Application of Regression and Pseudo-Inverse Method

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from numpy.linalg import inv
from copy import copy
from numpy.random import default_rng
```

To import and use required libraries for solving questions and implementing them

Numpy:
To use its functions and features working with matrixes.

Matplotlib:
To show and plot the input diagram and draw the driven model

In [2]:
```
def pol2cart(l, teta):

    x = l * np.cos(teta* np.pi /180)
    y = l * np.sin(teta* np.pi /180)

    return x, y
```

Using this function, we convert complex numbers into numbers with two dimensions.

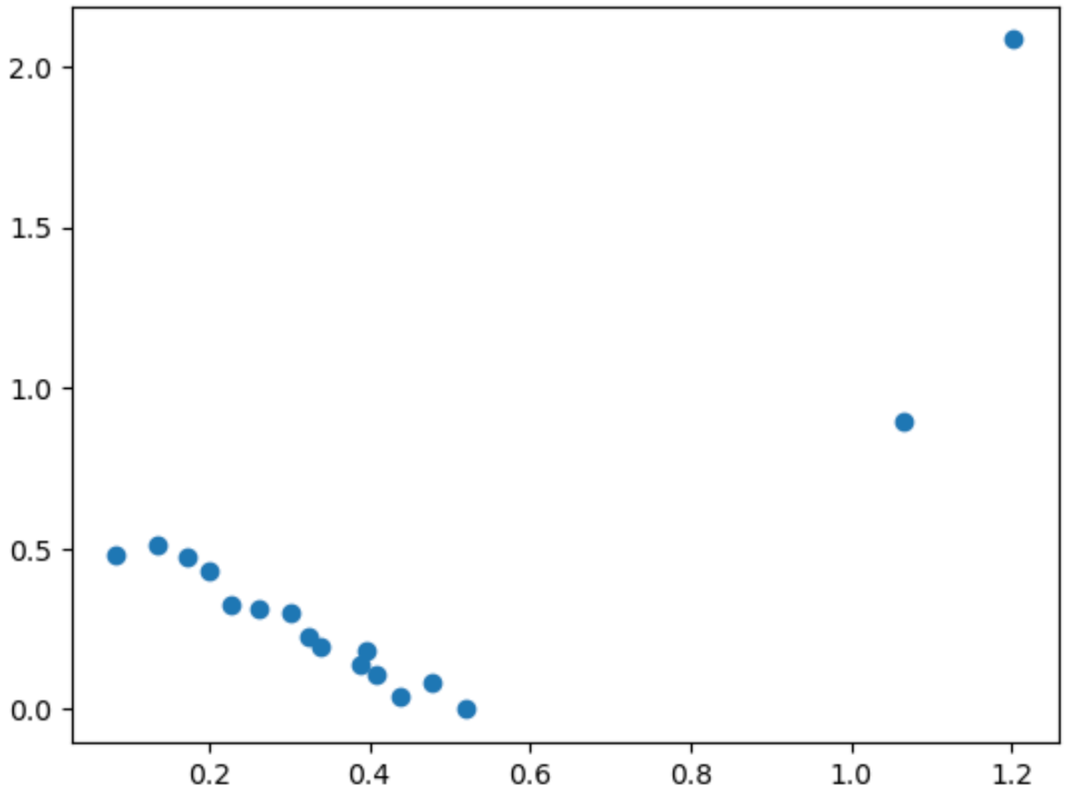
In [3]:
```
def all_convert(cart):
    x_array=np.array([])
    y_array=np.array([])
    for i in range(len(cart)):
        x,y=pol2cart(cart[i][1],cart[i][0])
        x_array=np.hstack((x_array,x))
        y_array=np.hstack((y_array,y))

    return x_array,y_array
```

This array receives the complex numbers, converts them into x-array and y-array, and then returns them as the output.

In [20]:
```
# zavie,fasele
carty=[[0,0.5197],[5,0.4404],[10,0.4850],[15,0.4222],[20,0.4132],
# print(all_convert(carty))
x,y = all_convert(carty)
```

In this part, the array that the robot has received by its sensors is transformed into two dimensions.

In [5]:
```
plt.scatter(x,y)
plt.show()
```



The numbers are plotted after being transformed into two dimensional coordinates.

In [6]:
```
x_train=x.reshape((17,1))
y_train=y.reshape((17,1))


bias=np.ones((17,1))
x_train=np.append(bias,x_train,axis=1)
```

The arrays x and y are transformed as 17 and 1; the bias is shown as a diminution that constantly equal one.

In [7]:
```
def update_weight(x_train,y_train):
    W=np.dot(np.dot(inv(np.dot(x_train.T,x_train)),x_train.T),y_train)
    return W
```

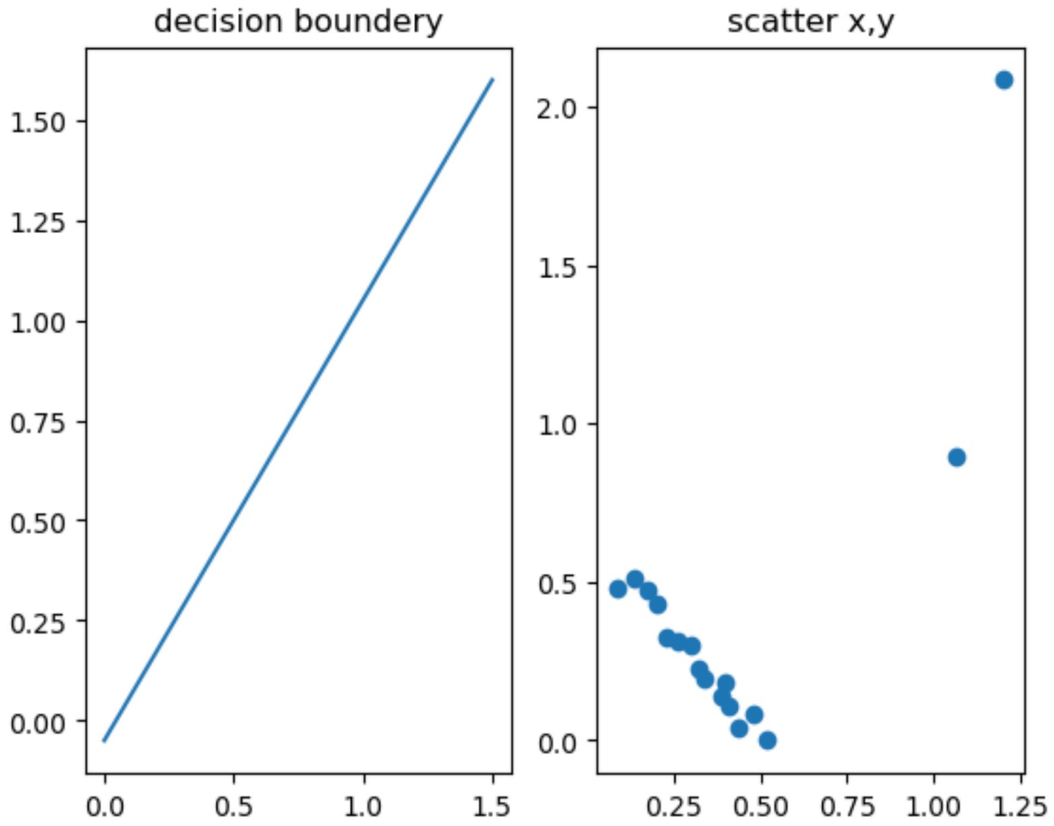
The weight matrix is updated based on pseudo-inverse law.

In [8]:
```
W = update_weight(x_train,y_train)
```

In [9]:
```
def forward_propagation(x,W):
    return  np.dot(W[1],x) + W[0]
```

this function calculates neuron's output according to the derived weight.

In [10]:
```
x_test = np.linspace(0, 1.5 , 100)
x_test=x_test.reshape((1,100))
y_trained=forward_propagation(x_test,W)
figure, axis = plt.subplots(1, 2)
axis[0].plot(x_test[0],y_trained )
axis[0].set_title("decision boundery")
axis[1].scatter(x,y )
axis[1].set_title("scatter x,y")
plt.show()
```



the obtained model by trained weights is drawn on the chart.

The analysis of this diagram is on the final slide.

In [11]:
```
def square_error_loss(y_true, y_pred):
    return (y_true - y_pred) ** 2
```

In [12]:
```
def mean_square_error(y_true, y_pred):
    return np.sum(square_error_loss(y_true, y_pred)) / y_true.shape[0]
```

In [13]:
```
print('MSE for sudo inverse :'+str(mean_square_error(y_train.reshape(17,1), forward_propagation(x.reshape(1,17),W).resh
```

```
MSE for sudo inverse :0.12057585388293435
```

The error obtained by this method is obtained by using the two above functions that measure the MSE error.

In [14]:

```
rng = default_rng()

class RANSAC:
    def __init__(self, n=10, k=1000, t=0.05, d=10, model=None, loss=None, metric=None):

        self.n = n
        self.k = k
        self.t = t
        self.d = d
        self.model = model
        self.loss = loss
        self.metric = metric
        self.best_fit = None
        self.best_error = np.inf

    def fit(self, X, y):

        for _ in range(self.k):
            ids = rng.permutation(X.shape[0])

            maybe_inliers = ids[: self.n]
            maybe_model = copy(self.model).fit(X[maybe_inliers], y[maybe_inliers])

            thresholded = (
                self.loss(y[ids][self.n :], maybe_model.predict(X[ids][self.n :]))
                < self.t
            )

            inlier_ids = ids[self.n :][np.flatnonzero(thresholded).flatten()]

            if inlier_ids.size > self.d:
                inlier_points = np.hstack([maybe_inliers, inlier_ids])
                better_model = copy(self.model).fit(X[inlier_points], y[inlier_points])

                this_error = self.metric(
                    y[inlier_points], better_model.predict(X[inlier_points])
                )

                if this_error < self.best_error:
                    self.best_error = this_error
                    self.best_fit = maybe_model

        return self

    def predict(self, X):
        return self.best_fit.predict(X)
```

RANSAC

Another method of obtaining a model that is suitable, can recognize the noise (outlier) data, and can ignore this data in training is the method of RANSAC whose codes are written in this class:

In [15]:
```
class LinearRegressor:
    def __init__(self):
        self.params = None

    def fit(self, X: np.ndarray, y: np.ndarray):
        r, _ = X.shape
        X = np.hstack([np.ones((r, 1)), X])
        self.params = np.linalg.inv(X.T @ X) @ X.T @ y
        return self

    def predict(self, X: np.ndarray):
        r, _ = X.shape
        X = np.hstack([np.ones((r, 1)), X])
        return X @ self.params
```

In [16]:
```
regressor = RANSAC(model=LinearRegressor(), loss=square_error_loss, metric=mean_square_error)
```

In [17]:
```
x_train=np.append(x,x).reshape(-1,1)
y_train=np.append(y,y).reshape(-1,1)
regressor.fit(x_train,y_train)
```

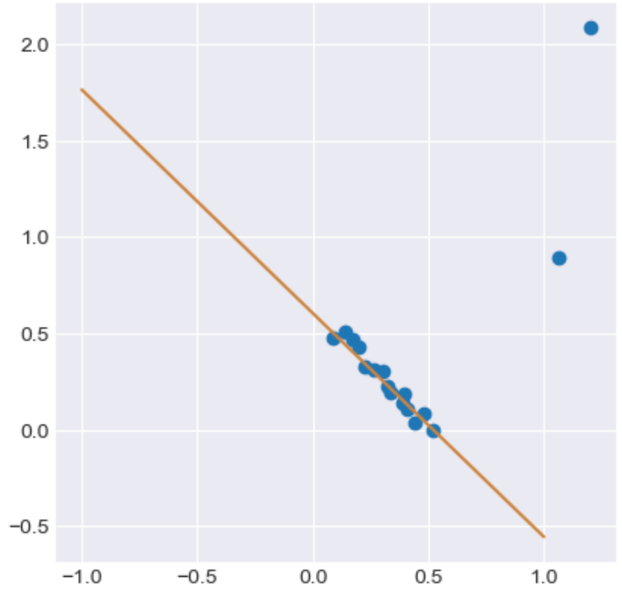
The model and the parameters (the weight matrix) is trained using RANSAC method.

The class that is shown (the Linear Regression class) finds the parameters of the model based on the “FIT” method and the quasi - inverse approach so that the line is best fitted on input data and ignores the outliers.

The model is being trained.

Then, a sample class whose model is linear and whose error function is MSE, like the class shown, should be written.

In [18]:
```
plt.style.use("seaborn-darkgrid")
fig, ax = plt.subplots(1, 1)
ax.set_box_aspect(1)

plt.scatter(x, y)

line = np.linspace(-1, 1, num=100).reshape(-1, 1)
plt.plot(line, regressor.predict(line), c="peru")
plt.show()
```



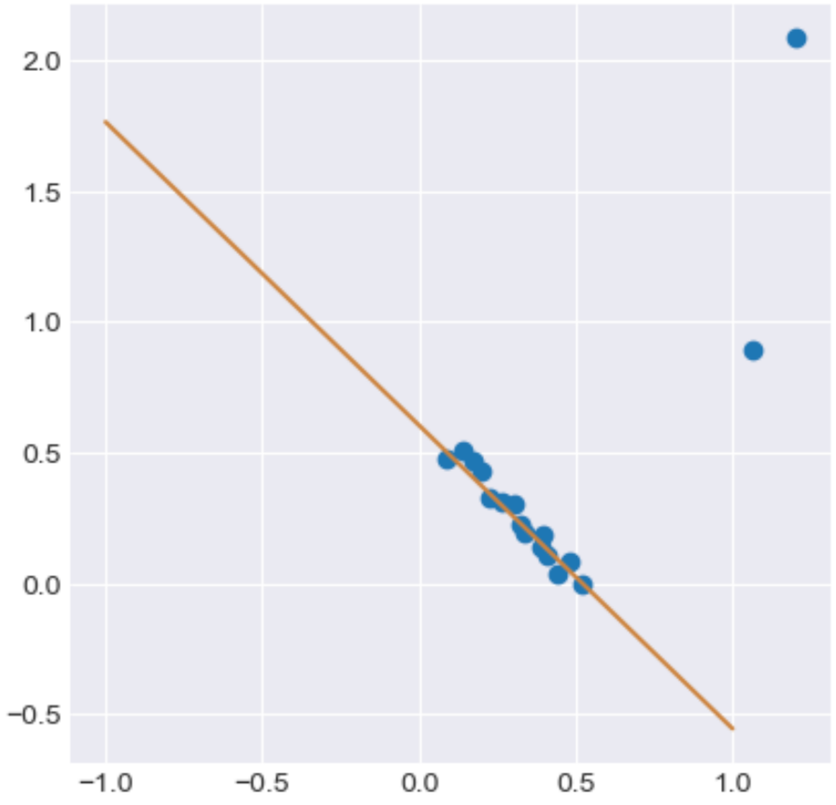
The driven line or model and the spots on the graph are shown on plot.

The analysis of this diagram comes in the next pages.

The MSE errors of this method are shown:

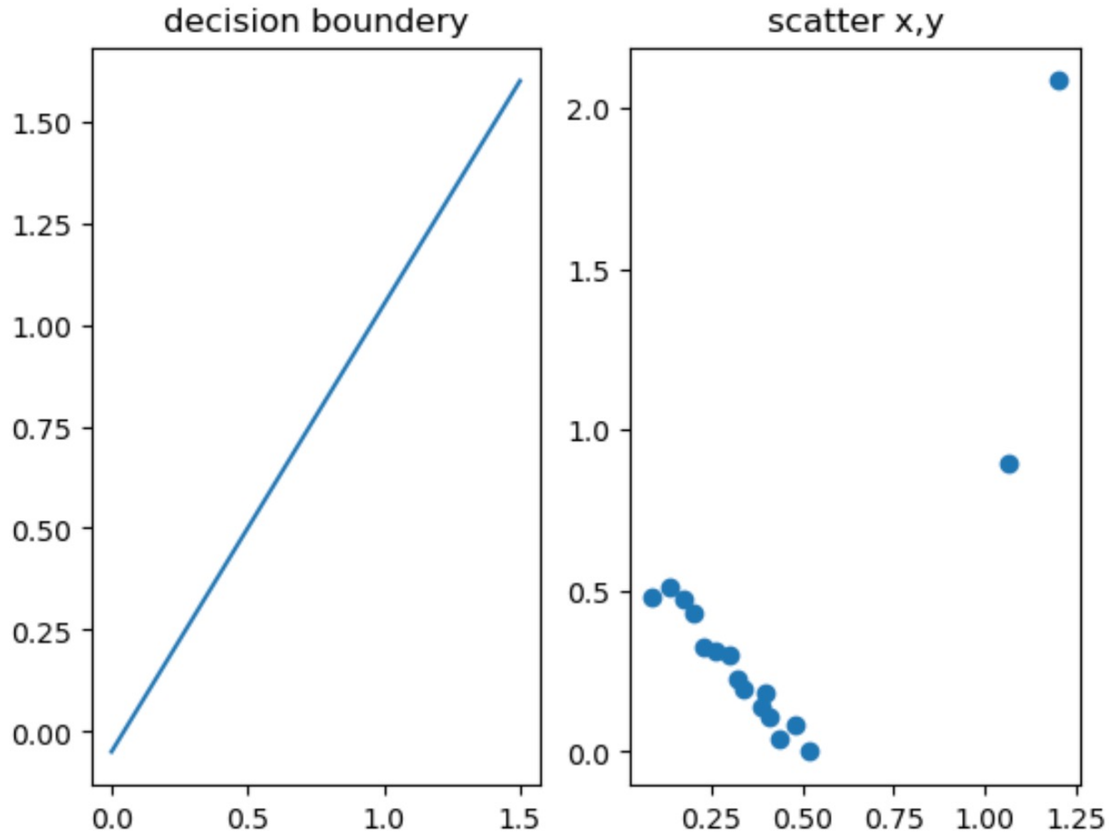
In [19]:
```
print('MSE for RANSAC :'+str(mean_square_error(y.reshape(17,1),regressor.predict(x.reshape(-1, 1)).reshape(17,1))))
```

```
MSE for RANSAC :0.6230943439018531
```

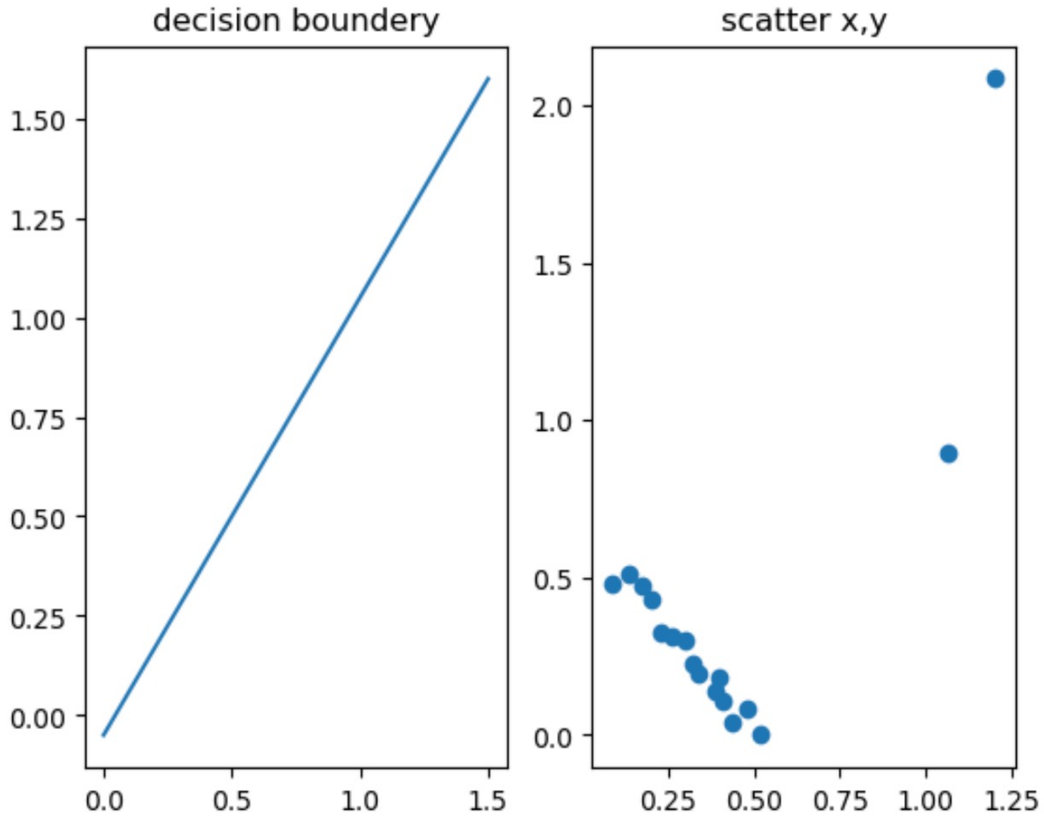


RANSAC

VS



Pseudo-inverse

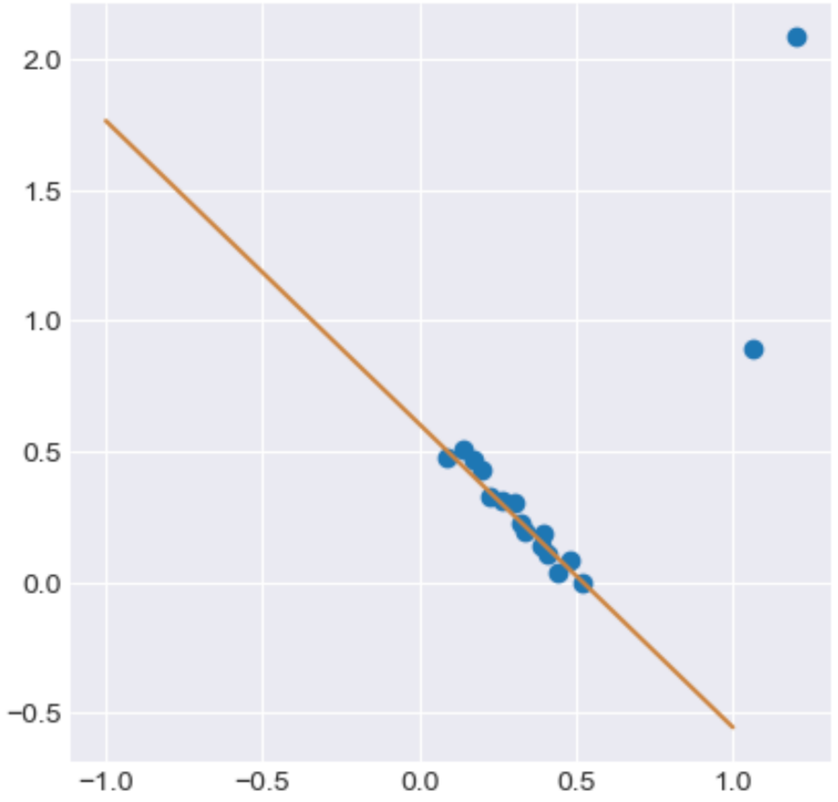


Part A: The analysis of the driven line

Because the number of points for training the weights is enough and because there are noise data and outliers in this dataset, the training line is oriented towards the outliers in order to generate quantitative error. Due to the large distance between outliers, this line passes through the main data. The noise data of this line passes through two noise inputs, too.

Measuring this method's error:

```
In [11]: def square_error_loss(y_true, y_pred):
             return (y_true - y_pred) ** 2

In [12]: def mean_square_error(y_true, y_pred):
             return np.sum(square_error_loss(y_true, y_pred)) / y_true.shape[0]

In [13]: print('MSE for sudo inverse :'+str(mean_square_error(y_train.reshape(17,1), forward_propagation(x.reshape(1,17),W).resh

         MSE for sudo inverse :0.12057585388293435
```



Part C: RANSAC method

This method first was used by Fischler and Bolles. They used this method to find an answer for Location Determination Problems. This method is a repetitious method for estimating a mathematical model's parameters of a set of observed data. Provided that the outliers do not have any effect on the model's parameters and that one can delete these outliers and estimate a function without considering them, this method can be used. Furthermore, this method is an indecisive method, which means with more repetition of this algorithm, there is higher probability of a better fitted model. Therefore, this method can be used for recognizing and identifying the outliers.

Measuring this method’s error:

In [19]:
```
print('MSE for RANSAC :'+str(mean_square_error(y.reshape(17,1),regressor.predict(x.reshape(-1, 1)).reshape(17,1))))
```

```
MSE for RANSAC :0.6230943439018531
```