Apprendimento supervisionato

Nel Supervised Learing (SL) vengono passati all'algoritmo gli input - detti anche “features”, e gli output corrispondenti agli input, detti anche “targets” o "labels".
L'algoritmo produce quindi una funzione (f) (f minuscola) , in input alla funzione vengono passate le features detta anche x (x minuscola) e ritorna le y^ (y cappello) che rappresentano i valori predetti dalla funzione.
NB La funzione f è detta anche “modello”.
La differenza tra y e y^ è che la y è il training set di features (quindi i valori noti da passare durante la fase di training) metre i valori y^ sono i valori stimati che il modello prevede in base agli input. (post fase di traing)

- Regressione lineare
- Regressione lineare multipla
- Regressione polinomiale

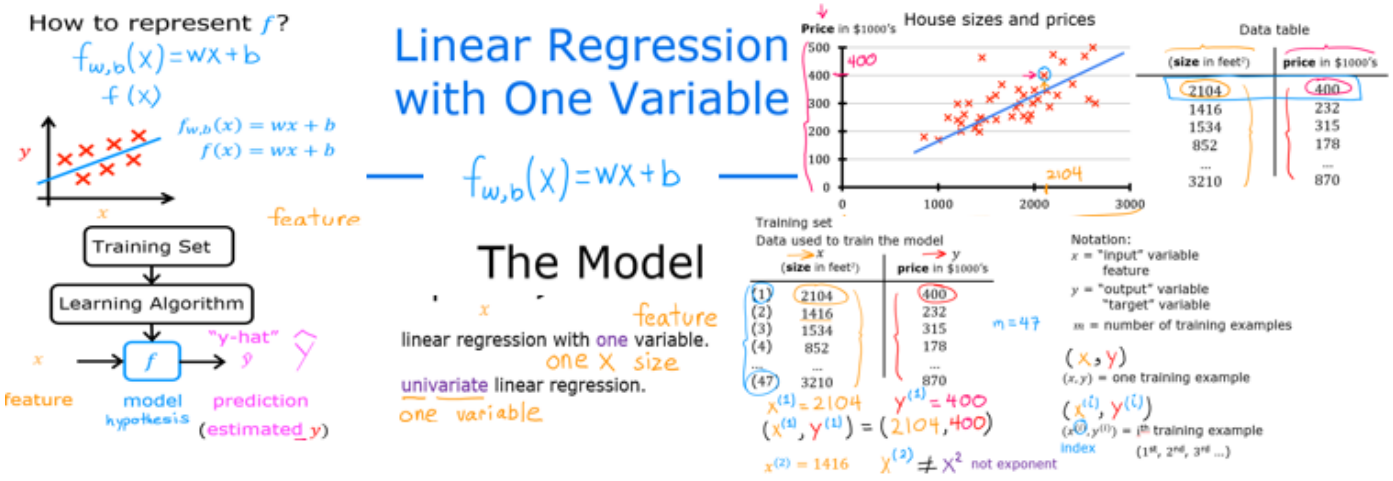
Regressione lineare

Tramite la *regressione lineare* univariata, viene determinato matematicamente l'output della funzione dato l'input, o se vogliamo dirla in altro modo, viene calcolata la Y in funzione della X. (tipicamente determinato la funzione che meglio fitta di valori X e Y)

La RL serve per determinare i valori della funzione che meglio soddisfano l'andamento di un fenomeno relativo ai valori appartenenti all'insieme delle fetures/labels (x,y)

La funzione si può rappresentare come y = b + wx dove **b** è “intercetta” (ovvero il delta y rispetto allo zero detto anche “bias” in inglese) mentre la **w** è il “coefficiente angolare. (ovvero l'inclibazione della retta detto anche ”weight" in inglese o “slope” ovvero pendenza nel caso della funzione)

Nel Machine Learning (ML) i parametri “w” e “b” sono detti anche coefficienti o pesi.

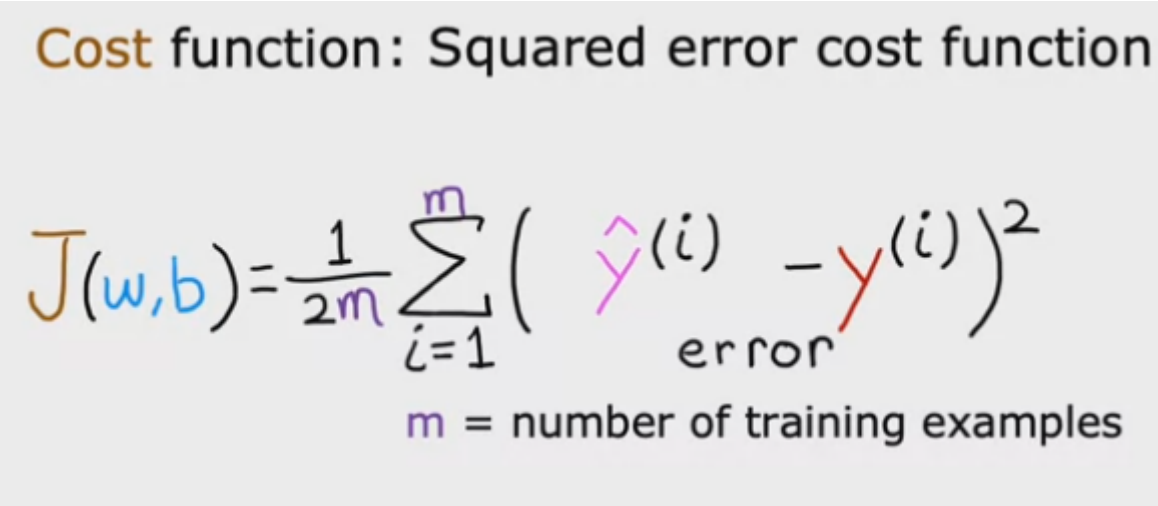


Costo della funzione

La Cost Function (CF) nella pratica confronta il valore “predetto” y^ e il valore di training y, nella pratica è una differenza tra i due valori che definiamo come “errore”, ponendo il delta al quadrato, ovvero:

Sommatoria di (y^ - y)^2 per tutti i valori “m” del training set; diviso per “m” ovvero, viene ritornato la media degli errori -> questo si chiama “metodo dei minimi quadrati”

L'intento è quindi quello di trovare il valore minimo della funzione di costo.



Si tratta quindi di trovare la funzione che minimizza l'errore.

Nell'immagine sotto riportata a sx vengono visualizzate le funzioni ottenibili al variare del permarametro “w”

NB: per semplicità l'esempio pone b= 0 per rendere il grafico più facilemente interpretabile perchè in questo modo il gradico è 2D, se considerassimo anche

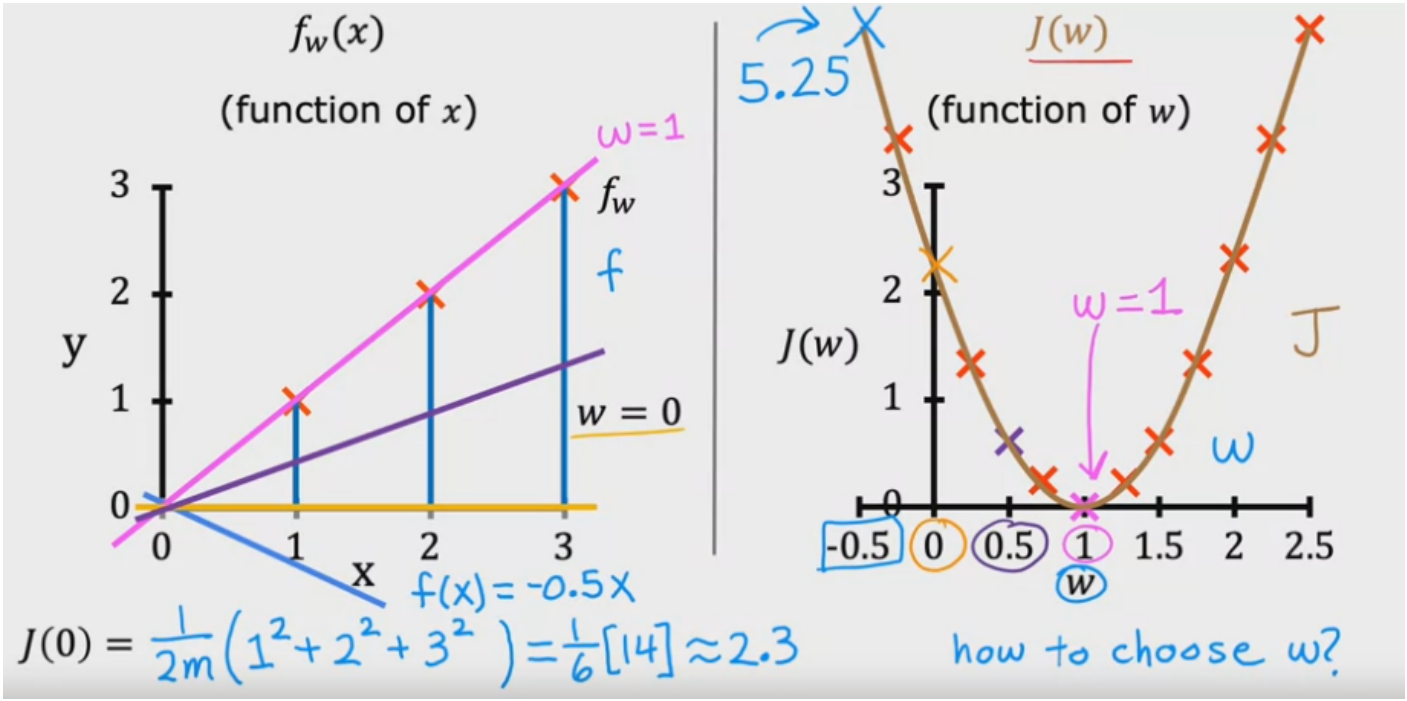
il valore b, allora il grafico sarebbe 3D e quindi un pò più difficile da leggere. (vedi es. grafico 3D riportato in pagina)

Si può notare che nella parte sx al varia della funzione si ottengono diversi valori J (ovvero errore) che, se rappresenti graficamente disegnano una curva

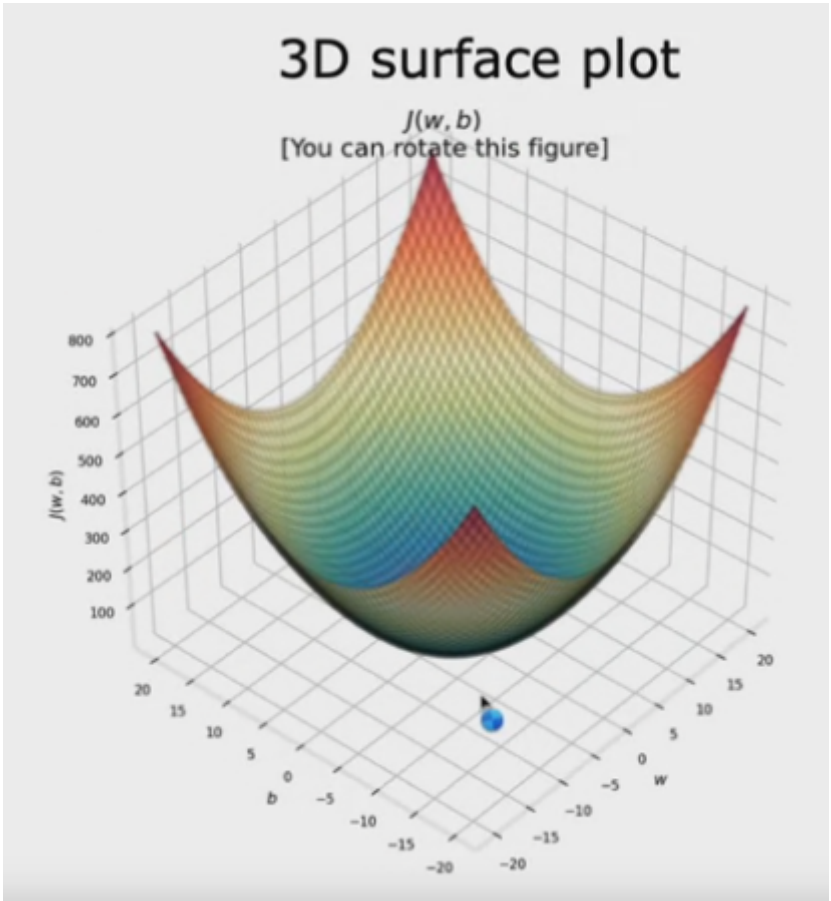
dove l'errore minimo si trova quando, per es. nel caso specifico, w vale 1. (grafico a DX)

Il grafico a DX disegna sulle ordinate (y) l'errore mentre sulle ascisse (x) il valore w ottenuti dall'applicazione del metodo

dei minimi quadrati. (vedi grafico a SX)



Nel caso in cui considerassimo anche il parametro “b", il grafico del cost function sarebbe renderezzito in 3D, come sotto rappresentato.



Considerazioni finali

Per determinare il “costo della funzione minimo” o l'errore minimo bisogna determinare i parametri “w” e “b” che avvicinano la funzione al set di dati.

Per determinare questi parametri in maniera algoritmica viene utilizzata la tecnica matematica denominata “gradiant descent” o “discesa del gradiente” analizzata nel paragrafo successivo.

Discesa del gradiente

La discesa del gradiente (GD) è un modo sistematico per determinare i parametri “w” e “b” in modo che minimizzino

il "costo della funzione" (l'errore della funzione)

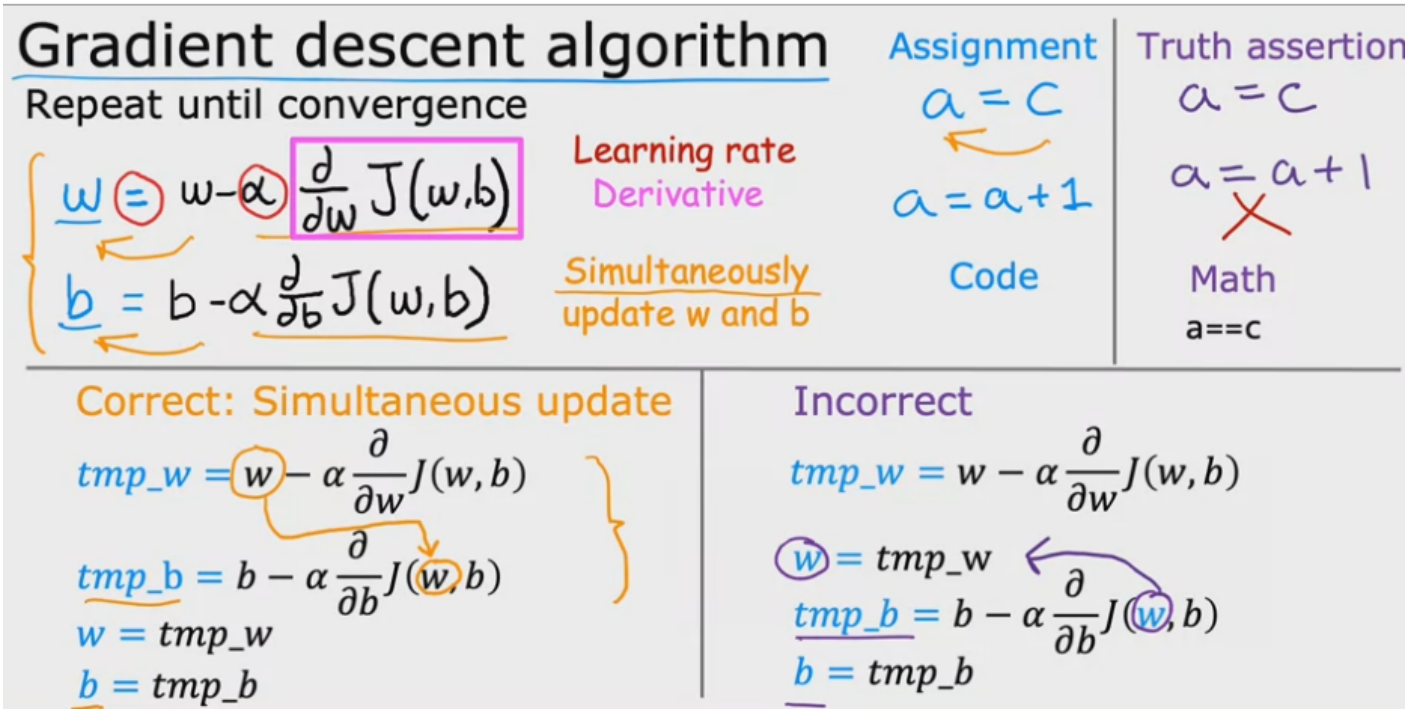
Ricordo che stiamo parlando della funzione i cui coefficienti (w e b) contribuiscono a determinare

i valori che più si avvicinano al set di dati passati in input (features e labels)

Quindi J(w,b) -> costo della funzione (ovvero la sommatoria dei delta dati da labels - labels calcolate con i coifficienti w)

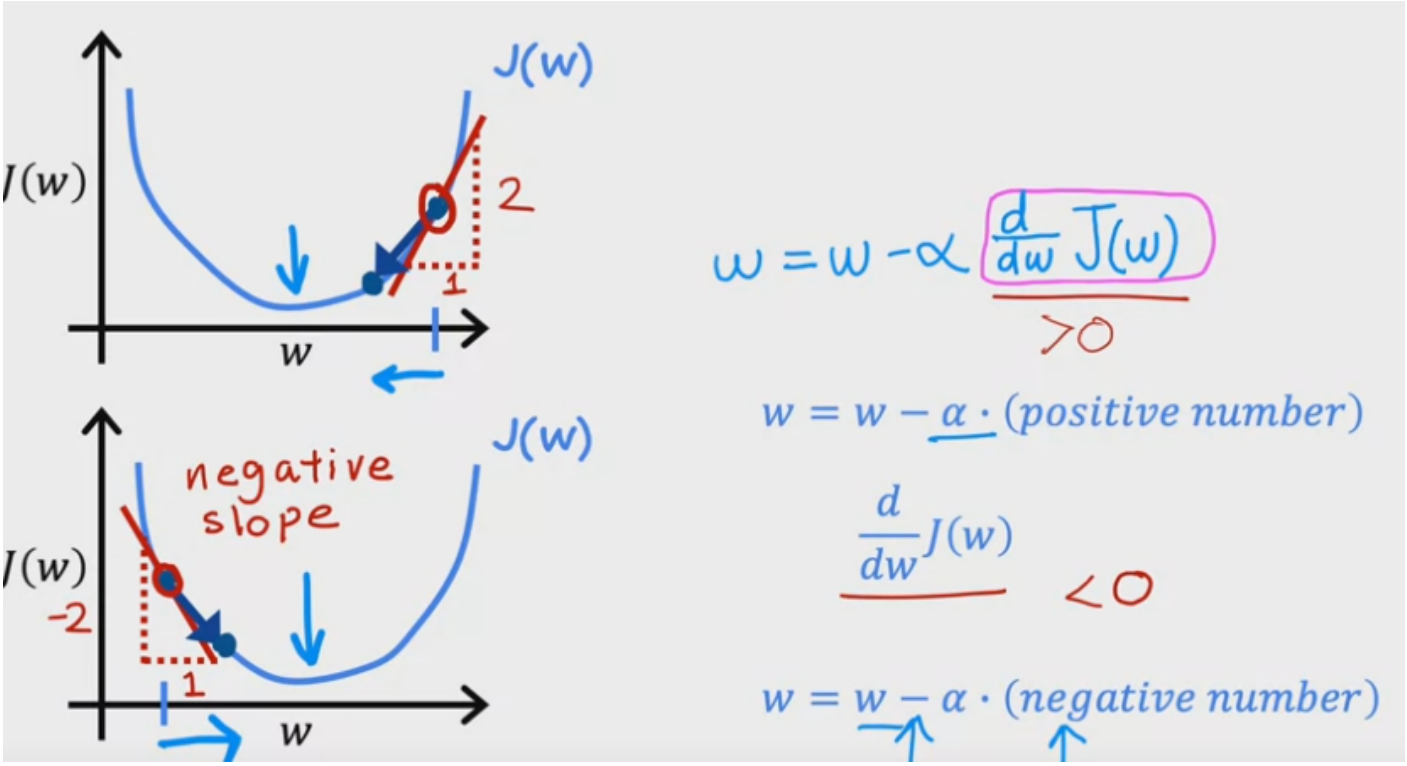
Tramite questo metodo si procede per “piccoli passi” al fine di trovarel l'errore minimo J(x,b), nota che

possono esistere più minimi (detti anche “minimi locali”) che dipendono dal valore di partenza che in genere è randomico.



Nella pratica si tratta di looppare i valori “w” e "b" calcolati ad ogni iterazione del ciclo. (vedi immagine sopra)
Il valore di w è quindi settato a = “w” meno un parametro “alpha” (detto anche learning rate, valore piccolo compreso tra 0 e 1, es. 0,001) moltiplicato
per la derivata del costo della funzione J(w,b).
Stessa cosa per il parametro b.

Il valore di “learing rate” (LR) è indica la grandezza del “passo”, ovvero la velocità con la quale saliamo i discendiamo il grafico
del costo della funzione.
Attenzione che andare veloce può causare un “salto” eccessivo e farci perdere il punto di minimo.



Sopra viene indicato il calcolo della derivata parziale rispetto al valore “w”. Vengono rappresentati due casi, il primo

dove viene preso a caso un valore “w” a DX dal minimo; In questo caso la derivata parziele ritornerà un numero positivo

in quanto, la derivata indica l'inclinazione della tangente passante per il punto scelto sulla curva avente per coordinate (w1,j(w1))

e in questo caso indica che la tangente è ascendente.

La derivata del costo della funzone indica se la funzione è a un minimo (anche locale) oppure se si trova in discesa o in salita. In pratica

restituisce la pendenza (o la direzione) del punto tangente alla funzione di costo nelle coordinate indicate.

Ora il nuovo “w” viene calcolato sotrraendo il " LR x lla derivata" al valore “w” , essendo positivo, diminuirà il valore finare di “w”.

Stessa cosa, ma invertita di segno in quanto il valore di w preso a SX del minimo è discendete. Quindi, il valore della derivato

del costo della funzione viene negato in quanto la formula sottrae sempre il valore della derivta parziale del costo di “w” e di “b”.

Idem per la variabile “b” fino a quando i valori convergono intorno al minimo della funzione.

Learning rate

Il valore di “learning rate” (LR) nella pratica serve per fare “lo step” nella direzione del minimo in cui valore è dato dal vosto della funzione.

LR però non può essere un valore troppo piccolo in quanto rallenterebbe in maniera importante la determinazione del minimo e, non può

essere nemmeno troppo grande in quanto rischia di far saltare il punto di minimo, sia esso locale che globale.

Per questo motivo il modo migliore per settare il parametro alpha (detto anche LR) è gestire dinamicamente il valore da algoritmo in modo

che sia un valore relativamente grande se la pendenza (slope) dato dalla derivata nel punto della funzione di costo è elevato, piccolo se lo

slope tende allo zero in quanto significa che siamo prossimi al minimo.

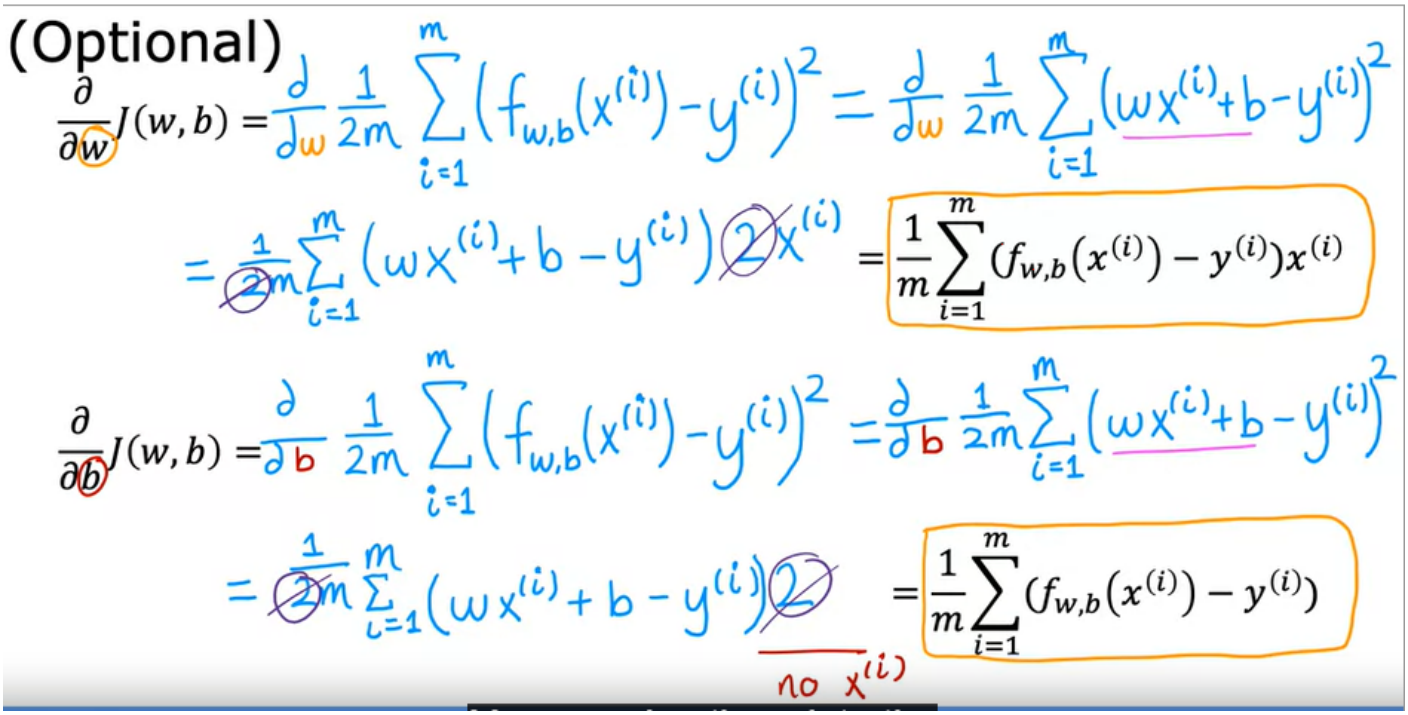
Di seguito viene mostrato il calcolo delle deritavate parziali rispetto a “w” e rispetto a “b" per determinare il minumo del costo della funzione.

NOTA di calcolo, si tratta di applicare il teorema della derivata della funzione composta *(detta anche "chain rule)* rispetto a **w** e rispetto a **b** ovvero:

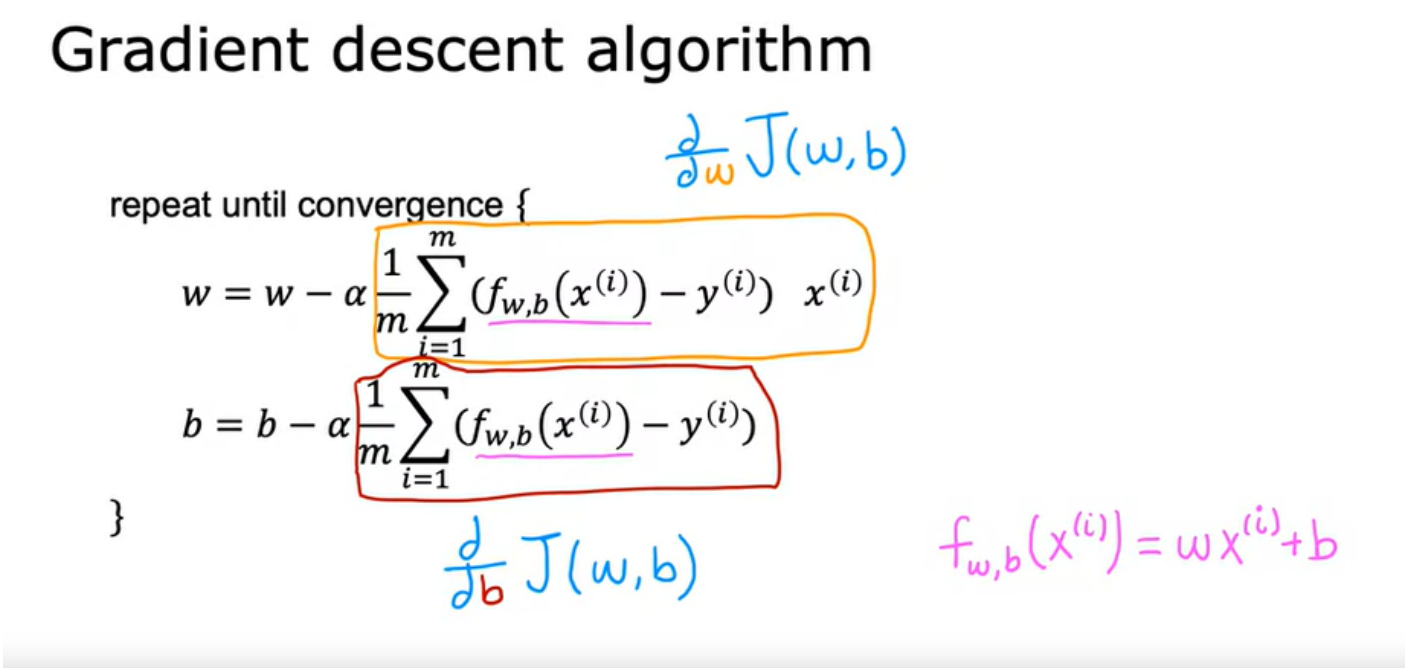
$$\frac{d}{dx}h(x) = g'(f(x)) \cdot f'(x)$$

In parole povere, la derivata della funzione composta h(x)=g(f(x)) è data dalla derivata della funzione più esterna, *con argomento invariato*, moltiplicata per la derivata della funzione più interna. Con funzione più esterna si intende l'ultima funzione che si applica nella composizione (per noi la g) mentre la più interna è la prima che si applica (f).

che applicando il tutto alla funzione di costo si ottiene:



Quindi l'algoritmo di calcolo della discesa del gradiente si calcola come:



Conclusione

In conclusione, la regressione lineare consente di determinare il valore minimo relativo al costo della funzione.

Il costo della funzione ritorna la sommatoria degli scostamenti tra la funzione (ad una o più variabili) e l'insieme dei valori (campioni)

labels-features al variare dei coefficiente “w” e “b”.

Al variare di questi coefficienti si genera un insieme di valori dove ciascun valore è un totale che se rappresentato

graficamente, disegna un grafico in 2D (se varia solo un valore) in 3D se variano sia “w” che “b”, del quale dobbiamo trovare

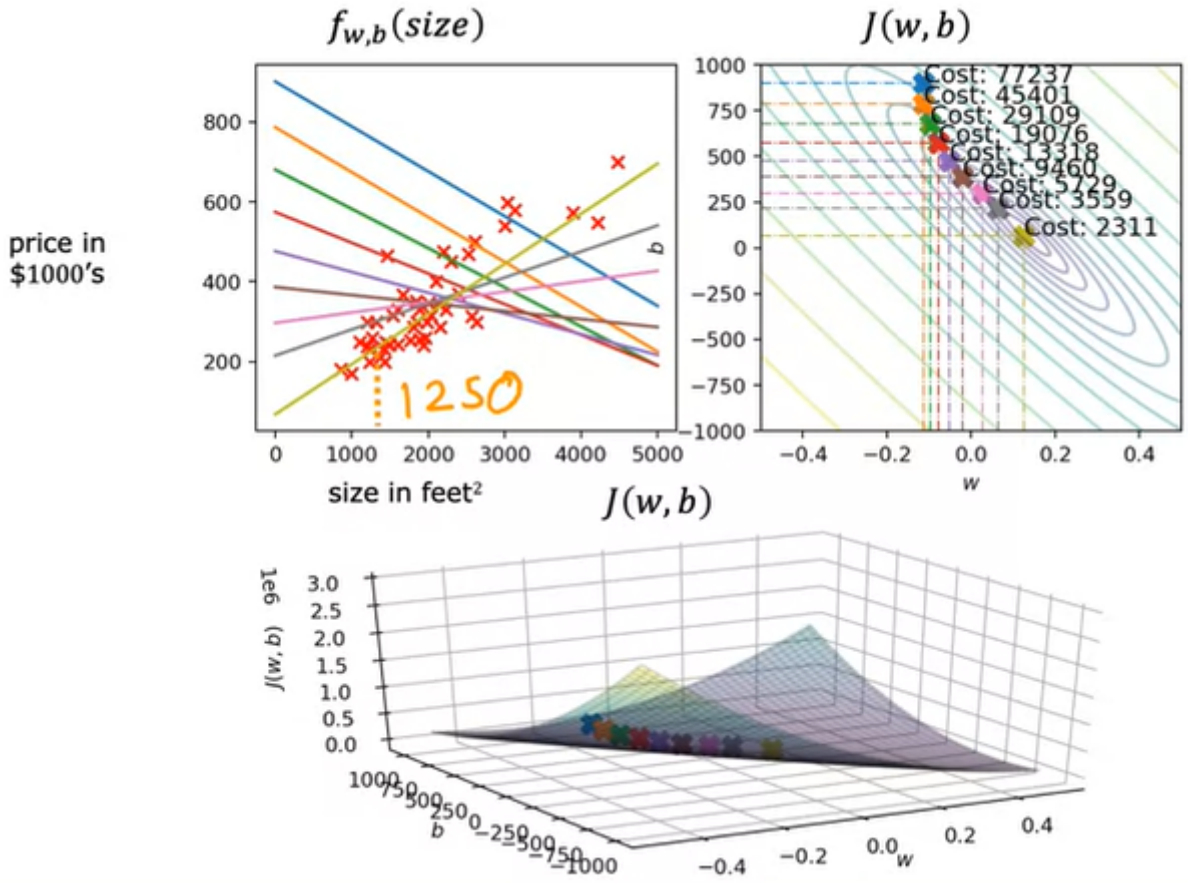
il minimo per identificare la funzione che minimizza l'errore.

Per trovare l'errore minimo si utilizza il metodo della discesa del gradiende, che consente tramite l'utilizzo delle derivate parziali

nel punto iesimo, di avvicinarsi progressivamente al “minimo locale” oppure al “minimo assoluto” avanzando tramite un passo definito “**learing rate**”.

Di seguito un esempio di come al variare dell'intercetta e del coefficiente angolare, andando con passo (LR) la funzione

giunge al suo costo minimo.


$f_{w,b}(size)$
price in $1000's
size in feet²
1250
$J(w,b)$
Cost: 77237
Cost: 45401
Cost: 29109
Cost: 19076
Cost: 13318
Cost: 9460
Cost: 5729
Cost: 3559
Cost: 2311
w
b
$J(w,b)$
$J(w,b)$
1e6

Regressione lineare multipla

PREMESSA:

f(w,b) = modello

J(wb) = costo della funzione -> f (w,b) - y -> dove y sono le label

DISCESA DEL GRADIENTE = d/dw J(w,b) in sistema con d/db J(wb)

Nella regressione lineare univariata (RLS) abbiamo solo una "feature" e una corrispettiva “label”, per es. metri di un appartemento e corrispettivo prezzo.

Nel caso della regressione lineare multipla (RLM) invece, abbiamo più features a fronte di una label, per es. oltre ai metri quadrati dell'appartemento

abbiamo anche il numero di stanze, l'età dell'immobile, piano, etc etc e ovviamente come label il prezzo.

NB: la RLM non è la regressione lineare multivariata, che non conosco.

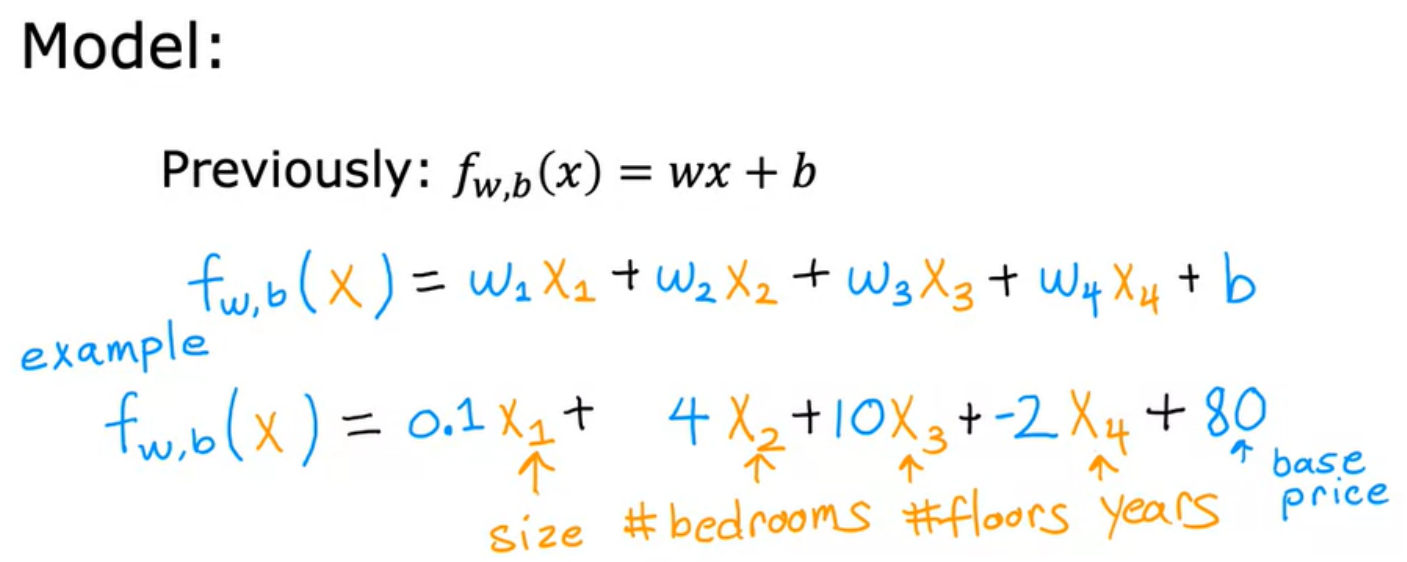
Il modello che era stato definito per la regressione lineare singola si basa sulla funzione f(x) = wx + b che in pratica definisce la funzione

(attraverso l'opportuno settaggio dei paramerti w e b tramite la discesa del gradiente) che meglio si accosta alle features/label definite per il training.

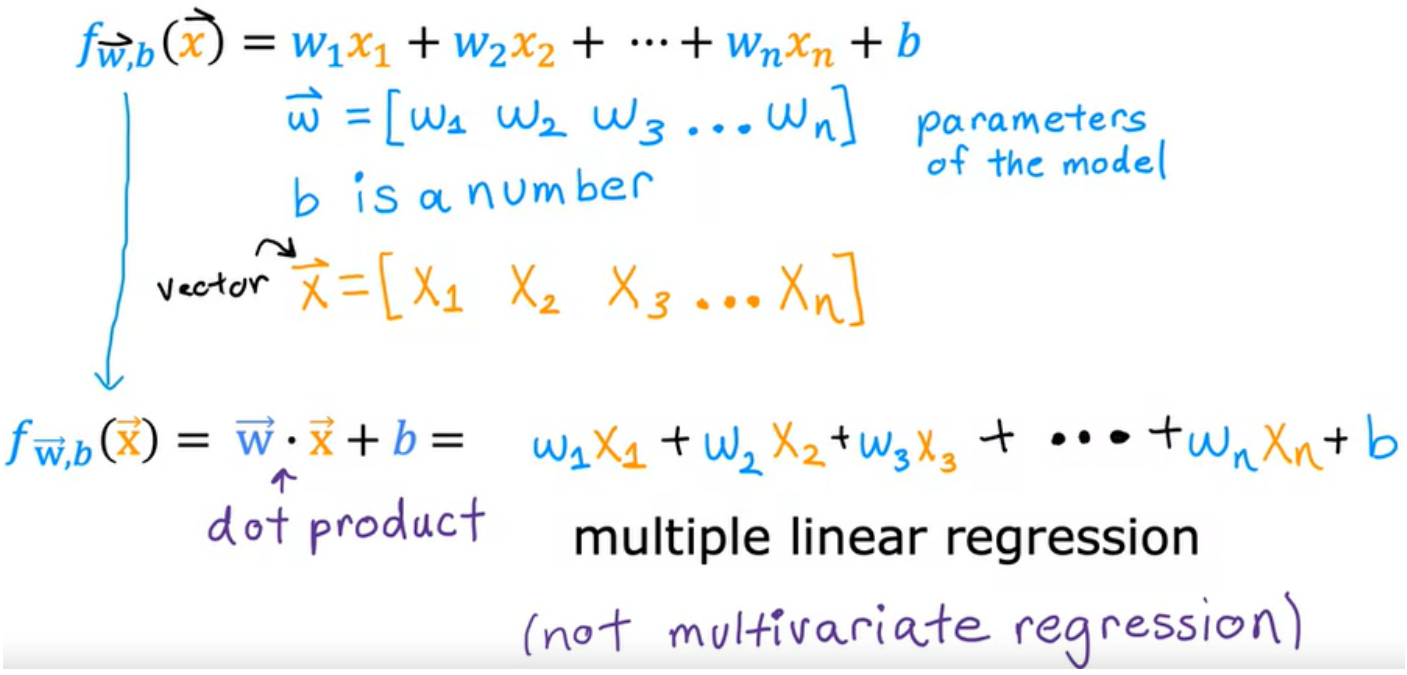
Nel caso invece della RLM, la formula diventa un polinomio del tipo (consideranto 4 features) f(x) = w1x1 + w2x2 + w3x3 + w4x4 + b

Dove x1,2,3,4 sono le features, mentre w1,2,3,4 sono i coeffienti angolari tutti potenzialmente diversi.

es:



che si può rappresentare anche:



Per risolvare questa equazione viene utilizzato il metodo della Vettorizzazione. (Vectorization)

La Vettorizzazione consente nella pratica di efettuare la moltiplicazione tra vettori/matrici utilizzando la libria numpy che sfrutta appino l'hardware della macchina.

Discesa del gradiente

Il GD a più variabili è simile a quello univariato con la differenza che al posto di un solo “w” e una sola “b” c'è un vettore di w

Il calcolo è simile se non per il fatto che bisogna iterare per il numero di features

Nella pratica la regressione lineare multipla si calcola in 2 marco step:

0) date le features e le labels di esempio:

X_train = np.array([[2104, 5, 1, 45],

[1416, 3, 2, 40],

[852, 2, 1, 35]])

matrice di 3 righe di traing con 4 colonne di features per ogni riga

e

y_train = np.array([460, 232, 178])

una riga di labels

e

dati dei valorei a caso di “w” e “b”

b_init = 785.1811367994083

w_init = np.array([0.39133535, 18.75376741, -53.36032453, -26.42131618])

1) calcolo della funzione di costo su variabili multiple, come sotto riportato

The equation for the cost function with multiple variables $J(\mathbf{w}, b)$ is:

$$J(\mathbf{w}, b) = \frac{1}{2m} \sum_{i=0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)})^2 \tag{3}$$

where:

$$f_{\mathbf{w},b}(\mathbf{x}^{(i)}) = \mathbf{w} \cdot \mathbf{x}^{(i)} + b \tag{4}$$

In contrast to previous labs, $\mathbf{w}$ and $\mathbf{x}^{(i)}$ are vectors rather than scalars supporting multiple features.

Below is an implementation of equations (3) and (4). Note that this uses a *standard pattern for this course* where a for loop over all `m` examples is used.

```
[11]: def compute_cost(X, y, w, b):
          """
          compute cost
          Args:
            X (ndarray (m,n)): Data, m examples with n features
            y (ndarray (m,)) : target values
            w (ndarray (n,)) : model parameters
            b (scalar)       : model parameter

          Returns:
            cost (scalar): cost
          """
          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
```

```
[12]: # Compute and display cost using our pre-chosen optimal parameters.
      cost = compute_cost(X_train, y_train, w_init, b_init)
      print(f'Cost at optimal w : {cost}')
```

```
Cost at optimal w : 1.5578904880036537e-12
```

Expected Result: Cost at optimal w : 1.5578904045996674e-12

Il metodo “compute_cost” serve per calcolare il costo della funzione per i valore w e b passati per una SOLA iterazione.

Sarà poi la fase successiva a variare i w e b richiamando poi la funzione di costo per determinare il costo totale per poi ritare “w” e “b” di conseguenza

2) Calcolo della discesa del gradiente con variabili multiple

applicando la derivata del “calcolo della funzione di costo” loopando fino a che i valore “w” e “b” minimizzano il costo

***NB**: ricordo che si applica il metodo della "derivata della funzione composta" accennato nella sezione “GRADENT DESCENT” relativa al paragrafo*

della regressione lineare univariate.

Gradient descent for multiple variables:

repeat until convergence: {

$$w_j = w_j - \alpha \frac{\partial J(\vec{\mathbf{w}}, b)}{\partial w_j} \qquad \text{for j = 0..n-1}$$

$$b = b - \alpha \frac{\partial J(\vec{\mathbf{w}}, b)}{\partial b}$$

}

where, n is the number of features, parameters w_j, b, are updated simultaneously and where

$$\frac{\partial J(\vec{\mathbf{w}}, b)}{\partial w_j} = \frac{1}{m} \sum_{i=0}^{m-1} (f_{\vec{\mathbf{w}},b}(\mathbf{x}^{(i)}) - y^{(i)}) x_j^{(i)}$$

$$\frac{\partial J(\vec{\mathbf{w}}, b)}{\partial b} = \frac{1}{m} \sum_{i=0}^{m-1} (f_{\vec{\mathbf{w}},b}(\mathbf{x}^{(i)}) - y^{(i)})$$

- m is the number of training examples in the data set
- $f_{\mathbf{w},b}(\mathbf{x}^{(i)})$ is the model's prediction, while $y^{(i)}$ is the target value

Nella pratica il calcolo delle derivate parziali in “w” e “b” si traduce in:

```python
def compute_gradient(X, y, w, b):
    """
    Computes the gradient for linear regression
    Args:
      X (ndarray (m,n)): Data, m examples with n features
      y (ndarray (m,)) : target values
      w (ndarray (n,)) : model parameters
      b (scalar)       : model parameter

    Returns:
      dj_dw (ndarray (n,)): The gradient of the cost w.r.t. the parameters w.
      dj_db (scalar):       The gradient of the cost w.r.t. the parameter 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
```

```python
#Compute and display gradient
tmp_dj_db, tmp_dj_dw = compute_gradient(X_train, y_train, w_init, b_init)
print(f'dj_db at initial w,b: {b_init:.9f} {tmp_dj_db:.9f}')
print(f'dj_dw at initial w,b: \n {tmp_dj_dw}')
```

```
dj_db at initial w,b: 785.181136799 -0.000001674
dj_dw at initial w,b:
 [-2.73e-03 -6.27e-06 -2.22e-06 -6.92e-05]
```

Expected Result:
dj_db at initial w,b: -1.6739251122999121e-06
dj_dw at initial w,b:
[-2.73e-03 -6.27e-06 -2.22e-06 -6.92e-05]

che viene richiamata in un loop di N iterazioni, ovvero:

```python
def gradient_descent(X, y, w_in, b_in, cost_function, gradient_function, alpha, num_iters):
    """
    Performs batch gradient descent to learn theta. Updates theta by taking
    num_iters gradient steps with learning rate alpha

    Args:
      X (ndarray (m,n))   : Data, m examples with n features
      y (ndarray (m,))    : target values
      w_in (ndarray (n,)) : initial model parameters
      b_in (scalar)       : initial model parameter
      cost_function       : function to compute cost
      gradient_function   : function to compute the gradient
      alpha (float)       : Learning rate
      num_iters (int)     : number of iterations to run gradient descent

    Returns:
      w (ndarray (n,)) : Updated values of parameters
      b (scalar)       : Updated value of parameter
      """

    # 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
        dj_db,dj_dw = gradient_function(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( cost_function(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(f"Iteration {i:4d}: Cost {J_history[-1]:8.2f}   ")

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

In the next cell you will test the implementation.

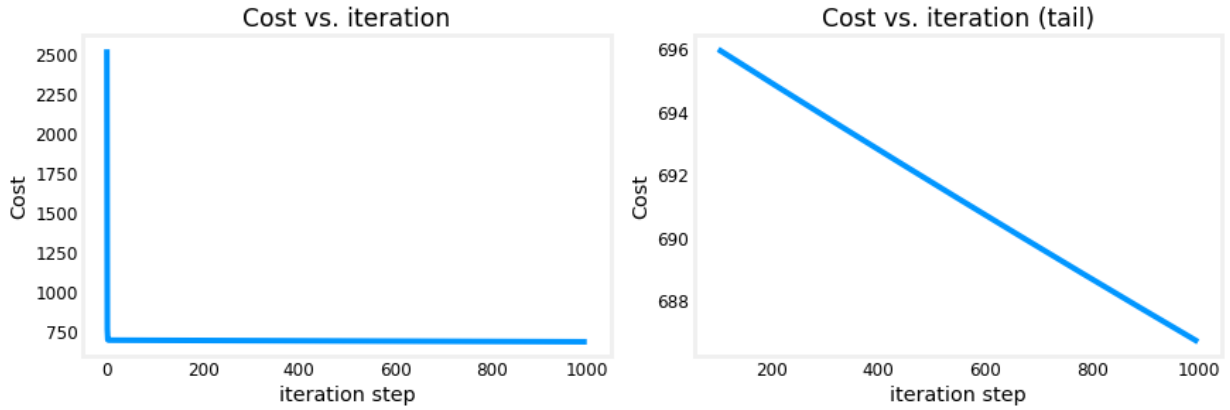
```python
# initialize parameters
initial_w = np.zeros_like(w_init)
initial_b = 0.
# some gradient descent settings
iterations = 1000
alpha = 5.0e-7
# run gradient descent
w_final, b_final, J_hist = gradient_descent(X_train, y_train, initial_w, initial_b,
                                                    compute_cost, compute_gradient,
                                                    alpha, iterations)
print(f"b,w found by gradient descent: {b_final:0.2f},{w_final} ")
m,_ = X_train.shape
for i in range(m):
    print(f"prediction: {np.dot(X_train[i], w_final) + b_final:0.2f}, target value: {y_train[i]}")
```

```
Iteration    0: Cost  2529.46
```

In conclusione questo è la discesa del gradiente, purtroppo i risultati ottenuti non sono particolarmente brillanti, dopo
verrà illustrato come migliorare l'algoritmo.

Expected Result:
b,w found by gradient descent: -0.00,[0.2 0. -0.01 -0.07]
prediction: 426.19, target value: 460
prediction: 286.17, target value: 232
prediction: 171.47, target value: 178

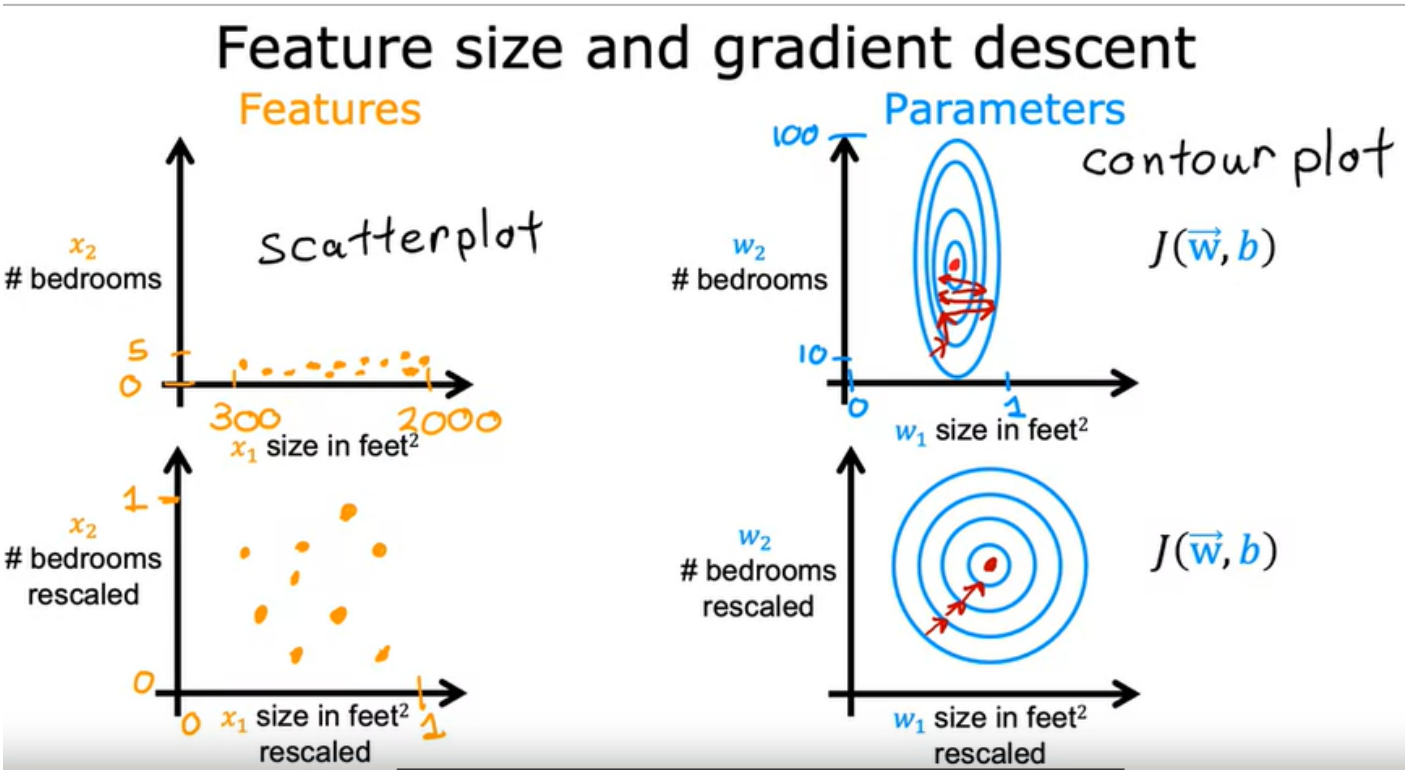
1]:
```
# plot cost versus iteration
fig, (ax1, ax2) = plt.subplots(1, 2, constrained_layout=True, figsize=(12, 4))
ax1.plot(J_hist)
ax2.plot(100 + np.arange(len(J_hist[100:])), J_hist[100:])
ax1.set_title("Cost vs. iteration");  ax2.set_title("Cost vs. iteration (tail)")
ax1.set_ylabel('Cost')     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



These results are not inspiring! Cost is still declining and our predictions are not very accurate. The next lab will explore how to improve on this.

Scaling delle features e dei parametri

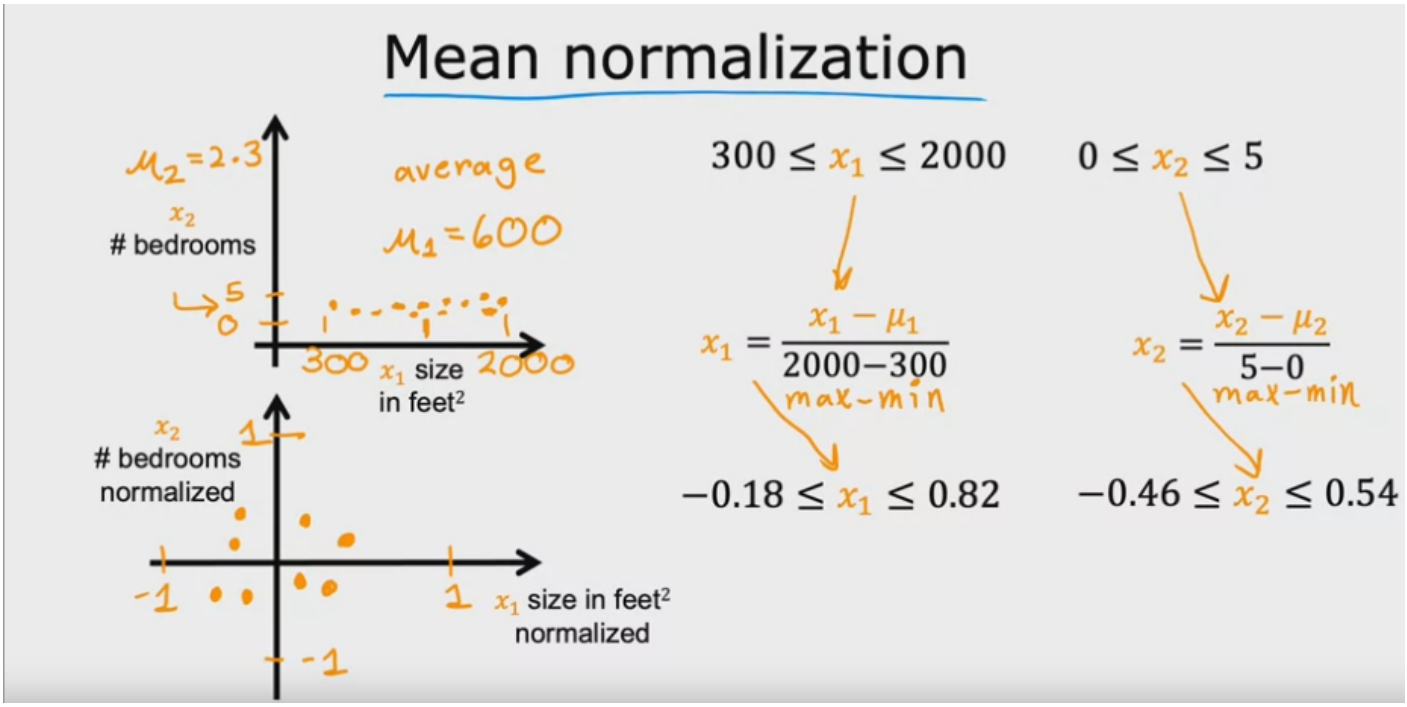
Nel caso in cui ci siano più features (come nella regressione lineare multipla) è importante scalare i valori delle diverse tipologie di parametri in input.
Es. se abbiamo due features come i metri quadrati e il numero di stanze di un appartmento, è bene riportare entrambi gli insiemi di valori
in un range compreso tra -1 e 1.
Il motivo è dettato dal fatto che in questo modo la regressione lineare trova più facilmente (velocemente) il suo minimo.
(vedi i grafici sotto riportati che indicano il minimo dei costi nella parte SX)



SCALING:

Le features e le label vanno quindi “normalizzati” scalandoli principalmente in questi modi:
1) dividere tutti i valori per il massimo
2) “centrando” i valori intorno allo zero applicando la formula (valori-valore medio)/(valore max - valore min)

di seguito un esempio:



un metodo ottimale per normalizzare i dati è detto Z-SCORE che riporto di seguito:

Z-SCORE

Questo metodo nella pratica va a “centrare” le features e le labels inforno allo zero. In questo modo il calcolo della regressione

lineare risulterà più veloce e più accurato nella ricerca del valore minimo relativo al costo della funzione.

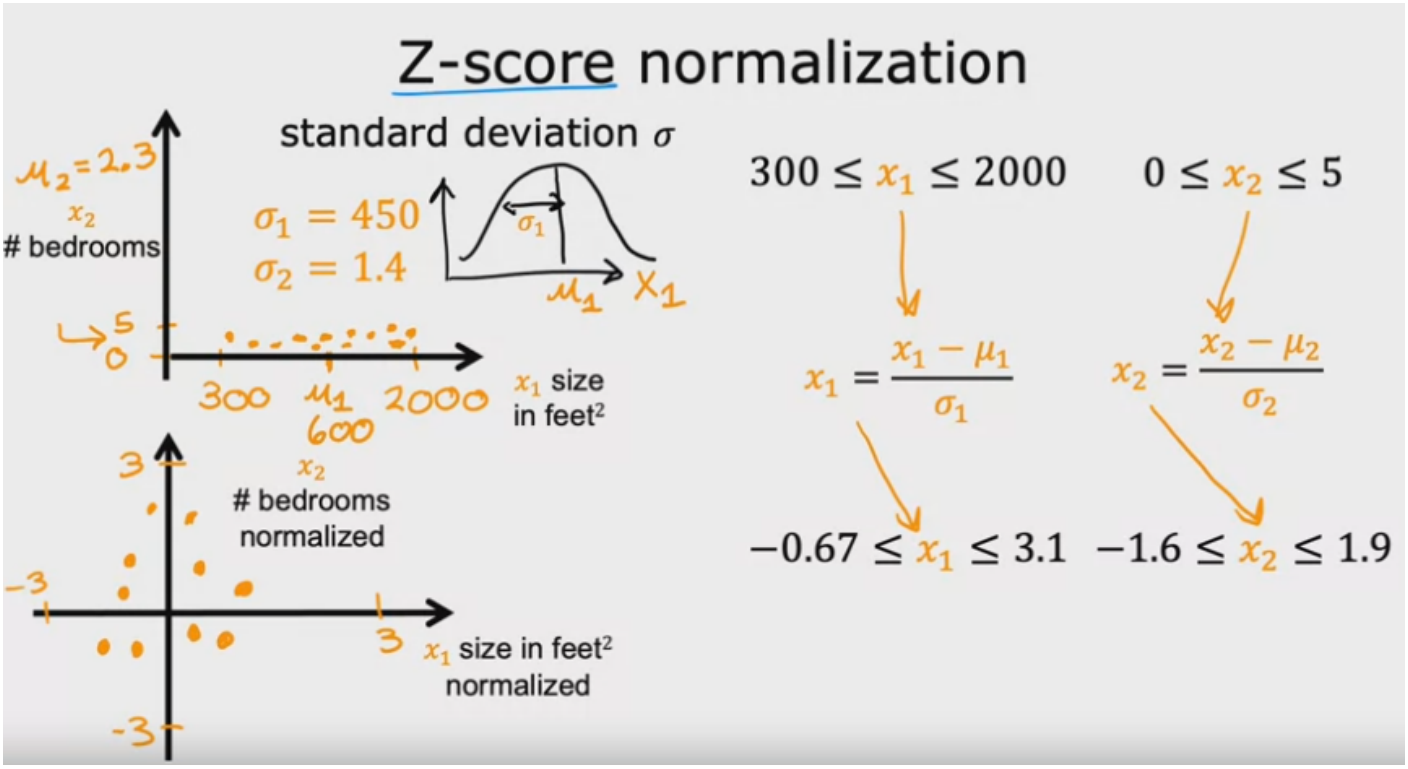
DEVIAZIONE STANDARD (o scarto quadratico medio)

La DS rappresenza rappresenta la distanza dei valori di una serie rispetto alla media e si calcaola come:

1) calcolare la media dei valori -> media semplice

2) calcare la varianza dei valori -> è la differenza tra il singolo valore e la media al quadrato il tutto diveso per il totale dei campioni

3) calcolare la daviazione -> è la radice quadrata della varianza



implementazione dell'algoritmo di ZScore

```
def zscore_normalize_features(X):
    """
    computes  X, zcore normalized by column

    Args:
      X (ndarray (m,n))     : input data, m examples, n features

    Returns:
      X_norm (ndarray (m,n)): input normalized by column
      mu (ndarray (n,))     : mean of each feature
      sigma (ndarray (n,))  : standard deviation of each feature
    """
    # find the mean of each column/feature
    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)
```

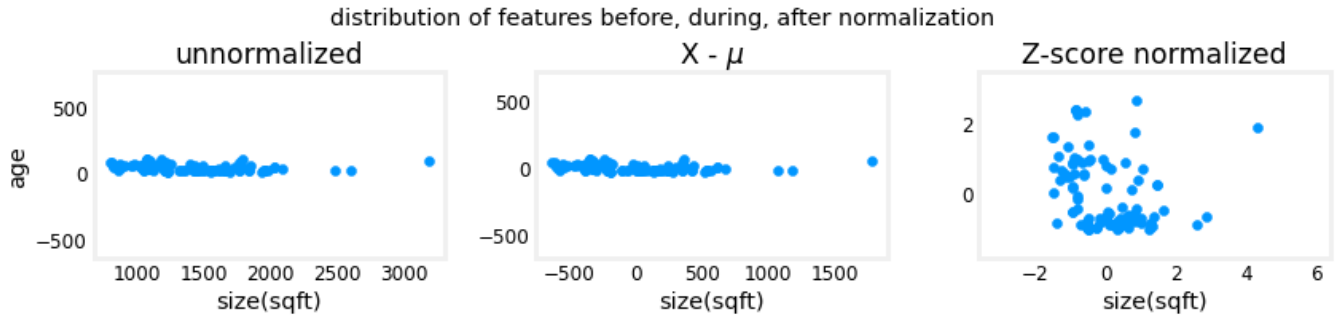
il cui output nell'esempio:

```
mu     = np.mean(X_train,axis=0)
sigma  = np.std(X_train,axis=0)
X_mean = (X_train - mu)
X_norm = (X_train - mu)/sigma

fig,ax=plt.subplots(1, 3, figsize=(12, 3))
ax[0].scatter(X_train[:,0], X_train[:,3])
ax[0].set_xlabel(X_features[0]); ax[0].set_ylabel(X_features[3]);
ax[0].set_title("unnormalized")
ax[0].axis('equal')

ax[1].scatter(X_mean[:,0], X_mean[:,3])
ax[1].set_xlabel(X_features[0]); ax[0].set_ylabel(X_features[3]);
ax[1].set_title(r"X - $\mu$")
ax[1].axis('equal')

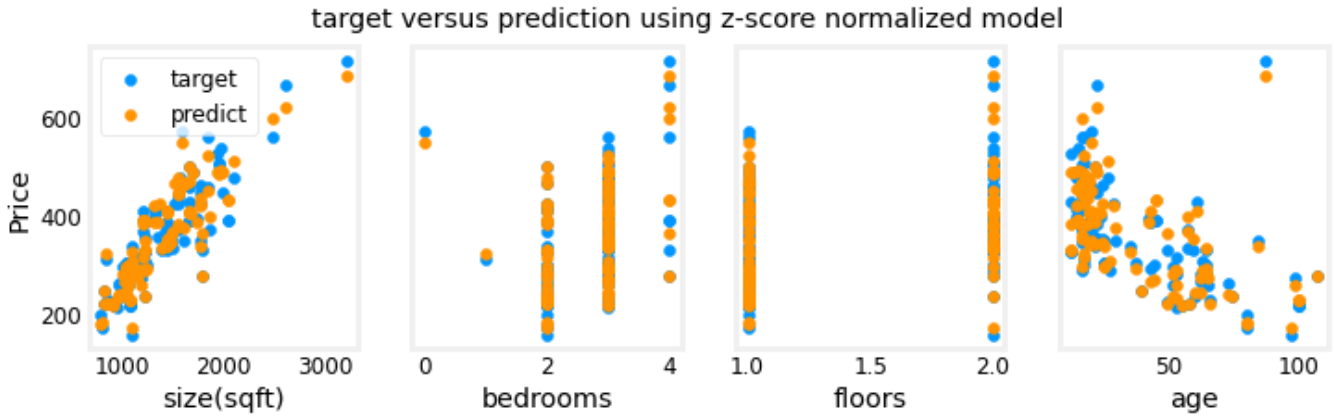
ax[2].scatter(X_norm[:,0], X_norm[:,3])
ax[2].set_xlabel(X_features[0]); ax[0].set_ylabel(X_features[3]);
ax[2].set_title(r"Z-score normalized")
ax[2].axis('equal')
plt.tight_layout(rect=[0, 0.03, 1, 0.95])
fig.suptitle("distribution of features before, during, after normalization")
plt.show()
```



e adesso proviamo a predirre i valori.

```
#predict target using normalized features
m = X_norm.shape[0]
yp = np.zeros(m)
for i in range(m):
    yp[i] = np.dot(X_norm[i], w_norm) + b_norm

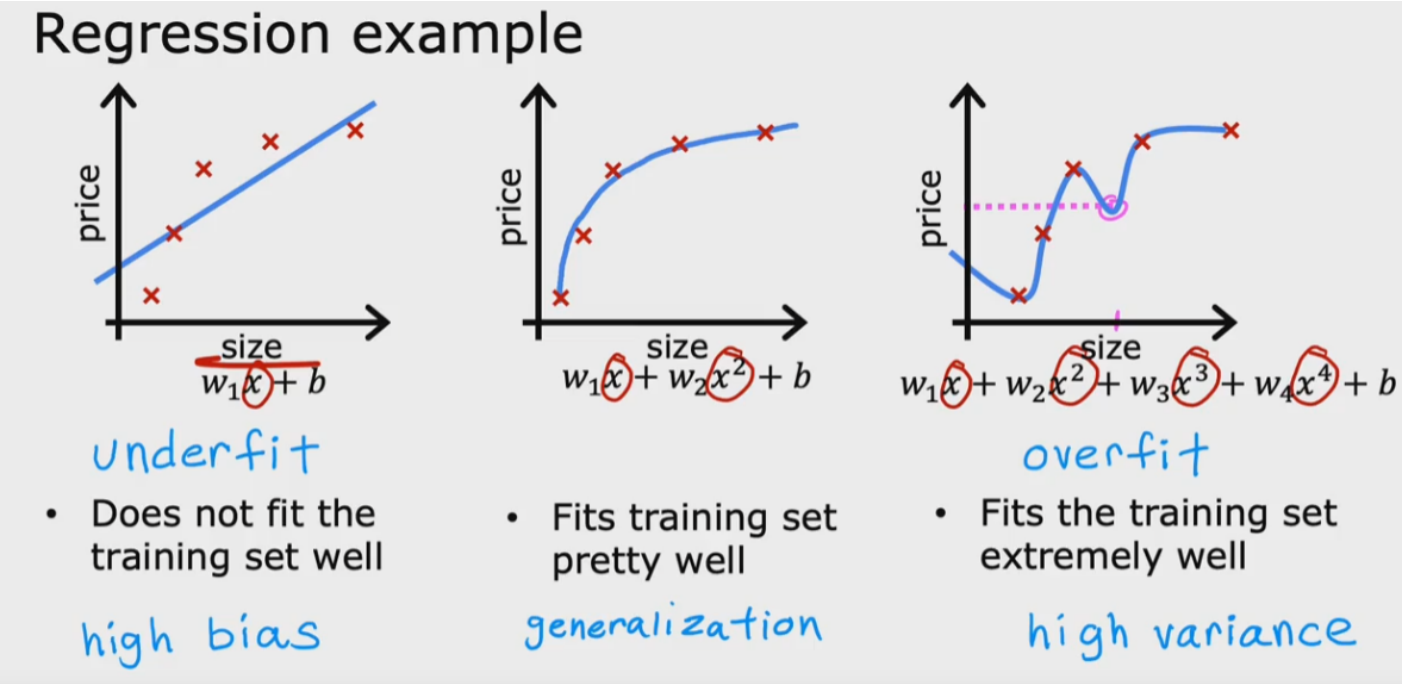
    # plot predictions and targets versus original features
fig,ax=plt.subplots(1,4,figsize=(12, 3),sharey=True)
for i in range(len(ax)):
    ax[i].scatter(X_train[:,i],y_train, label = 'target')
    ax[i].set_xlabel(X_features[i])
    ax[i].scatter(X_train[:,i],yp,color=dlc["dlorange"], label = 'predict')
ax[0].set_ylabel("Price"); ax[0].legend();
fig.suptitle("target versus prediction using z-score normalized model")
plt.show()
```



Regressione polinomiale

Nella regressione lineare polinomiale la funzone di costo ragione per curve e non per linee rette come
invece accade della regressione lineare (singola o multipla)
Per ottenere questo risultato vengono utilizzate dei confficienti quadrati o cububici o valori esponenziali > di 3.
oppure utilizzare, al contrario, la radice quadrata della feature.

es:



che poi viene dettaglio nel capitolo relativo all'overfitting/underfitting