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'}
```

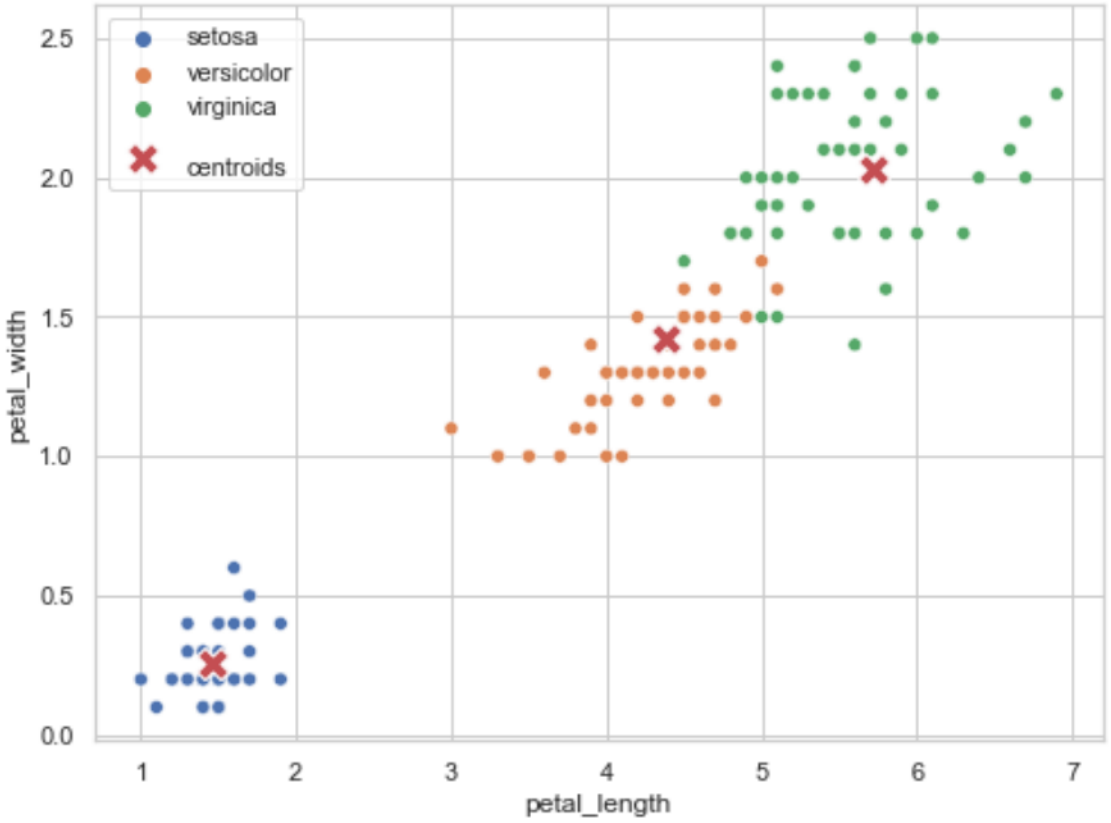
- The question does not specify the details of what to plot, so it's up to you to provide a correct and useful interpretation. You learned how to make useful plots, just be confident.
- The simplest is of course to reproduce what we saw so far: one plot, `petal_width` vs. `petal_length`. As they are 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]
```

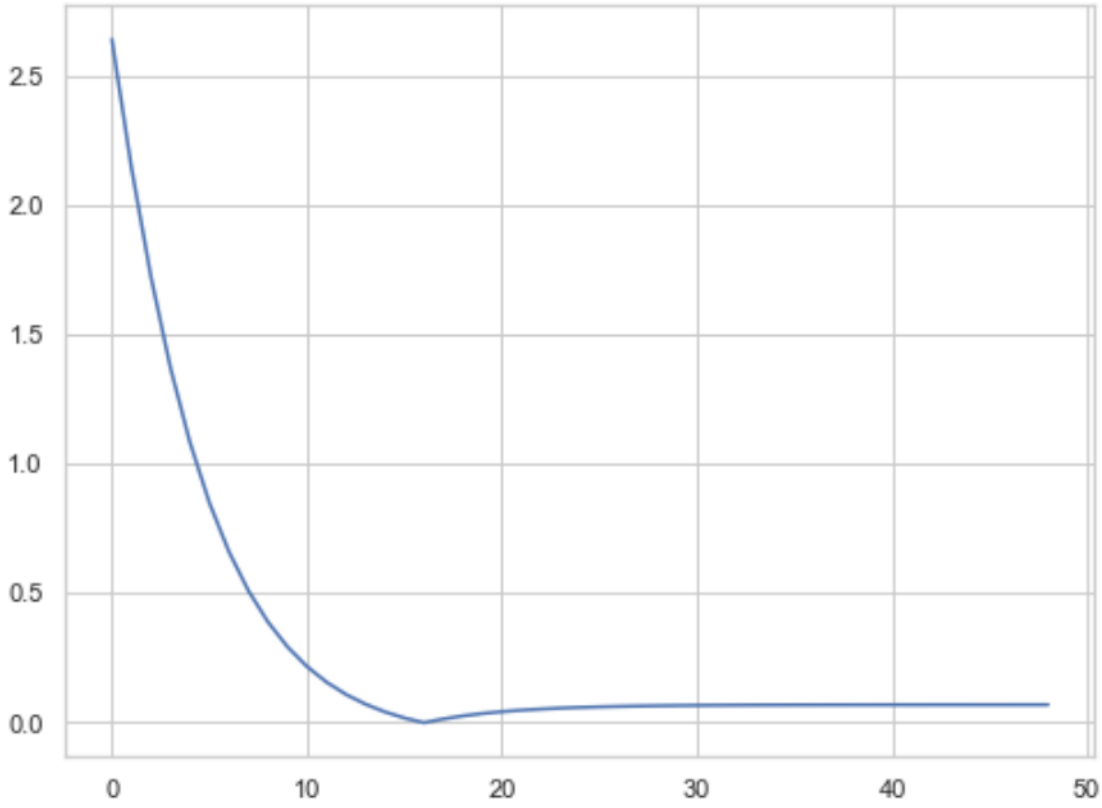
```
Target: 6.0
Initial estimate: 9.2
Estimate 10: 6.38941334370923
Estimate 20: 5.973550066828601
Estimate 30: 5.9343849251704155
Estimate 40: 5.930753144243958
```

```
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

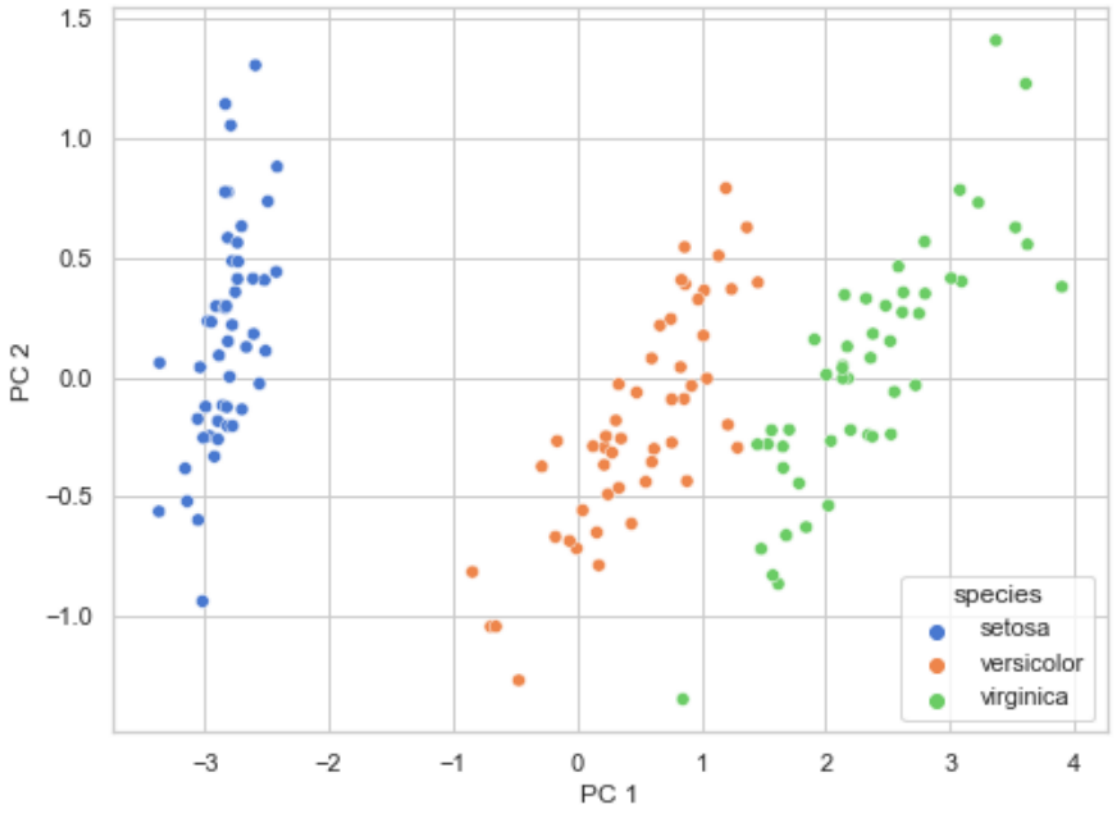
palette!

[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.