École Nationale des Ponts et Chaussées

Landtype identification based on satellite imagery

IMI department project

By

Mathis Wauquiez, Gabriel Mariadass, Vincent Gefflaut

Under supervision of

Konstantin Kuznetsov and Pavel Litvinov from GRASP Earth

Contents

Abstract

The project is supervised by GRASP (Generalized Retrieval of Atmosphere and Surface Properties). This research-based company deploys satellites in orbit to carry out a variety of measurements from space, in particular of variables related to the optical properties of the soil. Its objectives are diverse, ranging from climatology to the study of the evolution of vegetation cover. The company offers its services in particular to space agencies (ESA, NASA, JAXA) and meteorological agencies (EUMETSAT).

This paper shows how to retrieve some land types from optical measurements made from these satellites. Based on a dataset containing a few of these measurements made on a large batch of points at the surface of Earth, and thanks to some Machine Learning techniques, you can find clusters of the same land types.

1 A first view and analysis of the data

1.1 Description of the dataset

We are working with a climatological data file named 'POLDER' with a spatial resolution of 1.1 degrees created by GRASP, which summarizes one year of climatological data previously processed in depth by the source company (Chen et al., 2020; Herrera et al., 2022)

This file comprises 180 (latitude) x 360 (longitude) points (multi-index) (totaling 64800 points), and for each of these points, we have 131 various climatological data corresponding to averages over previous years. Here is a non-exhaustive list - only the first two physical parameters will be used for our clustering (see Section 1.2 Variables selection and visualization):

- **NDVI**: Normalized Difference Vegetation Index.
- **DHR1020 to DHR865**: Bidirectional Reflectance Factor at different wavelengths.
- AAOD1020 to AAOD865: Aerosol Absorption Optical Depth at different wavelengths.
- AExp: Absorption Exponent.
- AOD1020 to AOD865: Aerosol Optical Depth at different wavelengths.
- AODC1020 to AODC865: Corrected Aerosol Optical Depth at different wavelengths.
- AODF1020 to AODF865: Filtered Aerosol Optical Depth at different wavelengths.
- ImagRefIndC1020 to ImagRefIndC865: Imaginary Refractive Index for coastal phase at different wavelengths.
- ImagRefIndF1020 to ImagRefIndF865: Imaginary Refractive Index for oceanic phase at different wavelengths.
- LandBPDFMaignanBreon670: Earth Diffusion Parameter at 670 nm.
- LandPercentage: Percentage of Land Coverage.
- ResidualRelative: Relative Residual.
- SSAF1020 to SSAF865: Filtered Single Scattering Albedo at different wavelengths.
- SizeDistrLogNormBin1 to SizeDistrLogNormBin5: Log-Normal Particle Size Distribution in different bins.
- SphereFraction: Spherical Fraction.

We decided to focus on data that only concerns land, and not oceans because it is not useful to us.

1.2 Variables selection and visualization

We focus on the NDVI and the DHR for different wavelengths because GRASP's experience shows that these are the most representative variables of soil type.

Regarding the plots, in order to better understand the data we are working with and the physical phenomena they describe, we sought to develop an appropriate visualization technique. We have come to the conclusion that it is a good idea to attempt to represent the variables on a satellite view in order to directly perceive the physics they describe.

Here is a visualization of some of these 7 variables on a image satellite background after normalization of the data (see 2.1 About plotting clusters and normalization):

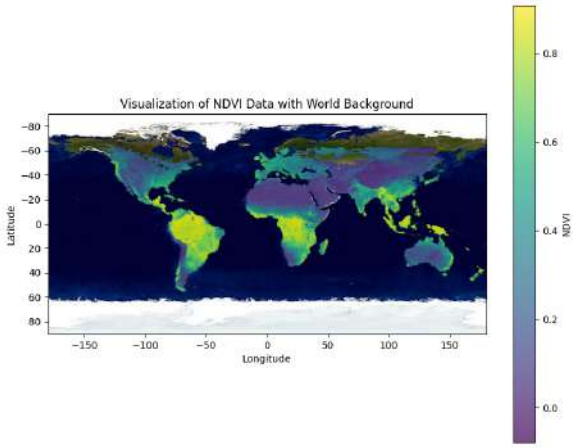


Figure 1: NDVI

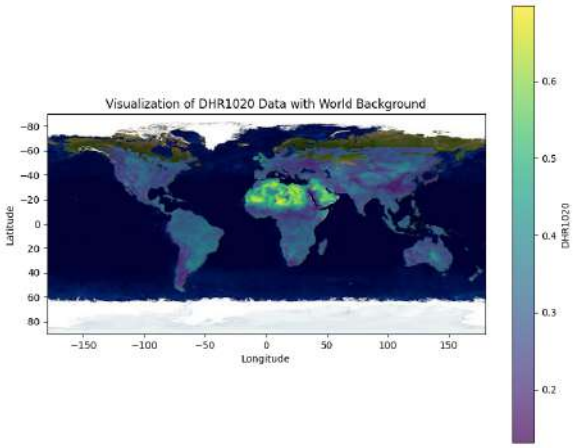


Figure 2: DHR1020

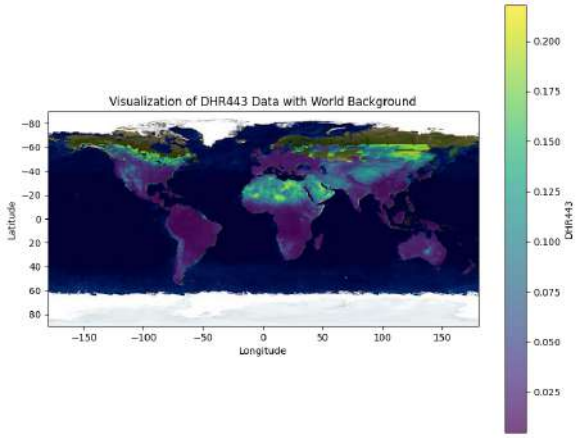


Figure 3: DHR443

We can see that there is missing data for regions located in the north, but we have enough data to perform a proper clustering of the majority of the world's regions.

The NDVI index, which corresponds to a vegetation index, is particularly high in the northern part of South America (Amazon), Central Africa, and Southeast Asia (Indonesia and neighboring countries), and conversely quite low in North Africa and the Middle East regions, which is consistent with our expectations.

Regarding the DHR indices that can provide information about terrain relief and types, in addition to the NDVI index that varies from one region to another around the world as we have seen, reflectance data also vary from one region to another and from one wavelength to another, allowing us to have sufficiently diverse data to classify different areas into distinct groups.

1.3 Correlation between the variables

It's a good practice to study the correlations between our variables to ensure they are all relevant. Here we visualize it on a correlation matrix.

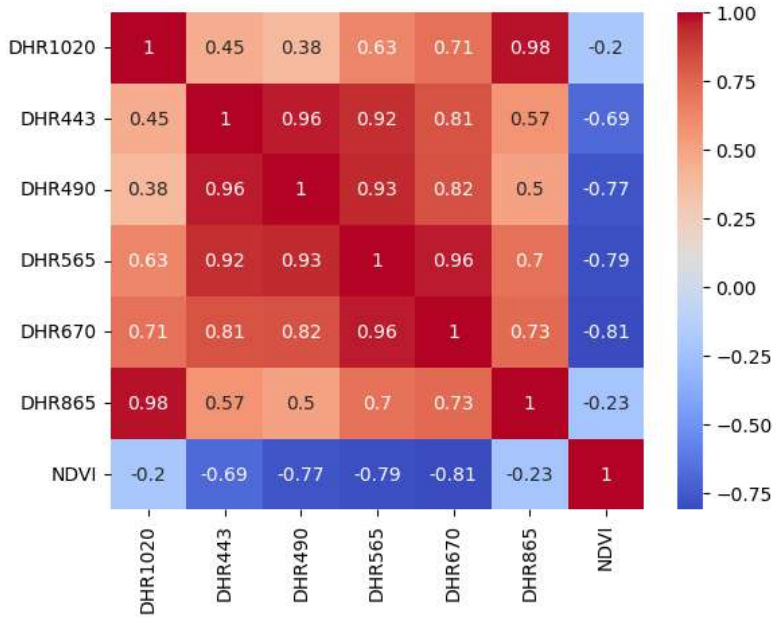


Figure 4: Correlation matrix of the 7 variables of our dataset

Figure 4 is the result of code 5. We observe really high correlations between the couples DHR443-DHR490, DHR565-DHR490, DHR670-DHR665 and DHR865-DHR1020 and it can be explained by the fact that they refer to close wavelengths. It means we could delete 4 out of 7 variables without losing much information.

2 Clustering on non-reduced data

First, we applied clustering algorithms on non-reduced data, that is to say only after the selection of the 7 variables of interest that we chose (see I.2) but without PCA or other dimensionality reduction methods.

2.1 About plotting clusters and normalization

In order to visualize our results, we need to plot our clusters. As for the visualization of variables, in addition to classic plots (i.e. simply drawing on a white background), we sought to represent the clusters on a satellite view in order to have a direct overview of the results that we we obtain and thus be able to evaluate them.

Regarding normalization, we normalized our data before clustering in order to give an equivalent weight to each physical parameter.

2.2 HAC Clustering

Firstly, we sought to implement the unsupervised clustering algorithm called HAC, also known as agglomerative clustering.

The steps of the HAC algorithm are as follows:

1. Initially, each point is considered as an individual cluster.

2. We calculate the similarity between each pair of clusters (The default similarity measure used in scikit-learn for agglomeration is Euclidean distance), then we merge the two most similar clusters. This similarity calculation between each pair is computationally expensive for the computer, making it computationally more challenging than the K-Means algorithm.

3. We repeat step 2 until the specified number of clusters is reached. Although this number of clusters may be known in advance, this is not our case. When this number is not known, another approach is to not specify the expected number of clusters and to perform step 2 until only one cluster is obtained, then to plot the dendrogram to visualize the hierarchical structure of the clusters, and then choose the appropriate number of clusters by cutting the tree at a certain level.

The HAC algorithm has the advantage of being able to inspect the hierarchical structure of clusters to then choose the appropriate number of clusters, which will be useful for us, but it has the drawback of being heavier in computation, which will force us to work on subsets of the dataset.

We plot the dendrogram on our dataset (clustering data normalized contains 10870).

We obtain :

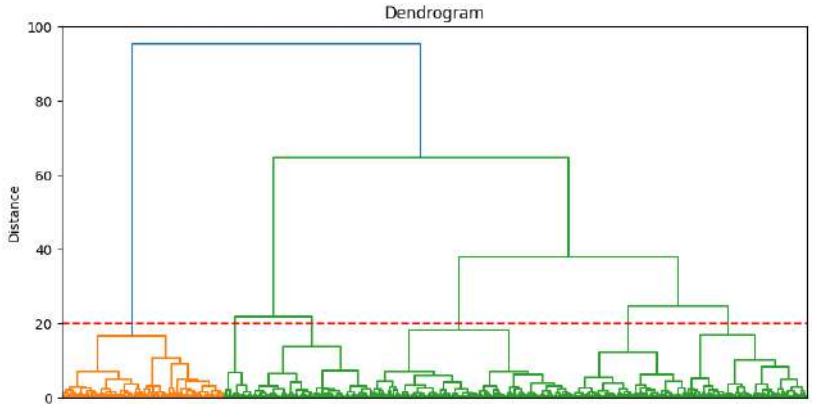


Figure 5: Dendrogram

We can see that 6 clusters seems to be a reasonable number to classify our different terrains. We will therefore use this number of clusters later (num clusters = 6).

We obtain :

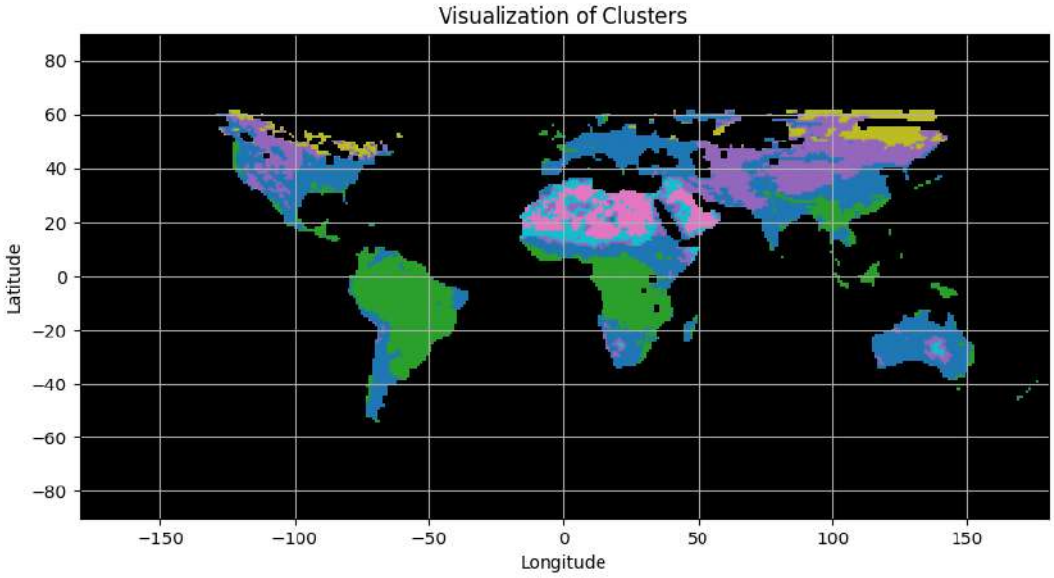


Figure 6: 6-HAC

We can see that the result is relevant and that all the regions of the world retained in the data seem to have been correctly categorized. Indeed, we find a good classification concerning Africa with different groups for the north (rather dry and desert) and the center (with rather strong vegetation similar to that from the north east of South America).

We also find an expected similarity between South America and Africa, which were previously adjacent lands (we notice a continuation of the clusters on these two territories if we "glue" them to each other).

2.3 K-Means Clustering

We then sought to use the K-Means algorithm, which is another clustering method. The idea of the K-Means algorithm is to divide the data into k groups ($k = 6$ in our case, determined in Part II.2 through the HAC algorithm and result analysis) so that each sample belongs to the group whose centroid is closest.

The steps of the K-Means algorithm are as follows:

1. The algorithm starts by choosing k random samples that will serve as initial centroids for the clusters.
2. It then assigns each point to a cluster whose centroid is closest (using Euclidean distance).
3. The algorithm then updates the centroids by taking the mean of each cluster.
4. Steps 2 and 3 are then repeated until no significant changes occur (there is convergence of the cluster centroids that no longer change, or a stopping criterion is reached).

The logic of this algorithm is a priori different from that of HAC clustering, it is a test of another algorithm.

We obtain:

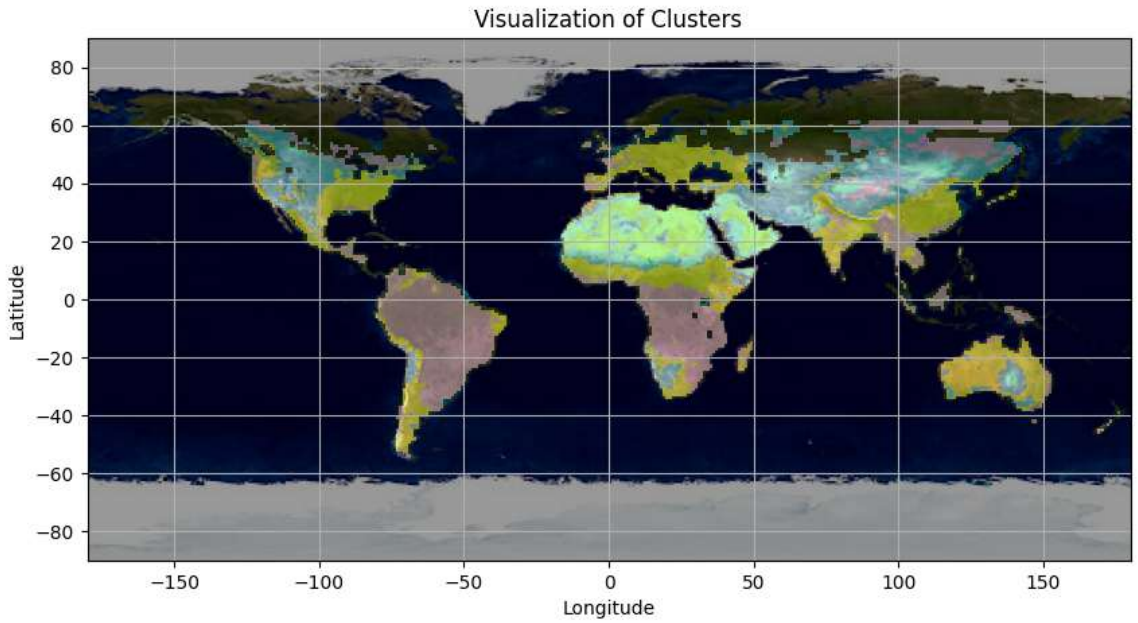


Figure 7: Visualization with world image background

We obtain a result similar to that obtained using the HAC algorithm. This similarity is positive because we find a classification of similar terrain with two different a priori clustering algorithms (see 2.4 HAC - K-means comparison).

As the result is similar to the HAC algorithm, we find the fact that the classification of regions seems relevant and with intuitive results that we find such as that of the classification of African terrains, or even the continuity between south American and African terrains.

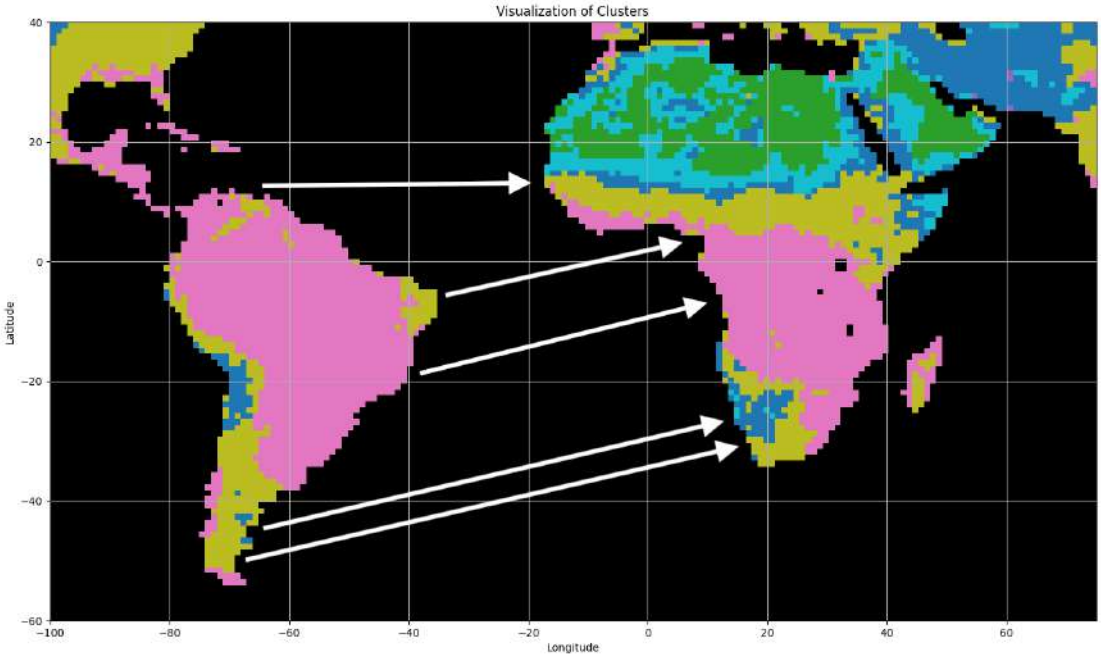
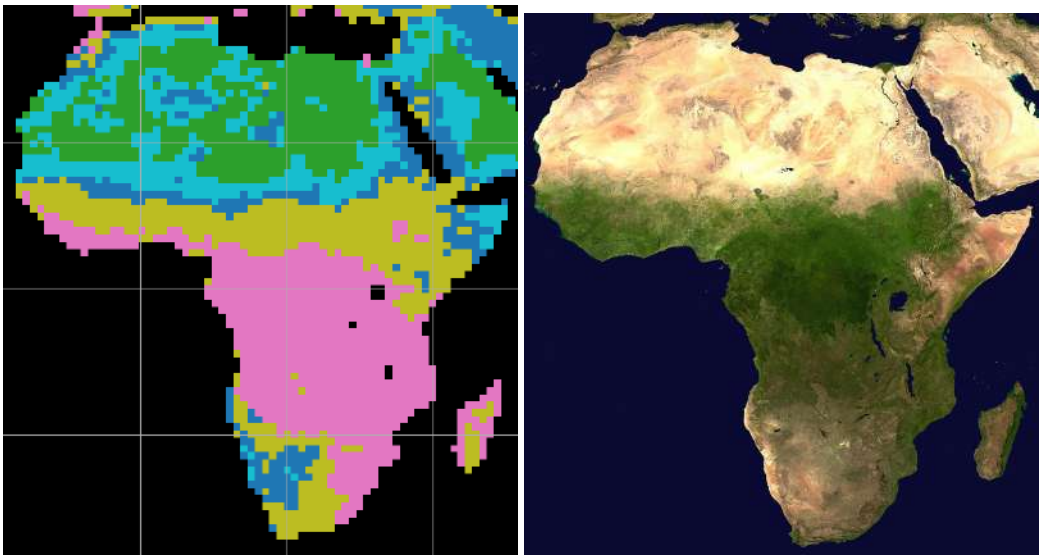


Figure 8: Continuity between south American and African terrains

To go further, we can zoom in on these regions and compare them to a satellite view:



(a) Africa Clusters (b) Africa Image

Figure 9: Africa Clustering

We can see that the clustering has been correctly carried out for the allowed precision (the number of points), and that small areas of water are recognizable for example.

2.4 HAC - K-means comparison

We can further study the similarity between K-Means and HAC algorithms.

We can make a comparison of the silhouette score between the two clusterings. The silhouette score, introduced by Rousseeuw, 1987, ranges from -1 and 1, and is a measure that indicates on average how similar a point is to its cluster compared to other clusters. A score of 1 indicates that the points are well grouped, 0 that there is some overlap of clusters and -1 that the points would on average be better located in other clusters.

This score is determined by calculating two distances: D_1, the average distance between the point and all other points in the same cluster, and D_2, the average distance between the point and all the points in the nearest neighboring cluster (the norm used is the Euclidean norm). The score is then given by $s = \frac{D_2 - D_1}{\max(D_1, D_2)}$.

We obtain : Silhouette Score of K-means = 0.3347414 ; Silhouette Score of HAC = 0.25490317

These scores indicate firstly that the points are rather well placed in each of their clusters. The similarity of these two scores for the two clustering methods also indicates that the results are rather similar and thus makes it possible to add a quantification to the similarity of the results that we had simply observed using the plots. We then see that the k-means algorithm is more efficient than the HAC method used first.

3 Clustering on reduced data

Thus, we saw that the clustering methods that we used in part 2 were effective. In order to be able to process more complex data sets (more points) in order to have more precision, we sought to test these clustering methods on reduced data.

3.1 PCA

3.1.1 Application

As some of our variables are highly correlated it would be interesting to apply methods to reduce their number (7). One of the most common approaches to do so it to apply Principal Component Analysis (PCA) (Wold et al., 1987). It is a linear operation to convert our correlated vectors into the same number of vectors, but uncorrelated and sorted by their explained variance ratio, meaning how much relevant information these new variables contain. At the end we can select a certain proportion of the first components that explain a pretty high proportion of the dataset.

After applying PCA to the data it is necessary to plot the explained variance ratio by component. The code 12 gives the result figure 10.

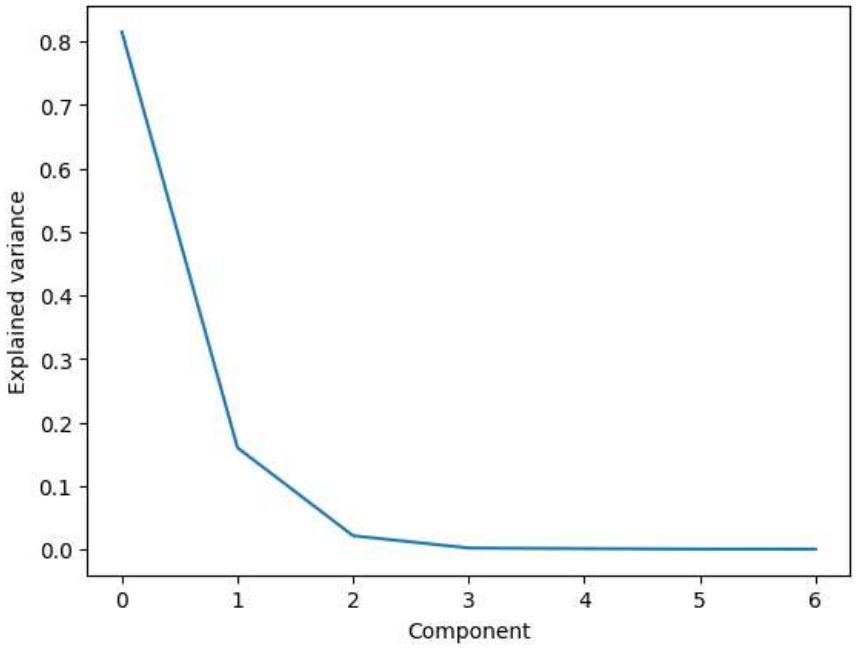


Figure 10: Explained variance ration by component

The majority of the information (>95%) is contained in the three first principal components. This result is coherent with the high correlations between the variables that we found before.

3.1.2 Density of the components

Now let's plot the histogram of the principal components to understand their probability density better.

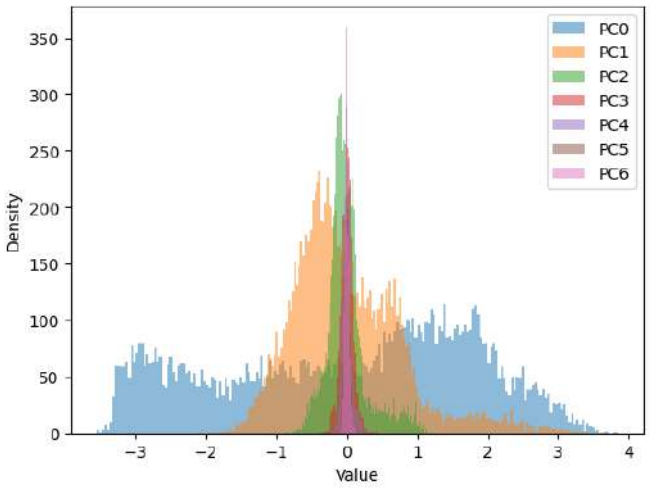


Figure 11: Density of the 7 components of PCA

The distributions of the three first principal components, obtained by code 13 shown on figure 11 are largely spread and contains several peaks and holes showing that they would be useful for clustering. On the opposite, the following components look much more like centered Gaussians with a very small variance and tend towards a Dirac for the last of them.

3.1.3 Scatter plot with density

Another interesting analysis can be made by plotting a principal component as a function of the previous one. And to extract more information we can also include the density with color shades.

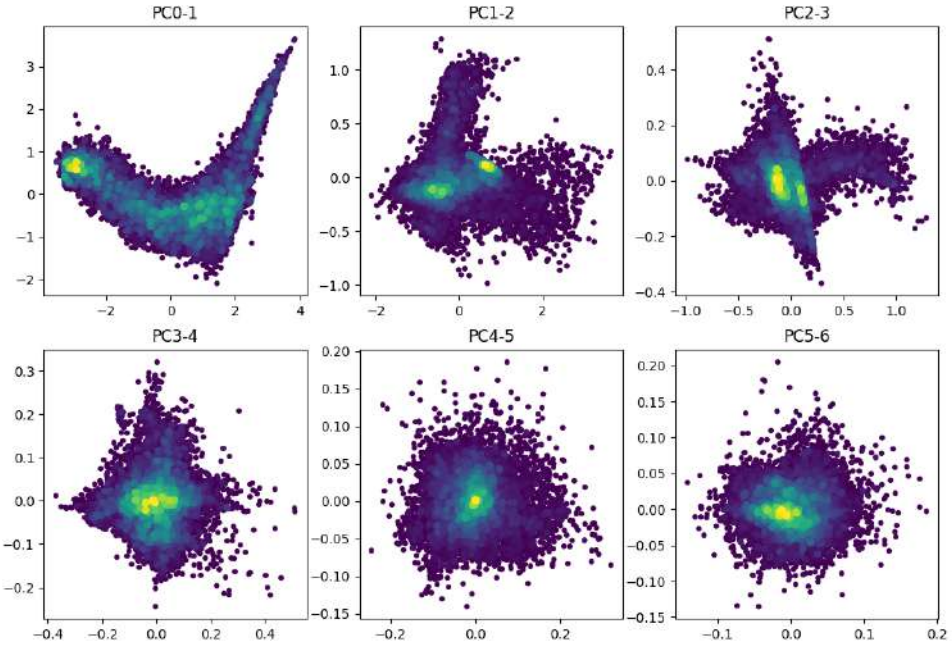


Figure 12: Scatter plots with density

For example, the top left figure is the principal component (PC) 1 as a function of PC 0, and the yellow areas are the most dense ones, and the blue ones, the lowest. As expected only the first three plots give interesting

results, with unique shapes that translate the variety of the data. The more you plot high index PCs, the more they look like 2D gaussians, meaning they won't give much interesting information for clustering.

3.2 Clustering on PCA-reduced data

Now it's time to apply our clustering to dimension-reduced data.

3.2.1 Number of clusters

To determine the optimal number of clusters we plot the Within Cluster Sum of Square (WCSS) evolution with the number of clusters. The optimal number of clusters is determined by the "elbow" of the curve.

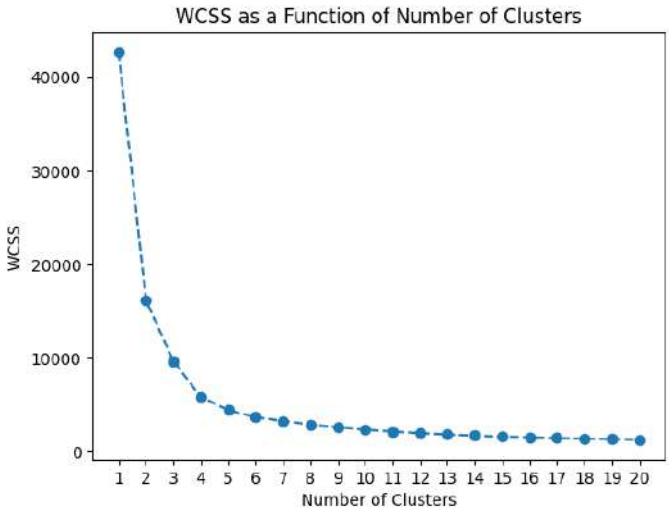


Figure 13: Elbow Curve

On figure 13 we can determine 6 as a reasonable optimal number of clusters (consistent with the dendrogram, see 2.2)

3.2.2 Application

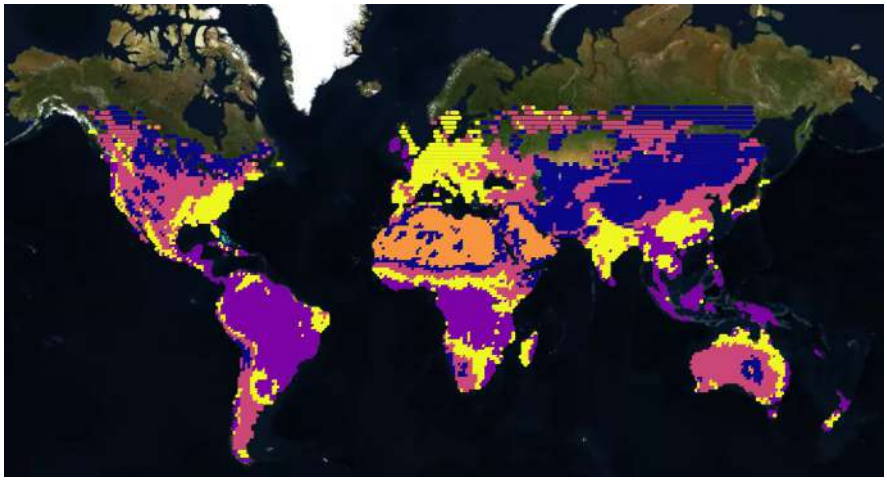


Figure 14: Result of the clustering on the 3 PC

The clustering visible on figure 14 and made by plot 17 is almost the same as the one obtained without PCA (figure 7).

3.2.3 Link with the scatter plots

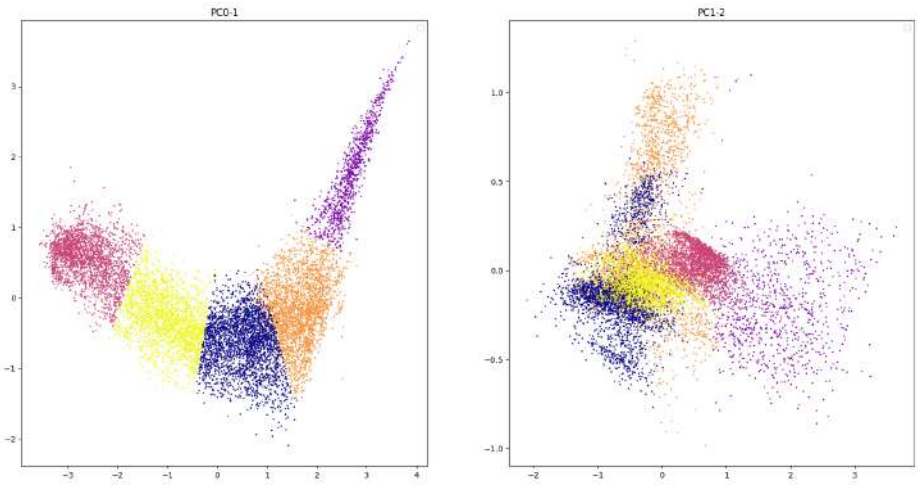


Figure 15: Scatter plots with colored clusters

Figure 15 is the result of code 18. It shows how we can almost discriminate all the clusters with only the two first PC. For the few points that are uncertain we can go further on the next scatter plot.

3.3 Variational auto-encoder

3.3.1 Theory behind a variational auto-encoder

This work is based on the article by Kingma and Welling, 2022, "Auto-Encoding Variational Bayes", and was partially inspired by this article from towardsdatascience : Variational Autoencoders are an improvement of Autoencoders. The article "Image clustering by autoencoders", A S Kovalenko1 , Y M Demyanenko1, gave the idea to use MNIST as an example. Doersch, 2021, gave us the details onto how to implement a variational autoencoder in practice.

Autoencoders are a dimensionality reduction method, whose idea is to use two neural networks, one encoder and a decoder, to learn the best encoding-decoding scheme using an iterative optimization process. Mathematically speaking, we solve the following optimization problem, where L_{rec} is the reconstruction loss between x and d(e(x)) (it can be the L_2 distance for example):

$$\min\left(L_{rec}(x, d(e(x)))\right)$$

$$\text{for } d : R^d \to R^p \text{ and } e : R^p \to R^d$$

If we suppose that both our encoder and decoder are a single linear layer, we can see a clear link between PCA: just like for PCA, we are looking for the best linear subspace to project data on it with as few information loss as possible. Encoding and decoding matrices obtained with PCA define naturally one of the solutions we would be satisfied to reach by gradient descent, but we should outline that this is not the only one. In fact, one could choose different basis for the same optimal subspace, and so several encoder decoder pairs can give the optimal reconstruction error.

If we now assume that both the encoder and the decoder are deep and non-linear, the more complex the architecture is, the more it can proceed to a high dimensionality reduction. With a “deep enough” network, one could in theory reduce the number of dimensions to 1, without loss. But this should come as a price: without any reconstruction loss, we lack interpretability and structure in the latent space (i.e. the output space of the decoder). Second, we should not forget that the final purpose of dimensionality reduction is not to only reduce the number of dimensions of the data, but rather to reduce this number of dimensions while keeping a major part of the data structure information in the reduced representations, to latter use it for clustering purposes. Moreover, since our data is noisy (around 5% error), if our autoencoder is overfitting our dataset,

small perturbations due to noise, or just small variations, will give totally different latent representations. We conclude that our latent space needs two properties: continuity (two close points in the latent space should not give two completely different contents once decoded), and completeness (a point sampled from the latent space should give "meaningful" content once decoded, i.e. content that makes sense according to the distribution of data we dispose). These issues are tackled using a variational autoencoder: it is an autoencoder whose training is regularized to avoid overfitting and ensure the continuity and completeness properties of the latent space. This is done by adding one step to the encoding-decoding process: instead of encoding a sample to a single point, with encode it to a distribution; we then sample a point from this distribution and try to reconstruct as best as possible the original sample. To sum up the process:

$$x \xrightarrow{\textbf{encoding}} p(x|z) \xrightarrow{\textbf{sampling}} z \xrightarrow{\textbf{decoding}} \hat{x}$$

It is common practice to choose to encode the distributions as normal distributions, whose mean and variance are determined by the encoder. To keep the reconstruction error low, the encoder will try to separate as best as possible the encoded distributions with respect to the input data, so that the decoder can then decode the samples from the distributions with a minimal reconstruction loss. In theory, without any regularization loss on the encoded distributions, a variational autoencoder could encode 0-variance distributions, and would behave exactly like a classic autoencoder. So, to avoid these effects, we must regularize both the covariance matrix and the mean of the distributions returned by the encoder. This regularization is done by enforcing distributions to be close to a standard normal distribution: we do so by adding a divergence term, the Kulback-Leibler divergence between the returned distribution and a standard Gaussian. This divergence term encourages as much as possible returned distributions to "overlap", satisfying this way the expected continuity and completeness conditions. This comes at the cost of a higher reconstruction loss on the training data. There then is a tradeoff between the reconstruction loss and the divergence loss, that we tackled by assigning a different weight to each loss. The loss minimized during the optimization process is then : $L_{rec}(x, \hat{x}) + \alpha * L_{div}(\mu, \text{logvar})$, with α controlling the tradeoff between the reconstruction loss and the divergence loss.

The architecture of the encoder and the decoder are simple multilayer perceptrons, with ReLU non-linearities. We choose a high number of hidden dimensions regarding the input dimension: 38 (it was found by trial and error). There is a small trick to calculate the gradient backpropagation: if we sample a point from the distribution from the latent space, then we can't calculate the gradient descend before this sampling. This is tackled using the reparameterization trick: rather than saying that $z \sim \mathcal{N}(\mu(x), \text{var}(x))$, we say that $z = \text{var}(x) * \zeta + \mu(x)$, allowing us to propagate the gradient backward for each sample from the distribution.

Before training our model on our data, we tried to run it on the MNIST dataset. It allowed us to understand the complex behavior of the model, to understand the latent space, to make sure our model works well, to see the denoising effect of the model, and to explore image generation.

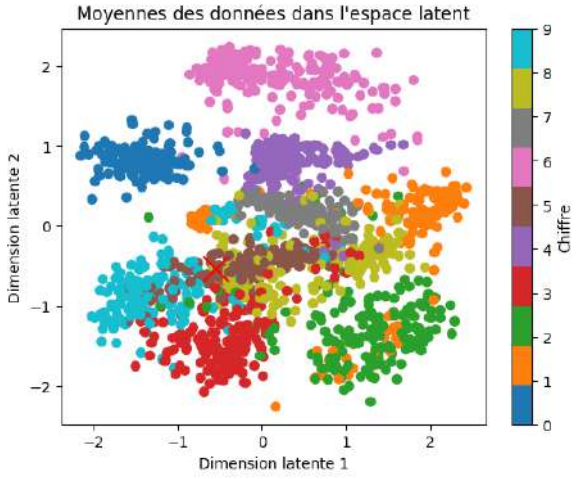


Figure 16: Latent space projection of the MNIST dataset

We created this figure above, for different values of α. It is the mean's coordinates calculated for each sample of the dataset, colored by class (0-9). The more we increased the value of α, the more the clusters

tended to overlap, and the lower was our R^2. Here, the plot shows a high value of α, leading to some overlaps, but that makes sense: indeed, twos for example overlap with ones. Or 9 with 5: this is because those digits are really similar. We compared this dimensionality reduction with PCA.

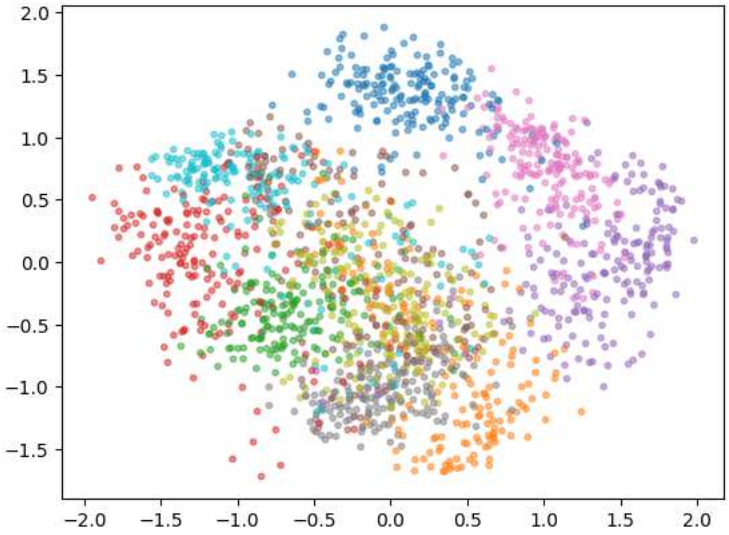


Figure 17: Principal components of the MNIST dataset

PCA seems much less interpretable than a variational autoencoder, and using the latent space of a variational autoencoder seems like a better idea for clustering than using PCA. The red cross in the figure 16 corresponds to a point we choose in the latent space, non-existent in the original dataset. We then tried to decode it, and obtained the result in the below figure (18):

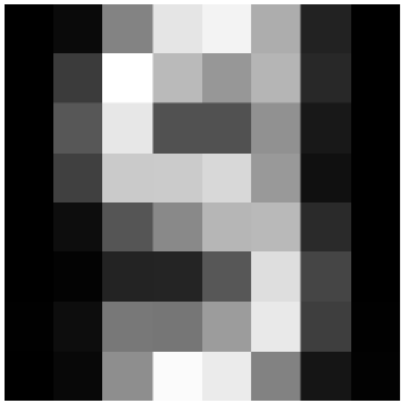


Figure 18: Number generation

This digit is like both a 9 and a 5! And it makes sense, since it is in an area of the latent space where the two classes are projected. It indicates that our latent space has a good amount of information.

3.3.2 Model training

The next step was to train our model on our dataset. During the training process, we calculated and kept interesting statistics: the losses during the training, the R^2 of the model, and the histograms of the log-variance and the mean. We then compared the statistics with low values of α to those with a high value.

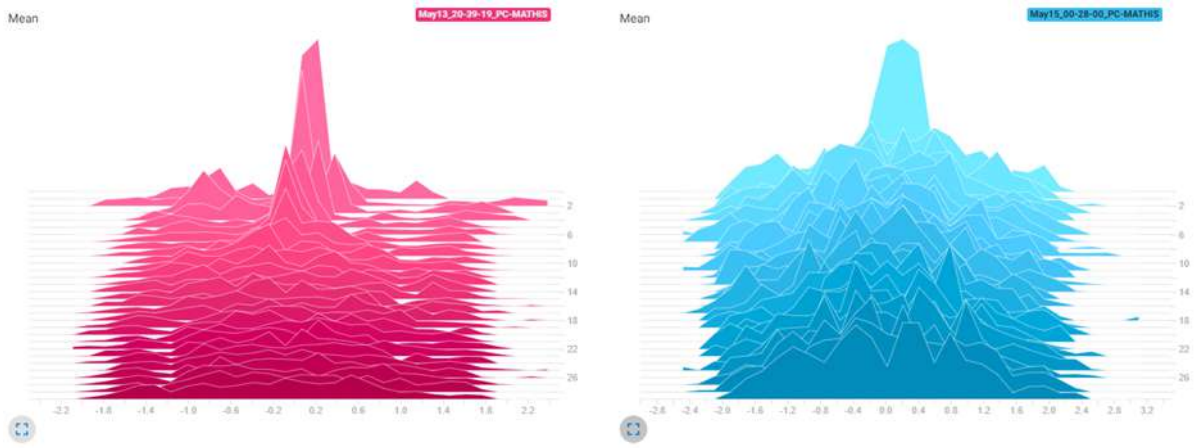


Figure 19: log-variances
Left: high alpha; right: low alpha

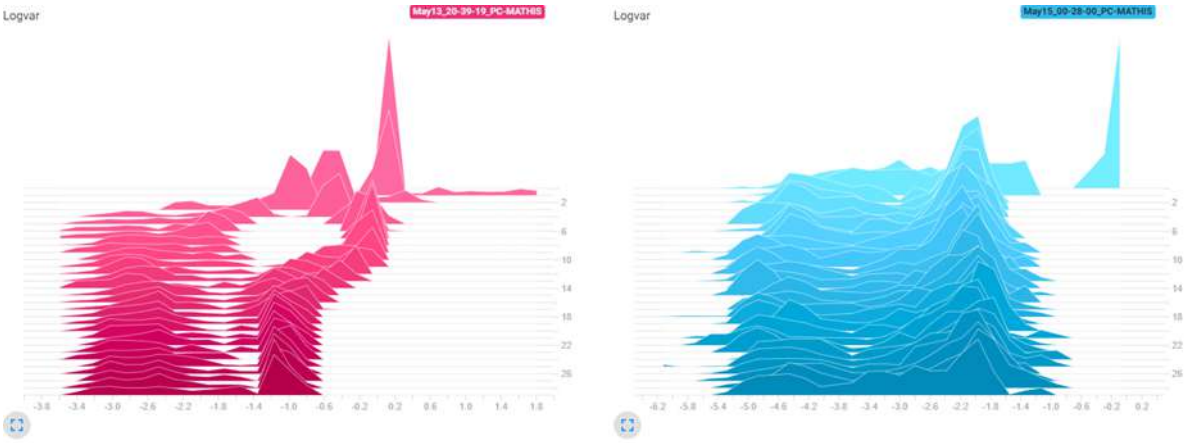


Figure 20: μ distributions
Left: high alpha; right: low alpha

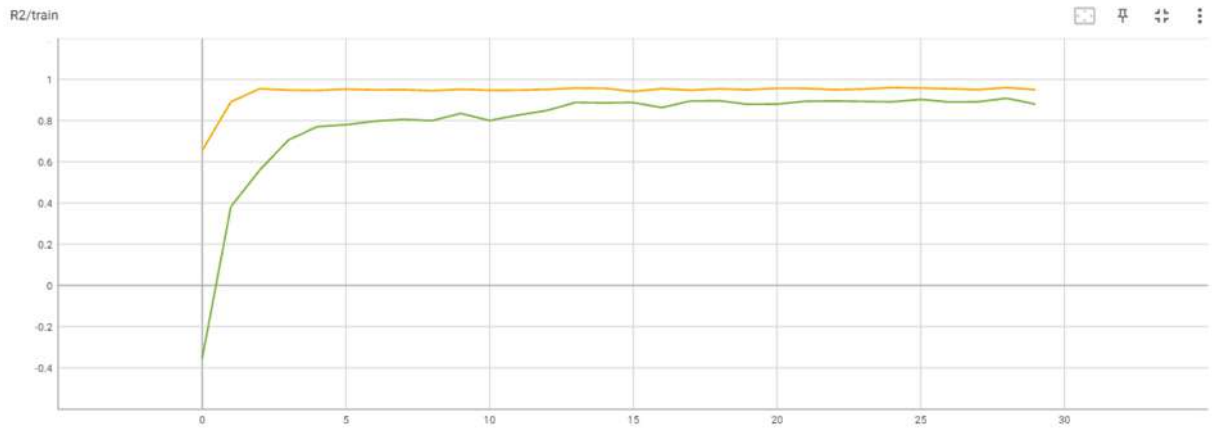


Figure 21: R^2 during training
Green: high alpha; orange: low alpha

We indeed see that when using a high value of α, the log-variance is greater than we using a low value, and the mean is closer to 0. We see as well that after 1 epoch, the mean is near 0, and the log-variance near to 0 as well, meaning that in the first steps of the training, minimizing the divergence loss is easier than minimizing the reconstruction loss. We also see that higher values of α leads to lower R^2.

We now focus onto analyzing the regularity of the latent space. There is no absolute measure of regularity, but we expect it to have some good properties. One of them is the conservation of distances: two close points in the latent space should not give completely different content once decoded. This gives us two properties: the first is that the norm of the decoded points should not vary fast, because of the continuity property of the latent space. For this purpose, we calculated the gradient of $\|\hat{x}\|$ with respect to z, and took its magnitude 22. We observe a reasonable magnitude, meaning our decoder did not overfit.

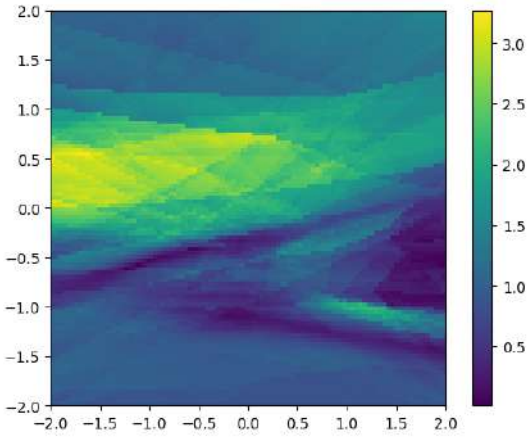


Figure 22: Magnitude of the gradient of the norm

The second fact we expect is for the distance between $\hat{x}_1$ and $\hat{x}_2$ to be increasing with the distance between z_1 and z_2: points close to each other in the latent space should be close to each other in the decoded space. If this is true for z_1 and z_2 far from each other, it ensures the regularity of our latent space.

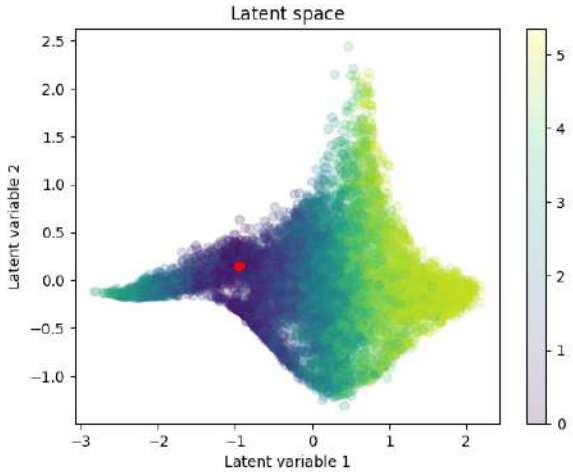


Figure 23: Distances in the decoded space from the red sample

3.3.3 Clustering on the latent space

We are now sure that our latent space is more informative and less sensitive to noise than the original space. It makes it particularly suited for enhancing clustering performances, which we will now focus on.

Before clustering, lets plot the distribution density of $\mu(x)$, and lets compare it to the density of the PCA:

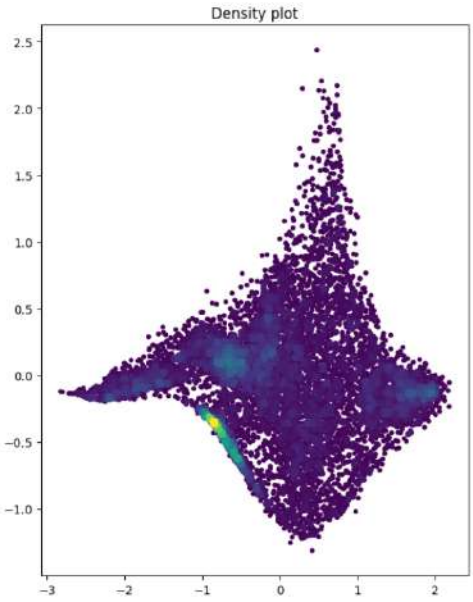


Figure 24: VAE density plot

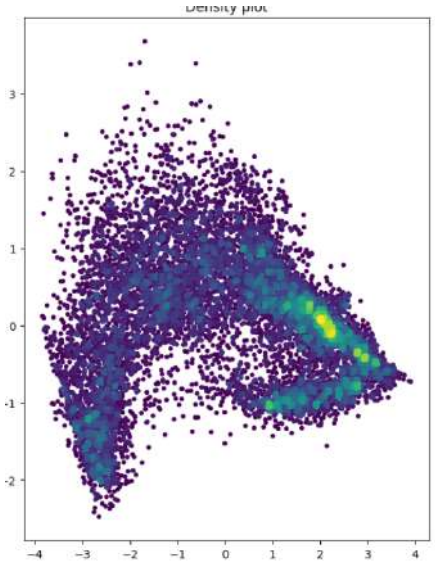


Figure 25: PCA density plot

Figure 26: Density plots

After this, we plot the elbow curve, that indicates us that the optimal number of clusters for k-means clustering is seven. This is the number used throughout the following section.

The results for the clustering are summarized in the following tables, and are plotted in the figure **??**:

Method	Silhouette score
Original data	0.381912
PCA	0.266512
Variational autoencoder	0.403451

Table 1: Silhouette scores

Method	Davies-Bouldin
Original data	1.108778
PCA	0.807471
Variational autoencoder	0.749098

Table 2: Davies-Bouldin scores

3.3.4 1D - Variational Autoencoder

The next step was to try to train a variatonal autoencoder with a latent space of dimension 1. This assumes that the terrain properties (albedo) can be generated from only one variable. Before delving into the maths and the optimization process, lets think a bit about what we will do: is it possible to find such a mapping? Multiple type of lands already exists: we can cite snow, barren land, deciduous forests, evergreen forests, shrub, grassland, pasture, cultivated crops, woody wetlands, herbaceous wetlands, urbanized areas, ... Except for the lands affected by human activity, it seems like all land types (and so all albedos) can be determined from two variables: the average temperature, and the average humidity. Using variational autoencoders will show us if there exists a single variable that can determine pretty accurately the values of the albedo. We will then be able to do the clustering on this variable only, giving us a more interpretable clustering, based on segments.

With the same balance between the reconstruction and divergence losses as before, we obtained $R^2 = 0.8898$, with a much slower convergence. The distribution of the encoded means is in the below figure 27

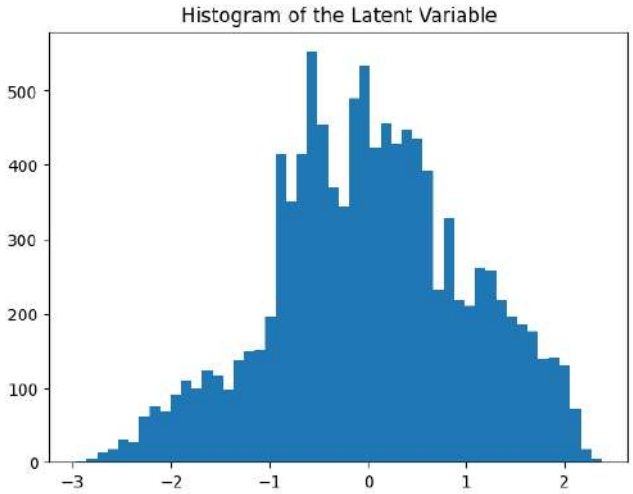


Figure 27: Histogram

The means encoded by the encoder are plotted on the below figure 28.

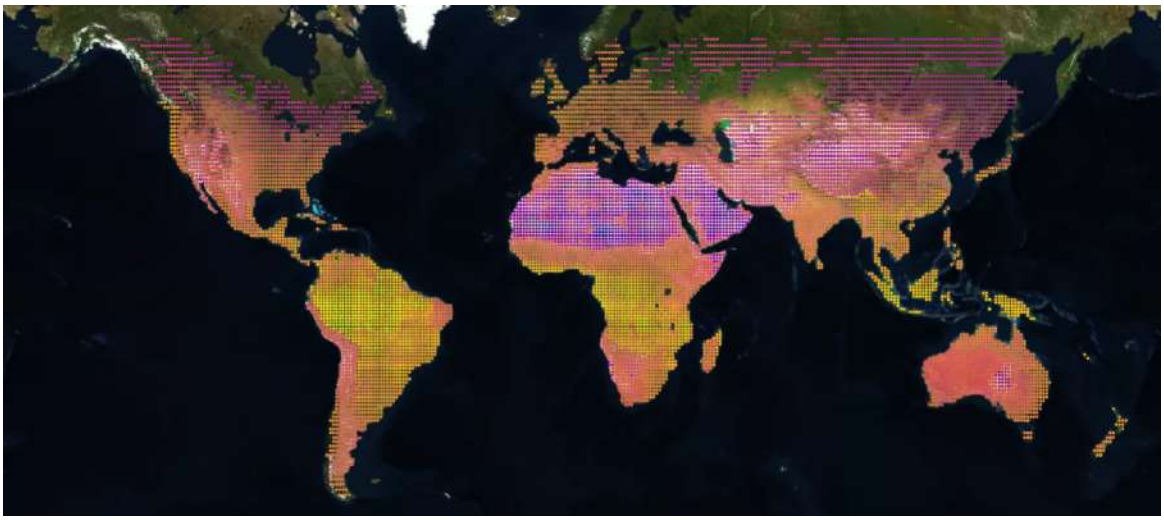


Figure 28: Latent representation

The clusters determined from this representation are shown in the figure 29. The advantage of a 1D latent space is that it is easy to label the clusters: we can label them according to their center. This allowed us to define a measure of similarity between clusters, and thus to add more clusters, while keeping their information.

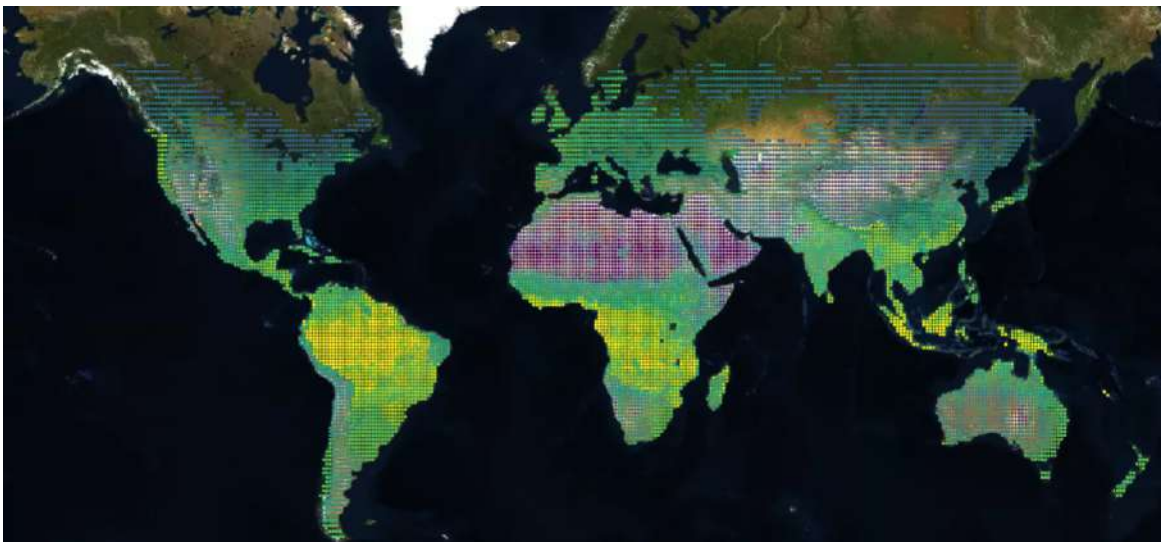


Figure 29: 1D clustering

4 Conclusion

Initially, clustering methods on the unreduced data gave us fairly satisfactory initial results.

Hoping to work with more comprehensive datasets (more data points) for better results, we then turned to data reduction. We used PCA as a first approach to reduce our 7 dimensions into 3 while keeping the essential of the information. Indeed only 3 components explained more than 95% of the variance, which allows us to lighten our data. Then the elbow method ensured us that we can maintain the same number of clusters (6) obtained by KMeans. Finally, the KMeans clustering gives results that are visually comparable to the one obtained with the full-dimension data.

However, PCA has limitations in interpretability and latent space structure, impacting the clustering performances. To address these, we implemented a Variational Autoencoder (VAE), which regularizes the latent space to ensure continuity and completeness. Our experiments with the MNIST dataset demonstrated that VAEs provide clearer and more meaningful clustering than PCA, with distinct clusters and effective new sample generation.

The VAE's latent space proved more informative and less sensitive to noise, enhancing clustering performance. Density plots and silhouette scores showed the VAE outperformed PCA in clustering tasks. Additionally, a 1D VAE demonstrated high R^2 values and meaningful clustering, capturing complex data structures with a single latent dimension.

Another data reduction method is Tensor Train Decomposition (TTD). It could have been an effective method to decompose our data tensor of size (N latitude, N longitude, N physical parameters) into a series of smaller-sized tensors. It was a key element in our initial goals; however, we first noticed that working on unreduced data yielded quite good results, and later experimented with methods using PCA or a VAE which gave us satisfactory results as well. We spent some time trying to develop tools using TTD, but it did not yield any conclusive results that could be presented in this document.

We could also have explored deep clustering: it is a novel field, based on a differentiable "soft clustering" layer. Vardakas et al., 2024, proposes the Deep Clustering using Soft Silhouette (DCSS) method. We can also talk about Guo et al., 2017, who propose a deep clustering method with local structure preservation.

5 Annex

5.1 Figures - Clusters over the world

5.1.1 Variational Autoencoder Clustering Results - 2 dimensions

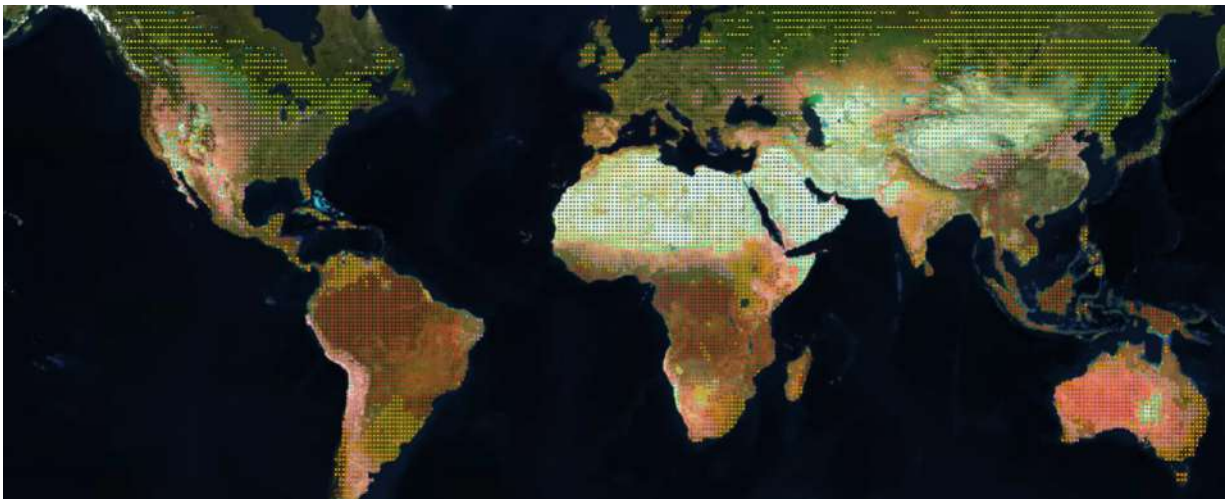
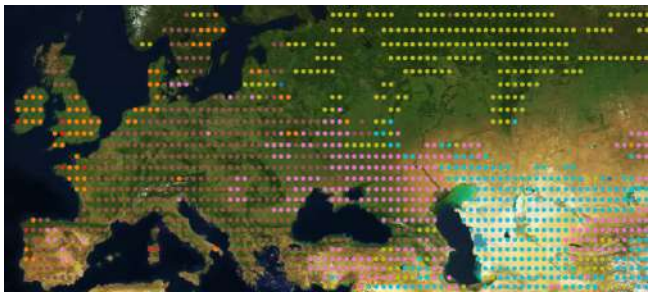
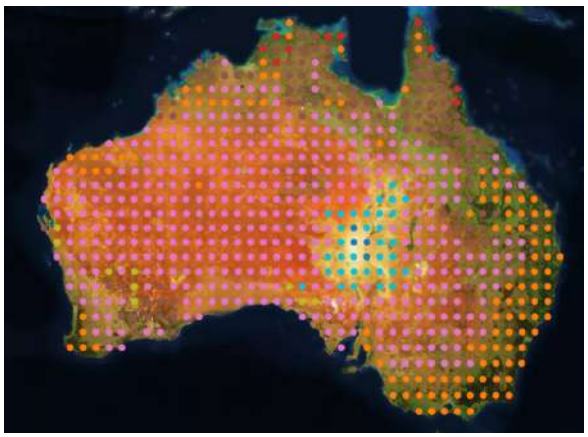
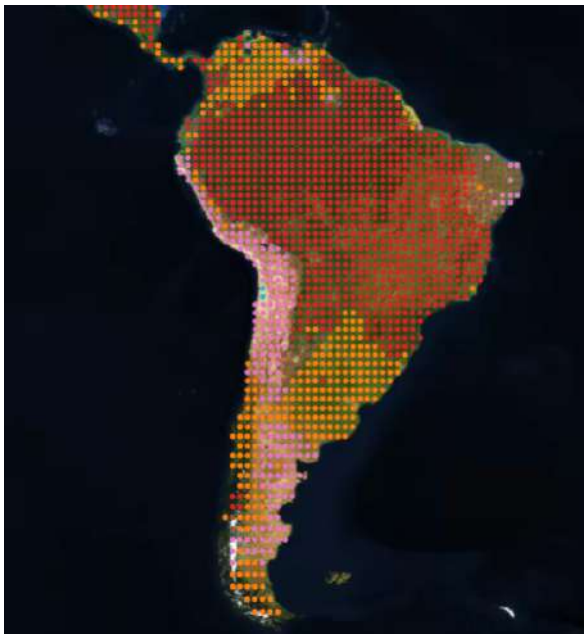
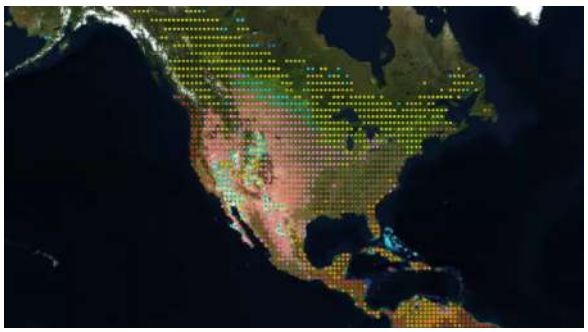


Figure 30: Clusters over the world









5.2 Code

5.2.1 1.1

We had two methods to import the data, the first method uses the xarray library:

```
import xarray as xr
import pandas as pd

dataset = xr.open_dataset('GRASP_POLDER_L3_climatological_S1.1degree.nc')
data = dataset.to_dataframe()
dataset.close()
data = data[data['LandPercentage'] == 100]
clustering_data = data[clustering_variables].dropna()
```

Listing 1: Importation method with xarray

The second method uses the netCDF4 library:

```
def get_data(selection, clustering_variables, filename='GRASP_POLDER_L3_climatological_S1.1degree.nc', fill_value=-1000):
    original_dataset = nc.Dataset(filename)
    img_shape = original_dataset.variables['DHR1020'][:].shape

    # mask creation, to select the land
    if selection == 'sea':
        mask = (original_dataset.variables['LandPercentage'][:] == 100).filled(True)
    elif selection == 'land':
        mask = (original_dataset.variables['LandPercentage'][:] != 100).filled(True)
    else:
        mask = original_dataset.variables['DHR1020'][:].mask

    # flattening the variables / masking
    data = {}
    for var in original_dataset.variables:
        if original_dataset[var][:].shape == img_shape:
            data[var] = original_dataset[var][:].flatten()
            data[var][mask.flatten()] = fill_value

    # creating the dataframe
    df = pd.DataFrame(data)
    df = df.replace(fill_value, np.nan)
    df = df[clustering_variables + ['Latitude', 'Longitude']]
    df.dropna(inplace=True)
    data = df[clustering_variables].values
    data = normalize(data)
    return data, df, mask, img_shape

# Load the data
file_path = 'GRASP_POLDER_L3_climatological_S1.1degree.nc'
data, df, mask, img_shape = get_data('land', clustering_variables, file_path, fill_value=-1000)
dataset = torch.utils.data.DataLoader(data, batch_size=BATCH_SIZE, shuffle=True)
```

Listing 2: Importation method with netCDF4

For the normalize function, see section 5.2.4. Concerning the codes in this document, without particular specification, the code is compatible with data imported with netCDF4, otherwise with xarray.

5.2.2 1.2

```
DHR_variables = ['DHR1020', 'DHR443', 'DHR490', 'DHR565', 'DHR670', 'DHR865']
clustering_variables = DHR_variables + ['NDVI']
```

Listing 3: Clustering variables

```
# This function is compatible with data imported using xarray

def visualize_variable(variable, world_image_Yinverted, data):
    world_image = plt.imread(world_image_Yinverted)
    im = data[variable].to_numpy()
    plt.figure(figsize=(10, 8))
    plt.imshow(world_image, extent=[-180, 180, -90, 90])
    plt.scatter(data.index.get_level_values(1), data.index.get_level_values(0), c=im,
    cmap='viridis', alpha=0.7, s=1)
    plt.gca().invert_yaxis()
    plt.colorbar(label=variable)
    plt.title(f'Visualization of {variable} Data with World Background')
    plt.xlabel('Longitude')
    plt.ylabel('Latitude')
    plt.grid(False)
    plt.show()
```

Listing 4: Variable plot functions

5.2.3 1.3

```
import seaborn as sns

correlation_matrix = pd.DataFrame(data, columns=['DHR1020', 'DHR443', 'DHR490', 'DHR565',
    'DHR670', 'DHR865', 'NDVI']).corr()

sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm')
plt.show()
```

Listing 5: Correlation matrix Python

5.2.4 2.1

```
# Without satellite image background
# This function is compatible with data imported using xarray

def visualize_clusters(data_with_clusters):
    im = np.full((180, 360, 3), np.nan)
    for index, row in data_with_clusters.iterrows():
        if not np.isnan(row['Cluster']):
            cluster_color = colors[int(row['Cluster'])][:3]
            im[int(index[0] + 90), int(index[1] + 180)] = cluster_color
    plt.figure(figsize=(10, 8))
    plt.imshow(im, extent=[-180, 180, -90, 90])
    plt.title('Visualization of Clusters')
    plt.xlabel('Longitude')
    plt.ylabel('Latitude')
    plt.grid(True)
    plt.show()

# Without satellite image background, but with the possibility to select a specified
    region
# This function is compatible with data imported using xarray

def region_visualize_clusters(data_with_clusters, min_lat, max_lat, min_lon, max_lon):
    im = np.full((max_lat - min_lat + 1, max_lon - min_lon + 1, 3), np.nan)
    for index, row in data_with_clusters.iterrows():
        if not np.isnan(row['Cluster']):
            lat, lon = index
```

```
            lat = -lat
            if min_lat <= lat <= max_lat and min_lon <= lon <= max_lon:
                cluster_color = colors[int(row['Cluster'])][:3]
                im[int(max_lat - lat), int(lon - min_lon)] = cluster_color
    plt.figure(figsize=((max_lon - min_lon + 1) / 10, (max_lat - min_lat + 1) / 10))
    plt.imshow(im, extent=[min_lon, max_lon, min_lat, max_lat])
    plt.title('Visualization of Clusters')
    plt.xlabel('Longitude')
    plt.ylabel('Latitude')
    plt.grid(True)
    plt.show()

# With satellite image background - This V1 follows the logic of the first function by overlaying transparent colored points on an image
# This function is compatible with data imported using xarray

def visualize_clusters_map(data_with_clusters, world_image):
    world_image = plt.imread(world_image)
    im = np.full((180, 360, 3), np.nan)
    for index, row in data.iterrows():
        if not np.isnan(row['Cluster']):
            cluster_color = colors[int(row['Cluster'])][:3]
            cluster_color = [min(1, c * 1.5) for c in cluster_color] # Increase color intensity
            im[int(index[0] + 90), int(index[1] + 180)] = cluster_color
    plt.figure(figsize=(10, 8))
    plt.imshow(world_image, extent=[-180, 180, -90, 90])
    plt.imshow(im, extent=[-180, 180, -90, 90], alpha=0.5) # Transparency coefficient
    plt.title('Visualization of Clusters')
    plt.xlabel('Longitude')
    plt.ylabel('Latitude')
    plt.grid(True)
    plt.show()

# With satellite image background - A more sophisticated V2 developed later in the project allowing to obtain an html file where we can zoom in on the map, thus providing a detailed view of each region and the superimposed colored points associated with each sample of the dataset assigned to a clustering group

def plot_variable(df, variable, colormap = cm.viridis):
    cmap = lambda x : f'rgba{tuple(list((np.array(colormap(x)) * 255)[:3].astype(int)) + [128])}' # function to convert the values to corresponding rgba code
    mask = np.logical_not(np.isnan(df[variable])) # we only want to plot the non nan values
    norm = plt.Normalize(min(df[variable][mask]), max(df[variable][mask])) # we normalize the values to get the colors
    normalized_data_values = norm(df[variable][mask])
    viridis_colors = [cmap(x) for x in normalized_data_values] # we get the colors
    trace = go.Scattermapbox(   # we create the scattermapbox plot
        lat=df['Latitude'][mask],
        lon=df['Longitude'][mask],
        mode='markers',
        marker=go.scattermapbox.Marker(
            size=10,
            color=viridis_colors
        ),
        text=df[variable][mask].map("Value : {:.3f}".format)
    )
    fig = go.Figure(data=trace) # we create the figure
    fig.update_layout( # we update the layout
        autosize=True,
        hovermode='closest',
        mapbox=dict(
            center=dict(
                lat=0,
                lon=0
```

```
            ),
            accesstoken=mapbox_access_token,
            bearing=0,
            pitch=0,
            zoom=1,
            style="satellite"
        ),
        width = 1500,
        height = 750
    )
    fig.show()

def plot_clusters(df, variable, cluster_centers, colormap = cm.viridis):
    labels = df[variable]
    df[variable] = reorder_clusters(cluster_centers, labels)
    plot_variable(df, variable, colormap)
```

Listing 6: Clusters plot functions

5.2.5 2.2

```
# For data imported with xarray

from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
clustering_data_normalized = scaler.fit_transform(clustering_data)

# Compatible with the logic of the get_data function (1.1) which uses data imported with
    netCDF4

def normalize(data):
  return (data - data.mean()) / data.std()
```

Listing 7: Normalization

5.2.6 2.3

```
# This code is compatible with data imported using xarray
from sklearn.cluster import AgglomerativeClustering
from scipy.cluster.hierarchy import dendrogram, linkage

# HAC clustering
hac = AgglomerativeClustering(n_clusters=1)
subset_labels = hac.fit_predict(clustering_data_normalized)

# Adding the results to our data table
data['Cluster'] = np.nan
data.loc[clustering_data.index[:len(clustering_data_normalized)], 'Cluster'] =
    subset_labels

# Calculating the linkage matrix and plotting the dendrogram
linkage_matrix = linkage(clustering_data_normalized, method='ward')
plt.figure(figsize=(10, 5))
dendrogram(linkage_matrix, truncate_mode='lastp')
plt.title('Dendrogram')
plt.ylabel('Distance')
plt.show()
```

Listing 8: HAC and dendrogram

```
# This code is compatible with data imported using xarray
hac = AgglomerativeClustering(n_clusters=num_clusters)
```

```
labels = hac.fit_predict(clustering_data_normalized)
data['Cluster'] = np.nan
data.loc[clustering_data.index[:len(subset_data)], 'Cluster'] = labels

# Assigning a color to each cluster
colors = plt.cm.tab10(np.linspace(0, 1, num_clusters))
```

Listing 9: 6 - HAC clustering

5.2.7 2.4

```
# This code is compatible with data imported using xarray
from sklearn.cluster import KMeans

# Same logic as for the HAC
kmeans = KMeans(n_clusters=num_clusters, random_state=42)
kmeans.fit(clustering_data_normalized)
data['Cluster'] = np.nan
data.loc[clustering_data.index, 'Cluster'] = kmeans.labels_

colors = plt.cm.tab10(np.linspace(0, 1, num_clusters))
```

Listing 10: 10 - K-Means clustering

In the case of scikit-learn's KMeans, this algorithm uses as a stopping criterion the fact that the sum of the squares of the centriole variations between two iterations is less than 1e-4.

5.2.8 2.5

```
# This code is compatible with data imported using xarray
from sklearn.metrics import silhouette_score

# K-means
kmeans = KMeans(n_clusters=num_clusters, random_state=42)
kmeans_labels = kmeans.fit_predict(clustering_data_normalized)
kmeans_silhouette_score = silhouette_score(clustering_data_normalized, kmeans_labels)

# HAC
hac = AgglomerativeClustering(n_clusters=num_clusters)
hac_labels = hac.fit_predict(clustering_data_normalized)
hac_silhouette_score = silhouette_score(clustering_data_normalized, hac_labels)

print("Silhouette Score (K-means):", kmeans_silhouette_score)
print("Silhouette Score (HAC):", hac_silhouette_score)
```

Listing 11: Silhouette score

5.2.9 3.1

```
from sklearn.decomposition import PCA

n_components = 7

pca = PCA(n_components=n_components)

# Create a color map for the different numbers
cmap = plt.get_cmap('tab10')

# use the same data as before
pca_data = pca.fit_transform(data)
```

```

# Inverse transform
pca_data = pca.inverse_transform(pca_data)

MSE = F.mse_loss(torch.Tensor(pca_data), torch.Tensor(data))
R2 = 1 - MSE/data.var()
print(f'MSE : {MSE.mean():.4f} ; R2 : {R2.mean():.4f}')

pca_data = pca.fit_transform(data)

plt.plot(pca.explained_variance_ratio_)
```

Listing 12: Apply PCA to our data

```
for i in range(n_components):
    plt.hist(pca_data[:,i], label=f'PC{i}', bins=200, alpha=0.5)
plt.legend()
plt.show()
```

Listing 13: Apply PCA to our data

```
def plot_with_density(x,y, log=False):
    # Step 1: Create a 2D histogram for density estimation
    hist, x_edges, y_edges = np.histogram2d(x,y, bins=(100, 100))
    # Step 2: Assign density values to points
    x_idx = np.digitize(x, x_edges) - 1  # Get the bin index for each x
    y_idx = np.digitize(y, y_edges) - 1  # Get the bin index for each y
    # Handle points that may fall on the right edge of the last bin
    x_idx[x_idx == hist.shape[0]] -= 1
    y_idx[y_idx == hist.shape[1]] -= 1
    # Use the histogram to assign a density value to each point
    if log:
        point_density = np.log(hist[x_idx, y_idx])
    else:
        point_density = hist[x_idx, y_idx]

    idx = point_density.argsort()
    x, y, point_density = x[idx], y[idx], point_density[idx]

    # Step 3: Color points by density in the scatter plot
    cmap = plt.get_cmap('viridis')  # Choose a color map
    return x,y,point_density

# Create a 2x3 subplot grid
fig, axs = plt.subplots(2, 3, figsize=(12, 8))

# Plot the images on the subplot grid
for i, ax in enumerate(axs.flatten()):
    x,y,point_density = plot_with_density(pca_data[:, i], pca_data[:, i+1], log=False)
    ax.scatter(x, y, c=point_density, cmap='viridis', s=10)
    ax.set_title(f'PC{i}-{i+1}')

plt.show()
```

Listing 14: Scatter plots

```
n_components = 3

pca = PCA(n_components=n_components)

pca_data = pca.fit_transform(data)
```

Listing 15: Transform data with a 3-component PCA

```
wcss = []
```

```
for k in range(1, 21):
    kmeans = KMeans(n_clusters=k, n_init=10)
    kmeans.fit(pca_data)
    wcss.append(kmeans.inertia_)

plt.plot(range(1, 21), wcss, marker='o',linestyle='--')
plt.xlabel('Number of Clusters')
plt.ylabel('WCSS')
plt.title('WCSS as a Function of Number of Clusters')
plt.xticks(range(1, 21))
plt.show()
```

Listing 16: Plot WCSS

```
from utils import plot_variable,show_results

n_clusters=6
kmeans_pca = KMeans(n_clusters=n_clusters)
kmeans_pca.fit(pca_data)
labels_pca = kmeans_pca.labels_
t = df.copy()[clustering_variables]
t.dropna(inplace=True)
t['clusters'] = kmeans_pca.labels_
t['Latitude'] = df['Latitude']
t['Longitude'] = df['Longitude']

plot_variable(t, 'clusters', colormap = cm.plasma)
```

Listing 17: Clustering PCA-reduced data Python

```
fig, axs = plt.subplots(1, 2, figsize=(20, 10))

# Plot the images on the subplot grid
for i, ax in enumerate(axs.flatten()):
    ax.scatter(pca_data[:, i], pca_data[:, i+1], c=labels_pca, cmap=cm.plasma, s=1, alpha
    =1)
    ax.set_title(f'PC{i}-{i+1}')
    ax.legend()
```

Listing 18: Scatter plots with colored clusters

5.2.10 3.3

```
class VariatonalAutoEncoder(nn.Module):
    """ Implementation of a Variational Autoencoder """

    def __init__(self, input_dim, hidden_dim, latent_dim):

        super().__init__()

        self.encoder = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.LeakyReLU(0.2),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            )

        self.mean_layer = nn.Linear(hidden_dim, latent_dim)
        self.logvar_layer = nn.Linear(hidden_dim, latent_dim)

        self.decoder = nn.Sequential(
```

```
            nn.Linear(latent_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, input_dim))

    def reparameterization(self, mean, logvar):
        std = torch.exp(0.5*logvar)
        epsilon = torch.randn_like(std).to(device)
        z = mean + std*epsilon
        return z

    def forward(self, x):
        x = self.encoder.forward(x)
        mu, logvar = self.mean_layer(x), self.logvar_layer(x)
        z = self.reparameterization(mu, logvar)
        x_hat = self.decoder.forward(z)
        return x_hat, mu, logvar

def ressemblance_metric(x, x_hat):
    return torch.mean((x - x_hat)**2)

def divergence_metric(mu, logvar):
    return -0.5 * torch.sum(1 + logvar - mu**2 - logvar.exp())
```

Listing 19: Variational Variatonal AutoEncoder implementation

```
def fit_model(model, optimizer, dataset, device, EPOCHS, alpha=1e-6, verbose=True):
    """ Trains the model on the dataset for a given number of epochs """

    globals().update({'device': device})

    model.train().to(device)
    writer = SummaryWriter() # tensorboard writer, to show the diffrerent histograms / metrics during the training

    for epoch in range(EPOCHS):

        # Metrics
        losses = []
        mses = []
        ressemblances = []
        divergences = []

        for batch in dataset:
            batch = batch.to(device)
            optimizer.zero_grad()

            x_hat, mean, logvar = model.forward(batch)

            # Loss
            ressemblance = ressemblance_metric(batch, x_hat)
            divergence = divergence_metric(mean, logvar)
            loss = ressemblance + alpha*divergence

            # Backpropagation
            loss.backward()
            optimizer.step()

            # Metrics
            batch_mse = F.mse_loss(batch, x_hat)
            losses.append(loss.item())
            mses.append(batch_mse.item())
            ressemblances.append(ressemblance.item())
```

```
37                 divergences.append(divergence.item()*alpha)
38
39             # Metrics
40             mean_mse = np.mean(mses)
41             rsquared = 1 - mean_mse/batch.var()
42             mean_loss = np.mean(losses)
43             writer.add_scalar('Loss/train', mean_loss, epoch)
44             writer.add_scalar('MSE/train', mean_mse, epoch)
45             writer.add_scalar('R2/train', rsquared, epoch)
46             writer.add_scalar('Ressemblance/train', np.mean(ressemblances), epoch)
47             writer.add_scalar('Divergence/train', np.mean(divergences), epoch)
48
49             # Histograms
50             writer.add_histogram('Mean', mean, epoch)
51             writer.add_histogram('Logvar', logvar, epoch)
52             writer.add_histogram('x_hat', x_hat, epoch)
53             writer.add_histogram('x', batch, epoch)
54
55             if verbose:
56                 print(f'Epoch N {epoch}/{EPOCHS} ; MSE : {mean_mse:.4f} ; Rsquared : {
       rsquared:.4f}; Average loss : {mean_loss:.4f}; Ressemblance : {np.mean(ressemblances)
       :.4f}; Divergence : {np.mean(divergences):.4f}')
```

Listing 20: Fitting the model

References

Chen, C., O. Dubovik, D. Fuertes, et al. (2020). "Validation of GRASP algorithm product from POLDER/-PARASOL data and assessment of multi-angular polarimetry potential for aerosol monitoring". In: *Earth System Science Data* 12.4, pp. 3573–3620. DOI: 10.5194/essd-12-3573-2020. URL: https://essd.copernicus.org/articles/12/3573/2020/ (cit. on p. 3).

Doersch, C. (2021). *Tutorial on Variational Autoencoders*. arXiv: 1606.05908 [stat.ML] (cit. on p. 12).

Guo, X., L. Gao, X. Liu, and J. Yin (2017). "Improved Deep Embedded Clustering with Local Structure Preservation". In: *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pp. 1753–1759. DOI: 10.24963/ijcai.2017/243. URL: https://doi.org/10.24963/ijcai.2017/243 (cit. on p. 19).

Herrera, M. E., O. Dubovik, B. Torres, et al. (2022). "Estimates of remote sensing retrieval errors by the GRASP algorithm: application to ground-based observations, concept and validation". In: *Atmospheric Measurement Techniques* 15.20, pp. 6075–6126. DOI: 10.5194/amt-15-6075-2022. URL: https://amt.copernicus.org/articles/15/6075/2022/ (cit. on p. 3).

Kingma, D. P. and M. Welling (2022). *Auto-Encoding Variational Bayes*. arXiv: 1312.6114 [stat.ML] (cit. on p. 12).

Rousseeuw, P. J. (1987). "Silhouettes: A graphical aid to the interpretation and validation of cluster analysis". In: *Journal of Computational and Applied Mathematics* 20, pp. 53–65. DOI: https://doi.org/10.1016/0377-0427(87)90125-7. URL: https://www.sciencedirect.com/science/article/pii/0377042787901257 (cit. on p. 8).

Vardakas, G., I. Papakostas, and A. Likas (2024). *Deep Clustering Using the Soft Silhouette Score: Towards Compact and Well-Separated Clusters*. arXiv: 2402.00608 [cs.LG] (cit. on p. 19).

Wold, S., K. Esbensen, and P. Geladi (1987). "Principal Component Analysis". In: *Chemometrics and Intelligent Laboratory Systems* 2, pp. 37–52. DOI: 10.1016/0169-7439(87)80084-9 (cit. on p. 9).