Initialization

In [1]:
%load_ext autoreload
%autoreload 2
%matplotlib inline

In [2]:
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

Training Set
The training set contains \( m \) examples.

- Each example is a pair, comprising a vector of feature values and the value of the dependent variable, \( \langle x, y \rangle \).

- Here are three examples from a property prices dataset:
  \[
  \begin{bmatrix}
  92.9 \\
  3 \\
  2
  \end{bmatrix},
  \begin{bmatrix}
  171.9 \\
  4 \\
  3
  \end{bmatrix},
  \begin{bmatrix}
  79 \\
  3 \\
  1
  \end{bmatrix}
  \]
  (The prices are in thousands of €)

- But we need notation that distinguishes one example from another. We will use a superscript.
  - \( \langle x^{(i)}, y^{(i)} \rangle \) will be the \( i \)th example
  - The first example in the training set is \( \langle x^{(1)}, y^{(1)} \rangle \), the second is \( \langle x^{(2)}, y^{(2)} \rangle \), \ldots, the last is \( \langle x^{(m)}, y^{(m)} \rangle \).
  - We’re writing the superscript in parentheses to make it clear that we are using it for indexing. It is not ‘raising to a power’. If we want to raise to a power, we will drop the parentheses.
    - So if \( y^{(2)} = 9 \), what is \( y \)? (Trick question) And if \( y^2 = 9 \), what is \( y \)?
  - We can use both superscript and subscripts to select feature values from particular examples.
    - In the property price dataset shown above, what is \( x_2^{(1)} \)?
    - What is \( x_1^{(2)} \)?

From a training set \( \{ \langle x^{(1)}, y^{(1)} \rangle, \langle x^{(2)}, y^{(2)} \rangle, \ldots, \langle x^{(m)}, y^{(m)} \rangle \} \), we can construct a matrix \( X \) and a vector \( y \) as follows:

\[
X = \begin{bmatrix}
  x_1^{(1)} & x_2^{(1)} & \cdots & x_n^{(1)} \\
  x_1^{(2)} & x_2^{(2)} & \cdots & x_n^{(2)} \\
  \vdots & \vdots & \ddots & \vdots \\
  x_1^{(m)} & x_2^{(m)} & \cdots & x_n^{(m)}
\end{bmatrix},
\]

\[
y = \begin{bmatrix}
  y^{(1)} \\
  y^{(2)} \\
  \vdots \\
  y^{(m)}
\end{bmatrix}
\]

- You can think of row \( i \) as the transpose of \( x^{(i)} \)
- For the three property price examples given above, we have
  \[
  X = \begin{bmatrix}
  92.9 & 3 & 2 \\
  171.9 & 4 & 3 \\
  79 & 3 & 1
  \end{bmatrix},
  y = \begin{bmatrix}
  175 \\
  435 \\
  85
  \end{bmatrix}
  \]

**Linear Regression using Ordinary Least Squares**

**Hypotheses**
We will study how to learn a linear model using linear regression. The algorithm must choose from an infinite set of hypotheses, each one a linear function of \( x \). Assuming \( n \) features, each hypothesis will have the following form:

\[
h_\beta(x) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n
\]

\( \beta \) is a vector of parameters — in effect, the learning algorithm must find the parameter values that give a hypothesis that fits the training data well.

Before we see how to do that, let’s extend vector \( x \) to make the maths easier:

- instead of a vector of \( n \) elements indexed from 1 to \( n \)
- it is now a vector of \( n + 1 \) elements indexed from 0 to \( n \)
- but we will stipulate that the value of element \( x_0 \) is always 1
- this affects matrix \( X \) that we showed above: it will now have an extra element \( x_0^{(i)} \) in each row \( i \), all of which will be 1

\[
X = \begin{bmatrix}
    x_0^{(1)} & x_1^{(1)} & x_2^{(1)} & \ldots & x_n^{(1)} \\
    x_0^{(2)} & x_1^{(2)} & x_2^{(2)} & \ldots & x_n^{(2)} \\
    \vdots & \vdots & \vdots & \ddots & \vdots \\
    x_0^{(m)} & x_1^{(m)} & x_2^{(m)} & \ldots & x_n^{(m)}
\end{bmatrix}
\]

e.g. \( X = \begin{bmatrix} 1 & 92.9 & 3 & 2 \\ 1 & 171.9 & 4 & 3 \\ 1 & 79 & 3 & 1 \end{bmatrix} \)

This changes our hypotheses:

- They are now of the form:

\[
h_\beta(x) = \beta_0 x_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n
\]

with \( x_0 = 1 \)

- And, as we know, if we are given a set of examples \( X \) and we want to evaluate the hypothesis on each example, we can use vectorization:

\[
X \beta
\]

The learning algorithm is faced with lots of hypotheses, \( h_\beta \), each with a different vector of parameters \( \beta \). It wants to find the vector \( \beta \) such that, for each example \( x^{(i)} \) in the training set, \( h_\beta(x^{(i)}) \) is as close to the actual value \( y^{(i)} \) as possible. In vector terms, we want vector \( X \beta \) to be close to vector \( y \).

We make this more precise by defining a loss function that can measure how close \( h_\beta(x^{(i)}) \) is to \( y^{(i)} \) (or \( X \beta \) is to \( y \)).

**The Loss Function for Ordinary Least-Squares Regression**
The first linear regression algorithm that we will study is called ordinary least-squares regression (OLS). It gets its name from the form of its loss function.

- In ordinary least-squares regression, the loss function, $J$, is the square of the difference between the two values, and this must be summed (or, as here, averaged) over all examples in the training set:
  $$J_\beta(X, y) = \frac{1}{2m} \sum_{i=1}^{m} (h_\beta(x^i) - y^{(i)})^2$$

- Why do we square the differences?
- Does it make any difference that we halve the total?

So the goal is to find parameters $\beta$ that minimize $J(\beta)$.

The Loss Function in Python

The parameters of this function are beta (the vector of parameter values, represented by a 1d numpy array), X (the training set, which is a 2D numpy array), y (the target values, a 1D numpy array). So we can use a lot of vectorization:

```
In [3]: def J(beta, X, y):
   ...:     """
   ...:     Loss function for OLS regression
   ...:     """
   ...:     h = X.dot(beta)
   ...:     differences = h.flatten() - y
   ...:     sq_differences = differences ** 2
   ...:     return (1.0 / (2 * y.size)) * sq_differences.sum()
```

2D Visualization of $J_\beta$

Let's visualize $J_\beta$ using the Cork property dataset as the training set. We'll assume that $\beta_j = 0$ for all $j$ except $j = 1$. In other words, we are pretending that floor area is the only relevant feature. Then we can plot $J_\beta$ on the vertical axis against different values of $\beta_1$ on the horizontal axis.
```python
import matplotlib.gridspec as gridspec

# Use pandas to read the CSV file
df = pd.read_csv("dataset-corkA.csv")

# Get the feature-values and the target values into separate numpy arrays of numbers
X = df[['flarea']].values
y = df['price'].values

# Set up the two subplots
fig = plt.figure(figsize=(8, 4))
gs = gridspec.GridSpec(1, 2, width_ratios=[2, 1])
# Lefthand diagram
ax0 = plt.subplot(gs[0])
plt.title("Training set and hypothesis")
plt.xlabel("Floor area (sq metres")
plt.xlim(-500, 500)
plt.ylabel("Price (000 euros")
plt.ylim(0, 1000)
# Righthand diagram
ax1 = plt.subplot(gs[1])
plt.title("Loss function")
plt.xlabel("$beta_1$")
plt.xlim(-10, 10)
plt.ylabel("$J$")
plt.ylim(0, 1500000)
fig.tight_layout()

# Hypothesis (which assumes beta_i = 0 for all i except i = 1)
def h(beta1, x):
    return beta1 * x

# Here's the statement you should vary: try different values of beta1
# To see anything on these axes, try beta1 between -10 and +10
beta1 = 8

# Scatter plot of the training set (lefthand diagram)
ax0.scatter(X, y, color = "green")

# Straight line plot of the hypothesis for this value of beta1 (lefthand diagram)
xvals = np.linspace(-500, 500, 3)
ax0.plot(xvals, h(beta1, xvals), color = 'blue')

# Show the value of the loss function for this value of beta1 (righthand diagram)
ax1.scatter(beta1, J(beta1, X, y), color = "red")
plt.show()
```
Another 2D Visualization of $J_\beta$

Instead of making manual adjustments, let's use a loop to try several values for $\beta_1$. 
In [5]:
# Set up the two subplots
fig = plt.figure(figsize=(8, 4))
gs = gridspec.GridSpec(1, 2, width_ratios=[2, 1])
# Lefthand diagram
ax0 = plt.subplot(gs[0])
plt.title("Training set and hypothesis")
plt.xlabel("Floor area (sq metres)")
plt.xlim(-500, 500)
plt.ylabel("Price (000 euros)"
plt.ylim(0, 1000)
# Righthand diagram
ax1 = plt.subplot(gs[1])
plt.title("Loss function")
plt.xlabel("$\beta_1$")
plt.xlim(-10, 10)
plt.ylabel("$J$"
plt.ylim(0, 150000)
fig.tight_layout()

# Different values of beta1
betas = np.linspace(-10, 10, 21)

# Scatter plot of the training set (leifthand diagram)
ax0.scatter(X, y, color = "green")

# Straight line plot of the hypothesis for this value of beta1 (left hand diagram)
xvals = np.linspace(-500, 500, 3)
for beta1 in betas:
    ax0.plot(xvals, h(beta1, xvals), color = 'blue')

# Show the value of the loss function for this value of beta1 (right hand diagram)
ax1.scatter(betas, [J(beta1, X, y) for beta1 in betas], color = "red")
plt.show()
The loss function is **convex**. Informally, this means:

- in 2D, it is u-shaped
- it has a unique minimum

This is no accident: it follows from the way the loss function has been defined.

### 3D Visualization of $J_\beta$

Let's visualize $J_\beta$ again using the Cork property dataset as the training set. This time, we'll assume that $\beta_j = 0$ for all $j$ except $j = 2$ and $j = 3$. In other words, we are pretending that the number of bedrooms and bathrooms are the only relevant features. This will be a 3D plot with $J_\beta$ on the vertical axis against different values of $\beta_2$ and $\beta_3$ on the horizontal axes.

```python
In [6]: from mpl_toolkits.mplot3d import Axes3D

# Get the feature-values and the target values into separate numpy arrays
X = df[['bdrms', 'bthrms']].values
y = df['price'].values

fig = plt.figure()
ax = Axes3D(fig)
ax.set_title("Loss function")
ax.set_xlabel("$\beta_2$")
ax.set_ylabel("$\beta_3$")
ax.set_zlabel("$J$")
xvals = np.linspace(-100, 200, 301)
yvals = np.linspace(-100, 200, 301)
xxvals, yyvals = np.meshgrid(xvals, yvals)
zs = np.array([J([beta2, beta3], X, y) for beta2, beta3 in zip(np.ravel(xxvals), np.ravel(yyvals))])
zvals = zs.reshape(xxvals.shape)
ax.plot_surface(xxvals, yyvals, zvals)
plt.show()
```
Another Visualization of $J_\beta$

Here is the same data on a different kind of plot, called a contour plot. In effect, it flattens the diagram above. The lines connect points that have the same value for $J_\beta$.

```python
In [7]:
fig = plt.figure(figsize = (6, 6))
plt.title("Loss function")
plt.xlabel("$\beta_2$")
plt.ylabel("$\beta_3$")
xvals = np.linspace(-100, 200, 301)
yvals = np.linspace(-100, 200, 301)
xxvals, yyvals = np.meshgrid(xvals, yvals)
zs = np.array([J([beta2, beta3], X, y) for beta2, beta3 in zip(np.ravel(xxvals), np.ravel(yyvals))])
zvals = zs.reshape(xxvals.shape)
C = plt.contour(xxvals, yyvals, zvals, 15, colors = 'black', linewidth = 0.5)
plt.clabel(C, inline=1, fontsize=10)
plt.show()
```

The 3D visualization and the contour plot show that this too is convex. Informally, this means:

- in 3D it is bowl-shaped
- again there is a unique minimum

Gradient of $J_\beta$
We can find the gradient of $J_\beta$ by taking its partial derivatives

$$\frac{\partial J_\beta(X, y)}{\partial \beta_0}, \frac{\partial J_\beta(X, y)}{\partial \beta_1}, \ldots, \frac{\partial J_\beta(X, y)}{\partial \beta_n}$$

If you do the calculus, you find that

$$\frac{\partial J_\beta(X, y)}{\partial \beta_j} = \frac{1}{m} \sum_{i=1}^{m} (h_\beta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

**The Normal Equation**

Since $J_\beta$ is convex, we know it has a unique minimum and we can find this point by setting the gradient to zero. Then, with some algebraic manipulation, we can come up with what is known as the normal equation, which solves for $\beta$. In other words, the normal equation gives us the parameters that minimize the loss function:

$$\beta = (X^T X)^{-1} X^T y$$

So, using numpy.linalg, we can easily write a Python program to solve for $\beta$.

In [8]: import numpy.linalg as npla

    # Insert the extra feature (all ones)
    df.insert(loc=0, column='ones', value=1)

    # Get the feature-values and the target values into separate numpy arrays of numbers
    X = df[['ones', 'flarea', 'bdrms', 'bthrms']].values
    y = df['price'].values

    # Solve the normal equation
    beta = npla.inv(X.T.dot(X)).dot(X.T).dot(y)

    # Display beta
    beta

Out[8]: array([ 62.50261909,  4.6900952 , -72.28169508, -57.05464555])

You might notice that these are the same values that we obtained in the introductory lecture.

But, as we know, there’s a problem. The normal equation requires that $X^T X$ has an inverse. And it might not.

When we discussed this in the previous lecture, we mentioned that, in some cases, we can use the pseudo-inverse instead. And this is one of those cases. So the more robust way of writing this program is:

In [9]: # Solve the normal equation - but using the pseudo-inverse
    beta = npla.pinv(X.T.dot(X)).dot(X.T).dot(y)

    # Display beta
    beta

Out[9]: array([ 62.50261909,  4.6900952 , -72.28169508, -57.05464555])
The simplest algorithms for computing the inverse (or pseudo-inverse) of a $n \times n$ matrix is cubic in $n$ (although there are algorithms that are somewhere between quadratic and cubic). That makes this a feasible approach to least-squares regression when the number of features is no more than a few thousand.

We're going to look at another method, gradient descent, and we will compare them afterwards.