assignment_03

March 16, 2024

Please fill in your name and that of your teammate.

You: **Ahonon Gobi Parfait**

Teammate:

1 Introduction

Welcome to the third lab. There is much to go through today so we will keep extra concepts to a minimum. There is no new library introduced at this lecture as we will keep using `numpy` for the heavy lifting, `scikit-learn` for the algorithms, and `seaborn` / `matplotlib` for plotting. Careful about reusing variable names in the notebook and computing cells out of order: frequent calls to `Kernel -> Restart and Run All` can save you from headaches.

The assignment starts getting math-heavy. Here's a new tool to aid you with the debugging. Explicitly `import IPython` at the beginning of a notebook (or Python file) to have access to the computational Python kernel. You can then call `IPython.embed()` at an arbitrary place in your code (say, inside a loop) and it will pause the computation and drop you into an interactive console. You can then evaluate Python code in the context where it was called. Here is an example:

```
import numpy as np
for i in range(10):
    guess = np.random.normal()
    function_that_fails_because_of(guess, i)
```

Let's say your function fails for `i==9`, how would you find the error? Typically you may want to edit the code and print `guess` and `i` to see what is happening, but it is slow and passive. What if you want to try to pass `i+1` and see if that works? What if you want to try a few other random numbers with the same `i`? Enter `IPython.embed()`:

```
import numpy as np
import IPython
for i in range(10):
    guess = np.random.normal()
    if i==9: IPython.embed()
    function_that_fails_because_of(guess, i)
```

If you execute this code, the cell output will show an interactive console in your output cell. Here you can send commands to be interpreted by the Python kernel of the notebook. You could then try something like the following lines for example (one at a time):

```
i #=> prints value of i
guess  #=> prints value of guess
function_that_fails_because_of(guess, i) #=> fails and shows you the error
function_that_fails_because_of(guess, i+1) #=> change parameters: will it work?
function_that_fails_because_of(guess, i-1) #=> what about this one?
guess = np.random.normal() #=> overwrites the value of `guess` in the kernel
function_that_fails_because_of(guess, i) #=> will this work this time?
```

As you can see you can test your code in the context of the function (or, here, loop), find the code that works, then you can go ahead and copy+paste in your actual code. If you need to exit the console and resume the computation (with whatever change you executed, as the kernel is the same) just type `exit()`.

Bonus: you can ask the kernel to drop you into a *debugger* session every time you get an error. This can be tricky with Jupyter Notebooks, so use with caution, but can also be a lifesaver if you are willing to learn about postmortem debugging. You do that by adding the following lines on top of your code (need to execute them only once):

```
# DEBUG: uncaught exceptions drop you into ipdb for postmortem debugging
import sys, IPython; sys.excepthook = IPython.core.ultratb.ColorTB(call_pdb=True)
```

After this line, if you encounter an error or uncaught exception in your code, rather than terminating you will be dropped in an `ipdb` (fancier version of `pdb`) console where you can interrogate the program about the conditions causing the crash.

Good hunting!

1.0.1 How to pass the lab?

Below you find the exercise questions. Each question awarding points is numbered and states the number of points like this: **[0pt]**. To answer a question, fill the cell below with your answer (markdown for text, code for implementation). Incorrect or incomplete answers are in principle worth 0 points: to assign partial reward is only up to teacher discretion. Over-complete answers do not award extra points (though they are appreciated and will be kept under consideration). Save your work frequently! (`ctrl+s`)

You need at least 22 points (out of 33 available) to pass (66%).

2 1. Fundamentals

Let's make sure some of the core points are clear before addressing the specific algorithms.

1.1 [1pt] Write the equation of the Gaussian density. Use Latex inside the Markdown cell. I suggest you type it out rather than copy+paste from the Internet: the goal of this question is to *force* you to read one term at a time, and understand which is clear to you and which is not. For example, the equation you will write here a norm, while our later applications of the formula do not, since we will actually be using the Gaussian Probability Density Function (PDF) equation instead. Do you understand why? Did you study this before?

$$f(x, \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Where : x is the random variables μ is the mean(average) of the distribution σ^2 is the variance of the distribution

1.2 [1pt] Explain why we maximize the log-likelihood rather than the likelihood. In particular, what is the advantage in using the log **rather than another operation?** R: Maximizing the log-likelihood rather than the likelihood is advantageous because it simplifies computations and helps avoid numerical underflow or overflow issues. The logarithm function compresses large ranges of values into a more manageable scale, making it easier to work with and compute gradients during optimization.

1.3 [1pt] Why the equation maximizing the log-likelihood of a Gaussian does not include the parameter σ? R: The equation maximizing the log-likelihood of a Gaussian does not include the parameter σ because it is a constant in the optimization problem. The log-likelihood function is maximized with respect to the parameters of the distribution, and σ is a parameter of the distribution, not a variable to be optimized.

1.4 [1pt] Explain the meaning of i.i.d. (in English), using the simplest words you can. R: i.i.d. stands for independent and identically distributed. It means that the random variables in a sample are independent of each other and have the same probability distribution. In other words, the outcome of one random variable does not affect the outcome of another, and they all come from the same underlying distribution.

1.5 [2pt] Write the equation of the Bayes' Rule (use Latex). Then write below how to read it in English.

$$P(A|B) = \frac{P(B|A) \times P(A)}{P(B)}$$

Reading: The probability of A given B is equal to the probability of B given A times the probability of A divided by the probability of B

3 2. Linear Regression

2.1 [2pt] Explain the meaning of $y_i = \langle w, x_i \rangle + \epsilon_i, \quad \epsilon \sim \mathcal{N}(0, \sigma^2)$ (in English). Utilize the word 'prototype'. R: This equation is a linear relationship between a dependant variables y_i with an independant variables x_i with an additional noise ϵ_i . The termes prototypes refers to the ideal value or true values of the dependant variables y_i for a sets of features values x_i

Enough theory, let's get our hands in there. Since we are working with a regression task, let us generate some data from an underlying linear function with some noise.
[think: the process below is (correct but) unnecessary convoluted: would you be able (yet) to simplify it? Always prefer simpler code!]

[1]:
```
import numpy as np
trg_fn = lambda x: 2*x - 1 # hi I'm lambda, remember me?
some_noise = lambda: np.random.normal(0,2) # Gaussian noise with mu=0, sigma=2
# Below we add a 1 to every row as the bias (constant) input
# Think about each part and discuss if you do not understand something yet!
```

```
data = np.array([[x, 1, trg_fn(x) + some_noise()] for x in np.linspace(-10, 10,␣
↪50)])
*x, y = data.transpose() # easier using splat and numpy: do you understand it?
x = np.array(x).transpose() # back to *rows* with input features and bias input
```

2.2 [1pt] Write a (Python) function that takes a data point in input and returns the squared error Loss. Test it by using a constant prediction model $y = 1$ and compute the Risk over all the data. here is the formular : Squared Error Loss $= \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$

[2]:
```
def square_error_loss(data_points):
    sum_squared_errors = 0

    for data_point in data_points:
        y_actuel = data_point[-1]
        y_predicted  = predict(data_point)

        squared = (y_actuel - y_predicted) **2
        sum_squared_errors += squared

    mean_squared_error = sum_squared_errors/len(data_points)
    return  mean_squared_error

def predict(data_point):
    return 1
```

- Numpy's linear algebra library provides matrix inversion, but you should use instead the pseudo-inverse to cope with singular covariance matrices: `np.linalg.pinv()`
- Numpy's array provides inner product with the function `dot()`
- Typically `dot()` will find the right direction for one-dimensional arrays, which means that you should never need to transpose them
- Using `dot()` with matrices instead *always* requires you to `transpose()` to the right orientation! Write the math and keep track of what you are doing.
- Remember that matrix product is not commutative: `A.dot(B)` is NOT equal to `B.dot(A)`. Refresh also how `A.dot(B)` requires the number of columns of `A` to be the same as the number of rows of `B`, and the result will have the same number of rows of `A` and the number of columns of `B`.
- Linear regression uses a closed-form solution, **not a loop**, and you will not use the implementation of the loss above in the rest of the assignment (because it is implicit in the algorithm's solution).

2.3 [2pt] Write a function that takes in input a list of data points and a list of labels, and returns the w vector using the closed-form solution of Linear Regression. Test it on the data above and print the computed w.

[3]:
```
def linear_regression(data_points, labels):
    x = np.array([data_point[:-1] for data_point in data_points])
    #y = np.array([data_point[-1] for data_point in data_points])
```

```
    y = np.array(labels)
    x_pseudo_inv = np.linalg.pinv(x)
    w = x_pseudo_inv.dot(y)
    return w

w = linear_regression(data, y)
print(w)
```

```
[ 2.08598218 -1.02708754]
```

2.4 [2pt] Predict the labels for all points using your Linear Regression implementation and the `w` vector from the previous question. For each data point, print the triplet of `(label, prediction, loss)`. Then compute and print the (squared error) Risk over the dataset. Remember that you need to build the linear model (careful handling the bias), then you are doing *regression* not *classification*, which means you are predicting the value $\hat{y}$ based on your x. You should get back m and q close to 2 and -1, and the risk roughly proportional to the square of the expected error multiplied by the number of points (or close below that).
[think: what is the expected error? Do you understand why? You can generate the points and compute the risk multiple times to verify your hypothesis]

[4]:
```
# predictions for all data points using linear regression
predictions  = np.dot(x, w)
# squared error loss for each data point
losses = (y - predictions)**2
# risk
risk = np.mean(losses)

# Print the triplet (label, prediction, loss) for each data point
for i in range(len(data)):
    print(f"Data point {i+1}: (label={y[i]}, prediction={predictions[i]},␣
 ↪loss={losses[i]})")
print(f"Risk: {risk}")
```

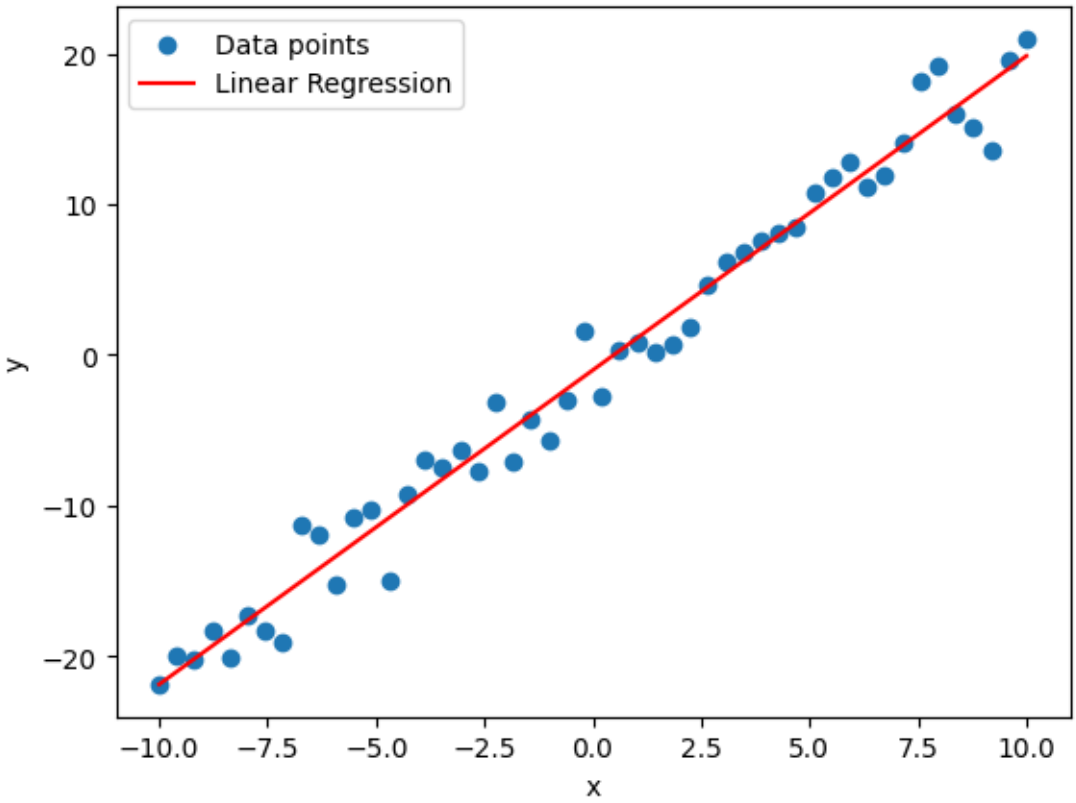
```
Data point 1: (label=-21.84462353076772, prediction=-21.886909313147445,
loss=0.0017880873914654727)
Data point 2: (label=-20.017452375721824, prediction=-21.035488016413943,
loss=1.0363965657194136)
Data point 3: (label=-20.273480822024005, prediction=-20.184066719680445,
loss=0.007994881697904773)
Data point 4: (label=-18.28317031108592, prediction=-19.332645422946943,
loss=1.1013980104157026)
Data point 5: (label=-20.098565769110607, prediction=-18.481224126213444,
loss=2.6157939898492937)
Data point 6: (label=-17.241853746843866, prediction=-17.629802829479946,
loss=0.1505044907181761)
Data point 7: (label=-18.328754204287463, prediction=-16.778381532746447,
loss=2.403655420661227)
```

```
Data point 8: (label=-19.11506353954248, prediction=-15.926960236012947,
loss=10.164002673975915)
Data point 9: (label=-11.266038464966632, prediction=-15.075538939279449,
loss=14.512293863789573)
Data point 10: (label=-12.00500349129056, prediction=-14.22411764254595,
loss=4.924467616301929)
Data point 11: (label=-15.247284714810423, prediction=-13.37269634581245,
loss=3.51408155318248)
Data point 12: (label=-10.847499741295257, prediction=-12.52127504907895,
loss=2.8015237809463955)
Data point 13: (label=-10.313575872435754, prediction=-11.669853752345452,
loss=1.8394896875323439)
Data point 14: (label=-15.061373687544464, prediction=-10.818432455611951,
loss=18.002550297632986)
Data point 15: (label=-9.269493004775526, prediction=-9.967011158878451,
loss=0.48653157530315144)
Data point 16: (label=-6.923067344354758, prediction=-9.115589862144951,
loss=4.807154991017049)
Data point 17: (label=-7.436712961607535, prediction=-8.264168565411453,
loss=0.6846827762665059)
Data point 18: (label=-6.327311123499216, prediction=-7.412747268677955,
loss=1.1781716252604804)
Data point 19: (label=-7.778426196425441, prediction=-6.561325971944455,
loss=1.4813329564316668)
Data point 20: (label=-3.184327807892655, prediction=-5.709904675210954,
loss=6.378538512733311)
Data point 21: (label=-7.041779780369002, prediction=-4.858483378477455,
loss=4.766783178512576)
Data point 22: (label=-4.238648735501638, prediction=-4.007062081743957,
loss=0.05363237819868)
Data point 23: (label=-5.739411393018788, prediction=-3.155640785010455,
loss=6.675870554807751)
Data point 24: (label=-3.003178573928197, prediction=-2.3042194882769564,
loss=0.4885438034144184)
Data point 25: (label=1.6139263145924247, prediction=-1.452798191543458,
loss=9.404799196534373)
Data point 26: (label=-2.7024017103052405, prediction=-0.6013768948099596,
loss=4.414305275326979)
Data point 27: (label=0.2638818756114715, prediction=0.25004440192353883,
loss=0.00019147567806422825)
Data point 28: (label=0.8360960593041952, prediction=1.1014656986570412,
loss=0.0704210454902596)
Data point 29: (label=0.16236969626553766, prediction=1.9528869953905397,
loss=3.205952198465892)
Data point 30: (label=0.6574369079347213, prediction=2.804308292124038,
loss=4.609056740250953)
Data point 31: (label=1.8459231305757102, prediction=3.6557295888575405,
loss=3.2753994164386224)
```

```
Data point 32: (label=4.67292591137513, prediction=4.507150885591039,
loss=0.02748135917371614)
Data point 33: (label=6.205469131341964, prediction=5.358572182324537,
loss=0.7172344422550255)
Data point 34: (label=6.804419257442849, prediction=6.209993479058036,
loss=0.3533420060083906)
Data point 35: (label=7.596173261242606, prediction=7.061414775791533,
loss=0.2859666377619247)
Data point 36: (label=8.06225976919814, prediction=7.912836072525035,
loss=0.022327441127456243)
Data point 37: (label=8.466606878448728, prediction=8.764257369258534,
loss=0.08859581467931847)
Data point 38: (label=10.810091119386493, prediction=9.615678665992034,
loss=1.4266211088237721)
Data point 39: (label=11.750773564738985, prediction=10.467099962725536,
loss=1.647817916506183)
Data point 40: (label=12.85760279874944, prediction=11.318521259459034,
loss=2.3687719845845243)
Data point 41: (label=11.074329333883657, prediction=12.169942556192533,
loss=1.2003683328980375)
Data point 42: (label=11.883337513947438, prediction=13.021363852926031,
loss=1.2951039482090188)
Data point 43: (label=14.078391799229566, prediction=13.87278514965953,
loss=0.04227409434741574)
Data point 44: (label=18.126418911914822, prediction=14.724206446393028,
loss=11.575049660551885)
Data point 45: (label=19.17440174809348, prediction=15.575627743126532,
loss=12.95117433882585)
Data point 46: (label=15.999193824491732, prediction=16.42704903986003,
loss=0.183060085317853)
Data point 47: (label=15.13638562392046, prediction=17.27847033659353,
loss=4.588526916267659)
Data point 48: (label=13.609383116932761, prediction=18.129891633327027,
loss=20.434997246793085)
Data point 49: (label=19.578818367658364, prediction=18.981312930060525,
loss=0.35701274795898486)
Data point 50: (label=20.96750582828887, prediction=19.832734226794027,
loss=1.2877065875591729)
Risk: 3.5182148257918966
```

2.5 [2pt] Plot the data and the model. You should be able to partially reuse the printing code from the last lab (particularly plot and params-to-boundary conversion), but feel free to customize it as you need. I will keep repeating this for a while more: careful with the bias!
(Think: you can plot the model's predictions very easily if you use linear algebra, do you understand what is `x.dot(w)`?)

[5]:
```
import matplotlib.pyplot as plt

# Plot the data
plt.scatter(x[:,0], y, label='Data points')
# Plot the model
plt.plot(x[:,0], predictions, label='Linear Regression', color='red')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.show()
```



2.6 [2pt] Find Linear Regression in scikit-learn and train a model on the data. The input to the `fit()` function should be a matrix and a vector, so try forcing `actual_x` into a $n \times 1$ matrix by using `actual_x.reshape((-1, 1))`. If you want to predict the outputs, you simply need to pass the same data matrix to the method `predict()`.

[6]:
```
from sklearn.linear_model import LinearRegression

# create linear regression object
model = LinearRegression()
# reshaping
```

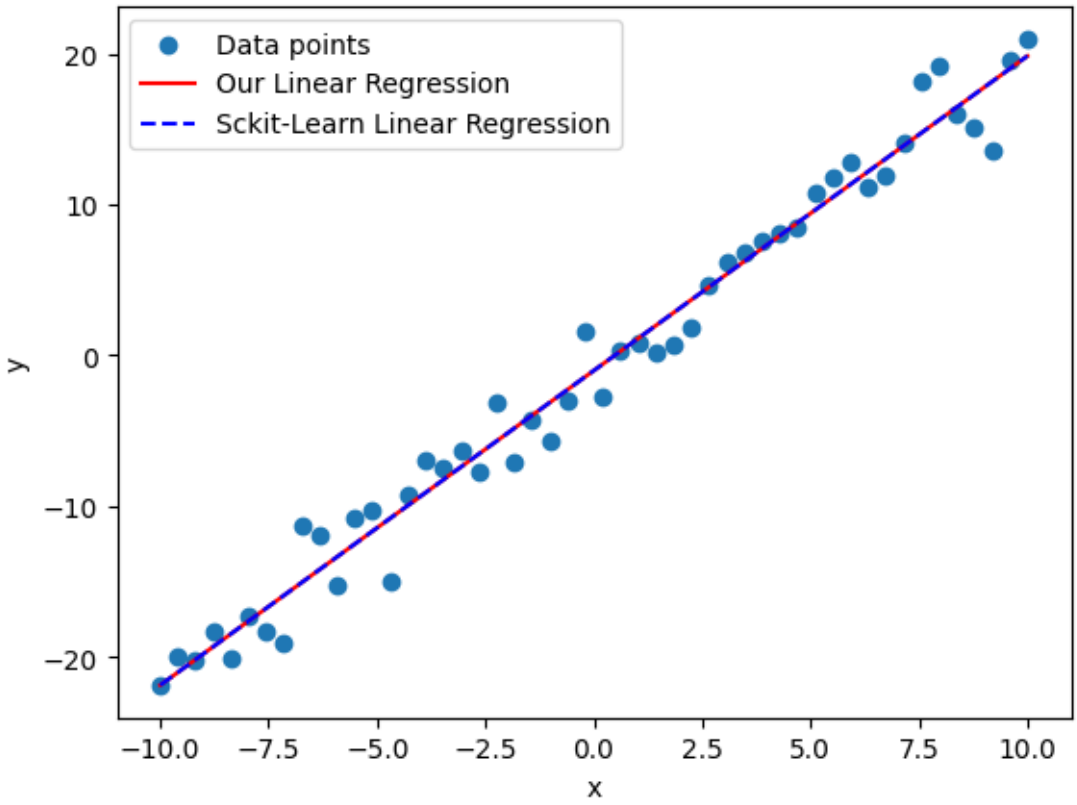
```
X_column_vector = x[:,0].reshape((-1,1))
# fit the model to the model
trained =  model.fit(X_column_vector,y)
# make the prediction on the X_column_vector
predictions_skl = model.predict(X_column_vector)
# Show it to see ? Let us do it.
print("We find out:", predictions_skl)
```

```
We find out: [-21.88690931 -21.03548802 -20.18406672 -19.33264542 -18.48122413
 -17.62980283 -16.77838153 -15.92696024 -15.07553894 -14.22411764
 -13.37269635 -12.52127505 -11.66985375 -10.81843246  -9.96701116
  -9.11558986  -8.26416857  -7.41274727  -6.56132597  -5.70990468
  -4.85848338  -4.00706208  -3.15564079  -2.30421949  -1.45279819
  -0.60137689   0.2500444    1.1014657    1.952887     2.80430829
   3.65572959   4.50715089   5.35857218   6.20999348   7.06141478
   7.91283607   8.76425737   9.61567867  10.46709996  11.31852126
  12.16994256  13.02136385  13.87278515  14.72420645  15.57562774
  16.42704904  17.27847034  18.12989163  18.98131293  19.83273423]
```

2.7 [1pt] Plot in a single figure: (i) the data points as a scatterplot; (ii) the model you learned using your implementation of Linear Regression; (iii) the model you trained using the scikit-implementation. Careful, it may be that you plot both but they are exactly the same, so they are superimposed and you only see one line. To verify this you can try plotting them with different colors and different thickness, or changing the linestyle (e.g. one normal line and one dashed line). A list of available linestyles can be found in the matplotlib documentation.

To get the parametrization of the sklearn implementation use the following:
`w_skl = [trained.coef_[0], trained.intercept_]`
Do you understand what this does? Can you see how this corresponds to what we did last week? In future assignments you will need to find the model parametrization by yourself!

[7]:
```
w_skl = [trained.coef_[0], trained.intercept_]  # w_skl = [slope, intercept]

plt.scatter(x[:,0], y, label='Data points')

# Plot the model
plt.plot(x[:,0], predictions, label='Our Linear Regression', color='red')
#
plt.plot(x[0:,0],x.dot(w_skl), label='Sckit-Learn Linear Regression',␣
 ↪linestyle="dashed", color='blue')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.show()
```



4 3. Linear Discriminant Analysis

We switch now into *binary classification*. Let's load the iris dataset once again for this exercise, and carefully selecting the data to have an easy binary classification problem (for now). Notice we are **not** using classes $-1, +1$ anymore, because we do not need to compute a Margin here.

[8]:
```
import numpy as np
from sklearn.datasets import load_iris
iris_x, iris_y = load_iris(return_X_y=True) # print these points to understand␣
 ↪them!
x1 = np.array([r[0] for r in iris_x]) # first feature
x2 = np.array([r[2] for r in iris_x]) # third feature
print(x.shape)
x = np.array([x1, x2]).transpose() # numpy gives us transpose() for free
# Reduce the three classes into two for binary classification {1, 2}
y = np.array([1 if y in [1,2] else 2 for y in iris_y])
```

```
(50, 2)
```

To solve LDA we need to find the parametrization $\theta_y = (\mu_y, \Sigma, \pi_y)$. Since μ and π are class-dependent, remember to first split the input data based on which class it belongs to.

3.1 [1pt] Write a (Python) function that takes a dataset (inputs and labels) in input and returns a dictionary hashing each of the m classes to the a list of the points belonging to that class. We will call this `partition` in the next questions. Hint: the method `dict.get(<key>, <def>)` can be used to fetch values from a dictionary same as `dict[<key>]`, but when the key is not found it returns the second argument, which is the *default value*... what if you pass an empty list...

[9]:
```
def partition_dataset(inputs, labels):
    # create a dictionary to store the partition
    partition = {}
    # loop through the inputs and labels
    for data_point, label in zip(inputs,labels):
        if label not in partition:
            partition[label] = []
        partition[label].append(data_point)
    return partition

partition = partition_dataset(x, y)
print(partition)
```

```
{2: [array([5.1, 1.4]), array([4.9, 1.4]), array([4.7, 1.3]), array([4.6, 1.5]),
array([5. , 1.4]), array([5.4, 1.7]), array([4.6, 1.4]), array([5. , 1.5]),
array([4.4, 1.4]), array([4.9, 1.5]), array([5.4, 1.5]), array([4.8, 1.6]),
array([4.8, 1.4]), array([4.3, 1.1]), array([5.8, 1.2]), array([5.7, 1.5]),
array([5.4, 1.3]), array([5.1, 1.4]), array([5.7, 1.7]), array([5.1, 1.5]),
array([5.4, 1.7]), array([5.1, 1.5]), array([4.6, 1. ]), array([5.1, 1.7]),
array([4.8, 1.9]), array([5. , 1.6]), array([5. , 1.6]), array([5.2, 1.5]),
array([5.2, 1.4]), array([4.7, 1.6]), array([4.8, 1.6]), array([5.4, 1.5]),
array([5.2, 1.5]), array([5.5, 1.4]), array([4.9, 1.5]), array([5. , 1.2]),
array([5.5, 1.3]), array([4.9, 1.4]), array([4.4, 1.3]), array([5.1, 1.5]),
array([5. , 1.3]), array([4.5, 1.3]), array([4.4, 1.3]), array([5. , 1.6]),
array([5.1, 1.9]), array([4.8, 1.4]), array([5.1, 1.6]), array([4.6, 1.4]),
array([5.3, 1.5]), array([5. , 1.4])], 1: [array([7. , 4.7]), array([6.4, 4.5]),
array([6.9, 4.9]), array([5.5, 4. ]), array([6.5, 4.6]), array([5.7, 4.5]),
array([6.3, 4.7]), array([4.9, 3.3]), array([6.6, 4.6]), array([5.2, 3.9]),
array([5. , 3.5]), array([5.9, 4.2]), array([6., 4.]), array([6.1, 4.7]),
array([5.6, 3.6]), array([6.7, 4.4]), array([5.6, 4.5]), array([5.8, 4.1]),
array([6.2, 4.5]), array([5.6, 3.9]), array([5.9, 4.8]), array([6.1, 4. ]),
array([6.3, 4.9]), array([6.1, 4.7]), array([6.4, 4.3]), array([6.6, 4.4]),
array([6.8, 4.8]), array([6.7, 5. ]), array([6. , 4.5]), array([5.7, 3.5]),
array([5.5, 3.8]), array([5.5, 3.7]), array([5.8, 3.9]), array([6. , 5.1]),
array([5.4, 4.5]), array([6. , 4.5]), array([6.7, 4.7]), array([6.3, 4.4]),
array([5.6, 4.1]), array([5.5, 4. ]), array([5.5, 4.4]), array([6.1, 4.6]),
array([5.8, 4. ]), array([5. , 3.3]), array([5.6, 4.2]), array([5.7, 4.2]),
array([5.7, 4.2]), array([6.2, 4.3]), array([5.1, 3. ]), array([5.7, 4.1]),
array([6.3, 6. ]), array([5.8, 5.1]), array([7.1, 5.9]), array([6.3, 5.6]),
array([6.5, 5.8]), array([7.6, 6.6]), array([4.9, 4.5]), array([7.3, 6.3]),
array([6.7, 5.8]), array([7.2, 6.1]), array([6.5, 5.1]), array([6.4, 5.3]),
```

```
array([6.8, 5.5]), array([5.7, 5. ]), array([5.8, 5.1]), array([6.4, 5.3]),
array([6.5, 5.5]), array([7.7, 6.7]), array([7.7, 6.9]), array([6., 5.]),
array([6.9, 5.7]), array([5.6, 4.9]), array([7.7, 6.7]), array([6.3, 4.9]),
array([6.7, 5.7]), array([7.2, 6. ]), array([6.2, 4.8]), array([6.1, 4.9]),
array([6.4, 5.6]), array([7.2, 5.8]), array([7.4, 6.1]), array([7.9, 6.4]),
array([6.4, 5.6]), array([6.3, 5.1]), array([6.1, 5.6]), array([7.7, 6.1]),
array([6.3, 5.6]), array([6.4, 5.5]), array([6. , 4.8]), array([6.9, 5.4]),
array([6.7, 5.6]), array([6.9, 5.1]), array([5.8, 5.1]), array([6.8, 5.9]),
array([6.7, 5.7]), array([6.7, 5.2]), array([6.3, 5. ]), array([6.5, 5.2]),
array([6.2, 5.4]), array([5.9, 5.1])]}
```

3.2 [1pt] Write a function that takes the `partition` in input and returns a dictionary hashing each class to its corresponding prototype μ_y. Hint: function `dict.items()` returns a list of pairs `(key, value)` from the dictionary. Then a *dictionary comprehension* works same as a list comprehension, with a `for` loop that generates elements. Only this time you need to pass both a key and a value like this:

```
d = { the_key: compute_value(a, b) for a, b in another_dict.items() }
```

If you can use that then the answer is basically one line. If it is complicated instead, write explicit `for` loops, which are exactly equivalent (and perhaps even more readable:

```
d = {}
for a, b in another_dict.items():
    d[the_key] = compute_value(a, b)
```

Just find a style that is comfortable with you. May take a few weeks but you will get there.

[10]:
```
def prototype(partition):
    # create a dictionary to store the prototypes
    prototypes = {}
    # loop through the partition
    for class_label, data_points in partition.items():
        prototypes[class_label] = np.mean(data_points, axis=0)
    return prototypes

prototypes = prototype(partition)
print(prototypes)
```

```
{2: array([5.006, 1.462]), 1: array([6.262, 4.906])}
```

3.3 [1pt] Write a function that takes the `partition` in input and returns a dictionary hashing each class to its corresponding prior π_y.

[11]:
```
def prior(partition):
    # create a dictionary to store the priors
    prior = {}
    total_count = sum(len(point) for point in partition.values())
    for class_label, data_points in partition.items():
        prior[class_label] = len(data_points)/total_count
```

```
    return prior

prior = prior(partition)
print(prior)
```

```
{2: 0.3333333333333333, 1: 0.6666666666666666}
```

3.4 [4pt] Write a function that takes the `partition` in input and the class-wise center estimates (the means from above) and returns the corresponding Σ (one for all classes and all inputs). You may need to use `np.concatenate()` to join the $x_i - \hat{\mu}_{y_i}$ from each class. Print the `array.shape` (property not method so no `()`) to verify if your linear algebra is on point so far: the covariance matrix between all inputs should have as many rows (and columns) as the number of features (hint: that's 2). Example: `assert sigma.shape == (2,2)` should not raise an error.

[12]:
```
def corresponding_sum(partition,means):
    diffs = []
    for class_label, data_points in partition.items():
        diffs.extend([data_point - means[class_label] for data_point in␣
 ↪data_points])
    # Convert the list of differences to a NumPy array
    diffs = np.array(diffs)
    # covarience
    sigma = np.dot(diffs.T, diffs) / len(diffs)
    return sigma
# Test
sigma = corresponding_sum(partition,prototypes)
print(sigma)
#assert sigma.shape == (2,2)
```

```
[[0.33055867 0.30456133]
 [0.30456133 0.45969467]]
```

3.5 [5pt] Write the function for the decision boundary of LDA $f(x)$. You can find the logarithm in the `math` module: `from math import log` then just `l = log(a)`. You need to implement the equations for w and q from the slides: this cell will be very math heavy, be careful though and you should be able to get it right in few lines. Remember to check for the shape of w and b to verify if your matrix products are computed the right way. You want w to be of length 2, and b is a scalar.

[13]:
```
from math import log

# calculate the decision boundary
def calculate_decision_boundary(x,partition,prototypes,sigma,prior):
    w = np.dot(np.linalg.inv(sigma), (prototypes[1] - prototypes[2]))
    b = -0.5 * (np.dot(np.dot(prototypes[1].T, np.linalg.inv(sigma)),␣
 ↪prototypes[1]) - np.dot(np.dot(prototypes[2].T, np.linalg.inv(sigma)),␣
 ↪prototypes[2])) + log(prior[2]/prior[1])
```

```
    print(w)
    print(b)
    return np.dot(w.T, x) + b

# Test
# x = np.array([1, 2])
print(calculate_decision_boundary(np.array([1,␣
 ↪2]),partition,prototypes,sigma,prior))
```

```
[-7.96528842 12.76916894]
3.526253846831006
21.099303314898165
```

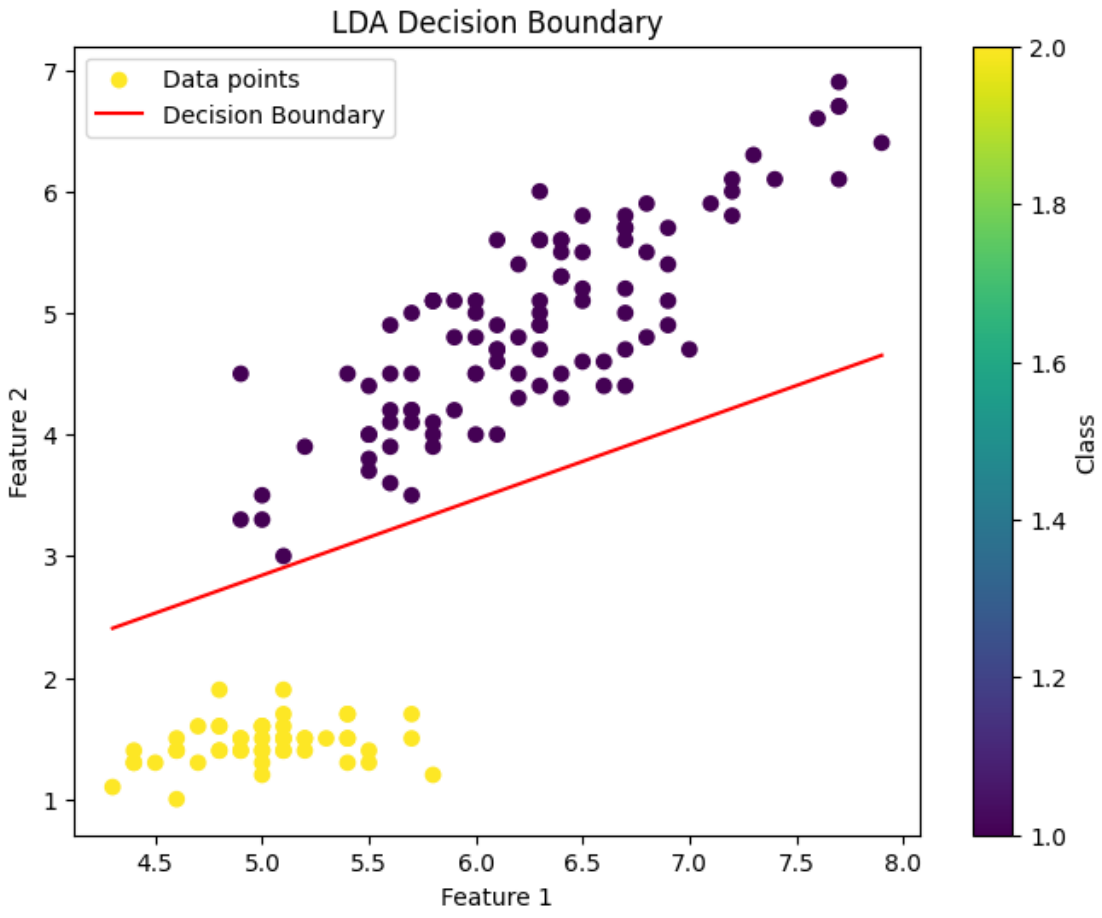
3.6 [2pt] Plot the LDA decision boundary on top of the data.

- Notice that we are using different data from the Linear Regression questions above, and that our model now generated w and b separately, so you **need** to adapt the plotting functions – actually it's quicker to just rewrite them.
- Plotting is one of few applications where generalizing your code just makes for a stunted replica of the original (better) interface, so specialize when needed but for different applications just write a new one rather than reusing your code.
- Remember that we already had a good plotting function for a classification problem just last week, why don't you check it out again?
- Finally, remember that you already have code that converts a parametrization from w and b to m and q (last assignment), you can simply copy+paste it here to simplify your generation of the model's points.

[14]:
```
def calculate_decision_boundary_params(partition, prototypes, sigma, prior):
    w = np.dot(np.linalg.inv(sigma), (prototypes[1] - prototypes[2]))
    b = -0.5 * (np.dot(np.dot(prototypes[1].T, np.linalg.inv(sigma)),␣
 ↪prototypes[1]) - np.dot(np.dot(prototypes[2].T, np.linalg.inv(sigma)),␣
 ↪prototypes[2])) + log(prior[2] / prior[1])
    return w, b

# call of the function to have w and b
w, b = calculate_decision_boundary_params(partition, prototypes, sigma, prior)
print("w:", w)
print("b:", b)


# Function to calculate the LDA decision boundary
def calculate_decision_boundary(x, w, b):
    return (-b - w[0] * x) / w[1]

# Parameters of the LDA decision boundary of the obtained w and b
# w = np.array([-7.91218649, 12.68404115])
# b = 3.4981245066483897
```

```
# Plot the data points
plt.figure(figsize=(8, 6))
plt.scatter(x[:, 0], x[:, 1], c=y, cmap='viridis', label='Data points')

# plot the LDA decision boundary
x_values = np.linspace(min(x[:, 0]), max(x[:, 0]), 100)
plt.plot(x_values, calculate_decision_boundary(x_values, w, b), color='red',␣
 ↪label='Decision Boundary')

plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('LDA Decision Boundary')
plt.colorbar(label='Class')
plt.legend()
plt.show()
```

```
w: [-7.96528842 12.76916894]
b: 3.526253846831006
```



3.7 [1pt] Find LDA on scikit-learn; train a model on the data and add it to the print above (data + model from your implementation). You ned to pass the correct `solver` parameter to the sklearn constructor, check the documentation to understand what I mean. If you do not the result should still look exactly the same as your implementation (because the data is linearly separable), but you should be aware of which technique your library uses and we have not gotten to SVD yet.
Remember to make sure that you can distinguish the two boundaries even if they overlap (e.g. use different colors). If you use our conversions from last exercise you should also see the printed values of m and q and they are likely to differ in the least significant digits even though the graph looks the same.
Also consider that LDA is a **multiclass method**, and so its parametrization is in principle a list for the many boundaries: you need to access the coefficients of the *first* (and here, only) boundary using `[trained_model.coef_[0], trained_model.intercept_[0]]`.

[15]:
```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

# Create LDA object
lda = LinearDiscriminantAnalysis()
# Train the model
trained = lda.fit(x, y)
# print the two trained_model coefficients and intercepts
print(trained.coef_[0], trained.intercept_[0])
```

```
[  7.85908457 -12.59891336] -4.856289527585768
```

4.1 At the end of the exercise

Bonus question reward no points! Answering this will have no influence on your scoring, not at the assignment and not towards the exam score. But solving it will reward you with skills that will make the next lectures easier, give you real applications, and will be very good practice towards the exam.

The solution for this questions will not be included in the regular lab solutions pdf, but you are welcome to open a discussion on the Moodle: we will support your addressing it, and you may meet other students that choose to solve this, and find a teammate for the next assignment that is willing to do things for fun and not only for score :)

Let's see some multiclass classification. Copy the code loading the Iris dataset, you want to extract the same features (so you can plot in 2D), but keep the three classes.
[think: does it matter what label does each class have? Could you use strings such as `['a', 'b', 'c']`?]
Then run the scikit-learn LDA on the data to obtain a trained model. At this point you can open up the trained coefficients again, and rather than taking only the first like you did with `[trained.coef_[0], trained.intercept_[0]]`, you should have TWO w vectors and TWO b constants **per each pair** of classes. *[think: the space is actually split in several subspaces. Can you derive how many? Can you design a decision tree on top of the boundaries to do the classification as the number of boundaries grow?]*

BONUS [ZERO pt] Plot the boundaries classifying the three species of Iris in the dataset based on the two features used so far.

[ ]:

4.1.1 Final considerations

Stop for a moment and think how hard it was to derive these equations (in the lecture), and how hard it was instead to implement them (once you get them right). These are two very different skills.
To understand the derivation you need to think hard, express your concept in math (actually requiring broad knowledge of many of its subfields), see it through with absolute precision, and finally correctly solve the equations.
To implement the method, you need to map the math to the correct function calls (hard, but arguably less), and you only ever work with the final solution, but you deal with programming languages and libraries and documentations.
This is the reason why so many people nowadays broadly advertise machine learning skills after taking short tutorials. But if you do not understand what a parameter is for, you will only be guessing which value to use.

The reason why you are sweating so much on this course is to gain an edge over all of those who only ever learn to *use* the tools: by instead *making* the tools you understand them from the inside out, their applications and limitations, and even become capable of adapting and improving them. Keep up: this course is not easy, but machine learning has become unavoidable in your field, and these foundations will enable you to bend the whole field to your needs.