Régression & optimisation par descente de gradient

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 correpondant à une situation,
- travailler sur les techniques d'optimisation par descente de gradient

In [4]:

```
import numpy as np
import matplotlib.pyplot as plt
```

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 [5]:

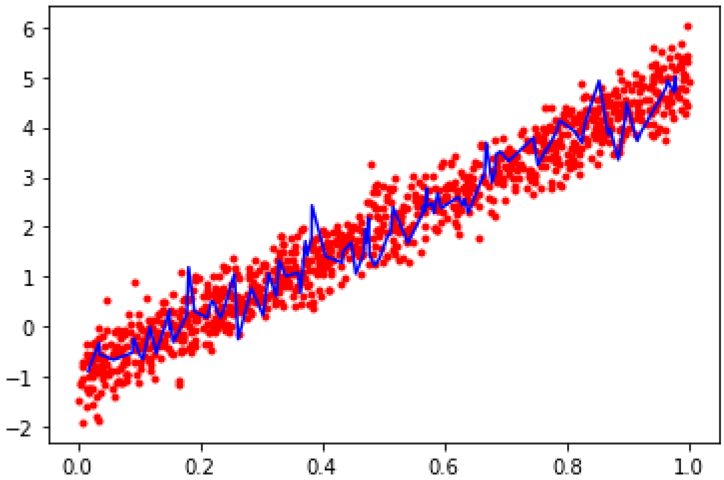
```
#données jouets
def gen_data_lin(a, b, sig, N=500, Ntest=1000):
    #Create an array of the given shape and populate it with random samples from a uniform distribution
    X_train = np.sort(np.random.rand(N)) # sort optionnel, mais ça aide pour les plots ( pour simplifier traitement ultérieur)
    X_test  = np.sort(np.random.rand(Ntest))
    #standard deviation=ecart type =sigma
    Y_train = a*X_train+b+np.random.normal(0.0,sig, N )
    Y_test  = a*X_test+b+np.random.normal(0.0,sig, Ntest )
    return X_train, Y_train, X_test, Y_test
```

In [6]:

```
# génération de données jouets:
a = 6.
b = -1.
N = 100
sig = .4 # écart type

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

plt.figure()
plt.plot(X_test, y_test, 'r.')
plt.plot(X_train, y_train, 'b')
```

Out[6]:

```
[<matplotlib.lines.Line2D at 0x7f5b145b65f8>]
```



Vous devez obtenir quelque chose de la forme:
données jouet

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

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

  Estimer les paramètres, calculer l'erreur au sens des moindres carrés sur les données d'apprentissage et de test, puis tracer la droite de régression

In [7]:

```
def modele_lin_analytique(X_train, y_train):
    #utiliser des estimateurs des esperances et des covariances,
    #j'ai estimé l'esperance avec mean, est-ce que c'est juste?
    ahat=np.cov(X_train, y_train)[0][1]/np.var(X_train)
    bhat=np.mean(y_train)- ahat* np.mean(X_train)
    return ahat, bhat

ahat, bhat = modele_lin_analytique(X_train, y_train)
print(ahat, bhat)
```

```
5.832502265780935 -0.9483719998304234
```

In [8]:

```
def erreur_mc(y, yhat):
    return ((y-yhat)**2).mean()


yhat_train = ahat*X_train+bhat
yhat_test  = ahat*X_test+bhat

print('Erreur moyenne au sens des moindres carrés (train):', erreur_mc(yhat_trai
n, y_train))
print('Erreur moyenne au sens des moindres carrés (test):', erreur_mc(yhat_test,
y_test))
```

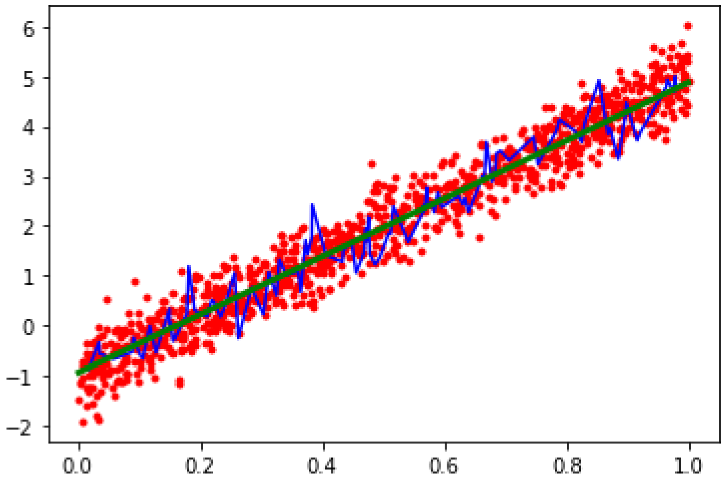
```
Erreur moyenne au sens des moindres carrés (train): 0.12892499331355
964
Erreur moyenne au sens des moindres carrés (test): 0.164065634002243
58
```

In [9]:

```
plt.figure()
plt.plot(X_test, y_test, 'r.')
plt.plot(X_train, y_train, 'b')
plt.plot(X_test, yhat_test, 'g', lw=3)
```

Out[9]:

```
[<matplotlib.lines.Line2D at 0x7f5b11251780>]
```



Formulation au sens des moindres carrés

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

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

Le code de la fonction d'enrichissement est donné ci-dessous.

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.

In [10]:

```
def make_mat_lin_biais(X): # fonctionne pour un vecteur unidimensionel X
    N = len(X)
    N2=1
    if (X.shape[0]<X.size):
        #print("si, yo se que yo hace")
        N2=X.shape[1]
    return np.hstack((X.reshape(N,N2),np.ones((N,1))))
```

In [11]:

```
Xe = make_mat_lin_biais(X_train)
# construction de A
A=np.transpose(Xe)@Xe
B=np.transpose(Xe)@y_train
w=np.linalg.solve(A,B)
print(w)
```

```
[ 5.77417724 -0.91939843]
```

solution précédante, est-ce que c'est normal que ce ne soit pas exactement la même chose

5.832502265780935 -0.9483719998304234

Soit les données polynomiales générées avec la fonction ci-dessous

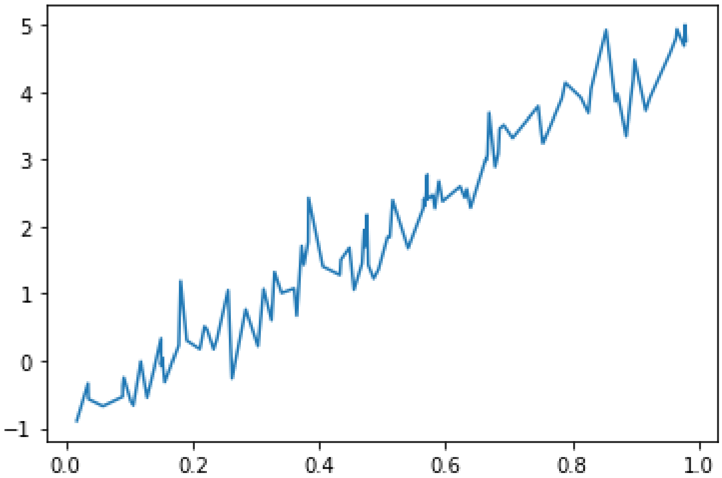
- proposer & une solution d'enrichissement (vue en cours et TD)
- résoudre analytiquement le problème des moindres carrés
- calculer l'erreur au sens des moindes carrés en apprentissage ET en test
- tracer les données et la solution

In [12]:

```
def gen_data_poly2(a, b, c, sig, N=500, Ntest=1000):
    '''
    Tire N points X aléatoirement entre 0 et 1 et génère y = ax^2 + bx + c + eps
    eps ~ N(0, sig^2)
    '''
    X_train = np.sort(np.random.rand(N))
    X_test  = np.sort(np.random.rand(Ntest))
    y_train = a*X_train**2+b*X_train+c+np.random.randn(N)*sig
    y_test  = a*X_test**2 +b*X_test +c+np.random.randn(Ntest)*sig
    return X_train, y_train, X_test, y_test

Xp_train, yp_train, Xp_test, yp_test = gen_data_poly2(10, -10, 5, 0.1, N=100, Ntest=100)
plt.figure()
plt.plot(X_train, y_train)
```

Out[12]:

```
[<matplotlib.lines.Line2D at 0x7f5b111c39e8>]
```



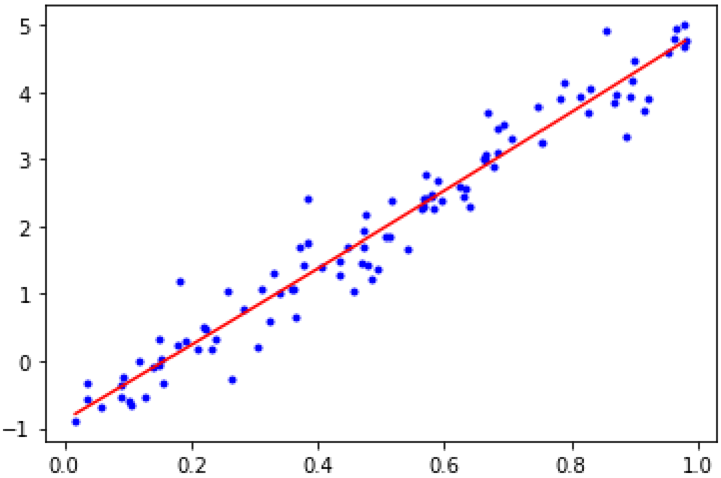
In [13]:

```
def make_mat_poly_biais(X): # fonctionne pour un vecteur unidimensionel X
    # enrichissement en ajoutant une colonne de X²
    X2=X*X
    N=len(X)
    N2=1
    #print(X.size)
    if (X.shape[0]<X.size):
        #print("estoy aqui!")
        N2=X.shape[1]
    #print(X.shape[0])
    #print(N2)
    return np.hstack((X2.reshape(N,N2),make_mat_lin_biais(X)))

Xe   = make_mat_poly_biais(X_train)
Xe_t = make_mat_poly_biais(X_test)
#résolution analytique

##
A=np.transpose(Xe)@Xe
B=np.transpose(Xe)@y_train
w=np.linalg.solve(A,B)

##
a=w[0]
b=w[1]
c=w[2]

yhat   = a*X_train**2+b*X_train+c
yhat_t = a*X_test**2+b*X_test+c
print('Erreur moyenne au sens des moindres carrés (train):', erreur_mc(yhat, y_t
rain))
print('Erreur moyenne au sens des moindres carrés (test):', erreur_mc(yhat_t, y_
test))

plt.figure()
plt.plot(X_train, y_train, 'b.')
plt.plot(X_train, yhat, 'r')
```

```
Erreur moyenne au sens des moindres carrés (train): 0.12848476185046
037
Erreur moyenne au sens des moindres carrés (test): 0.165802397337141
36
```

Out[13]:

```
[<matplotlib.lines.Line2D at 0x7f5b11184978>]
```



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:

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

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

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

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

- $w_0 \leftarrow init$ par exemple : 0
- boucle
  - $w_{t+1} \leftarrow w_t - \epsilon \nabla_w C(w_t)$

Compléter le squelette d'implémentation fourni ci-dessous:

il n'y a pas un carré ici pour c??

c'est vraiment à 0 qu'il faut initialiser w????

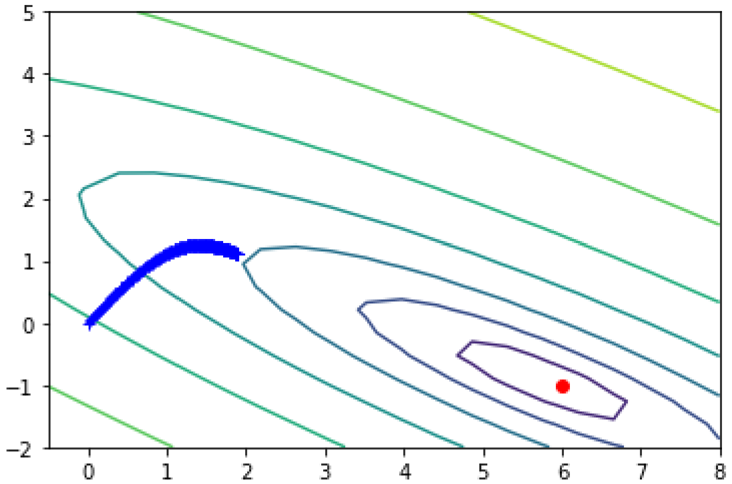
In [14]:

```
# pour travailler en matrice: (re)construction de la matrice contenant les X et un biais
Xe = make_mat_lin_biais(X_train) # dataset linéaire, transformation lineaire des données
# l'initial vaut 0, j'aurais préféré une autre valeur !!

#_____________________________________________________________________________
a=6
b=-1

wstar = np.array([a,b])# A compléter # pour se rappeler du w optimal
#je n'ai pas compris par quoi ils veulent qu'on remplisse ici ??? valeur déjà trouvé avant ??
#_____________________________________________________________________________


def descente_grad_mc(X, y, eps=1e-4, nIterations=100):
    w = np.zeros(X.shape[1]) # init à 0
    allw = [w]
    for i in range(nIterations):
        # A COMPLETER => calcul du gradient vu en TD
        # gradient avec la fonction donnée dans l'énoncé
        #grad_a=-np.sum(X[:,0])# sans 1
        #grad_b=-len(X)

        # dans le cas ou la fonction de cout est au carré
        grad_b=np.sum(-2*(y-w[1]-w[0]*X[:,0]))

        grad_a=np.sum(-2*X[:,0]*(y-w[1]-w[0]*X[:,0]))

        w=w-eps*np.array([grad_a, grad_b])
        allw.append(w) # stockage de toutes les valeurs intermédiaires pour analyse
    allw = np.array(allw)
    return w, allw # la dernière valeur (meilleure) + tout l'historique pour le plot

w, allw = descente_grad_mc(Xe, y_train, eps=1e-4, nIterations=200)
```

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

In [15]:

```
# 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')
```



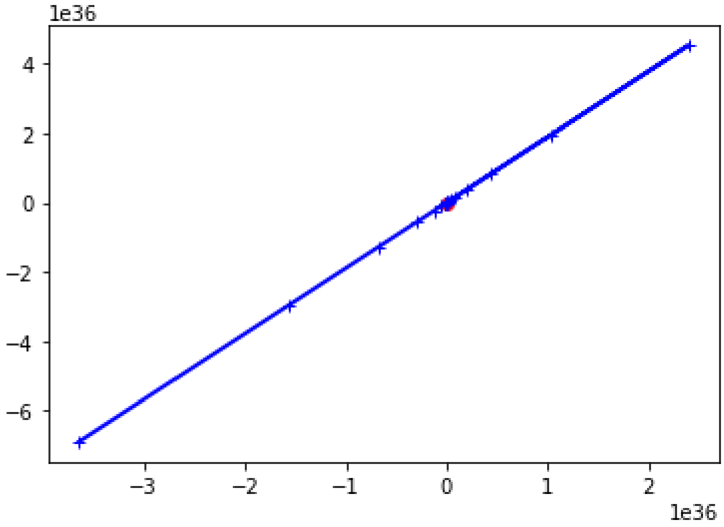
Vous devez obtenir un image de la forme :
Descente de gradient

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

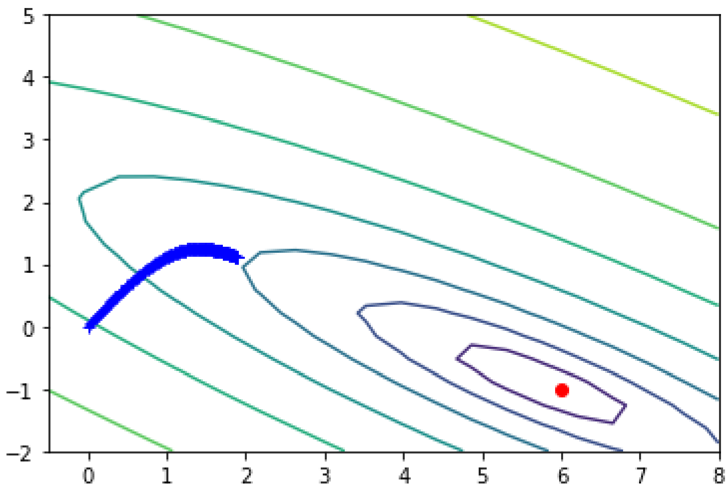
In [16]:

```
# divergence du gradient dans le cas ou les pas sont trop grands, donc eps grand
w, allw = descente_grad_mc(Xe, y_train, eps=1e-2, nIterations=200)
plot_parametres( allw, Xe, y_train, opti=wstar)
```



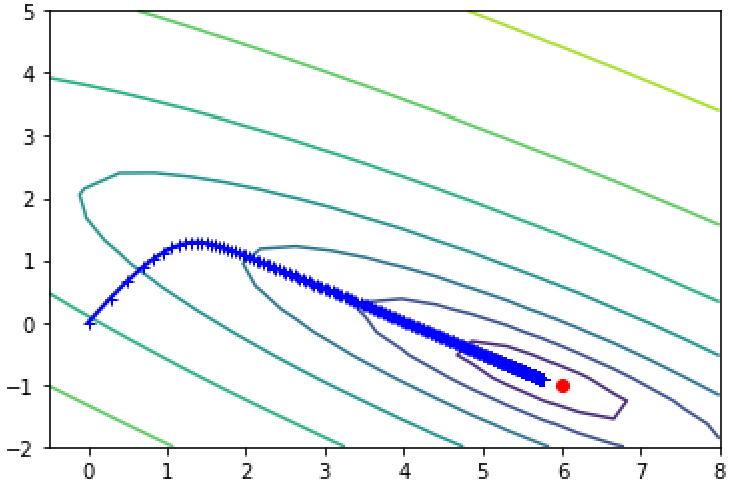
In [17]:

```
#convergence incomplète, nombre d'itérations insuffisant car le pas est trop pet
it
w, allw = descente_grad_mc(Xe, y_train, eps=1e-4, nIterations=200)
plot_parametres( allw, Xe, y_train, opti=wstar)
```



In [22]:

```
# convergence idéale
w, allw = descente_grad_mc(Xe, y_train, eps=1e-3, nIterations=500)
plot_parametres( allw, Xe, y_train, opti=wstar)
```



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 (https://archive.ics.uci.edu/ml/datasets/auto+mpg)](https://archive.ics.uci.edu/ml/datasets/auto+mpg)

voiture

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.

solution analytique des moindres carrés

In [23]:

```
import pandas as pd
# Chargement des données
data = pd.read_csv('data/auto-mpg.data', delimiter='\s+', header=None) # comme n
p.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 ma
libu
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 satel
lite
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 to
rino
```

In [24]:

```
print(X[0:4,:])
```

```
[[   8.   307.   130.  3504.    12.    70. ]
 [   8.   350.   165.  3693.    11.5   70. ]
 [   8.   318.   150.  3436.    11.    70. ]
 [   8.   304.   150.  3433.    12.    70. ]]
```

In [25]:

```
# separartion app/test
import random as rd
def separation_train_test(X, y, pc_train=0.75):
    # A compléter
    # 1) générer tous les index entre 0 et N-1
    N=len(X)
    index=[i for i in range(N)]
    # 2) mélanger la liste
    #sort randomly
    rd.shuffle(index)
    napp = int(len(y)*pc_train) # nb de points pour le 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)
```

on a 8 paramètres, si on considère une solution analytique avec une regression lineaire

y=a1 x1+a2 x2+ a3 x3+ a4 x4+ a5 x5 + a6 x6 +a7 x7+ a8 x8

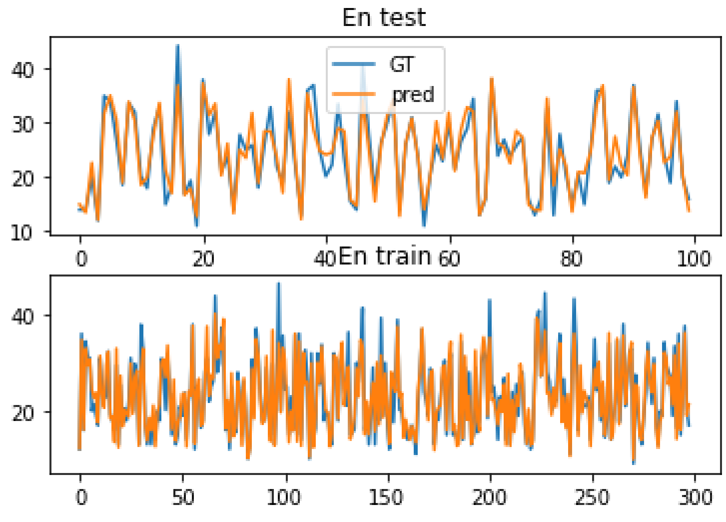
In [26]:

```
# Résolution analytique

Xe=make_mat_poly_biais(X_train)
Xe_t=make_mat_poly_biais(X_test)
A=np.transpose(Xe)@Xe
B=np.transpose(Xe)@y_train
w = np.linalg.solve(A,B)
yhat   = Xe@w
yhat_t = Xe_t@w
print('Erreur moyenne au sens des moindres carrés (train):', erreur_mc(yhat, y_train))
print('Erreur moyenne au sens des moindres carrés (test):', erreur_mc(yhat_t, y_test))
```

```
Erreur moyenne au sens des moindres carrés (train): 7.61593800751163
9
Erreur moyenne au sens des moindres carrés (test): 7.329963299077433
```

In [27]:

```
def plot_y(y_train, y_test, yhat, yhat_t):
    # tracé des prédictions:
    plt.figure()
    plt.subplot(211)
    plt.plot(y_test, label="GT")
    plt.plot(yhat_t, label="pred")
    plt.title('En test')
    plt.legend()
    plt.subplot(212)
    plt.plot(y_train, label="GT")
    plt.plot(yhat, label="pred")
    plt.title('En train')
    return
plot_y(y_train, y_test, yhat, yhat_t)
```



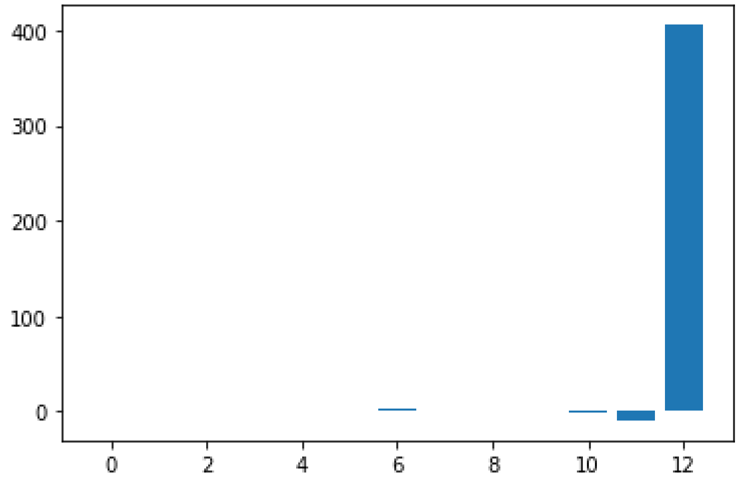
le modèle semble bien correspondre pour le training et le test set. le choix du modèle linéaire semble être bon

In [28]:

```
# interprétation des poids
plt.figure()
plt.bar(np.arange(len(w)), w)
```

Out[28]:

```
<BarContainer object of 13 artists>
```



à quoi sert ce schéma

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 seront 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 $\{\mu_j, \sigma_j\}$.

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

In [29]:

```
A=np.array([[2,1],[2,1]])
np.mean(A, axis=0)
```

Out[29]:

```
array([2., 1.])
```

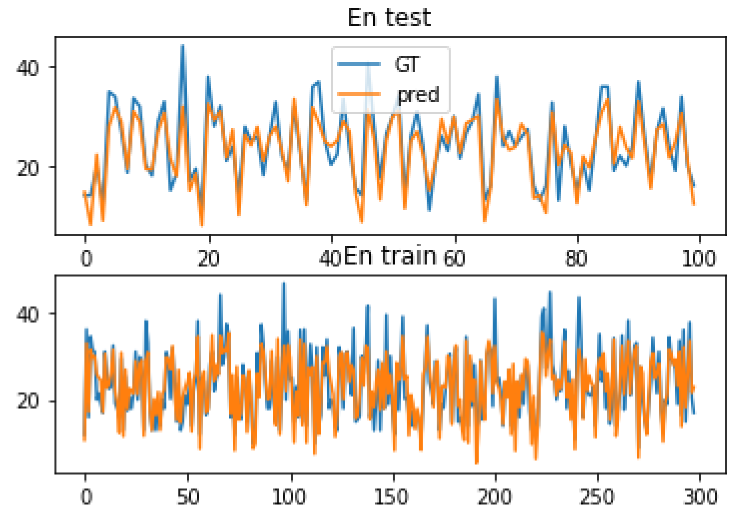
In [31]:

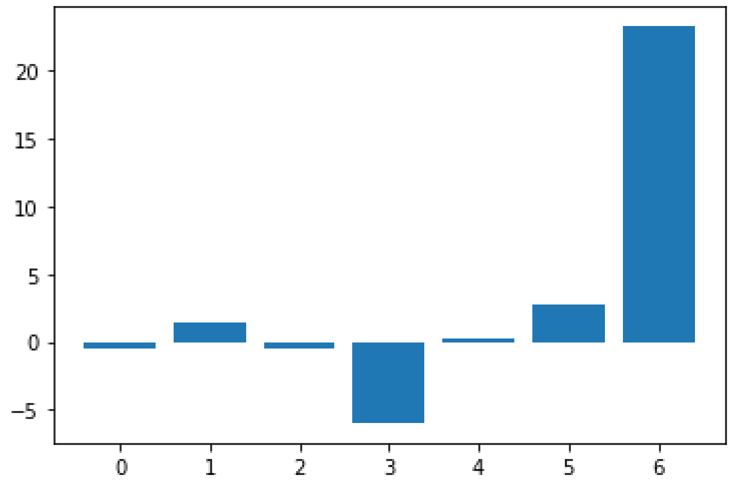
```
def normalisation(X_train, X_test):
    '''
    Fonction de normalisation des données pour rendre les colonnes comparables
    Chaque variable sera assimilée à une loi normale qu'il faut centrer + réduir
e.
    ATTENTION: il faut calculer les moyennes et écarts-types sur les données d'a
pprentissage seulement
    '''
    # A compléter
    # 1) calcul des moyennes et écarts types pour chaque colonne
    #X_train
    means=np.mean(X_train, axis=0)
    stds=np.std(X_train, axis=0)
    #X_test
    means_t=np.mean(X_test, axis=0)
    stds_t=np.std(X_test, axis=0)

    #X_train
    # 2) normalisation des colonnes
    np.reshape(means, (1, X_train.shape[1]))
    np.reshape(stds, (1, X_train.shape[1]))
    Xn_train=(X_train-means)/stds
    #X_test
    np.reshape(means_t, (1, X_test.shape[1]))
    np.reshape(stds_t, (1, X_test.shape[1]))
    Xn_test=(X_test-means_t)/stds_t
    # 3) Ajout d'un biais: fourni ci-dessous)
    Xn_train = np.hstack((Xn_train, np.ones((Xn_train.shape[0], 1))))
    Xn_test   = np.hstack((Xn_test, np.ones((X_test.shape[0], 1))))
    return Xn_train, Xn_test

Xn_train, Xn_test = normalisation(X_train, X_test)
A=np.transpose(Xn_train)@Xn_train
B=np.transpose(Xn_train)@y_train

w = np.linalg.solve(A,B)

yhat   = Xn_train@w
yhat_t = Xn_test@w
print('Erreur moyenne au sens des moindres carrés (train):', erreur_mc(yhat, y_t
rain))
print('Erreur moyenne au sens des moindres carrés (test):', erreur_mc(yhat_t, y_
test))
plot_y(y_train, y_test, yhat, yhat_t)

plt.figure()
plt.bar(np.arange(len(w)), w)
```

```
Erreur moyenne au sens des moindres carrés (train): 11.7795741742674
93
Erreur moyenne au sens des moindres carrés (test): 12.31761856404299
3
```

Out[31]:

<BarContainer object of 7 artists>





In [32]:

```
X_train.shape
```

Out[32]:

```
(298, 6)
```

Questions d'ouverture

Sélection de caractéristiques

Quels sont les résultats obtenus en éliminant toutes les variables servent 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 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:

- $w_0 \leftarrow init$ 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/ (https://ruder.io/optimizing-gradient-descent/)](https://ruder.io/optimizing-gradient-descent/)

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

In [ ]:

```
```