In [22]:
```
# we will need matplotlib for plotting
import matplotlib.pyplot as plt
import numpy as np

%matplotlib inline
```

Linear algebra: Review

Equation of a 2D line

There are two equations of lines

1. $y = mx + c$
2. $ax + by + c = 0$

What are the advantages or disadvantages of both of those?

A 2D line is the set of all points (x, y) that satisfy the an equation $ax + by + c = 0$ for given a, b, c. For the same line there can be multiple valid parameters a, b, c. Either a or b can be zero. But both a and b cannot be zero at the same time.

Ideally the equation of line must be written in the set notation:

$$\mathcal{L}(a, b, c) = \{(x, y) : ax + by + c = 0, x \in \mathbb{R}, y \in \mathbb{R}\}$$

This equation is read as: the line $\mathcal{L}$ defined by the paramters a, b, c is the set of all points (x, y) such that it the x, y satisfy the equation $ax + by + c = 0$ and x and y are in the set of all real numbers.

Implicit form

Line is a type of curve. Other curves can be parabola, circle etc. Every curve can be represented in implicit form like we did for the line above. In implicit form, the points are *constrained* by one or more equations to lie on the curve. The equations define a test whether the point lies on the curve or not.

$$\mathcal{C}(p_1, p_2, \ldots, p_n) = \{(x, y) : f(x, y; p_1, p_2, \ldots, p_n) = 0, x \in \mathbb{R}, y \in \mathbb{R}\}$$

This equation is read as the curve $\mathcal{C}$ defined by the paramters $p_1, p_2, \ldots, p_n$ is the set of all points (x, y) such that it the x, y satisfy the equation $f(x, y; p_1, p_2, \ldots, p_n) = 0$ and x and y are in the set of all real numbers.

Take circle with center (x_0, y_0) and radius r, as an example. The implicit form is:

Paramteric form of 2D Line

A parameteric form defines how the points on the curve are generated from a free paramters. For a line:

$$\mathcal{L}(d_x, d_y, x_0, y_0) = \{(d_x r + x_0, d_y r + y_0) : r \in \mathbb{R}\}$$

This equation is read as: the line $\mathcal{L}$ defined by the paramters d_x, d_y, x_0, y_0 is the set of all points $(d_x r + x_0, d_y r + y_0)$ such that r is any real number.

Take circle with center (x_0, y_0) and radius r, as an example. The parameteric form is:

$$\mathcal{C}(x_0, y_0, r) = \{(r\cos(\theta) + x_0, r\sin(\theta) + y_0) : \theta \in [0, 2\pi)\}$$

In general, the paramteric form of a curve depend on some free paramters and the points are defined as functions of the free parameters.

In [12]:
```
def stylizeax(ax, limits):
    """Set ax style"""
    minx, maxx, miny, maxy = limits
     # x-axis, y=0
    ax.annotate("",
                xy=(minx, 0),
                xytext=(maxx, 0),
                arrowprops=dict(arrowstyle="<->"),
                color='k')
    ax.text(maxx, 0, "x")
     # y-axis, x=0
    ax.annotate("",
                xy=(0, miny),
                xytext=(0, maxy),
                arrowprops=dict(arrowstyle="<->"),
                color='k')
    ax.text(0, maxy, "y")


    ax.grid(True, which='both') # show the grid
    ax.set_aspect('equal') # set aspect ratio of the grid to 1:1


def points_on_line(a, b, c, Npts=6, scale=10):
    """Generate points on the line ax + by + c = 0"""
    # ax + by + c = 0
    # In parameteric form with free parameter r
    #   (x, y) = (-b*r + x0, a*r + y0)
    # where
    #   x0 = - a*c / (a*a + b*b)
    #   y0 = - b*c / (a*a + b*b)
    # (x0, y0) is the point on the line closest to the origin
    uniformgrid = [i/Npts for i in range(-scale*Npts//2, scale*Npts//2,
    x0 = -a*c/(a*a + b*b)
    y0 = -b*c/(a*a + b*b)
    x = [-b*r + x0 for r in uniformgrid]
    y = [ a*r + y0 for r in uniformgrid]
    return x, y
```

Matplotlib

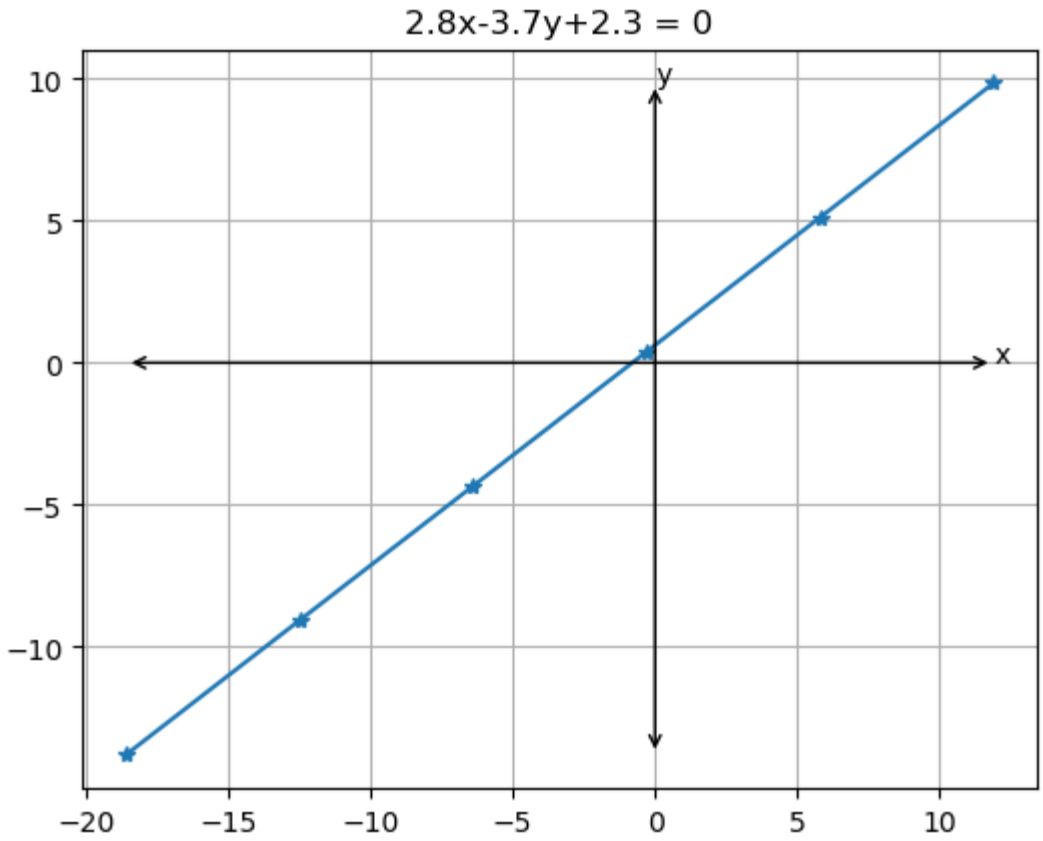
In [13]:
```
# Plot a line ax + by + c = 0
# a, b, c = 2.5, -1, -5 # pick numbers by hand

# pick a, b, c at random
import random
scale = 10
a, b, c = [scale*(random.random()-0.5) for _ in range(3)] # random numb

# Generate some sample points on a line
x, y = points_on_line(a, b, c, scale=scale)

# Plot the points
fig, ax = plt.subplots()
stylizeax(ax, (min(x), max(x), min(y), max(y)))
ax.plot(x, y, '*-') # the line
ax.set_title(f'{a:.1f}x{b:+.1f}y{c:+.1f} = 0') # print the equation
```

Out[13]:
```
Text(0.5, 1.0, '2.8x-3.7y+2.3 = 0')
```



Vectors

We will denote vectors with bold font notations instead of the usualy $\vec{x}$ notation. The arrow notation vectors are sometimes called geometric vectors. We will make no such distinction. The set of all real numbers will be denoted as $\mathbb{R}$. The set of all real 2D vectors is written as $\mathbb{R}^2$. When we write $\mathbf{x} \in \mathbb{R}^2$, it means that $\mathbf{x}$ is in the set of real 2D vectors, hence a 2D real vector. We will write $\|\mathbf{x}\|$ for the magnitude of the vector, and $\mathbf{x} \cdot \mathbf{y}$ for dot product between two vector

$\mathbf{x}$ and $\mathbf{y}$.

n-D vector

A n-D vector is also written as $\mathbf{x} \in \mathbb{R}^n$ and the vector has $\mathbf{x} = [x_1; \ldots; x_n]$ n real components. Every vector has a magnitude $\|\mathbf{x}\|$ and a direction $\hat{\mathbf{x}}$. The magnitude and direction are given by:

$$\|\mathbf{x}\| = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^n} \in \mathbb{R}$$
$$\hat{\mathbf{x}} = \frac{1}{\|\mathbf{x}\|}\mathbf{x} \in \mathbb{R}^n$$

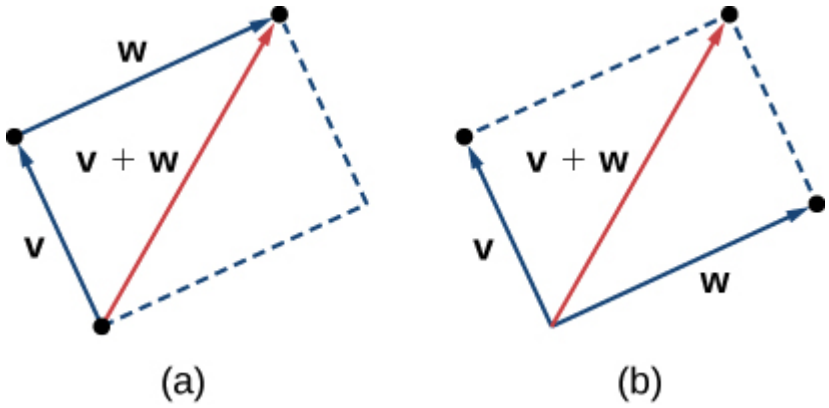
The direction vector $\hat{\mathbf{x}}$ is a unit vector because its magnitide is one i.e. $\|\hat{\mathbf{x}}\| = 1$.

Vector addition

Vector addition is element-wise addition

$$\mathbf{v} + \mathbf{w} = \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix} + \begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix} = \begin{bmatrix} v_1 + w_1 \\ \vdots \\ v_n + w_n \end{bmatrix}$$

Geometrically the resulting vector can be obtained by triangle law or the parallelogram law.



Reference: [1 (https://openstax.org/books/calculus-volume-3/pages/2-1-vectors-in-the-plane)]

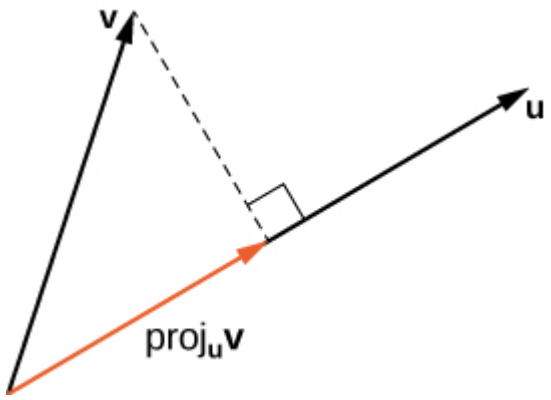
Dot product of vectors

Dot product of two vectors is a scalar given by sum of element-wise product.

$$\mathbf{v} \cdot \mathbf{u} = \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix} \cdot \begin{bmatrix} u_1 \\ \vdots \\ u_n \end{bmatrix} = v_1 u_1 + v_2 u_2 + \cdots + v_n u_n$$

Geometrically, dot product is closely related to the projection. Projection of vector $\mathbf{v}$ on $\mathbf{u}$ is the dot product of $\mathbf{v}$ with the direction of $\mathbf{u}$

$$\text{proj}_{\mathbf{u}}\mathbf{v} = \mathbf{v} \cdot \hat{\mathbf{u}}$$



Dot product of vector with itself gives the square of the magnitude $\mathbf{v} \cdot \mathbf{v} = ||\mathbf{v}||^2$.

Reference: [2 (https://openstax.org/books/calculus-volume-3/pages/2-3-the-dot-product)]

Matrices

Matrices are a group of vectors. A matrix can be obtained by vertical stacking of row (horizontal) vectors or horizontal stacking of column (vertical) vectors. It is common to represent all vectors as column vectors unless specified otherwise, so we consider a matrix $\mathbf{V}$ as a horizontal concatenation of column vectors $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$. Let each vector be m-dimensional $\mathbf{v}_i \in \mathbb{R}^m$.

$$\mathbf{V} = \begin{bmatrix} \mathbf{v}_1 & \mathbf{v}_2 & \ldots & \mathbf{v}_n \end{bmatrix} = \begin{bmatrix} \mathbf{v}_1[1] & \mathbf{v}_2[1] & \ldots & \mathbf{v}_n[1] \\ \mathbf{v}_1[2] & \mathbf{v}_2[2] & \ldots & \mathbf{v}_n[2] \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{v}_1[m] & \mathbf{v}_2[m] & \ldots & \mathbf{v}_n[m] \end{bmatrix}$$

Such a matrix is said to be a $m \times n$ matrix. It is also written as $\mathbf{V} \in \mathbb{R}^{m \times n}$.

Transpose of a Matrix

Transpose of a matrix $\mathbf{V}$ (denoted as $\mathbf{V}^\top$) is an operation that swaps rows with columns and columns with rows. For example, the transpose of the above matrix will make it a vertical concatenation of row vectors.

$$\mathbf{V}^\top = \begin{bmatrix} \mathbf{v}_1^\top \\ \mathbf{v}_2^\top \\ \vdots \\ \mathbf{v}_n^\top \end{bmatrix} = \begin{bmatrix} \mathbf{v}_1[1] & \mathbf{v}_1[2] & \ldots & \mathbf{v}_n[m] \\ \mathbf{v}_2[1] & \mathbf{v}_2[2] & \ldots & \mathbf{v}_n[m] \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{v}_n[1] & \mathbf{v}_2[2] & \ldots & \mathbf{v}_n[m] \end{bmatrix}$$

If $\mathbf{V} \in \mathbb{R}^{m \times n}$, then $\mathbf{V}^\top \in \mathbb{R}^{n \times m}$. The two dimensions get swapped.

Tranpose of a column vector

All vectors are also matrices. By convention, all vectors are considered column matrices and hence called column vectors. A n-D vector $\mathbf{v} = [v_1; \ldots; v_n] \in \mathbb{R}^n$ is by convention considered a $n \times 1$ column matrix i.e. $\mathbf{v} \in \mathbb{R}^{n \times 1}$.

$$\mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} \in \mathbb{R}^{n \times 1}$$

The transpose of a column vector is a row vector

$$\mathbf{v}^\top = \begin{bmatrix} v_1 & v_2 & \ldots & v_n \end{bmatrix} \in \mathbb{R}^{1 \times n}$$

Row vectors are always denoted with a tranpose of their corresponding column vector.

Matrix-vector product

For those who know dot product, matrix-vector product is best defined as a collection of dot products. Define a matrix $\mathbf{V} \in \mathbb{R}^{m \times n}$ as a vertical concatenation of m n-dimensional row-vectors $\mathbf{v}_1^\top, \ldots \mathbf{v}_m$

$$\mathbf{V} = \begin{bmatrix} \mathbf{v}_1^\top \\ \mathbf{v}_2^\top \\ \vdots \\ \mathbf{v}_m^\top \end{bmatrix}$$

The matrix $\mathbf{V} \in \mathbb{R}^{m \times n}$ can be multiplied by a n-dimensional column vector $\mathbf{u} \in \mathbb{R}^n$ with the product defined as the vector-wise dot product vertically concatenated to result in another column vector:

$$\mathbf{V}\mathbf{u} = \begin{bmatrix} \mathbf{v}_1 \cdot \mathbf{u} \\ \mathbf{v}_2 \cdot \mathbf{u} \\ \vdots \\ \mathbf{v}_m \cdot \mathbf{u} \end{bmatrix} \in \mathbb{R}^m$$

When $m = 1$, then $\mathbf{V} = \mathbf{v}_1^\top$ and the matrix product is $\mathbf{v}_1^\top \mathbf{u} = \mathbf{v}_1 \cdot \mathbf{u}$. Dot product between two vectors $\mathbf{v}$ and $\mathbf{u}$ is also written as $\mathbf{v}^\top \mathbf{u}$. Going forward we will prefer $\mathbf{v}^\top \mathbf{u}$ notation for dot product instead of $\mathbf{v} \cdot \mathbf{u}$.

Matrix-matrix product

Matrix-matrix product between two matrices $\mathbf{V} \in \mathbb{R}^{m \times n}$ and $\mathbf{U} \in \mathbb{R}^{n \times p}$ can be defined in terms of matrix-vector product by writing $\mathbf{U}$ as a horizontal concatenation of p n-dimensional column vectors $\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_p$.

$$\begin{aligned}\mathbf{VU} &= \begin{bmatrix} \mathbf{V}\mathbf{u}_1 & \mathbf{V}\mathbf{u}_2 & \ldots & \mathbf{V}\mathbf{u}_p \end{bmatrix} \in \mathbb{R}^{m \times p} \\ &= \begin{bmatrix} \mathbf{v}_1^\top \mathbf{u}_1 & \mathbf{v}_1^\top \mathbf{u}_2 & \ldots & \mathbf{v}_1^\top \mathbf{u}_p \\ \mathbf{v}_2^\top \mathbf{u}_1 & \mathbf{v}_2^\top \mathbf{u}_2 & \ldots & \mathbf{v}_2^\top \mathbf{u}_p \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{v}_m^\top \mathbf{u}_1 & \mathbf{v}_m^\top \mathbf{u}_2 & \ldots & \mathbf{v}_m^\top \mathbf{u}_p \end{bmatrix}\end{aligned}$$

The result is a horizontal concatenation of matrix-vector products, where the left matrix $\mathbf{V}$ gets multiplied with each column vector of right matrix $\mathbf{U}$.

Another interpretation is that the matrix-matrix product are all possible dot products between left matrices' row vectors with right matrices' column vectors.

Matrix-matrix product or short matrix products do not commute i.e $\mathbf{VU} \neq \mathbf{UV}$ in general.

Identity matrix

$$\mathbf{I}_n = \begin{bmatrix} 1 & 0 & \ldots & 0 \\ 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & 1 \end{bmatrix}$$

Square matrix

A square matrix is a matrix with number of rows equal to the number of columns.

Inverse of a square matrix

A matrix $\mathbf{V}^{-1}$ is called the inverse of a square matrix $\mathbf{V}$ if $\mathbf{V}^{-1}\mathbf{V} = \mathbf{V}^{-1} = \mathbf{I}_n$. The inverse of a square matrix exists only when it is singular i.e the determinant of the matrix is non-zero $\det(\mathbf{V}) \neq 0$.

Using vectors for 2D line notation

We started from 2D linear models, but we want to work with N-D models where N can be even in thousands or millions. It makes sense to simplify the notation by using vector notation.

Recall that the implicit equation for a line is

$$\mathcal{L}(a, b, c) = \{(x, y) : ax + by + c = 0, x \in \mathbb{R}, y \in \mathbb{R}\}$$

We will represent a 2D point (x, y) by a 2D vector $\mathbf{x} = [x; y]$ and the parameters (a, b) with weight vector $\mathbf{w} = [a; b]$. Let's compute the dot product between the two newly defined vectors :

$$\mathbf{w} \cdot \mathbf{x} = ax + by$$

The equation of the line under new notation in full its full glory is

$$\mathcal{L}(\mathbf{w}, c) = \{\mathbf{x} : \mathbf{w} \cdot \mathbf{x} + c = 0, \mathbf{x} \in \mathbb{R}^2\}$$

Unique line notation

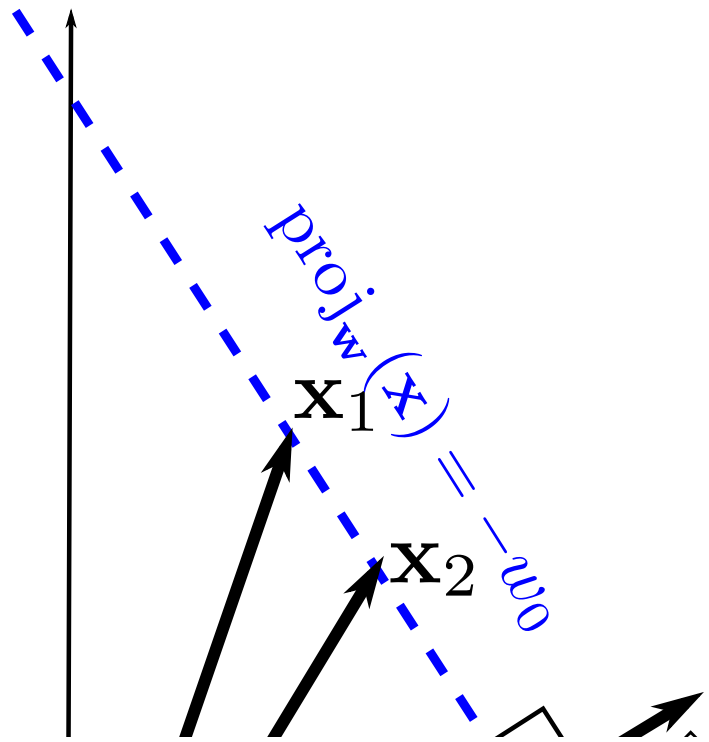
Same line $ax + by + c = 0$ can be represented by multiple equations for the same form. This representation of line is not unique. For example, equations $5x + 2y + 10 = 0$ and $10x + 4y + 20 = 0$ represent the same line. In general, for any real number $\alpha \neq 0$ all equations $\alpha ax + \alpha by + \alpha c = 0$ represent the same line. Once can choose an arbitrary non-zero α for making the equation unique.

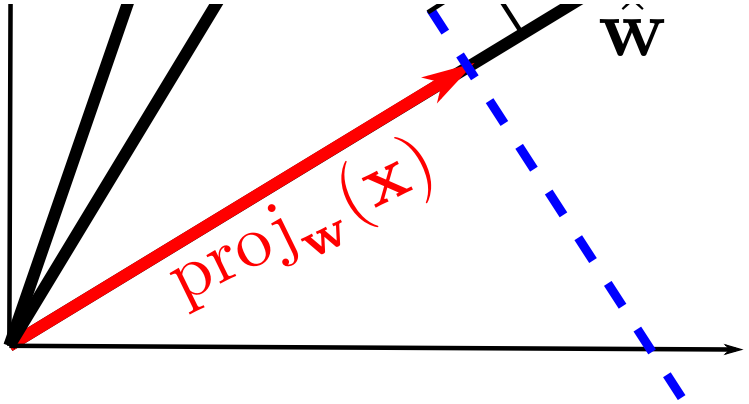
In vector notation, all non-zero $\alpha\mathbf{w} \cdot \mathbf{x} + \alpha c = 0$ represent the same line. One good candidate for α is $\alpha = \frac{1}{\|\mathbf{w}\|}$, because this changes $\alpha\mathbf{w}$ to $\hat{\mathbf{w}}$ a unit vector.

$$\mathcal{L}(\hat{\mathbf{w}}, w_0) = \{\mathbf{x} : \hat{\mathbf{w}} \cdot \mathbf{x} + w_0 = 0, \mathbf{x} \in \mathbb{R}^2\}$$

where $w_0 = \frac{c}{\|\mathbf{w}\|}$.

Geometric interpretaion





The new equation of the line has a convinient geometric interpretation. Recall that $\hat{\mathbf{w}} \cdot \mathbf{x}$ is the projection of $\mathbf{x}$ on $\hat{\mathbf{w}}$. In other words, the equation $\text{proj}_{\hat{\mathbf{w}}}\mathbf{x} + w_0 = 0$ constrains all vectors on the line to have a constant projection $\text{proj}_{\hat{\mathbf{w}}}\mathbf{x} = -w_0$.

This means that vector $\hat{\mathbf{w}}$ is perpendicular to the line and cuts the line a distance $|w_0|$ from the origin.

In [14]:
```
import numpy as np # a vector algebra library

a = np.array([0, 1, 2, 3]) # a vector
print("a=", a)
b = np.array([4, 5, 6, 7]) # another vector
print("b=", b)
C = np.array([[0, 1, 2, 3],
              [4, 5, 6, 7]]) # A matrix
print("C=", C)
D = np.zeros((2, 4)) # a 2x4 matrix of zeros
print("D=", D)
E = np.random.rand(2,5) # Random 2x5 matrix of numbers between 0 and 1
print("E=", E)
```

```
a= [0 1 2 3]
b= [4 5 6 7]
C= [[0 1 2 3]
 [4 5 6 7]]
D= [[0. 0. 0. 0.]
 [0. 0. 0. 0.]]
E= [[0.24267745 0.34908614 0.31547851 0.15059988 0.17537179]
 [0.60868919 0.31716426 0.10530595 0.53841394 0.49799488]]
```

In [15]:
```
print("a*0.1 = ", a * 0.1) # element-wise multiplication
print("C*0.2 = ", C * 0.2) # element-wise multiplication
print("a*b = ", a * b)   # element-wise multiplication (Note: different
print("a*b*0.2 = ", a * b * 0.2)   # element-wise multiplication
print("C @ a = ", C @ a)   # matrix-vector product
print("C.T = ", C.T)     # matrix transpose
print("C.T @ D = ", C.T @ D) # matrix-matrix product
print("a * C = ", a * C)   # so called broadcasting; numpy specific
```

```
a*0.1 =  [0.  0.1 0.2 0.3]
C*0.2 =  [[0.  0.2 0.4 0.6]
 [0.8 1.  1.2 1.4]]
a*b =  [ 0  5 12 21]
a*b*0.2 =  [0.  1.  2.4 4.2]
C @ a =  [14 38]
C.T =  [[0 4]
 [1 5]
 [2 6]
 [3 7]]
C.T @ D =  [[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
a * C =  [[ 0  1  4  9]
 [ 0  5 12 21]]
```

Numpy: General Broadcasting Rules

When operating on two arrays, NumPy compares their shapes element-wise. It starts with the trailing (i.e. rightmost) dimension and works its way left. Two dimensions are compatible when

1. they are equal, or
2. one of them is 1.

Otherwise a ValueError is raised

Ref: https://numpy.org/doc/stable/user/basics.broadcasting.html (https://numpy.org/doc/stable/user/basics.broadcasting.html)

In the following example, both the A and B arrays have axes with length one that are expanded to a larger size during the broadcast operation:

```
A      (4d array):  8 x 1 x 6 x 1
B      (3d array):      7 x 1 x 5
Result (4d array):  8 x 7 x 6 x 5
```

In [16]:
```
A = np.random.rand(8, 1, 6, 1)
B = np.random.rand(7, 1, 5)
(A * B).shape # Returns the shape of the multi dimensional array
```

Out[16]:
```
(8, 7, 6, 5)
```

Here are some more examples:

```
A      (2d array):  5 x 4
B      (1d array):      1
Result (2d array):  ?

A      (2d array):  5 x 4
B      (1d array):      4
Result (2d array):  ?

A      (3d array):  15 x 3 x 5
B      (3d array):  15 x 1 x 5
Result (3d array):

A      (3d array):  15 x 3 x 5
B      (2d array):       3 x 5
Result (3d array):  ?

A      (3d array):  15 x 3 x 5
B      (2d array):       3 x 1
Result (3d array):  ?
```

In [17]:

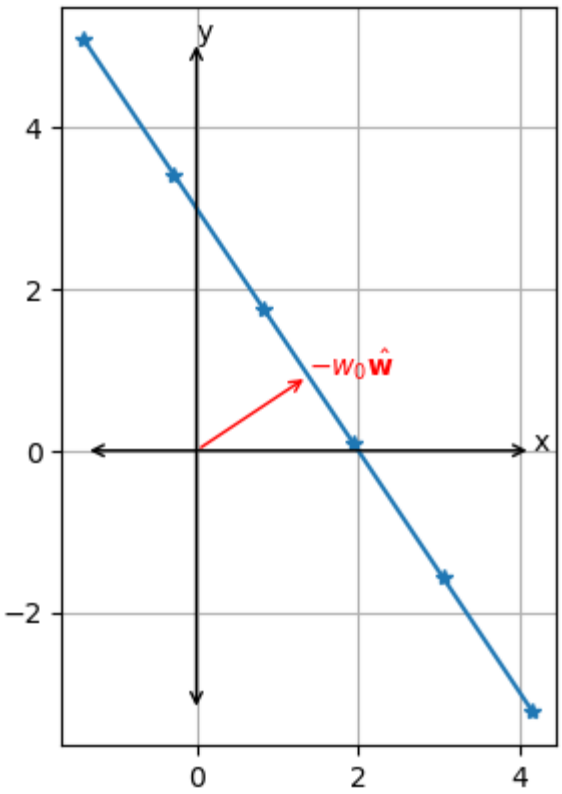
```
def points_on_line(hatw, w0, Npts=6, scale=10):
    """ Generate some sample points on a line """
    assert hatw.shape == (2,) # only works for 2D
    perp_hatw = np.array([-hatw[1], hatw[0]])# vector perpendicular to
    uniformgrid = np.linspace(-scale//2, scale//2, Npts)
    return perp_hatw * uniformgrid[:, None] - w0*hatw
```

In [18]:
```
# Plot a line ax + by + c = 0
scale = 10
# a, b, c = [scale*(random.random()-0.5) for _ in range(3)] # random nu
# abc = scale*(np.random.rand(3)-0.5)  # random number from -10 to 10
abc = [3, 2, -6] # pick your favorite line
w = abc[:2]
hatw = w / np.linalg.norm(w) # What does np.linalg.norm do?
w0 = abc[2] / np.linalg.norm(w)

# Generate some sample points on a line
x = points_on_line(hatw, w0, Npts=6, scale=scale) # Npts x 2 array

# Plot the points
fig, ax = plt.subplots()
stylizeax(ax, (x[:, 0].min(), x[:, 0].max(), x[:, 1].min(), x[:, 1].max
ax.plot(x[:, 0], x[:, 1], '*-') # the line
pt0 = -w0*hatw
ax.annotate("", xytext=(0, 0), xy=(pt0[0], pt0[1]),
            arrowprops=dict(arrowstyle="->", color='r'))
ax.text(pt0[0], pt0[1], r"$-w_0\hat{\mathbf{w}}$", color='r')
```

Out[18]:
```
Text(1.3846153846153848, 0.9230769230769231, '$-w_0\\hat{\\mathbf
{w}}$')
```



Linear regression: review

Let's take the simple linear regression example from STS332 textbook (uploaded on brightspace;page 300; Table 6-1).

"As an illustration, consider the data in Table 6-1. In this table, y is the salt concentration (milligrams/liter) found in surface streams in a particular watershed and x is the percentage of the watershed area consisting of paved roads."

In [19]:
```
%%writefile saltconcentration.tsv
#Observation    SaltConcentration   RoadwayArea
1   3.8 0.19
2   5.9 0.15
3   14.1    0.57
4   10.4    0.4
5   14.6    0.7
6   14.5    0.67
7   15.1    0.63
8   11.9    0.47
9   15.5    0.75
10  9.3 0.6
11  15.6    0.78
12  20.8    0.81
13  14.6    0.78
14  16.6    0.69
15  25.6    1.3
16  20.9    1.05
17  29.9    1.52
18  19.6    1.06
19  31.3    1.74
20  32.7    1.62
```

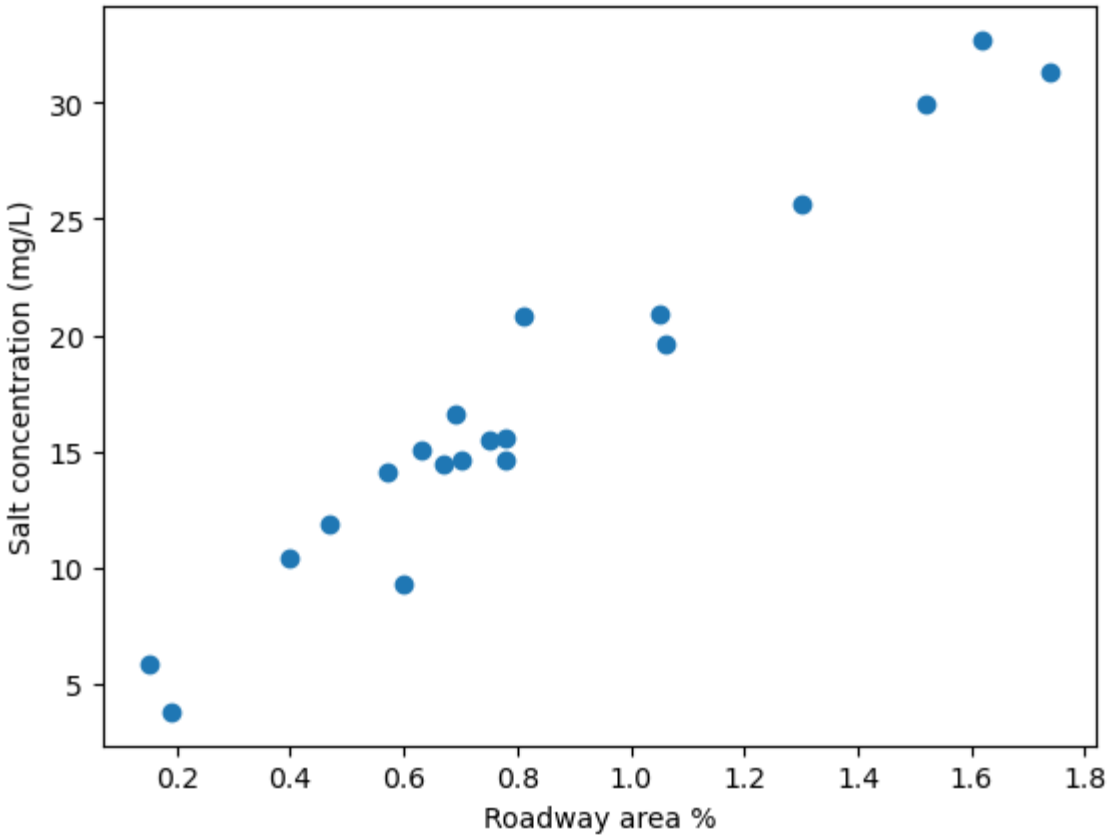
```
Writing saltconcentration.tsv
```

In [20]:
```
# numpy can import text files separated by seprator like tab or comma
salt_concentration_data = np.loadtxt("saltconcentration.tsv")
salt_concentration_data
```

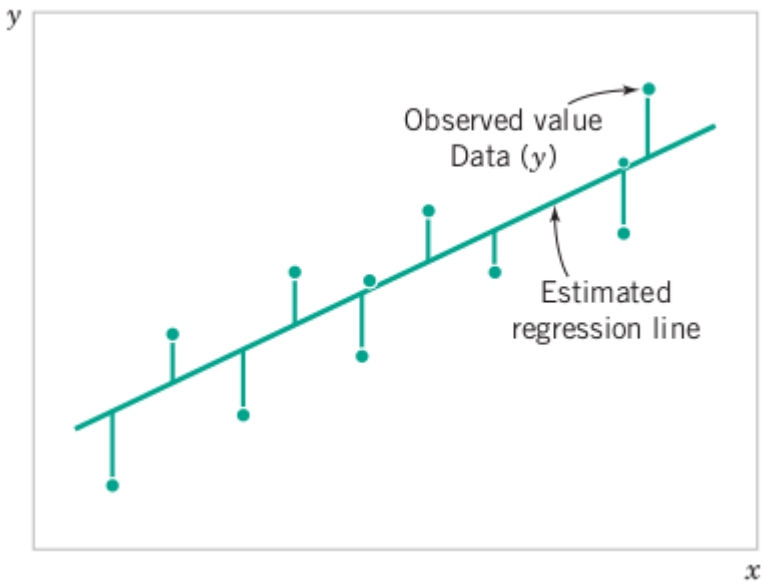
Out[20]:
```
array([[ 1.  ,  3.8 ,  0.19],
       [ 2.  ,  5.9 ,  0.15],
       [ 3.  , 14.1 ,  0.57],
       [ 4.  , 10.4 ,  0.4 ],
       [ 5.  , 14.6 ,  0.7 ],
       [ 6.  , 14.5 ,  0.67],
       [ 7.  , 15.1 ,  0.63],
       [ 8.  , 11.9 ,  0.47],
       [ 9.  , 15.5 ,  0.75],
       [10.  ,  9.3 ,  0.6 ],
       [11.  , 15.6 ,  0.78],
       [12.  , 20.8 ,  0.81],
       [13.  , 14.6 ,  0.78],
       [14.  , 16.6 ,  0.69],
       [15.  , 25.6 ,  1.3 ],
       [16.  , 20.9 ,  1.05],
       [17.  , 29.9 ,  1.52],
       [18.  , 19.6 ,  1.06],
       [19.  , 31.3 ,  1.74],
       [20.  , 32.7 ,  1.62]])
```

In [21]:
```
# Plot the points
fig, ax = plt.subplots()
# Scatter plot using matplotlib
ax.scatter(salt_concentration_data[:, 2], salt_concentration_data[:, 1]
ax.set_xlabel(r"Roadway area %")
ax.set_ylabel(r"Salt concentration (mg/L)")
```

Out[21]:
```
Text(0, 0.5, 'Salt concentration (mg/L)')
```



Least squares regression



The problem of linear regression is to find a line that "best fits" the given data. That is we want all the points $\{(x_1, y_1), \ldots, (x_n, y_n)\}$ to satisfy the equation of the line $y = mx + c$. Since we know that there exists no such line, so we will try to make $y \approxeq mx + c$, by minimizing some error/distance/cost/loss function between y and $mx + c$ for every point (x_i, y_i) in the dataset. The simplest error function that results in nice answers is squared distance:

$$e(x_i, y_i) = (y_i - (mx_i + c))^2$$

Then we can minimize the total error to find the line:

$$m^*, c^* = \arg \min_{m,c} \sum_{i=1}^{n} e(x_i, y_i)$$

Geometrically, this error minimization corresponds to minimizing the stubs in the following figure:

Vectorization of Least square regression

Recall that the magnitude of a vector $\|\mathbf{v}\| = \sqrt{v_1^2 + v_2^2 + \cdots + v_n^2}$ has a similar form to the error function. This suggests that we can define an error vector with the signed error for each data point as it's elements

$$\mathbf{e} = \begin{bmatrix} y_1 - (mx_1 + c) \\ y_2 - (mx_2 + c) \\ \vdots \\ y_n - (mx_n + c) \end{bmatrix}$$

Minimizing the total error is same as minimizing the square of error vector magnitude

$$m^*, c^* = \arg \min_{m,c} \|\mathbf{e}\|^2$$

While we are at at it let us define $\mathbf{x} = [x_1; \ldots; x_n]$ to denote the vector of all x coordinates of the dataset and $\mathbf{y} = [y_1; \ldots; y_n]$ to denote y coordinates. Then the error vector is:

$$\mathbf{e} = \mathbf{y} - (\mathbf{x}m + \mathbf{1}_n c)$$

where $\mathbf{1}_n$ is a n-D vector of all ones. Finally, we vectorize parameters of the line $\mathbf{m} = [m; c]$. We will also need to horizontally concatenate $\mathbf{x}$ and $\mathbf{1}_n$. Let's call the result $\mathbf{X} = [\mathbf{x}, \mathbf{1}_n] \in \mathbb{R}^{n \times 2}$. Now, the error vector looks like this:

$$\mathbf{e} = \mathbf{y} - \mathbf{X}\mathbf{m}$$

Expanding the error magnitude:

$$\|\mathbf{e}\|^2 = (\mathbf{y} - \mathbf{X}\mathbf{m})^\top (\mathbf{y} - \mathbf{X}\mathbf{m})$$
$$= \mathbf{y}^\top \mathbf{y} + \mathbf{m}^\top \mathbf{X}^\top \mathbf{X}\mathbf{m} - 2\mathbf{y}^\top \mathbf{X}\mathbf{m}$$

Our minimization problem in vectorized form is:

$$\mathbf{m}^* = \arg \min_{\mathbf{m}} \mathbf{y}^\top \mathbf{y} + \mathbf{m}^\top \mathbf{X}^\top \mathbf{X}\mathbf{m} - 2\mathbf{y}^\top \mathbf{X}\mathbf{m}$$

This is a quadratic equation in $\mathbf{m}$ that can be minimized by equating the derivate to zero.

Two rules of vector derivatives

There are two conventions in vector derivatives:

1. Gradient convention
2. Jacobian convention

Gradient convention

Under gradient convention the derivative of scalar-valued vector function function $f(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}$ is defined as vertical stacking of element-wise derivatives

$$\frac{\partial}{\partial \mathbf{x}} f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f(\mathbf{x})}{\partial x_1} \\ \vdots \\ \frac{\partial f(\mathbf{x})}{\partial x_n} \end{bmatrix} \in \mathbb{R}^n$$

Jacobian convention

Under gradient convention the derivative of scalar-valued vector function function $f(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}$ is defined as horizontal stacking of element-wise derivatives

$$\frac{\partial}{\partial \mathbf{x}} f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f(\mathbf{x})}{\partial x_1} & \dots & \frac{\partial f(\mathbf{x})}{\partial x_n} \end{bmatrix} \in \mathbb{R}^{1 \times n}$$

For a vector-value vector function $\mathbf{f}(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}^m$, Jacobian of $\mathbf{f}(\mathbf{x})$ is the vertical concatentation of gradients transposed, resulting in $m \times n$ matrix

$$\mathbf{J}_\mathbf{x}(\mathbf{f}(\mathbf{x})) = \frac{\partial}{\partial \mathbf{x}} \mathbf{f}(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_1(\mathbf{x})}{\partial \mathbf{x}} \\ \dots \\ \frac{\partial f_m(\mathbf{x})}{\partial \mathbf{x}} \end{bmatrix}$$

We will use Jacobian convention in this course, because it works nicely with chain rule.

Derivative of a linear function

All scalar-valued linear functions of $\mathbf{x}$ can be written in the form $f(\mathbf{x}) = \mathbf{c}^\top \mathbf{x}$.

$$\frac{\partial}{}$$

Derivative of a quadratic function

All scalar-valued homogeneous quadratic functions of $\mathbf{x}$ can be written in the form $f(\mathbf{x}) = \mathbf{x}^\top \mathbf{A}\mathbf{x}$.

$$\frac{\partial}{\partial \mathbf{x}} \mathbf{x}^\top \mathbf{A}\mathbf{x} = \mathbf{x}^\top (\mathbf{A} + \mathbf{A}^\top)$$

Back to Least square regression

$$\begin{aligned} \mathbf{0}^\top &= \frac{\partial}{\partial \mathbf{m}} (\mathbf{y}^\top \mathbf{y} + \mathbf{m}^\top \mathbf{X}^\top \mathbf{X}\mathbf{m} - 2\mathbf{y}^\top \mathbf{X}\mathbf{m}) \\ &= 2\mathbf{m}^{*\top} \mathbf{X}^\top \mathbf{X} - 2\mathbf{y}^\top \mathbf{X} \end{aligned}$$

This gives us the solution

$$\mathbf{m}^* = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}$$

The symbol $\mathbf{V}^{-1}$ is called inverse of matrix $\mathbf{V}$.

The term $(\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top$ is also called the pseudo-inverse of a matrix $\mathbf{X}$, denoted as $\mathbf{X}^\dagger$.

In [46]:
```
n = salt_concentration_data.shape[0]
bfx = salt_concentration_data[:, 2:3]
bfy = salt_concentration_data[:, 1:2]
bfX = np.hstack((bfx, np.ones((bfx.shape[0], 1))))
bfX
```

Out[46]:
```
array([[0.19, 1.  ],
       [0.15, 1.  ],
       [0.57, 1.  ],
       [0.4 , 1.  ],
       [0.7 , 1.  ],
       [0.67, 1.  ],
       [0.63, 1.  ],
       [0.47, 1.  ],
       [0.75, 1.  ],
       [0.6 , 1.  ],
       [0.78, 1.  ],
       [0.81, 1.  ],
       [0.78, 1.  ],
       [0.69, 1.  ],
       [1.3 , 1.  ],
       [1.05, 1.  ],
       [1.52, 1.  ],
       [1.06, 1.  ],
       [1.74, 1.  ],
       [1.62, 1.  ]])
```

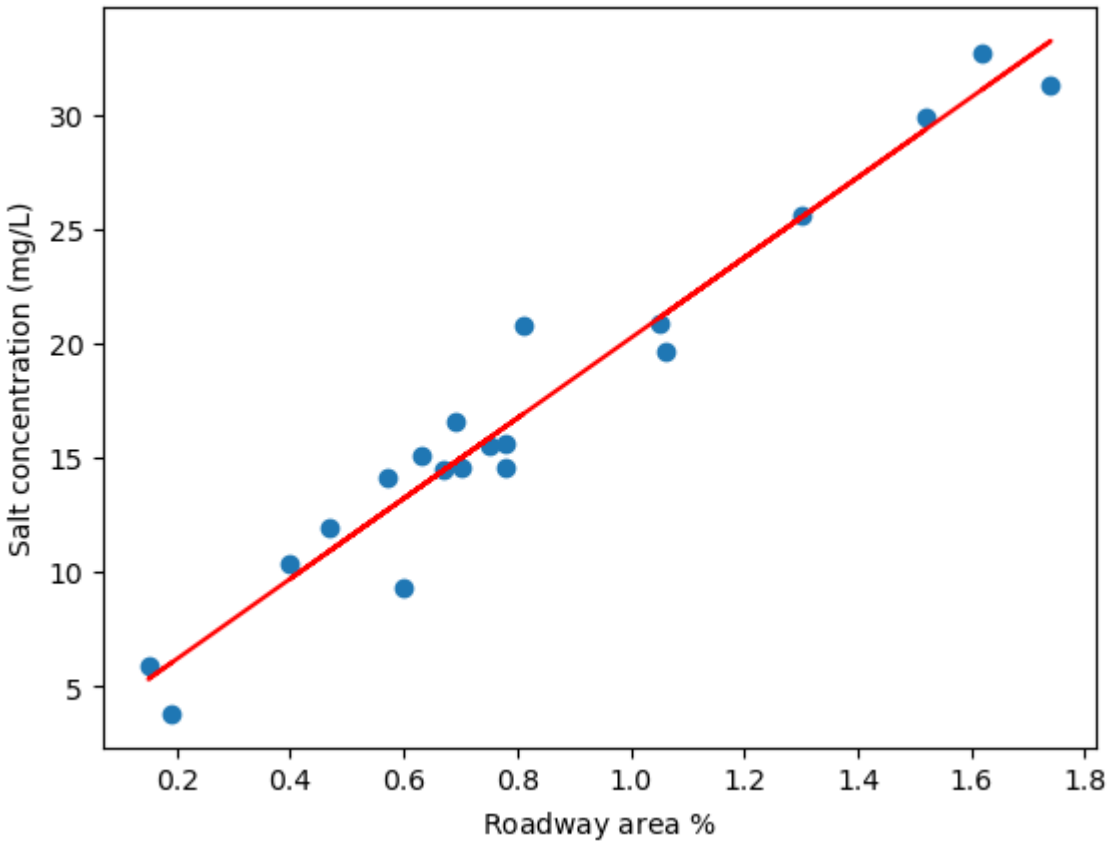
In [47]:
```
bfm = np.linalg.inv(bfX.T @ bfX) @ bfX.T @ bfy
print(bfm)
bfm, *_ = np.linalg.lstsq(bfX, bfy, rcond=None)
print(bfm)
```

```
[[17.5466671 ]
 [ 2.67654631]]
[[17.5466671 ]
 [ 2.67654631]]
```

In [48]:
```
m = bfm.flatten()[0]
c = bfm.flatten()[1]

# Plot the points
fig, ax = plt.subplots()
ax.scatter(salt_concentration_data[:, 2], salt_concentration_data[:, 1]
ax.set_xlabel(r"Roadway area $\%$")
ax.set_ylabel(r"Salt concentration (mg/L)")
x = salt_concentration_data[:, 2]
y = m * x + c
# Plot the points
ax.plot(x, y, 'r-') # the line
```

Out[48]: [<matplotlib.lines.Line2D at 0x7fbf437f67c0>]



Exercise 1

Derive the equations for least square linear regression when the equation of line is $\hat{\mathbf{w}}^\top \mathbf{x} + w_0 = 0$ instead of $y = mx + c$.

Hint: Convert the least square problem into equation of the form $\mathbf{v}^* = \arg \min_{\mathbf{v}} ||\mathbf{L}\mathbf{v}||^2$ such that $\mathbf{v}^\top \mathbf{v} = 1$. Solve by finding null space of $\mathbf{L}$. $\mathbf{v}$ lies in the nullspace of $\mathbf{L}$. The nullspace of $\mathbf{L}$ is the last eigenvector (corresponding to the smallest eigenvalue) of $\mathbf{L}^\top \mathbf{L}$.

The error $e(x_i, y_i) = (y - (mx + c))^2$ can be visualized as distance of observed point from the fit line parallel to y-axis. Draw the visual for the errors of the form: $e(\mathbf{x}_i) = (\hat{\mathbf{w}}^\top \mathbf{x}_i + w_0 - 0)^2$. You do not need to use matplotlib. You can draw by hand or