assignment_08

April 23, 2024

Please fill in your name and that of your teammate.

You: *Ahonon Gobi Parfait*

Teammate:

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%).

[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")
```

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. R: One example of a Machine Learning application for which the Supervised Learning paradigm is not directly applicable is the case of anomaly detection. In this case, the data is not labeled, and the goal is to identify patterns that do not conform to expected behavior. Anomaly detection can be used in various fields, such as fraud detection, network security, and health monitoring, among others. In this scenario, the lack of labeled data makes it difficult to apply supervised learning algorithms, as they require labeled data to learn the relationship between input features and output labels. Instead, unsupervised learning algorithms, such as clustering or density estimation, can be used to identify anomalies in the data without the need for labeled examples.

1.2 [1pt] Write an example (in English) of a Machine Learning application for which the Unsupervised Learning paradigm is an ideal choice. R: One example of a Machine Learning application for which the Unsupervised Learning paradigm is an ideal choice is customer segmentation. In this case, the goal is to group customers based on their purchasing behavior, preferences, or other characteristics. Since the data is not labeled, unsupervised learning algorithms can be used to identify patterns and group customers into distinct segments without the need for labeled examples. This can help businesses better understand their customer base, tailor marketing strategies, and improve customer satisfaction.

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, then close the slide and imagine a friend with only basic technical background in front of you (aka "rubberducking", Google it!). Now tell this person the things that you decided to mention.
- No need to go crazy. This is a type of (vague!) question that you will need to answer over and over, typically to convince your boss to let you use a particular method, or to guide someone with less knowledge in the field. It's not a right/wrong question: you need to show that you have competence, list the key points, be brief and to the point.
- And of course, copy+paste+change words from the slide will score you 0 points :) while a sincere, fair try that is not wrong will pass, so no worries.

The k-means algorithm is a data clustering technique that divides a dataset into k distinct groups, called clusters. The user needs to define the number of clusters (k) to create. The algorithm then iterates to assign data points to the nearest clusters and updates the cluster centers until convergence, aiming to minimize the variation within each cluster.

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
print(converters)
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)
```

```
[{'setosa': 0, 'versicolor': 1, 'virginica': 2}, {'setosa': 0, 'versicolor': 2,
'virginica': 1}, {'setosa': 1, 'versicolor': 0, 'virginica': 2}, {'setosa': 1,
'versicolor': 2, 'virginica': 0}, {'setosa': 2, 'versicolor': 0, 'virginica':
1}, {'setosa': 2, 'versicolor': 1, 'virginica': 0}]
```

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.

- Of course you want to pass $k = 3$.

- Passing the trained `KMeans` object to `print()` shows several useful parameters and their default values.
- 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 is the normal case in real applications).
- For the performance score, remember this is clustering, not classification, which means `score()` will ignore the labels and print "strange numbers". No worries, you know how to make your own scoring from the past labs, right?
- 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`.

[3]:
```python
from sklearn.datasets import load_iris
from sklearn.cluster import KMeans
import numpy as np

def calculate_wcss(data, k):
    """
    Calculate the within-cluster sum of squares (WCSS) for a given dataset and␣
 ↪number of clusters (k).

    Parameters:
        data (array-like): The input data.
        k (int): The number of clusters.

    Returns:
        float: The WCSS score.
    """
    # Applying k-means algorithm with k clusters
    kmeans = KMeans(n_clusters=k, max_iter=300, n_jobs=-1, init='k-means++')
    kmeans.fit(data)

    # Calculate the within-cluster sum of squares (WCSS)
    wcss = sum(np.min(kmeans.transform(data), axis=1)**2)

    return wcss

# Load the Iris dataset
iris = load_iris()
data = iris.data  # use the data (features) only, excluding the labels

# Define the number of clusters
k = 3

# Applying k-means algorithm with k clusters
kmeans = KMeans(n_clusters=k, max_iter=300, n_jobs=-1, init='k-means++')
kmeans.fit(data)

# Print some useful information about the fitted model
```

```python
print("Parameters of the trained KMeans object:")
print(kmeans)

# Calculate and print the within-cluster sum of squares (WCSS) as a performance␣
 ↪score
wcss_score = calculate_wcss(data, k)
print("WCSS Score:", wcss_score)
```

```
Parameters of the trained KMeans object:
KMeans(n_clusters=3, n_jobs=-1)
WCSS Score: 78.85144142614621
```

```
/home/gobi/anaconda3/lib/python3.9/site-packages/sklearn/cluster/_kmeans.py:792:
FutureWarning: 'n_jobs' was deprecated in version 0.23 and will be removed in
1.0 (renaming of 0.25).
  warnings.warn("'n_jobs' was deprecated in version 0.23 and will be"
/home/gobi/anaconda3/lib/python3.9/site-packages/sklearn/cluster/_kmeans.py:792:
FutureWarning: 'n_jobs' was deprecated in version 0.23 and will be removed in
1.0 (renaming of 0.25).
  warnings.warn("'n_jobs' was deprecated in version 0.23 and will be"
```

```python
[4]: import numpy as np
from sklearn.cluster import KMeans
from sklearn.datasets import load_iris
from sklearn.preprocessing import StandardScaler
import seaborn as sns
import itertools

# Load the Iris dataset
iris = load_iris()
X = iris.data
y = iris.target

# Standardize the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Define the number of clusters (k)
k = 3

# Apply k-means algorithm
kmeans = KMeans(n_clusters=k, random_state=42)
kmeans.fit(X_scaled)

# Define the function that counts misclassified points
def misclass_fn(model, x_test, y_test):
    predictions = model.predict(x_test)
    misclassified = np.sum(predictions != y_test)
    return misclassified

# Define species names and possible mappings between labels and cluster IDs
species_names = iris.target_names
possible_codes = itertools.permutations(range(len(species_names)))
converters = [dict(zip(species_names, perm)) for perm in possible_codes]

# Function to map cluster IDs to labels and compute misclassified points
def cluster_to_class(model, fn_that_counts_misclassified, x_test, y_test):
    min_score = np.Infinity
    right_conversion = None
    for converter in converters:
        conv_y_test = [converter[species_names[label]] for label in y_test]
        misclassified = fn_that_counts_misclassified(model, x_test, conv_y_test)
        if misclassified < min_score:
            min_score = misclassified
            right_conversion = converter
    return right_conversion, min_score

# Function to compute accuracy -  score
def compute_accuracy(model, x_test, y_test):
    predictions = model.predict(x_test)
    correct = np.sum(predictions == y_test)
    total = len(y_test)
    accuracy = correct / total
    return accuracy


# Map cluster IDs to labels and compute misclassified points
right_conversion, misclassified = cluster_to_class(kmeans, misclass_fn,␣
 ↪X_scaled, y)
accuracy = compute_accuracy(kmeans, X_scaled,␣
 ↪[right_conversion[species_names[label]] for label in y])

print("Right Conversion:", right_conversion)
print("Misclassified Points:", misclassified)
print("Accuracy:", accuracy)
```

```
Right Conversion: {'setosa': 1, 'versicolor': 2, 'virginica': 0}
Misclassified Points: 25
Accuracy: 0.8333333333333334
```

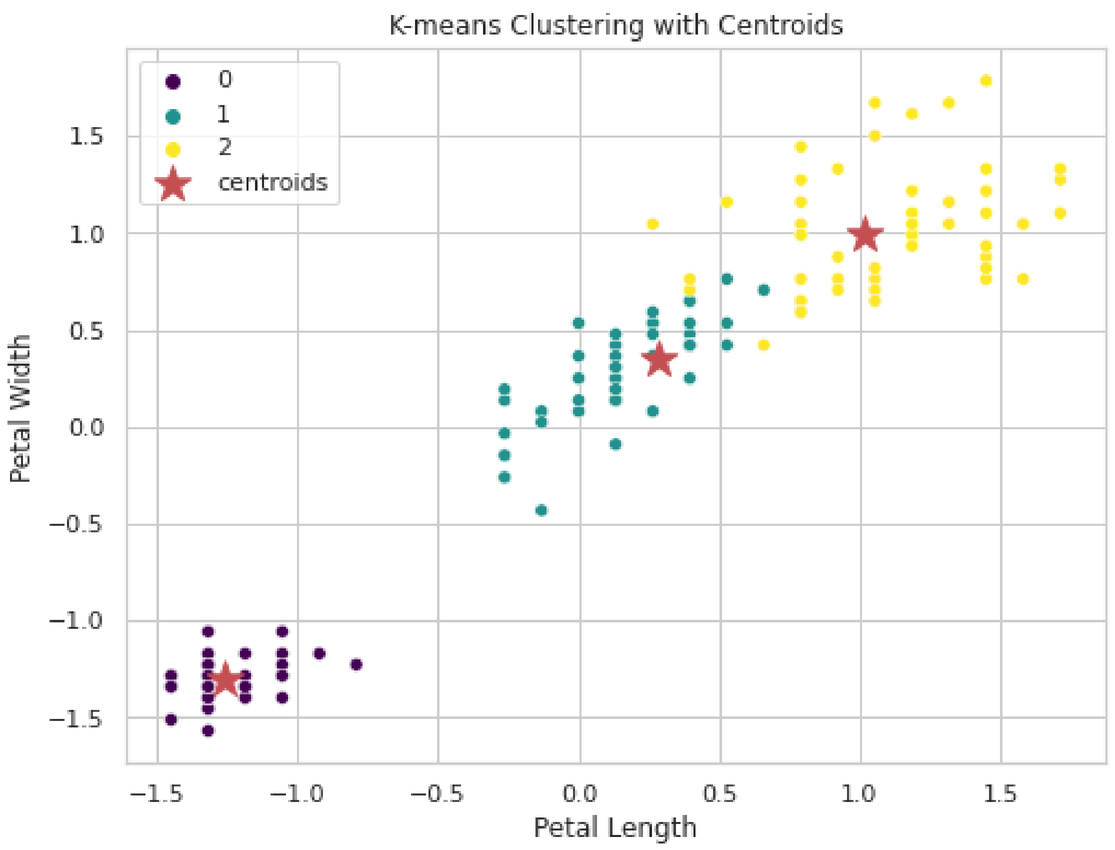
2.3 [1pt] Plot the centroids learned with k-means on top of the data.

- 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()`.

[5]:
```python
import seaborn as sns
import matplotlib.pyplot as plt

# Get the centroids coordinates
centroids = kmeans.cluster_centers_

# Create a scatter plot of the data points using Seaborn
sns.scatterplot(x=X_scaled[:, 3], y=X_scaled[:, 2], hue=y, palette='viridis')

# Overlay the centroids on the scatter plot using Matplotlib
plt.scatter(centroids[:, 3], centroids[:, 2], marker='*', color='r', s=300,␣
 ↪label='centroids')

# Set labels and title
plt.xlabel('Petal Length')
plt.ylabel('Petal Width')
plt.title('K-means Clustering with Centroids')

# Add legend
plt.legend()

# Show plot
plt.show()
```



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?

[6]:
```python
from sklearn.svm import OneClassSVM

# Select only the data points corresponding to the versicolor class
versicolor_mask = iris.target == 1
X_versicolor = X_scaled[versicolor_mask]

# Train the One-Class SVM
svm = OneClassSVM()
svm.fit(X_versicolor)

# Convert test labels so that versicolor is labeled as 1 and the others are␣
 ↪labeled as -1
y_test_binary = np.where(iris.target == 1, 1, -1)

# Define the function that counts misclassified points
def misclass_fn_svm(model, x_test, y_test):
    predictions = model.predict(x_test)
    misclassified = np.sum(predictions != y_test)
    return misclassified
# Compute the number of missed outliers based on the predictions made by the␣
 ↪One-Class SVM
missed_outliers = misclass_fn_svm(svm, X_scaled, y_test_binary)
print("Number of missed outliers:", missed_outliers)
```

```
Number of missed outliers: 25
```

[7]:
```python
from sklearn.svm import OneClassSVM
import numpy as np

# select the versicolor data and target
versicolor_data = iris.data[iris.target == 1]  # Sélectionner les données pour␣
 ↪la classe versicolor
versicolor_target = np.ones(len(versicolor_data))  # Créer des étiquettes pour␣
 ↪la classe versicolor (1)

# Train the OneClassSVM model
svm_model = OneClassSVM()
svm_model.fit(versicolor_data)

# Convert the labels for the test data
def convert_labels(labels):
    return np.where(labels == 1, 1, -1)  # Convertir versicolor en 1 et les␣
 ↪autres en -1

# Calculate the number of missed outliers
def calculate_missed_outliers(model, x_test, y_test):
    y_pred = model.predict(x_test)
    y_test_converted = convert_labels(y_test)
    missed_outliers = np.sum(y_pred != y_test_converted)
    return missed_outliers

# Convert the labels for the test data
test_labels = np.ones(len(versicolor_data))  # Les étiquettes sont toutes␣
 ↪versicolor (1)

# Calculate the number of missed outliers
missed_outliers = calculate_missed_outliers(svm_model, versicolor_data,␣
 ↪test_labels)
print("Nombre d'outliers manqués:", missed_outliers)
```

```
Nombre d'outliers manqués: 24
```

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.

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 110 column index 1, 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.

```
[8]: import seaborn as sns
     import numpy as np

     # Load the Iris dataset
     iris = sns.load_dataset('iris')

     # drop the species column using drop() with axis=1 or axis='columns'
     iris = iris.drop('species', axis=1)
     # Select the value in row index 110 in target_row_idx variable and  , column␣
      ↪index 1 in target_col_idx
     target_row_idx  = 110
     target_col_idx = 1

     value_to_store = iris.iloc[target_row_idx, target_col_idx]

     print(value_to_store)

     # Delete the value from the dataset using
     iris.iloc[110, 1] = np.nan

     # Verify changes by printing a few rows around the target
     print(iris.iloc[108:112, :])
```

```
3.2
     sepal_length  sepal_width  petal_length  petal_width
```

```
108            6.7           2.5            5.8           1.8
109            7.2           3.6            6.1           2.5
110            6.5           NaN            5.1           2.0
111            6.4           2.7            5.3           1.9
```

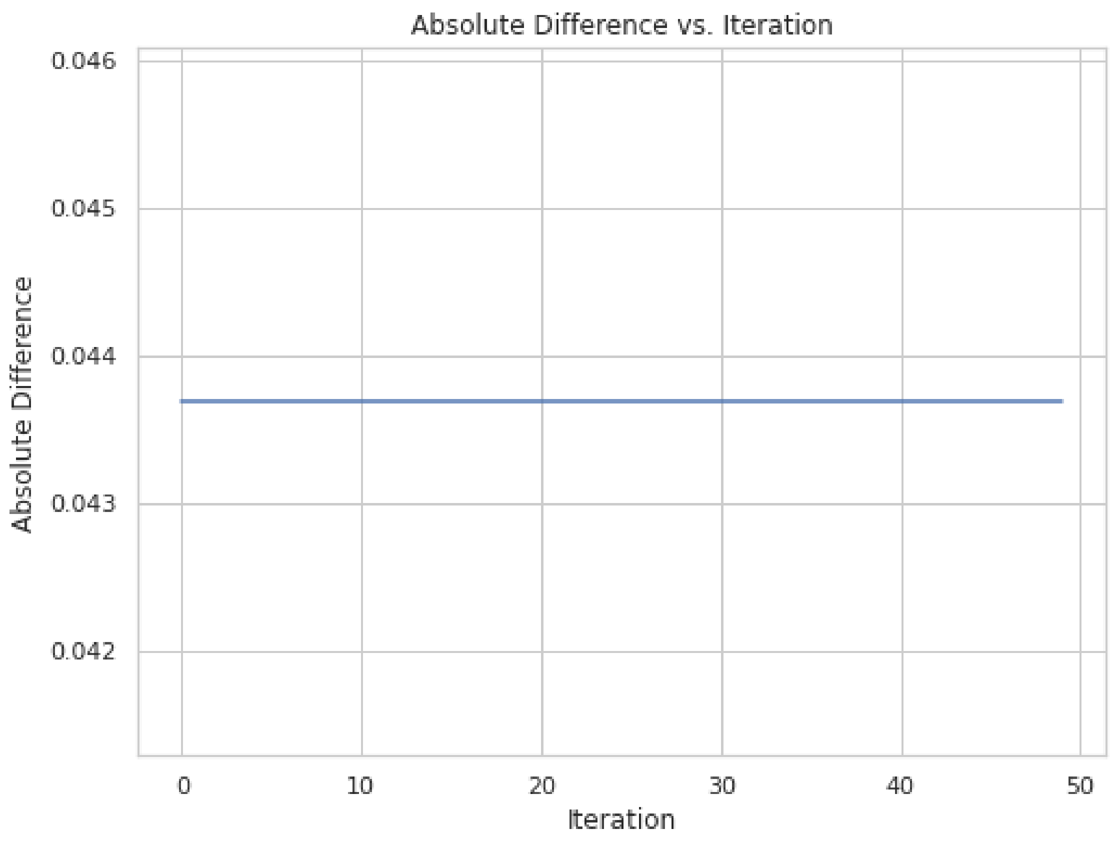
4.2 [4pt] Reconstruct (impute) the missing value using SVD and dimensionality reduction-based denoising. Do not use scikit-learn. Use the SVD method from `np.linalg`. Print the (absolute) difference between the original and reconstructed values.

- 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?

[9]:
```python
import numpy as np
import pandas as pd


# 1. Patch the missing value with the average of the values above and below
def patch_missing_value(data, row_idx, col_idx):
    above_value = data.iloc[row_idx - 1, col_idx]
    below_value = data.iloc[row_idx + 1, col_idx]
    patched_value = (above_value + below_value) / 2
    data.iloc[row_idx, col_idx] = patched_value

# 2. Perform SVD decomposition
def svd_decomposition(data):
    u, s, vh = np.linalg.svd(data, full_matrices=False)
    return u, s, vh

# 3. Drop the least contributing eigenvector
def drop_least_contributing_eigenvector(u, s, vh):
    min_idx = np.argmin(s)
    s[min_idx] = 0
    u[:, min_idx] = 0
    vh[min_idx, :] = 0
    return u, s, vh

# 4. Reconstruct the data matrix
def reconstruct_data(u, s, vh):
    sigma = np.diag(s)
    reconstructed_data = np.dot(u, np.dot(sigma, vh))
    return reconstructed_data

# 5. Calculate the absolute difference between original and reconstructed values
def calculate_absolute_difference(original_value, reconstructed_value):
    abs_diff = np.abs(original_value - reconstructed_value)
    return abs_diff

# 6. Iterate the denoising process
def iterative_denoising(data, target_row_idx, target_col_idx, num_iterations):
    errors = []
    for _ in range(num_iterations):
        patch_missing_value(data, target_row_idx, target_col_idx)
        u, s, vh = svd_decomposition(data.values)
        u, s, vh = drop_least_contributing_eigenvector(u, s, vh)
        reconstructed_data = reconstruct_data(u, s, vh)
        reconstructed_value = reconstructed_data[target_row_idx, target_col_idx]
        original_value = data.iloc[target_row_idx, target_col_idx]
        abs_diff = calculate_absolute_difference(original_value,␣
 ↪reconstructed_value)
        errors.append(abs_diff)
    return reconstructed_data, errors

# Load the Iris dataset
#iris = sns.load_dataset('iris')
```

```python
#Select the value in row index 110, column index 1
#target_row_idx = 110
#target_col_idx = 1

# Store the original value before patching
# original_value = iris.iloc[target_row_idx, target_col_idx]
original_value = value_to_store

# Patch the missing value
patch_missing_value(iris, target_row_idx, target_col_idx)

# Define the number of iterations for iterative denoising
num_iterations = 50

# Perform iterative denoising
reconstructed_data, errors = iterative_denoising(iris, target_row_idx,␣
  ↪target_col_idx, num_iterations)

# Calculate the final reconstructed value
final_reconstructed_value = reconstructed_data[target_row_idx, target_col_idx]

# Print the absolute difference between the original and reconstructed values
print(original_value)
print(final_reconstructed_value)
print("Absolute difference between original and reconstructed values:", np.
  ↪abs(original_value - final_reconstructed_value))

# Plot the errors at each iteration
import matplotlib.pyplot as plt
plt.plot(errors)
plt.xlabel('Iteration')
plt.ylabel('Absolute Difference')
plt.title('Absolute Difference vs. Iteration')
plt.show()
```

```
3.2
3.193684639334072
Absolute difference between original and reconstructed values:
0.0063153606659280825
```

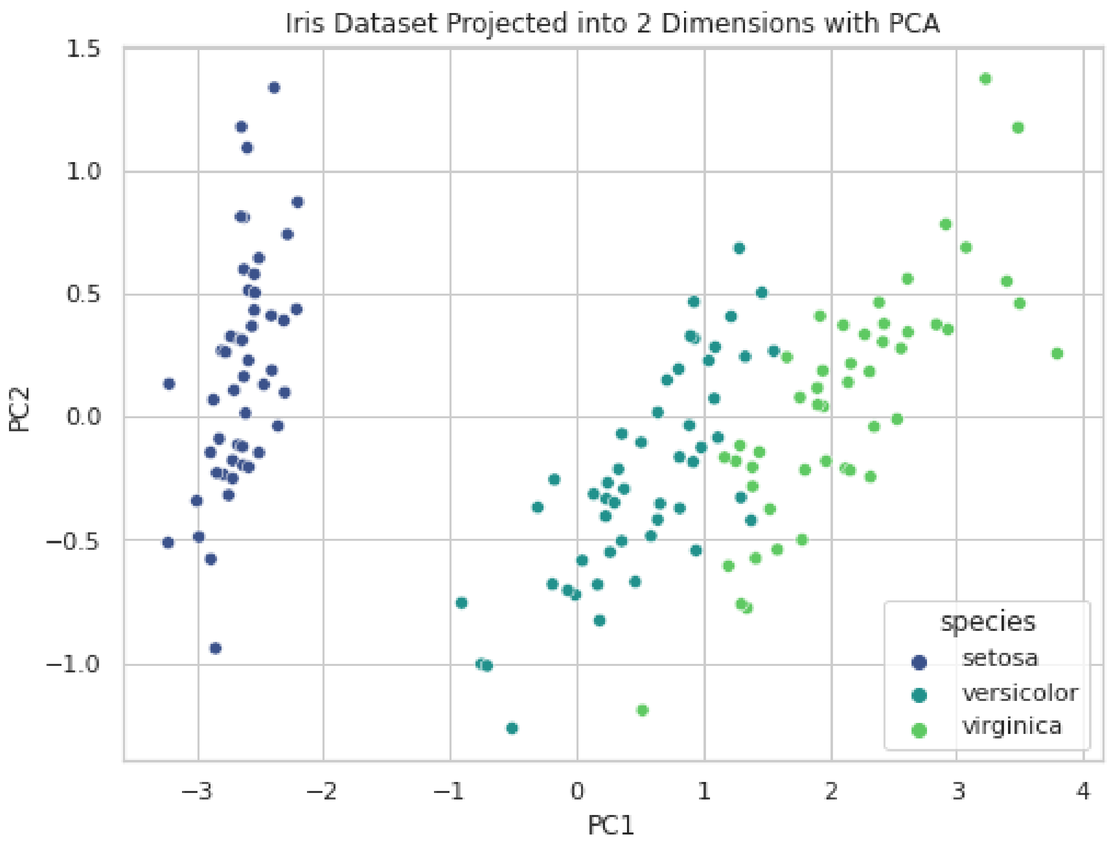


4.3 [3pt] Plot the entire Iris dataset (no split, keep the classes) projected into 2 dimensions using PCA. Use scikit-learn to compute the principal components.

- 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!

[10]:
```python
import seaborn as sns
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.decomposition import PCA

# Load the Iris dataset
iris = load_iris()
X = iris.data
```

```
y = iris.target

# Convert labels to numeric using astype('category').cat.codes
y_numeric = pd.Series(y).astype('category').cat.codes

# Perform PCA with 2 components
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)

# Create a DataFrame with the projected data
pca_df = pd.DataFrame(X_pca, columns=['PC1', 'PC2'])


# Add back the species column for hue
pca_df['species'] = iris.target_names[y]

# Plot the entire Iris dataset projected into 2 dimensions using Seaborn
sns.scatterplot(data=pca_df, x='PC1', y='PC2', hue='species', palette='viridis')
plt.title('Iris Dataset Projected into 2 Dimensions with PCA')
plt.show()
```



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.

Classic recommender systems aim to suggest personalized content based on past user interactions. They often use matrix decomposition to understand user preferences and item characteristics. Denoising comes in when we need to fill in missing values in the interaction matrix to improve recommendations.

In modern systems, we've moved beyond matrix methods to more flexible approaches like neural networks. These systems directly learn from user-item interactions, capturing complex patterns for better recommendations.

A recommender system essentially maps user and item features to predicted preferences. By learning this mapping, it can make accurate predictions even for new users or items. This mapping approach is more adaptable and scalable compared to traditional methods. It can capture non-linear relationships and complex patterns in the data, leading to more accurate and personalized recommendations.

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 MRI scans. *Enhance! (cit.)*

6.0.1 Final considerations

- Supervised Learning implies the presence of a *supervisor* in the data preparation, an expert *oracle* that provides the *correct* labels. This is typically either a human (which means limited data, human errors, time constraints, etc.), or (lately more common) another algorithm that is 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.