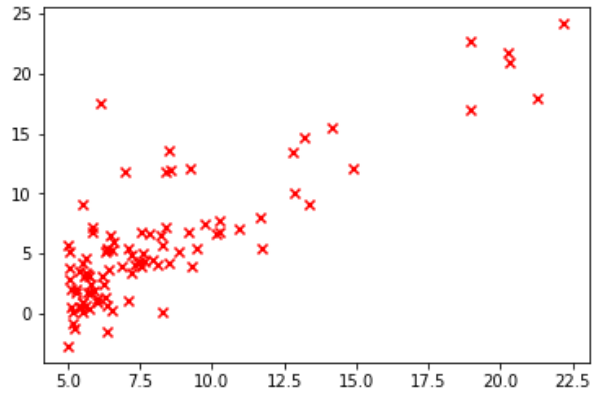
Práctica 1: Regresión Lineal

Hemos implementado un algoritmo que, mediante regresión lineal, es capaz de predecir valores de salida en base a una o más variables de entrada. Los datos de entrenamiento se han obtenido de dos archivos, *ex1data1.csv* y *ex1data2.csv*. Al leer los datos del primer fichero, obtenemos una distribución como la que se muestra en la siguiente figura:

```
In [3]: data = carga_csv("ex1data1.csv")
        X = data[:, :-1]
        Y = data[:, -1]

        plt.scatter(X, Y, color='red', marker='x')
        plt.show()
```



Aunque en este primer ejemplo trabajamos con una sola variable, vamos a generalizar el código para que sea extensible a varias variables. En primer lugar, añadiremos una columna de unos a la matriz *X* obtenida de nuestro fichero. El tamaño de esta columna es el número de ejemplos de entrenamiento, es decir, *m*:

```
In [4]: m = np.shape(X)[0]
        X = np.hstack([np.ones([m, 1]), X])
        X.shape
```

```
Out[4]: (97, 2)
```

De este modo también garantizamos que podremos expresar la función h_θ como producto de matrices $X \cdot \theta$. La idea es ir ajustando el valor de los elementos de una matriz *Theta* (*Z*) en varias iteraciones, para que, al aplicar la función *J* a los ejemplos de entrenamiento, el coste vaya disminuyendo. Esta matriz *Z* cuenta con tantos elementos θ como variables de entrada tenga nuestro modelo; matemáticamente, esto se traduce en el número de columnas de la matriz *X*, incluyendo la columna de unos. En cada iteración, se calcula el valor de cada elemento de *Z* en base a los resultados obtenidos en la iteración anterior. En la primera iteración, se presupone un valor de 0 para todos los elementos de *Z*.

```
In [16]: m = np.shape(X)[0]
         n = np.shape(X)[1]
         Theta = np.zeros(n)
```

La función *gradiente* calcula el valor de las θ en cada iteración. Es una implementación directa de la expresión matemática del gradiente:

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

In [12]:
```
def gradient(X, Y, Z, a):
    m = np.shape(X)[0]
    n = np.shape(X)[1]

    z_new = Z
    H = np.dot(X, Z)
    Aux = (H - Y)

    for i in range(n):
        Aux_i = Aux * X[:, i]
        z_new[i] -= (a / m) * Aux_i.sum()

    return z_new
```

La función *coste* calcula cuánto se desvía la predicción de $X \cdot \theta$ del valor real de *Y*. Es una implementación directa de la expresión matemática del coste por diferencia de cuadrados:

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

In [14]:
```
def j(X, Y, Z):
    H = np.dot(X, Z)
    Aux = (H - Y) ** 2
    return Aux.sum() / (2 * len(X))
```

Así, con un factor de aprendizaje $a = 0.01$ y 1500 iteraciones, obtenemos los valores aproximados de *Theta* que minimizan la función de coste:

In [13]:
```
def gradient_descent(X, Y, a):
    m = np.shape(X)[0]
    n = np.shape(X)[1]

    Thetas = np.ndarray((1500, n))
    costes = np.ndarray(1500)
    Theta = np.zeros(n)

    for i in range(1500):
        Thetas[i] = gradient(X, Y, Theta, a)
        costes[i] = j(X, Y, Thetas[i])

    return Thetas, costes
```

Como podemos comprobar, el coste ha ido disminuyendo en cada una de las iteraciones:

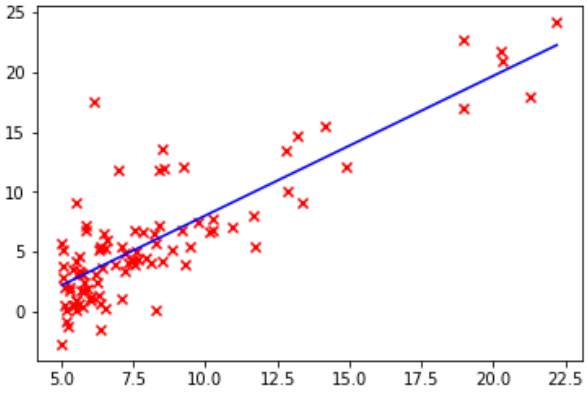
In [23]:
```
a = 0.01
Thetas, costes = gradient_descent(X, Y, a)
costes
```

Out[23]:
```
array([6.73719046, 5.93159357, 5.90115471, ..., 4.48343473, 4.48341145,
       4.48338826])
```

Aplicando a los parámetros de entrada los últimos valores de *Theta* obtenidos en el descenso de gradiente, obtenemos nuestra recta de regresión lineal:

In [48]:
```
x_sample = np.array([np.amin(X), np.amax(X)])
x_aux = np.hstack([np.ones([2, 1]), x_sample.reshape(2, 1)])
y_sample = np.dot(x_aux, Thetas[1499])

plt.scatter(X, Y, color='red', marker='x')
plt.plot(x_sample, y_sample, color='blue')
plt.show()
```

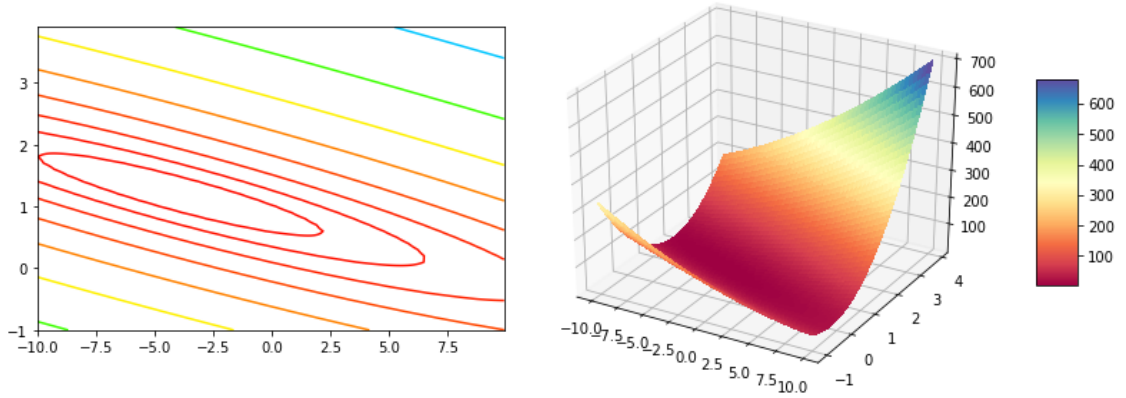


Una forma más visual de presentar los datos sobre el coste es con funciones de superficie tridimensional y funciones de contorno, aunque primero debemos preparar los datos:

In [57]:
```
def make_data(Z0_range, Z1_range, X, Y):
    step = 0.1
    z0 = np.arange(Z0_range[0], Z0_range[1], step)
    z1 = np.arange(Z1_range[0], Z1_range[1], step)
    Z0, Z1 = np.meshgrid(z0, z1)

    J = np.empty_like(Z0)
    for ix, iy in np.ndindex(Z0.shape):
        J[ix, iy] = j(X, Y, [Z0[ix, iy], Z1[ix, iy]])

    return [Z0, Z1, J]
```



Para realizar el descenso de gradiente con varias variables sólo necesitaremos ajustar los datos de entrada, ya que durante la primera parte de la práctica nos hemos ocupado de generalizar los algoritmos para cualquier número de variables. En primer lugar, debemos normalizar la matriz *X* obtenida del archivo *ex1data2.csv*:

In [77]:
```
def normalize(X):
    mu = np.mean(X, axis=0)
    sigma = np.std(X, axis=0)

    for ix, iy in np.ndindex(X.shape):
        X[ix, iy] = (X[ix,iy] - mu[iy]) / sigma[iy]

    return X, mu, sigma
```

Con la matriz *X* normalizada (*X_norm*) podemos realizar el descenso de gradiente igual que antes. Ponemos a prueba nuestro algoritmo de predicción con unos valores de 1650 pies cuadrados y 3 habitaciones; estos valores deben normalizarse previamente para que el resultado sea correcto. Obtenemos una predicción *Y* de precio 293098.4666 dólares:

In [92]:
```
data = carga_csv("ex1data2.csv")
X = data[:, :-1]
Y = data[:, -1]

m = np.shape(X)[0]
X_norm, mu, sigma = normalize(X)
X_norm = np.hstack([np.ones([m, 1]), X_norm])

a = 0.01
Thetas, costes = gradient_descent(X_norm, Y, a)

print(Thetas[1499])
print(np.dot([1.0, (1650.0 - mu[0]) / sigma[0],
              (3.0 - mu[1]) / sigma[1]], Thetas[1499]))
```

```
[340412.56301439 109370.05670466  -6500.61509507]
293098.4666757651
```

Sin embargo, existe un método directo para minimizar la función de coste y obtener los valores de θ adecuados. La función *ecuación normal* minimiza la función de coste, y es una implementación directa de su expresión matemática:

$$\theta = (X^T X)^{-1} X^T Y$$

In [93]:
```
def normal_equation(X, Y):
    return np.linalg.inv(X.T @ X) @ X.T @ Y
```

Aplicando esta ecuación normal no es necesario normalizar los datos de entrada, por lo que repetimos el proceso para predecir el precio de un piso con 1650 pies cuadrados y 3 habitaciones:

In [96]:
```
data = carga_csv("ex1data2.csv")
X = data[:, :-1]
Y = data[:, -1]

m = np.shape(X)[0]
X = np.hstack([np.ones([m, 1]), X])

Theta = normal_equation(X, Y)
print(Theta)
print(np.dot([1.0, 1650.0, 3.0], Theta))
```

```
[89597.9095428    139.21067402 -8738.01911233]
293081.46433489426
```

Como puede observarse, el precio apenas varía en 17 dólares, por lo que podemos concluir que nuestra estimación por descenso de gradiente era bastante buena, aunque no óptima. Si quisiéramos ajustar nuestra estimación, deberíamos modificar la tasa de aprendizaje. Así, como último ejercicio, vamos a comprobar el efecto sobre el coste de variar la tasa de aprendizaje en el descenso de gradiente:

In [98]:
```
a = 0.3
Thetas, costes = gradient_descent(X_norm, Y, a)
plt.plot(costes, color='red')

a = 0.1
Thetas, costes = gradient_descent(X_norm, Y, a)
plt.plot(costes, color='yellow')

a = 0.03
Thetas, costes = gradient_descent(X_norm, Y, a)
plt.plot(costes, color='green')

a = 0.01
Thetas, costes = gradient_descent(X_norm, Y, a)
plt.plot(costes, color='blue')

a = 0.003
Thetas, costes = gradient_descent(X_norm, Y, a)
plt.plot(costes, color='magenta')

plt.show()
```