Project 1

M2177.005800 Basic Mathematics and Programming Practice for Machine Learning

SOHEIL HOUSNI 2024-81641
TO ALINEJAD LASHKARIANI HAMIDREZA

2024/06/10 SEOUL NATIONAL UNIVERSITY

Contents

Linear regression

Explaination of the code :

Generation of the dataset and pre-steps

The first part of my code is the construction of the gradient descent algorithm that I will use to produce the results that we will discuss later in this report

First I imported the different modules usefull to it, so numpy (which allows to use arrays, and so vectorization, but also to use the random functionalitites and finally to make calculations), sklearn.datasets (in order to generate the dataset that I used for training my algorithm) and finally matplotlib.pyplot (for the visualisation). This is the way I implemented those modules.

Then I used: np.random.seed(42). This option allows to make in sort to control randomity. Thus, when I will generate my dataset with make_regression, I will always obtain the same values for my features and my target variable. This makes easier the discussion of results and the comparison of them when I will try different parameters and initializations for my algorithm.

After that, I generated my dataset with: x,y=make_regression(100,2,noise=10). This line means that I generated a dataset for regression problems that contains 100 data (examples) and 2 features x. Values for the target variable y were also generated.

After that, I made: X=np.hstack((x,np.ones((x.shape[0],1)))). With np.hstack I gathered in one numpy array the features part of my dataset (x) and one column full of one (np.ones) that has for dimensions the number of lines of x (100 as x contains 100 values) and one column. Those two tabs are gathered horizontally. This column of ones added is the bias column. It will allow to facilitate the calculus of w1x1+w2x2+b by simply doing a matrices product of the matrix X obtained with the matrix of parameters (we will see that below).

After that, I reshaped my y column with y=y.reshape((y.shape[0],1)). Indeed, y was a 1 dimension array, but we need it to be a 2 dimensions in order to use it for matrices calculation. Therefore, y keeps its number of lines (y.shape[0]), but we add one column (,1)

Construction of the Gradient Descent algorithm

Once those pre-steps are done, we can start building the algorithm:

Definition of the model:

The first thing to do is the definition of the model that we will use:

```
def model(X,parameters):
    return np.dot(X,parameters)
```

To do that, I defined a function model that takes for arguments X and parameters. This model returns the matrices product of X and parameters with the use of np.dot. Indeed, for a linear regression, our model takes the form of x1w1+x2w2+b.

Definition of the cost function:

Then we have to define the cost function:

```
def cost_function(X,y,parameters):
    m=len(y)
    return (1/(2*m))*np.sum((model(X,parameters)-y)**2)
```

The cost function that I developed takes for argument X, y and parameters, and returns the MSE. m=len(y) means that m takes for value the length of the column y which represents the number of data (of examples). np.sum calculates the sum of the values it takes, here the sum of the results of the calculation (model(X,parameters)-y)**2). This calculation returns an array of shape (X.shape[0],1) and np.sum calculate the sum of the values of this array.

Definition of the Gradients:

We can now define our gradient:

```
def grad(x,X,y,parameters):
    m=len(y)
    grad=[]
    for i in range(x.shape[1]):
        column=x[:,i].reshape((x.shape[0],1))
        grad_w = (1/m)*np.sum((model(X,parameters)-y)*column)
        grad.append(grad_w)
    grad_b = (1/m)*np.sum(model(X,parameters)-y)
    grad.append(grad_b)
    grad=np.array(grad).reshape(len(grad),1)
    return grad
```

This function takes for argument x,X,y and parameters. As earlier, m is the number of data. "grad" is an empty list. The idea of this function is that, for each feature x_i of x (the number of features of x is its number of columns, so x.shape[1]), we calculate the derivative of the cost function with respect to the corresponding coefficient of this feature x_i (w_i). Then, we add this derivative to the "grad" list. We can do that, as with respect to each of the coefficient w_i of x_i, the formula of the derivative of the cost function stays the same:
grad_ w_i = (1/m)*np.sum((model(X,parameters)-y)*column). "colum" extracts the values of the column i of x and puts them in a 2-dimensions array of a shape (x.shape[0],1) (a column that as the same number of lines as the number of examples in each column of x) "*column" means that we multiply by the values of the feature i corresponding, by taking into account all elements of the i-th column of x (it is the only thing that varies in this calculation). The (model(X,parameters)-y)*column is a 2-dimensional array and np.sum calculate the sum of all its elements, which is then multiply by 1/m.

After that, we calculate the derivative of the cost function with respect to b that we also add to the "grad" list: grad_b = (1/m)*np.sum(model(X,parameters)-y) / grad.append(grad_b)

Then, we have to transform our "grad" list (that contains all our gradients) into an array that have for number of lines the number of gradients (so the length of the "grad" list) and one column: grad=np.array(grad).reshape(len(grad),1). Our function return this grad array.

Construction of the Gradient Descent algorithm:

We can now construct the gradient_descent algorithm, and to do that, I define a function that take for argument x,X,y,parameters and the learning rate, the number of iterations and the tolerance. We first define two empty lists: cost_history and N_step. Then, in a loop that will work for our number of iterations, we will first stock the values of our gradient function in a variable call "gradient":

```
gradient=grad(x,X,y,parameters).
if np.linalg.norm(gradient) < tolerance:
        break
```

This part of the code indicates that if the norm of our gradient vector is inferiror to the tolerance's threshold that we defined, the loop (and the algorithm) have to stop working and we will obtain the value of our parameters determined by the alogithm at this point. In the algorithm, parameters are calculated and actualized with the formula of the gradient descent: parameters=parameters-learning_rate*gradient. We also report the cost obtained at each iteration in our cost_history list: cost_history.append(cost_function(X,y,parameters)). Finally, we report the step in which we currently are in the N_step list: N_step.append(i). The function returns then the parameters finally obtained after the gradient descent algorithm, the cost history list and the N steps list. We can stock those three returns in variables with: final_parameters, cost_history, N_step=grad_descent(x,X, y, parameters, learning_rate, n_iterations, tolerance).

Final predictions and calculation of the R squared value

Then we can determine the final predictions of our model using as arguments X and our final parameters resulting of the gradient descent: final_predictions=model(X,final_parameters).

This variable "final prediction" is useful for calculating the R squared of our model. To do that, we define a function that takes for argument y and final-predictions. In this function, as earlier, m=len(y) means that m is the number of examples. Then we calculate the mean of y values with: y_mean=np.sum(y)/m. We then calculate the ESS (ESS=np.sum((final_predictions-y_mean)**2)) and the TSS (TSS=np.sum((y-y_mean)**2)) for finally calculating the R_squared (R_squared=ESS/TSS) which is the ratio between the two of them.

After that, we can use the model with different values of parameters, learning rate, number of iterations and tolerance and obtain our results and determine which final parameters, R squared and cost we obtain.

Reconstruction of the model with Sklearn

In order to determine if our model works properly, we can recreate it using sklearn. First we have to import all necessary modules: from sklearn.linear_model import LinearRegression (to use the linear regression model), from sklearn.metrics import mean_squared_error (to calculte the mean squared error), from sklearn.metrics import r2_score (to calculte the R squared of our model). Then we define the model that we want to use (model=LinearRegression()), we fit it to our data x and y (model.fit(x,y)), and we determine the predictions of our model on our values x (y_pred=model.predict(x)). Then we can determine the coefficients parameters obtained (coefficients = model.coef_) and our intercept parameter (intercept = model.intercept_). Finally, we can calculate the mean squared error (mse = mean_squared_error(y, y_pred)) and our R squared (r2=r2_score(y, y_pred)). We can now compare the results obtained by the model constructed and the one of sklearn algorithm.

Discussion of results:

Let's now discuss the results of our model.

Fisrt of all, we will note that the result obtained by the sklearn model are w1=86.003, w2=74.117 and b=0.216. The MSE obtained is 115.014 and the R squared is 0.99.

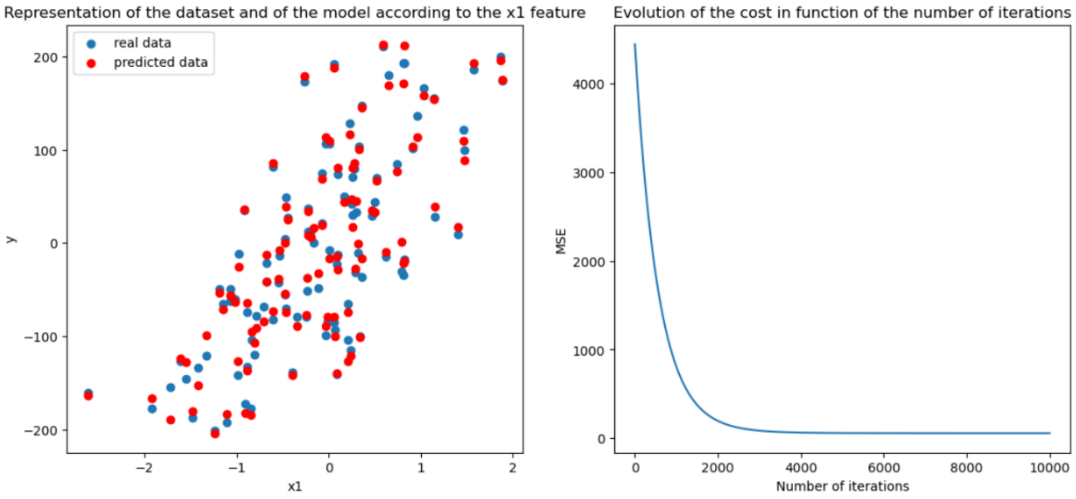
To discuss the result our model, we will use different values of initialization parameters and different values of learning rate.

Different values of initialization parameters:

Let's start with the different values of initialization parameters used with a learning rate of 0.001, a number of iterations of 10 000 and a tolerance of 1e-6 .

With initial values w1=10, w2=10 and b=10

I first tried using w1=10, w2=10 and b=10. I obtained as values of final parameters w1=85.947, w2= 74.12 and b=0.197. The R squared was of 0.99 and the final cost was 57.51. Here are the graphs obtained of the visualization of my dataset and of my model, and of the cost curve in function of the number of steps:

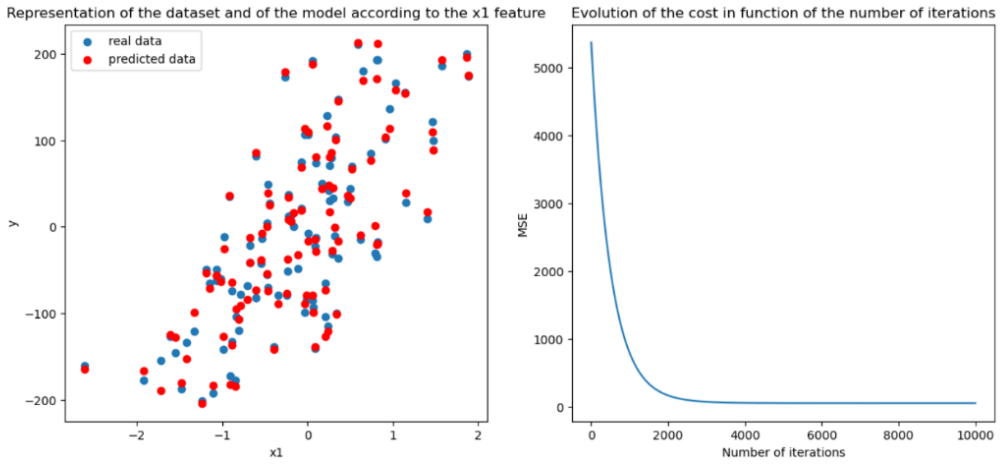


As we can see on the left graph, the predicted values of my model correspond well to the real data. Indeed, we can observe that each predicted values almost confuse with the real data on the graph. This good performance can be shown by the cost curve that decreases strongly during the first 2500 iterations and then stabilizes. Moreover, the values of the paramaters that I obtained are really closed to those given by the sklearn model with respectively: 85.947 against 86.003 for w1, 74.12 against 74.117 for w2 and 0.197 against 0.216. Furthermore, the cost of my model is amply inferior to the MSE of sklearn with respectively: 57.51 against 115.014, and the R squared is similar (0.99).

With initial values of w1=100, w2=100 and b=100

Then I tried to inialize parameters with bigger values(w1=100, w2=100 and b=100) with keeping the number of iterations, the learning rate and the tolerance same as in the previous training. As earlier, my model realized good performances. I obtained again final values of my parameters really closed to those given by sklearn with respectively: 86.043 against 86.003 for w1, 74.112 against 74.12 for w2 and 0.235 against 0.197 for b. The cost that I obtained is

57.5076, which is slightly better than the previous one and still better than the cost of the sklearn model, and the R squared is similar with again a value of 0.99. We can see those good performances with the graphs:



Like in the previous graphs, we can observe on the left graph that each of the predicted points are really closed and almost touch their real corresponding points, and on the right graph that the cost curve, as the previous one, decreases strongly during the first 2000 iterations and then stabilizes.

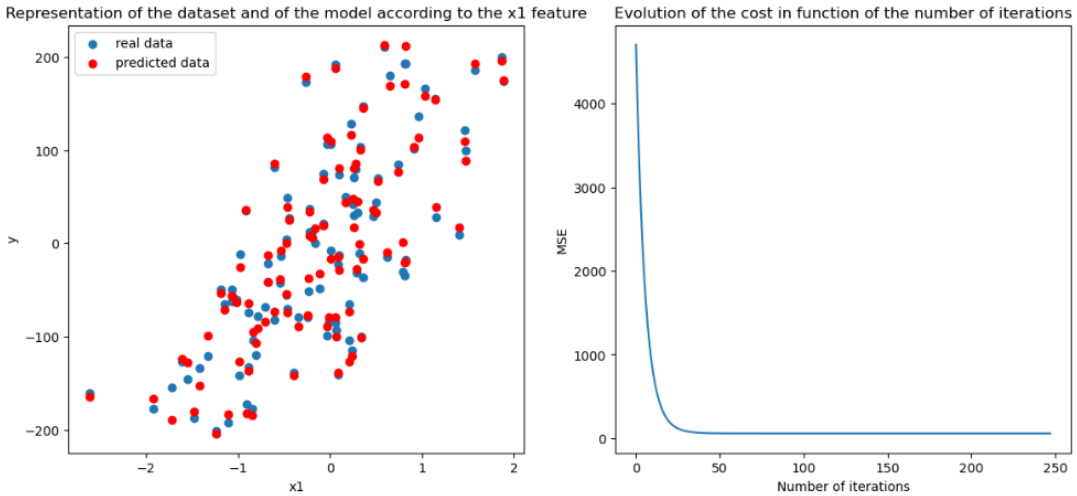
Different values of learning rate:

We can now try the model with different values of learning rate. For this, I chose for initialization parameters: w1=0, w2=0 and b=0, a number of iterations of 10 000 and a tolerance of 1e-6.

For learning rate of 0.1

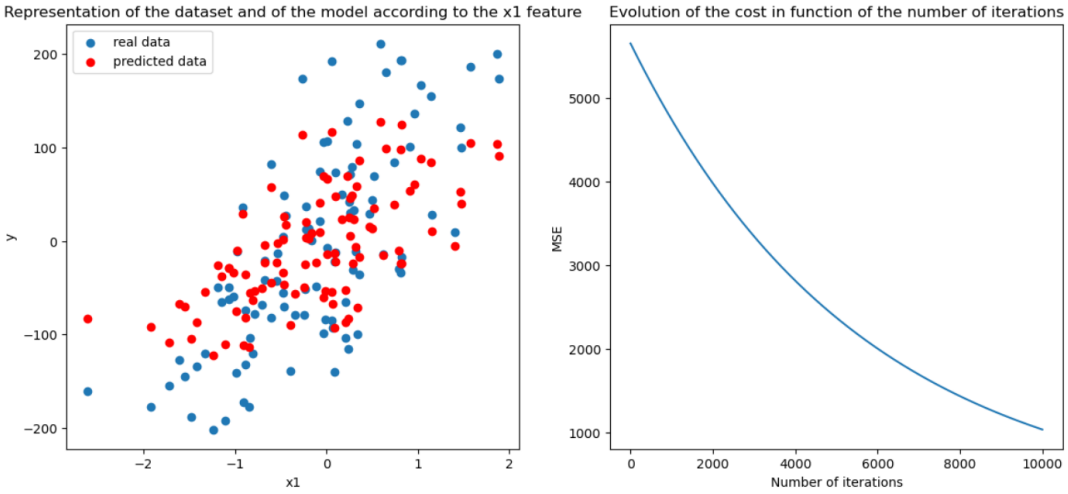
The first value of learning rate that I tried is 0.1. The model did a really good performance, with values of final parameters really close to the ones given by the sklearn model, with respectively: 86.003 against 86.003 for w1, 74.117 against 74.12 for w2 and 0.216 against 0.197 for b. The cost that I obtained is 57.507, which is amply better than the sklearn model cost (115.014) and the R sqared value is similar with 0.99.

Those good performances can be oberved on the graph:



For learning rate of 0.0001

Then I tried a really small values for the learning rate with 0.0001, with keeping the number of iterations, the tolerance, the initial values for parameters as the same.
This time, the performance of the model increasingly decreased. First, the values of the final parameters that I obtained are far from those obtained by slearn, with respectively: 45.462 against 86.003 for w1, 47.448 against 74.12 for w2 and -3.09 against 0.197 for b. The cost that I obtained is also important, and far more superior to the one obtained by sklearn with 1035.02 against 115.014, and the R squared is really low and far less inferior to the one obtained by sklearn with 0.341 against 0.99. This decrease of performance can be observed with the graphs:



On the left graph, we can see that the predicted points don't match with the real data and that they are far away. Moreover, on the right graph, we can see that the cost curve decreases, but really slowly, with a really low decrease rate. However, we can observe that this cost curve doesn't stabilize, which is logic. Indeed, the decrease of performance is due to the fact that the learning rate is too small. Therefore, it would have required a far larger number of iterations to obtain a good performance at the end. This is why the cost curve doesn't stabilize: because it could have continue to decrease if the number of iterations was more important.

Logistic regression

Explaination of the code:

Generation of the dataset and pre-steps

The first part of my code is the gradient descent algorithm that I will use to produce the results that we will discuss later in this report. I first imported the modules I needed for my code: numpy, matplotlib and sklearn.datasets. make_classification (to generate the dataset that we will use).

Like earlier, I used np.random.seed(3) to control the randomity of the generation of the dataset, which will make easier the discussion of the results.

Then, I generated the dataset that we will use for the algorithm with x,y=make_classification(n_samples=100, n_features=2, n_informative=2, n_redundant=0, n_clusters_per_class=1, n_classes=2). make_classification allows to generate a dataset adapted for classification problem. Here, the number of data (examples) is 100 and there are two features (that are both informative) and two classes (and points of the same classes are gathered).

Then, like earlier, I reshaped the 1 dimension y array with y=y.reshape((y.shape[0],1)), so it will become a 2 dimensions array that keeps its number of lines and that has one column. I also combined horizontally the array x and a column full of ones that has the same number of lines as the array x. To do that, I used np.hstack with X=np.hstack((x,np.ones((x.shape[0],1)))). This column of ones added is the bias column. It facilitates the calculation of w1x1+w2x2+b by simply doing a matrices product of the matrix X obtained with the matrix of parameters (we will see that below).

After those pres-teps are done, we can now construct the algorithm.

Construction of the Gradient Descent algorithm

Construction of the model

I first created the model used for making predictions. For that, I defined a function that takes for arguments X and parameters. In this function, we must first calculate z, which is the calculation of w1w1+w2x2+b. Therfore, for calculating z, we can simply put: z=np.dot(X,parameters), which calculates the matrices product of X (which contains the bias column) and the matrix of parameters (which contains w1,w2 and b). Then, we have to calculate f, which is the sigmoid of z: f=1/(1+np.exp(-z)). Our model function returns f.

Definition of the cost function:

After that, we can calculate our cost. To do that, I defined a function that takes for arguments (X,y,parameters). Like earlier, m=len(y) represents the number of data (so the length of y column) and this function returns the calculation of the log-loss: (1/m)*np.sum(-y*np.log(model(X,parameters))-(1-y)*np.log(1-model(X,parameters)))

Definition of the Gradients:

Now, we can calculate the gradients that we will use for our gradient descent:

```
def grad(x,X,y,parameters):
    m=len(y)
    grad=[]
    for i in range(x.shape[1]):
        column=x[:,i].reshape((x.shape[0],1))
        grad_w = (1/m)*np.sum((model(X,parameters)-y)*column)
        grad.append(grad_w)
    grad_b = (1/m)*np.sum(model(X,parameters)-y)
    grad.append(grad_b)
    grad=np.array(grad).reshape(len(grad),1)
    return grad
```

To do that, I created a function that takes for arguments x,X,y,parameters. m=len(y) represents the number of data. "grad" is an empty list. The idea is that, for each feature x_i of x (the number of features of x is the number of columns of the x array, so x.shape[1]), we calculate the derivative of the cost function with respect to the corresponding coefficient of this feature x_i (w_i). Then, we add this derivative to the "grad" list. We can do that, as with respect to each of the coefficient w_i of x_i, the formula of the derivative of the cost function stays the same: grad_w = (1/m)*np.sum((model(X,parameters)-y)*column. colum extracts the value of the column i of x and puts them in a 2-dimensions array of a shape (x.shape[0],1) (a column that has the same number of lines as the number of examples in each column of x) "*column" means that we multiply by the values of the feature i corresponding, by taking into account all elements of the i-th column of x (it is the only thing that varies in this calculation). The (model(X,parameters)-y)*column is a 2-dimensions array, and np.sum calculates the sum of all its elements, which is then multiply by 1/m.

After that, we calculate the derivative of the cost function with respect to b that we also add to the "grad" list: grad_b = (1/m)*np.sum(model(X,parameters)-y) / grad.append(grad_b)

Then, we have to transform our "grad" list (that contains all our gradients) into an array that have for number of lines the number of gradients (so the length of the "grad" list) and one column: grad=np.array(grad).reshape(len(grad),1). Our function return this grad array.

Construction of the Gradient Descent algorithm:

We can now construct the gradient descent algorithm, and to do that, I defined a function that take for arguments x,X,y,parameters and the learning rate, the number of iterations and the tolerance. We first define two empty lists: cost_history and N_step. Then, in a loop that will works for our number of iterations, we will first stock the value of our gradient function in a variable call "gradient": gradient=grad(x,X,y,parameters).

```
if np.linalg.norm(gradient) < tolerance:
        break
```

This part of the code indicates that if the norm of our gradient vector is inferior to the tolerance's threshold that we defined, the loop (and the algorithm) have to stop working and we will obtain the value of our parameters determined by the alogithm at this point. In the algorithm, parameters are calculated and actualized with the formula of the gradient descent: parameters=parameters-learning_rate*gradient. We also report the cost obtained at each iteration in our cost_history list: cost_history.append(cost_function(X,y,parameters)). Finally,

we report the step in which we currently are in the N_step list: N_step.append(i). The function returns then the parameters finally obtained after the gradient descent algorithm, the cost history list and the N steps list. We can stock those three returns in variables with: final_parameters, cost_history, N_step=grad_descent(x,X, y, parameters, learning_rate, n_iterations, tolerance).

Final predictions and calculation of the confusion matrix, Accuracy, Precision, Recall and F_1 score

Then we can determine the final probabilities predictions of our model using as arguments X and our final parameters resulting of the gradient descent: final_predictions=model(X,final_parameters).

Then we can determine the final predictions of our model. To do that, we first define a function that takes for argument X and parameters. In this function we return np.where(model(X,parameters) >= 0.5, 1, 0). This calculation means that the function return 1 if the value of our model(X,parameters) (that we constructed earlier and which returns the value of the sigmoid of z (z is the function w1x1+w2x2+b)) is greater or equal than 0.5, and return 0 in the other case. This function returns an array that has the same number of lines as y and one column and which contains all the final predictions.

We can then stock our final predictions, resulting of our predicition function that takes for argument X and final_parameters, in a variable : final_predictions=predictions(X,final_parameters)

Then, we can calculate our confusion matrix. To that, we must first determine the elements of our final_predictions array that are true positives, true negatives, false positives and false negatives. To do that, we first create an empty list for each of those categories (4 empty lists in total). Then, we create a for loop that will verify the value of each elements i of our final_predictions array, and if the corresponding element j of the y array is equal to the same value. Then, if i=1 and j=1, element i is added to the True positives list; if i=1 and j=0, element i is added to the False positives list; if i=0 and j=0, element i is added to the True negatives list; and finally if i=0 and j=1, element i is added to the False negatives list. To realize all those Ifs, we use for i,j in zip(final_predictions,y) that allows to iterate on the element of final_predictions and y at the same time. Then, we calculate the length of each of the final lists, which will give us the number of True Positives, True Negatives, False Positives and False Negatives (respectively NTP,NTN,NFP and NFN). We can finally construct our confusion matrix in the form of a numpy array with: confusion_matrix=np.array([[NTP,NFN],[NFP,NTN]]).

Then, with NTP,NFN,NFP and NTN, we can calcualte the Accuracy, Precision, Recall and F_1 score:

```
Accuracy=(NTP+NTN)/(NTP+NTN+NFP+NFN)
Precision=NTP/(NTP+NFP)
Recall=NTP/(NTP+NFN)
F1_score=2*((Precision*Recall)/(Precision+Recall))
```

Construction of the decision boundary

Finally, in order to calculate our decision boundary, we first have to create the x axis of our graph. To do that we use: x_boundary=np.linspace(-20, 20, 100) which will generate a list of 100 values between -20 and 20 equally separated that x_boundary can take. Then we calculate the values of our decision boundary in function of x-boundary with:

y_boundary=(-final_parameters[2] -final_parameters[0]*x_boundary)/final_parameters[1]

Indeed, the decision boundary is the line of points for which z=0, which here means when w1x1+w2x2+b=0. In order to express x2 in function of x1 to visualize this boundary, we can transform this equation into x2=(-b-w1x1)/w2 = (-final_parameters[2] - final_parameters[0]*x_boundary)/final_parameters[1]

Reconstruction of the model with Sklearn

Finally, in order to determine if our moder works properly, we can recreate it using sklearn functionnalities. To do that, we must first import the modules that we will use: from sklearn.linear_model import LogisticRegression (for importing the logistic regression model), from sklearn.metrics import log_loss (for importing the log loss cost function) and from sklearn.metrics import confusion_matrix, classification_report. Then we define the logisitc regression model that we will use (model=LogisticRegression()), we fit the model to our data x and y (model.fit(x,y)), we determine the predictions of our model on our data x (y_pred=model.predict(x)), we determine the coefficients parameters (coefficients = model.coef_) and the interecept (intercept = model.intercept_) used by our model. Then we calculate the probabilities that our predicitions y_pred take the value y_pred=1 (model.predict_proba(x)[:,1]) (therefore, we only extract the values of the second column of model.predict_proba(x), because the first coulumn indicate the probabilities that our y_pred take the value 0). Finally, we can calculate the log loss cost of our model with log_loss_value = log_loss(y, y_pred). We can then determine our confusion matrix (cm = confusion_matrix(y, y_pred)) and our classification report (classification_rep = classification_report(y, y_pred))

Discussion of results:

Let's now discuss the results of our model.

Fisrt of all, we will note that the result obtained by the sklearn model are w1= -0.762, w2=-3.071 and b=0.387. The log loss value obtained is 1.442. The accuracy score is 0.96, the precision score is 0.94, the recall score is 0.98 and the f1 score is 0.96.

To discuss the result our model, we will use different values of initialization parameters and different values of learning rate.

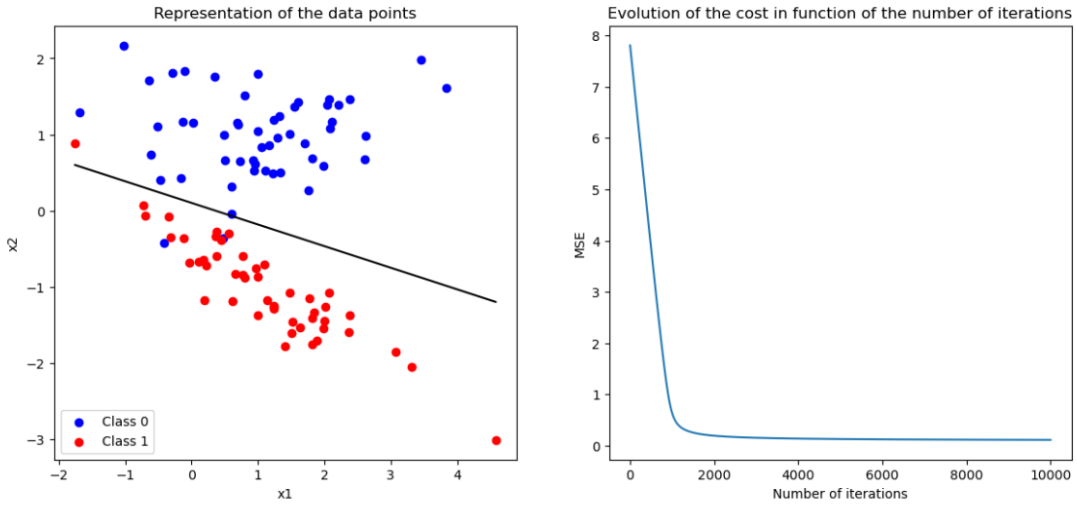
Different values of initialization parameters:

Let's start with the different values of initialization parameters used with a learning rate of 0.01, a number of iterations of 10 000 and a tolerance of 1e-6 .

With initial values w1=5, w2=5 and b=5

First, I tried with the following intial values for my parameters: w1=5, w2=5 and b=5. The values of my final parameters are almost closed to those given by the sklearn model, with respectively: -1.151 against -0.762 for w1, -4.058 against -3.071 for w2 and 0.413 against 0.387 for b. However, the final cost obtained is 0.115 against 1.442 for the sklearn model. The accuracy is 0.97 against 0.96, the precision 0.959 against 0.94, the recall is 0.979 against 0.98 and the f1 score is 0.969 against 0.96.

Therefore, we can say that the model has good performances, which can be shown with the graphs:

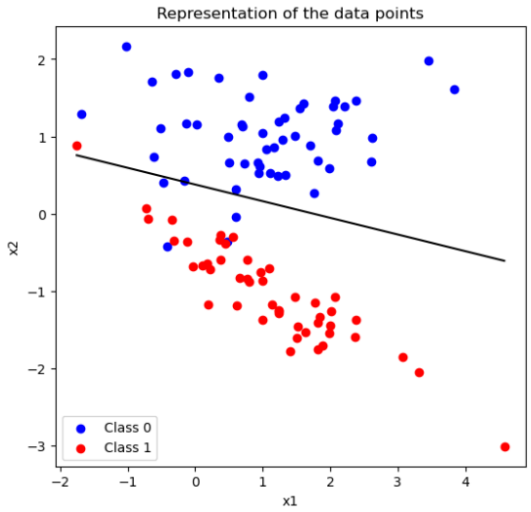


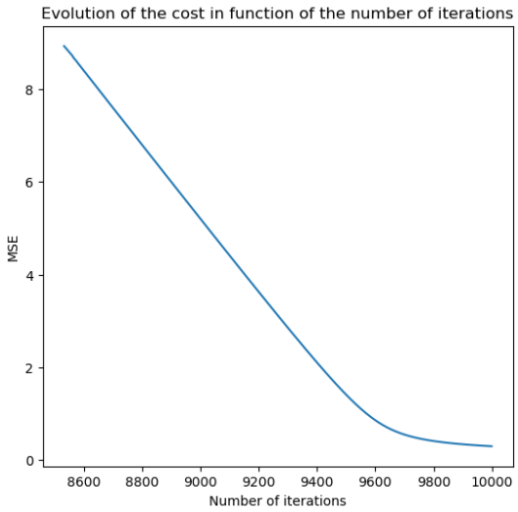
On the left graph, we can see that most of the points are in the good side of the decision boundary, and on the right graph, we can observe that the cost curve decreases strongly until the 1500th iterations and then stabilizes, which demonstrates that the model works well.

With initial values w1=50, w2=50 and b=50

Then, I tried to initialize my parameters with w1= 50, w2= 50 and b= 50, with keeping the number of iterations, the learning rate and the tolerance as the same as in the previous training.

In this case, the model performs less well than in the previous time. Indeed, the values of the parameters obtained are closed to the values given by the sklearn model with respectively: -0.278 against -0.762 for w1, -1.29 against -3.071 for w2 and 0.487 against 0.387 for b; but we can observe a slight increase of the distance for the value of w2 from its ideal value. This can be explained by the fact that the initialization values of the parameters were really far from their optimized values. However, the cost is 0.293 against 1.442 for the sklearn model. The accuracy decreased from 0.97 to 0.95 against 0.96 for the sklearn model, the precision decreased from 0.959 to 0.922 compared to 0.94 for the sklearn model, the recall stayed the same with 0.979 against 0.98, and the f 1 score decreased from 0.969 to 0.949 against 0.96. This slight decrease of performance can be observed in the graphs:





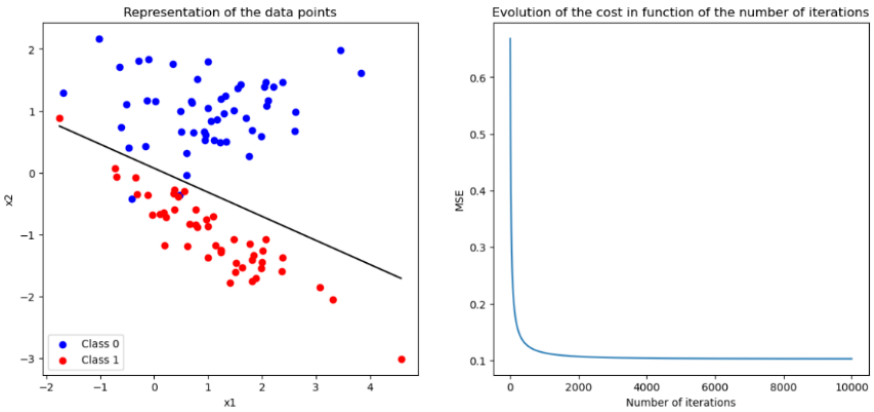
We can especially see that the decrease of the cost curve function is less intense and slower than in the previous graph. As mentionned above, this decrease of performance can be explained with intial values for parameters that are too far from their optimized valued.

Different values of learning rates:

Let's now study the results with different values of learning rate used with initialized parameters of w1=0,w2=0 and b=0, a number of iterations of 10 000 and a tolerance of 1e-6 .

With a learning rate of 0.1

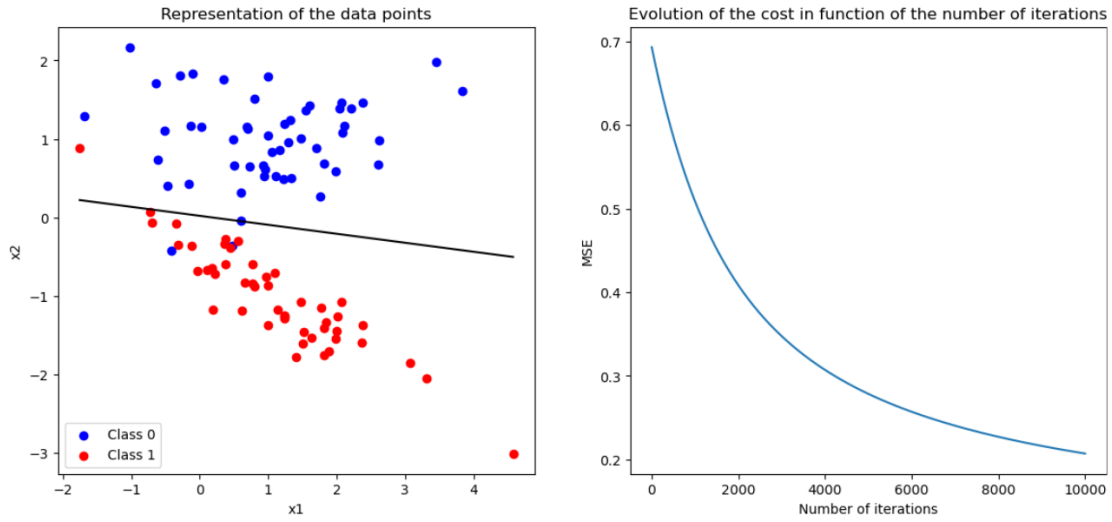
The first value of learning rate that I tried is 0.1. With this try, the model performed well. However, the values of the final paramaters are far from the values given by the sklearn model with respectively: -2.362 against -0.762 for w1, -6.080 against -3.071 for w2 and 0.435 against 0.387 for b. However, the final cost is 0.103 against 1.442 for the sklearn model. The accuracy is 0.97 against 0.96, the precision 0.959 against 0.94, the recall is 0.979 against 0.98 and the f1 score is 0.969 against 0.96. All those metrics show that the model performs well, which we can also observe with the graphs:



On the left graph, we can see that almost all of the points are on the good side of the decision boundary, and on left graph that the cost curve decreases strongly during the first 1500th iterations and then stabilized.

With a learning rate of 0.001

Then, I tried with a value for the learning rate of 0.001, with keeping the initialization values, the number of iterations and the tolerance as the same as the previous training. The performance obtained slightly decreased compared to the previous case. We first note that the values for our final parameters are closer to the values given by the sklearn model than in the previous situation, with: -0.221 against -0.762 for w1, -1.93 against -3.071 for w2 and 0.039 against 0.387 for b. However, the final cost obtained is higher, passing from 0.103 to 0.208 against 1.442 for the sklearn model. The accuracy stayed at 0.97 against 0.96 for the sklearn model, the precision stayed at 0.959 compared to 0.94 for the sklearn model, the recall stayed the same with 0.979 against 0.98 and the f 1 score stayed also at 0.969 against 0.96. This slight decrease of performance, demonstrated by the increase of the final cost obtained, can be observed witht the graphs:



We especially note on the left graph that the cost curve decreases slower, at a lower rate, compared to the previous cost curve. This can be explained by the fact that the learning rate might be too small and that the number of iterations was not sufficient to find the optimal values for our parameters. This is demonstrated by the fact that, contrary to the previous cost curve, this one does not stabilized at the end, which means that it could have continue to decrease if more iterations were executed.

KNN

Explaination of the code

Generation of the dataset

I first imported the modules that I needed for my code: numpy, matplotlib and sklearn.datasets. make_classification (to generate the dataset that we will use).

Like earlier, I used np.random.seed(0) to control the randomity of the generation of the dataset, which will make easier the discussion of the results.

Then, I generated the dataset that we will use for the algorithm with x,y=make_classification(n_samples=100, n_features=2, n_informative=2, n_redundant=0, n_clusters_per_class=1, n_classes=2). make_classification allows to generate a dataset adapted fo classification problem. Here, the number of data (examples) is 100, there are two features (which are both informatives) and two classes (and the data of the same classes are clustered).

Construction of the KNN algorithm

Then, I created my KNN algorithm with a function that takes for argument point (the new point) and k. In this function, I first defined an empty list name "distance_list" that will contain distances of the new point from all the other points in the dataset. Then, I created a loop that says that for each data of x ("for i in x" indicates "for each line of the array x", and each line is a data that have for coordinate x1 and x2, which are the columns of the array x), we calculate the euclidian distance with our new point: distance=math.sqrt((i[0]-point[0])**2+(i[1]-point[1])**2). In this calculation, math.sqrt allows to calculate the square root, i[0] is the value of x1 of the i-th point and point[0] is the value of x1 of our new point, and in the same way, i[1] is the value of x2 of the i-th point and point[1] is the value of x2 of our new point. Indeed, x is an array. Once the distance is calculated, we add it to our "distance_list" list.

When the loop have finished to work and that our list is completed, we have to determine the elements of the list that are the smallest, in order to determine the data that have the closest distance with our new point. To do that, we use sort=np.argsort(distance_list), which will stock a list of indexes in the variable "sort". This list contains the indexes of the elements sorted in an ascending order. In other words, the list contains the index of the element with the smallest distance from our new point in first position, to the index of the element with the largest distance from our new point at last position, in order.

Then, we create two new empty lists "number_0" and "number_1" that will respectively include the value of the k nearest data from our point whereas they take the value 0 or 1.

After that, I created a new loop that says that for each of k data that have the smallest distance from our new point, if it has a y=0, it will go to the number_0 list, and if it has a y=1, it will go to the number_1 list:

```
for i in range(k):
    if y[sort[i]]==0:
        number_0.append(y[sort[i]])
    if y[sort[i]]==1:
        number_1.append(y[sort[i]])
```

y [sort[i]] means the element y that has for index the value of the "sort" list that has for index i. For exemple y[sort[1]] will be the element of y that has for index the first value of the sort list. This makes sense because sort is the list of index that sort the distances in ascending order. So y[sort[1]] will be the element of y for which the distance of its corresponding data x from the new point is the smallest, y[sort[2]] would be the element of y for which the distance of its corresponding data x from the new point is the seond smallest and so on.
At the end, I proceeded to the majority voting with if and elif conditions. This one says thay if the length of number_0 list is superior to length of number_1 list, which means that number_0 list contains more elements than number_1 list, the predicition would be 0. In the other sense, the prediction would be 1.

Finally, the function returns the prediction.

Reconstruction of the model with Sklearn

In order to determine if the model performs well, we can compare the result with the KNN model of SKLearn. First, we need to import the useful module: from sklearn.neighbors import KneighborsClassifier. Then, we define the model we will use (knn_sk = KNeighborsClassifier(n_neighbors=10)), we fit the model to our data x and y (knn_sk.fit(x, y)) and we finally make our predictions (y_pred = knn_sk.predict(x))

Discussion of the results:

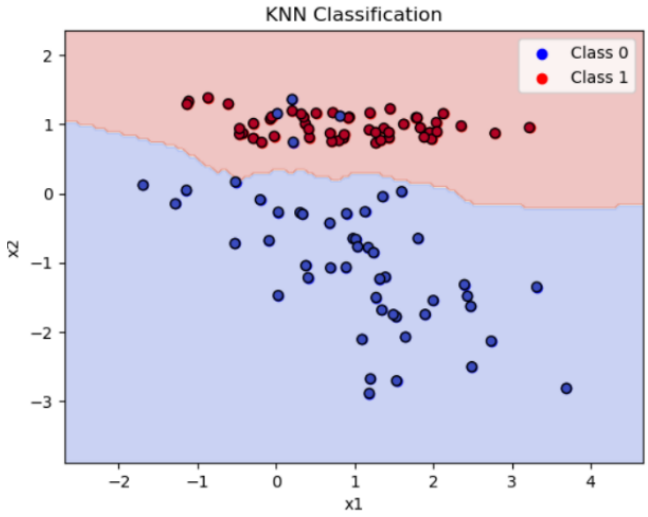
Different values of k (number of nearest neighbours):

We can now compare the results for different values of k (the number of nearest neighbours).

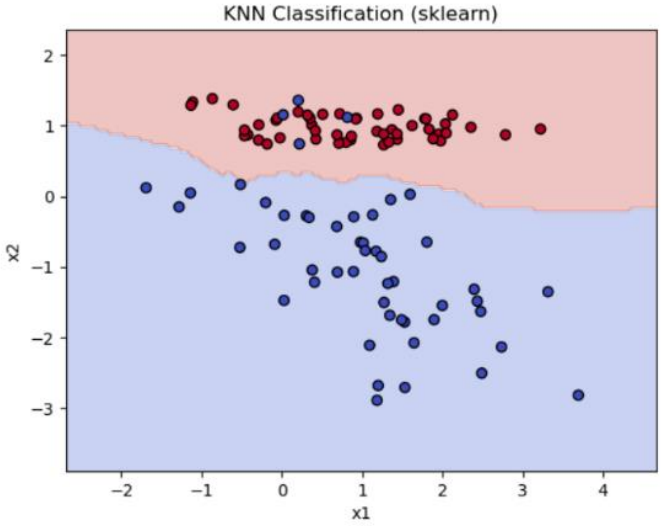
With k=5

I first chose k=5 and I keep my dataset similar.

Here is the graph that I obtained from my model:



Here is the graph that I obtained from the sklearn model:

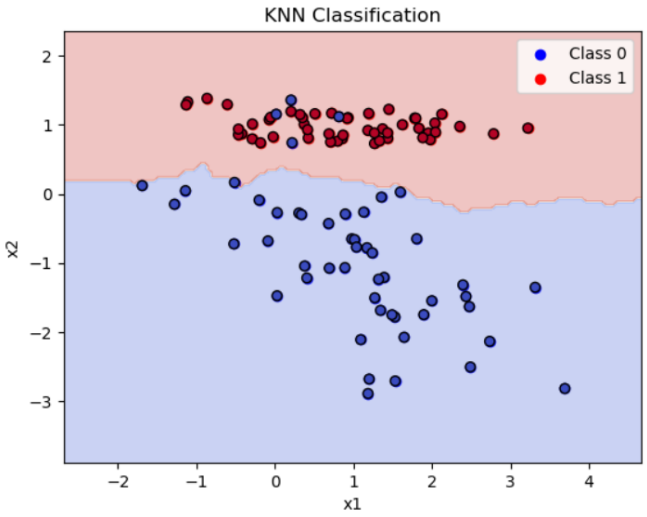


We can see that both graphs are really similar (if not the same), which shows that the model works well.

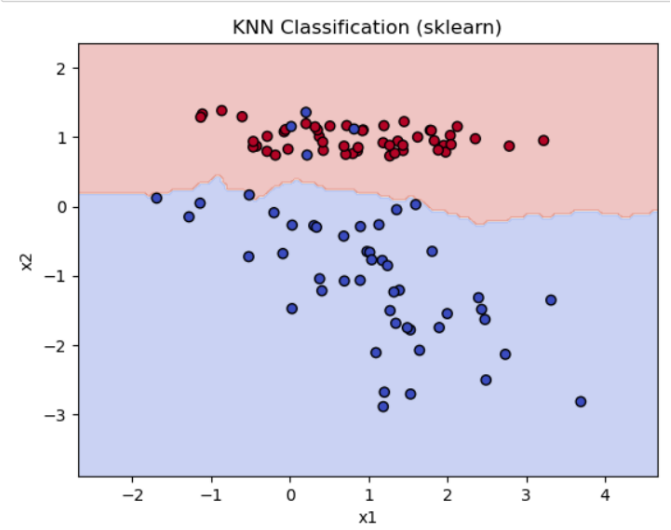
With k=10

We can now try with a value of k=10.

Here is the graph that I obtained from my model:



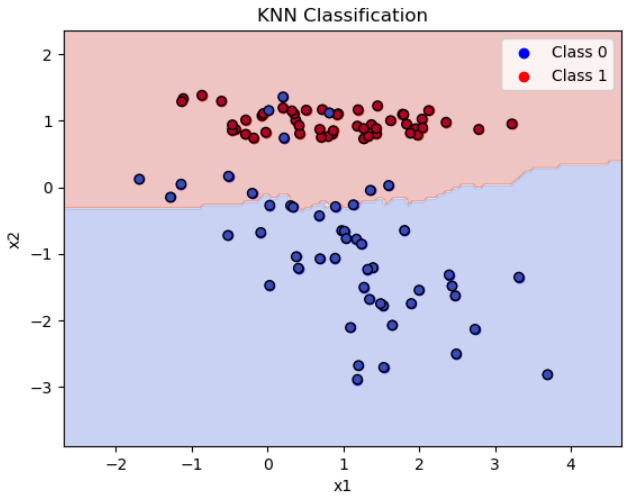
Here is the graph that I obtained from the sklearn model:

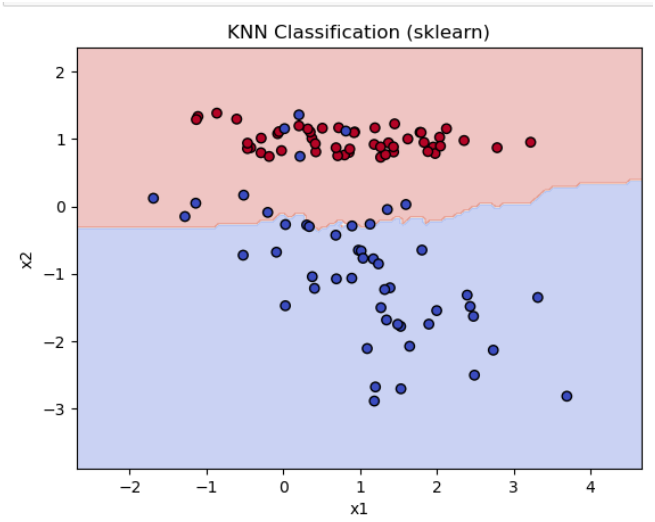


Like in the previous case, both graphs are the same, which shows that the model works well.

With k=60

Finally, for k=60, we obtain the following graphs respectively for my model and for the sklearn model:





Like in the previous case, both graphs are the same, which shows that the model works well. However, we can observe that, compared to both previous cases, more blue points close to the decision boundary are positionned on the wrong side of the decision boundary (on the class 1 side). This can be explained by the fact that the number of neighbors, with k=60, is to high (and above 50, so above half of the total number of data). Therefore, as there are more red points (class 1) than blue points (class 0), the prediction of classes for blue points that are closed to the decision boundary will include more red points than blue points for the majority voting. We face here the beginning of underfitting and an oversimplication of the classification problem: the bias of the model starts to become too strong. This explains why more of those points are on the wrong side (on the red side).

However, for the two previous cases (for k=5 and k=10), as the number of neighbors is smaller, we can observe that the predicitions are more precise, because only a restricted number of close neighbors are taken into account, which allows to avoid oversimplication and to reduce the bias of the model.

K mean algorithm

Explaination of the code

Generation of the dataset

I first imported the modules that I needed for my code: numpy, matplotlib.pyplot, from sklearn.datasets import make_classification, math,random and from sklearn.datasets import make_blobs (for generating a dataset)

Like earlier, I used np.random.seed(0) to control the randomity of the generation of the dataset, which will make easier the discussion of the results.

X, _ = make_blobs(n_samples=100, n_features=2, centers=3, cluster_std=1, random_state=42) generates a dataset adapted for classification that contains 100 samples, 2 features, 3 centers, a standard deviation of cluster of 1, and a random state of 42.

Construction of the K-mean algorithm

Then, I defined the algorithm that will generate the K-mean clustering on the values of the dataset with a function that take for arguments k (the number of clusters) and the number of iterations. In this function, I first initialized the values of the centroids that will be random values. np.random.choice(X.shape[0], k, replace=False) generates a random sample of k indexes from the number of lines of X (X.shape[0]). At the end, centroids=X[np.random.choice(X.shape[0], k, replace=False)] will generate a numpy array, of k lines and two columns, that takes for values the k values from X that have for indexes the indexes generated by np.random.choice(X.shape[0], k, replace=False) (it keeps the portion of the array X that has the k values that have for indexes the indexes generated by np.random.choice). This method allows to initializate the values of centroids with values of the dataset, which allow to make sure that the alogrithm is consistent and works properly.

Then, I created a loop that works for the number iterations wanted (for n in range(n_iterations)). This loop has for purpose to determine the cluster of each points of the dataset. The first step of the loop is the creation of those clusters.
clusters=[[] for i in range(k)] create a list of k empty lists, and each empty lists represents a cluster (so this "clusters" list that contains the k empty lists generates in fact k clusters).

Then, I created a second loop (for i in X) in this first loop that has for purpose to assign each point i of the dataset to one of the k clusters created.
The first step of this second loop is the creation of an empty list named "distance". The idea of this second loop is to calculate the euclidian distance between the point i of the dataset and each point of the centroids array. Those distances will be added in the "distance" list. We then determine the minimum value of this list (that contains k elements, as there are k distances to calculate because there are k centroids), and we add the point i to the corresponding cluster (that has the minimum distance with this point).

To do that, in the second loop (for i in X) and after the creation of the "distance list", we create a third loop (for j in centroids) that iterates on the lines of the centroids array (to proceed to the calculation for each point of the centroids array). In this loop, we calculate the euclidian distance between the point i of the dataset X and the centroid j: euclidian=math.sqrt((j[0]-i[0])**2+(j[1]-i[1])**2). Then, we add this distance to the "distance" list: distance.append(euclidian). Once the loop is done, we determine the index of

the minimum value of the distance list that we stock in a variable: smallest=np.argmin(distance). Finally, we assign the point i of the dataset X to the cluster that has the minimum distance, so to the element of the cluster list (that contain all the clusters in the form of lists) that has for index the index of the minimum values of the "distance" list: clusters[smallest].append(i).

After that this second loop (for i in X) is done and that all elements of the dataset are assigned to their corresponding clusters, we need to update the values of the centroids, which are the mean of the values contained in their corresponding clusters. To do that, and to avoid any calculation and programmation mistakes, we first convert each cluster in numpy arrays (because at this moment, each clusters is a list):

```
for t in range(len(clusters)):
        clusters[t]=np.array(clusters[t])
```

for t in range(len(clusters)) means that we apply this transformation to each cluster contained in the "clusters" list that contain len(clusters) clusters.

We can now update the value of each centroids:

```
for l in range(k):
        centroids[l]=np.mean(clusters[l], axis=0)
```

The new value taken by each centroid is the mean of the values of its corresponding cluster. It will have for coordinate x1 the mean of all x1 coordinates of the elements of the corresponding cluster, and for coordinate x2 the mean of all x2 coordinates of the elements of the corresponding cluster. Therefore, for each cluster (which are now arrays), we need to calculate the mean of each of its columns (columns represent x1 and x2) and that is why in np.mean(clusters[l], axis=0) we precise the option axis=0 : this means that we calculate the mean according to the axis 0, so for each column.

Finally, this k_mean_clustering algorithm returns the final clusters obtained and their corresponding centroids.

Predicition for a new point

In order to make the prediction for a new point, we create a new function that takes for arguments the new_point, centroids and clusters.
We first respectively stock the values of the centroids and the clusters in different variables. Then, the logic is the same as in the previous algorithm. We first create a "distance" empty list. Then, for each centroids, we calculate the euclidian distance between the new point and the centroid and we add this distance to the "distance" list. We do that in a loop that iterates on the values of the centroids array:

```
for j in centroids:
    euclidian=math.sqrt((j[0]-new_point[0])**2+(j[1]-new_point[1])**2)
    distance.append(euclidian)
```

Once the loop is done, we determine the index of the minimum value in the "distance" list that we stock in a variable: smallest=np.argmin(distance). Finally, the function return this value, which corresponds to the number of the cluster to which the new point should belong.

When a value of k and of the number of iteration is defined, we can stock the clusters and centroids resulting from the k_mean_clustering algorithm in variables clusters and centroids. For example: clusters,centroids=k_mean_clustering(5,1000). We can then proceed to the prediction of the cluster of the new point by launching the prediction function that will take for arguments new_point, centroids and clusters (the variables that contains respectively the

centroids and the clusters resulting from the k_mean function):
final_prediction=prediction(new_point,centroids,clusters)

Definition of the cost function

Finally, we can calculate the cost of the function. We first create a "cost" empty list. Then, for each cluster, we calculate the sum of the difference between each point of the cluster and the corresponding centroid of the cluster put in squared:

```
cost=[]
for i in range(k):
    cost_cluster=np.sum((clusters[i]-centroids[i])**2)
    cost.append(cost_cluster).
```

In this calculation, clusters[i] is an array that have len(cluster[i]) lines and two columns, and centroids[i] is an array of dimension that has for value 2. Therefore, this calculation is possible thanks to the broadcasting. np.sum allows to calculate the final sum of all the elements contained in the array of the differences between the points of the cluster and the corresponding centroid. Then, we add this cost of the i-th cluster to the "cost" list, making in sort that at the end, we have the cost for each of the k clusters.

We can finally calculate the distortion of our model with: distortion=np.sum(cost)/X.shape[0] np.sum(cost) allow to calculate the total cost of our model, by summing the cost of each of the cluster contained in the "cost" list. Then, we devide this total cost by the number of samples, so X.shape[0], in order to find the mean cost.

Reconstruction of the model with Sklearn

In order to determine if our program works properly, we can compare with the model of sklearn. To do that, we first import the module used : from sklearn.cluster import Kmeans. Then, we defined the model that we will use: kmeans = KMeans(n_clusters=k, random_state=42). We then execute the model on the data of our dataset X : kmeans.fit(X). We then determine the centroids finally found by the model:
centroids_ = kmeans.cluster_centers_. Then, we calcualte te inertia:
inertia = kmeans.inertia_, that we devise by the number of samples in order to find the distortion: distortion = inertia/X.shape[0].

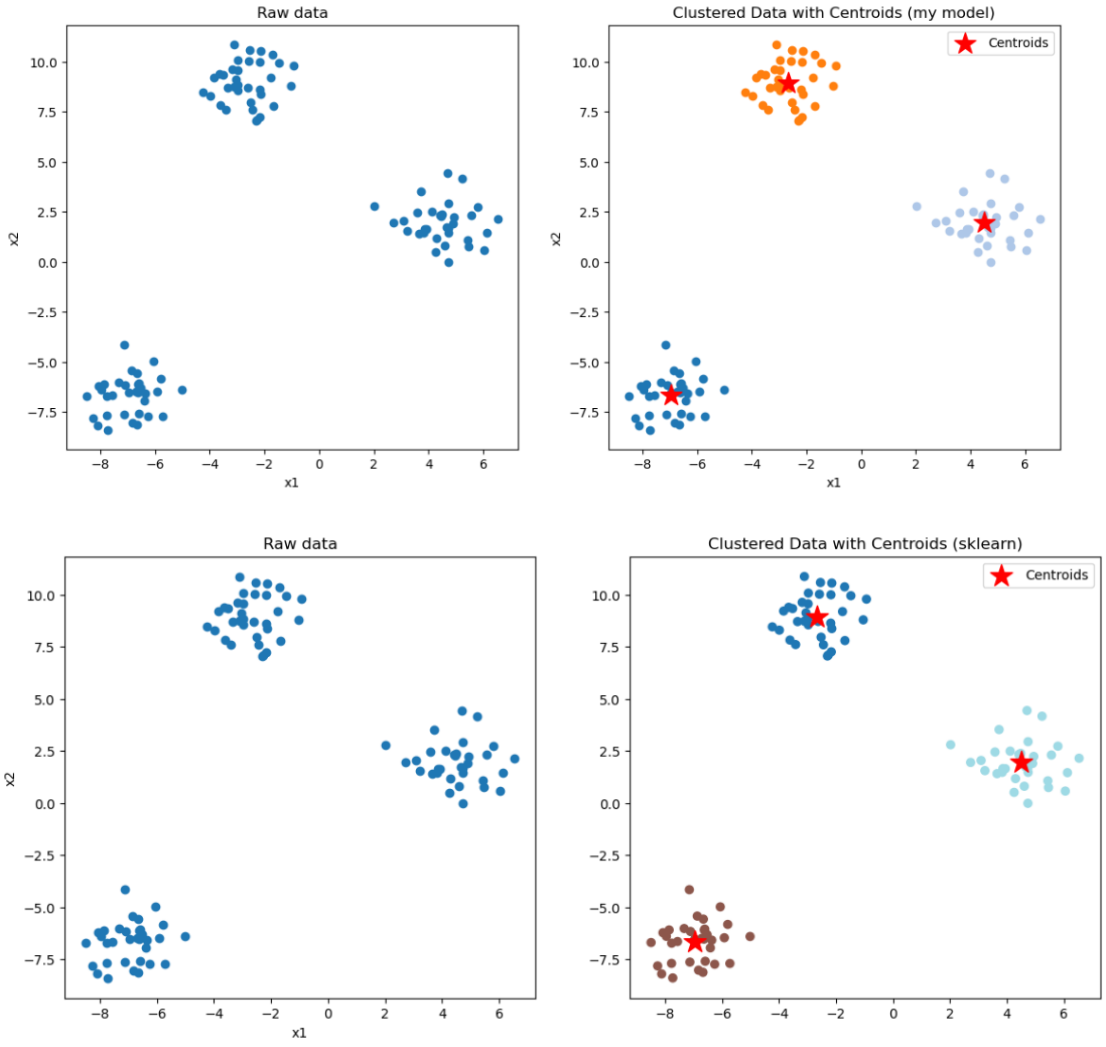
Discussion of the results:

Variation of the value of K (number of clusters)

I first tried my model with different number of cluster (value of K), but all the initial values for the centroids are generated randomly with a np.random.seed(0).

With k=3

I first tried my model with a number of clusters (k) equal to 3. For this case, obtained the following coordinates for my centroids (-6.952, -.676), (4.5, 1.939) and (-2.668, 8.936) and a distortion of 1.719. With the same number of clusters, the sklearn model found the following coordinates for the centroids: (-6.952, -6.676), (4.5, 1.939) and (-2.668, 8.936) and a distortion of 1.719. Therefore, the coordinates for the centroids and the distrotion found by my model are the same as those found by the sklearn model, which shows that my model performs well.



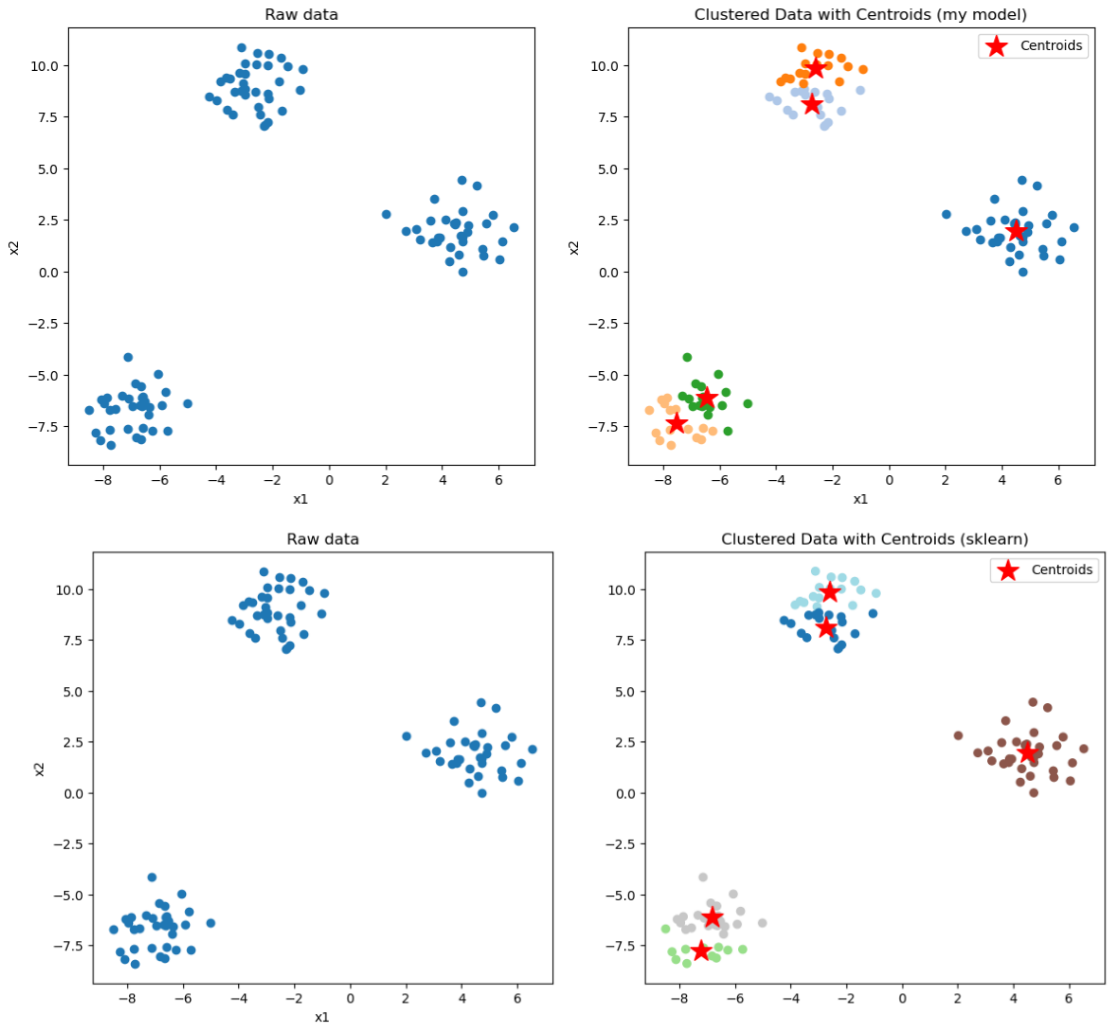
Both graph are really similar and present the same centroids.

With k=5

I then try with a value of k=5 and for my model, I found the following coordinates for my cen troids: (4.49951001 1.93892013), (-2.72550354 8.12321323), (-2.60289185 9.84987659), (-7.53386167 -7.33385427), (-6.46658291 -6.12818538) and a distortion of 1.253.
The sklearn model, for a k=5, found the close coordinates for the centroids as those found with our model:

- [-2.72550354 8.12321323]
- [-7.23000553 -7.78364976]
- [4.49951001 1.93892013]
- [-6.81256166 -6.12250016]
- [-2.60289185 9.84987659]

However, the distortion of the distortion model is slightly inferior with a value of 1.250. Therefore, in this case again, my model performed well and we can observe that with the graphs that are really similar:



With number of clusters such as k=3 or k=5, we can see that the model and the sklearn model works well and in a similar manner, as they positioned the centroids at similar positions. This can be explained by the fact that, as the number of clusters is small in those cases, there is less possibilities that are available to distribute the points among the clusters.

With k=10

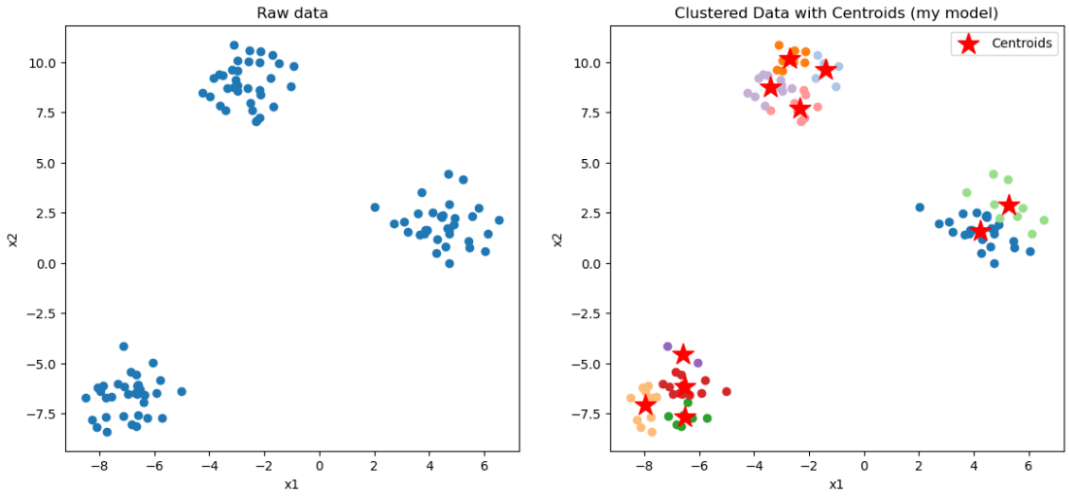
Finally, with a k=10, these are the coordinates for the centroids found by my model:

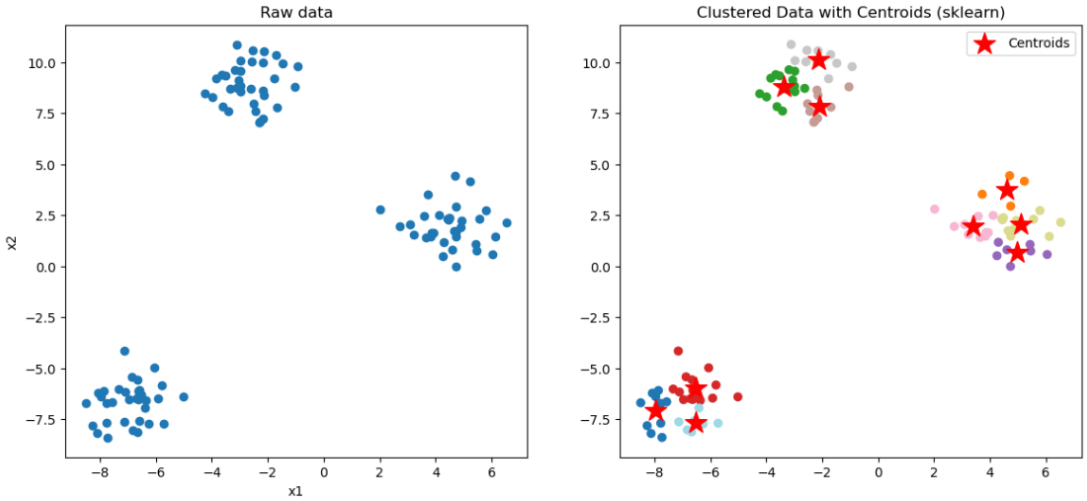
- [4.21590645 1.58564637]
- [-1.38382692 9.6143955]
- [-2.71218748 10.1574605]
- [-7.95554432 -7.08933118]
- [-6.51141471 -7.68117609]
- [5.25578614 2.88098348]
- [-6.50494429 -6.17931067]
- [-2.34822918 7.70519735]
- [-6.60092553 -4.57162852]
- [-3.41288635 8.76145215]

The coordonates found by the sklearn model are different:

- [-7.95554432 -7.08933118]
- [4.59606399 3.76623897]
- [-3.36910997 8.79481445]
- [-6.51694195 -5.9783504]
- [4.97267452 0.69257797]
- [-2.08448776 7.83703373]
- [3.40178376 1.94440845]
- [-2.1408294 10.13603431]
- [5.10608458 2.05227316]
- [-6.51141471 -7.68117609]

Moreover, the distortion calculated by my model is of 0.742, which is higher than the distortion calculated by the sklearn model which is of 0.596. However, the model still performs well and we can see that with the graphs:





However, on the graphs, we can see that the centroids are not positionned at the same place. This can be explained by the fact that the number of centroids increased to become a larger number (here 10), so it will be more difficult for my model to do the same predictions and have the same results as the sklearn model, because more possibilities of clusters for data are available.

Variation of the initial values of centroids

Initialization of centroids with a random state of 1

Let's try the model with different initial values for centroids, with a k=4.

First, let's try with np.random.seed(1).

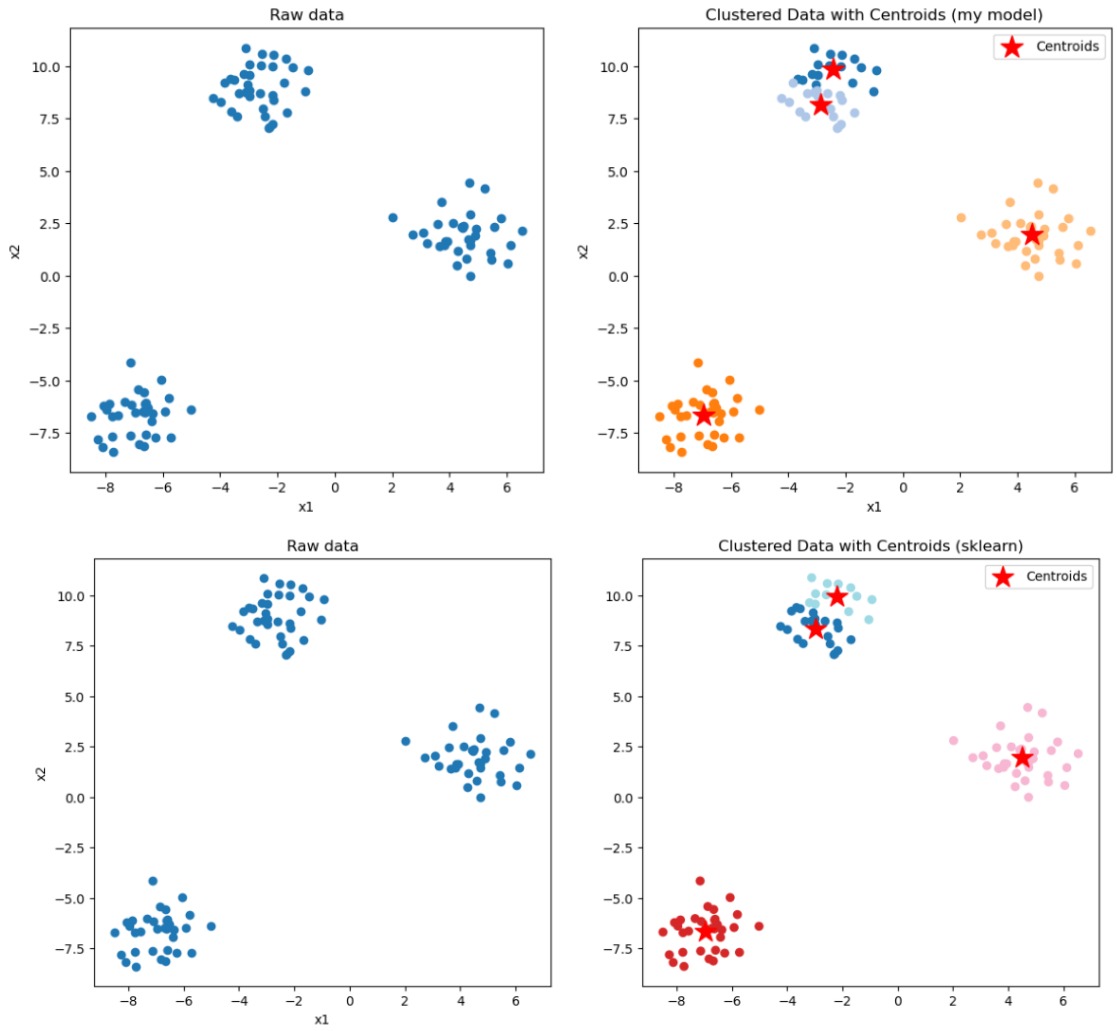
The coordinates for the centroids obtained by model are :

- [-2.42827717 9.82346174]
- [-2.88071658 8.14669309]
- [-6.95170962 -6.67621669]
- [4.49951001 1.93892013]

The coordinates found by the sklearn model are close:

- [-2.95660471 8.30877152]
- [-6.95170962 -6.67621669]
- [4.49951001 1.93892013]
- [-2.20127958 9.94858935]

However, the distortion of my model is slightly higher, with 1.463 against 1.457.
The well performance of my model can be observed with the graphs that are really similar:



Initialization of centroids with a random state of 56

We can now try new initialization values for the centroids, generated randomly with np.random.seed(56)

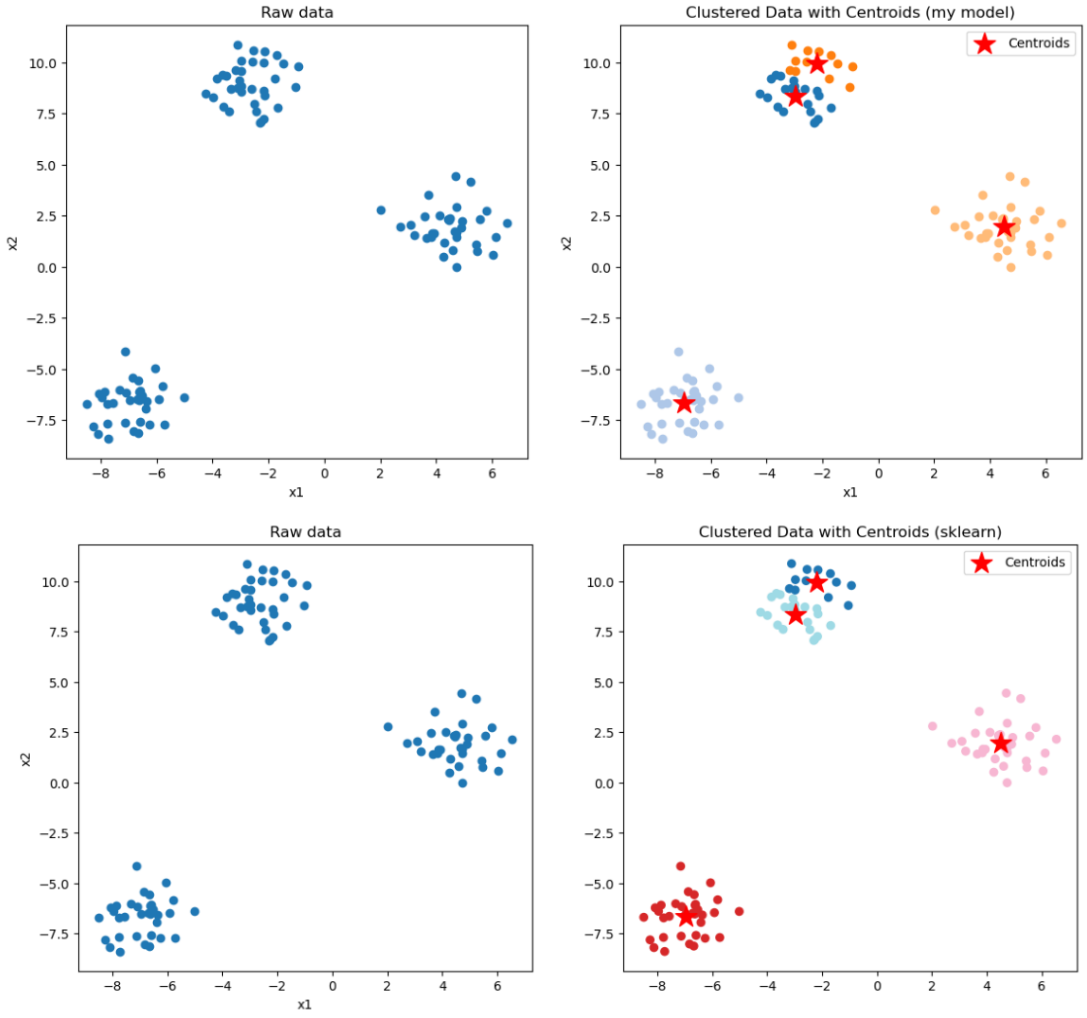
Compared to the previous case, the coordinates of the centroids change really slightly but globally stayed the same:

- [-2.95660471 8.30877152]
- [-6.95170962 -6.67621669]
- [-2.20127958 9.94858935]
- [4.49951001 1.93892013]

Here are the coordinates found by the sklearn model and that remain close:

- [-2.20127958 9.94858935]
- [-6.95170962 -6.67621669]
- [4.49951001 1.93892013]
- [-2.95660471 8.30877152]

However, in this case, the distortion is the same with a value of 1.457. This good performance of the model can be shown with the graphs that are really similar:



Therefore, even when the initialization of centroids change, the results obtained by my model and the sklearn model stay similar and consitent. This can be explained by the fact that the number of clusters used is small (here it is 4), so less possibilities are available for distributing the points among different clusters. However, this consistency also shows that the model works well as, whatever the initialization of centroids is, the results obtained will be really similar.