

Régression & optimisation par descente de gradient

```
In [34]: %load_ext autoreload
%autoreload 2

import tme8
```

The autoreload extension is already loaded. To reload it, use:
`%reload_ext autoreload`

Ce tme a deux objectifs:

- acquérir les connaissances de base pour faire face au problème de la régression, c'est à dire de l'estimation d'un score réel correspondant à une situation,
- travailler sur les techniques d'optimisation par descente de gradient

```
In [35]: import numpy as np
import matplotlib.pyplot as plt

np.random.seed(0) # pour avoir les même résultats aléatoires à chaque run
```

Génération de données jouet & construction d'une solution analytique

Dans un premier temps, générons des données jouets paramétriques:

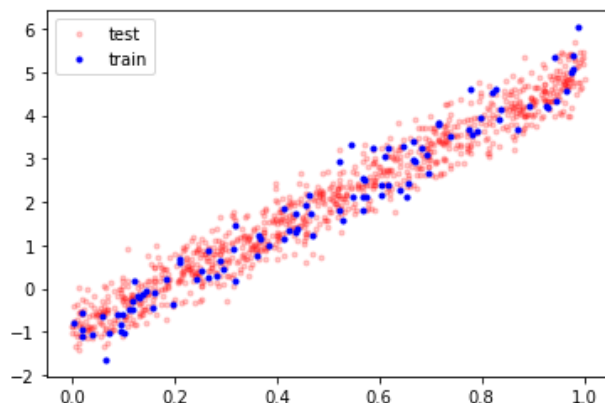
- N : nombre de points à générer
- $x \in [0, 1]$ tirage avec un simple `rand()` ou un `linspace()` -au choix-
 - Si vous optez pour un tirage aléatoire des abscisses, triez les points pour simplifier les traitements ultérieurs
- $y = ax + b + \epsilon, \epsilon \sim \mathcal{N}(0, \sigma^2)$
 - Rappel : en multipliant un tirage aléatoire selon une gaussienne centrée réduite par σ on obtient le bruit décrit ci-dessus

Afin de travailler sur les bonnes pratiques, nous distinguerons un ensemble d'apprentissage et un ensemble de test. Les deux sont tirés selon la même distribution. L'ensemble de test comptera -arbitrairement- 1000 points.

```
In [36]: # génération de données jouets:
a = 6.
b = -1.
N = 100
Ntest=1000
sig = .4 # écart type

X_train, y_train, X_test, y_test = tme8.gen_data_lin(a, b, sig, N, Ntest)

plt.plot(X_test, y_test, 'r.', alpha=0.2, label="test")
plt.plot(X_train, y_train, 'b.', label="train")
plt.legend();
#plt.savefig('gen_data.png')
```



Validation des formules analytiques

Nous avons vu deux types de résolutions analytique: à partir des estimateurs des espérances et co-variances d'une part et des moindres carrés d'autre part. Testons les deux méthodes.

Estimation de paramètres probabilistes

- $\hat{a} = \frac{\text{cov}(X,Y)}{\sigma_x^2}$
- $\hat{b} = E(Y) - \frac{\text{cov}(X,Y)}{\sigma_x^2}E(X)$

Estimer les paramètres \hat{a} et \hat{b} dans la fonction suivante `modele_lin_analytique`.

N.B. : dans les formules, il faut utiliser la covariance $\text{cov}(X, Y)$ empirique (non corrigée). Pour $\text{cov}(X, Y)$, utiliser `bias=True` dans `np.cov` (pour la variance σ_x^2 , `np.var` donne la variance non corrigée).

```
In [37]: ahat, bhat = tme8.modele_lin_analytique(X_train, y_train)
print(f"{ahat=:0.4f}, {bhat=:0.4f}")
```

```
ahat=6.3449, bhat=-1.1989
```

Calculer les valeurs prédites par ces paramètres pour un X puis calculer l'erreur au sens des moindres carrés sur les données d'apprentissage et de test

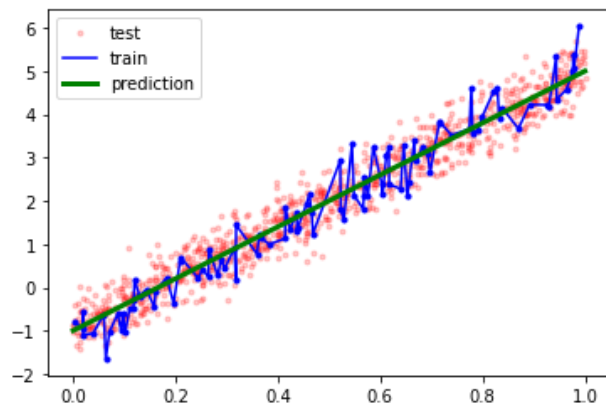
```
In [38]: yhat_train = tme8.calcul_prediction_lin(X_train, ahat, bhat)
yhat_test = tme8.calcul_prediction_lin(X_test, ahat, bhat)

print(f"{tme8.erreur_mc(y_train, yhat_train)=:0.4f}")
print(f"{tme8.erreur_mc(y_test, yhat_test)=:0.4f}")
```

```
tme8.erreur_mc(y_train, yhat_train)=0.1605
tme8.erreur_mc(y_test, yhat_test)=0.1625
```

Dessiner les nuages de points et dessiner la droite de régression.

```
In [39]: tme8.dessine_reg_lin(X_train, y_train, X_test, y_test, a, b)
```



Formulation au sens des moindres carrés

Nous partons directement sur une écriture matricielle. Du coup, il est nécessaire de construire la matrice enrichie Xe :

$$Xe = \begin{bmatrix} X_0 & 1 \\ \vdots & \vdots \\ X_N & 1 \end{bmatrix}$$

Il faut ensuite poser et résoudre un système d'équations linéaires de la forme:

$$Aw = B$$

Rappel des formules vues en cours/TD:

$$A = X^T X$$

$$B = X^T Y$$

Fonction de résolution: `np.linalg.solve(A,B)` Vous devez obtenir la même solution que précédemment.

Commencer par écrire la fonction pour générer les données enrichies X_e

```
In [40]: Xe = tme8.make_mat_lin_biais(X_train)
print(f"{Xe[:10]}")
```

```
[0.00469548 1.      ]
[0.0187898  1.      ]
[0.02010755 1.      ]
[0.0202184  1.      ]
[0.03918779 1.      ]
[0.06022547 1.      ]
[0.0641475  1.      ]
[0.07103606 1.      ]
[0.0871293  1.      ]
[0.09394051 1.      ]]
```

```
In [41]: w = tme8.reglin_matriciel(Xe,y_train)

print(f"{w=}")

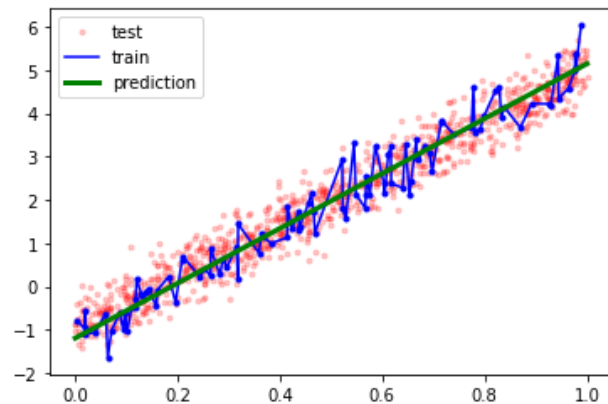
print(f"{tme8.erreur_mc(y_train, tme8.calcul_prediction_matriciel(Xe,w))=:.4f}")

print()
print(f"Pour rappel : {ahat=:0.4f}, {bhat=:0.4f}")
print(f"{tme8.erreur_mc(y_test, yhat_test)=:.4f}")

tme8.dessine_reg_lin(X_train, y_train, X_test, y_test,w[0],w[1])
```

```
w=array([ 6.34487547, -1.19890317])
tme8.erreur_mc(y_train, tme8.calcul_prediction_matriciel(Xe,w))=0.1605
```

```
Pour rappel : ahat=6.3449, bhat=-1.1989
tme8.erreur_mc(y_test, yhat_test)=0.1625
```



Données polynomiales

Soit les données polynomiales générées comme suit :

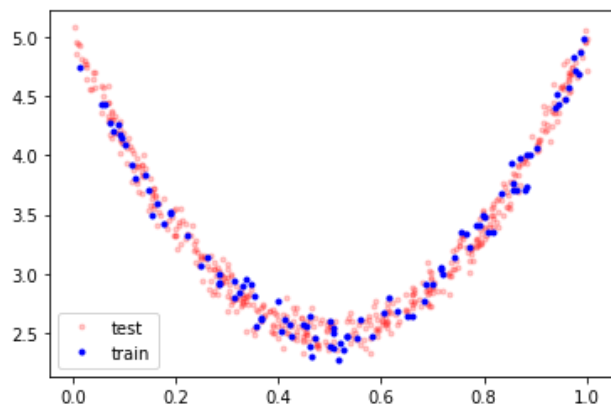
$$y = ax^2 + bx + c + eps$$

avec $eps \sim \mathcal{N}(0, sig^2)$

- proposer & une solution d'enrichissement (vue en cours et TD)
- résoudre analytiquement le problème des moindres carrés
- calculer l'erreur au sens des moindres carrés en apprentissage ET en test
- tracer les données et la solution

```
In [42]: Xp_train, yp_train, Xp_test, yp_test = tme8.gen_data_poly2(a=10, b=-10, c=5, sig=0.1, N=100, Ntest=500)

plt.plot(Xp_test, yp_test, 'r.',alpha=0.2,label="test")
plt.plot(Xp_train, yp_train, 'b.',label="train")
plt.legend();
```



```
In [43]: #enrichissement
Xe      = tme8.make_mat_poly_biais(Xp_train)
Xe_t    = tme8.make_mat_poly_biais(Xp_test)

print(Xe[:10])

[[2.15086305e-04 1.46658210e-02 1.00000000e+00]
 [2.97004896e-03 5.44981555e-02 1.00000000e+00]
 [3.82477721e-03 6.18447832e-02 1.00000000e+00]
 [5.19554909e-03 7.20801574e-02 1.00000000e+00]
 [6.43310868e-03 8.02066623e-02 1.00000000e+00]
 [8.03577352e-03 8.96424761e-02 1.00000000e+00]
 [8.51552414e-03 9.22795977e-02 1.00000000e+00]
 [9.37086326e-03 9.68032193e-02 1.00000000e+00]
 [1.06612276e-02 1.03253221e-01 1.00000000e+00]
 [1.32567245e-02 1.15137850e-01 1.00000000e+00]]
```

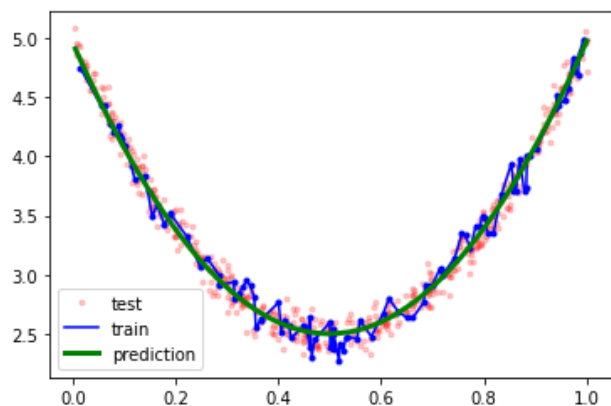
```
In [44]: w = tme8.reglin_matriciel(Xe,yp_train)
print(w)

[ 9.86980437 -9.8542074  4.95940741]
```

```
In [45]: yhat      = tme8.calcul_prediction_matriciel(Xe,w)
yhat_t    = tme8.calcul_prediction_matriciel(Xe_t,w)
print(f'Erreur moyenne au sens des moindres carrés (train): {tme8.erreur_mc(yhat, yp_train)=.4}')
print(f'Erreur moyenne au sens des moindres carrés (test): {tme8.erreur_mc(yhat_t, yp_test)=.4}')

tme8.dessine_poly_matriciel(Xp_train,yp_train,Xp_test,yp_test,w)
```

Erreur moyenne au sens des moindres carrés (train): tme8.erreur_mc(yhat, yp_train)=0.009482
 Erreur moyenne au sens des moindres carrés (test): tme8.erreur_mc(yhat_t, yp_test)=0.008867



Fonction de coût & optimisation par descente de gradient

Comme vu en TD et en cours, nous allons maintenant résoudre le problème de la régression par minimisation d'une fonction de coût:

$$C = \sum_{i=1}^N (y_i - f(x_i))^2$$

Soit un problème avec des données $(x_i, y_i)_{i=1, \dots, N}$, une fonction de décision/prédiction paramétrée par un vecteur w et une fonction de coût à optimiser $C(w)$. Notre but est de trouver les paramètres w^* minimisant la fonction de coût:

$$w^* = \arg \min_w C(w)$$

```
In [46]: # pour travailler en matrice: (re)construction de la matrice contenant les X et un biais
Xe = tme8.make_mat_lin_biais(X_train) # dataset linéaire, transformation lineaire des données

wstar=tme8.reglin_matriciel(Xe,y_train) # pour se rappeler du w optimal
print(f"{wstar=}")
```

```
wstar=array([ 6.34487547, -1.19890317])
```

l'algorithme de la descente de gradient est le suivant (rappel):

- $w_0 \leftarrow \text{init}$ par exemple : 0
- boucle

$$- w_{t+1} \leftarrow w_t - \epsilon \nabla_w C(w_t)$$

Proposer une implémentation dans `tme8.descente_grad_mc(X, y, eps=1e-4, nIterations=100)` qui rend `w,wall` : le `w` obtenu et l'historique des itérations.

```
In [47]: w, allw = tme8.descente_grad_mc(Xe, y_train, eps=1e-4, nIterations=500)
print(f"{w=}")
print(allw[-10:])
```

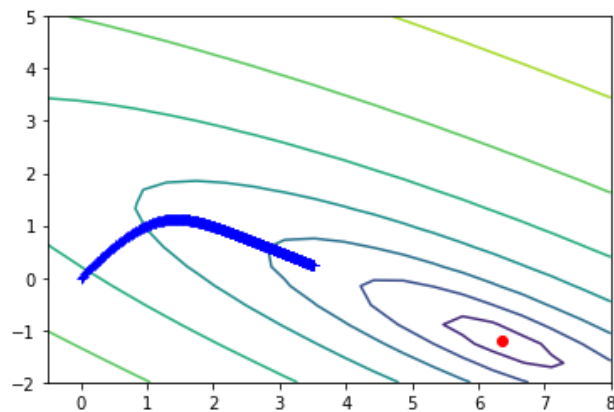
```
w=array([3.5169631 , 0.23420253])
[[3.48260125 0.25161441]
 [3.48643977 0.24966937]
 [3.49027314 0.24772694]
 [3.49410136 0.2457871 ]
 [3.49792445 0.24384987]
 [3.50174241 0.24191522]
 [3.50555525 0.23998317]
 [3.50936297 0.23805371]
 [3.51316559 0.23612683]
 [3.5169631  0.23420253]]
```

On s'intéresse ensuite à comprendre la descente de gradient dans l'espace des paramètres. Le code ci-dessous permet de tracer le coût pour un ensemble de paramètres (toutes les valeurs de paramètres prises par l'algorithmes au fil du temps).

```
In [48]: # tracer de l'espace des couts
def plot_parametres( allw, X, y, opti = [], ngrid = 20, extract_bornes=False):
    """
    Fonction de tracer d'un historique de coefficients
    ATTENTION: ca ne marche qu'en 2D (évidemment)
    Chaque w doit contenir 2 valeurs

    Il faut fournir les données (X,y) pour calculer le cout associé
    à un jeu de paramètres w
    ATTENTION X = forme matricielle des données
    """
    w_min = [-0.5, -2] # bornes par défaut, uniquement pour notre cas d'usage
    w_max = [8, 5]
    if extract_bornes: # bornes générales
        w_min = np.min(allw,0) # trouver les bornes
        w_max = np.max(allw,0)
    # faire une grille régulière avec tous les couples possibles entre le min et le max
    w1range = np.linspace(w_min[0], w_max[0], ngrid)
    w2range = np.linspace(w_min[1], w_max[1], ngrid)
    w1,w2 = np.meshgrid(w1range,w2range)
    #
    # calcul de tous les couts associés à tous les couples de paramètres
    cost = np.array([[np.log((X @ np.array([w1i,w2j])-y)**2).sum()) for w1i in w1range] for w2j in w2range])
    #
    plt.figure()
    plt.contour(w1, w2, cost)
    if len(opti) > 0:
        plt.scatter(wstar[0], wstar[1],c='r')
    plt.plot(allw[:,0],allw[:,1], 'b+-' ,lw=2 )
    return
```

```
plot_parametres( allw, Xe, y_train, opti=wstar)
# plt.savefig('fig/grad_descente.png')
```



Tester différents jeux de paramètres pour mettre en évidence les phénomènes suivants:

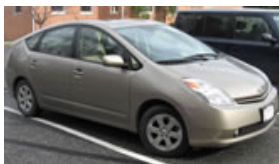
- Divergence du gradient
- Convergence incomplète (trop lente ou pas assez d'itération)
- Convergence idéale: pas de gradient suffisamment grand et nombre d'itérations bien choisi

Passage sur des données réelles

Après avoir étudié trois manières de faire face au problème de la régression, nous proposons d'étudier un cas réel: la prédiction de la consommation des voitures en fonction de leurs caractéristiques.

Dans le cas présent, nous allons baser la solution sur la résolution analytique du problème des moindres carrés (`np.linalg.solve(A,B)`), qui semble la mieux adaptée au problème qui nous intéresse.

Le jeu de données est issu des datasets UCI, un répertoire parmi les plus connus en machine learning. Les données **sont déjà téléchargées et présentes dans le tme** mais vous voulez plus d'informations: <https://archive.ics.uci.edu/ml/datasets/auto+mpg>



Après avoir importé les données (fonction fournie), vous construirez une solution optimale et l'évaluerez au sens des moindres carrés en apprentissage et en test.

```
In [49]: import pandas as pd
# Chargement des données
data = pd.read_csv('res/auto-mpg.data', delimiter='\s+', header=None) # comme np.loadtxt mais en plus robuste
# remplacement des données manquantes '?' => Nan pour travailler sur des nombres
data.iloc[:,[3]] = data.iloc[:,[3]].replace('?', None)
data.iloc[:,[3]] = data.iloc[:,[3]].astype(float)
# remplacement des valeurs manquantes par la moyenne
data.iloc[:,[3]] = data.iloc[:,[3]].fillna(data.iloc[:,[3]].mean())

print(data.head()) # visualiser ce qu'il y a dans les données

X = np.array(data.values[:,1:-2], dtype=np.float64)
y = np.array(data.values[:,0], dtype=np.float64)

      0  1    2    3    4    5    6  7      8
0  18.0  8  307.0  130.0  3504.0  12.0  70  1  chevrolet chevelle malibu
1  15.0  8  350.0  165.0  3693.0  11.5  70  1      buick skylark 320
2  18.0  8  318.0  150.0  3436.0  11.0  70  1    plymouth satellite
3  16.0  8  304.0  150.0  3433.0  12.0  70  1          amc rebel sst
4  17.0  8  302.0  140.0  3449.0  10.5  70  1          ford torino

/var/folders/l1/9w2s76mx37v40x1429jpfm3m0000gn/T/ipykernel_35319/2944173861.py:6: FutureWarning: In a future
version, `df.iloc[:, i] = newvals` will attempt to set the values inplace instead of always setting a new ar
ray. To retain the old behavior, use either `df[df.columns[i]] = newvals` or, if columns are non-unique, `d
f.iat[i, i] = newvals`
      data.iloc[:,[3]] = data.iloc[:,[3]].astype(float)
```

```
In [50]: # separation app/test
def separation_train_test(X, y, pc_train=0.75):
    index = np.arange(len(y))
    np.random.shuffle(index) # liste mélangée
    napp = int(len(y)*pc_train)
    X_train, y_train = X[index[:napp]], y[index[:napp]]
    X_test, y_test = X[index[napp:]], y[index[napp:]]
    return X_train, y_train, X_test, y_test

X_train, y_train, X_test, y_test = separation_train_test(X, y, pc_train=0.75)
```

```
In [51]: # utiliser les fonctions déjà définies au maximum =
w, yhat, yhat_t = tme8.application_reelle(X_train, y_train, X_test, y_test)
```

```
w=array([-0.69525116,  0.02030259, -0.00908044, -0.00754606,  0.0189724 ,
         0.61081869])
Erreur moyenne au sens des moindres carrés (train): 10.89
Erreur moyenne au sens des moindres carrés (test): 15.62
```

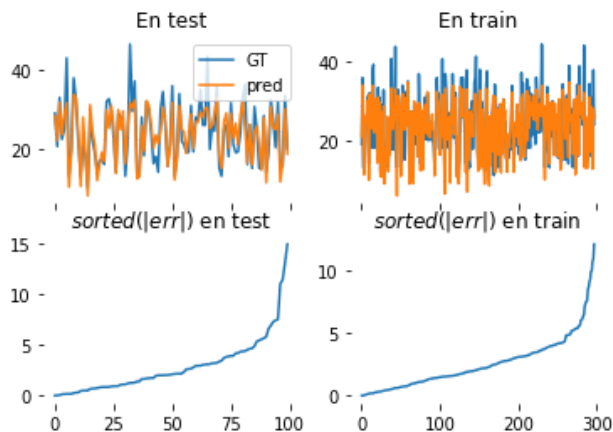
```
In [52]: def plot_y(y_train, y_test, yhat, yhat_t):
    # tracé des prédictions:
    fig, (ax1, ax2) = plt.subplots(ncols=2, nrows=2, sharex='col', subplot_kw=dict(frameon=False))

    ax1[0].set_title('En test')
    ax1[0].plot(y_test, label="GT")
    ax1[0].plot(yhat_t, label="pred")
    ax1[0].legend()

    ax1[1].set_title('En train')
    ax1[1].plot(y_train, label="GT")
    ax1[1].plot(yhat, label="pred")

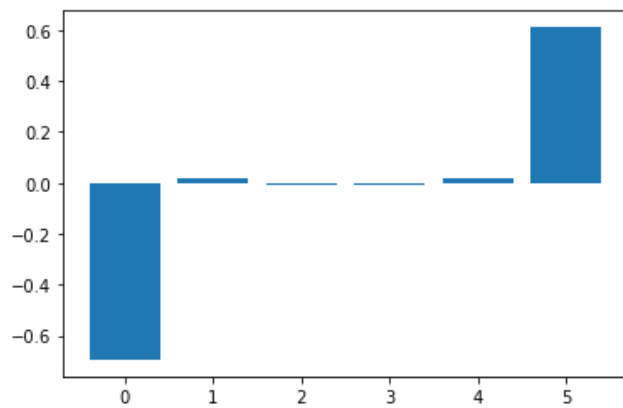
    ax2[0].set_title('$sorted(|err|)$ en test')
    ax2[0].plot(sorted(abs(y_test-yhat_t)), label="diff")

    ax2[1].set_title('$sorted(|err|)$ en train')
    ax2[1].plot(sorted(abs(y_train-yhat)), label="diff")
    return
plot_y(y_train, y_test, yhat, yhat_t)
```



```
In [53]: # interprétation des poids
plt.figure()
plt.bar(np.arange(len(w)), w)
```

Out[53]: <BarContainer object of 6 artists>



Normalisation

Sur le diagramme ci-dessus, on ne voit pas grand chose pour une raison évidente: on ne peut pas comparer ces poids qui comparent des variables dont les ordres de grandeur sont différents.

Nous allons donc assimiler chaque colonne X_j à une variable suivant une loi normale et nous allons revenir à une Normale centrée réduite selon la formule de base:

$$X_j \sim \mathcal{N}(\mu_j, \sigma_j^2)$$

$$\Rightarrow Z_j = \frac{X_j - \mu_j}{\sigma_j} \sim \mathcal{N}(0, 1)$$

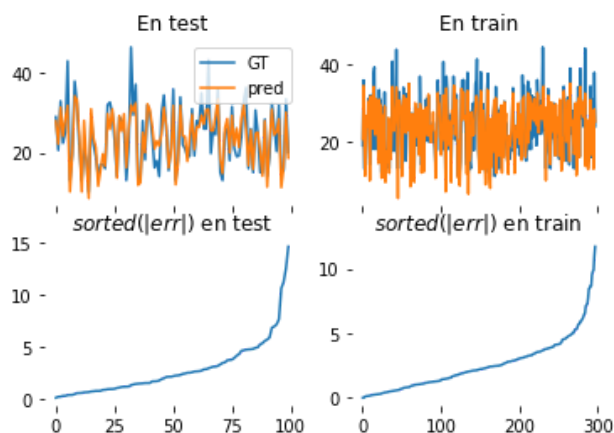
Tous les Z_j sont comparables et nous serons en mesure de comprendre l'impact de chaque variables sur les résultats.

ATTENTION: on ne se basera que sur les données d'apprentissage pour le calcul des $\{\mu_j, \sigma_j\}$.

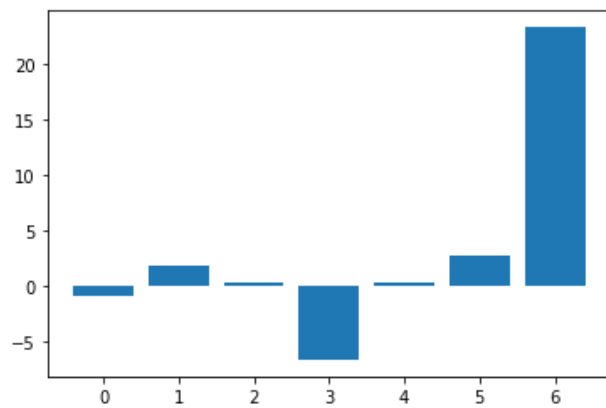
Une fois la normalisation effectuée, analyser l'impact des différentes variables descriptives sur la valeur à prédire.

```
In [54]: Xn_train, Xn_test = tme8.normalisation(X_train, X_test)
w, ynhathat, ynhathat_t = tme8.application_reelle(Xn_train, y_train, Xn_test, y_test)
plot_y(y_train, y_test, ynhathat, ynhathat_t)
```

w=array([-0.91167546, 1.83686731, 0.29370969, -6.71094367, 0.31889282,
 2.69635739, 23.37013423])
Erreur moyenne au sens des moindres carrés (train): 10.7
Erreur moyenne au sens des moindres carrés (test): 15.01



```
In [55]: plt.figure()
plt.bar(np.arange(len(w)), w);
```

Questions d'ouverture

Sélection de caractéristiques

Quels sont les résultats obtenus en éliminant toutes les variables servant moins?

Feature engineering

En étudiant la signification des variables du problèmes, on trouve:

1. mpg: continuous
2. cylinders: multi-valued discrete
3. displacement: continuous
4. horsepower: continuous
5. weight: continuous
6. acceleration: continuous
7. model year: multi-valued discrete
8. origin: multi-valued discrete

D'après la question précédente, le poids, l'année du modèle et le biais sont des facteurs important pour le calcul de la consommation... Jusqu'ici, nous n'avons pas pris en compte l'origine qui était difficile à coder.

Encodage de l'origine

La variable origine est accessible de la manière suivante:

```
origine = data.values[:,-2]
```

Il faut le faire au début du traitement pour bien conserver la séparation en l'apprentissage et le test.

Au moins les deux derniers facteurs discrets pourraient être traités différemment en one-hot encoding:

$$X_j = x \in \{1, \dots, K\} \Rightarrow [0, 0, 1, 0] \in \{0, 1\}^K$$

La valeur x donne l'index de la colonne non nulle.

Encodage de l'année

Pour l'année, il est possible de procéder de la même manière, mais il est préférable de découper les années en 10 catégories puis d'encoder pour limiter le nombre de dimensions.

In []:

Question d'ouverture sur le gradient

La normalisation a-t-elle un impact sur le gradient?

La normalisation des données peut au moins nous aider à régler plus facilement le pas (qui sera toujours du même ordre de grandeur... Mais cela a-t-il un impact sur la manière dont nous nous rapprochons de la solution optimale?

Gradient stochastique

Dans la plupart des algorithmes modernes d'optimisation liés aux réseaux de neurones, le gradient est calculé de manière stochastique, sur un exemple à la fois:

- $w_0 \leftarrow \textit{init}$ par exemple : 0
- boucle
 - tirage d'une donnée i : (x_i, y_i)
 - $w_{t+1} \leftarrow w_t - \epsilon \nabla_w C_i(w)$

Etudier le fonctionnement de cet algorithme sur les exemples jouets précédents.

Amélioration du gradient

Le blog de S. Ruder explique particulièrement bien les améliorations possibles sur les descentes de gradient.

<https://ruder.io/optimizing-gradient-descent/>

Comparer une descente de gradient stochastique avec et sans moment sur les données jouets des premières questions.