Linear algebra: Review

Equation of a 2D line

Implicit form

Paramteric form of 2D Line

Matplotlib

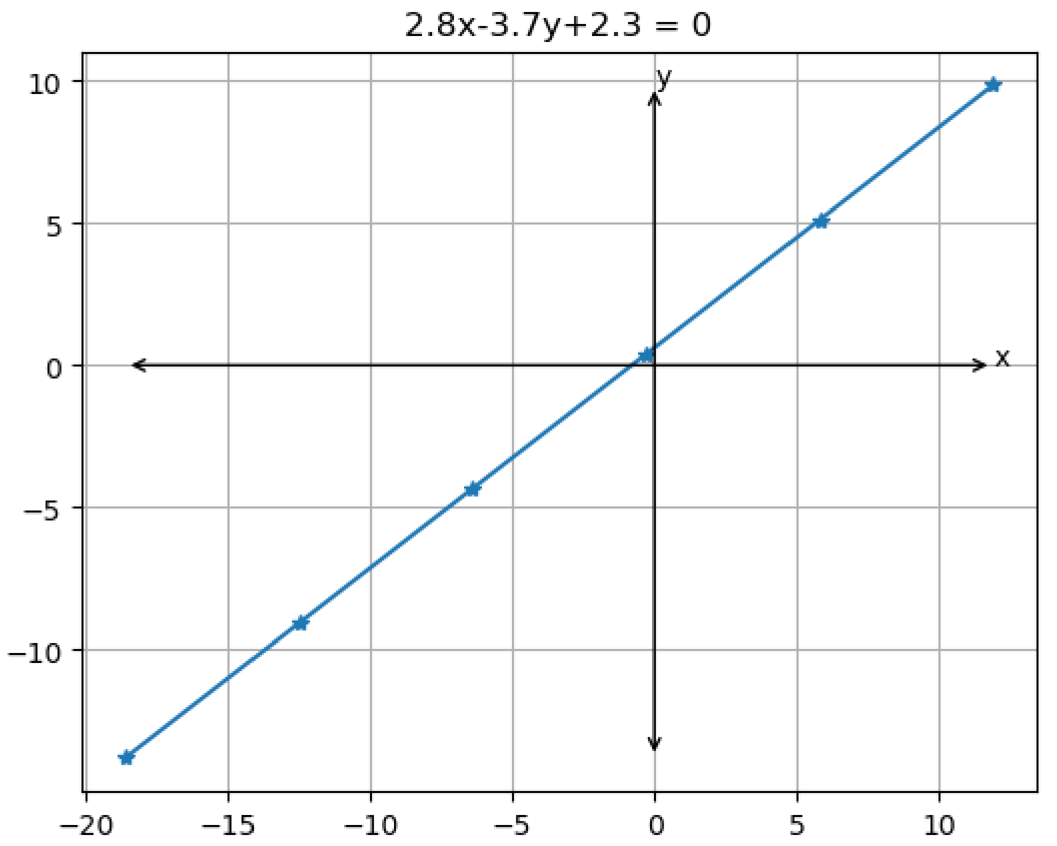
In [13]:
```
# Plot a line ax + by + c = 0
# a, b, c = 2.5, -1, -5 # pick numbers by hand

# 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')
```



2.8x-3.7y+2.3 = 0
y
x
10
5
0
−5
−10
−20
−15
−10
−5
0
5
10

Vectors

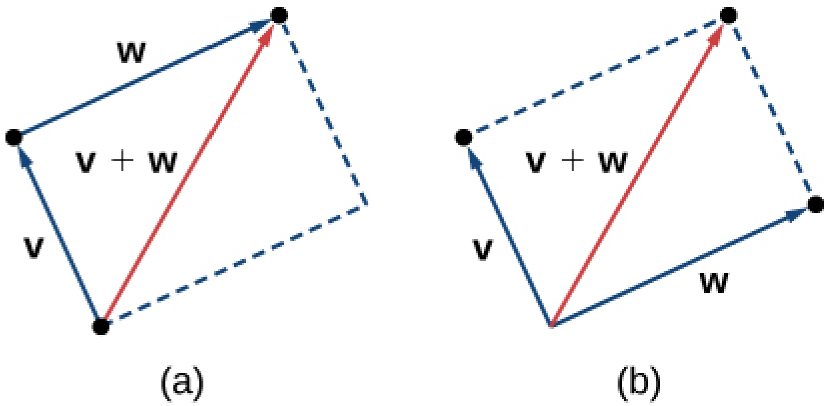
n-D vector

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]

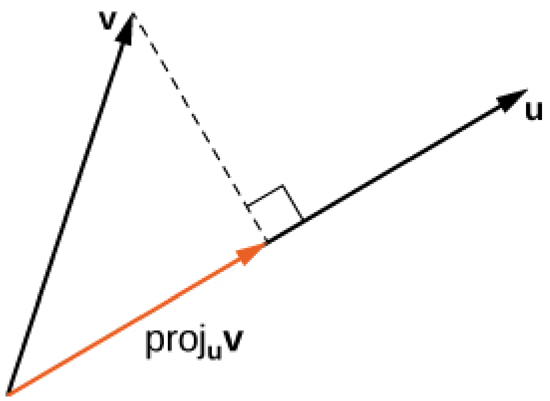
Dot product of vectors

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 $\mathbf{u}$

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



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

Reference: [2]

Matrices

Transpose of a Matrix

Tranpose of a column vector

Matrix-vector product

Matrix-matrix product

Identity matrix

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

Square matrix

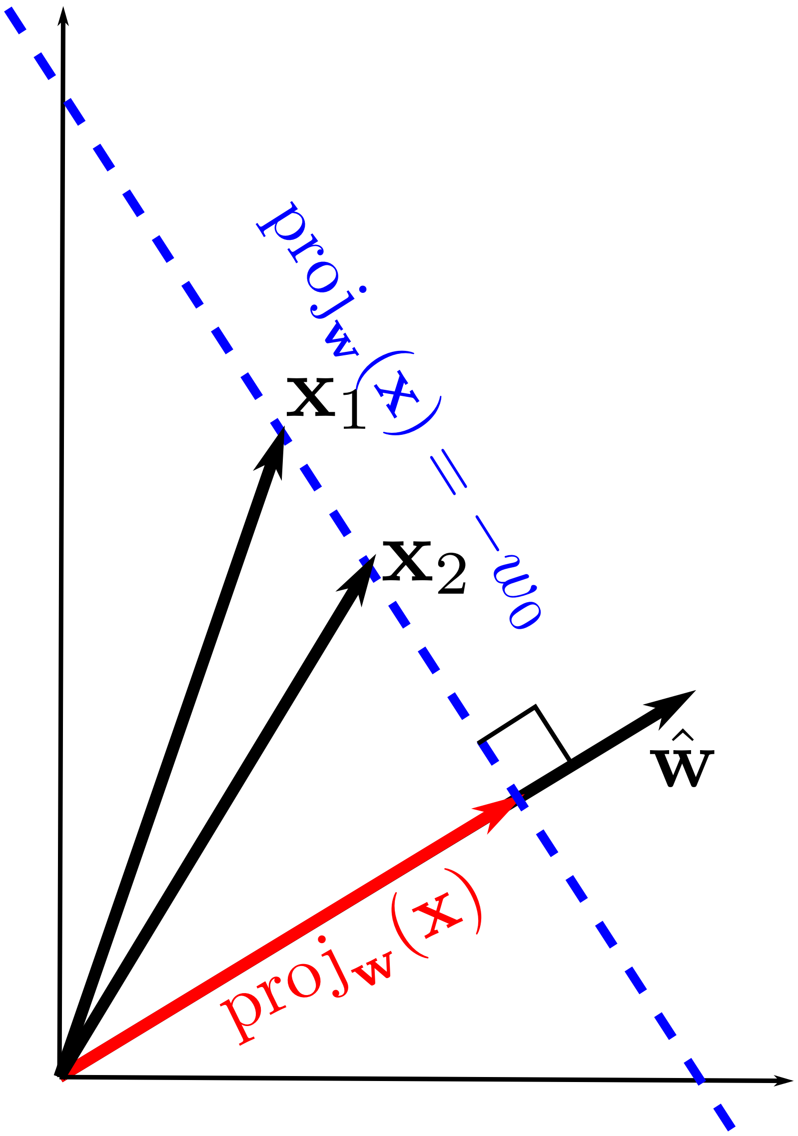
A square matrix is a matrix with number of rows equal to the number of columns.

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

Geometric interpretaion



In [14]:

```python
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

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):  ?
```

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

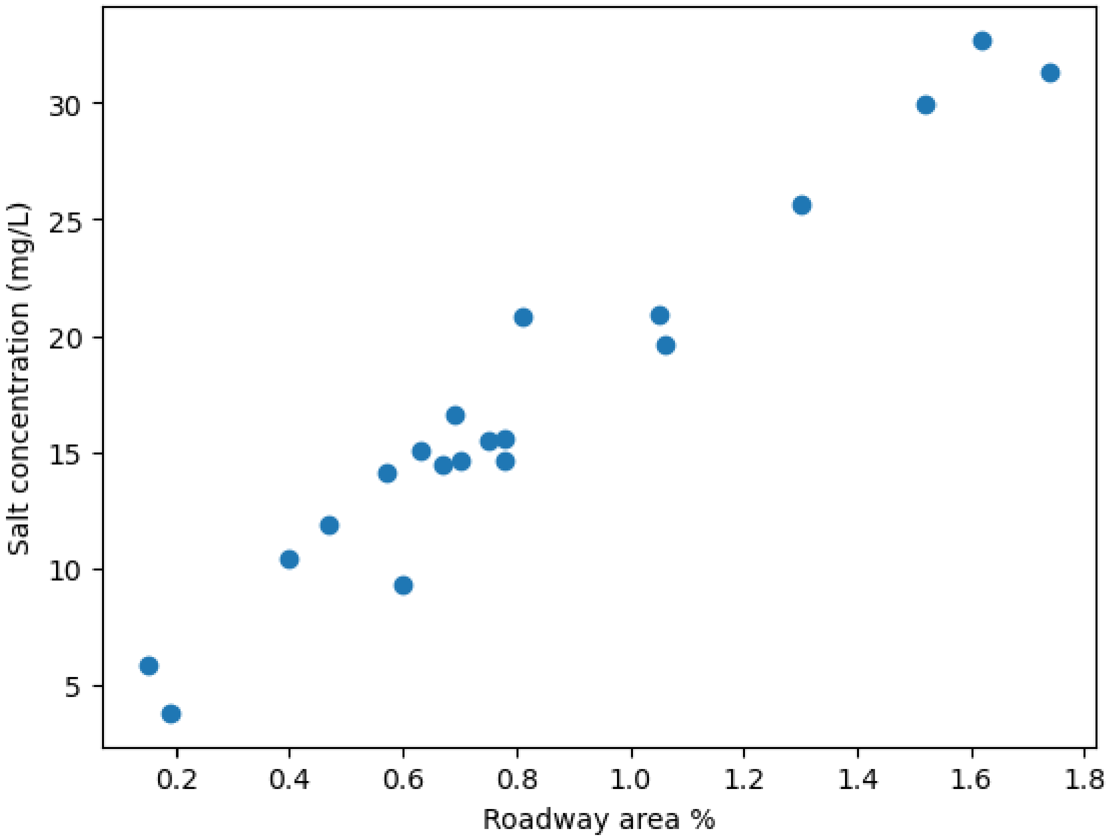
Out[20]:
```
array([[ 1.  ,  3.8 ,  0.19],
       [ 2.  ,  5.9 ,  0.15],
       [ 3.  , 14.1 ,  0.57],
       [ 4.  , 10.4 ,  0.4 ],
       [ 5.  , 14.6 ,  0.7 ],
       [ 6.  , 14.5 ,  0.67],
       [ 7.  , 15.1 ,  0.63],
       [ 8.  , 11.9 ,  0.47],
       [ 9.  , 15.5 ,  0.75],
       [10.  ,  9.3 ,  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]:

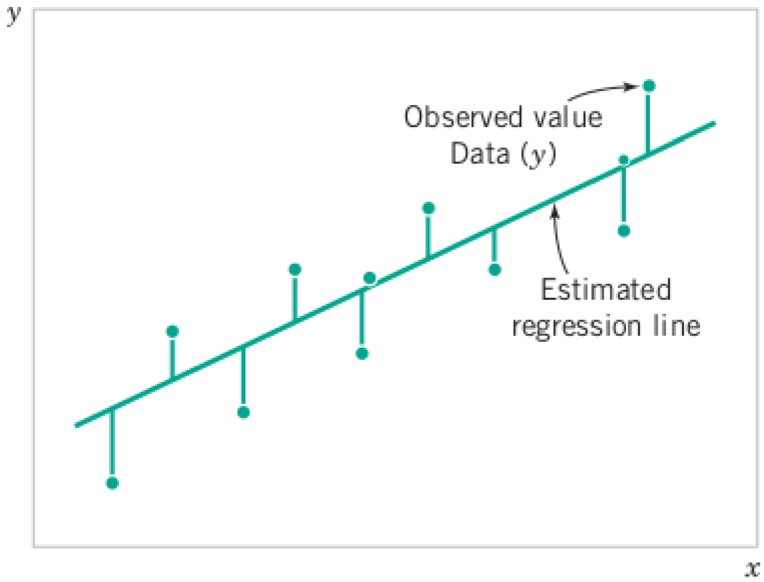
```
# Plot the points
fig, ax = plt.subplots()
# Scatter plot using matplotlib
ax.scatter(salt_concentration_data[:, 2], salt_concentration_data[:, 1]
ax.set_xlabel(r"Roadway area %")
ax.set_ylabel(r"Salt concentration (mg/L)")
```

Out[21]:

```
Text(0, 0.5, 'Salt concentration (mg/L)')
```



Least squares regression



Vectorization of Least square regression

Two rules of vector derivatives

There are two conventions in vector derivatives:

1. Gradient convention
2. Jacobian convention

Gradient convention

Jacobian convention

Derivative of a linear function

Derivative of a quadratic function

Back to Least square regression

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]]
```

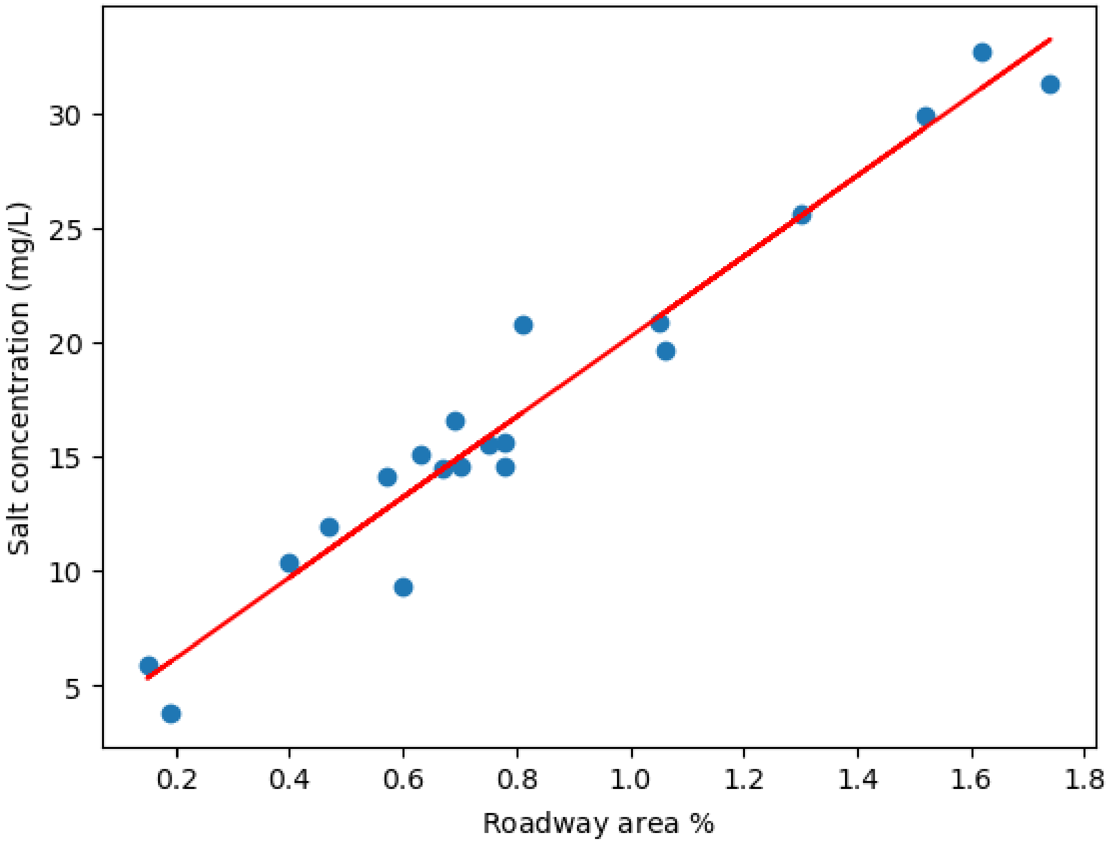
In [48]:

```
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>]
```



Salt concentration (mg/L)
30
25
20
15
10
5
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
1.8
Roadway area %

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 editing software.