assignment_01_solution

February 22, 2021

Please fill in your name and that of your teammate.

You:

Teammate:

1 Introduction

Welcome to the first lab. Take a moment to familiarize yourself with this interactive notebook. Each notebook is composed of cells. Each cell can be markdown *(easy-to-format text)* or **code** (Python 3). Markdown cells also accept LaTeX formatting. Feel free to double-click on these textual cells (switching to edit mode) to see how they were made.

When a cell is highlighted, it has two modes: command and input. Press `esc` to go to *command mode*, press `enter` to go into *edit/input mode*. There are a few shortcuts that only work in edit mode that can make your life easier. This will probably satiate your curiosity: [link].

A few shortcuts that will make your life easier: you can always press `shift+enter` to evaluate the current cell; in *command mode*, press `a` to create a new cell above the current, `b` to create one below, `m` to convert to markdown, `y` to convert to code. Cells can be also merged and split (did you check the shortcuts?).

And if you look for a command for which you don't know the shortcut yet, press `p` for the command palette and try typing what you are looking for in the search field.

1.0.1 (Computational) kernel

Jupyter as a server generates the interactive web interface that you are seeing (and using) now. To run the code, the server maintains a running instance of Python, what is called a *computational kernel*. Think of it as an open terminal window with the interactive Python open: each time you *evaluate* a cell, the code is run on that "kernel", and the output is displayed below the input cell. Try to run the cell below:

[1]:
```
a = 3+2
print(a)
```

```
5
```

Then you can use the variable `a` again in your next executions:

[2]:
```
print(a)
```

```
5
```

Just remember that the order in which you run the cells matters: for example try running next the cell below this text, and then the cell above this text once more.

[3]:
```
a = 3-2
```

Check the `Kernel` menu on top of the page for options on controlling the underlying Python execution. For example, `Restart & Run All` terminates the current kernel, launches a new fresh one, then executes all (code) cells in the notebook in order. The `Interrupt` command is also useful if a bug gets the execution stuck.

1.0.2 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: **[0pts]**. 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. Overcomplete 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 16 points (out of 24 available) to pass (66%).

2 1. Fundamentals

Let's start simple.

1.1 [1pt] Write a sentence that correctly employs the words *problem*, *solution*, *model* and *parameters set*. A problem's solution is described by a model and a fixed parameters set.

1.2 [1pt] Write the equation of a linear model. You should use LaTeX formatting, just wrap the equation in between dollar signs (e.g. `$\LaTeX$`). $y = mx + q$

1.3 [1pt] When is a system of equations *overdetermined*? When there are more know data points than unknown variables.

1.4 [1pt] Describe with your own words (i.e. English) what is a *Training Dataset*. Mind, there is not an explicit definition in the slides, you should understand the math (your lecture notes may help).

A dataset is a set of data points. A data point (in supervise learning) is a pair composed of an input (an element of the input space) and a target (element of the decision space). A training dataset is a dataset used for training a model.

1.5 [1pt] List the other two (main) learning paradigms beside *unsupervised learning*. Supervised and reinforcement.

1.6 [1pt] Which word describes when the model I use is too complex to capture the underlying simplicity of the data? Careful not to pick the wrong term.

Overfitting.

3 2. Error, loss and risk

Understanding the concept of loss and risk is fundamental to comprehending the general idea of an "error". The whole ML is founded on the basis of recognizing error and minimizing it. These questions go into a separate section to highlight how important it is that you understand what is going on here.

2.1 [3pt] About the *Loss Function*: why $L(\hat{y}, y) = 0$ if $\hat{y} = y, \ \forall y \in Y$? Because the loss quantifies the error between two elements. If those two elements are the same, there is no error, hence no loss.

If you want to implement the *Empirical Risk* in Python, you need to understand its mathematical form. Let's say that the *Loss* is a simple difference between prediction and target:

2.2 [3pt] What does $\hat{R}(h) = \sum_{i=1}^{n} L(h(x_i), y_i), \ \forall (x_i, y_i) \in D$ mean? In English here, though you will get to code it in one of the next questions.

The empirical risk is the sum of all differences between the value predicted by the model and the correct target, across all elements (i.e. data points) in the training set.

4 3. Simplest learning

Enough concepts, let's have some fun. I hope you are familiar with Python -- if not yet, you should become so by the end of the course. If your confidence is low you should start a discussion on Moodle, so that you can all help each other (and help us help you).

Do you know about `lambda` functions in Python? You can write a method that returns a function. The function can be used as if it was a method defined with `def`. Only be careful about 1. `lambda`s always (implicitly) `return` the result of their computation, and 2. you cannot write multiline lambdas (go ask Guido van Rossum why). Still, using them is easier than it sounds:

[4]:
```
def add_n(n): return lambda x: n+x
add_3 = add_n(3)
add_3(5)
```

[4]: 8

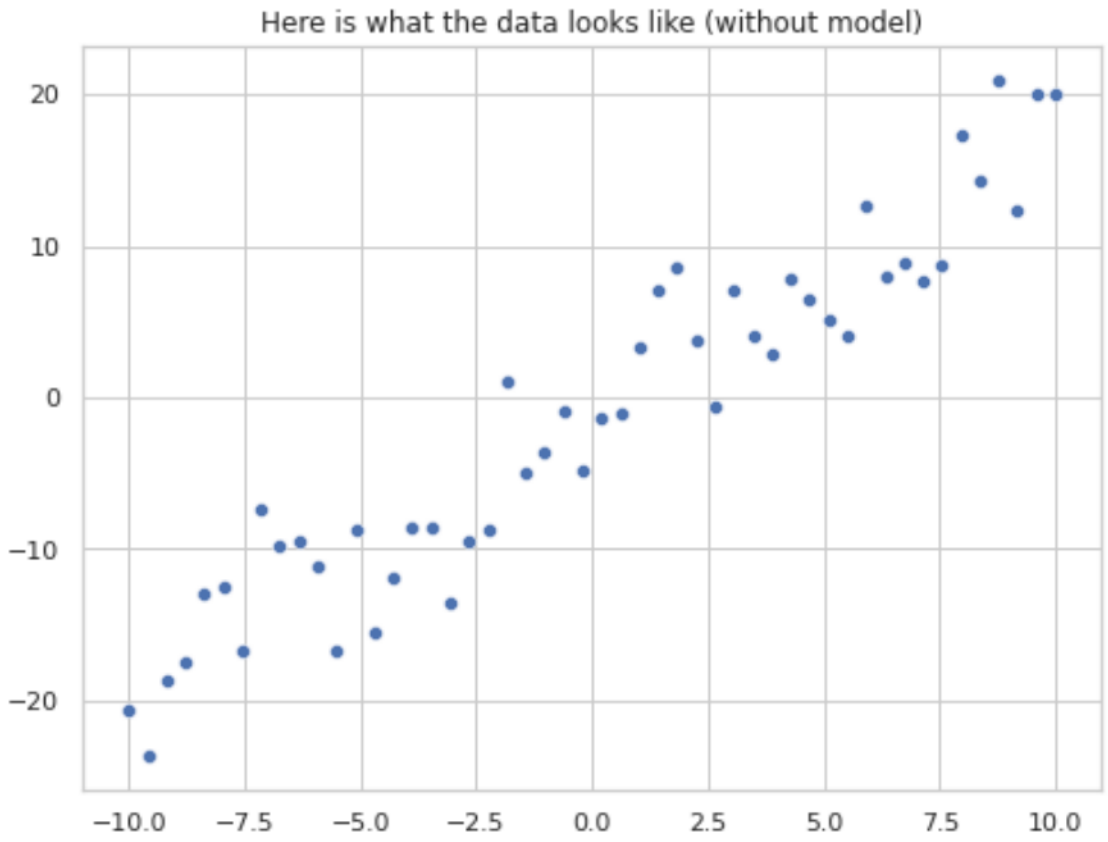
Ok how about we create and plot some artificial data? Study the code below, if there is any feature you are not yet familiar with you should make sure to learn it (ask on Moodle).

I mean it. Later on you will be required to use all of these functionalities yourself. Verify early in the course what you need to refresh and what is entirely new, because later on studying software engineering while working on the (much!) harder assignments may become a problem.

[5]:
```
# These lines are required for our plotting function below
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
# While this is the library for numerical manipulations
import numpy as np

# This is just some styling for the plotting library
sns.set(rc={'figure.figsize':(8,6)}, style="whitegrid")

# Let's create the data from a function you should be acquainted with by now
# (do you know lambdas yet? They are basically short, unnamed methods)
trg_fn = lambda x: 2*x - 1
# Of course we want the data to be a bit noisy
some_noise = lambda: np.random.normal(0,3)
# Let's generate it using numpy's linear space and a python list comprehension,
# just to make sure you know these too
data = [[x, trg_fn(x) + some_noise()] for x in np.linspace(-10, 10, 50)]

# You will find commonly data treated by axis/column rather than coordinate␣
↪pairs.
# This aggregates data series belonging to the same dimension (feature)
transpose = lambda lst: list(map(list, zip(*lst)))
data_x, data_y = transpose(data)

# And here's a canned plotting function that you are free to use (for now...)
def plot_data_and_model(model=None, text=None):
    ret = sns.scatterplot(x=data_x, y=data_y) # hard-coded data plotting␣
↪because we can
    if model is not None:
        sns.lineplot(x=data_x, y=[model(x) for x in data_x], color='darkred')
    if text is not None:
        plt.title(text)
    return ret
```

Here is what the data you just generated looks like:

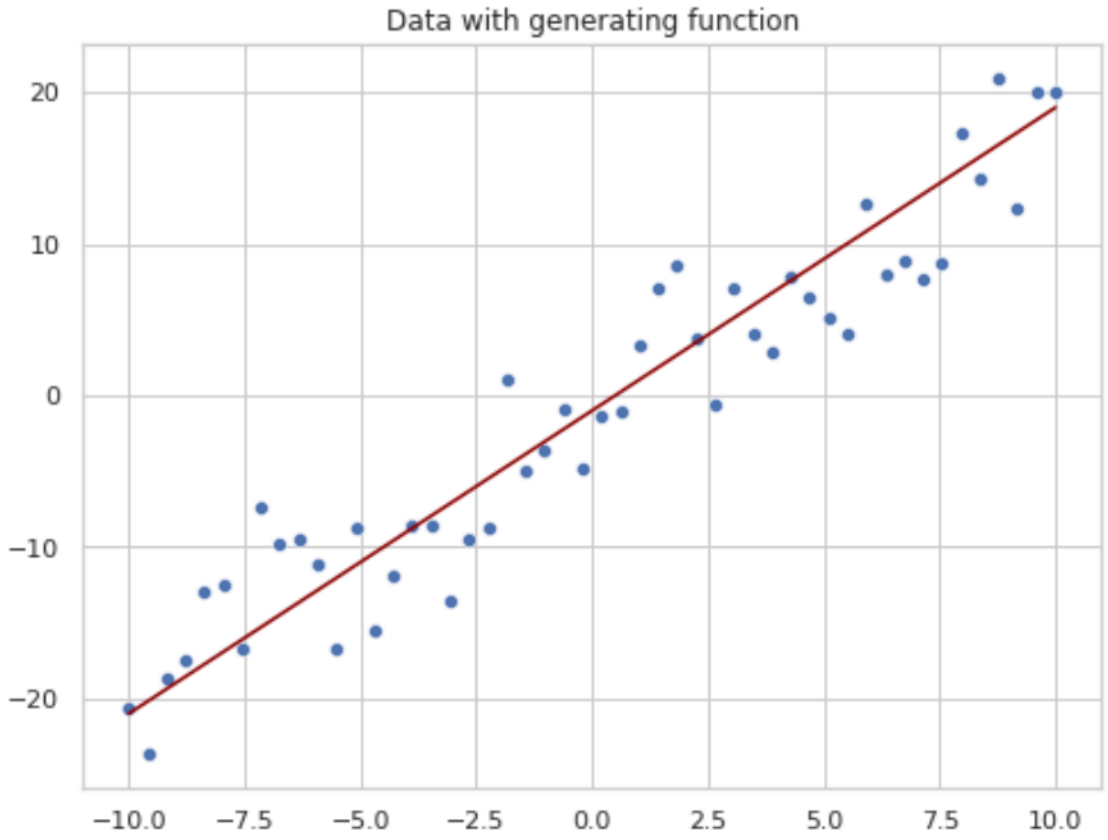
[6]:
```
plot_data_and_model(text="Here is what the data looks like (without model)");
```



And here is what it looks like if you cheat and plot the underlying function (ideally your final, learned model should look similar)

[7]:
```
plot_data_and_model(trg_fn, text="Data with generating function")
```

[7]:
```
<AxesSubplot:title={'center':'Data with generating function'}>
```



3.1 [2pt] Write your linear model as a method that takes `m` and `q` as input, and return a linear function of the form `mx + q`.

[8]:
```
lin_model = lambda m, q: lambda x: m*x+q
```

3.2 [2pt] Write your loss as a method that takes an `x` and a target and returns the absolute value of their difference (think: what happens if we forget the absolute value?)

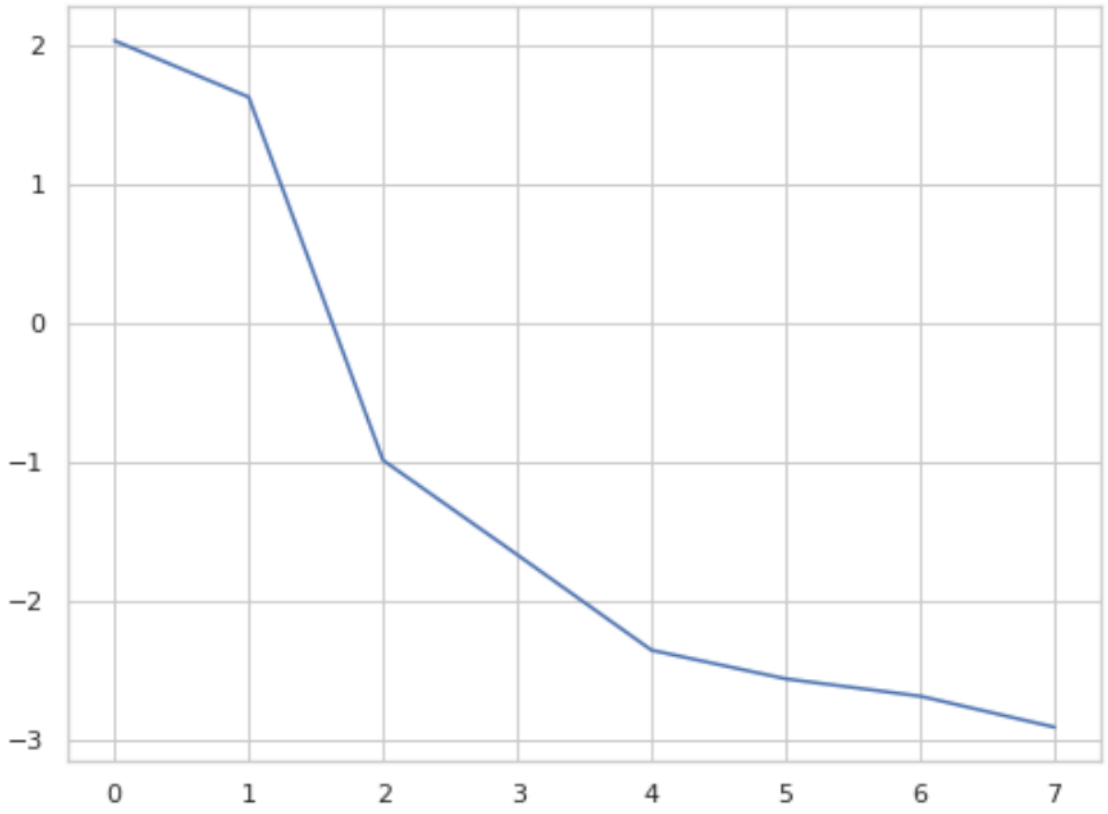
[9]:
```
loss = lambda x, trg: abs(trg - model(x))
```

3.3 [2pt] Write your risk as a method (or lambda) that takes a model as input, and returns the total loss over our (hard-coded) data

[10]:
```
risk = lambda model: sum([loss(x, trg) for x, trg in data])
```

Here is an example of a loop that generates random numbers and maintains a *minimum*.
(think: will you need to minimize or maximize the risk of your model?)

[11]:
```
min_guess = np.Infinity # higher than highest possible

best_guesses = []
for _ in range(100):
    guess = np.random.uniform(-3,3)
    if guess < min_guess:
        min_guess = guess
        best_guesses.append(min_guess)

sns.lineplot(x=range(len(best_guesses)), y=best_guesses);
```



TIP: it is always useful to visualize how the loss decreases over time, especially for debugging purposes. You can do the same next for your errors/losses.

3.4 [6pt] Randomly guess a model's parameters 1000 times. Then plot it using the call below.

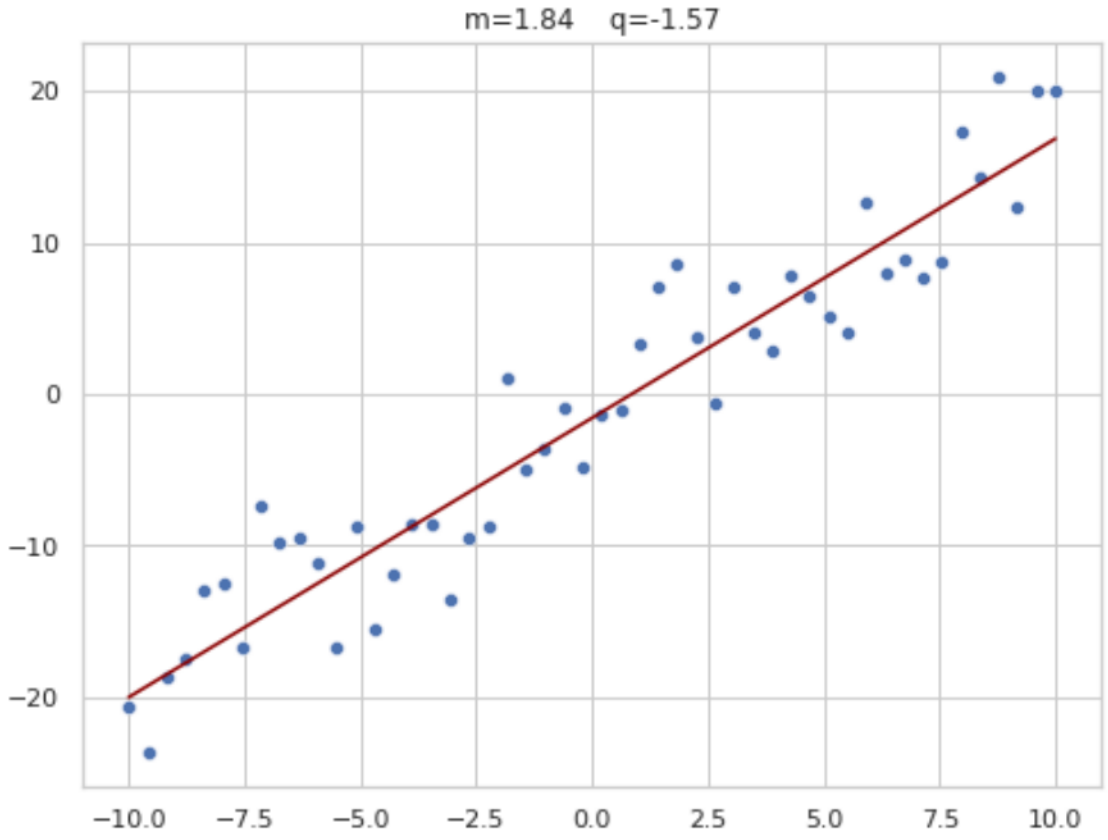
```
plot_data_and_model(lin_model(m, q), text=f"m={round(m,2)}     q={round(q,2)}")
```

Make sure you understand how string interpolation works when using the format `f"hello w{2+1-3}rld"`.

You will need to modify the loop above in order to maintain both a best guess for your model and its corresponding best risk/error.

[12]:
```
best_guess = None
best_error = np.Infinity

errors = []
for _ in range(1000):
    guess = np.random.uniform(-3,3,2)
    model = lin_model(*guess)
    error = risk(model)
    if error < best_error:
        best_guess = guess
        best_error = error
        errors.append(error)

m, q = best_guess
plot_data_and_model(lin_model(m, q), text=f"m={round(m,2)}    q={round(q,2)}")
```

[12]:
```
<AxesSubplot:title={'center':'m=1.84    q=-1.57'}>
```



assignment_02_solution

February 27, 2022

Please fill in your name and that of your teammate.

You:

Teammate:

1 Introduction

Welcome to the second lab. I hope this environment is starting to look more familiar, and that you learned some of the shortcuts. At least try to use shortcuts to evaluate a cell, create a cell above or below the current, and switch between code and markdown, as these will dramatically improve your efficiency and significantly cut the time it takes to complete the assignment.
Also, if you have not tried LaTeX in the previous assignment, give it a try in this one $y = mx+q$: a little practice goes a long way for when you will need to write more complex equations and LaTeX will be required (e.g. future assignments + exam).

Today's assignment is likely going to be a bit more time consuming than last week. And there is *a lot* of Python. I received multiple emails from people who are not confident in their Python skills or are worried having to learn it now in parallel with the main class content. I understand the concern and will be sure to hold your hand step by step in learning and practicing the required skills even if you have no prior experience.
Please understand nonetheless that proficiency in Python as seen here will be mandatory to pass the exam, so give it your best effort. You may want to start completing this assignment for example a few days before the deadline, to leave yourself time to ask any question on Moodle if needed.

Last week we introduced three libraries: `numpy` does the main number crunching in Python; `matplotlib` is the foundation of most plotting; and `seaborn` wraps the plotting in a convenient interface and eye-pleasing defaults.

Today we introduce **Scikit-learn**, another heavy-weight library in the field which provides basic (but high-quality and fast) data analysis and ML tools.
Head over to the home page of Scikit-learn: how many of the concepts are you already familiar with? Over the next week you will become confident in almost each single word used in that page. Also check out the user manual for an overview of the methods at your disposal.

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: **[0pts]**. 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 14 points (out of 21 available) to pass (66%).

2 1. Fundamentals

The topics of *Classification* and *Feature* are fundamental ML concepts. Today's lecture has barely scratched the surface, especially if you think Data Analysis as a goal rather than a simple step, so we will explore these topics in further detail in the upcoming weeks. Let's make sure to have a solid grasp of these concepts before moving ahead.

Again, let's start easy. Here is an example dataset of snakes. It has three fields: `head size`, `length` (in cm) and whether it is `poisonous` or not. It looks like this:

```
snakes = [['small', 38, False],
          ['small', 62, True],
          ['medium', 55, True]]
```

This simple list of lists puts the emphasis on the data points (the rows), which is an intuitive approach at first for humans to manually write down the data. You will often encounter tables, JSON and CSV files that look like this. We will explore more machine-friendly formats later (spoilers: column-oriented), so understand this approach is just a step for ease of comprehension.

Let's lighten the mood by focusing on cats and dogs instead for your exercise.

1.1 [1pt] Name three features that are highly discriminative to classify cats from dogs, and three which are not. Highly discriminative: [retractable_claws, night_vision, independence_level]

Less discriminative: [length_of_tail, is_household_pet, scared_by_fireworks]

1.2 [1pt] Is "number of legs" a numeric feature? Is it discrete or continuous? Is it ordinal? What about "length of tail"? Careful not to confuse "discrete cardinal" with "continuous". If you cannot tell the difference, refresh the concept!

Both are numeric; `number_of_legs` is discrete but it is *cardinal* not ordinal; `length_of_tail` is continuous, though positive and (hopefully) bounded (no negative nor infinite size).

1.3 [1pt] Write a dataset named `pets` containing cats and dogs, with at least 3 features and 5 entries. Write the corresponding labels in a variable named `labels`. Save the feature names in a variable named `feature_names`. Make sure to include some small dog and large cat, such that their size is not highly discriminant. You can write more data if you want, but do not go overboard, your goal for now is uniquely to pass the assignment. We will load some demo dataset with Scikit-learn later.

[1]:
```
feature_names = ['likes_water', 'agility', 'height']
pets = [[True, 8, 25],
        [True, 4, 50],
        [False, 7, 30],
        [False, 9, 20],
```

```
        [True, 2, 65],
        [False, 8, 30],
        [True, 6, 25],
        [True, 8, 40],
        [True, 9, 45],
        [False, 6, 35]]
labels = ['cat', 'dog', 'cat', 'cat', 'dog', 'cat', 'cat', 'dog', 'dog', 'cat']
```

Human input means human errors: let's validate the length of these lists using `assert`, which is a Python keyword that will raise an error if its parameter is false (and just do nothing otherwise). Here we use `map` to execute `len` over each element of a list (which here is: for each data row), then check if all values correspond to the length of feature names. The function `all` returns whether all of its arguments have truth value. We can also verify the number of labels against the number of data points.

[2]:
```
assert all(l == len(feature_names) for l in map(len, pets))
assert len(pets) == len(labels)
```

3 2. Decision trees

You are going to write a decision tree by hand, that classifies cats from dogs on your dataset, by using a simple chain of `if/else` statements. Do not overlook this task: it is an industry standard to integrate human expert knowledge in an automated ML system.

Include at least two questions, meaning the tree depth (max number of decision nodes between start and leaf) should be at least 2. The leaves should contain decision labels, i.e. either *cat* or *dog*, though you can have multiple instances of either.
I hope you find it obvious that the labels should not be passed to the function.

Do you know the splat operator? You may find it useful to pass data points to your function. Here is a short demonstration video.

With `map`, `zip` and the splat you should now be able to understand the `transpose()` function from last week:

```
transpose = lambda lst: list(map(list, zip(*lst)))
```

A decent Python skill level is nowadays a mandatory requirement for good Data Analysis or Machine Learning job positions.

2.1 [2pt] Implement a Decision Tree as a function that takes the features of a data point from the `data` defined above and returns a predicted label using an if/else chain. Run it over your `data` to obtain a list of predictions.

[3]:
```
def decision_tree(likes_dogs, agility, height):
    # note: ignoring `likes_dogs` as it is not discriminative
    if agility > 5:
        if height < 40:
            return 'cat'
```

```
        else:
            return 'dog'
    else:
        return 'dog'

# Here's another splat for practice (in another list comprehension):
# this deconstructs the data point into a parameters list of its features
predictions = [decision_tree(*point) for point in pets]
```

To quickly check if it got them all right you can use `zip`, which builds lists taking one element in turn from each of its input lists.

[4]:
```
for pair in zip(predictions, labels): print(pair)
```

```
('cat', 'cat')
('dog', 'dog')
('cat', 'cat')
('cat', 'cat')
('dog', 'dog')
('cat', 'cat')
('cat', 'cat')
('dog', 'dog')
('dog', 'dog')
('cat', 'cat')
```

Now we need to properly assess the performance of our classification. This is commonly done using the **Confusion Matrix**: two rows and two columns, indicating the count (over the dataset) of

$$\begin{pmatrix} \text{true positives} & \text{false negatives} \\ \text{false positives} & \text{true negatives} \end{pmatrix}$$

(edit this cell and notice above how you can use multi-line `latex` by wrapping your code in `$$` pairs)
(also it does not matter whether you pick *cat* or *dog* as the "positive" class, but be careful and consistent)

The confusion matrix if the foundation to most loss functions for classification, some of which can be extremely sophisticated.

Perfect classification means no false positives (positive answers for data points that should classify negative) and no false negatives (negative answers for data points that should classify positive) for all data points in your dataset.

Hint:

```
pos = 'cat'; neg = 'dog'
tp = 0; tn = 0; fp = 0; fn = 0
for pred, lab in zip(predictions, labels):
    # your code here
```

2.2 [2pt] Compute and display the Confusion Matrix. If your tree did not achieve perfect classification, write a new version that does. Print the result using string interpolation. There are three ways to interpolate strings in Python: using `format()`, using `%` and using f-strings. You can read about why you should switch to f-strings, but for now just try using something like this:

```
print(f"tp: {tp}, fn: {fn}\nfp: {fp}, tn: {tn}")
```

[5]:
```
# This is a very unsophisticated implementation: make your code as readable as you can!
pos = 'cat'; neg = 'dog'
tp = 0; tn = 0; fp = 0; fn = 0

for pred, lab in zip(predictions, labels):
    if pred == pos: # positive prediction, i.e. cat
        if lab == pos: # if label is also positive (true positive)
            tp += 1
        else: # if label is negative (false positive)
            fp += 1
    else: # negative prediction, i.e. dog
        if lab == pos: # if label is positive (false negative)
            fn += 1
        else: # if the label is negative (correct: true negative)
            tn += 1

print(f"tp: {tp}, fn: {fn}\nfp: {fp}, tn: {tn}")
```

```
tp: 6, fn: 0
fp: 0, tn: 4
```

[think: did you just write a decision tree? can you name the features?]

Doing these things by hand can be tedious, but provides a different type of confidence to then go and study the documentation of the library you would rather use in real applications.
Here is the scikit-learn implementation of a decision tree, and here is the main class. Let's load the implementation with the following:

[6]:
```
from sklearn.tree import DecisionTreeClassifier, export_text
```

2.3 [1pt] Train a scikit-learn DecisionTreeClassifier on your dataset.

[7]:
```
model = DecisionTreeClassifier()
trained = model.fit(pets, labels)
print(export_text(trained, feature_names=feature_names))
```

```
|--- height <= 37.50
|   |--- class: cat
|--- height >  37.50
|   |--- class: dog
```

2.4 [1pt] Compare the two trees (handmade and scikit-learn) in number of leaves, tree depth, selected features, and thresholds (in English). The `scikit-learn` tree is more optimized. It has less leaves (2 rather than 3), is shallower (depth 2 rather than 3), only uses one feature (only height rather than agility and height) and picks a more precise threshold (37.5 rather than 40).

4 3. Perceptron

This is our first proper learning algorithm, and also the first with an iterative implementation. Its implementation is simple: you should use this opportunity to become confident in its features, as we will find them in much more complex algorithms over the next weeks. Any extra work in this section will make the following assignment much, much easier.

3.1 [1pt] Write the equation of an hyperplane in $\mathbb{R}^k$, specifying the numeric set and dimensionality of each parameter. $y = \langle w, x \rangle + b, \;\; w \in \mathbb{R}^k, \;\; b \in \mathbb{R}$ (note: $x \in \mathbb{R}^k, \;\; y \in \mathbb{R}$)

3.2 [1pt] Write the definition of *Linearly Separable Dataset* in plain English (no math). A dataset where points belonging to different classes can be perfectly separated using an hyperplane, i.e. there exist a linear equation for which all points have positive margin.

For the next question, let's make sure the concept of Margin and its use is clear. Here is an example of point (x, y) and hyperplane parametrization (w, b).

$$\text{point:} \quad ((1, 3, -5, -2), +1) \text{params:} \quad ((2, 7, -3, 5), -2)$$

3.3 [1pt] Compute by hand (LaTeX not Python!) the *margin* for the point and hyperplane above: is the point correctly classified by the hyperplane? Why?

$f(x) = \langle w, x \rangle + b \mu = y \cdot f(x) = +1 \cdot (\langle w, x \rangle + b) \langle w, x \rangle = 2 + 21 + 15 - 10 = 28 \mu = +1 \cdot (28 - 2) = 26$

$\mu > 0 \rightarrow$ point is classified correctly.

For the next question we use `numpy`.
We use the definition of Affine Function for the parametrization: here `point` has been conveniently augmented with a trailing 1 representing the constant input for bias.
Careful from now on you will need to take care of that yourself, starting from one of the next questions. Here are two examples (marked `1` and `2`) of how it can be done.

```
x = [1, 3, -5, -2]
y = 1
w = [2, 7, -3, 5]
b = -2
point = [np.array([*x, 1]), y] # 1
params = np.append(np.array(w), b) # 2
```

Use `np.dot()` to compute the inner product.

3.4 [1pt] Write a function that takes a in input an hyperplane parametrization and a point, computes the margin, and returns a boolean indicating whether the classification is correct or not. Run it on the point and parametrization provided below ("The Inputs"), and print whether the classification is correct or not.

[8]:
```
# The Inputs -- do not change
import numpy as np
point = [np.array([1, 3, -5, -2, 1]), 1]
params = np.array([2, 7, -3, 5, -2])
```

[9]:
```
def correctly_classified(params, point):
    x, y = point # this is also a type of list deconstruction
    margin = y * np.dot(params, x)
    return margin > 0

if correctly_classified(params, point):
    is_correct = "_correct_"
else:
    is_correct = "_not correct_ "

print(f"The classification is {is_correct}.")
```

```
The classification is _correct_.
```

3.5 [1pt] Implement the Perceptron update rule for a single point as a method that takes a hyperplane parametrization and a point, and returns the updated parametrization. Run it on The Input above and print the updated parametrization. Notice this function will be run by the Perceptron algorithm only on points that are misclassified. But for now we are just testing the function with our input, so show its output on The Input regardless of whether it is misclassified or not.

[10]:
```
def perceptron_update(params, point):
    x, y = point
    return params + (y * x)
updated_params = perceptron_update(params, point)
print(updated_params)
```

```
[ 3 10 -8  3 -1]
```

3.6 [1pt] Print whether the updated parametrization from the last question correctly classifies the point from The Input (use and the margin-based function you wrote to answer two questions above).

[11]:
```
correctly_classified(updated_params, point)
```

[11]:
```
True
```

Alright, do you feel confident of your implementation so far? Let's scale it up: implement the Perceptron Algorithm and run it on a demo dataset from Scikit-learn.

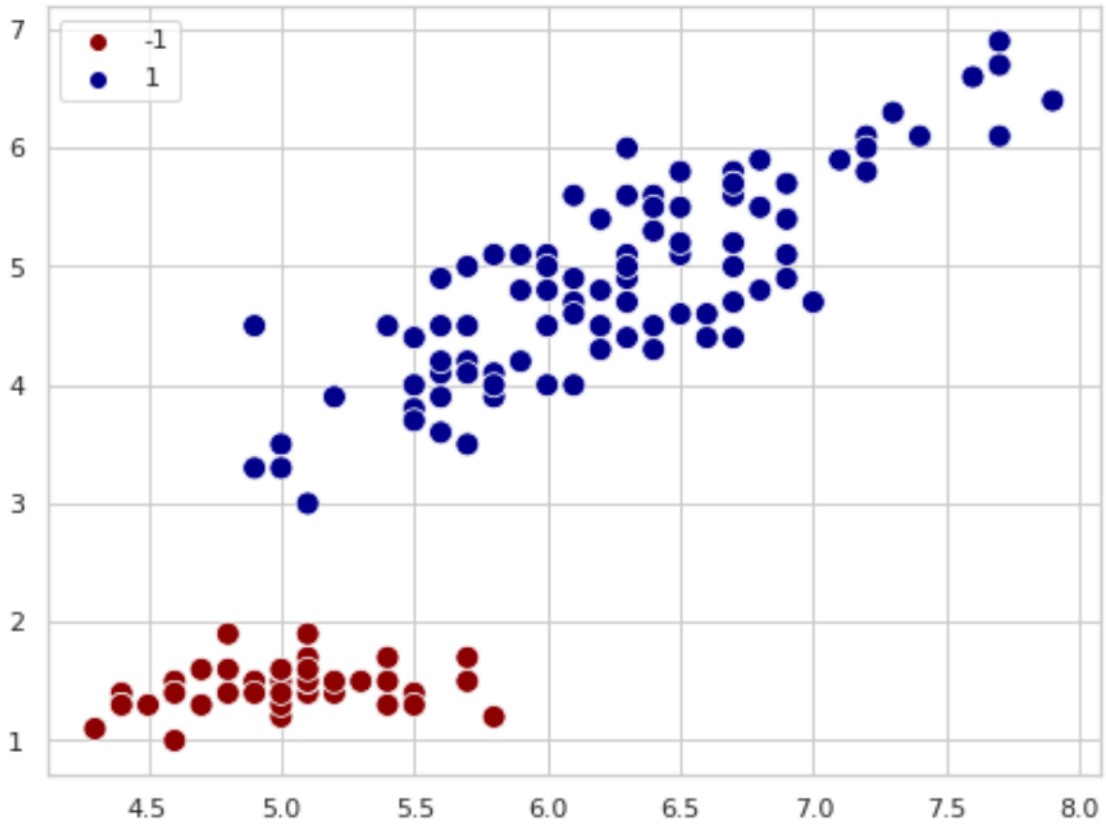
First we load the classic Iris dataset, its history is very interesting so make sure to have a look at it. Since we are studying linear binary classification, let's collapse two classes together and focus on two if its four features.

[12]:
```
from sklearn.datasets import load_iris
iris_x, iris_y = load_iris(return_X_y=True) # print these to understand
x1 = np.array([r[0] for r in iris_x]) # first feature
x2 = np.array([r[2] for r in iris_x]) # third feature
x = np.array([x1, x2]).transpose() # numpy transpose() for free
# Reduce to two binary classes {+1, -1}
labels = np.array([-1 if y==0 else +1 for y in iris_y])
```

Your Perceptron inputs should be x input vector and *labels* target labels / classes.
We learned last week what we need to plot such a dataset, right?

[13]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
sns.set(rc={'figure.figsize':(8,6)}, style="whitegrid")
```

[14]:
```
def todays_plot(): # learn to write your own
    sns.scatterplot(x=x1, y=x2,
      hue=labels, # let's use different colors for the two classes
      palette=sns.color_palette(['darkred', 'darkblue']),
      s=100)

todays_plot()
```



The problem is clearly linearly separable. Let's see how the Perceptron performs

3.7 [4pt] Implement the Perceptron Algorithm in a single cell (do not call any function defined above) as a function that takes the input vector and target labels and returns a trained hyperplane parametrization. Run it on the (partial) Iris data loaded above.

[15]:
```
def perceptron(inputs, targets, max_updates=10000):
    w = np.zeros(len(inputs[0]) + 1) # +1 for bias
    nupdate = 0
    misclassified = True # is any data point misclassified? Assume True at start
    # Remember: on a non-linearly separable dataset, the Perceptron never␣
↪terminates!
    while misclassified and nupdate < max_updates: # ... hence we limit the run
        for x, y in zip(inputs, targets):
            misclassified = False
            x = np.append(x, [1]) # here's the fixed input for the bias (affine␣
↪fn)
            margin = y * np.dot(w, x)
            if margin <= 0:
                misclassified = True # need to run more
                w += y * x
```

```
                nupdate += 1
                break # out of for
    if nupdate >= max_updates:
        print("FAIL: could not achieve linear separation in allotted time")
    else:
        print(f"Solved in {nupdate} iterations.")
    *w, b = w # inverse affine using splat
    return [w, b]

print(perceptron(x, labels))
```

```
Solved in 10 iterations.
[[-3.3999999999999977, 9.100000000000001], -2.0]
```

Ok we can plot the points and we can plot a $y = mx + q$ model. But how can we plot the $f(x) = \langle w, x \rangle + b$ decision boundary? Well we know the two classes are $f(x) > 0$ (positive, above) and $f(x) < 0$ (negative, below), so our boundary is in $f(x) = 0$. Then we can find the function coefficients in the w_1, w_2 space as:

$$f(x) = \langle w, x \rangle + b = 0 w = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} ax + by + c = 0 y = mx + q a = w_1, \;\; b = w_2, \;\; c = b w_1 x + w_2 y + b = 0 - w_2 y = w_1 x + by =$$

For now you can use the implementation below, but make sure you understand these simple (if tedious) steps above because next time you will need to implement it yourself.

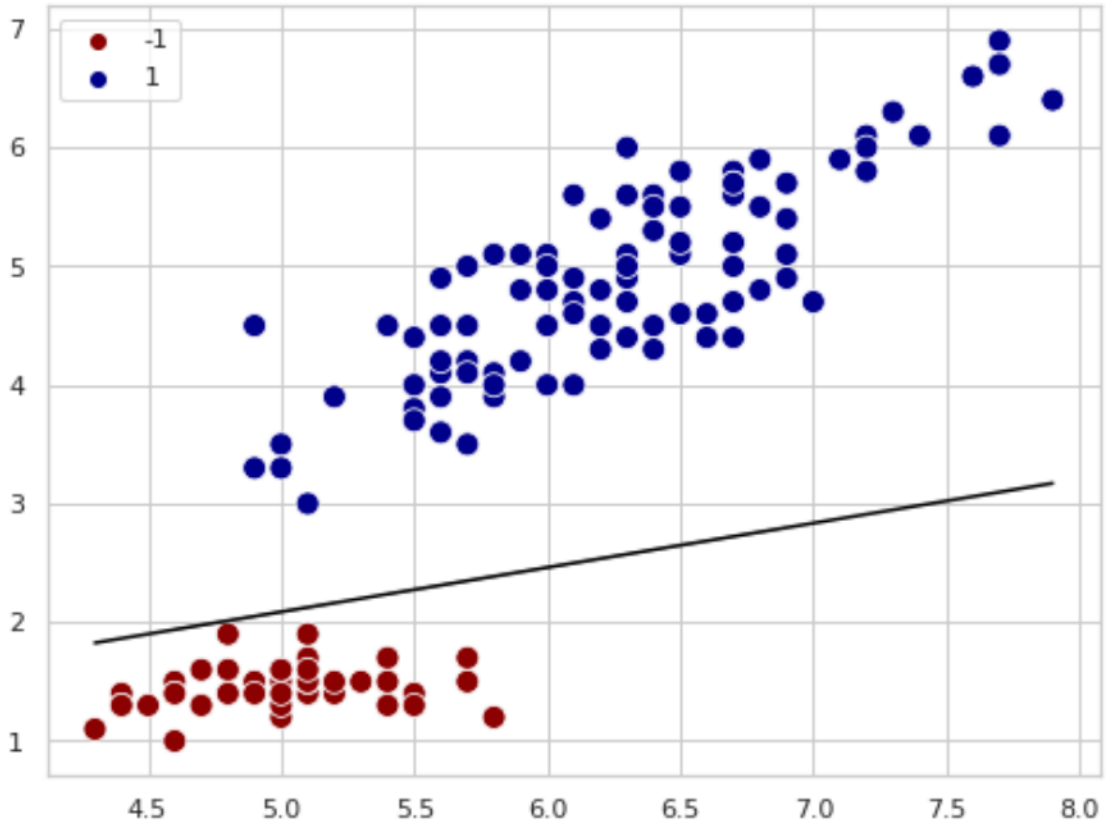
[16]:
```
def wb2mq(w, b):
    assert len(w) == 2, "This implementation only works in 2D"
    assert w[0] != 0 and w[1] != 0 and b != 0 # avoid edge cases for now
    return [w[0]/-w[1], b/-w[1]] # m and q

def params2boundary(w, b):
    m, q = wb2mq(w, b)
    print(f"m: {m}, q: {q}")
    return lambda x: m*x + q
```

I hope the code above comes to no surprise after having derived it mathematically. If you read it aloud it should sound obvious in English; if not go back and make sure you understand the mathematical derivation (and the code above) before moving forward or it is only going to be more confusing.
Now we can plot the model on top of our data:

[17]:
```
todays_plot()
perc_w, perc_b = perceptron(x, labels)
print(f"w: {perc_w}, b: {perc_b}")
perc_boundary = params2boundary(perc_w, perc_b)
sns.lineplot(x=x1, y=[perc_boundary(inp) for inp in x1], color='black')
```

```
Solved in 10 iterations.
w: [-3.3999999999999977, 9.100000000000001], b: -2.0
m: 0.3736263736263733, q: 0.21978021978021975
```

[17]: <AxesSubplot:>



Alright! Let's do the same with the scikit-learn Perceptron and we are done.

[18]:
```
from sklearn.linear_model import Perceptron
```

3.8 [1pt] Train a scikit-learn Perceptron on the same data as the last question. Note: it should take *no more than two lines*. You should still have the documentation open, the class you are looking for is `Perceptron`, and the method you need is typically called `fit` for most sklearn algorithms. Find out how it works and how to pass features and labels as arguments.

[19]:
```
model = Perceptron()
trained = model.fit(x, labels)
```

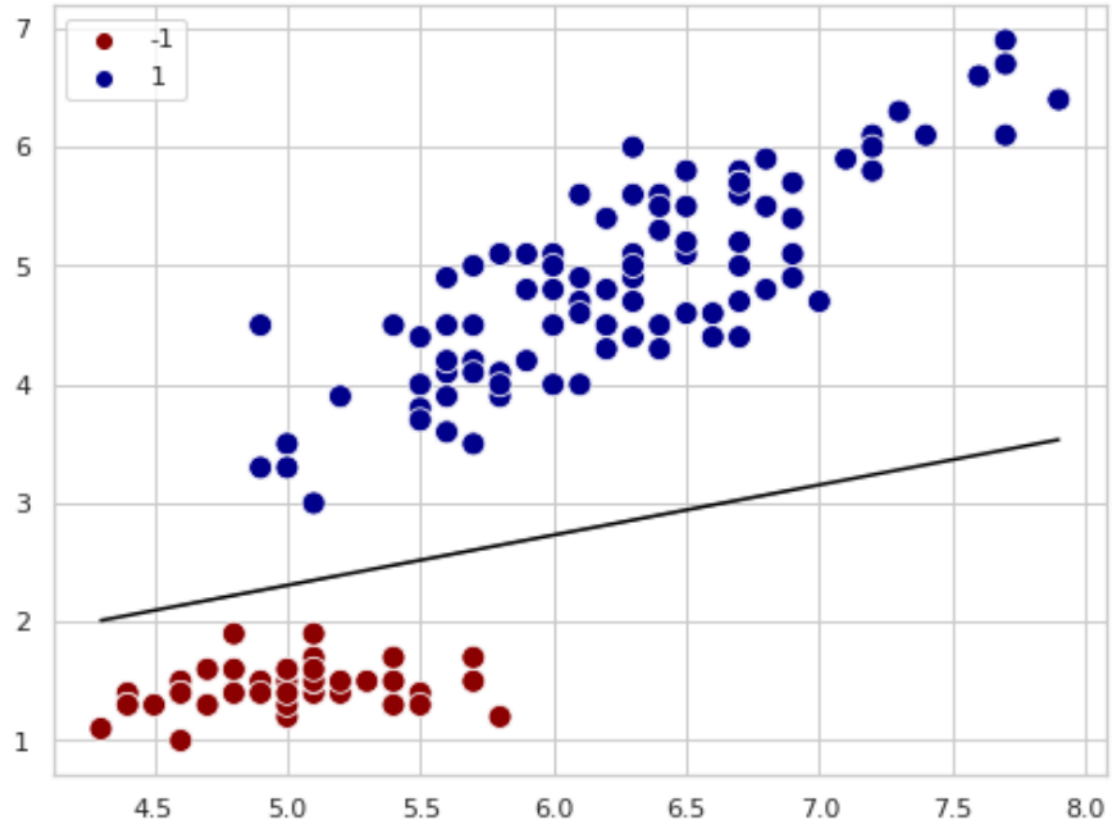
To visualize the model boundary we need extract the vectors w and b from a trained scikit-learn Perceptron: they are stored as `coef_` (coefficients stands for weights) and `intercept_` (which is another term for `q` or bias) *(as mentioned: be flexible with the nomenclature as each library adopts*

its own).
Note though that they are both `lists`, because the algorithm is written to scale to *multiclass classification*, where you have multiple classes and therefore need multiple hyperplanes to partition the space in multiple regions. Of course with two classes you only need one hyperplane so be sure to access only the first element.
I will be providing the code below yet again, but make sure you study and understand it so you will be able to write it yourself in the next assignments.

[20]:
```
w, b = [trained.coef_[0], trained.intercept_[0]]
d_boundary = params2boundary(w, b)
todays_plot()
boundary_preds = [d_boundary(inp) for inp in x1]
sns.lineplot(x=x1, y=boundary_preds, color='black')
```

```
m: 0.423076923076923, q: 0.19230769230769226
```

[20]:
```
<AxesSubplot:>
```



3.9 [1pt] Compare the resulting boundary against the one trained with your hand-made algorithm, and hypothesize why *in your opinion* they are similar/different (in English). This exercise (as many more in the future) trains your ability to express your opinion

and fundamental understanding of the topic using technical language. It is scored based on your expressiveness, not on the correctness (although you should be able to formulate a correct opinion here) or on the English per se. Show your reasoning!

They are very similar though the one trained with scikit-learn has a better margin overall. This suggests that the scikit-learn implementation is more sophisticated, as also hinted by the numerous parameters available. Our implementation is a simple implementation of the algorithm's foundation, and the boundary found reflects this.

assignment_03_solution

March 4, 2022

Please fill in your name and that of your teammate.

You:

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?

$$p(x) = \frac{1}{\sqrt{2\pi}\sigma} \cdot \exp\left(-\frac{\|x-\mu\|^2}{2\sigma^2}\right)$$

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?** We use the log because the likelihood has a product, which is hard to maximize, while the log transforms it into a much easier sum. The choice for the logarithmic is because the dependent term of the Gaussian is an exponential.

1.3 [1pt] Why the equation maximizing the log-likelihood of a Gaussian does not include the parameter σ? Because the terms with σ do not depend on w and thus do not influence the $argmax$.

1.4 [1pt] Explain the meaning of i.i.d. (in English), using the simplest words you can. Identically and independently distributed means that random values drawn from such a distribution do not influence each other. Drawing one does not change the probability of drawing the next one.

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

$$p(y \mid x) = \frac{p(x \mid y)\, p(y)}{p(x)}$$

The probability of class y given the input x is equal to: the probability of the class y generating such an input x, multiplied by the prior probability of selecting class y among all classes, divided by the overall probability of having input x.

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'. The label y_i of an input x_i can be interpreted as class prototype $\langle w, x_i \rangle$ plus a certain error ϵ drawn from a Gaussian (or normal) distribution centered in zero $\epsilon \sim \mathcal{N}(0, \sigma^2)$. The inverse is also true, as obvious by swapping the side of ϵ in the equation: the prototype predicted by the model with $\langle w, x_i \rangle$ corresponds to the exact label y_i plus some normal noise ϵ (which is centered in zero and symmetric, so the $+$ or $-$ sign does not matter).

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.

[2]:
```
def sq_loss(trg, pred): return (trg - pred)**2
sq_loss_risk = sum(sq_loss(trg, 1) for trg in y)
print(f"Risk for constant predictor 'y=1' using squared errors loss:␣
 ↪{sq_loss_risk}")
```

```
Risk for constant predictor 'y=1' using squared errors loss: 8367.948095222817
```

- 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 lr_w(x, y): return np.linalg.pinv(x.transpose().dot(x)).dot(x.transpose()).
 ↪dot(y)
w = lr_w(x, y)
print(w)
```

```
[ 2.13755392 -0.86214539]
```

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]

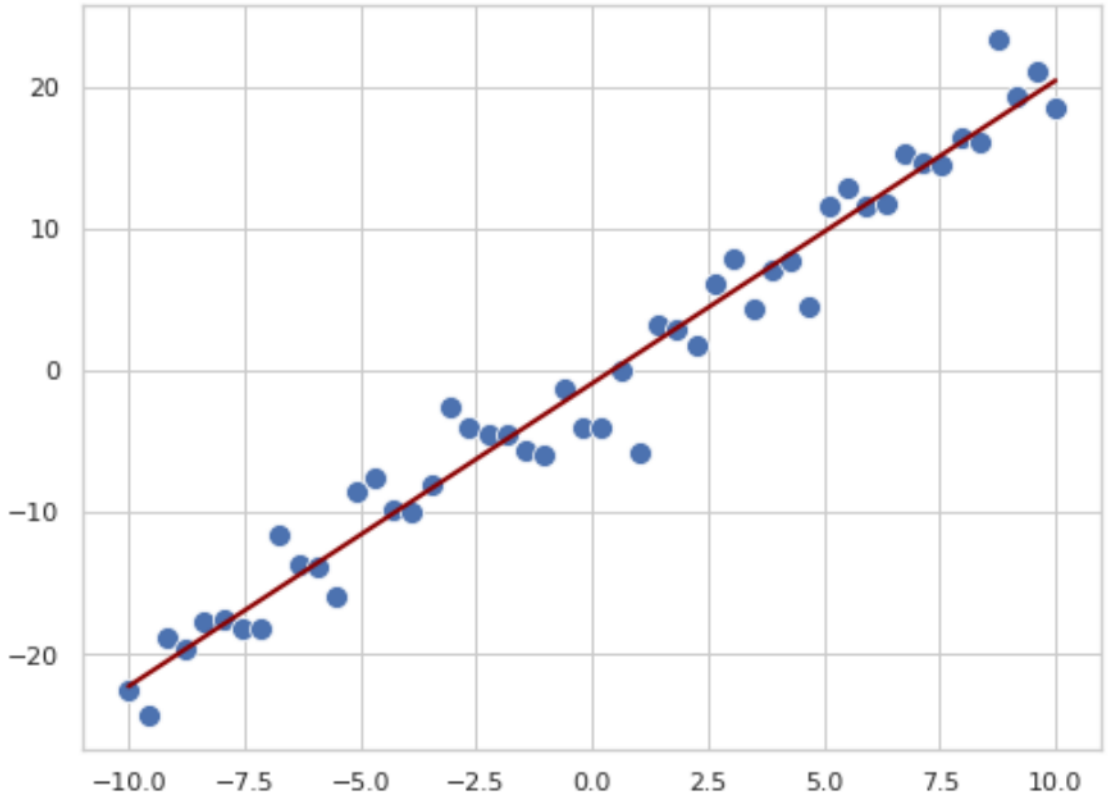
[4]:
```
predictions = list(map(round, w.dot(x.transpose())))
pred_pairs = list(zip(y, predictions))
losses = [round(sq_loss(pr, tr), 2) for tr, pr in pred_pairs]
triplets = list(zip(y, predictions, losses))
print(triplets)
risk = sum(losses)
print(f"\nRisk for trained predictor 'y = {round(w[0], 2)}x +"
      f"({round(w[1], 2)})' using squared errors loss: {round(risk, 2)}")
```

```
[(-22.453864154607444, -22, 0.21), (-24.355288554468324, -21, 11.26),
(-18.830239122972035, -20, 1.37), (-19.665900159366668, -20, 0.11),
(-17.590594067916417, -19, 1.99), (-17.48860480323548, -18, 0.26),
(-18.178304958883956, -17, 1.39), (-18.089088659308935, -16, 4.36),
(-11.52903912270708, -15, 12.05), (-13.717395400343177, -14, 0.08),
(-13.844148118394871, -14, 0.02), (-15.831761950240315, -13, 8.02),
(-8.526387229737626, -12, 12.07), (-7.461847746131552, -11, 12.52),
(-9.715655854404257, -10, 0.08), (-9.909317551003593, -9, 0.83),
(-8.045928551437639, -8, 0.0), (-2.531812785071322, -7, 19.96),
(-3.9803803043975448, -7, 9.12), (-4.47962380808473, -6, 2.31),
(-4.47374770595854, -5, 0.28), (-5.640332273994828, -4, 2.69),
(-5.9630644437187845, -3, 8.78), (-1.2021304001848205, -2, 0.64),
(-4.0095226892542115, -1, 9.06), (-3.946673301914443, 0, 15.58),
(0.0499153783108017, 0, 0.0), (-5.7507453206489565, 1, 45.57),
(3.309397567346861, 2, 1.71), (2.908114618478713, 3, 0.01), (1.8155796455017206,
4, 4.77), (6.158029201212644, 5, 1.34), (7.920590973028711, 6, 3.69),
(4.423001850855109, 7, 6.64), (7.074166622769325, 7, 0.01), (7.86603442423755,
8, 0.02), (4.536624663074363, 9, 19.92), (11.587999224641994, 10, 2.52),
(12.944521061201936, 11, 3.78), (11.71775175635889, 12, 0.08),
(11.84221548524487, 13, 1.34), (15.430996342049463, 14, 2.05),
(14.662467467973228, 14, 0.44), (14.533052724343113, 15, 0.22),
(16.422705771229143, 16, 0.18), (16.140145639855614, 17, 0.74),
(23.436430058554016, 18, 29.55), (19.470424677222745, 19, 0.22),
(21.204989635849472, 20, 1.45), (18.64897471130482, 21, 5.53)]

Risk for trained predictor 'y = 2.14x +(-0.86)' using squared errors loss:
266.82
```

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]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
sns.set(rc={'figure.figsize':(8,6)}, style="whitegrid")
```

```
# start a variable name with _ to declare you do not intend to use it (_ itself␣
 ↪is a valid name)
actual_x, _bias = x.transpose()

# did you learn to write your own plotting methods?
def lr_data_plot(): sns.scatterplot(x=actual_x, y=y, s=100)
def lr_model_plot(w, color='darkred', linewidth=2):
    # Seaborn utilized matplotlib underneath. You can grab `ax` objects for␣
 ↪further syling.
    sns.lineplot(x=actual_x, y=x.dot(w), color=color, linewidth=linewidth)

lr_data_plot()
lr_model_plot(w)
```



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()`.

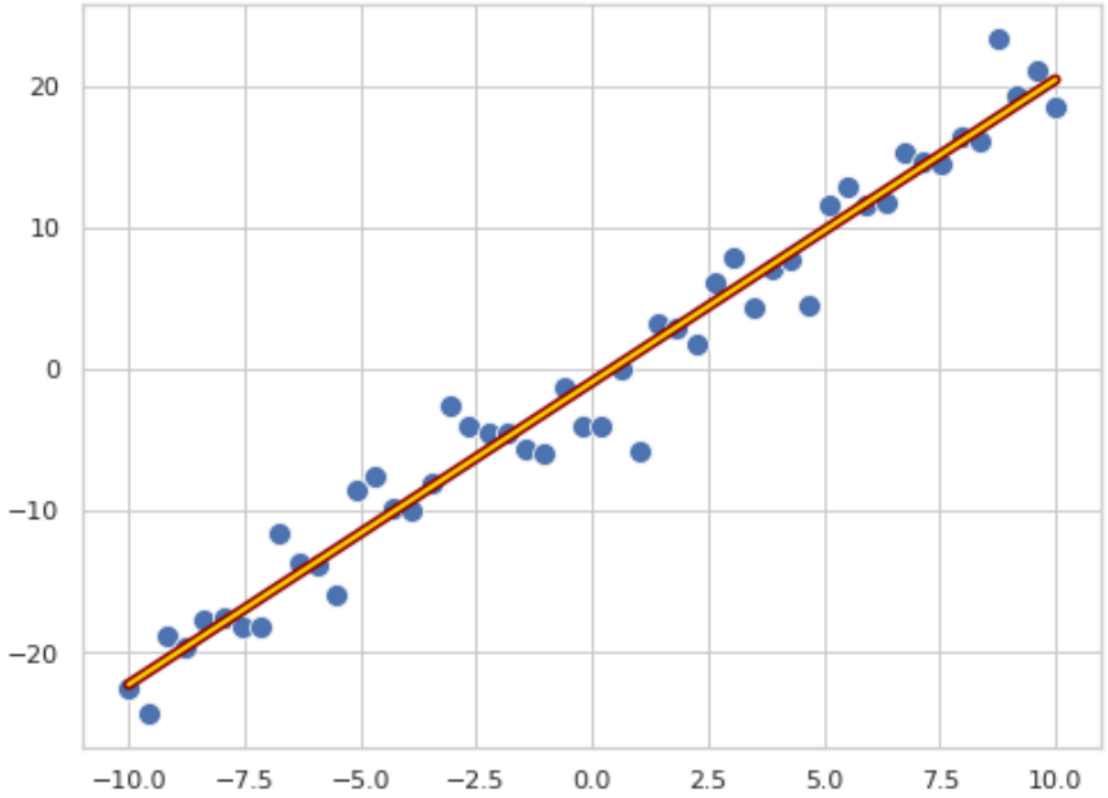
[6]:

```
from sklearn.linear_model import LinearRegression
x_mat = actual_x.reshape((-1, 1)) # do you understand this? can you change it␣
 ↪for the exam?
trained = LinearRegression().fit(x_mat, y)
preds = trained.predict(x_mat)
```

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]:
```
lr_data_plot()
lr_model_plot(w, linewidth=5)
w_skl = [trained.coef_[0], trained.intercept_]
lr_model_plot(w_skl, color='gold', linewidth=2)
```



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

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])
```

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_dset(inputs, classes):
    ret = {}
    for inp, cls in zip(inputs, classes):
        ret[cls] = ret.get(cls, []) + [inp]
    return ret

part = partition_dset(x, y)
```

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 lda_mus(part): return { cls: sum(pts)/len(pts) for cls, pts in part.items()␣
 ↪}
mus = lda_mus(part)
```

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 lda_pis(part):
    tot_len = sum(map(len, part.values()))
    return {cls: len(pts)/tot_len for cls, pts in part.items()}
pis = lda_pis(part)
```

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 lda_sigma(part, mus):
    x_minus_mu = np.concatenate([pts-mus[cls] for cls, pts in part.items()])
    return x_minus_mu.transpose().dot(x_minus_mu) / len(x_minus_mu)
sigma = lda_sigma(part, mus)
assert sigma.shape == (2,2)
```

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
sigma_inv = np.linalg.pinv(sigma)
lda_w = sigma_inv.dot(mus[1] - mus[2])
lda_b = 0.5*mus[2].dot(sigma_inv).dot(mus[2]) - \
        0.5*mus[1].dot(sigma_inv).dot(mus[1]) + \
        log(pis[1]) - log(pis[2])
print(f"w: {lda_w}\nb: {lda_b}")
f = lambda x: lda_w.dot(x) + lda_b
```

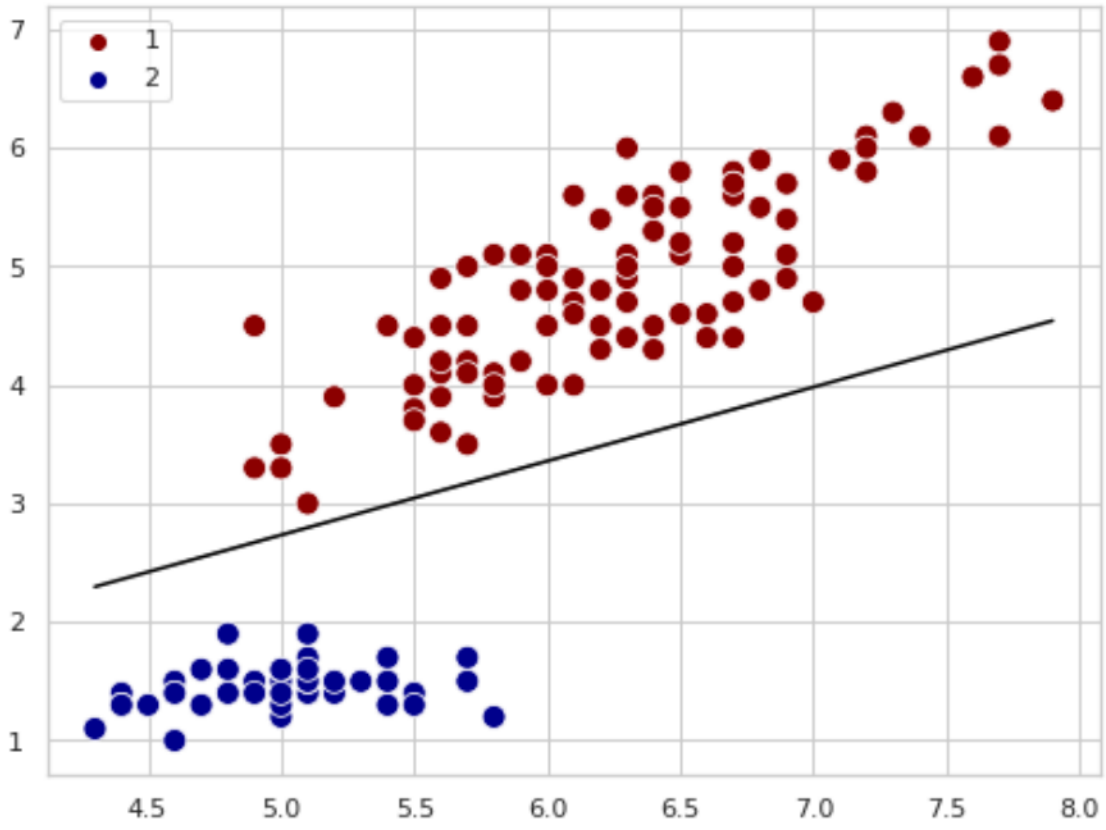
```
w: [-7.96528842 12.76916894]
b: 4.912548207950932
```

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]:
```python
def wb2mq(w, b):
    assert len(w) == 2, "This implementation only works in 2D"
    assert w[0] != 0 and w[1] != 0 and b != 0 # simplify
    return [w[0]/-w[1], b/-w[1]] # m and q

def params2boundary(w, b):
    m, q = wb2mq(w, b)
    print(f"m: {m}, q: {q}")
    return lambda x: m*x + q

def lda_data_plot():
    sns.scatterplot(x=x1, y=x2,
      hue=y, # let's use different colors for the two classes
      palette=sns.color_palette(['darkred', 'darkblue']),
      s=100)

def lda_model_plot(w, b, color='black'):
    model = params2boundary(w, b)
    model_points = [model(inp) for inp in x1]
    sns.lineplot(x=x1, y=model_points, color=color)

lda_data_plot()
lda_model_plot(lda_w, lda_b)
```

```
m: 0.6237906674023641, q: -0.3847194935007181
```

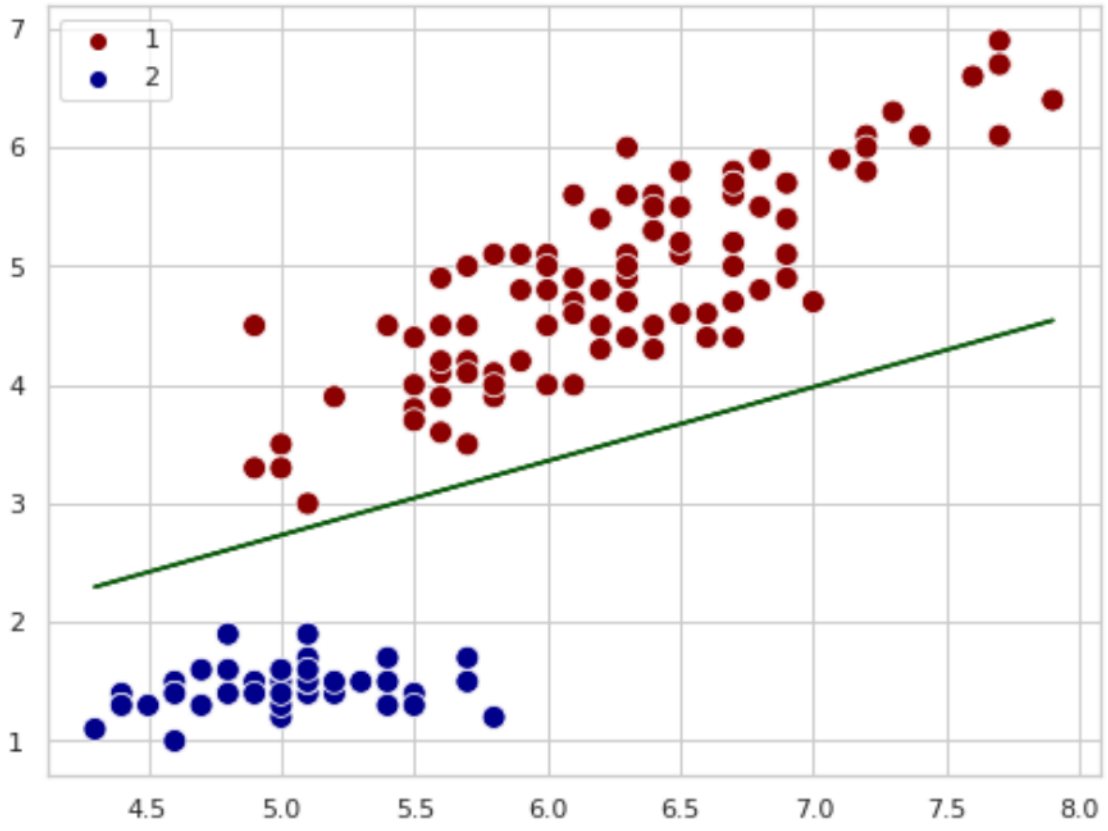


3.7 [1pt] Find LDA on scikit-learn; train a model on the data and add it to the print above (data + model from your implementation). You ned to pass the correct `solver` parameter to the sklearn constructor, check the documentation to understand what I mean. If you do not the result should still look 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
print(f"x shape: {x.shape} y shape: {y.shape}")
trained = LinearDiscriminantAnalysis('lsqr').fit(x, y)
w_skl, b_skl = [trained.coef_[0], trained.intercept_[0]]
lda_data_plot()
lda_model_plot(lda_w, lda_b, color='black')
```

```
lda_model_plot(w_skl, b_skl, color='darkgreen')
```

```
x shape: (150, 2) y shape: (150,)
m: 0.6237906674023641, q: -0.3847194935007181
m: 0.623790667402364, q: -0.38471949350071727
```



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.

assignment_04_solution

March 17, 2023

Please fill in your name and that of your teammate.

You:

Teammate:

1 Introduction

Welcome to the fourth lab. This week the math load is lower on purpose to give you a chance to catch up on the first 3 labs before we start to *use* all the foundations you learned so far. You may want to go back and fix your submissions with the help of the solutions, you will need the material for the exam.

This week we introduce the major data analysis library in Python: `pandas`. You can think of it as providing **feature-rich data containers**: load your data in a `pandas` object and you will have fast access, manipulation, statistics, even math with `numpy` and plotting with `matplotlib`, all well integrated.
Unfortunately it also has a fame of being frustrating, unintuitive and stubborn for the newcomers, so I suggest you spend some time practicing and download the documentation as offline pdf for the exam.

The two main classes are `Series`, for one-dimensional data, and `DataFrame` for *tensors*. You may have heard this word before: you can think of a tensor as a generic structure for feature-based data. A zero-dimensional tensor is a scalar; a 1D tensor is a vector; a 2D tensor is a matrix; you can imagine a 3D tensor as a cube, or as a list of matrices. Higher dimensions are of course hypercubes. You will start mostly using `DataFrame`s with 2D tables/matrices, but some methods will return (kind-of) higher-dimensional tensors (e.g. be careful with `groupby()`! Takes time to grasp).

This library is best learned hands-on, but before starting it is important to understand how the data is stored and accessed.

1.1 The feature's perspective

Let's look again at our dataset of snakes. It has three fields: `head size`, `length` (in cm) and whether it is `poisonous` or not. It looks like this:

```
snakes = [['small', 38, False],
          ['small', 62, True],
          ['medium', 55, True]]
```

This form puts the emphasis on the data points (the rows), which is an intuitive approach at first for humans to manually write down the data. We can scroll to the middle of a dataset in a CSV

or text file and read one line – in principle, if the columns are few and we remember their order.
Accessing the data by indices can also be confusing: if you want to know whether the second snake is poisonous you should use `snakes[1][2]`, which is prone to the type of misunderstandings that lead to bugs (e.g. writing instead `[2][1]`).
From the machine perspective this is not an issue, but this way of storing has the distinct (performance) disadvantage of having **multiple types** in the same data structure: here we have lists with strings, floats and booleans.

An alternative form to represent the data which is also very common (you already encountered it e.g. for data plotting) gives instead priority to the columns or *features*. It contains the same data, but rows and columns are *transposed*:

```
snakes = [['small', 'small', 'medium'],
          [38, 62, 55],
          [False, True, True]]
```

As you can see each feature is not in its own list, and all types are the same, meaning we could store it in a specialized array casted to the correct data type, and enjoy a big performance boost.
Moreover, we can now switch to a hash map (a `dict` in Python), allowing us to include the feature name and use them for indexing, making the data and its description self-contained:

```
import numpy as np
snakes = {'head_size' : np.array(['small', 'small', 'medium'], dtype='str'),
          'length'    : np.array([38, 62, 55], dtype='float'),
          'poisonous' : np.array([False, True, True], dtype='bool')}
```

This has the added advantage that you can now call the vast library of `numpy` methods on your features, such as: `snakes['length'].mean()`. Create a new cell below, copy+paste the code and give it a try!

Data access readability is also improved by using explicit feature names: `snakes['poisonous'][1]` does not leave room for misunderstandings, it is the `poisonous` field of the first data point. Writing `snakes[1]['poisonous']` will now raise an error (remember: errors are your friends when debugging, what is dangerous is a silent bug).
One last feature we could add is to add an explicit extra feature called `index` with incremental numbers for the data points, which allows us to explicitly sort the data or access by index as we did before:

```
import numpy as np
snakes = {'index'     : np.array([1, 2, 3], dtype='int'),
          'head_size' : np.array(['small', 'small', 'medium'], dtype='str'),
          'length'    : np.array([38, 62, 55], dtype='float'),
          'poisonous' : np.array([False, True, True], dtype='bool')}
```

Isn't that neat? Congratulations, you just derived a naïve implementation of a Pandas `DataFrame`. Get used to this name because you will be using it all the time over the next weeks.

1.2 Pandas objects

Let's actually convert our data to a `DataFrame` so we can play with it:

```
import pandas as pd
df = pd.DataFrame(snakes) # the constructor accept most sensible inputs
df.head() # this prints the first few lines of your (potentially large) dataframe
```

A few tips before we start: - DataFrames are composed of `Series` to hold the features: Series wrap the Numpy array to extend with more methods, so anything you can do with a Numpy array can be done with most Pandas structures, and much more. - Data always has an `index` (actually of class `Index`, a special Series), and if you took a database course you should know why (hint: primary key). You can provide an index explicitly, or it will automatically generate one on creation from a counter. - Accessing data with `[]` (e.g. `df['length']`) does not give direct pointer access at the stored data, but returns a copy-on-write handler. This means that creation is free, read access is fast, but trying to write or edit the data first generates a *copy* of the data, and then edits that. The original is unchanged by `[]`. The full truth is actually even more complex, e.g. try creating a new column `df['new_col'] = 5`, this requires no copy) - To access the data directly by column name and index, you should use `df.loc[]` (by index and column) and `df.iloc[]` (by indices). Yes they are both methods but they take square parenthesis (ah, python...). Data access is even more confusing than this, and you will need some serious practice to reliably get Pandas to do what you want. Do not skip on this! I suggest you follow this tutorial. - Sometimes you need to access ranges of values. Ranges in Python are also weird: keep in mind that you can use the explicit `range()` method, or more commonly the `:` format: `start:end:step`. Funny part, all three are optional: `:k` means from the first to the k^{th} element (excluded: the `end` is never included in a Python range, careful); `p:` means from element number `p` until the end; `::2` means (from beginning to end but only) every second element; and most importantly `:` means all elements. This is necessary as `iloc` always requires rows as first argument, so if you want the third column (implicitly: *values of all rows* for the third column) you need to write `df.iloc[:,2]` (remember indices start from 0). Again, complex to grasp, but super easy and obvious once you get it: practice it! - You can access the data by "conditions", by generating a boolean array that will hard-select the rows or columns you want. Yeah even more confusing :) bear with me. Try writing `df['length']<60`: this will return a *boolean numpy array*, with values `[True, False, True]`. This is the answer to the boolean question you asked. Now to get the lines of the dataframe that answered True, you need to write `df[df['length']<60]`. Read it and try it until it makes sense before moving forward. Also do yourself a favor and check also the second part of the tutorial above (boolean indexing): all the time you invest now in Pandas will pay dividends later in the harder assignments. - You can plot the dataframe data directly using the function `df.plot()` (takes `type=` as a parameter), which is sometimes useful for a quick peek. In almost every practical case though it is easier (and produces better results) to simply use `seaborn`, because it integrates DataFrames natively! Finally a good news! try `sns.barplot(data=df, y='length', x=df.index, hue='poisonous')` - Most DataFrame methods also return the result of a computation as a "copy" or "view" on the Pandas object. If you want your method calls to have persistent consequences, which means you want to modify the original data, you should either (i) capture the output in a variable and use that, or (ii) use the optional argument `inplace=True` which is available for most methods and forces modifying the original data.

1.2.1 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 18 points (out of 27 available) to pass (66%).

2 1. Fundamentals

By now you should be familiar with this. We start easy:

1.1 [2pt] Write a small DataFrame with at least 3 of the common problems you can find when dealing with unknown data. If you need inspiration you can reuse and expand your cats and dogs dataset from lab. Loading into a dataframe should be straightforward: here is the documentation, pass the table as `data=` and the labels as `columns=` to the constructor.

[1]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import pandas as pd
sns.set(rc={'figure.figsize':(8,6)}, style="whitegrid")

feature_names = ['likes_dogs', 'agility', 'height']
data = [[True, 8, 25],
        [True, 4, 50],
#         [False, 7, 30],
        [False, 7, None], # missing value
        [False, 9, 20],
        [True, 2, 65],
#         [False, 8, 30],
        ["Maybe", 8, 30], # wrong type
        [True, 6, 25],
#         [True, 8, 40],
        [True, 999, 40], # clipping
        [True, 9, 45],
        [False, 6, 35]]
labels = ['cat', 'dog', 'cat', 'cat', 'dog', 'cat', 'cat', 'dog', 'dog', 'cat']

df = pd.DataFrame(data=data, columns=feature_names)
df['labels'] = labels
df
```

[1]:

	likes_dogs	agility	height	labels
0	True	8	25.0	cat
1	True	4	50.0	dog
2	False	7	NaN	cat
3	False	9	20.0	cat
4	True	2	65.0	dog

```
5      Maybe        8    30.0    cat
6       True        6    25.0    cat
7       True      999    40.0    dog
8       True        9    45.0    dog
9      False        6    35.0    cat
```

1.2 [1pt] Explain with your own word why data defects mitigation is important and why it cannot ever be perfect. Here "own word" means that copying and pasting from the slides will actually fail the question. Feel free to discuss around the concepts if the words come hard, just prove that you got the meaning and that you can talk about it. Write as long as you need to show your skills, but no longer (brevity is also a skill that takes practice).

Mitigation is important because the tolerance to data defect of any ML algorithm is limited, and the algorithm performance depends on the data quality. On the other hand perfect reconstruction of the data is impossible without, well, having the perfect data in the first place. If we had a way of generating perfect data, that would make a great model, and we would not need ML here. Incidentally, if the data we have does not cover the process we want to model, we need to go back to the source and collect new data on purpose: we are using the actual process to generate data of "higher quality" (for our purpose), which we could not do from the data alone.

1.3 [1pt] Give two reasons why to drop lines with defects rather than fixing them.

(i) Faster; (ii) Often we have enough data anyway because of redundancy so it would be a wasted effort.

1.4 [1pt] Why does data quality impact learning more than data quantity? Mention one of the algorithms seen so far as an example. Because basic supervised learning model learn from all data points equally, without bias. There is no way for the algorithms we saw so far to distinguish outliers and decide to "learn less" from them. For example linear regression will return a wildly skewed model if the data has an outlier due to clipping, severely lowering its performance.

1.5 [1pt] Convert (by hand!) a categorical feature from your hand-written dataset into a numerical one using binary encoding. This means that you need to rewrite the hand-written dataset, but a previously categorical feature needs to be written in binary encoding.
If your dataset did not have a categorical feature yet, go ahead and simply add one.
IMPORTANT: you may want to go ahead and fix the defects that you introduced earlier, or you may encounter problems.

[2]:
```
feature_names = ['likes_dogs', 'agility', 'height']
data = [[1, 8, 25],
        [1, 4, 50],
        [0, 7, 30],
        [0, 9, 20],
        [1, 2, 65],
        [0, 8, 30],
        [1, 6, 25],
        [1, 8, 40],
        [1, 9, 45],
```

```
        [0, 6, 35]]
labels = ['cat', 'dog', 'cat', 'cat', 'dog', 'cat', 'cat', 'dog', 'dog', 'cat']

df = pd.DataFrame(data=data, columns=feature_names)
df['labels'] = labels
df
```

[2]:

	likes_dogs	agility	height	labels
0	1	8	25	cat
1	1	4	50	dog
2	0	7	30	cat
3	0	9	20	cat
4	1	2	65	dog
5	0	8	30	cat
6	1	6	25	cat
7	1	8	40	dog
8	1	9	45	dog
9	0	6	35	cat

1.6 [1pt] Consider 5-fold cross-validation: how many models does it train? Now take the first fold: how many of the models that were trained did in fact use this particular fold as part of their training set? 5-fold cross-validation will train 5 models. For any given fold, 4 of those models will use it for training, while the one that didn't will use it for testing.

3 2. Loading data

To simplify the process, here's a CSV string rather than a separate file. You are free to copy this string into a file if you like, the call below simply emulates an Input Output object (such a file) from a String using a `StringIO` object. It is often a useful trick to have on your tool belt.

IMPORTANT: do not edit the string! The defects and problems are on purpose! It simulates problematic data that was passed to you unclean, potentially too big to read for a human, and you need to use only Pandas to load and clean the data.

[3]:
```
from io import StringIO
```

```
csv_str =␣
 ↪StringIO('sepal_length$sepal_width$petal_length$petal_width$species$is_flower\n0$5.
 ↪1$-3.5$1.4$0.2$setosa$True\n1$-3.0$1.4$setosa$True\n2$5.0$-3.6$1.4$0.
 ↪2$setosa$True\n3$4.6$-3.1$1.5$0.2$setosa$I guess\n4$4.7$-3.2$1.3$0.
 ↪2$zetosa$True\n5$5.4$-3.9$1.7$0.4$setosa$True\n6$4.6$-3.4$1.4$0.
 ↪3$setosa$True\n7$5.0$-3.4$1.5$0.2$setosa$Fals\n8$4.4$-2.9$1.4$0.
 ↪2$setosa$True\n9$4.9$-3.1$1.5$0.1$setosa$True\n10$5.4$-3.7$1.5$0.2$setosa$I␣
 ↪guess\n11$4.8$-3.4$1.6$0.2$setosa$True\n12$4.8$-3.0$1.4$0.
 ↪1$setosa$True\n13$4.3$-3.0$1.1$0.1$setosa$True\n14$5.8$-4.0$1.2$0.
 ↪2$setosa$True\n15$-4.4$1.5$setosa$True\n16$5.4$-3.9$1.3$0.
 ↪4$setosa$True\n17$5.1$-3.5$1.4$0.3$setosa$True\n18$5.7$-3.8$1.7$0.
 ↪3$setosa$True\n19$5.1$-3.8$1.5$0.3$setosa$True\n20$5.4$-3.4$1.7$0.
 ↪2$setosa$True\n21$5.1$-3.7$1.5$0.4$setosa$True\n22$4.6$-3.6$1.0$0.
 ↪2$setosa$True\n23$5.1$-3.3$1.7$0.5$setosa$Fals\n24$4.8$-3.4$1.9$0.
 ↪2$setosa$True\n25$5.0$-3.0$1.6$0.2$setosa$True\n26$5.0$-3.4$1.6$0.
 ↪4$setosa$True\n27$5.2$-3.5$1.5$0.2$setosa$True\n28$5.2$-3.4$1.4$0.
 ↪2$setosa$True\n29$4.7$-3.2$1.6$0.2$setosa$True\n30$4.8$-3.1$1.6$0.
 ↪2$setosa$True\n31$5.4$-3.4$1.5$0.4$setosa$True\n32$5.2$-4.1$1.5$0.
 ↪1$setosa$Fals\n33$5.5$-4.2$1.4$0.2$setosa$True\n34$4.9$-3.1$1.5$0.
 ↪2$setosa$True\n35$5.0$-3.2$1.2$0.2$setosa$True\n36$5.5$-3.5$1.3$0.
 ↪2$setosa$True\n37$4.9$-3.6$1.4$0.1$setosa$True\n38$4.4$-3.0$1.3$0.
 ↪2$setosa$True\n39$5.1$-3.4$1.5$0.2$setosa$True\n40$5.0$-3.5$1.3$0.
 ↪3$setosa$True\n41$4.5$-2.3$1.3$0.3$setosa$True\n42$4.4$-3.2$1.3$0.
 ↪2$setosa$True\n43$5.0$-3.5$1.6$0.6$setosa$True\n44$5.1$-3.8$1.9$0.
```

```
print('Data loaded')
```

```
Data loaded
```

2.1 [2pt] Load the data despite the errors.

- Be patient
- Peruse the documentation
- If Python raises an error, read carefully the message at the end of the output cell
- You should not need a `try... except`: make Pandas get the format until there are no errors
- You will need to skip the lines with errors that impede loading (it will still generate warnings, but that is fine for now)
- To print the DataFrame, a simple `df` often looks better in Jupyter notebooks than an explicit `print(df)`

[4]:
```
df = pd.read_csv(csv_str, sep='$', on_bad_lines='skip') # try using 'warn'␣
 ↪instead to see where the problems are!
df
```

[4]:

	sepal_length	sepal_width	petal_length	petal_width	species	is_flower
0	5.1	-3.5	1.4	0.2	setosa	True
1	-3.0	1.4	setosa	True	NaN	NaN
2	5.0	-3.6	1.4	0.2	setosa	True
3	4.6	-3.1	1.5	0.2	setosa	I guess
4	4.7	-3.2	1.3	0.2	zetosa	True
..	…	…	…	…	…	…
145	6.7	-3.0	5.2	2.3	virginica	True
146	6.3	-2.5	5.0	1.9	virginica	True
147	6.5	-3.0	5.2	2.0	virginica	True
148	6.2	-3.4	5.4	2.3	virginica	True
149	5.9	-3.0	5.1	1.8	virginica	True

[148 rows x 6 columns]

2.2 [1pt] Explore the DataFrame, and try to spot the defects. Mitigate two by hand.

You may want to learn about the method `head()`. Also calling `unique()` on a Series will give you the set of different values available. Remember the goal of the question is to fix only two errors for now.

[5]:
```
df.drop(1, inplace=True)
df.loc[4, 'species'] = 'setosa'
df
```

[5]:

	sepal_length	sepal_width	petal_length	petal_width	species	is_flower
0	5.1	-3.5	1.4	0.2	setosa	True
2	5.0	-3.6	1.4	0.2	setosa	True
3	4.6	-3.1	1.5	0.2	setosa	I guess
4	4.7	-3.2	1.3	0.2	setosa	True

```
5             5.4         -3.9          1.7         0.4     setosa      True
..            …           …             …           …       …           
145           6.7         -3.0          5.2         2.3  virginica      True
146           6.3         -2.5          5.0         1.9  virginica      True
147           6.5         -3.0          5.2         2.0  virginica      True
148           6.2         -3.4          5.4         2.3  virginica      True
149           5.9         -3.0          5.1         1.8  virginica      True

[147 rows x 6 columns]
```

4 3. Cleaning the data

The following questions require you to either write a Python function or some automated Python code: manually selecting (like last question) for example rows with missing values is not accepted here, you need to provide code that will work with any (unknown) dataset and rows.
NOTE: no two students ever followed the same process in this exercises, but for those who copy from last year's solutions. As per University policy on plagiarism, students have been expelled from their course of study for something as cheap as an ungraded assignment. Unbelievable. Please not again.

3.1 [1pt] Remove all rows with missing values. This can help (and geeksforgeeks.org is a great resource to keep in mind). Also consider customizing the Pandas display options, for example: `pd.set_option("display.max_rows", 200)` to see more than just a few lines (see more options in the method's documentation).

[6]:
```
df.dropna(inplace=True)
df['is_flower'].unique()
```

[6]:
```
array(['True', 'I guess', 'Fals'], dtype=object)
```

[7]:
```
# For the next question, here are the correct types for the columns:
corr_types = {'sepal_length' : 'float',
              'sepal_width' : 'float',
              'petal_length' : 'float',
              'petal_width' : 'float',
              'species' : 'string',
              'is_flower' : 'bool'
             }
```

3.2 [1pt] Check the column types. Find the reason for the columns having wrong type, and write it. Mitigate the defects and cast the column to the correct type.

[8]:
```
print(df.dtypes)
df = df.astype(corr_types)
print()
print(df.dtypes)
```

```
sepal_length    float64
sepal_width     float64
petal_length     object
petal_width      object
species          object
is_flower        object
dtype: object

sepal_length    float64
sepal_width     float64
petal_length    float64
petal_width     float64
species          string
is_flower          bool
dtype: object
```

3.3 [1pt] Remove all columns with constant values. I used `loc` and a boolean array with `nunique`.

[9]:
```
df = df.loc[:, df.nunique()>1]
df
```

[9]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	-3.5	1.4	0.2	setosa
2	5.0	-3.6	1.4	0.2	setosa
3	4.6	-3.1	1.5	0.2	setosa
4	4.7	-3.2	1.3	0.2	setosa
5	5.4	-3.9	1.7	0.4	setosa
..	…	…	…	…	…
145	6.7	-3.0	5.2	2.3	virginica
146	6.3	-2.5	5.0	1.9	virginica
147	6.5	-3.0	5.2	2.0	virginica
148	6.2	-3.4	5.4	2.3	virginica
149	5.9	-3.0	5.1	1.8	virginica

[144 rows x 5 columns]

3.4 [3pt] There are still errors in the data: fix them all. This time, I suggest you do not just "answer" this question in order. It is easier to move to the next questions, see what errors you get, come back and write code that fixes them (remember: data analysis is an iterative process). If you can answer all questions then there should be no errors left. Also keep in mind that Python 3 strings handle UTF-8 characters if you have not noticed already

[10]:
```
df.loc[:,'sepal_width'] = df.sepal_width.abs()
df.loc[df['species'] == ' ', 'species'] = 'versicolor'
df
```

[10]:
```
     sepal_length  sepal_width  petal_length  petal_width    species
0             5.1          3.5           1.4          0.2     setosa
2             5.0          3.6           1.4          0.2     setosa
3             4.6          3.1           1.5          0.2     setosa
4             4.7          3.2           1.3          0.2     setosa
5             5.4          3.9           1.7          0.4     setosa
..            …            …             …            …       …
145           6.7          3.0           5.2          2.3  virginica
146           6.3          2.5           5.0          1.9  virginica
147           6.5          3.0           5.2          2.0  virginica
148           6.2          3.4           5.4          2.3  virginica
149           5.9          3.0           5.1          1.8  virginica

[144 rows x 5 columns]
```

5 4. Visualizing the data

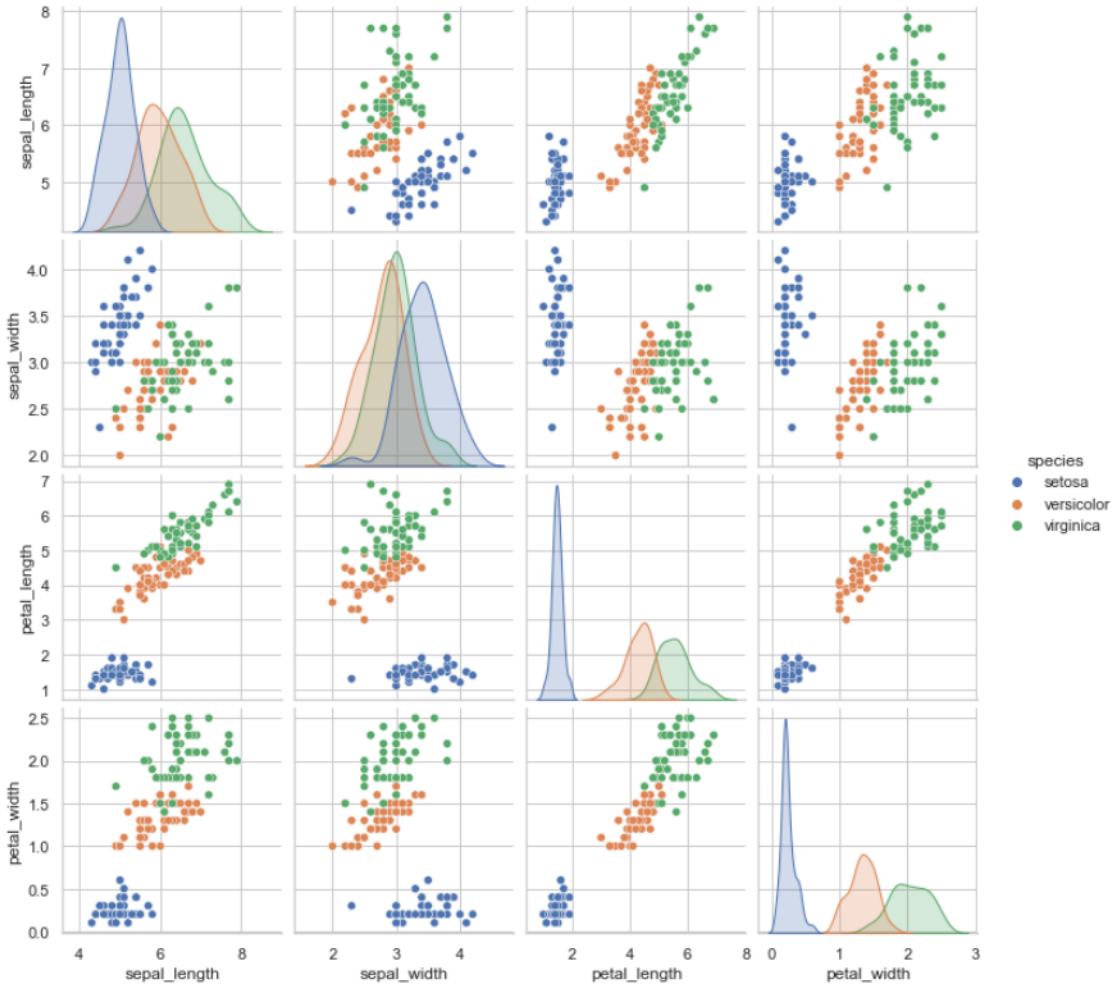
If you are still stuck with some error and need to give up on question 3.4, you can use the correct dataframe from `df = sns.load_dataset('iris')` for the following parts. But it would be really great if you were able to clean the data at the previous question and use that instead. As I mentioned: never underestimate the data cleaning.

[11]:
```
# I hope you will not uncomment the following line; but if you need to, it is␣
 ↪fine, do not worry:
# df = sns.load_dataset('iris'); df
```

4.1 [1pt] Start with plotting the pair-plot, using different colors depending on species. Explain the plots on the diagonal. Hint: a common error in the past has been if the types have been altered by the fixing in 3.4. To avoid that: use `loc` or `iloc` specifying *always* both row and column. To fix it with bruteforce: re-cast the Series to the correct `dtype` as you did in 3.1.

[12]:
```
sns.pairplot(data=df, hue='species')
```

[12]:
```
<seaborn.axisgrid.PairGrid at 0x7f75b0114520>
```

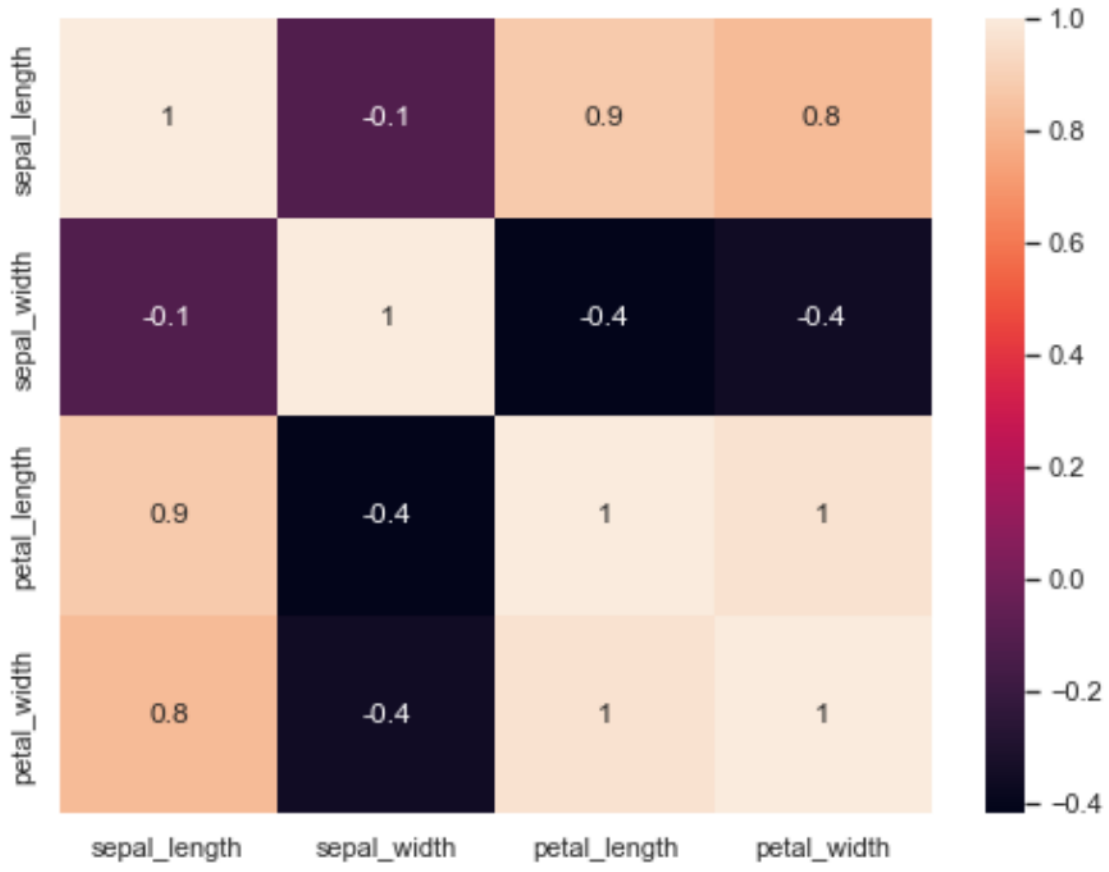


The plots on the diagonal show the distribution of the (one-dimensional) values for each class. Distributions that intersect correspond to data points that are not linearly separable.

4.2 [2pt] Plot the correlation matrix as a heatmap, adding the values on the squares as seen in the lecture's example. Explain what high values of covariance (>0.9 or <-0.9) mean from a statistical perspective, and the implications for the learning process. Hint: could it be a "common data problem"? Also add the parameter `fmt=` to round to 1 decimal.

[13]:
```
sns.heatmap(df.corr(), annot=True, fmt='.1g')
```

[13]: <AxesSubplot:>

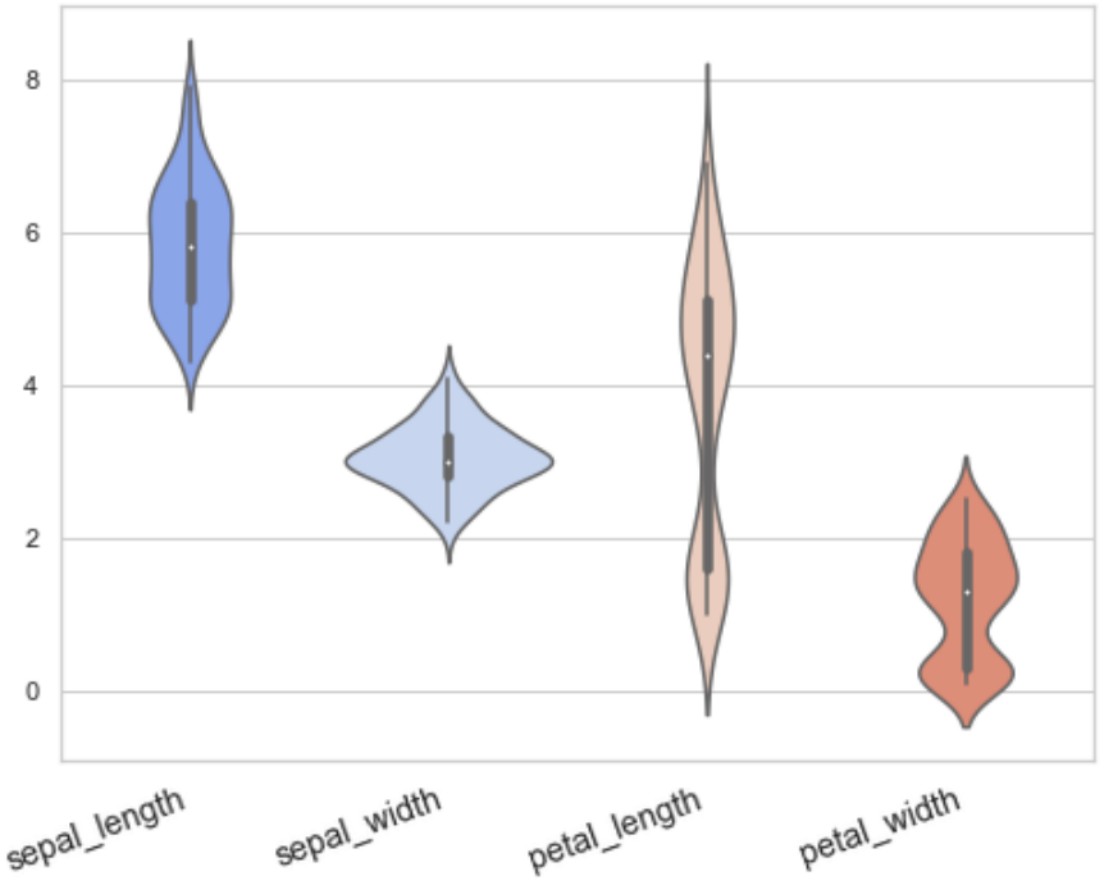


High (linear) correlation (in magnitude: close to $+1$ or -1) means that the two feature vary together in an indistinguishable manner. Here the high correlation between petal width and length means that the petal proportion is constant, and having both is redundant. We can learn the same if we drop one of the two.

4.3 [2pt] Plot a violin plot of the features. Use a custom palette. Briefly explain the advantage of the violin plot over the box plot. Palettes can be found [here]. You can use `plt.xticks(rotation=45)` (or similar) to rotate the x axis labels if they overlap.

[14]:
```
sns.violinplot(data=df,
               palette="coolwarm"
)
plt.xticks(
    rotation=20,
    horizontalalignment='right',
#     fontweight='bold',
    fontsize='large'
)
```

```
[14]: (array([0, 1, 2, 3]),
       [Text(0, 0, 'sepal_length'),
        Text(1, 0, 'sepal_width'),
        Text(2, 0, 'petal_length'),
        Text(3, 0, 'petal_width')])
```



The violin plot offers more details about the data distribution, especially when the feature is effectively composed of multiple clusters (e.g. both petal length and width).

6 5. Visualizing the performance

5.1 [2pt] Use scikit-learn to train a Linear Regression to predict the petal length ($\hat{y}$) from the petal width (x). Generate the Prediction over Observation plot. Careful, the integration between Pandas and Scikit-learn is relatively recent. If you find it confusing, stick to numpy arrays, for example:

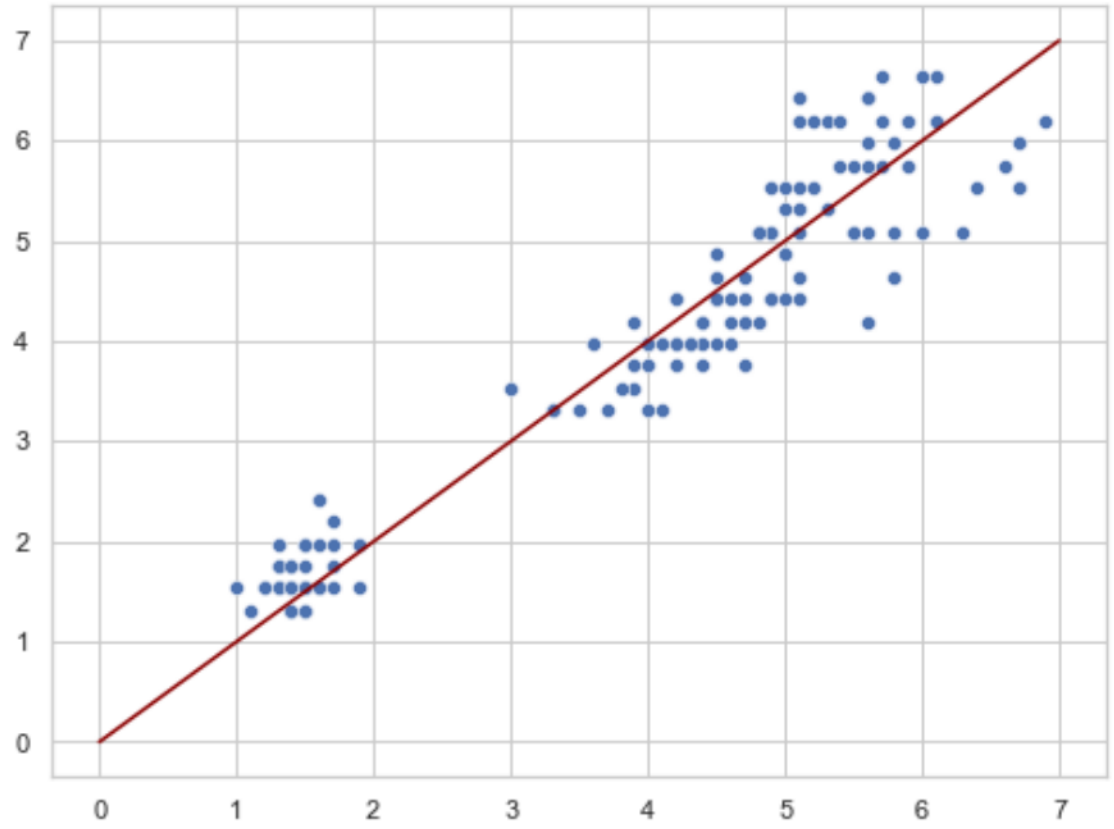
```
x = df['petal_width'].to_numpy().reshape((-1, 1))
y = df['petal_length'].to_numpy()
```

Remember scikit-learn needs 2D input, so you need reshaping. To plot the diagonal line a simple

line plot with two points will generate a segment, e.g. `sns.lineplot([0,7],[0,7])`.

[15]:
```
from sklearn.linear_model import LinearRegression
x = df['petal_width'].to_numpy().reshape((-1, 1))
y = df['petal_length'].to_numpy()
trained = LinearRegression().fit(x, y)
preds = trained.predict(x)
sns.scatterplot(x=y, y=preds)
sns.lineplot(x=[0,7],y=[0,7], color='darkred')
```

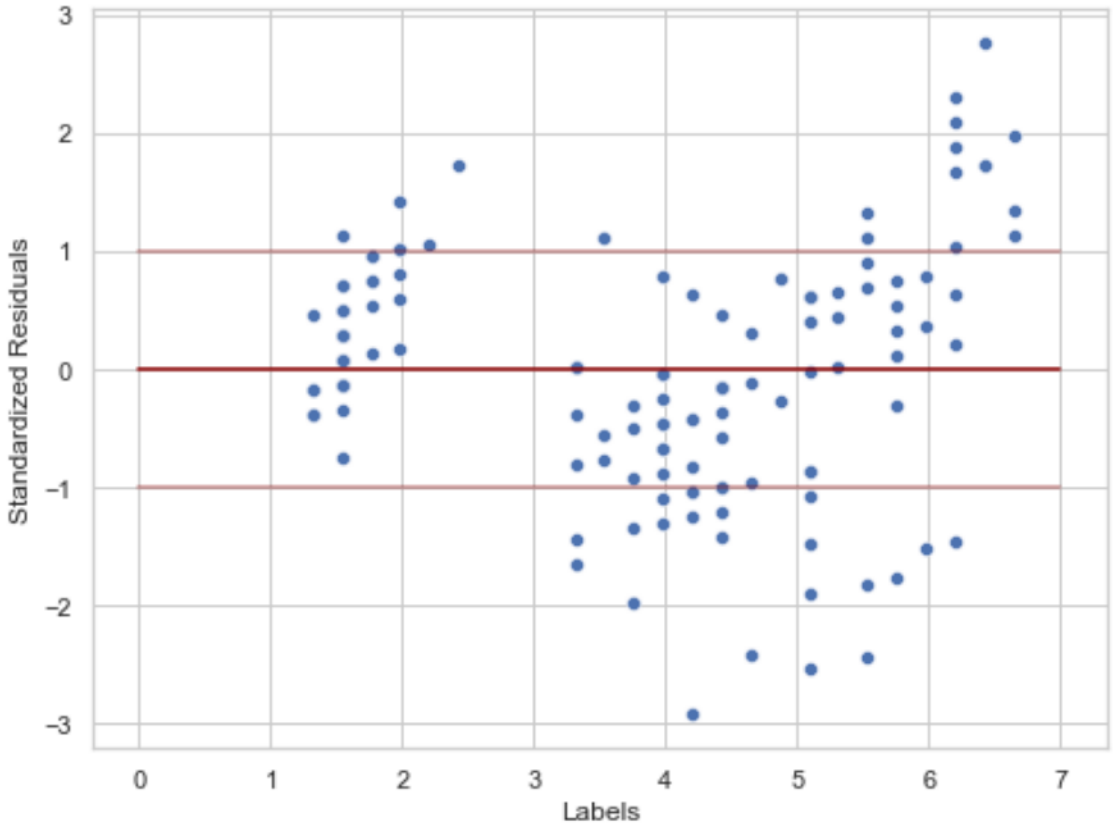
[15]: `<AxesSubplot:>`



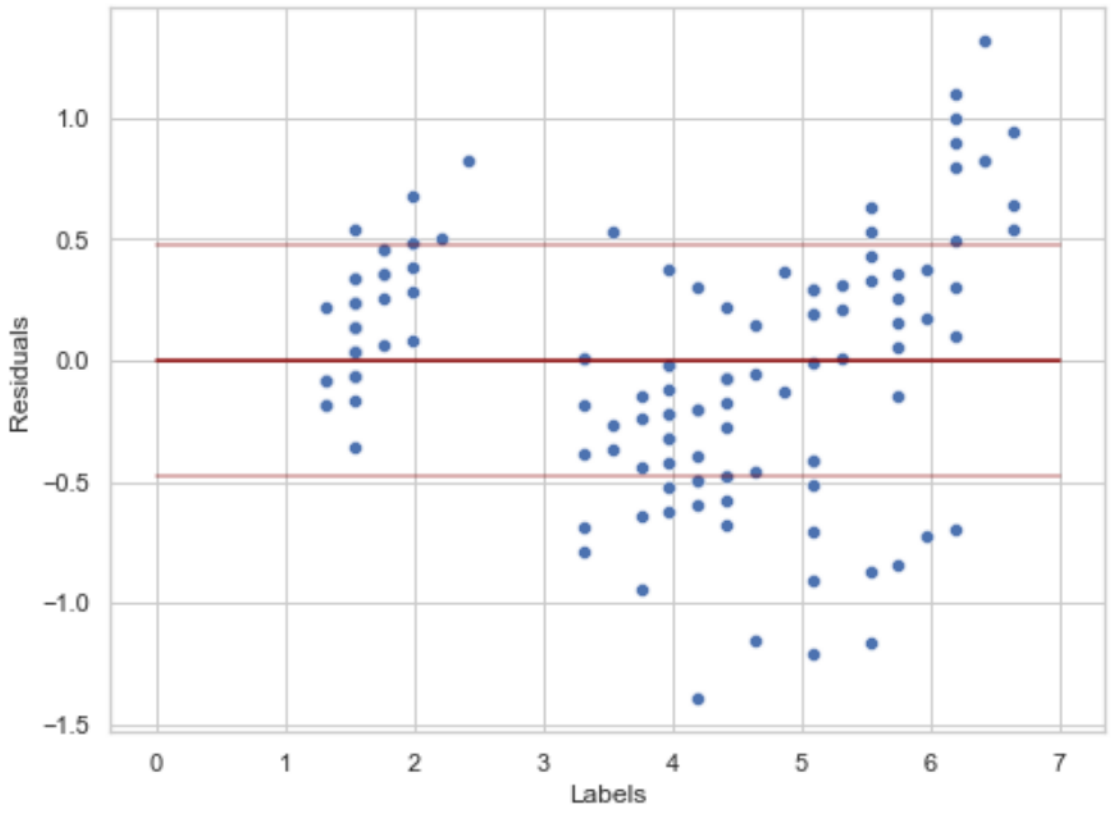
5.2 [2pt] Using the same model and predictions from the last point, generate the Errors over Predictions plot. Here you want at least one horizontal line for $y = 0$: use the segment trick from the previous question. Ideally it looks best with two more lines at $y = +1$ and $y = -1$ for the standard deviation. These can be made thinner or lighter in color to look even better :) I simply used the `alpha` option to make them slightly transparent.

[27]:
```
errors = preds - y
errors = errors / errors.std()
sns.scatterplot(x=preds, y=errors)
```

```
ax = sns.lineplot(x=[0,7],y=[0,0], color='darkred')
sns.lineplot(x=[0,7],y=[1,1], color='darkred', alpha=0.4)
sns.lineplot(x=[0,7],y=[-1,-1], color='darkred', alpha=0.4)
ax.set_xlabel('Labels')
ax.set_ylabel('Standardized Residuals');
```



[29]:
```
# Alternative solution: normalization just changes the axis labels!
# Sometimes having the magnitude of the error helps (note the label)
errors = preds - y
std = errors.std()
sns.scatterplot(x=preds, y=errors)
ax = sns.lineplot(x=[0,7],y=[0,0], color='darkred')
sns.lineplot(x=[0,7],y=[std, std], color='darkred', alpha=0.4)
sns.lineplot(x=[0,7],y=[-std,-std], color='darkred', alpha=0.4)
ax.set_xlabel('Labels')
ax.set_ylabel('Residuals');
```



5.3 [2pt] For both Prediction over Observation and Errors over Prediction, write a few sentences commenting on the performance of the model learned by the Linear Regression.

- Prediction over Observation: the points correctly cluster around the diagonal, showing that the prediction minimizes the risk. There are no outliers.
- Errors over Prediction: Most error is in the range between -1 and 1 with few outliers out by little. The model's predictions will have an expected noise $\epsilon \sim \mathcal{N}(0, 1)$.

7 At the end of the exercise

Bonus question with no points! Answering this will have no influence on your scoring, not at the assignment and not towards the exam score – really feel free to ignore it with no consequence. But solving it will reward you with skills that will make the next lectures easier, give you real applications, and will be 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 :)

BONUS [ZERO pt] Follow this tutorial to augment your answers to point 5 with 5-fold cross-validation.

BONUS [ZERO pt] Follow this tutorial but using our data, and generate all the plots of types that have not been mentioned in the lecture/lab. Comment a few sentences about which I think would be most useful to include in our set of basic plots for the next year.

BONUS [ZERO pt] Copy your Perceptron implementation from the second lab. Modify it so that at every update it computes the Risk (total Loss over all points) and saves it to a list (we did something similar at the first lab, remember?). Run your implementation on the data, then plot the Risk over Iterations, using a `sns.lineplot()`. If you discuss these points on Moodle I will participate :) plus it will give you an opportunity to find other people in the class that interested with going beyond the points: I strongly encourage these to team together (remember you can change teammate at each assignment).

7.0.1 Final considerations

- This lecture+lab actually gives you most of what you need for the high-pay job of Data Analyst, so I guess there is one direct market application from this course after all :)
- You will need to be strong in all the topics seen so far to carry on with the course.
- Fixing and polishing your past assignments will make passing the exam so much easier. Now is the perfect chance to prepare the first 3.

assignment_05_solution

March 17, 2023

Please fill in your name and that of your teammate.

You:

Teammate:

1 Introduction

Welcome to the fifth lab. Last week we had a break from math-heavy assignments to allow you to catch up on the fundamentals and tools seen so far. Well, as far as learning Pandas could be considered a break, anyway. We will learn more of this library over the following weeks, I hope you will develop an appreciation for its use over time.
"You may find its methods disagreeable, but you can't avoid appreciating the results" (cit.)

Spoiler alert: I lied last week, actually IMO Pandas' most confusing feature is `groupby()` :) but you got `loc` and `iloc` already in your pocket now, right?

1.1 Grouping in Pandas

It is time to introduce some of the math applications of Pandas DataFrame and Series, and to the unfriendly-but-oh-so-useful `groupby()`.

From now on we will be using Pandas containers for our data, even directly for the math calculation. Remember that they wrap around Numpy arrays (and you know how to use those now) while giving convenient indexing and extra capabilities. No need to e.g. split the points based on their class into a `dict` as we did for LDA: we can simply *group* data by label, then all operations will work on the whole feature arrays, and run for all classes at once (and using the underlying, faster C implementation).

Main hint: simply treat a DataFrame just like you would a multi-dimensional Numpy array. Just think of it as a matrix, as you already had, generalized to higher dimensions: a **tensor**. A Pandas Series on the other hand is just a one-dimensional Numpy array (vector). In either case, function calls will be *broadcasted* to its elements: this means that they will run independently on each element and their results will be aggregated back in a data structure of the same type (list, Series or DataFrame).

We will use `groupby()` extensively from today: typically it takes a while to grasp first, but your code will be legible, and you will need (almost) no more `for` loops nor `dicts`.
Careful though because the method returns a special `GroupBy` object, that is somehow unwieldy: **it removes a feature, adds a dimension (the grouping), and does not print directly from its output**. Read this last sentence a couple of times, then again *after* you start playing around

with `groupby()` (seriously, write it on a post-it or something), and it will help to slowly make sense of this.

Initially, try to follow each call to `groupby()` with a `describe()`, to really see what is happening. Also `print` the `groups` for an intuition on how the grouping works: it's basically a `dict` from each element of the "group" (e.g. the classes) to the `Index` values of the corresponding rows. With LDA we already did something similar, by hand, with a `dict` hashing the classes to the actual data (less efficient than using indices).

A `GroupBy` object is just an implicit (because performance) description on how to split/group the data: any operation you call on it will return multiple results, *one per group*, instead of just one value. This is a sort of "automated mapping", aka broadcasting. Go ahead and play with it a bit: understanding this point is necessary to solve the next questions.
The trick that did it for me, was to try and ignore its output *per se* (I don't even print it), imagine it's just a fancy version of LDA's per-class `dict` we built by hand, and just call functions on the output since *their* output makes sense again for me.

From this point on, it is important that you need to start thinking of the dataset as a whole, single, high-dimensional entity, not just a list of points. It's a forest, not trees. Explore it by selecting and slicing this object from different perspectives, as if you were "floating around it in space" rather than being stuck to "read one row at a time". When you use a DataFrame for math, just remember that you are manipulating multiple variables at the same time, and you will get vectorial answers: treat it like a special Numpy data structure, and everything should eventually become intuitive.

1.1.1 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 12 points (out of 18 available) to pass (66%).

[1]:
```
# Let me hit ctrl+c ctrl+v for you one last time
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import pandas as pd
sns.set(rc={'figure.figsize':(8,6)}, style="whitegrid")
```

2 1. Fundamentals

This time we start strong with an example that is simple but longer. Take your time to read and understand each part, follow the suggestions, and it should unravel without much trouble.

1.1 [4pt] You want to calculate the reliability of a weather forecast service. In the current season, you get rain on 25% of the days. You know that 10% of the time they forecast rain and it does not rain. You also know that 5% of the time they forecast good weather and they are wrong. Using Bayes' rule, calculate by hand the probability that one day it is going to rain given that they earlier forecasted rain. I suggest you proceed as follows: (i) fill the data you know in an events probability table, as seen in the lecture; (ii) your events are whether it is going to rain or not, and whether the forecast predicted rain or not; (iii) remember that probabilities sum to a constant over all possible events, so fill in the blanks in (a copy of) the table; (iv) state very clearly what are the posterior, prior, likelihood and evidence; (v) only assemble your Bayes' equation and calculate the numbers, once you are certain of your components.

For the target day, let us call `R` the event of *having rain*, and `F` the event of the weather services *forecasting rain*.
We want to compute the probability of having rain given the forecast of rain, i.e. $P(R \mid F)$.
We know that:

- 10% of the times they forecast rain and it does not rain
- 5% of the times they forecast not rain and it does rain
- A priori, we have 25% chance of rain
- The total probabilities must always sum up to 100

We can fill in the following squares in our probability table:

	F	not F	sum
R		5	25
not R	10		
sum			100

Then we can fill-in the blanks by deducing the missing terms with simple math (tak it one step at a time):

	F	not F	sum
R	20	5	25
not R	10	65	75
sum	30	70	100

We also know from Bayes' theorem that:

$$P(R \mid F) = \frac{P(F \mid R)\, P(R)}{P(F)}$$

Here are the components for our formula:

- The **likelihood** $P(F \mid R) = P(F, R) \,/\, P(R) = 20/25$
- The **prior** $P(R) = 25$
- The **evidence** $P(F) = 30$

So we can calculate our answer as:

$$P(R \mid F) = \frac{P(F \mid R)\, P(R)}{P(F)} = \frac{20/25 \cdot 25}{30} = 66.6\%$$

Alternatively, since we were able to construct all the data, we could also just use the simplified version:

$$P(R \mid F) = \frac{P(R,\, F)}{P(F)} = \frac{20}{30} = 66.6\%$$

Notice however that the joint probability $P(R,\, F)$ is in many cases unavailable. In most cases, it is easier to estimate the likelihood, prior and evidence independently. The second approach is available here only because we were able to first fill in all probabilities in the table, but don't expect this to necessarily happen at the exam.

1.2 [1pt] Explain $\hat{y} = \mathbf{arg}\max_{y \in Y} \{P(y \mid x)\}$. Naïve Bayes predicts the class of an input x as: the class that maximizes the conditional probability of the hypothesis (class y being the class that generated x), given that we observe x.
In practice this means that NB classification computes the probabilities for each class to have generated x, then returns the class corresponding to the highest probability – which is the exact same concept we saw behind LDA. What changes is in how the classes are modeled.

1.3 [1pt] How does NB differ from LDA in regards to the covariance of the distributions used to model the data?

- LDA maintains a single, dense covariance matrix constructed from all features of all points of all classes.
- NB instead maintains a different distribution for each feature for each class.

This means that NB does not maintain covariance information between features, but has more flexibility on modeling more precisely each feature and distinguish the variance information of each feature in each class independently.

3 2. Model Selection for Naïve Bayes

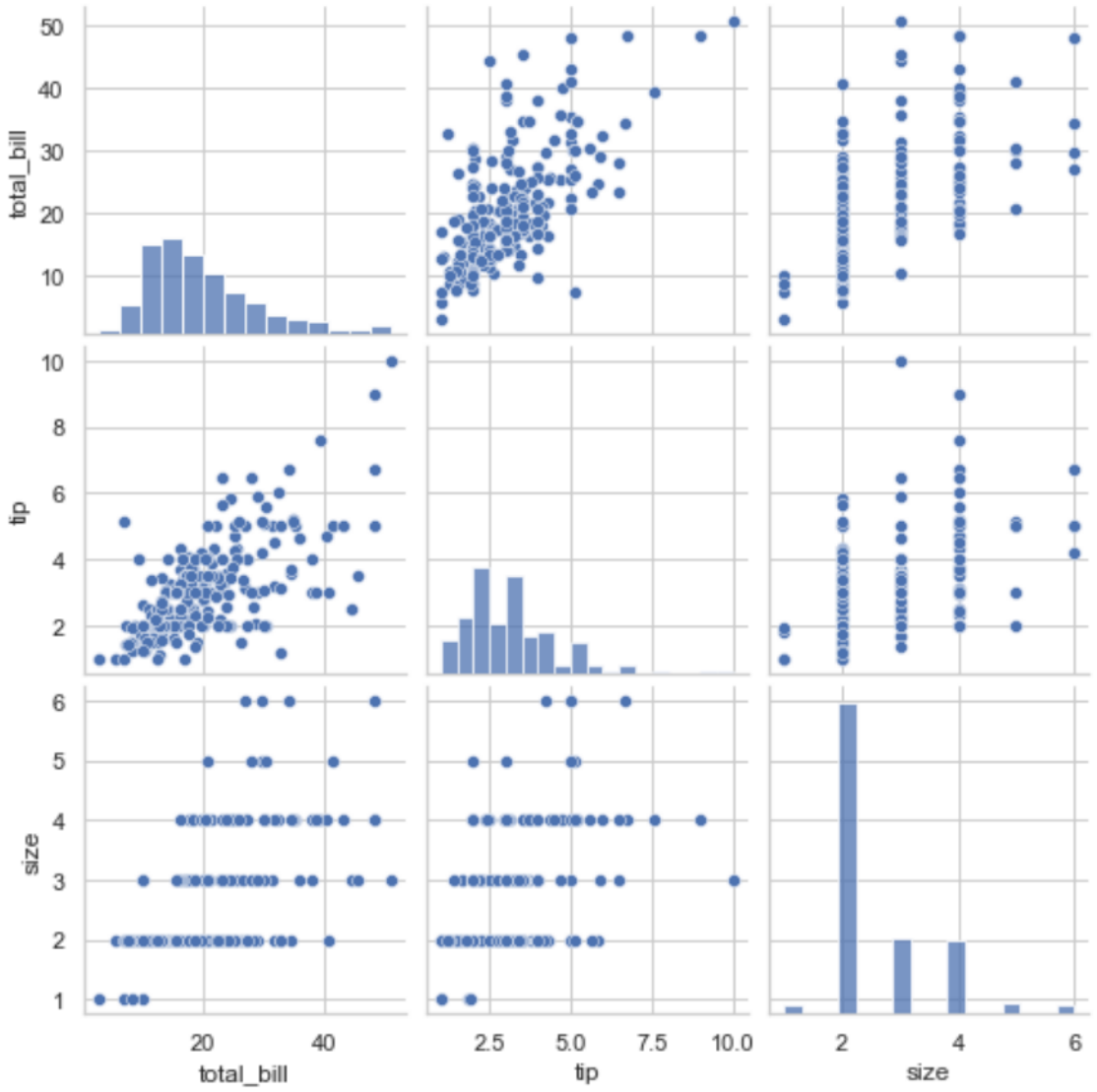
2.1 [3pt] Load the `tips` dataset from Seaborn (into a Pandas DataFrame). Which distribution would you use to model each of the features in the dataset? Explain your choices. You load the dataset the same way you did for `iris` before. Obviously you need to study it to be able to answer. You should find useful to consider (i) the list of dtypes for each feature, (ii) the number of unique values for each of the categorical features, (iii) you can use the `pairplot` to quickly inspect the data: can you do better than a simple Gaussian if there are multiple peaks or asymmetry in the distribution of the real-valued features?
The code cell below is to hold your analysis, while the real answer + motivations go in the Markdown cell just underneath.

[2]:
```
df = sns.load_dataset('tips')
print(df.dtypes)
```

```
print()
print(df.loc[:, df.dtypes == 'category'].nunique())
sns.pairplot(df)
```

```
total_bill     float64
tip            float64
sex           category
smoker        category
day           category
time          category
size             int64
dtype: object

sex       2
smoker    2
day       4
time      2
dtype: int64
```

[2]: <seaborn.axisgrid.PairGrid at 0x7fc20473f5b0>



I can use a Gaussian model for `total_bill`, `tip` and `size` since they have numerical values (they are `float64`, though I would expect any negative or high-precision decimals to be defects). I could use simple Gaussians since they all seem to be single-peaked, but using Mixture of Gaussians would give better precision because they are all asymmetrical (probably $g = 2$ would suffice). For `sex`, `smoker` and `time` I can use Bernoulli distributions since they are all binary features. Lastly, `day` is categorical but with 4 values, so I can apply a binomial distribution.

4 3. Naïve Bayes Classification

Let's write a Naïve Bayes classifier from scratch. We will work with the `iris` dataset (again, from Seaborn) since we know already the data. All features are continuous: for simplicity we can use simple Gaussians, but we should expect some misclassification.

From now on let's also introduce the train-test split so we can start verifying our model's performance the right way. Just use `train` for your answer instead of `df`, and leave `test` for the end.

[3]:
```
df = sns.load_dataset('iris')

from sklearn.model_selection import train_test_split
train, test = train_test_split(df, test_size=0.2) # 80-20 split
```

3.1 [2pt] Compute the priors for the three classes of the Iris dataset using the Pandas DataFrame, *in a single line of code* and without using loops (`for`, `while`, etc.). One-liners are typically bad practice (remember: readability first!), but here I need to force you to learn this new tool and stop writing `for` loops, since they will not scale from now on.

Careful as many tutorials online (such as this one will explicitly select the class and run the same calculation multiple times (and in multiple lines). This approach **does not scale** to problems with 100 or 10'000 classes: learn to use `groupby()` instead!
[Think: this course should make you confident enough to be the one writing the tutorials, and hopefully of much better quality!]

As a reference, you will need to (i) group the dataframe by species, (ii) select only the grouped elements (returning a Series), (iii) run the Numpy-backed `count()`, (iv) divide by the total number of elements. If you get lost on the `groupby`, try this: `groupby(feature_name)[feature_name]`.

And yes it's not a problem to add a print statement in a second line :)

[4]:
```
# Think: what's the problem with using `mean()` directly?
priors = train.groupby('species')['species'].count() / len(train)
priors
```

[4]:
```
species
setosa        0.350000
versicolor    0.283333
virginica     0.366667
Name: species, dtype: float64
```

3.2 [1pt] Compute the means and the standard deviations for each feature and for each class of the Iris dataset using the Pandas DataFrame (one line of code each). As a reference, you should obtain 12 means and 12 standard deviations. Again, the use of `groupby` followed by Numpy's functions will take literally 2 lines and no loops. Remember to use the `train` data!

[5]:
```
means = train.groupby('species').mean()
stds = train.groupby('species').std()
[means, '-'*63, stds] # why 63? :)
```

[5]:
```
[            sepal_length  sepal_width  petal_length  petal_width
 species
 setosa          5.021429     3.428571      1.466667     0.247619
```

```
versicolor      5.982353     2.785294      4.300000     1.347059
virginica       6.604545     2.984091      5.559091     2.027273,
'------------------------------------------------------------',
            sepal_length  sepal_width  petal_length  petal_width
species
setosa          0.371244     0.399564      0.160284     0.110956
versicolor      0.526550     0.309577      0.480530     0.177926
virginica       0.643376     0.306475      0.565816     0.268816]
```

Here is a freebie to save you some debugging time: the (stunted) equation for the Gaussian probability. Stunted in the sense that, since it is only used to maximize the class probability, parts that do not depend on the class have been dropped (as usual). It requires you to define first the variables `means` and `stds` from the previous question (both (3×4) DataFrames).

If you really want to understand what is going on (especially with Pandas), I challenge you to comment it out, pull the slides, and write your own. You did something very close for LDA, feel free to review your code. You don't need it to look identical as long as it does the same job.

Remember that Naïve Bayes computes the class likelihood as a product of the independent probabilities for each feature: this is done by the `product()` on the columns. If you remove that, you should have 12 values (give it a try).

When passing a line of input to `likelihood` be careful to remove the last column (the `species`) as in the example below (in our previous calculations this was done by the `groupby()`, which made a new dimension out of it).

Also something that can be important: sometimes `iloc[]` converts the type of the data slice, so you can have errors because a function cannot be broadcasted. In that case, remember that calling `.astype('float')` will force the dtype to `float` and address some of these errors. This is not the most elegant solution, I will leave it to you to find a better one ;)

[6]:
```
# What do you think of the style of this Gaussian one-liner?
likelihood = lambda x: (np.exp(-(x-means)**2/(2*stds**2))/stds).product(axis=1)
likelihood(train.iloc[0, :-1].astype('float'))
```

[6]:
```
species
setosa        2.123420e-205
versicolor     3.943315e-07
virginica      6.259911e+00
dtype: float64
```

3.3 [1pt] Write a Python function that takes a single line of input x and returns the prediction of its class $\hat{y}$. Run it on the same data point as the example cell above. Is the prediction the same as you would have from the cell above? Why / why not? As a sanity check: it should take a row as input (without labels, as for `likelihood` above) and return the string found in the `index` of the max value (the documentation is your friend).

[7]:
```
# What's the diffrence between using `:-1` and `:` (iloc below)?
predict_class = lambda x: (priors * likelihood(x)).idxmax()
```

```
predict_class(test.iloc[0, :-1].astype('float'))
```

[7]:
```
'versicolor'
```

The prediction is different because while the likelihood points to the `virginica` species, this does not take into account the *prior*. Once the prior is taken into account in the `predict_class` function, the prediction becomes instead `setosa`.

3.4 [2pt] Compute $\hat{y}$ for all points in the `test` dataset, in one line and without using Python loops (`for`, `while`, etc.). Compare it with the correct label y and print the number of misclassified points. And here is how you use the test set: after the training on the train set is complete, you evaluate its performance on data it was not trained on. This is absolutely **crucial** in machine learning. We will use this process from now on, and using the wrong dataset (either for training or testing) will be considered a major error (so careful with typos! Double-check every time!). If you wonder why so strict, check again the 4th lecture and ask yourself what are the consequences of getting it wrong in a work or research setting (and feel free to discuss on Moodle).

Again, no loops: you need both to drop the last column and then to apply the function to the rows. For example: `train.iloc[:, :-1].apply(my_predict_fn, axis=1)`. Can you make it look nicer/more readable?

Remember you can count the number of `True` values in a numpy array simply by calling `sum()` on it.

[8]:
```
preds = test.iloc[:, :-1].apply(predict_class, axis=1)
print(f"Misclassified: {(preds != test['species']).sum()}/{len(preds)} points")
```

```
Misclassified: 2/30 points
```

3.5 [1pt] Why did we not compute (nor need) the *evidence* for predicting the input's class? Because the evidence does not depend on the class and can thus be dropped from the argmax.

3.6 [2pt] Train a scikit-learn Naïve Bayes Gaussian classifier on the Iris train data using a Pandas Dataframe, and print the number of misclassified points on the test data. Remember that: - Now that we have a bit more experience with Pandas we can learn how to pass the DataFrames directly to scikit-learn. - The training data should always be 2D (i.e. DataFrame) and not have the label (`train.iloc[:,:-1]`, do you know what each `:` stands for?). - The labels should always be 1D (i.e. Series) and numerical. Rather than doing the conversion manually, you should convert the feature to categorical and then use its codes (`train['species'].astype('category').cat.codes`). - Mistakenly testing on the train set will fail the question, as will comparing the prediction against the train set labels (hint hint). - You will probably get better results with scikit-learn because it uses multivariate Gaussians and improved estimators (check the documentation).

[9]:
```
from sklearn.naive_bayes import GaussianNB
x_train = train.iloc[:,:-1]
y_train = train['species'].astype('category').cat.codes
```

```
x_test = test.iloc[:,:-1]
y_test = test['species'].astype('category').cat.codes
y_pred = GaussianNB().fit(x_train, y_train).predict(x_test)
misclassified = (y_test != y_pred).sum()
print(f"Misclassified: {misclassified}/{len(y_test)} points ")
```

```
Misclassified: 2/30 points
```

5 At the end of the exercise

Bonus question with no points! Answering this will have no influence on your scoring, not at the assignment and not towards the exam score – really feel free to ignore it with no consequence. But solving it will reward you with skills that will make the next lectures easier, give you real applications, and will be 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 :)

BONUS [ZERO pt] Do a bit of independent research, and propose below the simplest example you can, to make evident how the frequentist and Bayesian approaches are different. I advise against blind copy+paste from the Internet in this case, I have seen so many incorrect opinions and tutorials over the years it is frankly ridiculous. I suggest you rather argue a bit on the Moodle about the approaches themselves, so you can make sure your example is correct.

A good intro: [link].

BONUS [ZERO pt] Train a Gaussian NB (either your code or scikit-learn) on the full Iris dataset (no train-test split) and check the misclassifications. Train the same on the 80% training data, then check and aggregate misclassifications both on the train and test datasets. You will probably get the same number of total errors regardless of whether you trained on 80% or 100% of the data. Can you explain why? The reason was mentioned in the last lecture. Feel free to play with different splits until you find how low can you go with the training before increasing the number of errors. Use the term `statistically representative` in your explanation.

5.0.1 Final considerations

- This is the first core ML method we are covering in the course. As you see you need to know quite a few concepts before we can really discuss its inner workings.
- On the other hand, you now already own most of the glossary and knowledge needed, so you only need to put it all together. The rest of the course will follow this same pattern.
- This is also your first method capable of *nonlinear classification*. Notice how LDA used non-linear models for the data (Gaussian clusters) but still relied on linear separation boundaries (weirdly obtaining m and q from class-pair inequalities)? NB can work with multiple classes and different types of distributions (think Mixture of Gaussians), the division boundary is not (necessarily) a line anymore.

- In the next two lectures we will start learning about one of the bigger classic ML tools still state-of-the-art today: the Support Vector Machine, and the Kernel Trick. We are reaching the "cruise speed" level of complexity for the course, we will stay close to this level until the exam. Keep up with the lectures and exercises and you should have no trouble. Good luck!

assignment_06_solution

April 7, 2022

Please fill in your name and that of your teammate.

You:

Teammate:

1 Introduction

Welcome to the sixth lab. Last week we attacked classification problems expanding the concept of LDA, interpreting the data as generated from probability distributions, into Naïve Bayes. Today we follow the same plan from the perspective of the *margin*, starting with the Perceptron expanding it into the Support Vector Machine.

You will notice we will not implement the learning algorithm this time, because SVMs are solved through quadratic programming and that is not fun. But please feel free to check out quadratic programming yourself if you are interested.

Today we focus on the ample new glossary, then implementing the SVM objective function, and using off-the-shelf optimizers for training. Enjoy!

1.1 Load your libraries

FROM NOW ON: take a habit of loading your default libraries, plotting style and external tools at the top of the file (as I do here). I will not mention it further in the next labs, it will be up to you to load what you need.

[1]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import pandas as pd
sns.set(rc={'figure.figsize':(8,6)}, style="whitegrid")
```

1.1.1 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 18 points (out of 27 available) to pass (66%).

2 1. Fundamentals

1.1 [1pt] Write the equation for the margin (use latex).

$$y_i \cdot f(x_i)$$

1.2 [2pt] Write the equation for the maximum margin separation (use latex), then explain it in English below.

$$\underset{w,b}{\text{argmax}}\{\gamma\} \quad \text{s.t. } \forall i \in \{1, \dots, n\} : y_i \cdot (\langle w, x_i \rangle + b) \geq \gamma \ , \quad \|w\| = 1$$

Our goal is to maximize the boundary margin γ w.r.t. parametrization (w, b). The decision boundary is $\langle w, x_i \rangle + b$, and the margin is thus $y_i \cdot (\langle w, x_i \rangle + b)$. We want the margin to be greater than γ for all points in the (training) dataset D, while maintaining the norm-2 of w equal to 1.

1.3 [1pt] When is an *example* (in a dataset) considered a Support Vector? Write its definition using latex, then explain it in English. (x_i, y_i) is a Support Vector if $y_i \cdot f(x_i) = 1$. By definition this means that its distance from the decision boundary is $\gamma = 1/\|w\|$, which is $= 1$ for canonical SVM as $\|w\| = 1$.

1.4 [1pt] Explain in English why SVMs ignore examples that are not Support Vectors. Because their margin will be greater than γ and so ignored in the definition of γ: the boundary margin is the smallest across all point margins. Therefore in the SVM objective function, γ will be independent from all those margins, and they are dropped out of the optimization. Later we see the term $\|w\|^2$ rather than γ, but remember that the two terms are swapped in an equivalence (since $\gamma = 1/\|w\|$).

1.5 [2pt] Define each of the following concepts (in English): Quadratic Program, Slack Variable, Multiobjective Optimization, with your own words. Tip: if you want to write something with your own words, you can read a description of it, then put the description away and immediately try to explain it again as if to someone that has no background on it. This of course only works if you have understood the concepts, else you'll find yourself just changing words but reusing the sentences: please avoid that (as the goal is to force you to understand the concepts). Incidentally, this is also why studying in pairs/groups works (typically) better: explaining concepts to each other forces you to confront how much did you really understand of it.

- *Quadratic Program:* an optimization problem where the objective function is quadratic (in its decision variables), and the constraints are linear (equalities and) inequalities. It is typically designed to be convex.
- *Slack Variable ξ_i:* the distance between a point that violates the margin and its correct classification margin (i.e. the size of the violation).

- *Multiobjective Optimization:* optimization problem with multiple, contradictory goals.
  (if all goals agree, they can be summed together and optimized as a single objective)
  An optimization problem minimizes or maximizes a quantity defined by a function by changing one or more of the function parameters.

1.6 [3pt] Write the full formula for the Soft Margin SVM (geometric approach; use latex), then explain in English the role of the hyperparameter C (particularly: what do high or low values mean for the SVM).

$$\underset{w,b,\xi}{\operatorname{argmin}}\{\frac{1}{2}\|w\|^2 + C \cdot \sum_{i=1}^{n} \xi_i\} \quad \text{s.t.} \quad \forall i \in \{1, \dots, n\} : y_i \cdot (\langle w, x_i \rangle + b) \geq 1 - \xi_i \quad \text{and} \quad \xi_i \geq 0$$

The C hyperparameter allows SVM users to trade off the relative importance between margin size maximization and total slack minimization. High C values give higher importance (priority) to the margin violations over the margin size; low C values the opposite.

1.7 [2pt] Write the objective function for the Soft Margin SVM (use latex) using the Regularized Empirical Risk Minimization approach and a Squared Hinge Loss for `L(m)`. This is no copy-paste, think! Do you remember the difference between the Loss on a *point*, the Loss on a *boundary*, and the *Risk*? You need these concepts super clear for the exam!

$$\frac{1}{2}\|w\|^2 + C \cdot \sum_{i=1}^{n} (\max[0, 1 - y_i \cdot f(x_i)])^2$$

3 2. Soft Margin SVM

- Following what we learned in the last exercise: **all** of the questions below need to be solved using Pandas and **no Python loops** (`for`, `while`, etc).
- Let's classify the Iris dataset with a hand-made SVM. Working with margins though means we need to set up once again for **binary classification**. Last time we used the categorical feature type, but this time we need the numerical values for the two classes to be of integer type in $\{-1, 1\}$, because we work with margin calculations. How would you do it? Here's one way to get it done:

[2]:
```
df = sns.load_dataset('iris')
# sns.pairplot(df, hue='species')

df.loc[df['species'] == 'setosa', 'species'] = -1
df.loc[df['species'] != -1, 'species'] = 1
df['species'] = pd.to_numeric(df['species'])
print(df.dtypes)
# sns.pairplot(df, hue='species')
```

```
sepal_length    float64
sepal_width     float64
petal_length    float64
petal_width     float64
```

```
species           int64
dtype: object
```

Also, do not forget to **split your data** into train set and test set, because using the wrong data you will fail the question. This was introduced in the last lab and will hold until the end of the course. You can copy the code from last submission.

[3]:
```
from sklearn.model_selection import train_test_split
train, test = train_test_split(df, test_size=0.2) # 80-20 split
```

2.1 [1pt] Write a Python function that takes in input a row from the Iris dataset (concatenation of x_i and y_i), and a parametrization (w, b), and returns the margin. This is slightly different from what you did so far: rather than pre-splitting the inputs from the labels, you should do the splitting inside the function. You can test the method by passing a (full) single row of the dataset (e.g. `iloc[0]`: notice you do not drop the label). As you should expect, the correct output is one number.

[4]:
```
def margin(row, params):
    x = row[:-1]
    y = row[-1]
    w, b = params
    return y * (w.dot(x) + b)

w = np.random.rand(4)
b = np.random.rand()
params = (w, b)
print("params:", params, "\n")

print(margin(df.iloc[0], params))
```

```
params: (array([0.73111593, 0.17768423, 0.10074684, 0.76728067]),
0.7417954168959499)

-5.3868831663634245
```

2.2 [1pt] Write a Python function that takes in input an entire dataset and computes the margins for all points. Use Pandas' `apply()` (remember you cannot use Python loops). To unit-test these functions, particularly regarding getting the sizes right, get used to generating random parameter sets of the correct size (using `np.random`). The correct output is a Pandas Series that has the same length as your training set (that is 120 numbers if you went for a 80-20 split).

[5]:
```
def margins(dset, params):
    return dset.apply(lambda row: margin(row, params), axis=1)

margins(train, params)
```

[5]:
```
53     6.572059
66     6.973379
81     6.329419
48    -5.578718
36    -5.669255
        …
102    8.671467
97     7.220675
108    8.049920
9     -5.102933
107    8.610036
Length: 120, dtype: float64
```

2.3 [1pt] Write a Python function that computes the Squared Hinge Loss for the whole dataset. Re-use the functions defined in answering the previous questions.

- Feel free to define it in two steps, the first being the Loss function for a single number, the second another round of `apply()`.
- You will likely see a lot of zeros in the result, together with some high numbers. *Think: why so?* Look at the Loss function.
- You can either `apply()` a simple lambda to the output of `margins()`, or a more complex lambda to the dataset directly.

[6]:
```
sq_hloss = lambda m: np.max([0, 1 - m])**2

def sq_hlosses(dset, params):
    return margins(dset, params).apply(sq_hloss)

sq_hlosses(train, params)
```

[6]:
```
53      0.000000
66      0.000000
81      0.000000
48     43.279529
36     44.478960
         …
102     0.000000
97      0.000000
108     0.000000
9      37.245790
107     0.000000
Length: 120, dtype: float64
```

2.4 [2pt] Write a Python function computing the soft-margin objective function for parametrization (w, b) and hyperparameter `c`. Worth repeating: no loops, use Pandas. You should be looking for the "unconstrained convex optimization problem" form. Consider using the `linalg` module of numpy, which already gives you an implementation for `norm` (the "standard"

norm is already the Euclidean, or *norm-2*). More on norms here.
For testing purposes, a simple `c=1.0` will work.

[7]:
```
def opt_fn(dset, params, c):
    w, _ = params
    regularization = np.linalg.norm(w)**2 / 2
    violations = c * sq_hlosses(dset, params).sum()
    return regularization + violations

opt_fn(train, params, 1.0)
```

[7]:
```
1891.7160590075996
```

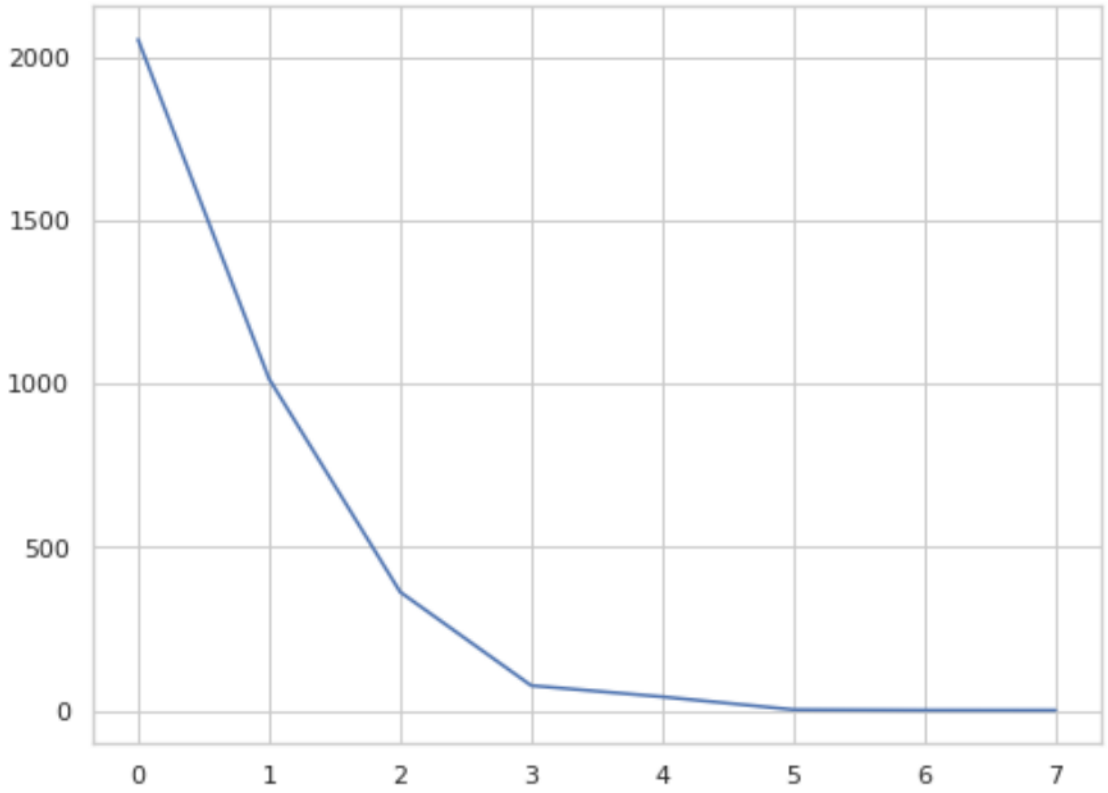
2.5 [4pt] Train the SVM using Parameter Guessing and print the number of misclassified points.

- We do not want to implement quadratic programming because it is nasty, but we understand that its only role is to find the parameters minimizing a function.
- Copy the *parameter guessing* method from the first lab, then adapt it to estimate the parameters that *minimize the SVM optimization function* defined above.
- You still need a loop for Parameter Guessing. The "forbidden loops" are the loops that go over the data points, as their goal is to force you to learn how to do the same, more efficiently, using Pandas. It should be clear by now if a loop is allowed or not.
- The classification with SVM is exactly the same as the Perceptron: positive margin is correct classification, negative is incorrect classification. Remember to use the correct dataset at the right step! (you did split the available data, right?)
- You can make the next question easier by defining two Python functions instead of one: `train_svm(dset_1, c, ntries)` and `test_svm(params, dset_2)` (again, what are those `dset`s?).
- You should not need many guesses to consistently get a decent margin here, try `ntries=100` and `c=1` as defaults, but feel free to play with them after you succeed once.
- For your code to run, you need to debug. This is not a waste of time: if you do it well, it will save time by not having to hunt for difficult errors. The best way to debug is to visualize the progress of the training. Why don't you use the plot of *errors over iterations* (for the best-so-far) that we first saw with the parameter guessing? If it goes down over time then it's working!

[8]:
```
def train_svm(dset, c=1, ntries=1000):
    min_score = np.Infinity
    best_scores = []
    best_params = None
    input_size = dset.shape[1] - 1 # -1 to remove 'species'
    guess_range = [-1,1] # guess parameters in this range
    for ntry in range(ntries):
        guess = (np.random.uniform(*guess_range, input_size), # w
                 np.random.uniform(*guess_range))             # b
        score = opt_fn(dset, guess, c)
        if score < min_score:
```

```
            min_score = score
            best_params = guess
            best_scores.append(score)
    print(f"Best: `{best_params}`\nScore: `{best_scores[-1]}`")
    sns.lineplot(x=range(len(best_scores)), y=best_scores)
    return best_params

def test_svm(params, dset):
    misclassified = margins(dset, params) < 0
    print(f"Number of misclassified points: {misclassified.sum()}")

svm_params = train_svm(train)
test_svm(svm_params, test)
```

```
Best: `(array([-0.27610793, -0.38218934,  0.84539862,  0.10704078]),
0.17325060452792562)`
Score: `0.9132140465951462`
Number of misclassified points: 0
```



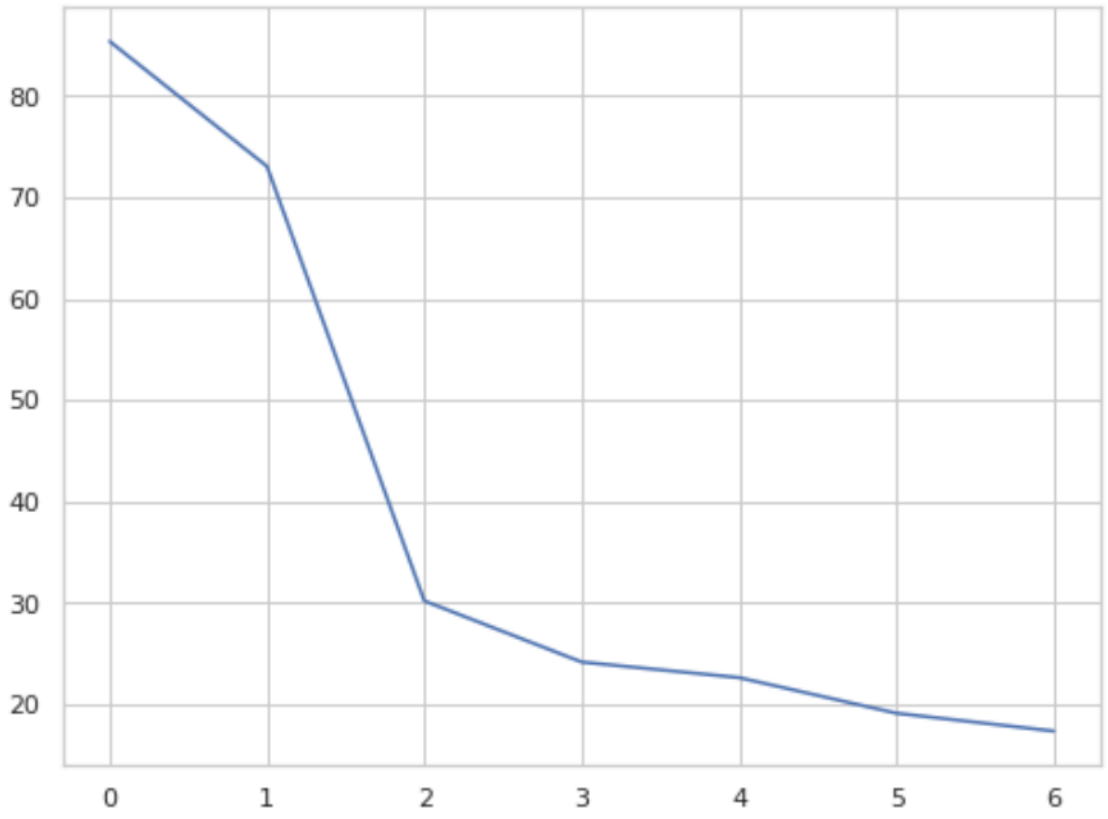
2.6 [3pt] Train another SVM, this time using only two features of the Iris dataset: `petal_length` and `petal_width`. Plot the decision boundary of the trained parameters,

on top of a scatterplot of the (2D) test set.

- You cannot plot higher-dimensional separation hyperplanes in 2D unless (a) they are perpendicular to the features you want to plot, or (b) you project the points parallel to the hyperplane. So if you want to plot you need to re-train for 2D data. If this is not clear, try to draw an example by hand using a 2D graph but 3D data.
- Careful because starting next week I will resume calling it just "the data", and using the wrong dataset when evaluating the model (i.e. training set) will cause you to **fail** the question. Always train on the *training* set, you can use cross-validation if you need statistically robust results on little data, you need the *validation* set for tuning the hyperparameter, and you finally show the results on the *test* set.
- Start by generating the `train_2d` and `test_2d` datasets, we have already seen how in the past. Then try running your SVM train code.
- Careful: if your implementation hardcoded the number of features, just go back and refactor your code to take the size from the data.
- You are learning parameters `w` and `b`: remember that for plotting you should convert them to `m` and `q`, just another copy+paste.
- Understand that you are not "training" the SVM as much as guessing one that is right. If you want to improve the margin to look nicer, try increasing the number of guesses.
- Feel free to play with `C`. A value of `1.0` here still works, but many applications may need it as high as `1e5` or as low as `1e-5`. Keep it in mind for the next assignment (kernels)!

[9]:
```
train_2d = train.loc[:,['petal_length', 'petal_width', 'species']]
test_2d = test.loc[:,['petal_length', 'petal_width', 'species']]
svm_params_2d = train_svm(train_2d, c=1, ntries=1000)
print(svm_params_2d)
test_svm(svm_params_2d, test_2d)
```

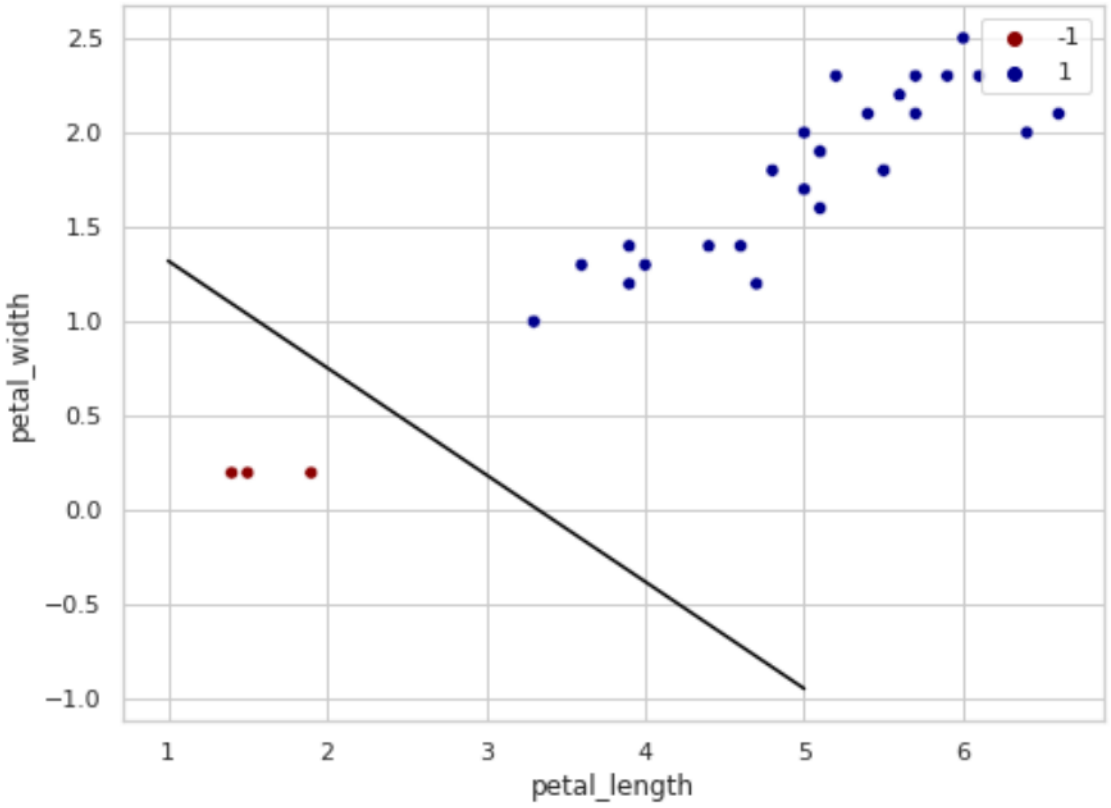
```
Best: `(array([0.29348757, 0.51853333]), -0.978766728493558)`
Score: `17.38515272752087`
(array([0.29348757, 0.51853333]), -0.978766728493558)
Number of misclassified points: 0
```



[10]:
```python
def wb2mq(w, b):
    assert len(w) == 2, "This implementation only works in 2D"
    assert w[0] != 0 and w[1] != 0 and b != 0 # simplify
    return [w[0]/-w[1], -b/w[1]] # m and q

def params2boundary(w, b):
    m, q = wb2mq(w, b)
    print(f"m: {m}, q: {q}")
    print(f"Boundary: `y = '{round(m, 2)}'x + '{round(q, 2)}'"")
    return lambda x: m*x + q

def plot_data():
    sns.scatterplot(data=test_2d,
                    x='petal_length',
                    y='petal_width',
                    palette=sns.color_palette(['darkred', 'darkblue']),
                    hue='species')

def plot_model(model):
    x_space = np.linspace(1,5)
    sns.lineplot(x=x_space, y=model(x_space), color='black')
```

```
boundary = params2boundary(*svm_params_2d) # remember the splat?
plot_data()
plot_model(boundary)
```

```
m: -0.5659955697509628, q: 1.8875676369624523
Boundary: `y = '-0.57'x + '1.89'
```



2.7 [2pt] Classify the 2D dataset using the scikit-learn implementation. Mandatory: (i) use the `LinearSVC` implementation (as we did not study kernels yet); (ii) use the same Loss function as in your hand-made implementation (should be easy enough to find), (iii) use a custom value of `C` that tends to ignore margin violations more than 1.0, but without misclassifying data.

- SVC stands for "Support Vector (Machine) Classifier", while SVR uses the same trick for Regression (**do not** use it here).
- Class `LinearSVC` underneath uses the `liblinear` library, which is highly optimized for linear SVMs. Class `SVC` on the other hand uses `libsvm`, which is more optimized for nonlinear SVMs using kernels. In some cases you may actually get better results on the default parameters with `SVC(kernel='linear')` than you do with `LinearSVC()`, but the latter has a more flexible parametrization (for linear boundaries only), so eventually you should have more room for improvement (you can use a validation set to search for better hyperparaters).

- Remember that this question requires you to access the right datasets for training and test phases, and that you need to split the `x` and `y` as seen last week with NB (that's Naïve Bayes and yes next time you see NB that is what you should expect to mean).
- You may be tempted to try passing a straight `0` to `C`. Many implementations give errors for `C=0`, but you can get around it by passing a good enough approximation :) remember you can use the scientific notation `1e-x` for decimals, but you really don't need it too small: with a low enough `C` you will start getting misclassified points! Give it a try: do you understand why this happens? (spoiler/hint: think about the English meaning of "maximizing the *margin* without restrictions", how big a margin can you make if you wish to?)

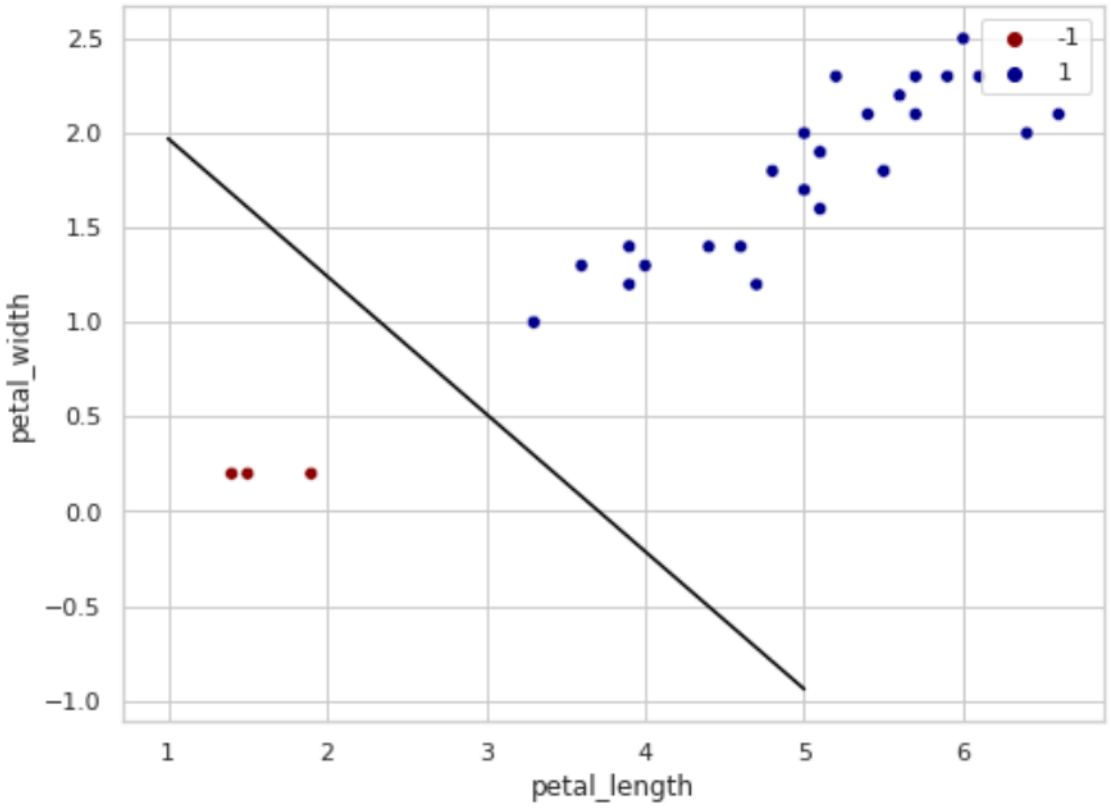
[11]:
```
from sklearn.svm import LinearSVC
x_train = train_2d.iloc[:,:-1]
y_train = train_2d['species']
x_test = test_2d.iloc[:,:-1]
y_test = test_2d['species']
model = LinearSVC(loss='squared_hinge', C=0.1)
trained = model.fit(x_train, y_train)
y_pred = trained.predict(x_test)
n_misclassified = (y_test != y_pred).sum()
print(n_misclassified)
```

```
0
```

2.8 [1pt] Plot the decision boundary of the SVM over the dataset. Think carefully and don't mess this up: *which dataset does the question refer to?* Remember, from now on, we always split our initial data into (at least) two datasets, and all questions referring to learning the parameters use one of the two datasets, while all questions referring to showing the learned model performance should use the other.

[12]:
```
skl_params = [trained.coef_[0], trained.intercept_[0]]
skl_boundary = params2boundary(*skl_params) # remember the splat?
plot_data()
plot_model(skl_boundary)
```

```
m: -0.727609823939731, q: 2.698682814602178
Boundary: `y = '-0.73'x + '2.7'
```



4 At the end of the exercise

Bonus question with no points! Answering this will have no influence on your scoring, not at the assignment and not towards the exam score – really feel free to ignore it with no consequence. But solving it will reward you with skills that will make the next lectures easier, give you real applications, and will be 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 :)

BONUS [ZERO pt] Train an SVM on the full Iris dataset (3 classes) using `libsvm`, and print the number of misclassified points. Play with `C` and see how this changes.

BONUS [ZERO pt] Train you hand-written SVM implementation using the PEGASOS algorithm. Now you have a state-of-the-art implementation, congrats! It will actually run better than the scikit-learn `LinearSVM` and `SVC` implementations. Remember this when you need a quick ML result in the future, especially over large data, and performance matters! To install pegasos (or any other GitHub repository) just switch to a local environment (as shown in the last lecture), then run (from your assignments folder):

```
pipenv install -e 'git+https://github.com/ejlb/pegasos.git#egg=pegasos'
```

Here is probably the most complete guide to Pipenv. I do not expect a full read (I didn't myself); but if you need something from Pipenv, a quick search in that page will likely point you in the right direction.

If you want to run on `colab`, this could be the right time for you to learn about using "bang commands" in Jupyter notebooks ("bang" being an exclamation mark `!`): they will be run in the shell that is running the Jupyter server. For example you can use `!pip install`. But as mentioned, this is just a hack as you have limited and blind control over your environment, and particularly it will give you problems as soon as you start collaborating with other people on other platforms (the solution is of course a local installation with Pipenv).

4.0.1 Final considerations

- SVMs are optimal – when applicable. In particular, if your data is bounded on all features, and you have enough samples from the whole range (so that new points in the test set are expected to be within the same range), then its performance are awesome. And if the data's underlying function is not overly complicated, you don't need an overly complicated model. In the real world and with real data, you are almost guaranteed to get to using a SVM for an application or another at some point in time.
- Remember that easy problems can be often solved with parameter guessing; this also means that if you can solve a problem with parameter guessing, it is an easy problem :) much depends on the model though, with the right model guessing the parameters is efficient, with the wrong model both guessing and training performance drops.
- The main limitation left at this point is handling non-linearities in the data. This is exactly the topic for the next lecture. Be ready as it is more math intensive than linear SVM, but you should have all the pieces now to grasp it completely in the next week.

assignment_07_solution

April 2, 2023

Please fill in your name and that of your teammate.

You:

Teammate:

[1]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import pandas as pd
sns.set(rc={'figure.figsize':(8,6)}, style="whitegrid")
```

1 Introduction

Welcome to the seventh lab. After learning about SVMs last week, we finally introduce the *kernel trick* and make them capable of tackling nonlinear data. We also introduced more generally the concept of *function mapping* and learned a bit about word embeddings.

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 16 points (out of 24 available) to pass (66%).

2 1. Function Mappings

1.1 [3pt] Give an *original* example for each of the following concepts (i.e. not one that you find in the slides!):

1. Mapping from an example data type to a decision space
2. Inverse mapping
3. Mapping from the example data to two destination feature spaces
4. Mapping from the two feature spaces above to a decision space

For example (here is what we presented in the slides):

1. Given a picture, decide if it represents an apple or an orange. Original space: $64 \times 64 \times 3$ images; destination space: {apple, orange}.
2. Given the label apple, map it to a $64 \times 64 \times 3$ picture of the fruit.
3. Map Φ_1 goes from the picture to an estimate of average color; map Φ_2 goes from the picture to a fruit width measured in pixels.
4. Map Φ_3 goes from the two features of estimated color and fruit width, to the decision space of classifying the fruit as apple or orange.

The example above is by definition already a correct solution.

1.2 [1pt] Explain advantages and disadvantages of Bag Of Words versus Word Embedding. Bag of Words produce a higher-dimensional feature map (size of the dictionary) that is very sparse (each text only uses a subset of the dictionary). Word Embeddings produce a smaller feature space, which is dense.
More importantly, BoW only uses tallying of the used words (counts the number of occurrences), while WE maintains the distance between words in the text, encoding words that tend to appear often together with closer vectors in feature space. This allows deduction and reasoning of word relationships based on vector similarity (distance and angle).

1.3 [2pt] Refer to the graph exemplifying Word Embedding in the slides, and its explanation. (i) What does it mean that the point representing Paris is close to the point representing Berlin? (ii) Why is the point for Paris closer to the point for France than to the point representing Italy?

(i) It means that Paris and Berlin refer to two similar concepts, that often appear together in the text corpus. (ii) It means that Paris tends to appear more often closer to France than to Italy in the text corpus.

3 2. Kernels theory

2.1 [1pt] Write the definition of kernel function (use latex).

$$k(x, y) = \langle \phi(x), \phi(y) \rangle$$

2.2 [2pt] Explain the kernel trick in English. The kernel trick states that the kernel function between two points is equivalent to computing the dot product between the mapping of the two data points into RKHS. This is useful because it allows us to use all the positive properties of RKHS while avoiding working with the mapping function directly, which can potentially be complex. The actual formulation of the kernel can be any form that produces a Gram matrix that is symmetric and positive semi-definite.

2.3 [1pt] Explain in English the required properties of a Mercer kernel.

- Symmetry: the order of the parameters to the kernel does not matter.
- Positive definiteness: the Gram matrix of the kernel is positive semi-definite for any finite subset of the input space.

2.4 [1pt] Calculate by hand the linear kernel on points $\{[2,4],[1,-2]\}$. The linear kernel is simply the dot product:

$$k([2,4],[1,-2]) = [2,4]^{\mathrm{T}} \cdot [1,-2] = 2 - 8 = -6$$

2.5 [1pt] How do you compute the entry of the Gram matrix for row i and column j for a Gaussian kernel?

$$K_{ij} = k(x_i, x_j) = \exp(-\gamma \cdot \|x_1 - x_2\|^2)$$

2.6 [2pt] Explain why does the Perceptron work with non-linearly separable data using Kernelization. Do you think Linear Regression would work with Kernelization? Explain your reasoning. Kernelization maps non-linearly separable data to a space where the data is hopefully linearly separable. At that point, the Perceptron is able to do the linear separation. The same works with any linear algorithm, including Linear Regression: once the Kernelization takes care of nonlinearities, the linear algorithm can do its job without problems.

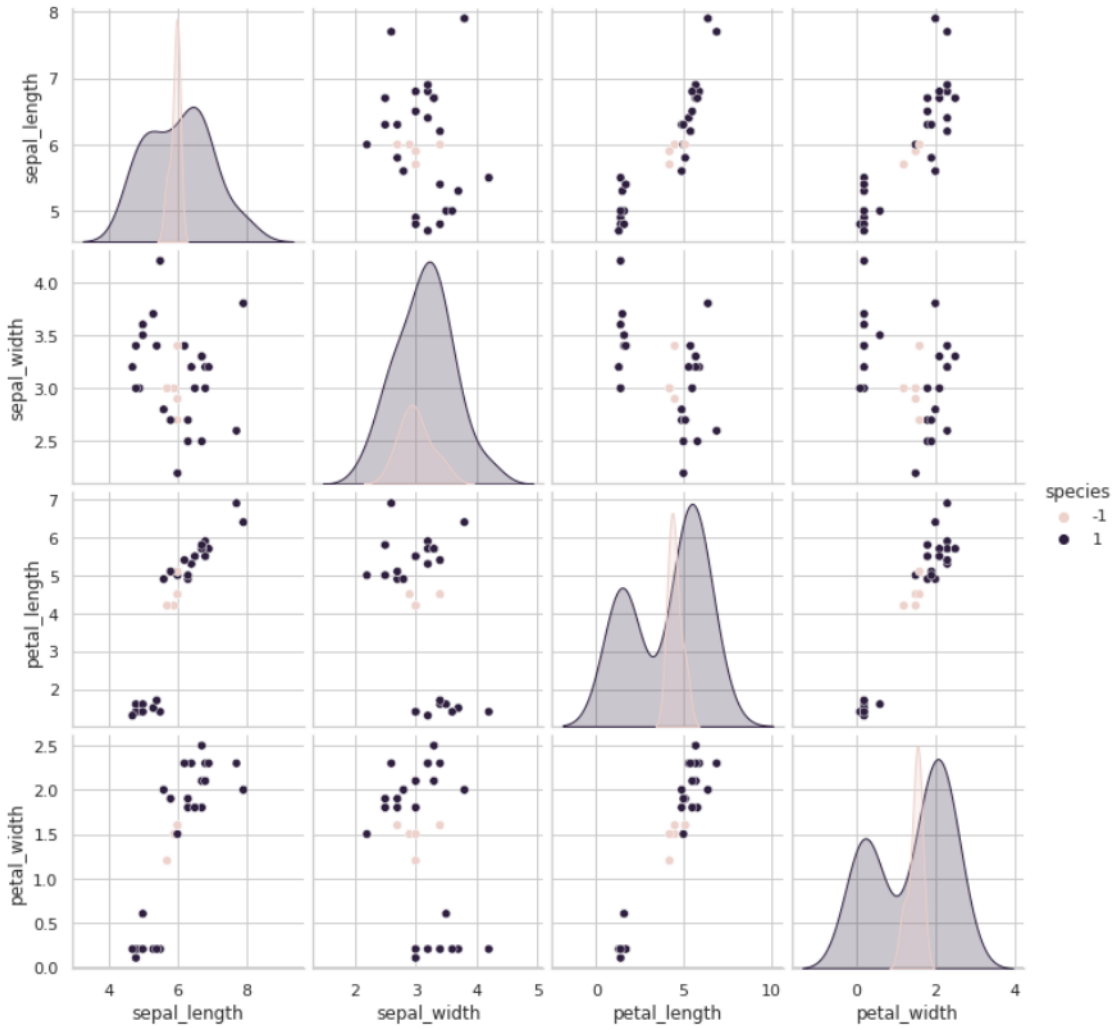
4 3. Kernels in practice

For simplicity, let's use once again a two-species adaptation of the Iris dataset. You can copy the code from the last assignment. This time though, to make it harder for linear classifiers let's separate the "central" species from the other two. This means that you should set label `versicolor` rather than `setosa` as class `-1`. I suggest you un-comment the `pairplots` to verify it works.
NOTE: all recommendation on how to handle and prepare the data from the past assignment(s) still hold. As do the warnings that using the wrong data sets will **invalidate the whole answer**.

[2]:
```
df = sns.load_dataset('iris')
# sns.pairplot(df, hue='species')

df.loc[df['species'] == 'versicolor', 'species'] = -1
df.loc[df['species'] != -1, 'species'] = 1
df['species'] = pd.to_numeric(df['species'])
print(df.dtypes)
# sns.pairplot(df, hue='species')

from sklearn.model_selection import train_test_split
train, test = train_test_split(df, test_size=0.2) # 80-20 split
sns.pairplot(test, hue='species')
```

```
sepal_length    float64
sepal_width     float64
petal_length    float64
petal_width     float64
species           int64
dtype: object
```

[2]: `<seaborn.axisgrid.PairGrid at 0x7f68b05b2130>`



3.1 [2pt] Train an SVM with linear kernel on the Iris data using Scikit-learn (this time you are required to use class `SVC`). Then do the same using a Gaussian kernel (still `SVC`) and compare the performance using its method `score()`.

- Remember to prepare inputs/labels for Scikit-learn; again the last assignment should help.
- Calling the method `score()` on the trained model just does the prediction and returns the percentage of correct answers. It is a useful function to learn to quickly check if your model is working.
- You expect the linear kernel to perform poorly. If the performance is close to the Gaussian kernel, it is possible that the test set was by chance not homogeneous. You can verify that by doing a pairplot on the test set, and if so just run the data loading and preparation again.
- No need to find an optimal value for `C` but pass it explicitly.

[3]:
```
from sklearn.svm import SVC

x_train = train.iloc[:,:-1]
```

```
y_train = train['species']
x_test = test.iloc[:,:-1]
y_test = test['species']

lin = SVC(kernel='linear', C=1).fit(x_train, y_train)
print(lin.score(x_test, y_test))

gau = SVC(kernel='rbf', C=1).fit(x_train, y_train)
print(gau.score(x_test, y_test))
```

```
0.7
0.9333333333333333
```

3.2 [2pt] Write a Python function that takes two data points and a value for `gamma` as input, and returns the Gaussian kernel of the points.

[4]:
```
kern = lambda x, y, gamma: np.exp(- gamma * np.linalg.norm(x - y)**2)
```

3.3 [3pt] Write a Python function that takes two dataset (and a gamma) and returns their Gram matrix for a Gaussian kernel.

- You need two datasets because you need to compute the *train* matrix between the train and itself, but the *test* matrix between the test and the train.
- Simplest method:
  - Create a return matrix, initially empty, shaped size_of_A times size_of_B, with dtype 'float64'
  - Run two loops with indices (i, j) in ranges up to size_of_A and size_of_B
  - Compute the kernel between row i in A and row j in B, and place it in the return matrix at row i column j
- Careful with Pandas' `iterrows()`, as the "index" it returns is the DataFrame index (i.e. for use with `loc[]`), not the ordinal index (i.e. for `iloc[]`).
- Generating the matrix automatically is harder, as there is no straightforward way to compute an `outer` in numpy or pandas with a custom function.
- One way is to use `column_stack` https://stackoverflow.com/a/21759340 then apply the kernel defined above.
- Another is to use `ufunc.outer` http://folk.uio.no/inf3330/scripting/doc/python/NumPy/Numeric/numpy-7.html which is only defined for Universal Functions (`ufuncs`). Look at the examples for `outer`, you can re-implement the function above starting with `np.subtract.outer(A, B)`, which generates the matrix (but check the shape!), then you can run the other operations using broadcast. Both outers and universal functions are super useful, it's worth the effort of learning them, more [here].
- `pandas.apply()` along rows is also an option you should be able to consider with by now. The function name for `-` is `np.subtract` (which is an `ufunc`, see above).

Above all, remember the first rule of a good BDD engineer: red, green, refactor! First make it work, then make it better ;) complex solutions are as good as bonus questions here.

Also, know that a common default value for gamma is one over the number of features.

[5]:
```
# This is not the most efficient version (by far). But should be readable by␣
 ↪everyone.
# Is your answer better?
def gram(dset_1, dset_2, gamma=None):
    npoints_1, nfeats_1 = dset_1.shape
    npoints_2, nfeats_2 = dset_2.shape
    assert nfeats_1 == nfeats_2 # always check for consistency
    if gamma is None: gamma = 1/nfeats_1
    gmat = np.empty([npoints_1, npoints_2], dtype='float64')
    for r in range(npoints_1):
        for c in range(npoints_2):
            # hint: which operations you need here and which can you broadcast?
            gmat[r][c] = kern(dset_1.iloc[r], dset_2.iloc[c], gamma)
    return gmat
```

3.4 [2pt] Compute the Gram matrix on the inputs of your datasets. Then train a new SVM, same settings as before with linear kernel, but this time using the Gram matrix('s rows) as the inputs. Print the `score` performance of this new SVM.

- With an 80-20 split you are looking at a 120×120 shape for the train, and 30×120 for the test

[6]:
```
x_train_gram = gram(x_train, x_train)
x_test_gram = gram(x_test, x_train)
print(x_train_gram.shape, x_test_gram.shape)
```

```
(120, 120) (30, 120)
```

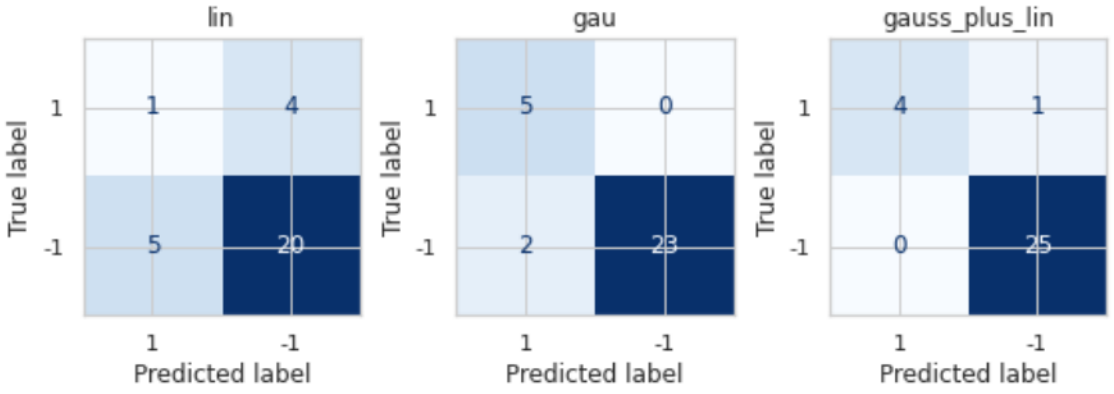
[7]:
```
gauss_plus_lin = SVC(kernel='linear').fit(x_train_gram, y_train)
print(gauss_plus_lin.score(x_test_gram, y_test))
```

```
0.9666666666666667
```

3.5 [1pt] Plot the confusion matrix for the three SVMs you trained in the past questions.

- Let's learn a convenient and easy function for this common, very useful metric: `ConfusionMatrixDisplay.from_estimator` [link here].
- So far we saw explicitly 4 cells: true and false positives, true and false negatives. More generally the confusion matrix can be scaled to any number of classes by having the correct labels on the rows, and the predictions on the columns. Errors will be outside the diagonal.
- You can use the `normalize` option to get percentages if you like. Which setting do you find most informative?
- It's easier if you write a `for` loop over the three models you trained in the previous questions – just make sure you gave them different names. Also careful as one takes a Gram matrix as input ;)

[8]:
```
# from sklearn.metrics import plot_confusion_matrix
from sklearn.metrics import ConfusionMatrixDisplay
class_names = y_test.unique()
# It's about time we learn to use subplots, especially for small things
fig, axes = plt.subplots(1,3)
fig.tight_layout()

# Here's another trick: you can list the _names_ of the variables,
# then use their name on the title, and fetch their values using `eval()`.
# This is not optimal here, but I wanted to expose you to this technique,
# as it opens the door to metaprogramming (you could _build_ the name-strings!)
for model_name, ax in zip(["lin", "gau", "gauss_plus_lin"], axes):
    if model_name == "gauss_plus_lin":
        x_dset = x_test_gram
    else:
        x_dset = x_test
    ConfusionMatrixDisplay.from_estimator(
        eval(model_name), x_dset, y_test,
        ax=ax,
        display_labels=class_names,
        colorbar=False,
        cmap=plt.cm.Blues,
        normalize=None)

    ax.set_title(model_name)
```



5 At the end of the exercise

Bonus question with no points! Answering this will have no influence on your scoring, not at the assignment and not towards the exam score – really feel free to ignore it with no consequence. But solving it will reward you with skills that will make the next lectures easier, give you real applications, and will be 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 :)

BONUS [ZERO pt] Use a contour plot to show the classification boundaries of your SVMs. [link here]

BONUS [ZERO pt] Learn to search for the best values for `gamma` and `C` [link here] . NOTE: this is extremely valuable experience for when (not even if!) you will need a SVM for a real application.

BONUS [ZERO pt] Generate points to run through with a regression algorithm like Linear Regression from our earlier exercises. This time start from a nonlinear equation (e.g. x^2), and add noise as usual. Then try your hand with SVR with a (linear or) nonlinear kernel, which is equivalent to running Linear Regression on the Gram matrix (yet another name: Kernel Ridge Regression) [link here].

5.0.1 Final considerations

- I once read a quote that restricting calculus to linear functions is like restricting biology to the study of great apes (help tracking its origin would be welcome). We start from linearity because it's easier to study; the real world is rarely so kind, so learning adaptations such as the kernel trick is simply invaluable.
- Trying (scikit-learn) Naïve Bayes or Linear Discriminant Analysis on the Gram matrices would take you just a minute and be invaluable experience. For example, I wouldn't be surprised if LDA performed better than NB (think: why?). But if we had a very large dataset, the Gram matrix would become too large for LDA to handle (remember it does not scale well on the number of features).
- **[IMPORTANT]** If you want to gain first-hand experience in tools you can actually use in the real world, consider submitting on the bonus questions from this point on, as I am switching the topic from "topics for curiosity" to "actual deployed value".

assignment_08_solution

April 15, 2023

Please fill in your name and that of your teammate.

You:

Teammate:

[1]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
sns.set(rc={'figure.figsize':(8,6)}, style="whitegrid")
```

1 Introduction

Welcome to the eighth lab. Today we have *way too many* topics to cover to do everything by hand as usual, so I selected different depths to each topic to make sure you gain full insight and applicable experience.

As you go through the exercise and you apply algorithm after algorithm, method after method, I want you to think about the actual **competence** you are accumulating over the months, both theoretical and applied. Think about it, and be confident: covering so many, so different algorithms in a single lab may sound like a challenge to the version of Past You from barely two months ago, but I believe Today's You is capable of taking on this whale of a lab and have space for more.

Working with many algorithms gives me another chance to shake you out of your confidence zone with respect to *data processing* . Basically each algorithm requires different formats, so you cannot just define the data on top and keep reusing it: you will need to re-load the dataset for each exercise, applying a different processing each time. Be flexible, and don't forget your train-test splits (and their correct usage) – I should not need to mention it anymore, right? :)

Good luck, have fun!

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 16 points (out of 24 available) to pass (66%).

2 1. Fundamentals

1.1 [1pt] Write an example (in English) of a Machine Learning application for which the Supervised Learning paradigm is not (directly) applicable. Autonomous Driving: for SL you need the correct output for every input, but with autonomous driving (i) there could be multiple equally correct reactions to a certain input (e.g. dodge a pedestrian vs. break), and (ii) building a dataset of correct reactions is unfeasible for a single driver, while aggregating information from multiple drivers means inconsistencies in the driver skills and decision process.

1.2 [1pt] Write an example (in English) of a Machine Learning application for which the Unsupervised Learning paradigm is an ideal choice. Image Segmentation: by finding groups of pixels in an image with (i) similar colors and (i) within a minimal distance from each other, it is possible to extract shapes and build masks. These approaches still today can offer better performance than Deep Learning in cases where the training data is scarce and the colors/shapes are obvious.

Careful: if you used the same example for both questions above, something may not be right. These questions are meant to make you think about the *difference* between SL and UL: there is something that is necessary for SL and ignored in UL, so an ideal SL application has that something, while an ideal UL application does not (or SL would be better suited!).

3 2. Clustering

2.1 [2pt] Explain the k-means algorithm using a few words of your own. Particularly, state any requirements, and what the user needs to define.

- To use "your own words", a trick is to read the slide, decide which things you need to mention (feel free to list a few keywords), then close the slides. Now imagine speaking to a friend who has only basic technical background (aka "rubberducking", Google it!). Explain to this friend the things that you previously decided to mention.
- No need to go crazy. This is a type of answer that you will need to repeat over and over, refining it over time: it cannot be perfect the first time. For example, to convince your boss to let you use a particular method at work, or to supervise someone with less knowledge in the field. It's not a right/wrong question: you need to show that you are competent, select important key points, be brief and precise.
- As usual, copy+paste from the slides scores 0 points, while a sincere, fair try (that is not horribly wrong) will give you a pass. So don't worry too much :)

The algorithm k-means is an Unsupervised Learning clustering method. It means that tries to group close/similar points together, and for each group it chooses a representative element that is kind of their average. To run the algorithm you only need inputs, no labels; you also need to pass a target number of clusters k and a similarity measure (e.g. the Euclidean distance works, if the features are continuous). The algorithm works by spreading those means as much as possible,

while maintaining them centered to clusters because it penalizes the cumulative distance between a cluster's mean and the cluster's points.

For the next question, we need to understand how to evaluate a clustering algorithm. The main difference between clustering and classification is that, well, it's UL not SL: the labels are not involved in the training, and they should not be involved in the testing. So how do you test the performance of a clustering algorithm?

Each mean/cluster gets a numerical identifier, the only problem is that the number does not correspond to our labels because it's assigned randomly based on initialization. The most naïve way then is to brute-force all mappings between the labels and the cluster numbers: the one that makes the most sense is the one that should be used for evaluation. Since this is orthogonal to the lecture and may take a long time to debug, here is a snippet of code that does that.

Read it, understand it, play with it, and possibly improve it. Bruteforcing is rarely optimal, which is the very reason why ML exists :)
(note: it may be easier to understand it if you first go ahead with answering the next question first, then come back to this)

[2]:
```
# Goal: convert the labels to the cluster numbers generated by k-means
import itertools
species_names = sns.load_dataset('iris').species.unique()
possible_codes = itertools.permutations(range(len(species_names)))
converters = [dict(zip(species_names, perm)) for perm in possible_codes]
# try printing each of these variables and understand what they do

def cluster_to_class(model, fn_that_counts_misclassified, x_test, y_test):
    min_score = np.Infinity # we saw how to write a minimizer already right?
    right_conversion = None
    for converter in converters:
        conv_y_test = y_test.replace(converter) # conveniently works with␣
 ↪`dict`s
        misclassified = fn_that_counts_misclassified(model, x_test, conv_y_test)
        if misclassified < min_score:
            min_score = misclassified
            right_conversion = converter
    return right_conversion

## to use this function, you will need something like this
# right_conversion = cluster_to_class(k_means_model, my_misclass_fn, x_test,␣
 ↪y_test)
# conv_y_test = y_test.replace(right_conversion)
# k_means_misclassified = my_misclass_fn(k_means_model, x_test, conv_y_test)
```

2.2 [3pt] Apply the scikit-learn implementation of the k-means algorithm to the Iris dataset (4 features, but drop the labels for training), and print a performance score of your choice.

- **IMPORTANT:** the "number of misclassified points" is not an UL performance score, be-

cause UL does not see "classes" and thus does not do "classification". The function `score()` will ignore the labels and print "strange numbers". What are those? No worries though, you know how to make your own scoring from the past labs, right?

- This also mean that if you have points originally belonging to another class, which are however mixed in the cluster of another class (e.g. because of noise), it is correct of the algorithm to put them in the same cluster, even though you will get "misclustered" below. Always interpret your performance metrics! Try printing the points and the cluster centroids for example.
- Of course you want to pass $k = 3$.
- Passing the trained `KMeans` object to `print()` shows several useful parameters and their used values (defaults unless otherwise specified).
- After you get it to work though, why don't you try $k = 2$ or $k = 4$ and see what happens when you have to *guess* k (which would be the normal case in a real application).
- Notice how k-means is *very* sensitive to initialization. To get a consistently better result you may want to explore options `max_iter`, `n_jobs` and of course `init`.

[4]:
```
iris = sns.load_dataset('iris')
# Convert species to numerical values
orig_species = iris['species'] # We can use this later
# iris['species'] = iris['species'].astype('category').cat.codes
train, test = train_test_split(iris, test_size=0.2) # 80-20 split

x_train = train.iloc[:,:-1]
# y_train = train['species'] # UL does not need this
x_test = test.iloc[:,:-1]
y_test = test['species'] # This is used to see how well we do _without_␣
 ↪training labels


from sklearn.cluster import KMeans
k = 3

k_means = KMeans(n_clusters=k, max_iter=100000).fit(x_train)

## No classifier, no score
# print(k_means.score(x_test, y_test))
## Nope, also this only supports classifiers
# plot_confusion_matrix(k_means, x_test, y_test)

# But luckily we know how to handle this, right? Easy!
def correctly_clustered(model, points, clusters, verbose=False,␣
 ↪text='Misclustered:'):
    predictions = model.predict(points)
    misclustered = (clusters != predictions).sum()
    if verbose:
        npoints = len(predictions)
```

```
        print(f"{text} {misclustered}/{npoints}"
              f"({round(misclustered*100/npoints)}%)")
    return misclustered

right_conversion = cluster_to_class(k_means, correctly_clustered, x_test,␣
 ↪y_test)
y_test_clusters = y_test.replace(right_conversion)
k_means_misclassified = correctly_clustered(k_means, x_test, y_test_clusters,␣
 ↪verbose=True)
```

```
Misclustered: 5/30(17%)
```

2.3 [1pt] Plot the centroids learned with k-means on top of the data.

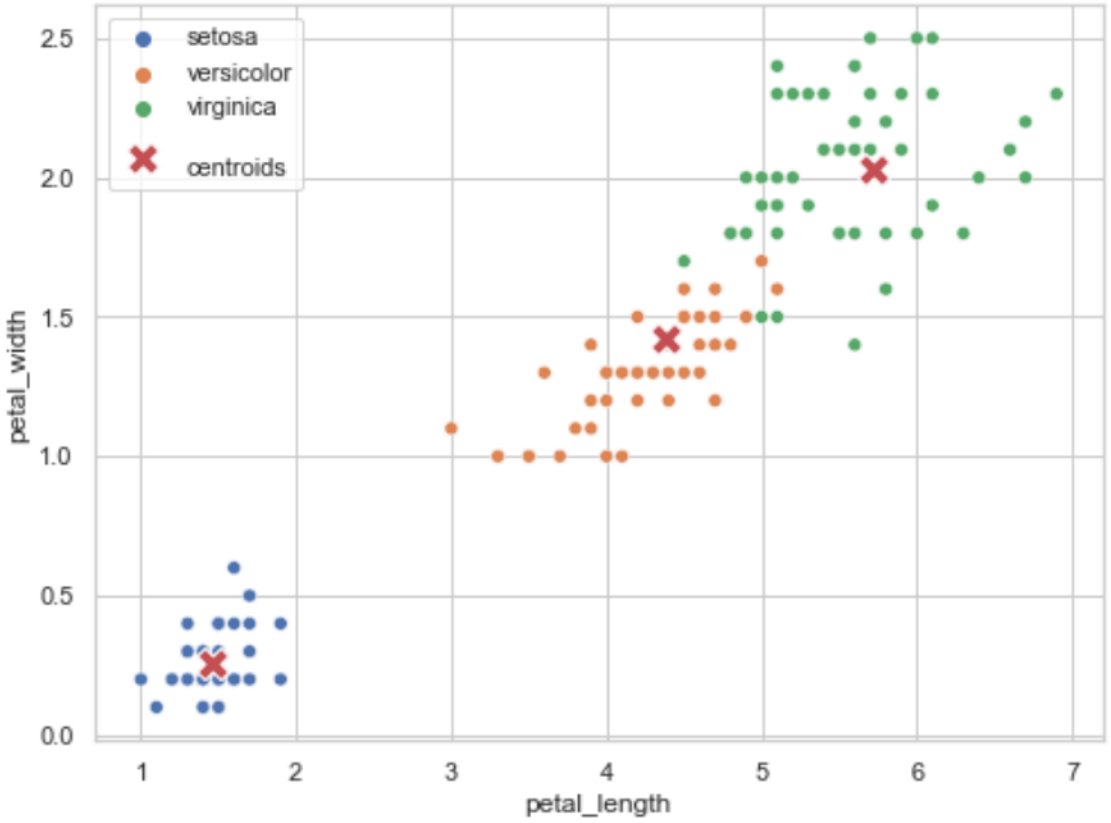
- To get the centroids coordinates, access attribute `cluster_centers_` of the KMeans object. Here are some options I passed to `scatterplot` for visibility (remember you can use the double-splat to transform the dict into keyword parameters):

```
kwargs = {'marker':'X', 'color':'r', 's':200, 'label':'centroids'}
```

- The question does not specify the details of what to plot, so it's up to you to provide a correct and useful interpretation. You learned how to make useful plots, just be confident.
- The simplest is of course to reproduce what we saw so far: one plot, `petal_width` vs. `petal_length`. As they are the last two features, make sure to pick the corresponding coordinates from the centroids. Remember your ranges and your `transpose()` ;)
- Even fancier: why not converting it to a DataFrame? Remember to drop the `species` column when constructing the `df`, as the centroids (learned with UL) have no species information (and thus one less column than iris): `columns=iris.columns.drop('species')`
- After answering correctly, if you want to learn something useful and fancy, try plotting a [PairGrid] that mimics a PairPlot but with added clusters off-diagonal. If you want the usual distributions on the diagonal, it is time to learn it is done with *Density Estimation* (which is what you learned to do for Gaussians in NB), automated in Seaborn with `kdeplot()`.

[4]:
```
sns.scatterplot(data=iris, x='petal_length', y='petal_width', hue='species')
centroids = k_means.cluster_centers_
# Remember the double splat `**`? It unwraps `dict`s into keyword parameters␣
 ↪(or vice-versa)
opts = {'marker':'X', 'color':'r', 's':200, 'label':'\ncentroids'} # what does␣
 ↪the '\n' do?

## Fancy
# cxs, cys = centroids[:,2:].transpose()
# sns.scatterplot(x=cxs, y=cys, **opts) # here's the d-splat

## Fancier? More readable code => less bugs, so always better explicit!
centroids = pd.DataFrame(centroids, columns=iris.columns.drop('species'))
sns.scatterplot(data=centroids, x='petal_length', y='petal_width', **opts) #␣
 ↪d-splat
```

[4]: `<AxesSubplot:xlabel='petal_length', ylabel='petal_width'>`



Note: this is a very basic application: while a decent knowledge of k-means can typically be useful in itself, the focus here is to cements your understanding of clustering, centroid, expectation maximization, and the difference between clustering and classification. For further reading, I strongly suggest you have a look at [this very complete tutorial].

2.4 [2pt] Train a scikit-learn OneClassSVM on the *versicolor* class of the Iris dataset, and print the number of missed outliers.

- Careful with the input: you need to train this SVM only on the subset of the train data where the species is `versicolor`. That's "one-class". The training should not have access to data from the other two species.
- Also remember to drop the species column (as always) after selecting the lines with `versicolor`.
- The test inputs should work as expected, but the test labels should be converted so that `versicolor` is `1` and the others are `-1` (because those are the model outputs for "normal" and "outlier").
- Again, to compute the missed outliers, you cannot use `score()` or `plot_confusion_matrix()` because technically it's not a classifier. But you already made a function of the scoring code for the previous question right?

[5]:
```
from sklearn.svm import OneClassSVM

iris = sns.load_dataset('iris')
train, test = train_test_split(iris, test_size=0.2) # 80-20 split
oc_x_train = train[train['species'] == 'versicolor'] # train _only_ on target␣
 ↪class
oc_x_train = oc_x_train.drop('species', axis=1) # remember to drop the label
# Everything that is not in our class gets predicted as "other" (`-1`)
oc_y_outliers = np.array([1 if label == 'versicolor' else -1 for label in␣
 ↪y_test])
# Of course you could have done the conversion between the split,
# but I find it more readable this way

oc_svm = OneClassSVM().fit(oc_x_train)

# Remember this is not classification! It is training only on one class,
# then detecting outliers. It cannot know that two of the species intersect.
# Thereby you should not expect to reach a perfect score, right?
oc_svm_missed = my_score(oc_svm, x_test, oc_y_outliers, verbose=True,␣
 ↪text='Missed')
```

```
Missed 5/30 (17%)
```

4 3. Compression and encoding

There are too many topics to cover for this lab. Having to choose one to cut off from practice, I had to objectively decide to remove my favorite one: compression and encoding. The reason is that in most jobs experience in the others will be more useful, while you will still see plenty of encoding techniques over the course. And the concept of dictionary building is related to k-means and feature extraction anyway.

On the other hand, understanding dictionaries as features, and encodings as mappings, allows for a much deeper competence and broader flexibility in the field than being stuck to only the "big guns" of Deep Learning for this task.

So my gift to you is one of my favorite papers so far: "The Importance of Encoding Versus Training with Sparse Coding and Vector Quantization", from A. Coates and A. Ng [link].

If you want to learn the state of the art, in any scientific field, you need to learn to read papers. A good suggestion not to be overwhelmed, especially at the beginning while building knowledge and glossary, is to read it this way

- First the abstract
- Then think about it and read the abstract again
- Now read the introduction, but don't fret about terms you don't understand, it's normal
- Next read the conclusion, and make sure their claims make sense with what you read so far
- The "discussion" explains how they interpreted their results and built the conclusion, which could be invaluable to understand their claims

- If you want more detail on the "how", check out the "method" section (here called "learning framework")
- The "related work" (sometimes "literature review") gives you pointers to extend your study on the field and applications
- Do not overlook "experimental results", as it will give you the means to reproduce their results. And yes, chances are your thesis (especially if Master) will begin by asking you to reproduce a paper's result. Repeatability and verification (the hypothesis needs to be falsifiable) are at the very core of the scientific process.

So: go at least through abstract, introduction and conclusions, and then answer the following question:

3.1 [4pt] In the conclusion of the paper "The Importance of Encoding Versus Training with Sparse Coding and Vector Quantization", which part do the authors find more effective, the encoding (i.e. decision-making) or the dictionary training (i.e. feature extraction)? Reflect on the consequences on modeling, and express your opinion on Feature Extraction and Decision Mappings. The solution will contain no "right" answer, just **my** answer. It actually constitutes one the foundations of my research. Full points will be awarded to anyone (i) asserting their opinion, and (ii) justifying it with findings from the paper.

The authors find the encoding much more significant than the dictionary training, to the point that sophisticated encodings can produce useful codes based on features generated from the data via random processes.

This implies that there is more strength to be found in function composition than in *careful* data preparation, and that the last decision mapping is incomparably more important than the initial, feature extraction mappings.

When attacking a problem with Machine Learning, (i) do enough cleaning to make algorithms work, (ii) do (possibly several layers of) feature extraction to increase the abstraction level, then (iii) very carefully prepare a final decision mapping for best performance.

You will see this pattern (always inherent, often unaware, sometimes ignored) in **all** the most successful ML applications to date.

5 4. Matrix decomposition

4.1 [1pt] For this data imputation exercise, use the entire Iris dataset (no split in train/test, but do drop `species`). Select the value in row index 100 column index 2, and store it in an outside variable. Then delete it from the dataset.

- To delete a value from a dataset, simply assign the "`not a number`" value (`np.nan`) to the corresponding element.
- Have you tried using `drop()` to remove a column? Remember that the default axis is the rows, so you need to pass `axis=1` or `axis='columns'` to drop a column by name.
- You can print a few rows around your target to verify everything is going as you expect.

[12]:
```
orig = sns.load_dataset('iris').drop('species', axis='columns')
nrow, ncol = (100, 2)
trg = orig.iloc[nrow, ncol] # splat seems not to work here?
```

```
print(f"Target: {trg}")
orig.iloc[nrow-2:nrow+2]
```

```
Target: 6.0
```

[12]:

	sepal_length	sepal_width	petal_length	petal_width
98	5.1	2.5	3.0	1.1
99	5.7	2.8	4.1	1.3
100	6.3	3.3	6.0	2.5
101	5.8	2.7	5.1	1.9

[13]:
```
orig.iloc[nrow, ncol] = np.nan
orig.iloc[nrow-2:nrow+2]
```

[13]:

	sepal_length	sepal_width	petal_length	petal_width
98	5.1	2.5	3.0	1.1
99	5.7	2.8	4.1	1.3
100	6.3	3.3	NaN	2.5
101	5.8	2.7	5.1	1.9

4.2 [4pt] Reconstruct (impute) the missing value using SVD and dimensionality reduction-based denoising. Do not use scikit-learn. Use the SVD method from `np.linalg`. Print the (absolute) difference between the original and reconstructed values.

- Ok, relax, this is not your first complex question. As usual, deconstruct the process in smaller, achievable goals, then work through them step by step.
- The first thing you need to do is to get rid of is the "hole" in the data, because SVD will not work with `nan` values. Simply patch it with an average of the values above and below in the same feature. We know this is not ideal, but no worries. BTW congratulations with this you just learned the foundation of the ***k*-nearest-neighbors algorithm (KNN)**.
- Now decompose the entire dataframe('s data matrix) using the SVD implementation in numpy's `linalg` module. Read carefully the documentation: it does not return `u`, Σ and `v` as expected from the lecture! Instead you get `v` as expected $n \times n$; `s` a vector of size n containing the σ eigenvalues (the diagonal of Σ, remember?); and `vh` $m \times m$ is the transposition of `v` – saves a transpose, but remember you will need to zero a *row* not a *column*.
- Next you want to drop the least contributing eigenvector. Find the smallest non-zero eigenvalue, and set to zero the corresponding column in `u` and row in `vh`. The relative size also tells you how much precision will you be losing with this reduction.
- Now you can already reconstruct the data. Remember the order of the dot products matters. Also, you need to build your Σ. There's an example in the documentation of SVD. Importantly: Σ is rectangular, the eigenvalues go in the diagonal of the first "square" of this matrix, the rest is zeros. You can set a range of rows and columns of a numpy rectangular matrix to (the values of) a diagonal matrix created with `np.diag()`, just match the sizes.
- Fetch the value in the target element's position in the reconstruction matrix. Has it changed w.r.t. its initial estimate? Print the difference with the original and technically you're done.
- If you are unsatisfied with the result though, you can run the cycle a few times. Place the code written so far in a function, so you can iterate multiple calls. Remember that you need

to insert the new value in the *original matrix* , and then loop all your calls to the denoising function on this matrix. Looping on the reconstruction is a common error which may cost you a lot! With every denoising you are losing a bit of information; copying only the value you are denoising reintegrates the information in the rest of the matrix, allowing for a much more accurate result.

- I converge (i.e. no more significant changes) to within 0.07 of the correct value in 50 iterations. I also simply save the errors (abs diff) at every iteration, then do the usual `lineplot`. Nothing new, but these sanity checks are priceless when working with ML.
- Alternatively: what happens if you drop two columns instead of one?

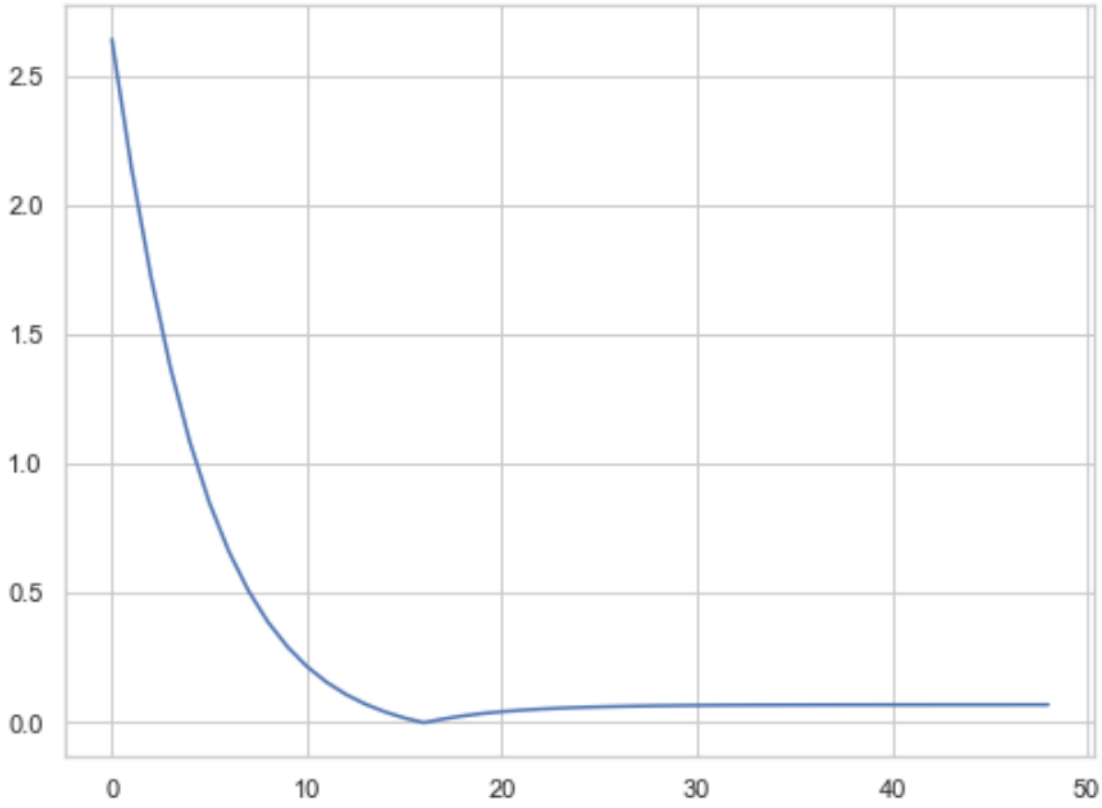
[18]:
```
first_estimate = orig.iloc[nrow-1, ncol] + orig.iloc[nrow+1, ncol]
orig.iloc[nrow, ncol] = first_estimate
print(f"Target: {trg}\nInitial estimate: {first_estimate}")

def denoise(mat):
    o_nrows, o_ncols = mat.shape
    u, s, vh = np.linalg.svd(mat, full_matrices=True)

    # Drop least contributing eigenvector
    u[:, -1] = 0 # here is the last column
    s[-1] = 0 # not necessary, but oh well consistency
    vh[-1, :] = 0 # here is a row

    # We need to reconstruct the data to the original dimensionality
    sigma = np.zeros((o_nrows, o_ncols), dtype=s.dtype)
    sigma[:o_ncols, :o_ncols] = np.diag(s)
    rec = pd.DataFrame(np.dot(u, np.dot(sigma, vh)))
    return rec

errors = []
for nstep in range(50-1): # initial estimate is the average above
    rec = denoise(orig).iloc[nrow, ncol]
    errors.append(abs(trg-rec))
    if (nstep+2)%10==0: print(f"Estimate {nstep+2}: {rec}")
    orig.iloc[nrow, ncol] = rec

print(f"\nImputation/reconstruction error: {round(abs(trg-rec), 2)}")
# You may see a *concave* in the error plot, and this is not wrong
# Remember that this is UL, thus BLIND! This is not a Loss!
sns.lineplot(x=range(len(errors)), y=errors)
orig.iloc[98:103]
```

```
Target: 6.0
Initial estimate: 9.2
Estimate 10: 6.38941334370923
Estimate 20: 5.973550066828601
Estimate 30: 5.9343849251704155
Estimate 40: 5.930753144243958
```

```
Estimate 50: 5.930416824203049

Imputation/reconstruction error: 0.07
```

[18]:

	sepal_length	sepal_width	petal_length	petal_width
98	5.1	2.5	3.000000	1.1
99	5.7	2.8	4.100000	1.3
100	6.3	3.3	5.930417	2.5
101	5.8	2.7	5.100000	1.9
102	7.1	3.0	5.900000	2.1



4.3 [3pt] Plot the entire Iris dataset (no split, keep the classes) projected into 2 dimensions using PCA. Use scikit-learn to compute the principal components.

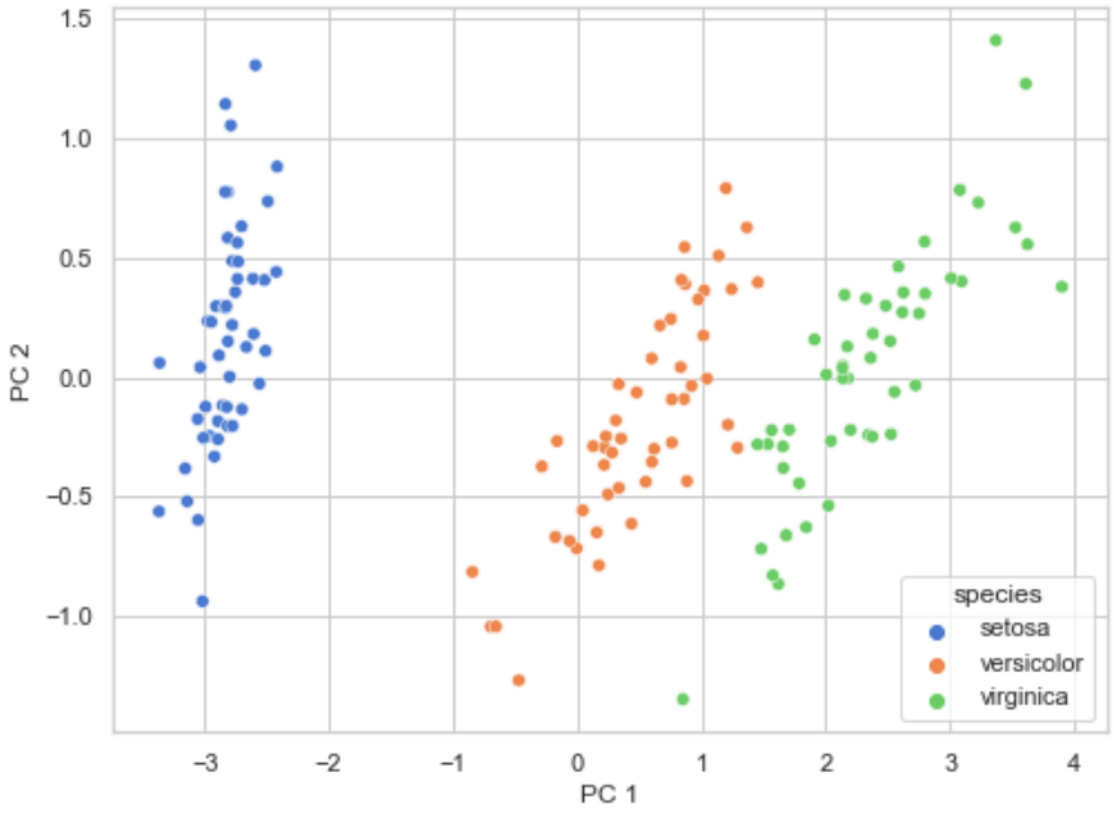
- Yes, you need to reproduce the same picture as in the slides :)
- You need a fresh copy of the Iris dataset, then `sklearn` requires the labels to be numeric. Do you remember the `astype('category').cat.codes` trick?
- Check the documentation of PCA. You need to set the `n_components` parameter.
- Projecting data on the principal components is much, much easier by using the `transform()` function.
- For a neat one-line plot with Seaborn, convert the projected data back to a DataFrame and name the columns! Then add back the `species` column so you can use `hue` ;) and use a nice

palette!

[9]:
```
from sklearn.decomposition import PCA
iris = sns.load_dataset('iris') # Full data, 4 features + labels, 3 classes
# Convert species to numerical values
orig_species = iris['species'] # We can use this later for the legend
iris['species'] = iris['species'].astype('category').cat.codes
# Train on the dataset, keep only two principal components
pca = PCA(n_components=2).fit(iris)

# Project the dataset to these two components (naming columns optional but neat)
iris_2D = pd.DataFrame(pca.transform(iris), columns=['PC 1', 'PC 2'])
iris_2D['species'] = orig_species # let's put the labels back for readability
print(iris_2D.iloc[::20]) # have you encountered the "step" range parameter yet?
sns.scatterplot(data=iris_2D, x='PC 1', y='PC 2', hue='species',␣
↪palette='muted')
```

```
        PC 1      PC 2     species
0  -2.865415  0.296295      setosa
20 -2.516995  0.405236      setosa
40 -2.945219  0.230261      setosa
60 -0.470386 -1.265860  versicolor
80 -0.061676 -0.683733  versicolor
100  2.723151 -0.033898   virginica
120  2.624107  0.353673   virginica
140  2.518935  0.150312   virginica
```

[9]: <AxesSubplot:xlabel='PC 1', ylabel='PC 2'>



4.4 [2pt] Explain (in English) the relationship between (classic) recommender systems and denoising. Then go one step further: to understand the current state of the art (not covered in the lecture), explain a recommender system in term of *mapping*.

- Modern recommender systems rarely use matrix decomposition approaches. Better results have been obtained modeling the mapping directly with flexible, generic function approximators with high generalization capabilities such as neural networks.

Data imputation corresponds to denoising an initial estimate of a missing value. Recommender systems impute user preferences (which have not been stated before) based on the available preferences (of the same user on other items, and of other users on the target item).

Recommender systems thus map the inherent characteristics of users and items into predicting missing preferences. This can be seen as a multi-step approach: the user features can be extracted by the preferences stated on other items, while the item's features can be extracted by the preferences stated on it by other users. A second step can then map these two informative features into a final *regression* of the target preference (of target user on target item).
This can of course be trained in a supervised learning fashion, by predicting the ratings actually present in the dataset, and computing a Loss over each prediction.

6 At the end of the exercise

Bonus question with no points! Answering this will have no influence on your scoring, not at the assignment and not towards the exam score – really feel free to ignore it with no consequence. But solving it will reward you with skills that will make the next lectures easier, give you real applications, and will be 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 :)

BONUS [ZERO pt] Clustering is the core of UL. Master k-means with this [tutorial].

BONUS [ZERO pt] Curious about implementing SVD in Python? It's not hard, here's a good tutorial: [link].

BONUS [ZERO pt] Over the years, I found that whipping out a quick PCA often makes visual analysis of complex data much clearer and with minimal investment. Follow this tutorial to get some experience at it. Challenge: no copy+paste allowed, type everything: muscle memory is much more effective at retaining experience than passive study.

Particularly useful is the discussion at the end: learn that Dimensionality Reduction *hides* information!! It is extremely dangerous to found your decisions on a PCA plot on a subset of axis (e.g. 2), people have lost entire careers on that! As always, each tools has its own utility and drawbacks, and you need to learn the consequences BEFORE you blindly call a library someone else wrote and bet your whole career on its output. :)

BONUS [ZERO pt] Dictionary-based learning has tons of applications. This scikit-learn page contains a good explanation, reference to a library algorithm, an example, and even a link to the paper which published the algorithm [link]. I definitely suggest you have a good look at it.

BONUS [ZERO pt] Once you have a dictionary, you can learn about Sparse Coding here: [link]. If you want to see a cool application, Google for "super resolution": although some recent results use Neural Networks, early Sparse Coding results were used to detect congenital heart problems in newborns, where their hearts are too small for defects to be visible on normal-resolution MRI scans. *Enhance! (cit.)*

6.0.1 Final considerations

- Supervised Learning implies the presence of a *supervisor* in the data preparation: an omniscient expert, an *oracle* that provides the *correct* labels. Yet this is still either a human (which means limited data, human errors, time constraints, etc.), or (lately more common) another algorithm, trusted to be exact (but have you ever heard of bug-free code?). In Deep

Reinforcement Learning we will see that the reward function is learned through SL; in Self-Supervised Learning and in Embeddings applications it is typically an Unsupervised Learning algorithm to provide the oracle.

- All applications of UL stem from two concepts:
  - **Similarity**: similar things are put together, different things are separated.
  - **Information**: data contains redundancy and noise which can be mitigated by studying/extracting global patterns and references. What interests us is the underlying *information*, the true behavior of the underlying generating function.
- Unsupervised Learning is almost always present one way or another in complex applications, yet rarely recognized or credited for what it is – it's always called something like "embedding", "pre-processing", "cleaning", "aggregation" etc. Plain "UL" is so old school that nobody wants to say they are doing it, basing their whole fancy Deep Learning models on its output. Learn to recognize when UL is applied, and the competence you gained today will find more applications than you imagine.

assignment_08_solution

April 15, 2023

Please fill in your name and that of your teammate.

You:

Teammate:

[1]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
sns.set(rc={'figure.figsize':(8,6)}, style="whitegrid")
```

1 Introduction

Welcome to the eighth lab. Today we have *way too many* topics to cover to do everything by hand as usual, so I selected different depths to each topic to make sure you gain full insight and applicable experience.

As you go through the exercise and you apply algorithm after algorithm, method after method, I want you to think about the actual **competence** you are accumulating over the months, both theoretical and applied. Think about it, and be confident: covering so many, so different algorithms in a single lab may sound like a challenge to the version of Past You from barely two months ago, but I believe Today's You is capable of taking on this whale of a lab and have space for more.

Working with many algorithms gives me another chance to shake you out of your confidence zone with respect to *data processing* . Basically each algorithm requires different formats, so you cannot just define the data on top and keep reusing it: you will need to re-load the dataset for each exercise, applying a different processing each time. Be flexible, and don't forget your train-test splits (and their correct usage) – I should not need to mention it anymore, right? :)

Good luck, have fun!

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 16 points (out of 24 available) to pass (66%).

2 1. Fundamentals

1.1 [1pt] Write an example (in English) of a Machine Learning application for which the Supervised Learning paradigm is not (directly) applicable. Autonomous Driving: for SL you need the correct output for every input, but with autonomous driving (i) there could be multiple equally correct reactions to a certain input (e.g. dodge a pedestrian vs. break), and (ii) building a dataset of correct reactions is unfeasible for a single driver, while aggregating information from multiple drivers means inconsistencies in the driver skills and decision process.

1.2 [1pt] Write an example (in English) of a Machine Learning application for which the Unsupervised Learning paradigm is an ideal choice. Image Segmentation: by finding groups of pixels in an image with (i) similar colors and (i) within a minimal distance from each other, it is possible to extract shapes and build masks. These approaches still today can offer better performance than Deep Learning in cases where the training data is scarce and the colors/shapes are obvious.

Careful: if you used the same example for both questions above, something may not be right. These questions are meant to make you think about the *difference* between SL and UL: there is something that is necessary for SL and ignored in UL, so an ideal SL application has that something, while an ideal UL application does not (or SL would be better suited!).

3 2. Clustering

2.1 [2pt] Explain the k-means algorithm using a few words of your own. Particularly, state any requirements, and what the user needs to define.

- To use "your own words", a trick is to read the slide, decide which things you need to mention (feel free to list a few keywords), then close the slides. Now imagine speaking to a friend who has only basic technical background (aka "rubberducking", Google it!). Explain to this friend the things that you previously decided to mention.
- No need to go crazy. This is a type of answer that you will need to repeat over and over, refining it over time: it cannot be perfect the first time. For example, to convince your boss to let you use a particular method at work, or to supervise someone with less knowledge in the field. It's not a right/wrong question: you need to show that you are competent, select important key points, be brief and precise.
- As usual, copy+paste from the slides scores 0 points, while a sincere, fair try (that is not horribly wrong) will give you a pass. So don't worry too much :)

The algorithm k-means is an Unsupervised Learning clustering method. It means that tries to group close/similar points together, and for each group it chooses a representative element that is kind of their average. To run the algorithm you only need inputs, no labels; you also need to pass a target number of clusters k and a similarity measure (e.g. the Euclidean distance works, if the features are continuous). The algorithm works by spreading those means as much as possible,

while maintaining them centered to clusters because it penalizes the cumulative distance between a cluster's mean and the cluster's points.

For the next question, we need to understand how to evaluate a clustering algorithm. The main difference between clustering and classification is that, well, it's UL not SL: the labels are not involved in the training, and they should not be involved in the testing. So how do you test the performance of a clustering algorithm?

Each mean/cluster gets a numerical identifier, the only problem is that the number does not correspond to our labels because it's assigned randomly based on initialization. The most naïve way then is to brute-force all mappings between the labels and the cluster numbers: the one that makes the most sense is the one that should be used for evaluation. Since this is orthogonal to the lecture and may take a long time to debug, here is a snippet of code that does that.

Read it, understand it, play with it, and possibly improve it. Bruteforcing is rarely optimal, which is the very reason why ML exists :)
(note: it may be easier to understand it if you first go ahead with answering the next question first, then come back to this)

[2]:
```
# Goal: convert the labels to the cluster numbers generated by k-means
import itertools
species_names = sns.load_dataset('iris').species.unique()
possible_codes = itertools.permutations(range(len(species_names)))
converters = [dict(zip(species_names, perm)) for perm in possible_codes]
# try printing each of these variables and understand what they do

def cluster_to_class(model, fn_that_counts_misclassified, x_test, y_test):
    min_score = np.Infinity # we saw how to write a minimizer already right?
    right_conversion = None
    for converter in converters:
        conv_y_test = y_test.replace(converter) # conveniently works with␣
 ↪`dict`s
        misclassified = fn_that_counts_misclassified(model, x_test, conv_y_test)
        if misclassified < min_score:
            min_score = misclassified
            right_conversion = converter
    return right_conversion

## to use this function, you will need something like this
# right_conversion = cluster_to_class(k_means_model, my_misclass_fn, x_test,␣
 ↪y_test)
# conv_y_test = y_test.replace(right_conversion)
# k_means_misclassified = my_misclass_fn(k_means_model, x_test, conv_y_test)
```

2.2 [3pt] Apply the scikit-learn implementation of the k-means algorithm to the Iris dataset (4 features, but drop the labels for training), and print a performance score of your choice.

- **IMPORTANT:** the "number of misclassified points" is not an UL performance score, be-

cause UL does not see "classes" and thus does not do "classification". The function `score()` will ignore the labels and print "strange numbers". What are those? No worries though, you know how to make your own scoring from the past labs, right?

- This also mean that if you have points originally belonging to another class, which are however mixed in the cluster of another class (e.g. because of noise), it is correct of the algorithm to put them in the same cluster, even though you will get "misclustered" below. Always interpret your performance metrics! Try printing the points and the cluster centroids for example.
- Of course you want to pass $k = 3$.
- Passing the trained `KMeans` object to `print()` shows several useful parameters and their used values (defaults unless otherwise specified).
- After you get it to work though, why don't you try $k = 2$ or $k = 4$ and see what happens when you have to *guess* k (which would be the normal case in a real application).
- Notice how k-means is *very* sensitive to initialization. To get a consistently better result you may want to explore options `max_iter`, `n_jobs` and of course `init`.

[4]:
```
iris = sns.load_dataset('iris')
# Convert species to numerical values
orig_species = iris['species'] # We can use this later
# iris['species'] = iris['species'].astype('category').cat.codes
train, test = train_test_split(iris, test_size=0.2) # 80-20 split

x_train = train.iloc[:,:-1]
# y_train = train['species'] # UL does not need this
x_test = test.iloc[:,:-1]
y_test = test['species'] # This is used to see how well we do _without_␣
 ↪training labels


from sklearn.cluster import KMeans
k = 3

k_means = KMeans(n_clusters=k, max_iter=100000).fit(x_train)

## No classifier, no score
# print(k_means.score(x_test, y_test))
## Nope, also this only supports classifiers
# plot_confusion_matrix(k_means, x_test, y_test)

# But luckily we know how to handle this, right? Easy!
def correctly_clustered(model, points, clusters, verbose=False,␣
 ↪text='Misclustered:'):
    predictions = model.predict(points)
    misclustered = (clusters != predictions).sum()
    if verbose:
        npoints = len(predictions)
```

```
        print(f"{text} {misclustered}/{npoints}"
              f"({round(misclustered*100/npoints)}%)")
    return misclustered

right_conversion = cluster_to_class(k_means, correctly_clustered, x_test,␣
 ↪y_test)
y_test_clusters = y_test.replace(right_conversion)
k_means_misclassified = correctly_clustered(k_means, x_test, y_test_clusters,␣
 ↪verbose=True)
```

```
Misclustered: 5/30(17%)
```

2.3 [1pt] Plot the centroids learned with k-means on top of the data.

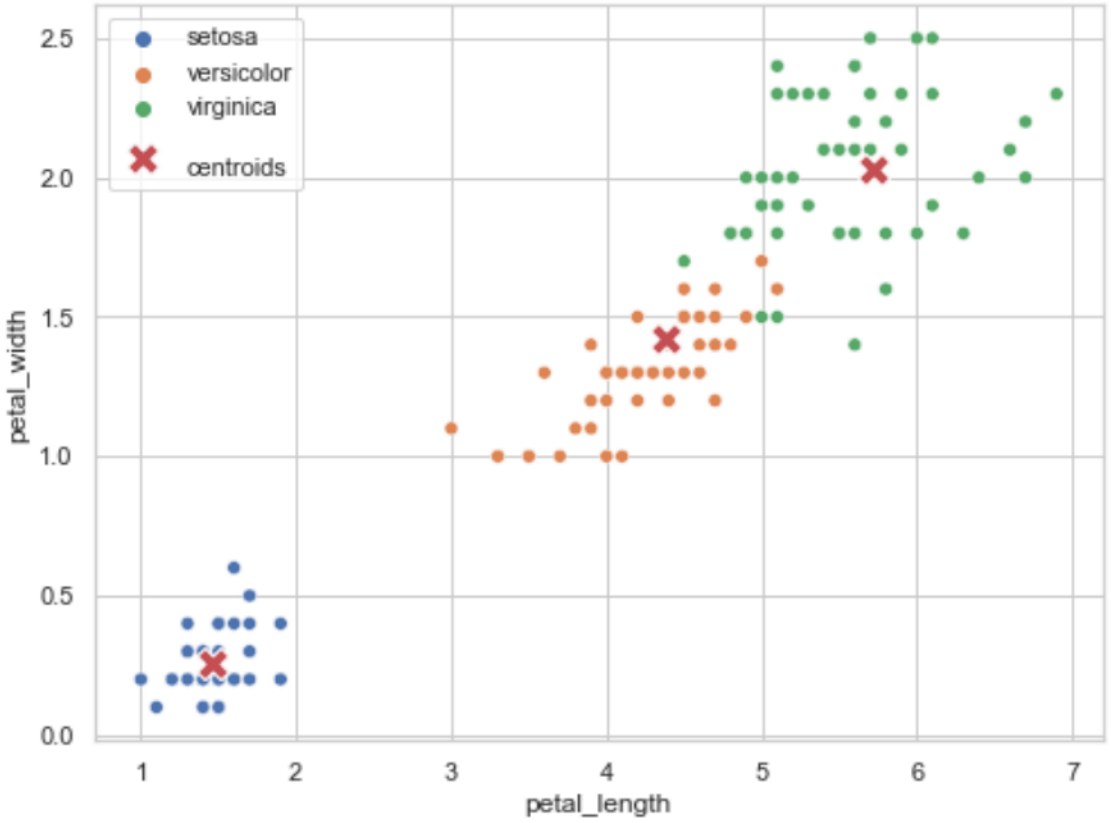
- To get the centroids coordinates, access attribute `cluster_centers_` of the KMeans object. Here are some options I passed to `scatterplot` for visibility (remember you can use the double-splat to transform the dict into keyword parameters):

```
kwargs = {'marker':'X', 'color':'r', 's':200, 'label':'centroids'}
```

- The question does not specify the details of what to plot, so it's up to you to provide a correct and useful interpretation. You learned how to make useful plots, just be confident.
- The simplest is of course to reproduce what we saw so far: one plot, `petal_width` vs. `petal_length`. As they are the last two features, make sure to pick the corresponding coordinates from the centroids. Remember your ranges and your `transpose()` ;)
- Even fancier: why not converting it to a DataFrame? Remember to drop the `species` column when constructing the `df`, as the centroids (learned with UL) have no species information (and thus one less column than iris): `columns=iris.columns.drop('species')`
- After answering correctly, if you want to learn something useful and fancy, try plotting a [PairGrid] that mimics a PairPlot but with added clusters off-diagonal. If you want the usual distributions on the diagonal, it is time to learn it is done with *Density Estimation* (which is what you learned to do for Gaussians in NB), automated in Seaborn with `kdeplot()`.

[4]:
```
sns.scatterplot(data=iris, x='petal_length', y='petal_width', hue='species')
centroids = k_means.cluster_centers_
# Remember the double splat `**`? It unwraps `dict`s into keyword parameters␣
 ↪(or vice-versa)
opts = {'marker':'X', 'color':'r', 's':200, 'label':'\ncentroids'} # what does␣
 ↪the '\n' do?

## Fancy
# cxs, cys = centroids[:,2:].transpose()
# sns.scatterplot(x=cxs, y=cys, **opts) # here's the d-splat

## Fancier? More readable code => less bugs, so always better explicit!
centroids = pd.DataFrame(centroids, columns=iris.columns.drop('species'))
sns.scatterplot(data=centroids, x='petal_length', y='petal_width', **opts) #␣
 ↪d-splat
```

[4]: <AxesSubplot:xlabel='petal_length', ylabel='petal_width'>



Note: this is a very basic application: while a decent knowledge of k-means can typically be useful in itself, the focus here is to cements your understanding of clustering, centroid, expectation maximization, and the difference between clustering and classification. For further reading, I strongly suggest you have a look at [this very complete tutorial].

2.4 [2pt] Train a scikit-learn OneClassSVM on the *versicolor* class of the Iris dataset, and print the number of missed outliers.

- Careful with the input: you need to train this SVM only on the subset of the train data where the species is `versicolor`. That's "one-class". The training should not have access to data from the other two species.
- Also remember to drop the species column (as always) after selecting the lines with `versicolor`.
- The test inputs should work as expected, but the test labels should be converted so that `versicolor` is `1` and the others are `-1` (because those are the model outputs for "normal" and "outlier").
- Again, to compute the missed outliers, you cannot use `score()` or `plot_confusion_matrix()` because technically it's not a classifier. But you already made a function of the scoring code for the previous question right?

[5]:
```
from sklearn.svm import OneClassSVM

iris = sns.load_dataset('iris')
train, test = train_test_split(iris, test_size=0.2) # 80-20 split
oc_x_train = train[train['species'] == 'versicolor'] # train _only_ on target␣
 ↪class
oc_x_train = oc_x_train.drop('species', axis=1) # remember to drop the label
# Everything that is not in our class gets predicted as "other" (`-1`)
oc_y_outliers = np.array([1 if label == 'versicolor' else -1 for label in␣
 ↪y_test])
# Of course you could have done the conversion between the split,
# but I find it more readable this way

oc_svm = OneClassSVM().fit(oc_x_train)

# Remember this is not classification! It is training only on one class,
# then detecting outliers. It cannot know that two of the species intersect.
# Thereby you should not expect to reach a perfect score, right?
oc_svm_missed = my_score(oc_svm, x_test, oc_y_outliers, verbose=True,␣
 ↪text='Missed')
```

```
Missed 5/30 (17%)
```

4 3. Compression and encoding

There are too many topics to cover for this lab. Having to choose one to cut off from practice, I had to objectively decide to remove my favorite one: compression and encoding. The reason is that in most jobs experience in the others will be more useful, while you will still see plenty of encoding techniques over the course. And the concept of dictionary building is related to k-means and feature extraction anyway.

On the other hand, understanding dictionaries as features, and encodings as mappings, allows for a much deeper competence and broader flexibility in the field than being stuck to only the "big guns" of Deep Learning for this task.

So my gift to you is one of my favorite papers so far: "The Importance of Encoding Versus Training with Sparse Coding and Vector Quantization", from A. Coates and A. Ng [link].

If you want to learn the state of the art, in any scientific field, you need to learn to read papers. A good suggestion not to be overwhelmed, especially at the beginning while building knowledge and glossary, is to read it this way

- First the abstract
- Then think about it and read the abstract again
- Now read the introduction, but don't fret about terms you don't understand, it's normal
- Next read the conclusion, and make sure their claims make sense with what you read so far
- The "discussion" explains how they interpreted their results and built the conclusion, which could be invaluable to understand their claims

- If you want more detail on the "how", check out the "method" section (here called "learning framework")
- The "related work" (sometimes "literature review") gives you pointers to extend your study on the field and applications
- Do not overlook "experimental results", as it will give you the means to reproduce their results. And yes, chances are your thesis (especially if Master) will begin by asking you to reproduce a paper's result. Repeatability and verification (the hypothesis needs to be falsifiable) are at the very core of the scientific process.

So: go at least through abstract, introduction and conclusions, and then answer the following question:

3.1 [4pt] In the conclusion of the paper "The Importance of Encoding Versus Training with Sparse Coding and Vector Quantization", which part do the authors find more effective, the encoding (i.e. decision-making) or the dictionary training (i.e. feature extraction)? Reflect on the consequences on modeling, and express your opinion on Feature Extraction and Decision Mappings. The solution will contain no "right" answer, just **my** answer. It actually constitutes one the foundations of my research. Full points will be awarded to anyone (i) asserting their opinion, and (ii) justifying it with findings from the paper.

The authors find the encoding much more significant than the dictionary training, to the point that sophisticated encodings can produce useful codes based on features generated from the data via random processes.

This implies that there is more strength to be found in function composition than in *careful* data preparation, and that the last decision mapping is incomparably more important than the initial, feature extraction mappings.

When attacking a problem with Machine Learning, (i) do enough cleaning to make algorithms work, (ii) do (possibly several layers of) feature extraction to increase the abstraction level, then (iii) very carefully prepare a final decision mapping for best performance.

You will see this pattern (always inherent, often unaware, sometimes ignored) in **all** the most successful ML applications to date.

5 4. Matrix decomposition

4.1 [1pt] For this data imputation exercise, use the entire Iris dataset (no split in train/test, but do drop `species`). Select the value in row index 100 column index 2, and store it in an outside variable. Then delete it from the dataset.

- To delete a value from a dataset, simply assign the "`not a number`" value (`np.nan`) to the corresponding element.
- Have you tried using `drop()` to remove a column? Remember that the default axis is the rows, so you need to pass `axis=1` or `axis='columns'` to drop a column by name.
- You can print a few rows around your target to verify everything is going as you expect.

[12]:
```
orig = sns.load_dataset('iris').drop('species', axis='columns')
nrow, ncol = (100, 2)
trg = orig.iloc[nrow, ncol] # splat seems not to work here?
```

```
print(f"Target: {trg}")
orig.iloc[nrow-2:nrow+2]
```

```
Target: 6.0
```

[12]:

	sepal_length	sepal_width	petal_length	petal_width
98	5.1	2.5	3.0	1.1
99	5.7	2.8	4.1	1.3
100	6.3	3.3	6.0	2.5
101	5.8	2.7	5.1	1.9

[13]:
```
orig.iloc[nrow, ncol] = np.nan
orig.iloc[nrow-2:nrow+2]
```

[13]:

	sepal_length	sepal_width	petal_length	petal_width
98	5.1	2.5	3.0	1.1
99	5.7	2.8	4.1	1.3
100	6.3	3.3	NaN	2.5
101	5.8	2.7	5.1	1.9

4.2 [4pt] Reconstruct (impute) the missing value using SVD and dimensionality reduction-based denoising. Do not use scikit-learn. Use the SVD method from `np.linalg`. Print the (absolute) difference between the original and reconstructed values.

- Ok, relax, this is not your first complex question. As usual, deconstruct the process in smaller, achievable goals, then work through them step by step.
- The first thing you need to do is to get rid of is the "hole" in the data, because SVD will not work with `nan` values. Simply patch it with an average of the values above and below in the same feature. We know this is not ideal, but no worries. BTW congratulations with this you just learned the foundation of the ***k*-nearest-neighbors algorithm (KNN)**.
- Now decompose the entire dataframe('s data matrix) using the SVD implementation in numpy's `linalg` module. Read carefully the documentation: it does not return `u`, Σ and `v` as expected from the lecture! Instead you get `v` as expected $n \times n$; `s` a vector of size n containing the σ eigenvalues (the diagonal of Σ, remember?); and `vh` $m \times m$ is the transposition of `v` – saves a transpose, but remember you will need to zero a *row* not a *column*.
- Next you want to drop the least contributing eigenvector. Find the smallest non-zero eigenvalue, and set to zero the corresponding column in `u` and row in `vh`. The relative size also tells you how much precision will you be losing with this reduction.
- Now you can already reconstruct the data. Remember the order of the dot products matters. Also, you need to build your Σ. There's an example in the documentation of SVD. Importantly: Σ is rectangular, the eigenvalues go in the diagonal of the first "square" of this matrix, the rest is zeros. You can set a range of rows and columns of a numpy rectangular matrix to (the values of) a diagonal matrix created with `np.diag()`, just match the sizes.
- Fetch the value in the target element's position in the reconstruction matrix. Has it changed w.r.t. its initial estimate? Print the difference with the original and technically you're done.
- If you are unsatisfied with the result though, you can run the cycle a few times. Place the code written so far in a function, so you can iterate multiple calls. Remember that you need

to insert the new value in the *original matrix* , and then loop all your calls to the denoising function on this matrix. Looping on the reconstruction is a common error which may cost you a lot! With every denoising you are losing a bit of information; copying only the value you are denoising reintegrates the information in the rest of the matrix, allowing for a much more accurate result.

- I converge (i.e. no more significant changes) to within 0.07 of the correct value in 50 iterations. I also simply save the errors (abs diff) at every iteration, then do the usual `lineplot`. Nothing new, but these sanity checks are priceless when working with ML.
- Alternatively: what happens if you drop two columns instead of one?

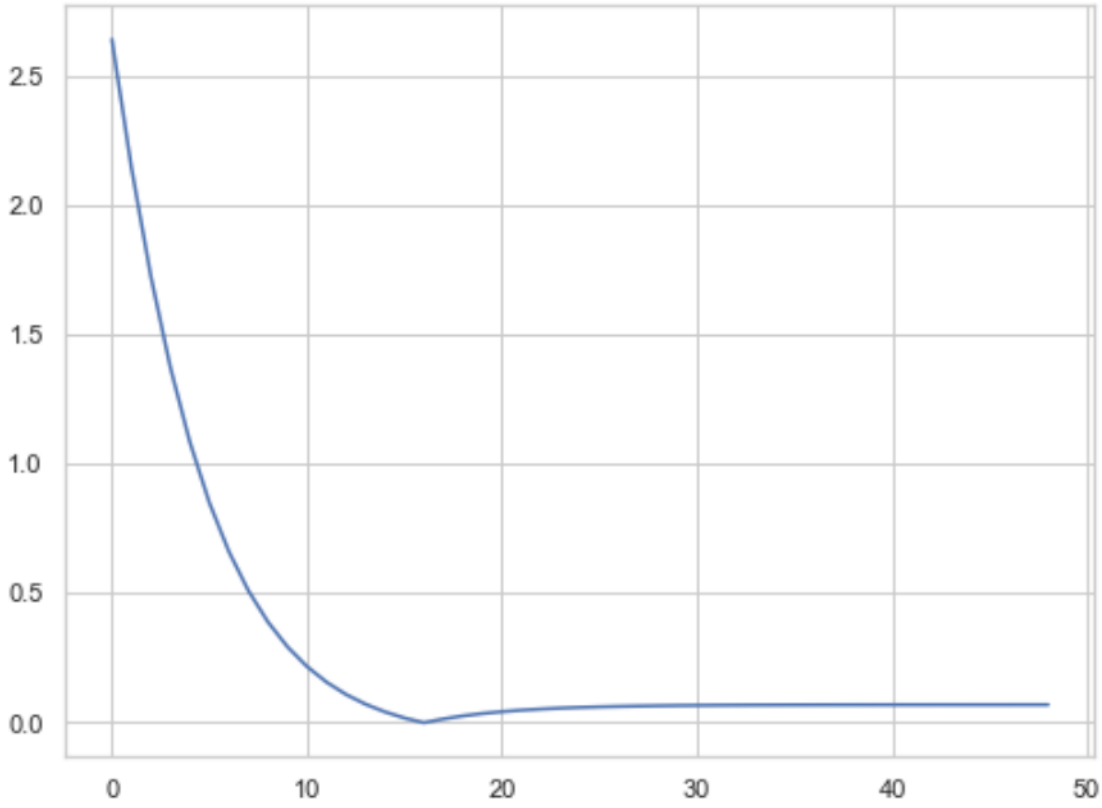
[18]:
```
first_estimate = orig.iloc[nrow-1, ncol] + orig.iloc[nrow+1, ncol]
orig.iloc[nrow, ncol] = first_estimate
print(f"Target: {trg}\nInitial estimate: {first_estimate}")

def denoise(mat):
    o_nrows, o_ncols = mat.shape
    u, s, vh = np.linalg.svd(mat, full_matrices=True)

    # Drop least contributing eigenvector
    u[:, -1] = 0 # here is the last column
    s[-1] = 0 # not necessary, but oh well consistency
    vh[-1, :] = 0 # here is a row

    # We need to reconstruct the data to the original dimensionality
    sigma = np.zeros((o_nrows, o_ncols), dtype=s.dtype)
    sigma[:o_ncols, :o_ncols] = np.diag(s)
    rec = pd.DataFrame(np.dot(u, np.dot(sigma, vh)))
    return rec

errors = []
for nstep in range(50-1): # initial estimate is the average above
    rec = denoise(orig).iloc[nrow, ncol]
    errors.append(abs(trg-rec))
    if (nstep+2)%10==0: print(f"Estimate {nstep+2}: {rec}")
    orig.iloc[nrow, ncol] = rec

print(f"\nImputation/reconstruction error: {round(abs(trg-rec), 2)}")
# You may see a *concave* in the error plot, and this is not wrong
# Remember that this is UL, thus BLIND! This is not a Loss!
sns.lineplot(x=range(len(errors)), y=errors)
orig.iloc[98:103]
```

```
Target: 6.0
Initial estimate: 9.2
Estimate 10: 6.38941334370923
Estimate 20: 5.973550066828601
Estimate 30: 5.9343849251704155
Estimate 40: 5.930753144243958
```

```
Estimate 50: 5.930416824203049

Imputation/reconstruction error: 0.07
```

[18]:

	sepal_length	sepal_width	petal_length	petal_width
98	5.1	2.5	3.000000	1.1
99	5.7	2.8	4.100000	1.3
100	6.3	3.3	5.930417	2.5
101	5.8	2.7	5.100000	1.9
102	7.1	3.0	5.900000	2.1



4.3 [3pt] Plot the entire Iris dataset (no split, keep the classes) projected into 2 dimensions using PCA. Use scikit-learn to compute the principal components.

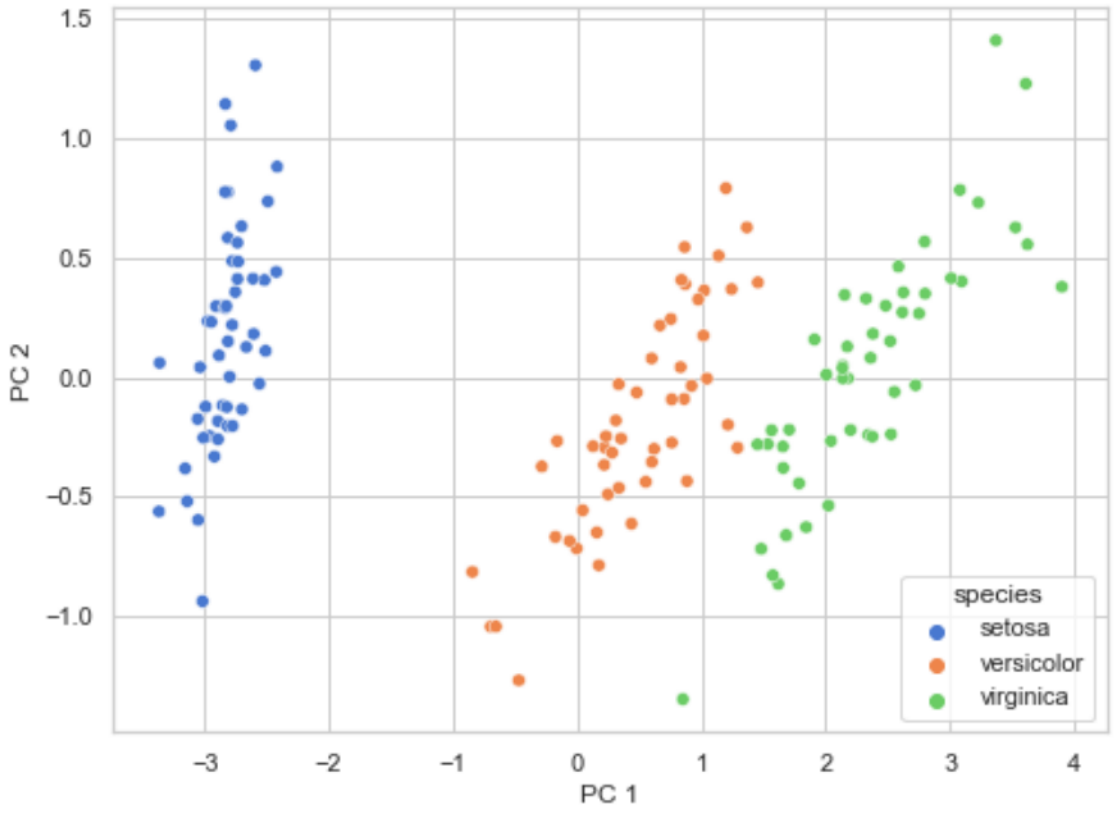
- Yes, you need to reproduce the same picture as in the slides :)
- You need a fresh copy of the Iris dataset, then `sklearn` requires the labels to be numeric. Do you remember the `astype('category').cat.codes` trick?
- Check the documentation of PCA. You need to set the `n_components` parameter.
- Projecting data on the principal components is much, much easier by using the `transform()` function.
- For a neat one-line plot with Seaborn, convert the projected data back to a DataFrame and name the columns! Then add back the `species` column so you can use `hue` ;) and use a nice

palette!

[9]:
```
from sklearn.decomposition import PCA
iris = sns.load_dataset('iris') # Full data, 4 features + labels, 3 classes
# Convert species to numerical values
orig_species = iris['species'] # We can use this later for the legend
iris['species'] = iris['species'].astype('category').cat.codes
# Train on the dataset, keep only two principal components
pca = PCA(n_components=2).fit(iris)

# Project the dataset to these two components (naming columns optional but neat)
iris_2D = pd.DataFrame(pca.transform(iris), columns=['PC 1', 'PC 2'])
iris_2D['species'] = orig_species # let's put the labels back for readability
print(iris_2D.iloc[::20]) # have you encountered the "step" range parameter yet?
sns.scatterplot(data=iris_2D, x='PC 1', y='PC 2', hue='species',␣
↪palette='muted')
```

```
        PC 1      PC 2     species
0  -2.865415  0.296295      setosa
20 -2.516995  0.405236      setosa
40 -2.945219  0.230261      setosa
60 -0.470386 -1.265860  versicolor
80 -0.061676 -0.683733  versicolor
100  2.723151 -0.033898   virginica
120  2.624107  0.353673   virginica
140  2.518935  0.150312   virginica
```

[9]: <AxesSubplot:xlabel='PC 1', ylabel='PC 2'>



4.4 [2pt] Explain (in English) the relationship between (classic) recommender systems and denoising. Then go one step further: to understand the current state of the art (not covered in the lecture), explain a recommender system in term of *mapping*.

- Modern recommender systems rarely use matrix decomposition approaches. Better results have been obtained modeling the mapping directly with flexible, generic function approximators with high generalization capabilities such as neural networks.

Data imputation corresponds to denoising an initial estimate of a missing value. Recommender systems impute user preferences (which have not been stated before) based on the available preferences (of the same user on other items, and of other users on the target item).

Recommender systems thus map the inherent characteristics of users and items into predicting missing preferences. This can be seen as a multi-step approach: the user features can be extracted by the preferences stated on other items, while the item's features can be extracted by the preferences stated on it by other users. A second step can then map these two informative features into a final *regression* of the target preference (of target user on target item).
This can of course be trained in a supervised learning fashion, by predicting the ratings actually present in the dataset, and computing a Loss over each prediction.

6 At the end of the exercise

Bonus question with no points! Answering this will have no influence on your scoring, not at the assignment and not towards the exam score – really feel free to ignore it with no consequence. But solving it will reward you with skills that will make the next lectures easier, give you real applications, and will be 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 :)

BONUS [ZERO pt] Clustering is the core of UL. Master k-means with this [tutorial].

BONUS [ZERO pt] Curious about implementing SVD in Python? It's not hard, here's a good tutorial: [link].

BONUS [ZERO pt] Over the years, I found that whipping out a quick PCA often makes visual analysis of complex data much clearer and with minimal investment. Follow this tutorial to get some experience at it. Challenge: no copy+paste allowed, type everything: muscle memory is much more effective at retaining experience than passive study.

Particularly useful is the discussion at the end: learn that Dimensionality Reduction *hides* information!! It is extremely dangerous to found your decisions on a PCA plot on a subset of axis (e.g. 2), people have lost entire careers on that! As always, each tools has its own utility and drawbacks, and you need to learn the consequences BEFORE you blindly call a library someone else wrote and bet your whole career on its output. :)

BONUS [ZERO pt] Dictionary-based learning has tons of applications. This scikit-learn page contains a good explanation, reference to a library algorithm, an example, and even a link to the paper which published the algorithm [link]. I definitely suggest you have a good look at it.

BONUS [ZERO pt] Once you have a dictionary, you can learn about Sparse Coding here: [link]. If you want to see a cool application, Google for "super resolution": although some recent results use Neural Networks, early Sparse Coding results were used to detect congenital heart problems in newborns, where their hearts are too small for defects to be visible on normal-resolution MRI scans. *Enhance! (cit.)*

6.0.1 Final considerations

- Supervised Learning implies the presence of a *supervisor* in the data preparation: an omniscient expert, an *oracle* that provides the *correct* labels. Yet this is still either a human (which means limited data, human errors, time constraints, etc.), or (lately more common) another algorithm, trusted to be exact (but have you ever heard of bug-free code?). In Deep

Reinforcement Learning we will see that the reward function is learned through SL; in Self-Supervised Learning and in Embeddings applications it is typically an Unsupervised Learning algorithm to provide the oracle.

- All applications of UL stem from two concepts:
  - **Similarity**: similar things are put together, different things are separated.
  - **Information**: data contains redundancy and noise which can be mitigated by studying/extracting global patterns and references. What interests us is the underlying *information*, the true behavior of the underlying generating function.
- Unsupervised Learning is almost always present one way or another in complex applications, yet rarely recognized or credited for what it is – it's always called something like "embedding", "pre-processing", "cleaning", "aggregation" etc. Plain "UL" is so old school that nobody wants to say they are doing it, basing their whole fancy Deep Learning models on its output. Learn to recognize when UL is applied, and the competence you gained today will find more applications than you imagine.