KNN avec GridSearchCV

November 22, 2020

[1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

[2]:
```
data = pd.read_csv('winequality-red.csv', sep=";")
```

[3]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1599 entries, 0 to 1598
Data columns (total 12 columns):
 #   Column                Non-Null Count  Dtype  
---  ------                --------------  -----  
 0   fixed acidity         1599 non-null   float64
 1   volatile acidity      1599 non-null   float64
 2   citric acid           1599 non-null   float64
 3   residual sugar        1599 non-null   float64
 4   chlorides             1599 non-null   float64
 5   free sulfur dioxide   1599 non-null   float64
 6   total sulfur dioxide  1599 non-null   float64
 7   density               1599 non-null   float64
 8   pH                    1599 non-null   float64
 9   sulphates             1599 non-null   float64
 10  alcohol               1599 non-null   float64
 11  quality               1599 non-null   int64  
dtypes: float64(11), int64(1)
memory usage: 150.0 KB
```

[4]:
```
data.head(13)
```

[4]:

	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	
5	7.4	0.660	0.00	1.8	0.075	
6	7.9	0.600	0.06	1.6	0.069	

```
7             7.3             0.650         0.00             1.2      0.065
8             7.8             0.580         0.02             2.0      0.073
9             7.5             0.500         0.36             6.1      0.071
10            6.7             0.580         0.08             1.8      0.097
11            7.5             0.500         0.36             6.1      0.071
12            5.6             0.615         0.00             1.6      0.089

    free sulfur dioxide  total sulfur dioxide  density    pH  sulphates  \
0                  11.0                  34.0   0.9978  3.51       0.56
1                  25.0                  67.0   0.9968  3.20       0.68
2                  15.0                  54.0   0.9970  3.26       0.65
3                  17.0                  60.0   0.9980  3.16       0.58
4                  11.0                  34.0   0.9978  3.51       0.56
5                  13.0                  40.0   0.9978  3.51       0.56
6                  15.0                  59.0   0.9964  3.30       0.46
7                  15.0                  21.0   0.9946  3.39       0.47
8                   9.0                  18.0   0.9968  3.36       0.57
9                  17.0                 102.0   0.9978  3.35       0.80
10                 15.0                  65.0   0.9959  3.28       0.54
11                 17.0                 102.0   0.9978  3.35       0.80
12                 16.0                  59.0   0.9943  3.58       0.52

    alcohol  quality
0       9.4        5
1       9.8        5
2       9.8        5
3       9.8        6
4       9.4        5
5       9.4        5
6       9.4        5
7      10.0        7
8       9.5        7
9      10.5        5
10      9.2        5
11     10.5        5
12      9.9        5
```

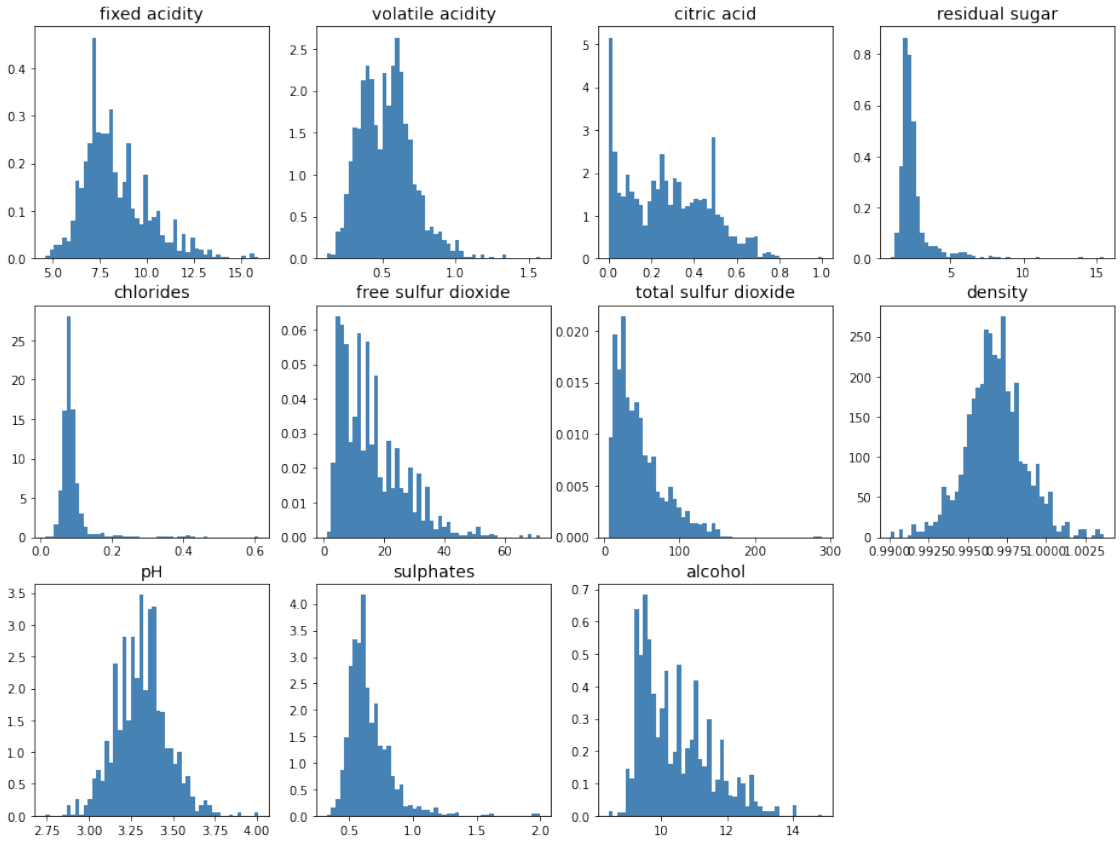
Nos données contiennent 11 colonnes, 10 qui correspondent à divers **indicateurs physico-chimiques** et 1 qui est **la qualité du vin** .

Nous allons extraire **deux arrays numpy de ces données**, un qui contient les points et l'autre qui contient les étiquettes

[5]:
```
X = data[data.columns[:-1]].values
y = data['quality'].values
```

On peut maintenant afficher **un histogramme** pour chacune de nos **variables** :

[6]:
```
fig = plt.figure(figsize=(16, 12))
for feat_idx in range(X.shape[1]):
    ax = fig.add_subplot(3,4, (feat_idx+1))
    h = ax.hist(X[:, feat_idx], bins=50, color='steelblue', density=True,␣
↪edgecolor='none')
    ax.set_title(data.columns[feat_idx], fontsize=14)
```



On remarque en particulier que ces variables prennent des valeurs dans des ensembles différents. Par exemple, **sulphates** varie de **0 à 1** tandis que **total sulfur dioxide** varie de **0 à 440** . Il va donc nous falloir **standardiser les données** pour que la deuxième ne domine pas complètement la première.

Nous allons commencer par transformer ce problème en un problème de classification : il s'agira de séparer les bons vins des vins médiocres :

[7]:
```
y_class = np.where(y<6, 0, 1)
```

[8]:
```
y_class
```

[8]:
```
array([0, 0, 0, …, 1, 0, 1])
```

Séparons nos données en **un jeu d'entraînement** et **un jeu de test**. Le jeu de test contiendra

30% des données .

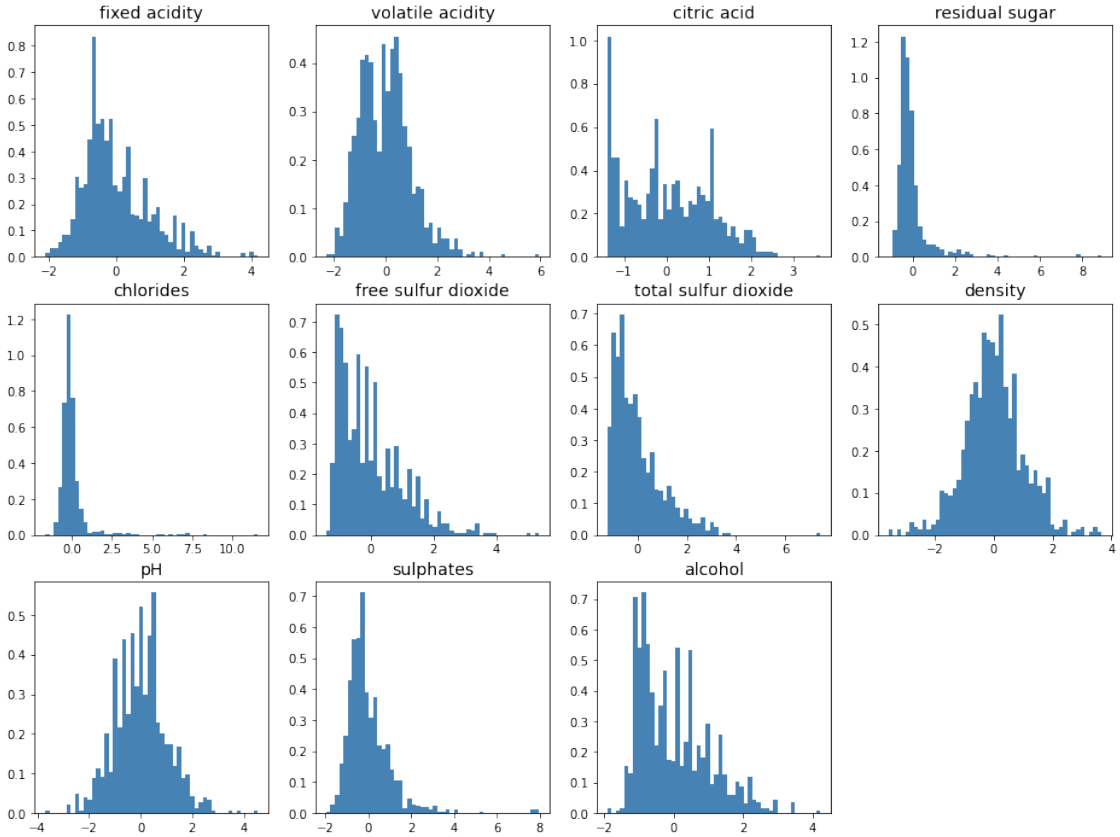
[9]:
```
from sklearn import model_selection

X_train, X_test, y_train, y_test = model_selection.train_test_split(X,␣
↪y_class,test_size=0.3) # 30% des données dans le jeu de test
```

Nous pouvons maintenant standardiser les données d'entraînement et appliquer la même transformation aux données de test :

[10]:
```
from sklearn import preprocessing
std_scale = preprocessing.StandardScaler().fit(X_train)
X_train_std = std_scale.transform(X_train)
X_test_std = std_scale.transform(X_test)
```

On peut visualiser de nouveau les données pour vérifier que les différentes variables prennent des valeurs qui ont maintenant des ordres de grandeur similaires.

[11]:
```
fig = plt.figure(figsize=(16, 12))
for feat_idx in range(X_train_std.shape[1]):
    ax = fig.add_subplot(3,4, (feat_idx+1))
    h = ax.hist(X_train_std[:, feat_idx], bins=50, color = 'steelblue',␣
↪density=True, edgecolor='none')
    ax.set_title(data.columns[feat_idx], fontsize=14)
```



[12]:
```
from sklearn import neighbors, metrics

# Fixer les valeurs des hyperparamètres à tester
param_grid = {'n_neighbors':[3, 5, 7, 9, 11, 13, 15]}

# Choisir un score à optimiser, ici l'accuracy (proportion de prédictions␣
 ↪correctes)
score = 'accuracy'

# Créer un classifieur kNN avec recherche d'hyperparamètre par validation␣
 ↪croisée
clf = model_selection.GridSearchCV(
    neighbors.KNeighborsClassifier(), # un classifieur kNN
    param_grid,     # hyperparamètres à tester
    cv=5,           # nombre de folds de validation croisée
    scoring=score   # score à optimiser
)

# Optimiser ce classifieur sur le jeu d'entraînement
clf.fit(X_train_std, y_train)

# Afficher le(s) hyperparamètre(s) optimaux
print("Meilleur(s) hyperparamètre(s) sur le jeu d'entraînement:")
print(clf.best_params_)

# Afficher les performances correspondantes
print("Résultats de la validation croisée :")
for mean, std, params in zip(
        clf.cv_results_['mean_test_score'], # score moyen
        clf.cv_results_['std_test_score'],  # écart-type du score
        clf.cv_results_['params']           # valeur de l'hyperparamètre
    ):

    print("{} = {:.3f} (+/-{:.03f}) for {}".format(
        score,
        mean,
        std*2,
        params
    ) )
```

```
Meilleur(s) hyperparamètre(s) sur le jeu d'entraînement:
{'n_neighbors': 15}
Résultats de la validation croisée :
accuracy = 0.702 (+/-0.049) for {'n_neighbors': 3}
accuracy = 0.702 (+/-0.037) for {'n_neighbors': 5}
```

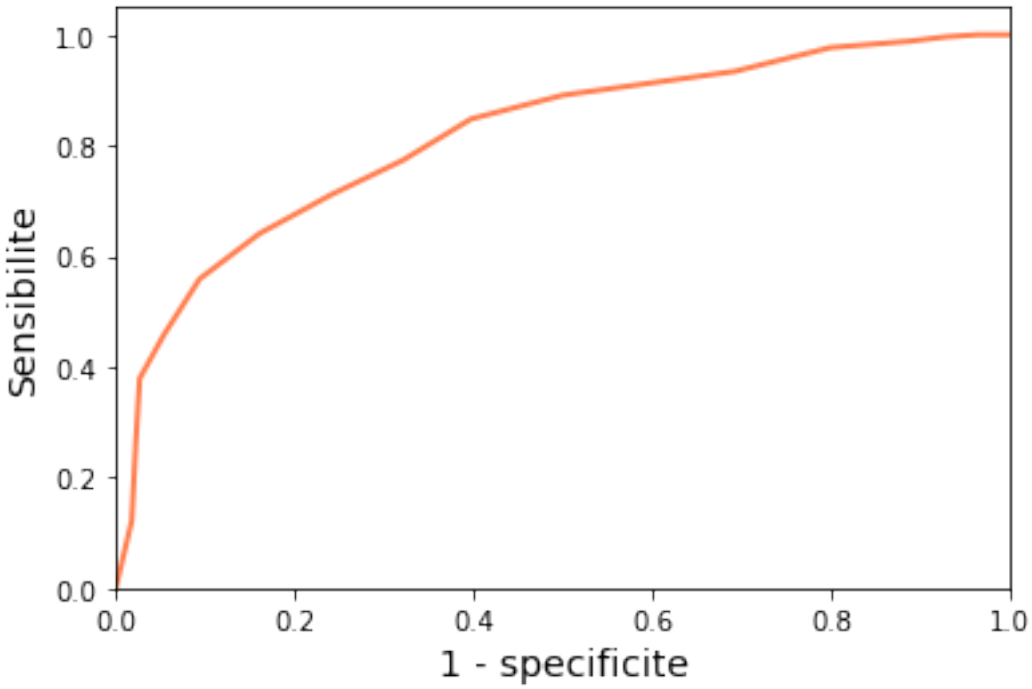
```
accuracy = 0.705 (+/-0.035) for {'n_neighbors': 7}
accuracy = 0.707 (+/-0.042) for {'n_neighbors': 9}
accuracy = 0.722 (+/-0.035) for {'n_neighbors': 11}
accuracy = 0.726 (+/-0.017) for {'n_neighbors': 13}
accuracy = 0.733 (+/-0.017) for {'n_neighbors': 15}
```

[13]:
```
y_pred = clf.predict(X_test_std)
print("\n Accuracy : Sur le jeu de test : {:.3f}".format(metrics.
 ↪accuracy_score(y_test, y_pred)))
```

```
 Accuracy : Sur le jeu de test : 0.729
```

[14]:
```
y_pred_proba = clf.predict_proba(X_test_std)[:, 1]
[fpr, tpr, thr] = metrics.roc_curve(y_test, y_pred_proba)
plt.plot(fpr, tpr, color='coral', lw=2)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('1 - specificite', fontsize=14)
plt.ylabel('Sensibilite', fontsize=14)
```

[14]:
```
Text(0, 0.5, 'Sensibilite')
```



[15]:
```
print(metrics.auc(fpr, tpr))
```

```
0.816162109375
```

[16]:
```
# indice du premier seuil pour lequel
# la sensibilité est supérieure à 0.95
idx = np.min(np.where(tpr > 0.95))

print("Sensibilité : {:.2f}".format(tpr[idx]))
print("Spécificité : {:.2f}".format(1-fpr[idx]))
print("Seuil : {:.2f}".format(thr[idx]))
```

```
Sensibilité : 0.98
Spécificité : 0.20
Seuil : 0.27
```

0.1 Un modèle kNN à des approches naïves

[17]:
```
data = pd.read_csv('winequality-red.csv', sep=";")

X = data[data.columns[:-1]].values
y = data['quality'].values
```

[18]:
```
from sklearn import model_selection

X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y,␣
↪test_size=0.3 ) # 30% des données dans le jeu de test
```

[19]:
```
std_scale = preprocessing.StandardScaler().fit(X_train)
X_train_std = std_scale.transform(X_train)
X_test_std = std_scale.transform(X_test)
```

Entraînons un kNN avec k=11 sur ces données :

[20]:
```
from sklearn import neighbors
knn = neighbors.KNeighborsRegressor(n_neighbors=11)

knn.fit(X_train_std, y_train)
```

[20]:
```
KNeighborsRegressor(n_neighbors=11)
```

Et appliquons le pour prédire les étiquettes de notre jeu de test :

[21]:
```
y_pred = knn.predict(X_test_std)
```

Calculons la RMSE correspondante :

[22]:
```
print("RMSE : {:.2f}".format(np.sqrt( metrics.mean_squared_error(y_test,␣
↪y_pred) )))
```

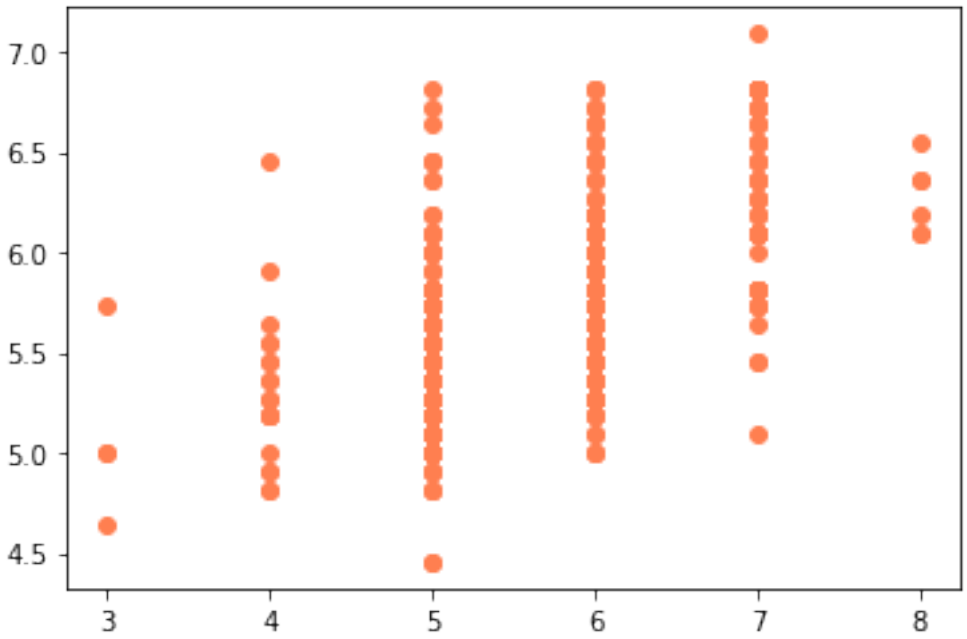
```
RMSE : 0.69
```

J'obtiens une **RMSE de 0.69** . Nos étiquettes étant des nombres entiers, nous faisons en moyenne une erreur inférieure à la plus petite différence possible entre deux notes.

Nous pouvons visualiser les résultats sur un graphique, en représentant en abscisse les vraies valeurs des étiquettes, et en ordonnée les valeurs prédites.

[23]:
```
plt.scatter(y_test, y_pred, color='coral')
```

[23]:
```
<matplotlib.collections.PathCollection at 0x21bdc35bf70>
```

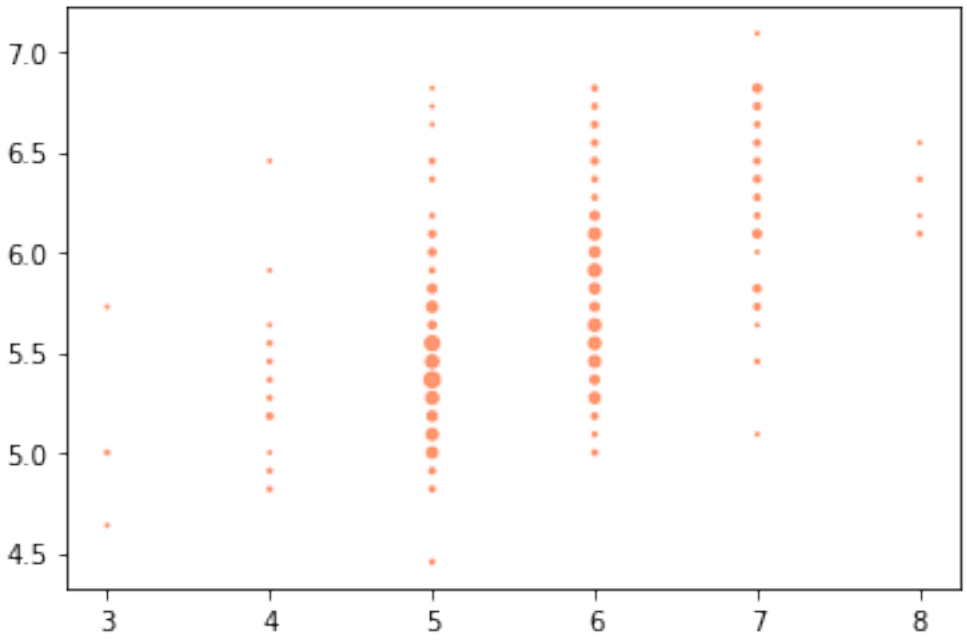


Comme nos étiquettes prennent des valeurs entières entre 3 et 8, nous avons beaucoup de points superposés aux même coordonnées. Pour mieux visualiser les données, nous pouvons utiliser comme marqueurs des cercles dont la taille est proportionnelle au nombre de points qui sont présents à ces coordonnées.

[24]:
```
sizes = {} # clé : coordonnées ; valeur : nombre de points à ces coordonnées
for (yt, yp) in zip(list(y_test), list(y_pred)):
    if (yt, yp) in sizes:
        sizes[(yt, yp)] += 1
    else:
        sizes[(yt, yp)] = 1

keys = sizes.keys()
plt.scatter(
        [k[0] for k in keys], # vraie valeur (abscisse)
        [k[1] for k in keys], # valeur predite (ordonnee)
```

```
        s=[sizes[k] for k in keys], # taille du marqueur
        color='coral', alpha =0.8)
```

[24]: <matplotlib.collections.PathCollection at 0x21bdbab0e50>



On note ainsi une accumulation de prédictions correctes sur la diagonale. Néanmoins le modèle n'est pas très précis dans ses prédictions.

Pour mieux comprendre notre modèle, comparons-le à une première approche naïve, qui consiste à prédire des valeurs aléatoires, distribuées uniformément entre les valeurs basse et haute des étiquettes du jeu de données d'entraînement.

Calculons la RMSE correspondante :

[25]:
```
y_pred_random = np.random.randint(np.min(y), np.max(y), y_test.shape)
print("RMSE : {:.2f}".format(np.sqrt(metrics.mean_squared_error(y_test,␣
 ↪y_pred_random))))
```

```
RMSE : 1.69
```

J'obtiens une RMSE de 1.69, ce qui est bien supérieur à la RMSE obtenue par notre modèle kNN. Notre modèle a ainsi réussi à bien mieux apprendre qu'un modèle aléatoire.

Cependant, beaucoup de nos vins ont une note de 6, et beaucoup de nos prédictions sont autour de cette valeur. Comparons maintenant notre modèle à un modèle aléatoire qui retourne systématiquement la valeur moyenne des étiquettes du jeu de données d'entraînement.

Nous pouvons utiliser pour cela la fonction correspondante du module **"dummy"** de scikit-learn.

[26]:
```
from sklearn import dummy
dum = dummy.DummyRegressor(strategy='mean')

# Entraînement
dum.fit(X_train_std, y_train)

# Prédiction sur le jeu de test
y_pred_dum = dum.predict(X_test_std)

# Evaluate
print("RMSE : {:.2f}".format(np.sqrt(metrics.mean_squared_error(y_test,␣
 ↪y_pred_dum)) ))
```

```
RMSE : 0.83
```