 6. 5. 5. 5. 6. 5. 5. 7. 7.
 5. 4. 7. 5. 7. 7. 5. 7. 6. 6. 6. 6. 5. 5. 5. 5. 6. 5. 5. 7. 8. 6. 7. 6.
 6. 6. 5. 7. 6. 5. 6. 5. 5. 6. 6. 7. 6. 6. 5. 6. 7. 5. 7. 7. 6. 5. 6. 7.
 4. 7. 5. 5. 5. 6. 5. 4. 7. 6. 5. 6. 6. 6. 7. 5. 7. 7. 5. 6. 7. 5. 5. 6.
 5. 7. 6. 6. 6.]
[6 6 7 5 5 5 5 6 6 5 7 6 6 5 6 5 6 6 5 5 7 7 5 6 5 5 6 7 5 5 6 5 5 5 6 7 6
 5 5 7 5 6 6 6 5 5 7 6 7 5 7 5 7 7 6 7 4 5 6 5 6 5 5 8 6 6 6 6 5 6 6 5 4 5
 5 7 6 5 5 4 6 5 6 5 6 6 5 6 7 5 7 5 6 5 7 5 6 6 7 5 5 4 5 5 7 6 5 6 6 7 8
 5 6 7 5 5 7 5 5 6 6 5 6 6 5]
Selected Model: 1
Train Accuracy (%):  49.6
Set Classification Error: 0.4720
```

In [ ]:
```
def plot_predict_model(model,X,y):
    predictions = model.predict(X)
    yhat= np.zeros(len(predictions))

    for loop in range(len(predictions)):
        yhat[loop]=locateMax(predictions[loop])

    sum=0
    for loop in range(len(yhat)):
        if(y[loop] == yhat[loop]):
            sum+=1
    print('Train Accuracy (%): ',(sum/len(yhat))*100)

    nn_test_error =calcul_error(yhat,y)

    print("Set Classification Error:",nn_test_error)

    m = X.shape[0]

        # plot predictions and targets versus original features
    fig,ax=plt.subplots(1,nb_feature,figsize=(20, 3),sharey=True)
    for i in range(len(ax)):
        ax[i].scatter([X[:,i]],y, label = 'target')
        ax[i].set_xlabel(X_features[i])
        ax[i].scatter([X[:len(yhat),i]],yhat,color="orange", label = 'predict')
    ax[0].set_ylabel("Quality"); ax[0].legend()
    fig.suptitle("target versus prediction using Reseau de Neurones avec Softmax")
    plt.show()
```

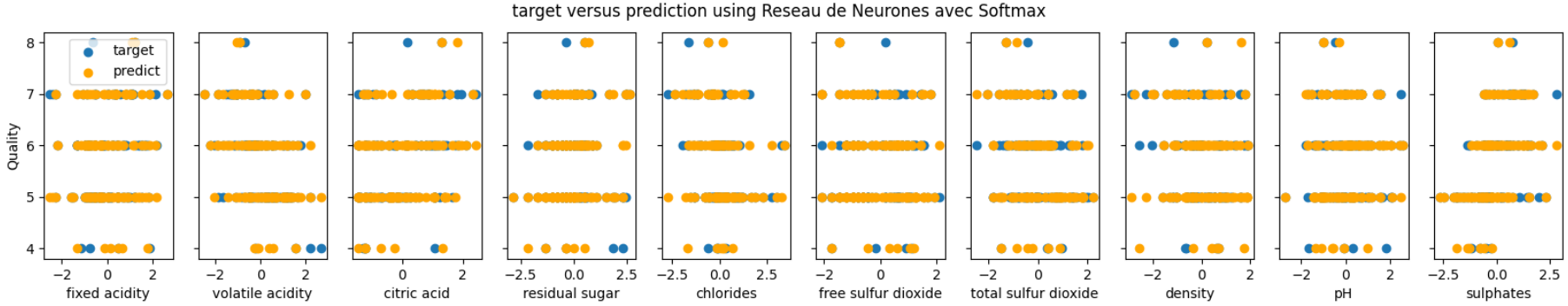
In [ ]:
```
model_num = 1
model=nn_models[model_num-1]
plot_predict_model(model,X_train,y_train)
```

```
1166/1166 [==============================] - 1s 839us/step
Train Accuracy (%):  99.46380697050938
Set Classification Error: [0.00268097]
```



In [ ]:
```
model_num = 2
model=nn_models[model_num-1]
plot_predict_model(model,X_test,y_test)
```

```
4/4 [==============================] - 0s 1ms/step
Train Accuracy (%):  51.2
Set Classification Error: 0.372
```



Le réseau de neurones parait plus adapté visuellement que la régression et Softmax, mais on voit un autre type de problème : une grande variance. En effet on ne réalise aucune erreur sur le jeu d'entraînement mais on obtient que 50% de précision sur le jeu de test. Il faut donc régulariser le modèle.

d) Régularisation du modèle

On va rajouter un paramètre de régularisation Lambda en gardant le même modèle.

In [ ]:
```
from tensorflow.keras.regularizers import l2
def build_models_Regularized():
    lambdas=[0,0.01,0.1]
    model_1 = Sequential(
        [
            tf.keras.Input(shape=(nb_feature,)),
            Dense(25, activation = 'relu',kernel_regularizer=l2(lambdas[0])),
            Dense(15, activation = 'relu',kernel_regularizer=l2(lambdas[0])),
            Dense(9, activation = 'softmax',kernel_regularizer=l2(lambdas[0]))
        ],
        name='model_1'
    )
    model_2 = Sequential(
        [
            tf.keras.Input(shape=(nb_feature,)),
            Dense(25, activation = 'relu',kernel_regularizer=l2(lambdas[1])),
            Dense(15, activation = 'relu',kernel_regularizer=l2(lambdas[1])),
            Dense(9, activation = 'softmax',kernel_regularizer=l2(lambdas[1]))
        ],
        name='model_2'
    )
    model_3 = Sequential(
        [
            tf.keras.Input(shape=(nb_feature,)),
            Dense(25, activation = 'relu',kernel_regularizer=l2(lambdas[2])),
            Dense(15, activation = 'relu',kernel_regularizer=l2(lambdas[2])),
            Dense(9, activation = 'softmax',kernel_regularizer=l2(lambdas[2]))
        ],
        name='model_3'
    )
    model_list = [model_1, model_2, model_3]
    return model_list
```

In [ ]:
```
from sklearn.metrics import mean_squared_error
# Initialize lists that will contain the errors for each model
nn_train_error = []
nn_cv_error = []


# Build the models
nn_models_r = build_models_Regularized()

# Loop over the the models
for model in nn_models_r:

    train_error,cv_error=try_model(model)
    nn_train_error.append(train_error)
    nn_cv_error.append(cv_error)

# Print the result
for model_num in range(len(nn_train_error)):
    print(
        "Model",(model_num+1),": Training Set Classification Error:", nn_train_error[model_num],
        "CV Set Classification Error:", nn_cv_error[model_num])
```

```
Training model_1...
Done!

1166/1166 [==============================] - 1s 866us/step
4/4 [==============================] - 0s 1ms/step
Training model_2...
Done!

1166/1166 [==============================] - 1s 959us/step
4/4 [==============================] - 0s 1ms/step
Training model_3...
Done!

1166/1166 [==============================] - 1s 855us/step
4/4 [==============================] - 0s 1ms/step
Model 1 : Training Set Classification Error: [0.] CV Set Classification Error: 0.5040322580645161
Model 2 : Training Set Classification Error: [0.20375335] CV Set Classification Error: 0.3548387096774194
Model 3 : Training Set Classification Error: [0.42895442] CV Set Classification Error: 0.47580645161290325
```

C'est lambda = 0.01 qui minimise l'erreur sur le jeu CV. On a augmenté l'erreur sur le jeu d'entraînement pour diminuer l'erreur sur le jeu de validation. Testons cette régularisation sur le jeu de test.

In [ ]:
```
# Select the model with the lowest error
model_num = 2
model=nn_models_r[model_num-1]
predict_model(model,X_test,y_test)
```

```
4/4 [==============================] - 0s 1ms/step
[5. 6. 7. 6. 5. 5. 5. 5. 6. 5. 6. 6. 7. 6. 5. 6. 6. 7. 6. 6. 6. 7. 5. 7.
 5. 5. 7. 6. 5. 6. 5. 5. 6. 5. 6. 5. 7. 6. 5. 7. 5. 5. 6. 6. 5. 5. 7. 7.
 7. 5. 7. 6. 7. 7. 5. 7. 7. 6. 6. 6. 5. 5. 5. 7. 5. 5. 6. 7. 7. 6. 6. 6.
 5. 5. 5. 7. 7. 6. 5. 5. 6. 6. 6. 5. 5. 6. 4. 6. 7. 5. 7. 6. 6. 5. 5. 5.
 5. 7. 6. 5. 5. 6. 5. 5. 7. 7. 5. 6. 6. 7. 7. 5. 7. 7. 5. 6. 6. 5. 6. 6.
 6. 6. 6. 6. 5.]
[6 6 7 5 5 5 5 6 6 5 7 6 6 5 6 5 6 6 5 5 7 7 5 6 5 5 6 7 5 5 6 5 5 5 6 7 6
 5 5 7 5 6 6 6 5 5 7 6 7 5 7 5 7 7 6 7 4 5 6 5 6 5 5 8 6 6 6 6 5 6 6 5 4 5
 5 7 6 5 5 4 6 5 6 5 6 6 5 6 7 5 7 5 6 5 7 5 6 6 7 5 5 4 5 5 7 6 5 6 6 7 8
 5 6 7 5 5 7 5 5 6 6 5 6 6 5]
Selected Model: 2
Train Accuracy (%):  56.00000000000001
Set Classification Error: 0.3000
```

On obtient 56% de précision. La régularisation a été un peu utile. Pour s'améliorer il faudrait trouver un équilibre biais-variance en utilisant la régularisation et en effectuant une transformation des données en amont.

4) Les Arbres de Décision

Nous allons tester maintenant les arbres de décision. Dans ces arbres chaque noeud divise les observations en 2 branches en fonction d'une condition (par exemple alcohol<=12) et chaque feuille de l'arbre correspond à une qualité égale à la moyenne de la qualité des observations présentes dans la feuille. Une nouvelle observation x n'aura qu'à suivre le chemin de l'arbre en fonction des conditions des noeuds pour atterir dans une feuille. La prédiction sera alors la qualité de la feuille.

a) Préparation des données (comme précédemment)

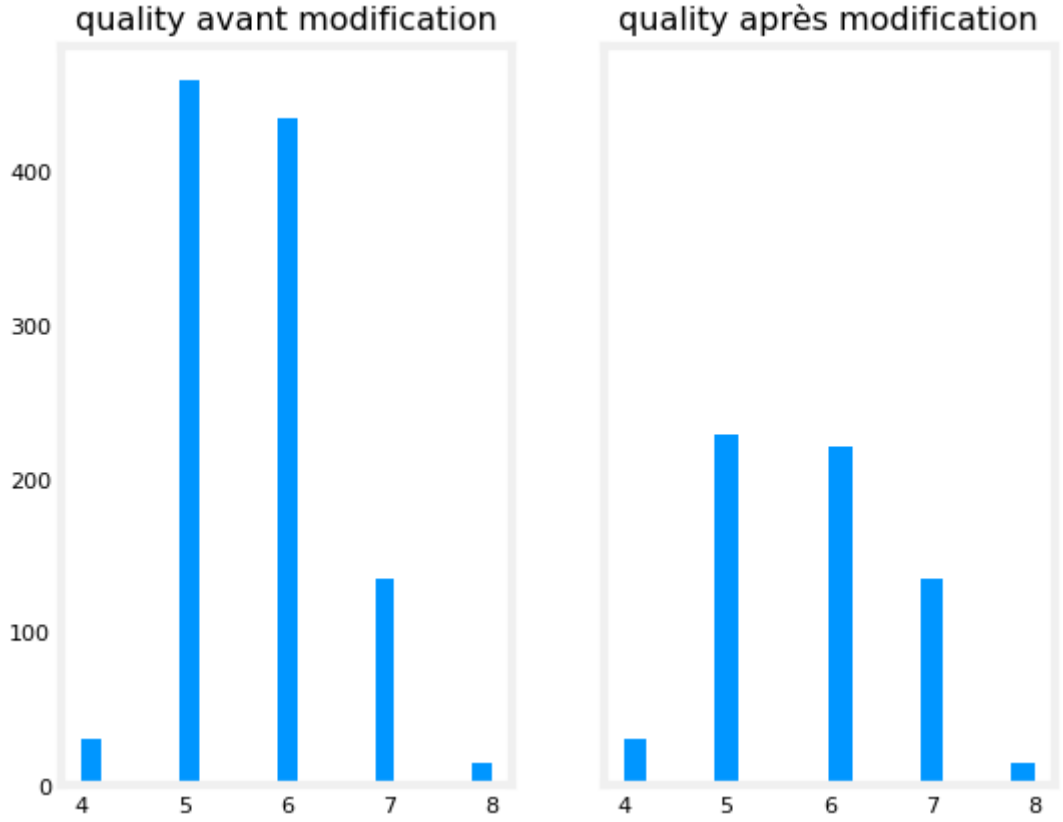
In [ ]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import random
df = pd.read_csv('WineQT.csv')
import numpy as np
import matplotlib.pyplot as plt

def plot_hists(df):
    fig, ax = plt.subplots(nrows=3, ncols=4, figsize=(20, 15))
    for n in range(12):
        i = n % 3
        j = n % 4
        ax[i, j].hist(df.iloc[:, n], bins='auto')
        ax[i, j].set_xlabel(df.columns[n])

#On normalise : mettre entre 0 et 1
def normalize(df, property, parameter):
    df[property] = np.log(df[property] + parameter)


normalize(df, "fixed acidity", -2.3)
normalize(df, "sulphates", -0.24)
normalize(df, "total sulfur dioxide", 5)
normalize(df, "residual sugar", -1.1)
normalize(df, "chlorides", -0.005)
normalize(df, "volatile acidity", 2)
normalize(df, "free sulfur dioxide", 2)
#plot_hists(df)

standardized = (df - df.mean()) / df.std()
standardized = standardized[(np.abs(standardized) < 3).all(axis=1)]
rows = np.setdiff1d(list(df.index), list(standardized.index))
df.drop(index=rows, inplace=True)
#plot_hists(df)

import numpy as np
import matplotlib.pyplot as plt

#Préparation des données
y = df['quality']
```

```
y = df[ 'quality' ]
X= [df['fixed acidity'],  df['volatile acidity']  ,df['citric acid']  ,df['residual sugar'],  df['chlorides'],df['f
X=np.transpose(np.array(X))
y=np.asarray(y)
print(X.shape)
print(y.shape)
X_features = ['fixed acidity',  'volatile acidity'  ,'citric acid'  ,'residual sugar',  'chlorides','free sulfur di
nb_feature=len(X_features)


#on supprime aleatoirement des valeurs de notes 5 et 6 (diviser par 2)
supp=[]
for i in range(len(y)):
    if y[i]==5 or y[i]==6:
        rand=random.random()
        if(rand>0.5):
            supp.append(i)
y2=np.delete(y,supp)

X2=np.delete(X,supp,0)

#Plot des modifications
fig,ax=plt.subplots(1,2,sharey=True)
ax[0].hist(y, bins='auto',label="quality")
ax[0].set_title("quality avant modification")

ax[1].hist(y2, bins='auto',label="quality")
ax[1].set_title("quality après modification")

#on créé les jeux de données
from sklearn.model_selection import train_test_split
X_train, X_tmp, y_train, y_tmp = train_test_split(X2, y2, test_size=0.4, random_state=42)
X_cv, X_test, y_cv, y_test = train_test_split(X_tmp, y_tmp, test_size=0.5, random_state=42)

def zscore_normalize_features(X):
    mu     = np.mean(X, axis=0)                 # mu will have shape (n,)
    # find the standard deviation of each column/feature
    sigma  = np.std(X, axis=0)                  # sigma will have shape (n,)
    # element-wise, subtract mu for that column from each example, divide by std for that column
    X_norm = (X - mu) / sigma

    return (X_norm, mu, sigma)
```

```
    return (X_norm, mu, sigma)

 # normalize the original features
X_train, X_mu, X_sigma = zscore_normalize_features(X_train)
X_cv, X_mu, X_sigma = zscore_normalize_features(X_cv)
X_test, X_mu, X_sigma = zscore_normalize_features(X_test)
```

```
/home/henri/.local/lib/python3.8/site-packages/pandas/core/series.py:726: RuntimeWarning: invalid value encountere
d in log
  result = getattr(ufunc, method)(*inputs, **kwargs)
(1074, 10)
(1074,)
```



b) Mise en place d'un arbre de décision

On va commencer par mettre en place un arbre de décision. Pour cela :

- **on commence à la racine avec tout le dataset qu'on veut split.**
- **on teste les splits sur toutes les caractéristiques du vin avec un certain nombre de valeurs (ex: sulfate <= valeurn°12).**
- **on décide le split choisi en calculant l'utilité = le gain d'information qui dépend de la pureté des noeuds résultant du split.**
- **on sépare le dataset en fonction du meilleur split et on refait récursivement la même chose sur les 2 nouveaux noeuds.**
- **on s'arrête lorsque un noeud est totalement pure (=tous les vins de même qualité), ou lorsqu'on atteint une certaine profondeur.**

Codes de la structure de l'arbre pour la prédiction du vin:

In [ ]:
```
class ArbreBinaireVin:
    def __init__(self):
        self.qualite = 0
        self.split=0
        self.carac=0
        self.enfant_gauche = None
        self.enfant_droit = None


    def insert_gauche(self):
        self.enfant_gauche = ArbreBinaireVin()

    def insert_droit(self):
        self.enfant_droit = ArbreBinaireVin()

    def get_valeur(self):
        return self.valeur

    def get_gauche(self):
        return self.enfant_gauche

    def get_droit(self):
        return self.enfant_droit


    def get_predictionVin(self,x):

        if(x[self.carac]<=self.split):
            if(self.enfant_gauche==None):
                return self.qualite
            else:
                return self.enfant_gauche.get_predictionVin(x)
        else:
            if(self.enfant_droit==None):
                return self.qualite
            else:
                return self.enfant_droit.get_predictionVin(x)

    def affiche(self):
        print(self.carac,X_features[self.carac],self.split,self.qualite)
        if(self.enfant_gauche!=None):
```

```
        if(self.enfant_gauche!=None):
            self.enfant_gauche.affiche()
        if(self.enfant_droit!=None):
            self.enfant_droit.affiche()
```

Codes de la construction de l'arbre avec le dataset d'entraînement:

In [ ]:
```
#calcul de l'impureté d'un noeuf, pour savoir à quel point le noeud est pur (=les vins qui s'y trouvent ont la même
def gini_Impurity(y):

    #on calcul le nombre de valeur par note de vin (0 à 8)
    tab_value=np.zeros(9)
    for loop in range(len(y)):
        tab_value[y[loop]]+=1
    #calcul de l'impureté
    impurity=1
    for loop in range(len(tab_value)):
        impurity-=(tab_value[loop]/sum(tab_value))**2
    return impurity

#split du noeud pour des valeurs continues (ex:split en fonction de la condition {X_alcohol<=12.355?})
def split_dataset_continue(X, node_indices, feature,t):

    left_indices = []
    right_indices = []
    for i in node_indices:
        if X[i,feature] <= t:
            left_indices.append(i)
        else:
            right_indices.append(i)

    return left_indices, right_indices

#calcul du gain d'information = utilité d'un split, permet de choisir sur quelle condition on va split le noeud
def compute_information_gain_continue(X, y, node_indices, feature, t):

    left_indices, right_indices = split_dataset_continue(X, node_indices, feature,t)

    X_node, y_node = X[node_indices], y[node_indices]
    X_left, y_left = X[left_indices], y[left_indices]
    X_right, y_right = X[right_indices], y[right_indices]

    information_gain = 0

    node_entropy = gini_Impurity(y_node)
    left_entropy = gini_Impurity(y_left)
    right_entropy = gini_Impurity(y_right)
    w_left = len(X_left) / len(X_node)
```

```
    w_left = len(X_left) / len(X_node)
    w_right = len(X_right) / len(X_node)
    weighted_entropy = w_left * left_entropy + w_right * right_entropy
    information_gain = node_entropy - weighted_entropy

    return information_gain

#garder la meilleur condition pour le meilleur split
def get_best_split_continue(X, y, node_indices):
    num_features = X.shape[1]

    best_feature = -1

    max_info_gain = 0
    tmax=0

    tab_max_feature=np.zeros(num_features)
    tab_min_feature=np.zeros(num_features)
    for loop in range(num_features):
        tab_max_feature[loop]=np.max(np.transpose(X)[loop])
        tab_min_feature[loop]=np.min(np.transpose(X)[loop])

    for feature in range(num_features):
        tab_t_feature=np.linspace(tab_min_feature[feature], tab_max_feature[feature], len(X)-1)

        for t in range(len(tab_t_feature)):
            info_gain = compute_information_gain_continue(X, y, node_indices, feature,tab_t_feature[t])

            if info_gain > max_info_gain:
                max_info_gain = info_gain
                best_feature = feature
                tmax=tab_t_feature[t]

    return best_feature,tmax,max_info_gain

#construction recursive de l'arbre de décision:
#on commence à la racine avec tout le dataset
#on teste les splits sur toutes les caractéristiques du vin avec un certain nombre de valeurs (ex: sulfate <= valeu
#on décide la condition choisie en calculant l'utilité = le gain d'information qui dépend de la pureté des noeuds r
#on sépare le dataset en deux et on refait récursivement la même chose sur les 2 nouveaux noeuds.
#on s'arrête lorsque un noeud est totalement pure (=tous les vins de meme qualité), ou à une certaine profondeur
def build_tree_recursive_continue(X, y, node_indices, branch_name, max_depth, current_depth, tree, arbreVin):
```

```
def build_tree_recursive_continue(X, y, node_indices, branch_name, max_depth, current_depth, tree, arbreVin):

    if current_depth == max_depth:
        qualite_node=np.mean(y[node_indices])
        formatting = " "*current_depth + "-"*current_depth
        print(formatting, "%s leaf node with indices" % branch_name, node_indices)
        print(formatting,"note moyenne attribuée à la feuille :",qualite_node,"(",round(np.mean(y[node_indices])),"

        arbreVin.qualite=round(qualite_node)

        return 0


    best_feature,tmax,max_info = get_best_split_continue(X, y, node_indices)
    arbreVin.carac=best_feature
    arbreVin.split=tmax

    formatting = "-"*current_depth
    print("%s Depth %d, %s: Split on feature: %s <= %s, pour un gain de %s" % (formatting, current_depth, branch_na

    left_indices, right_indices = split_dataset_continue(X, node_indices, best_feature,tmax)
    tree.append((left_indices, right_indices, best_feature,tmax))

    if(len(left_indices)>1):
        arbreVin.insert_gauche()
        build_tree_recursive_continue(X, y, left_indices, "Left", max_depth, current_depth+1, tree,arbreVin.enfant_
    if(len(right_indices)>1):
        arbreVin.insert_droit()
        build_tree_recursive_continue(X, y, right_indices, "Right", max_depth, current_depth+1, tree,arbreVin.enfan
    return tree
```

Test de l'arbre de décision avec une profondeur maximale de 4:

In [ ]:
```
tree = []
arbre = ArbreBinaireVin()
root_indices=list(range(0, len(X_train)))
build_tree_recursive_continue(X_train, y_train,root_indices, "Root", max_depth=4, current_depth=0, tree = tree, arb
```

```
/tmp/ipykernel_42683/506145069.py:13: RuntimeWarning: invalid value encountered in double_scalars
  impurity-=(tab_value[loop]/sum(tab_value))**2
```

```
 Depth 0, Root: Split on feature: sulphates <= -0.09272595611982037, pour un gain de 0.052469323042744875
- Depth 1, Left: Split on feature: total sulfur dioxide <= 0.853198588681813, pour un gain de 0.04485148373737924
-- Depth 2, Left: Split on feature: density <= 0.41867845114657687, pour un gain de 0.04332317923273732
--- Depth 3, Left: Split on feature: residual sugar <= 1.1300734780067812, pour un gain de 0.05027147181483804
    ---- Left leaf node with indices [2, 6, 13, 16, 17, 27, 33, 35, 37, 39, 43, 44, 46, 49, 56, 59, 63, 64, 70, 7
1, 73, 74, 76, 78, 79, 88, 93, 97, 103, 105, 110, 113, 118, 119, 120, 121, 125, 130, 136, 143, 150, 163, 164, 169,
173, 177, 179, 182, 183, 187, 188, 191, 194, 200, 209, 210, 215, 216, 219, 222, 225, 226, 228, 235, 236, 241, 243,
244, 248, 252, 255, 256, 260, 264, 266, 271, 272, 281, 283, 284, 286, 290, 291, 295, 306, 311, 314, 316, 320, 332,
335, 337, 344, 345, 355, 356, 365, 369, 374]
    ---- note moyenne attribuée à la feuille : 5.636363636363637 ( 6 )
    ---- Right leaf node with indices [42, 81, 92, 192, 279, 310, 324, 351]
    ---- note moyenne attribuée à la feuille : 5.875 ( 6 )
--- Depth 3, Right: Split on feature: total sulfur dioxide <= 0.5224497635627086, pour un gain de 0.05274334251606
9824
    ---- Left leaf node with indices [11, 22, 26, 29, 30, 55, 67, 91, 109, 111, 114, 124, 148, 158, 174, 178, 186,
190, 201, 229, 233, 238, 242, 245, 246, 247, 251, 262, 267, 285, 298, 307, 318, 339, 348, 366]
    ---- note moyenne attribuée à la feuille : 5.166666666666667 ( 5 )
    ---- Right leaf node with indices [53, 112, 153, 185, 224, 278, 349, 353]
    ---- note moyenne attribuée à la feuille : 5.625 ( 6 )
-- Depth 2, Right: Split on feature: fixed acidity <= -2.8656468717590577, pour un gain de 0.036946019743751335
--- Depth 3, Right: Split on feature: free sulfur dioxide <= 0.6597548370368371, pour un gain de 0.037524866296796
13
    ---- Left leaf node with indices [4, 5, 9, 57, 68, 80, 87, 123, 128, 217, 232, 240, 249, 265, 277, 303, 338, 3
57, 367]
    ---- note moyenne attribuée à la feuille : 5.315789473684211 ( 5 )
    ---- Right leaf node with indices [12, 47, 62, 65, 69, 104, 106, 116, 134, 137, 140, 146, 152, 154, 168, 176,
193, 206, 214, 254, 287, 299, 301, 333, 336, 362]
    ---- note moyenne attribuée à la feuille : 5.038461538461538 ( 5 )
- Depth 1, Right: Split on feature: citric acid <= 0.15676519921971788, pour un gain de 0.061485402736593864
-- Depth 2, Left: Split on feature: pH <= 0.3387519339878331, pour un gain de 0.05154936838208535
--- Depth 3, Left: Split on feature: density <= 0.6547929676033211, pour un gain de 0.1570550931430834
    ---- Left leaf node with indices [0, 19, 32, 51, 52, 58, 61, 66, 115, 122, 138, 159, 160, 184, 258, 261, 276,
315, 317, 325, 327, 329, 330, 375]
    ---- note moyenne attribuée à la feuille : 5.625 ( 6 )
    ---- Right leaf node with indices [25, 86, 94, 142, 151]
    ---- note moyenne attribuée à la feuille : 6.0 ( 6 )
--- Depth 3, Right: Split on feature: fixed acidity <= -2.350583798240976, pour un gain de 0.1004821389436773
    ---- Left leaf node with indices [135, 141, 308]
    ---- note moyenne attribuée à la feuille : 7.0 ( 7 )
    ---- Right leaf node with indices [10, 20, 21, 24, 40, 45, 101, 102, 107, 132, 133, 144, 157, 165, 172, 203, 2
08, 213, 218, 220, 223, 227, 250, 274, 288, 289, 300, 309, 321, 334, 340, 347, 350, 364, 372, 373]
```

```
    ---- note moyenne attribuée à la feuille : 5.75 ( 6 )
-- Depth 2, Right: Split on feature: total sulfur dioxide <= 0.7429489803087783, pour un gain de 0.027252765090602
815
--- Depth 3, Left: Split on feature: density <= 0.4029374833827939, pour un gain de 0.03418658363713323
    ---- Left leaf node with indices [1, 3, 7, 8, 15, 18, 23, 28, 34, 38, 48, 50, 54, 60, 75, 77, 83, 85, 89, 98,
99, 100, 117, 126, 129, 147, 155, 161, 162, 166, 180, 199, 202, 204, 207, 230, 231, 253, 263, 268, 270, 280, 296,
304, 322, 323, 326, 328, 341, 342, 343, 346, 370, 371]
    ---- note moyenne attribuée à la feuille : 6.888888888888889 ( 7 )
    ---- Right leaf node with indices [14, 36, 41, 72, 96, 127, 145, 149, 156, 171, 175, 181, 196, 197, 198, 211,
212, 221, 234, 237, 257, 259, 269, 275, 293, 294, 302, 312, 313, 319, 352, 354, 359, 360, 361, 363, 368]
    ---- note moyenne attribuée à la feuille : 6.486486486486487 ( 6 )
--- Depth 3, Right: Split on feature: total sulfur dioxide <= 1.245197196230381, pour un gain de 0.132500000000000
17
    ---- Left leaf node with indices [82, 84, 90, 95, 139, 167, 189, 195, 205, 282, 305, 331]
    ---- note moyenne attribuée à la feuille : 5.916666666666667 ( 6 )
    ---- Right leaf node with indices [31, 108, 131, 170, 239, 292, 297, 358]
    ---- note moyenne attribuée à la feuille : 6.25 ( 6 )
```

In [ ]:
```
x=X_train[50]
print("vecteur x",x)
print("note prédite:",arbre.get_predictionVin(x))
print("vraie note :",y_train[50])
```

```
vecteur x [ 1.65989152 -0.56212655  1.29038469 -1.73235125  0.93061168  0.88161709
  0.30550152  0.39591505 -1.94880838  0.87331486]
note prédite: 7
vraie note : 7
```

On a construit l'arbre de profondeur max 4, il ne reste plus qu'à faire passer notre jeu de test dedans.

In [ ]:
```
def calcul_error(yhat,y_test):

    cost = 0.0
    m=len(y_test)
    for i in range(m):
        cost = cost + (y_test[i] - yhat[i])**2   #scalar
    cost = cost / (2 * m)                  #scalar
    return cost

def predict_model(arbre,X,y):
    yhat=[]
    for loop in range(len(X)):
        yhat.append(arbre.get_predictionVin(X[loop]))

    sum=0
    for loop in range(len(yhat)):
        if(y[loop] == yhat[loop]):
            sum+=1
    print('Train Accuracy (%): ',(sum/len(yhat))*100)

    nn_test_error =calcul_error(yhat,y)

    print(f"Set Classification Error: {nn_test_error:.4f}")
```

In [ ]:
```
predict_model(arbre,X_test,y_test)
```

```
Train Accuracy (%):  53.96825396825397
Set Classification Error: 0.3135
```

Nous obtenons 53% de prédictions exactes sur le jeu de test. Pour améliorer ce résultat, nous allons utiliser une version évoluée de l'arbre de décision : la Random Forest.

c) Forêt d'arbres décisionnels pour la qualité du vin

Nous allons enfin mettre en place une random forest ou forêt d'arbres décisionnels à l'aide des bibliothèques sklearn et xgboost. Random forest va créer une multitude d'arbres de décision de ce type avec une part d'aléatoire dans le choix des splits et choisir l'arbre final grâce à un système de votes.

In [ ]:
```
import numpy as np
import pandas as pd
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from xgboost import XGBClassifier
import matplotlib.pyplot as plt
plt.style.use('./deeplearning.mplstyle')

RANDOM_STATE = 55 ## We will pass it to every sklearn call so we ensure reproducibility

from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
y_train = le.fit_transform(y_train)
y_test=le.fit_transform(y_test)
```

In [ ]:
```
xgb_model = XGBClassifier(n_estimators = 500, learning_rate = 0.1,verbosity = 1, random_state = RANDOM_STATE)
xgb_model.fit(X_train,y_train, eval_set = [(X_test,y_test)], early_stopping_rounds = 10)
```

```
[0]     validation_0-mlogloss:1.76413
[1]     validation_0-mlogloss:1.74212
[2]     validation_0-mlogloss:1.74340
[3]     validation_0-mlogloss:1.72256
/home/henri/.local/lib/python3.8/site-packages/xgboost/sklearn.py:835: UserWarning: `early_stopping_rounds` in `fi
t` method is deprecated for better compatibility with scikit-learn, use `early_stopping_rounds` in constructor or`
set_params` instead.
  warnings.warn(
```

```
[4]     validation_0-mlogloss:1.72616
[5]     validation_0-mlogloss:1.69494
[6]     validation_0-mlogloss:1.69315
[7]     validation_0-mlogloss:1.69298
[8]     validation_0-mlogloss:1.70397
[9]     validation_0-mlogloss:1.72146
[10]    validation_0-mlogloss:1.72980
[11]    validation_0-mlogloss:1.74967
[12]    validation_0-mlogloss:1.76279
[13]    validation_0-mlogloss:1.77983
[14]    validation_0-mlogloss:1.78464
[15]    validation_0-mlogloss:1.79472
[16]    validation_0-mlogloss:1.79150
```

Out[ ]:

▾ XGBClassifier

```
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric=None, feature_types=None,
              gamma=None, gpu_id=None, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=0.1, max_bin=None,
              max_cat_threshold=None, max_cat_to_onehot=None,
              max_delta_step=None, max_depth=None, max_leaves=None,
              min_child_weight=None, missing=nan, monotone_constraints=None,
```

In [ ]:
```
xgb_model.best_iteration
```

Out[ ]: 7

In [ ]:
```
print(f"Metrics train:\n\tAccuracy score: {accuracy_score(xgb_model.predict(X_train),y_train):.4f}\nMetrics test:\n

print(xgb_model.classes_)
#print(xgb_model.classes_)
```

```
Metrics train:
        Accuracy score: 0.8093
Metrics test:
        Accuracy score: 0.8093
[0 1 2 3 4 5]
```

Le modèle random Forest nous permet d'obtenir sur le jeu de test 80% de précision, c'est le meilleur score obtenu. On choisit donc cette méthode.

5) Utilisation de sklearn pour différentes méthodes

a) Data exploration

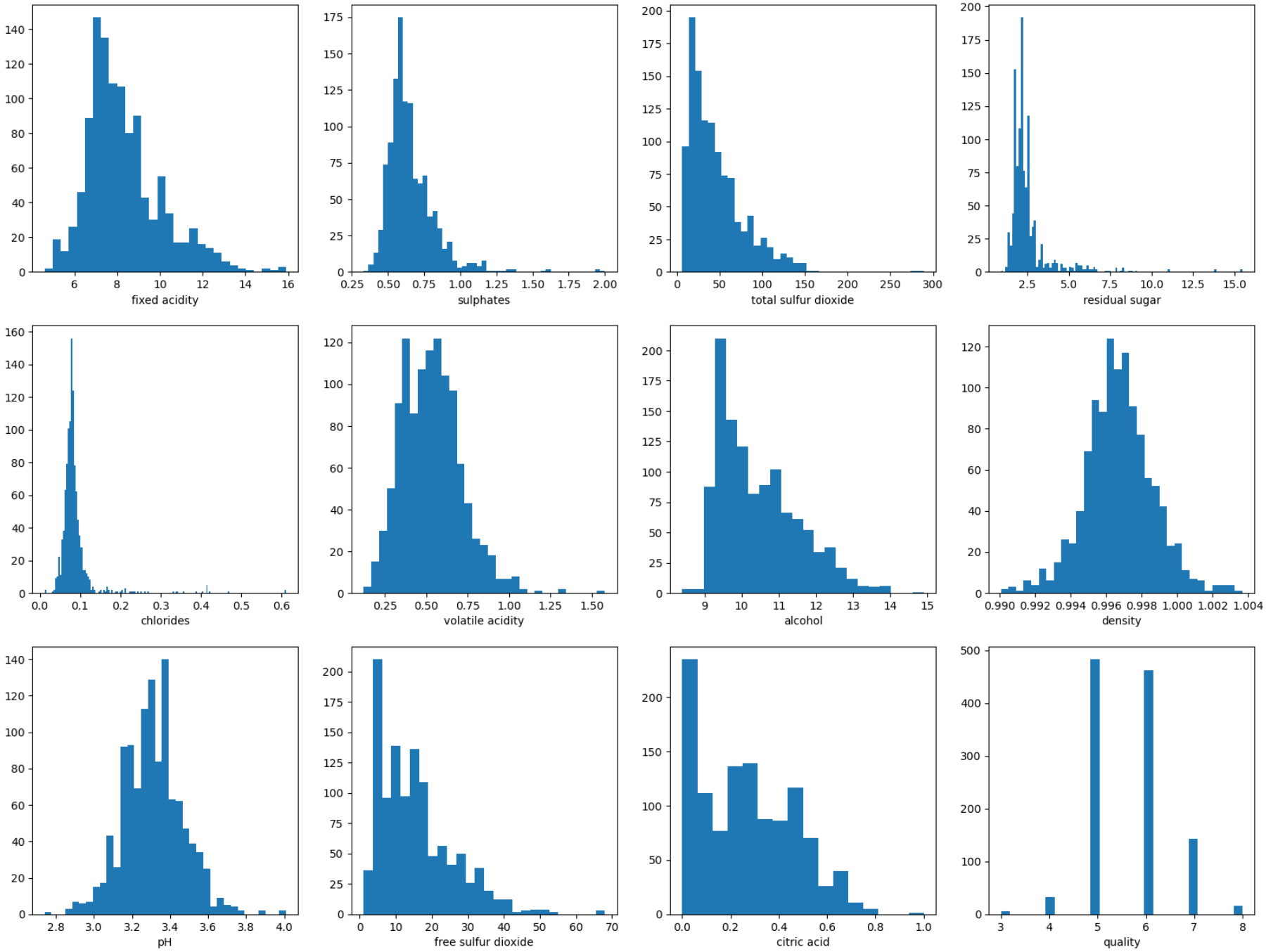
In [ ]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

In [ ]:
```
df = pd.read_csv("WineQT.csv", sep=",")
df = df.drop(columns="Id")
print(df.describe())
```

```
       fixed acidity  volatile acidity  citric acid  residual sugar  \
count    1143.000000       1143.000000  1143.000000     1143.000000
mean        8.311111          0.531339     0.268364        2.532152
std         1.747595          0.179633     0.196686        1.355917
min         4.600000          0.120000     0.000000        0.900000
25%         7.100000          0.392500     0.090000        1.900000
50%         7.900000          0.520000     0.250000        2.200000
75%         9.100000          0.640000     0.420000        2.600000
max        15.900000          1.580000     1.000000       15.500000

         chlorides  free sulfur dioxide  total sulfur dioxide      density  \
count  1143.000000          1143.000000           1143.000000  1143.000000
mean      0.086933            15.615486             45.914698     0.996730
std       0.047267            10.250486             32.782130     0.001925
min       0.012000             1.000000              6.000000     0.990070
25%       0.070000             7.000000             21.000000     0.995570
50%       0.079000            13.000000             37.000000     0.996680
75%       0.090000            21.000000             61.000000     0.997845
max       0.611000            68.000000            289.000000     1.003690

                pH    sulphates      alcohol      quality
count  1143.000000  1143.000000  1143.000000  1143.000000
mean      3.311015     0.657708    10.442111     5.657043
std       0.156664     0.170399     1.082196     0.805824
min       2.740000     0.330000     8.400000     3.000000
25%       3.205000     0.550000     9.500000     5.000000
50%       3.310000     0.620000    10.200000     6.000000
75%       3.400000     0.730000    11.100000     6.000000
max       4.010000     2.000000    14.900000     8.000000
```

In [ ]:
```
def plot_hists(df):
    _, ax = plt.subplots(nrows=3, ncols=4, figsize=(20, 15))
    for n in range(12):
        i = n % 3
        j = n % 4
        ax[i, j].hist(df.iloc[:, n], bins='auto')
        ax[i, j].set_xlabel(df.columns[n])
```

In [ ]:
```
plot_hists(df)
```

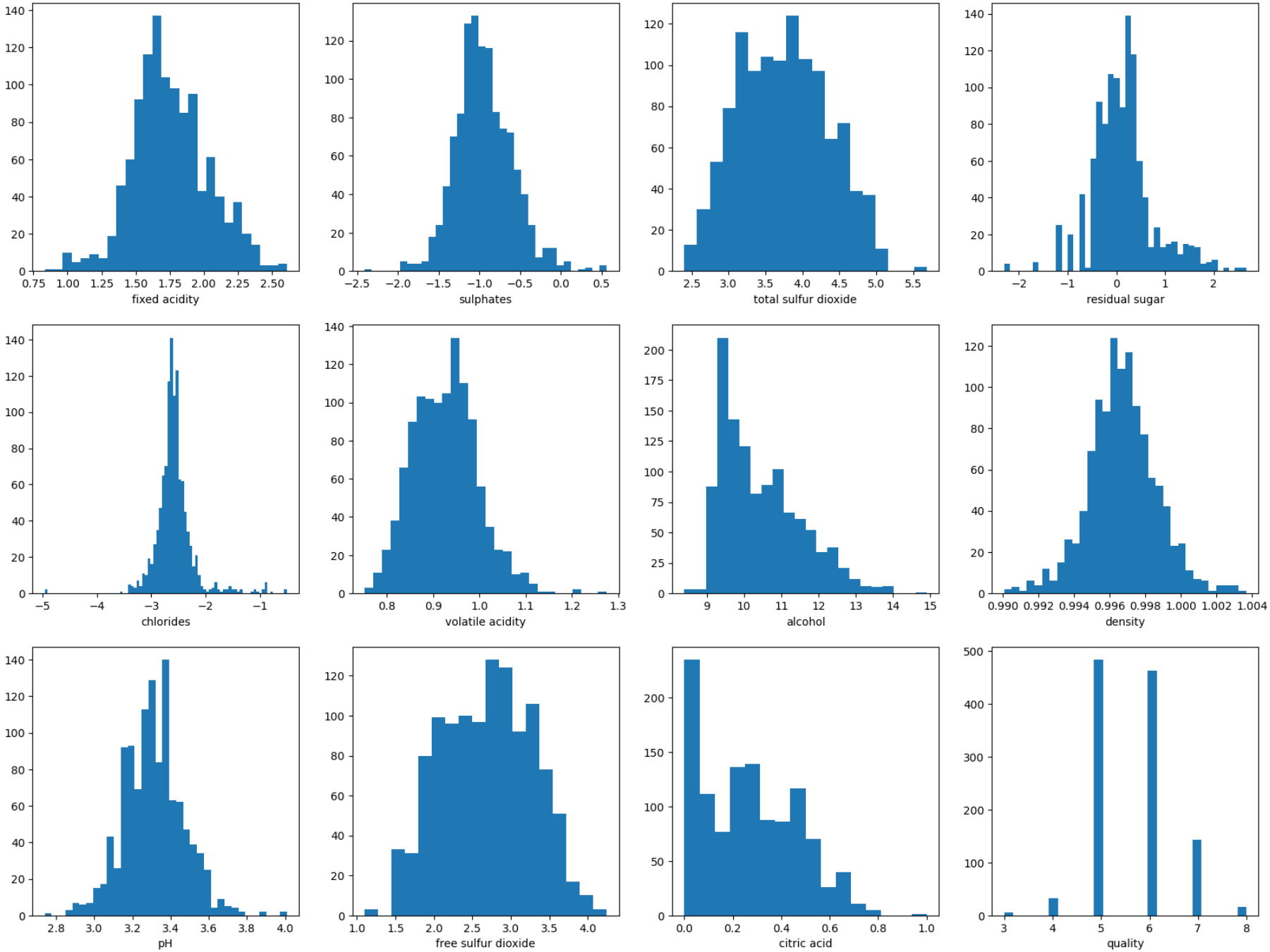

fixed acidity
sulphates
total sulfur dioxide
residual sugar
chlorides
volatile acidity
alcohol
density
pH
free sulfur dioxide
citric acid
quality

b) Data preprocessing

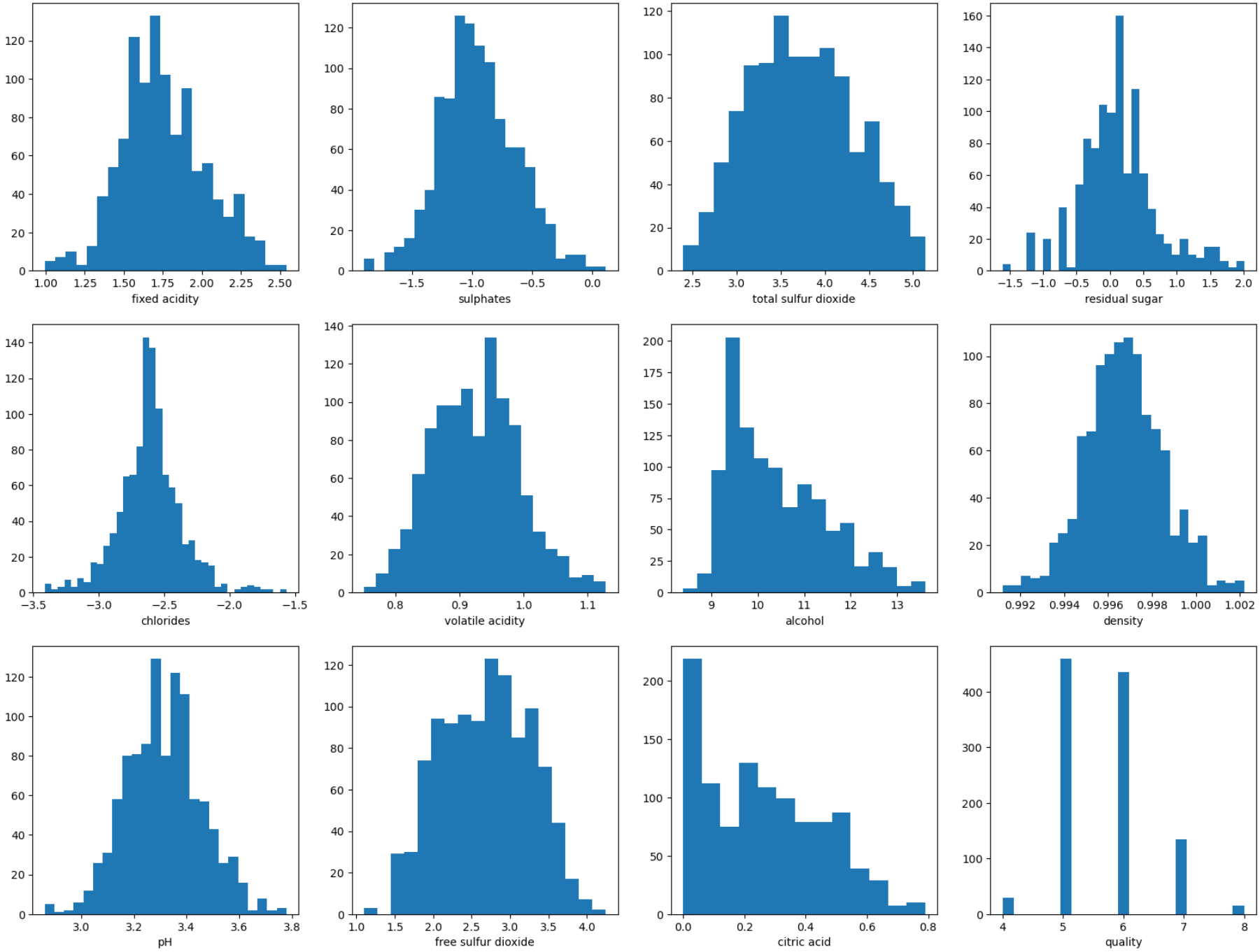
Anomaly detection:

In [ ]:
```
def normalize(df, property, parameter):
    df[property] = np.log(df[property] + parameter)

normalize(df, "fixed acidity", -2.3)
normalize(df, "sulphates", -0.24)
normalize(df, "total sulfur dioxide", 5)
normalize(df, "residual sugar", -1.1)
normalize(df, "chlorides", -0.005)
normalize(df, "volatile acidity", 2)
normalize(df, "free sulfur dioxide", 2)
plot_hists(df);
```

```
d:\Programs\Miniconda\envs\env\Lib\site-packages\pandas\core\arraylike.py:402: RuntimeWarning: invalid value encou
ntered in log
  result = getattr(ufunc, method)(*inputs, **kwargs)
```

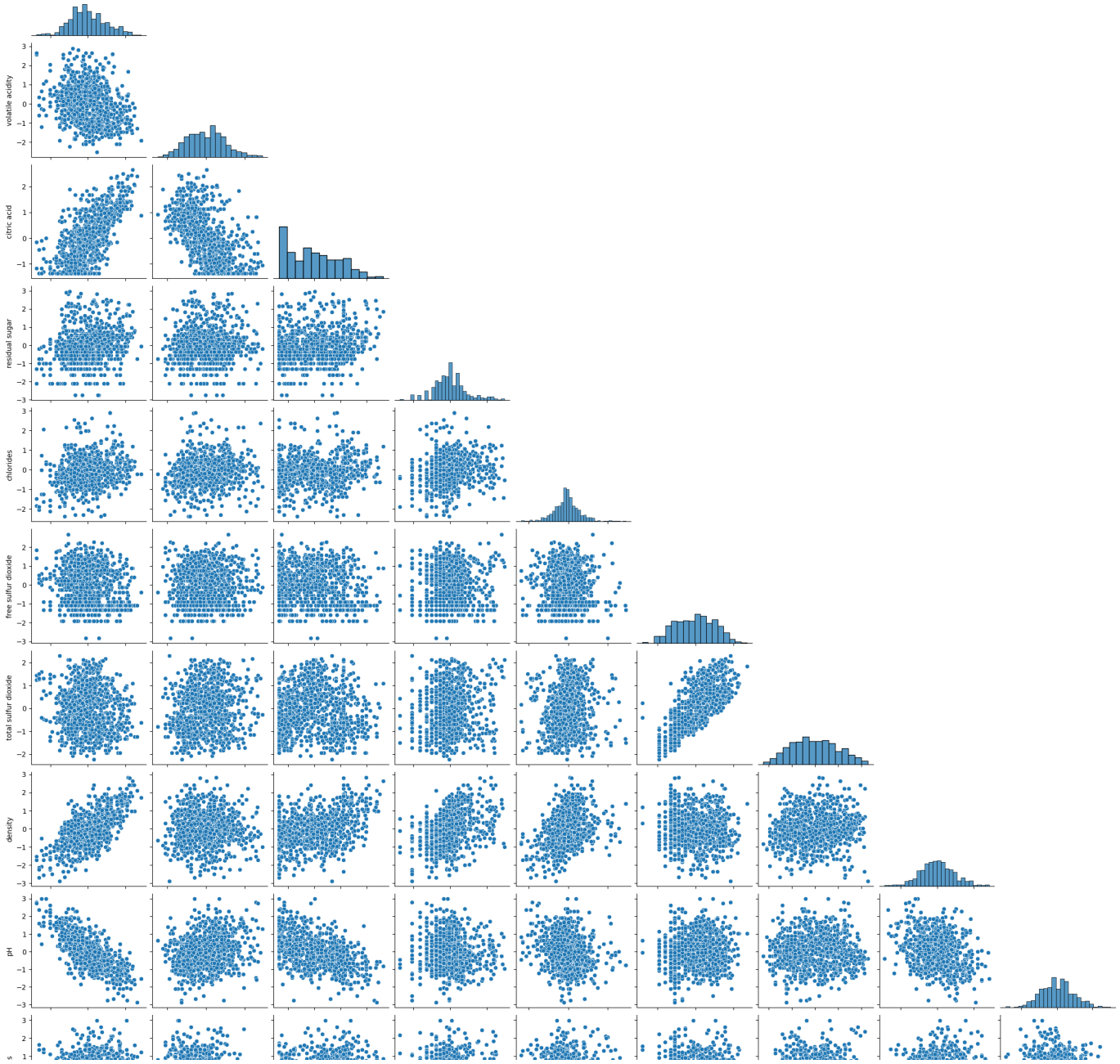

fixed acidity
sulphates
total sulfur dioxide
residual sugar
chlorides
volatile acidity
alcohol
density
pH
free sulfur dioxide
citric acid
quality

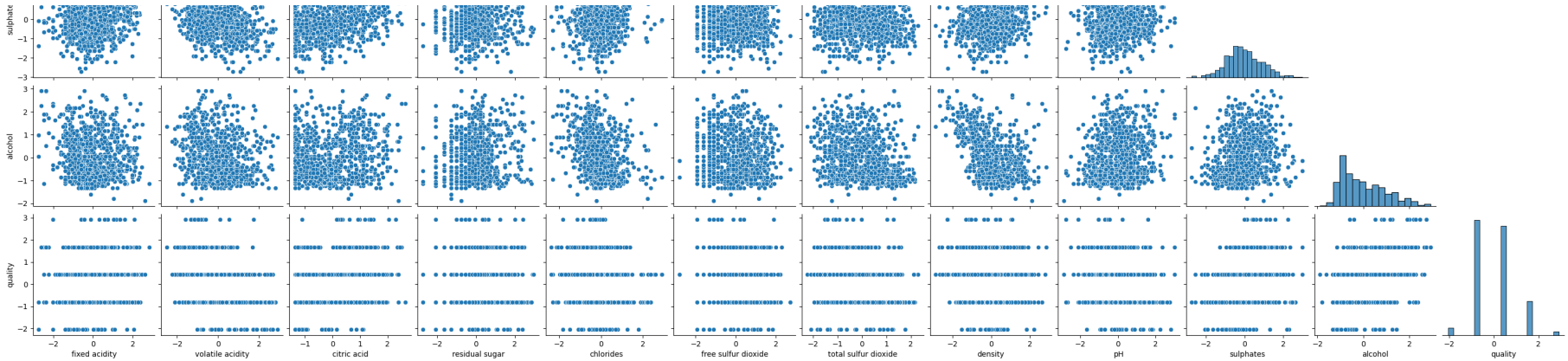
In [ ]:
```
standardized = (df - df.mean()) / df.std()
standardized = standardized[(np.abs(standardized) < 3).all(axis=1)]
rows = np.setdiff1d(list(df.index), list(standardized.index))
df.drop(index=rows, inplace=True) # Delete observations with std >= 3
plot_hists(df)
```


fixed acidity
sulphates
total sulfur dioxide
residual sugar
chlorides
volatile acidity
alcohol
density
pH
free sulfur dioxide
citric acid
quality

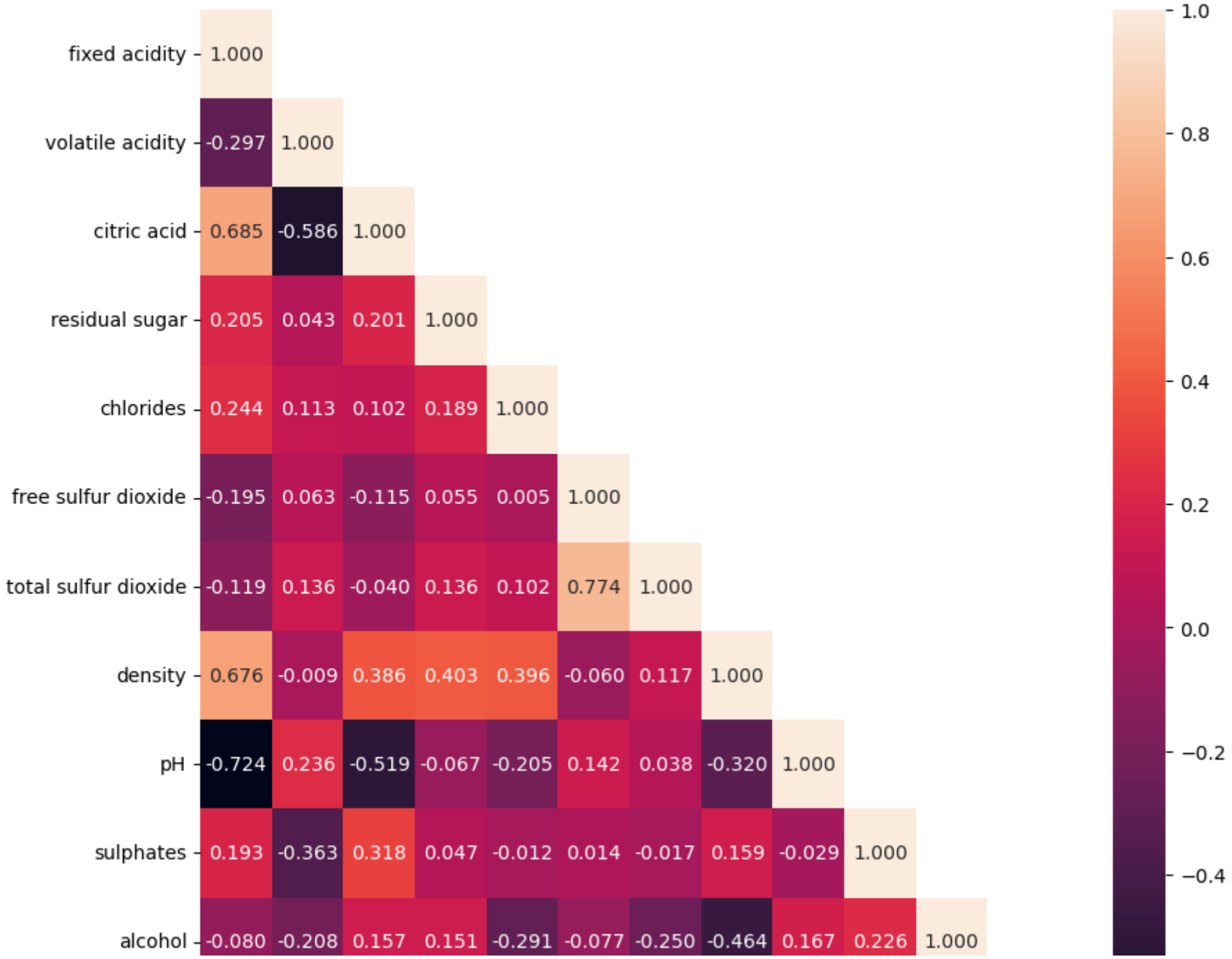
Mulitcolinearity detection

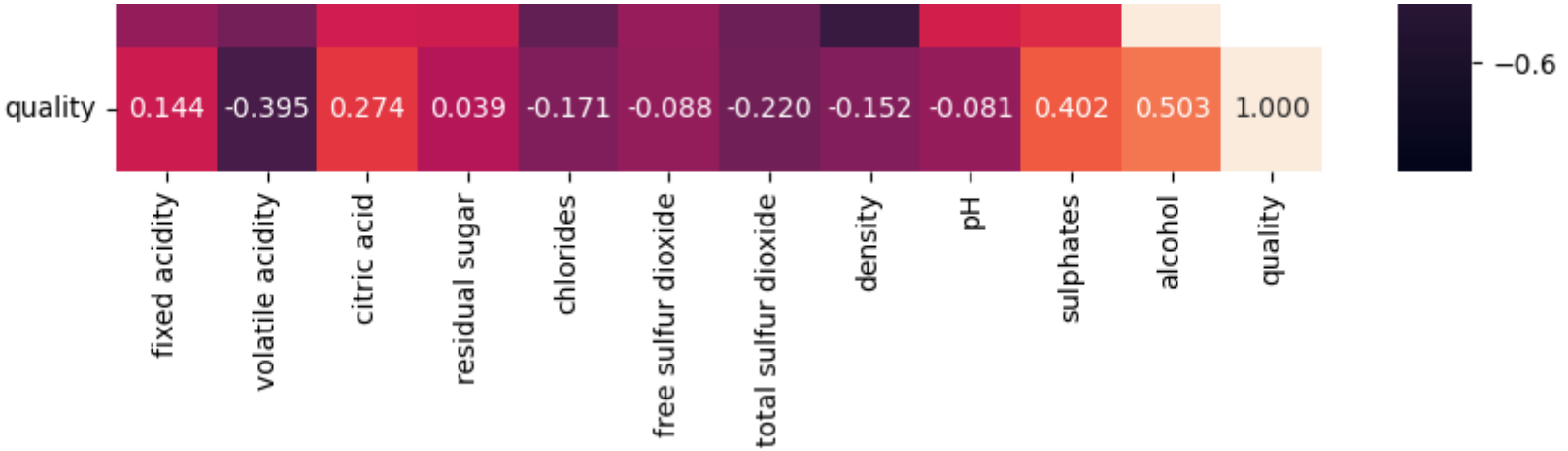
In [ ]:
```
sns.pairplot(standardized, corner=True);
```


volatile acidity
citric acid
residual sugar
chlorides
free sulfur dioxide
total sulfur dioxide
density
pH



In [ ]:
```
corr = standardized.corr()
matrix = np.triu(corr, 1)
fig, ax = plt.subplots(figsize=(10, 10))
ax = sns.heatmap(corr, annot=True, fmt=".3f", mask=matrix)
```


fixed acidity 1.000
volatile acidity -0.297 1.000
citric acid 0.685 -0.586 1.000
residual sugar 0.205 0.043 0.201 1.000
chlorides 0.244 0.113 0.102 0.189 1.000
free sulfur dioxide -0.195 0.063 -0.115 0.055 0.005 1.000
total sulfur dioxide -0.119 0.136 -0.040 0.136 0.102 0.774 1.000
density 0.676 -0.009 0.386 0.403 0.396 -0.060 0.117 1.000
pH -0.724 0.236 -0.519 -0.067 -0.205 0.142 0.038 -0.320 1.000
sulphates 0.193 -0.363 0.318 0.047 -0.012 0.014 -0.017 0.159 -0.029 1.000
alcohol -0.080 -0.208 0.157 0.151 -0.291 -0.077 -0.250 -0.464 0.167 0.226 1.000
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4



In [ ]:
```
np.fill_diagonal(corr.values, 0)
def delete(corr, df): # Delete all labels with coef >= 0.7
    tmp = corr.loc[:, (np.abs(corr) >= 0.7).any()]
    print((tmp.columns))
    if (len(tmp.columns) > 0):
        label = np.abs(tmp).sum().sort_values(ascending=False).index[0]
        corr.drop(index=label, columns=label, inplace=True)
        df.drop(columns=label, inplace=True)
        delete(corr, df)
delete(corr, df)
print(df.describe());
```

```
Index(['fixed acidity', 'free sulfur dioxide', 'total sulfur dioxide', 'pH'], dtype='object')
Index(['free sulfur dioxide', 'total sulfur dioxide'], dtype='object')
Index([], dtype='object')
       volatile acidity  citric acid  residual sugar    chlorides  \
count       1074.000000  1074.000000     1074.000000  1074.000000
mean           0.925244     0.262793        0.117516    -2.603734
std            0.067709     0.191576        0.577733     0.242205
min            0.751416     0.000000       -1.609438    -3.411248
25%            0.871293     0.090000       -0.223144    -2.733368
50%            0.924259     0.250000        0.095310    -2.603690
75%            0.970779     0.420000        0.405465    -2.476938
max            1.128171     0.790000        2.014903    -1.565421

       free sulfur dioxide      density           pH    sulphates  \
count          1074.000000  1074.000000  1074.000000  1074.000000
mean              2.702034     0.996727     3.314674    -0.952988
std               0.565918     0.001769     0.144323     0.321535
min               1.098612     0.991200     2.860000    -1.897120
25%               2.197225     0.995600     3.210000    -1.171183
50%               2.708050     0.996680     3.310000    -0.967584
75%               3.135494     0.997800     3.400000    -0.733969
max               4.248495     1.002200     3.780000     0.113329

           alcohol      quality
count  1074.000000  1074.000000
mean     10.432294     5.670391
std       1.029746     0.782687
min       8.400000     4.000000
25%       9.500000     5.000000
50%      10.200000     6.000000
75%      11.100000     6.000000
max      13.600000     8.000000
```

Feature scaling

In [ ]:
```
from sklearn.model_selection import train_test_split

X = np.c_[df.iloc[:, :-1].to_numpy(), np.ones(df.shape[0])]
y = df["quality"].to_numpy()

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

In [ ]:
```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler().fit(X_train)

X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)
```

c) Model selection

In [ ]:
```
from sklearn.model_selection import KFold
from sklearn.model_selection import GridSearchCV

cv = KFold(n_splits=10)
```

K-Neighbors

In [ ]:
```
from sklearn.neighbors import KNeighborsClassifier

parameters = {
    'n_neighbors': [10, 11, 12, 13, 14, 15],
    'weights': ('uniform', 'distance'),
    'p': [1, 2]
}
knc = KNeighborsClassifier(algorithm='auto')
clf = GridSearchCV(knc, parameters, scoring='accuracy', cv=cv)
clf.fit(X_train, y_train);
```

In [ ]:
```
print(f"Best score : {clf.best_score_}")
print(f"Best parameters : {clf.best_params_}")
```

```
Best score : 0.672859097127223
Best parameters : {'n_neighbors': 11, 'p': 1, 'weights': 'distance'}
```

SVC

In [ ]:
```
from sklearn.svm import SVC

parameters = {
    'kernel': ('linear', 'rbf'),
    'C': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
}
svc = SVC()
clf = GridSearchCV(svc, parameters, scoring='accuracy', cv=cv)
clf.fit(X_train, y_train);
```

In [ ]:
```
print(f"Best score : {clf.best_score_}")
print(f"Best parameters : {clf.best_params_}")
```

```
Best score : 0.6332831737346101
Best parameters : {'C': 5, 'kernel': 'rbf'}
```

Random Forest

In [ ]:
```
from sklearn.ensemble import RandomForestClassifier

parameters = {
    'n_estimators': [100, 200, 300],
    'criterion': ('gini', 'entropy', 'log_loss')
}
rfc = RandomForestClassifier()
clf = GridSearchCV(rfc, parameters, scoring='accuracy', cv=cv)
clf.fit(X_train, y_train);
```

In [ ]:
```
print(f"Best score : {clf.best_score_}")
print(f"Best parameters : {clf.best_params_}")
```

```
Best score : 0.6857318741450069
Best parameters : {'criterion': 'entropy', 'n_estimators': 200}
```

XGBoost

In [ ]:
```python
from xgboost import XGBClassifier

y_map = y_train - 4

parameters = {
    "learning_rate": [0.1, 0.3, 0.6, 1],
    "max_depth": [5, 6, 7, 8, 9, 10]
}
xgbc = XGBClassifier(objective="multi:softmax", random_state=42)
clf = GridSearchCV(xgbc, parameters, scoring='accuracy', cv=cv)
clf.fit(X_train, y_map);
```

In [ ]:
```python
print(f"Best score : {clf.best_score_}")
print(f"Best parameters : {clf.best_params_}")
```

```
Best score : 0.6601231190150479
Best parameters : {'learning_rate': 0.6, 'max_depth': 9}
```

d) Model evaluation

In [ ]:
```python
from sklearn.metrics import accuracy_score, f1_score

clt = RandomForestClassifier(n_estimators=200, criterion='entropy')
clt.fit(X_train, y_train)
y_pred = clt.predict(X_test)
```

In [ ]:
```
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy : {accuracy}")
f1 = f1_score(y_test, y_pred, average='weighted')
print(f"F1-score : {f1}")
```

```
Accuracy : 0.6372093023255814
F1-score : 0.6203341847622335
```