Probabilistic Graphical Models Projet
Weather Forecasting Data

Loris Bert Chloé Defour Thibaud Leteno Richard Serrano

Supervised by Rémi Emonet

January 2022

Contents

The code related to this report can be found at the following link :
`https://github.com/Morpheee/probabilistic_graphical_models`

1 Introduction

Following the challenge "Defi IA 2022" proposed by Meteo France, we had access to a database composed of several features ('month','latitude', 'longitude', 'altitude', 'wind_direction', 'wind_speed', 'temperature', 'humidity', 'dew_point', 'precipitations') which should allow us to predict the rainfall for the next day ('ground_truth').

It is interesting to note that this dataset contains missing values. For simplicity, we only keep the entries that do not contain missing values, resulting in a dataset of 82,120 examples. We will to proceed to use and test several algorithms using probability distributions.

First, we will examine sampling methods with Gibbs sampling, then an implementation of the Markov chain Monte Carlo method for imputation and prediction, before studying the Expectation Maximization algorithm, and finally detailing how a Variational Auto-Encoder model works.

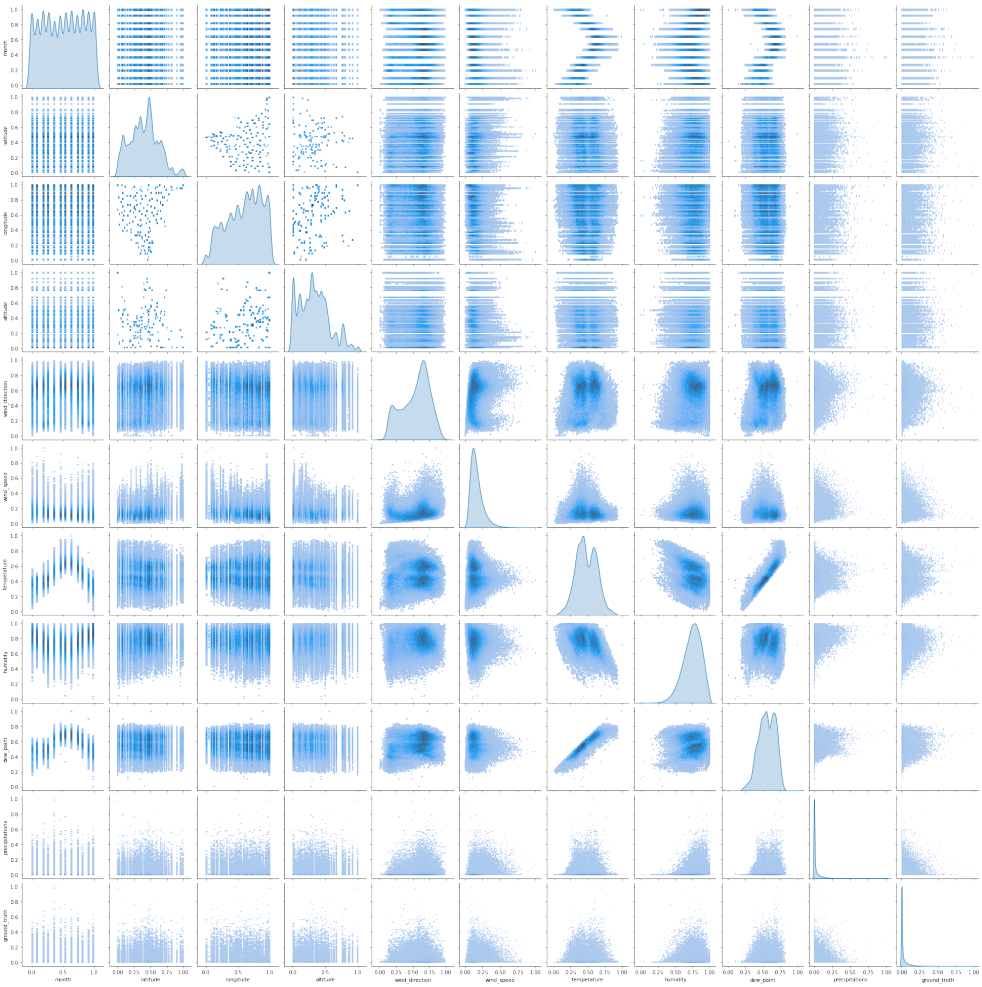

Figure 1: Pair-plot of the weather prediction dataset

2 Gibbs Sampling

2.1 Introduction

Gibbs Sampling [Stuart Geman, 1984] is a Markov Chain Monte Carlo method, it allows to generate vectors that follow a certain probability distribution. Its principle is simple: we randomly initialize the first vector, and at each step we compute each single coordinate of the vector according to its other coordinates.

With this method we can find the unknown parameters of a distribution or generate new samples, we will show it.

2.2 Parameters of a distribution

When we do not know the parameters of a distribution X, for example μ and σ in a Normal distribution, we can find them using the Gibbs Sampling method.

So we need to find μ and σ by deriving $p(\mu|X,\sigma)$ and $p(\sigma|X,\mu)$. Thanks to this, we can conclude that: $\mu \sim \mathcal{N}(\frac{\sum_i X_i}{N}, (\frac{\sigma}{\sqrt{N}})^2)$ and $\sigma^2 \sim \text{Inv-Gamma}(\frac{N}{2}-1, \frac{1}{2}\sum_i (X_i-\mu)^2))$. By repeating these computations several times, we get closer to the true parameters.

Here we will use this technique on different parameters, for example temperature (fig 2). We initialize our μ to 0, which initializes our σ to a very large value and our parameters will train to a value close to reality. We can note that it can be interesting to use the Gibbs Sampling with 2 Normal for the temperature because we find 2 pics, unfortunately for lack of time we did not succeed.

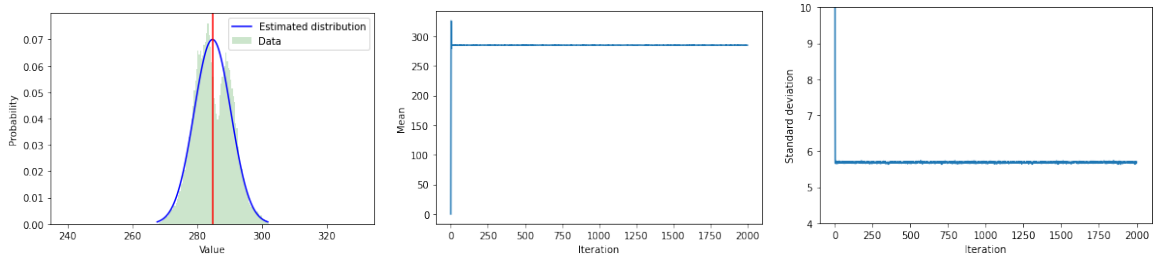

Figure 2: Gibbs sampling : find μ and σ on temperature distribution

We have also tried on other features (humidity, dew point) in the appendix (fig 24 and 23. Thus, we were able to better understand the Gibbs sampling algorithm, so we will use it to generate new samples.

2.3 Sampling with Gibbs sampling

We will use our Gibbs sampling method on 2 features of our database to create new samples. It is possible to use it for more features, but we focus on two features first. Here, the "parameters" we want to find are the coordinates of a sample with respect to the distribution of 2 features.

So we want to generate data $X \sim \mathcal{N}(\mu, \Sigma), \mu$ is a vector with the mean of our 2 distributions and Σ is the symmetric positive semi-definite covariance matrix. $\Sigma = [\Sigma_{00}\Sigma_{01}\Sigma_{10}\Sigma_{11}]$ The principle is simple: we first initialize a random point, then for as many samples as we want to create, we compute our new sample thanks to $p(x\_0|x\_1)$, must belong to the distribution of our first feature knowing our second feature then $p(x\_1|x\_0)$, must belong to the distribution of our second feature knowing our first feature. We need to compute μ and σ from the distribution of $p(x\_i|x\_j)$, with the derivation we find $\mu = mean[i] + (cov[i,j] * (p(x\_j|x\_i) - mean[j]))/cov[j,j]$ and $\sigma = np.sqrt(cov[i,i] - (cov[i,j] * *2)/cov[j,j])$, now we just need to compute a random number from the normal distribution with our new μ and σ.

It is clearer to see the code in appendix. Even if the distributions of our features are not perfect normal distributions, the results are quite good (fig 3). We can see some outliers, we can note that even if we initialize badly, the vast majority of the points are well in the true distribution.

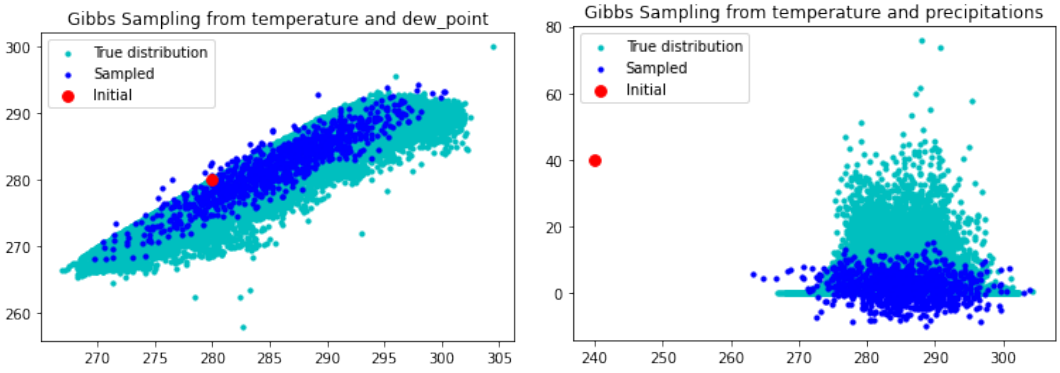

Figure 3: Gibbs sampling : sampling on temperature, dew point and temperature, precipitation

3 Data Imputation

Data imputation is key in Machine Learning when some data is missing, the task aims at filling the gap using various approaches like replacing missing values by the mean, and arbitrary value or using Machine Leaning approaches such as k-Nearest Neighbors.

The dataset we use for the Machine Learning and Data Mining project contains a lot of 'Not a Number' values (NaNs) : more than 40% of the entries contains NaNs. Thus, it has been an important task to ensure we can impute the data as close as possible to the real value would have been. We want to explore this task from a probabilistic perspective. Note that on all the works on data imputation, we used complete data where we artificially created NaNs to be able to evaluate the efficiency of the methods.

To tackle the problem, we consider Markov Chain Monte Carlo algorithm, more specifically we used No-U-Turn Sampler (NUTS) [Hoffman et al., 2014]. Finally, all the implementations are made using the NumPyro framework [Phan et al., 2019], [Bingham et al., 2018], the code is inspired from the examples of the documentation [bay, b], [bay, a].

3.1 No U-Turn Sampler (NUTS)

Hamiltonian Monte Carlo (HMC) is a specific MCMC algorithm, not sensitive to correlations between parameters. However, two parameters needs to be tuned : the number of steps in the algorithm and their size, those parameters can be hard to tune. Therefore, [Hoffman et al., 2014] introduce NUTS, an HMC extension that automatically deal with those parameters using a recursive algorithm, to simplify the posterior distribution contribution.

3.2 Imputation of hidden variables

We first investigate the idea of imputing the hidden variables instead of the variables themselves. For each feature, we draw the hidden variables (also for the missing entries) from a Gaussian distribution, $\mathcal{N}(\mu, \sigma)$ where μ and σ are respectively the mean and the standard deviation computed from the known entries of the features.
Even though, this approach allows computing the posterior distribution, we still have NaNs in the data. Therefore, we cannot use it to make predictions.

3.3 Use of probabilistic model for imputation and prediction

To face the issue of the previous section, we aim at using probabilistic models to fill directly the missing values. We will use the *Predictive class* of NumPyro to build a predictive distribution using the posterior. To find the posterior, we will keep using the NUTS algorithm.
For the imputation, we work feature by feature. First, we separate data with missing values for the current feature from the rest of the entries; Then, we do the NUTS algorithms with the data without NaNs to obtain the posterior distribution. Then, we feed the posterior and the data with missing entries to the *Predictive class*, to obtain a prediction of each missing values. The *Predictive class* draw 1000 samples for each NaN, and we take the mean of those samples to impute the missing entry. In a second time, we execute a new NUTS to obtain the posterior on the whole dataset (with imputed entries). Using the *Predictive class*, we can now estimate the amount of rain per data (our initial task).

3.3.1 Imputation of one feature

We study this approach by creating NaNs in only one feature of the dataset. We successively realize it on *'wind_direction'* which has a low correlation with the target

feature *'ground_truth'* (Pearson's correlation: 0.162, Spearman's correlation: 0.091), and *'precipitations'* which is more correlated to *'ground_truth'* (Pearson's correlation: 0.394, Spearman's correlation: 0.232). We realize the experiment with different ratios of NaNs : 1%, 10%, 25%, 50%, 75% and 85%.
We want to evaluate the quality of the imputation and of the predictions using this method. To do so, we will use two metrics, the Mean Absolute Error (MAE) and the Mean Absolute Percentage Error (MAPE).

$$MAE(y\_true, y\_pred) = \frac{1}{n_{samples}} \sum_{i=0}^{n_{samples}-1} |y\_true_i - y\_pred_i| \tag{3.1}$$

$$MAPE(y\_true, y\_pred) = \frac{1}{n_{samples}} \sum_{i=0}^{n_{samples}-1} \frac{|y\_true_i - y\_pred_i|}{max(\epsilon, |y\_true_i|)} \tag{3.2}$$

with ϵ a positive fixed constant.

Results

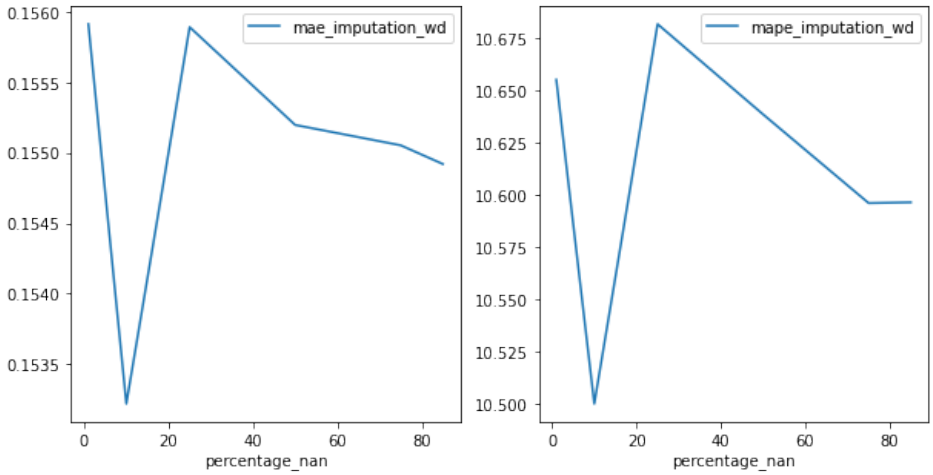

Figure 4: Evaluation of the imputation for the wind direction feature

We observe on fig. 4 and 5, that the MAPE and MAE are low for both dataset, which indicates a good ability of the algorithm to find imputed values close to the original data. Note that the percentage of missing data does not impact the quality of the imputation, even with 85% of missing values the algorithm obtains similar results : it indicates that 15% of our dataset (12318 entries) is sufficient to find the posterior distribution of the features. However, the distribution found for the *precipitations* is more accurate than the one found for the *wind_direction*.

Figure 6 shows the time to impute the missing entries (in seconds) depending on the rate of NaNs. We observe that imputing fewer values takes more time. Thus, we assume that the larger the data used to find the distribution is, the longer it will take

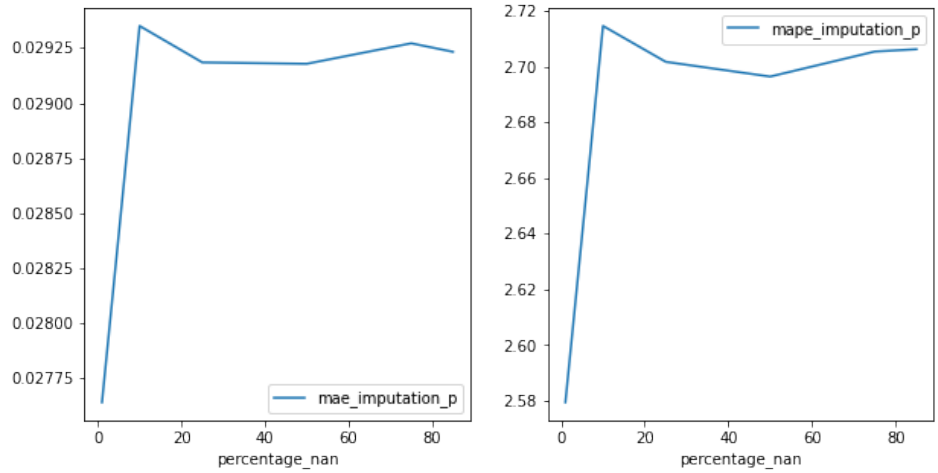

Figure 5: Evaluation of the imputation for the precipitations feature

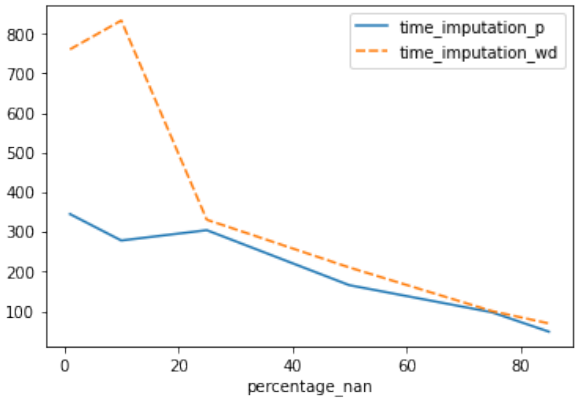

Figure 6: Imputation duration for features wind direction and precipitations, time is in seconds

to find the posterior distributions. It can be a drawback of the method when applied on large dataset.

On fig. 7, we focus on the quality of the predictions of the ground truth depending on the ratio of imputed entries. Remind that *precipitations* is more correlated with *ground_truth* than *wind_direction*. The amount of imputed values on the feature *wind_direction* does not seem to impact significantly the quality of the final prediction. On the other hand, when the amount of imputed values on the feature *precipitations* increases, we observe a trend where both metrics increase, indicating less accurate predictions. It shows that this method is not infallible and using it on features highly correlated with the target feature will conduct to inaccurate results. More, precisely, it implies that probabilistic models are sensitive to correlation with features. Therefore, if we use only known data, the model should perform well but by introducing uncertainty with the imputation of correlated features, the use of NUTS for the prediction of the

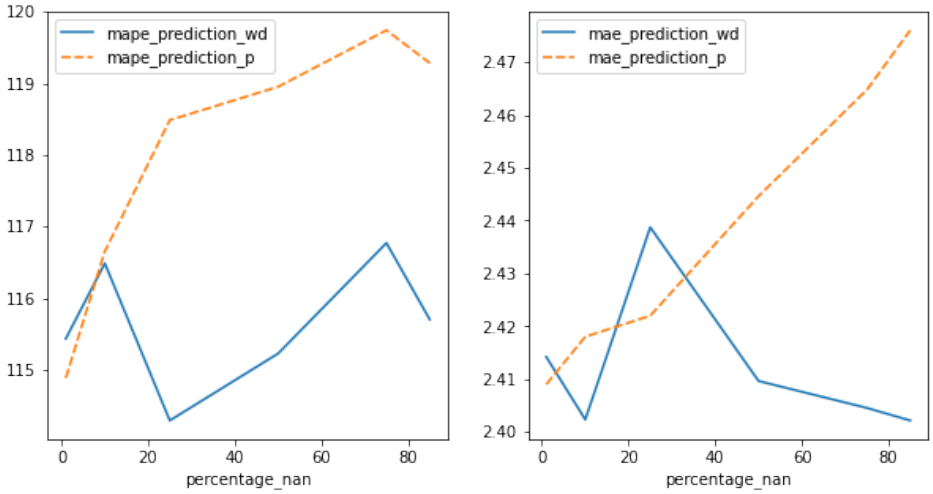

Figure 7: Evaluation of the ground truth predictions in function of the amount of NaNs in features wind direction or precipitations

ground truth with posterior might not be recommended.

3.3.2 Imputation of several features

In this section, we study the case where missing values are present in several features (here two features *wind_direction* and *precipitations*). The solution is two impute successively the two features using the same approach as in the previous session, however, features with NaNs cannot be taken in consideration during the imputation of other features.

Results

In figure 8, we see that the evaluations when missing values are present in both features are proportional to the one when only one feature is concerned by NaNs. However, both MAPE and MAE are higher in this case. We deduce that the imprecision caused by imputing a feature is cumulative. Thus, the more features contain NaNS, the worst the results are.

3.3.3 Discussion

Based on these experiments, the imputation using MCMC NUTS algorithm, is interesting. The algorithm perform well at estimating the feature's posterior distributions and sampling accurate values for missing entries. However, the algorithm is long and hardly scalable : estimating posterior on larger dataset increases the computation time. Finally, using it for a regression problem suffers from several drawbacks : the algorithm seems very sensitive to correlation within features even though NUTS is supposed to be robust to it, thus, if a feature correlated to the target one presents anomalies, it will

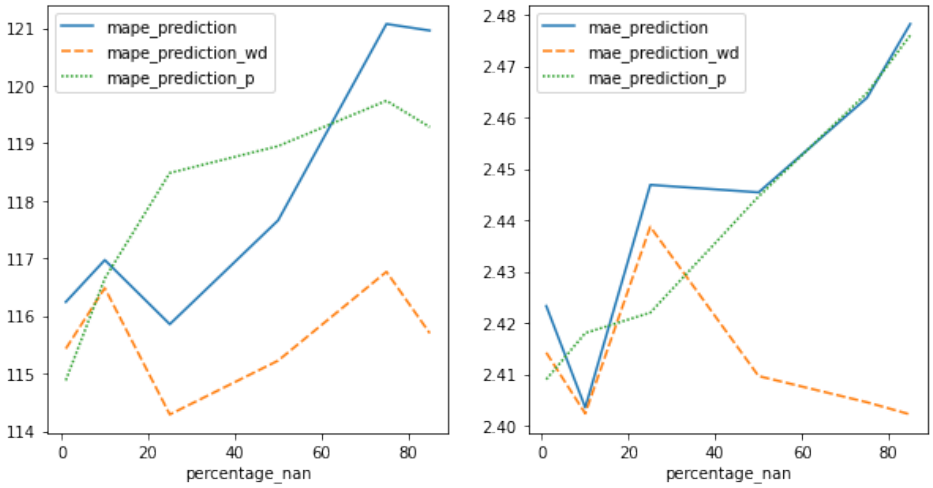

Figure 8: Evaluation of the ground truth predictions in function of the amount of NaNs in features wind direction, precipitations and both

have a strong impact on the final results. Also, using data with imputed values, which is often the case, will create variations in the feature's posterior distributions inducing poor predictions.
To sum up, this approach is usable for imputation but not recommended in estimating predictions from the posterior.

4 Expectation Maximization

4.1 Theory

The expectation Maximization (EM) algorithm [Moon, 1996] aims at finding the latent variables of a dataset's underlying distribution. The algorithms alternate between the Expectation (E) step and the Maximization (M) step. Most often, the EM algorithm aims at fitting a Gaussian Mixture Model (GMM) to the given data.

We assume that the database is composed of$M$ points : $X = (X_i)_{i \leq M} \in \mathbb{R}^M$. The goal will be to fit a GMM of N Gaussian distributions of the form :

$$GMM(X|\theta) = \sum_{k=1}^{N} w_i \times \mathcal{N}(X|\mu_k, \Sigma_k^2)$$

where $\theta = \{w_1, \ldots, x_N, \mu_1, \ldots, \mu_N, \Sigma_1, \ldots, \Sigma_N\}$, and such that $\sum_{i=1}^{N} w_i = 1$.

Here, we suppose that all parameters are unknown, we define RM such that is estimates them all, though, it is perfectly possible to fix some parameters beforehand.

Let's have a look at the two steps of the EM algorithm.

- **E-step** : Given the current parameters θ, give the likelihood of each Normal Distribution. The likelihood is a heuristic, as it is base on the parameters θ that we previously estimated. More formally, the E-step computes for each $k \leq N$:

$$\begin{aligned} R_{ik} &= p(Y_i = k|X_i, \theta) && (4.1)\\ &= \frac{p(X_i|Y_i = k, \theta)p(Y_i = k|\theta)}{\sum_{k'=1}^{N} p(X_i|Y_i = k', \theta)p(Y_i = k'|\theta)} && (4.2)\\ &= \frac{\mathcal{N}(X_i|\mu_{Y_i=k}, \Sigma_{Y_i=k}) \times w_{Y_i=k}}{\sum_{k'=1}^{N} \mathcal{N}(X_i|\mu_{Y_i=k'}, \Sigma_{Y_i=k'}) \times w_{Y_i=k'}} && (4.3)\\ &=: \frac{R'_{ik}}{\sum_{k'=1}^{N} R'_{ik'}} && (4.4) \end{aligned}$$

  Now that we have a closed form of R_{ik}, depending only on the heuristic we already have, we can continue with the M-step.

- **M-step** : Given the likelihood, this step consists of computing a new set of parameters θ^{new} such that it maximized the likelihood. To do so, we use the maximum likelihood estimation of θ and set $\theta^{new} = \theta_{MLE}$. That to say :

$$\begin{aligned} \theta^{new} &= arg\max_{\theta} \mathcal{L}(\theta|X, \theta^{old}) = arg\max_{\theta} Q(\theta|\theta^{old}) && (4.5)\\ &= arg\max_{\theta} \mathbb{E}_q(\ln(p(X, Y|\theta))) && (4.6)\\ &= arg\max_{\theta} \sum_{i=1}^{M}\sum_{k=1}^{N} R_{ik}\ln(w_k) - \frac{R_{ik}}{2} \times \left(\frac{X_i - \mu_k}{\Sigma}\right)^2 + K_1 && (4.7) \end{aligned}$$

  where $K_1 \in \mathbb{R}$ is a constant. Note that $q(Y_i = k) := R_{ik} = p(Y_i = k|X_i, \theta^{old})$. Also, as the logarithm is constantly increasing, it does not change the $arg\max$ result, thus we use the log-likelihood.

  In order to find θ^{new}, we then compute the Lagrangian of the previously found formula :

$$\mathcal{L}(\theta, \lambda) = \sum_{i=1}^{M}\sum_{k=1}^{N} R_{ik}\ln(w_k) - \frac{R_{ik}}{2} \times \left(\frac{X_i - \mu_k}{\Sigma}\right)^2 + K_1 + \lambda\left(\sum_{k=1}^{N} w_k - 1\right) \quad (4.8)$$

  We get the following:

$$\forall k \leq N,\ \frac{\partial\mathcal{L}(\theta, \lambda)}{\partial\mu_k} = 0 \iff \mu_k = \frac{\sum_{i=1}^{M} R_{ik}X_i}{\sum_{i=1}^{M} R_{ik}} \quad (4.9)$$

$$\frac{\partial \mathcal{L}(\theta, \lambda)}{\partial \lambda} = 0 \iff \sum_{k=1}^{N} w_k = 1 \tag{4.10}$$

$$\forall k \leq N,\ \frac{\partial \mathcal{L}(\theta, \lambda)}{\partial w_k} = 0 \iff w_k = \frac{\sum_{i=1}^{M} R_{ik}}{-\lambda} = \frac{\sum_{i=1}^{M} R_{ik}}{\sum_{k=1}^{N}\sum_{i=1}^{M} R_{ik}} = \frac{\sum_{i=1}^{M} R_{ik}}{M} \tag{4.11}$$

$$\tag{4.12}$$

$$\forall k \leq N,\ \frac{\partial \mathcal{L}(\theta, \lambda)}{\partial \Sigma_k} = 0 \iff \Sigma_k = \frac{\sum_{i=1}^{M} R_{ik}(X_i - \mu_k)(X_i - \mu_k)^T}{\sum_{i=1}^{M} R_{ik}} \tag{4.13}$$

The EM algorithm consists of iterating between those two steps into a convergence (the likelihood does not increase anymore) or a maximum number of iterations. When such a limit is reached, the algorithm outputs the latent parameters θ.

As one can see, the algorithm is dependent on its initialization. There are multiple ways of initializing the algorithm. One could decide to each initial parameter depend on a Normal distribution with a large variable so that the algorithm does not converge toward at the same local minima every time it runs. More formally, $\forall k \leq N$:

$$\mu_k^{(0)} \sim \mathcal{N}(0, 20) \quad ; \quad \Sigma_k^{(0)} \sim \mathcal{N}(10, 5) \quad ; \quad w_k^{(0)} \sim \mathcal{N}(1, 0.5) \tag{4.14}$$

then, as $\sum_{i=k}^{N} w_k = 1$, one must assign to $w_k \leftarrow \frac{w_k^{(0)}}{\sum_{k=1}^{N} w_k}$. Another way around would be to sample a small portion of the database and to compute its mean and variance, then to derive the initial parameters from this prior. Let's denote $(X_i^{samp})_{\phi(i) \leq M} \subset X$ a uniform sample of X, with ϕ a strictly increasing function. Then one can compute $\mu^{samp} = \mathbb{E}(X^{samp})$, $\Sigma^{samp} = \mathbb{V}(X^{samp})^{\frac{1}{2}}$ and draw the initial parameters as :

$$\mu_k^{(0)} \sim \mathcal{N}(\mu^{samp}, 20); \quad diag(\Sigma_k^{(0)}) \sim \mathcal{N}(diag(\Sigma^{samp}), 5); \quad w_k^{(0)} \sim \mathcal{N}(1, 0.5) \tag{4.15}$$

We decided to use the latter method to fast up the process of convergence, making the initialization responsive to any dataset. We still keep a high variance as we want to increase the spread of the initial parameters.

Note that one must ensure that $\Sigma^{(0)} \succ 0$ and $w^{(0)} \succ 0$.

One can find the Bayesian network of the algorithm in Figure 9

From now on, we consider the dataset X without the X^{sampl} part, as this would be biased to use the same data in the prior and in the algorithm. Every histogram of the data uses the whole original dataset, but the training part is done on $X \leftarrow X \backslash X^{samp}$.

Let's look at the results obtained with the EM algorithm that we implemented.

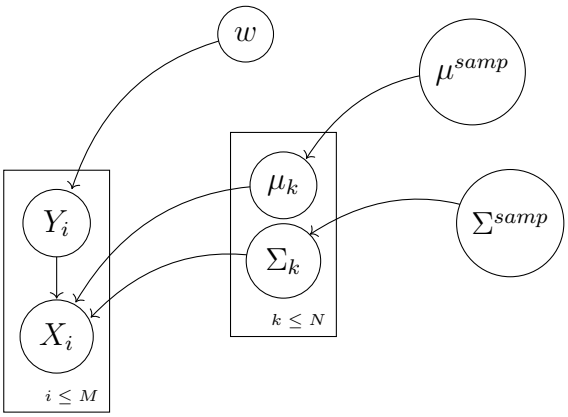

Figure 9: Bayesian Network of the EM algorithm with X^{samp} for prior.

4.2 EM in 1 dimension

4.2.1 The raw algorithm

Note that for the implementation of the algorithm, we used some code from this python file[1] by Dr. R. Emonet.

This first implementation takes as input a 1 dimensional dataset. For example, one can see in Figure 11 that we fit 2 and 3 normal distribution(s) to the *temperature* histogram.

In most cases, the RM algorithm succeeds to feat the distribution, and visually we can tell that 2 or 3 Gaussian distributions seems to be the actual underlying GMM distribution of the feature.

As we say above, the initialization is crucial, and we can get either good or bad results based of the initialization. For example, one can see in Figure 10 that, due to the initialization, the mean converged toward the mean of the whole distribution and the GMM does not describe properly the histogram.

In order to smooth the results and avoid the bad converge in average, we implemented an extension to the algorithm that one can see in the next section.

4.2.2 Going further with a k-means clustering of the distributions

In order to prevent the algorithm from outputting distributions that didn't converge properly, we run the algorithm multiple time and then average the distributions. In order to make a "smart" averaging, we compute a k-means algorithm on the space of the latent parameters to extract k clusters. k is set to be the desired number of distributions : $k \leftarrow N$. Once we have identified N clusters in the latent features space, we can compute the barycenter of each Normal distribution by averaging the mean,

[1] GitHub repository :`https://github.com/twitwi/Code4PGM/blob/main/lesson-02-s7-emgmm1d.ipynb`

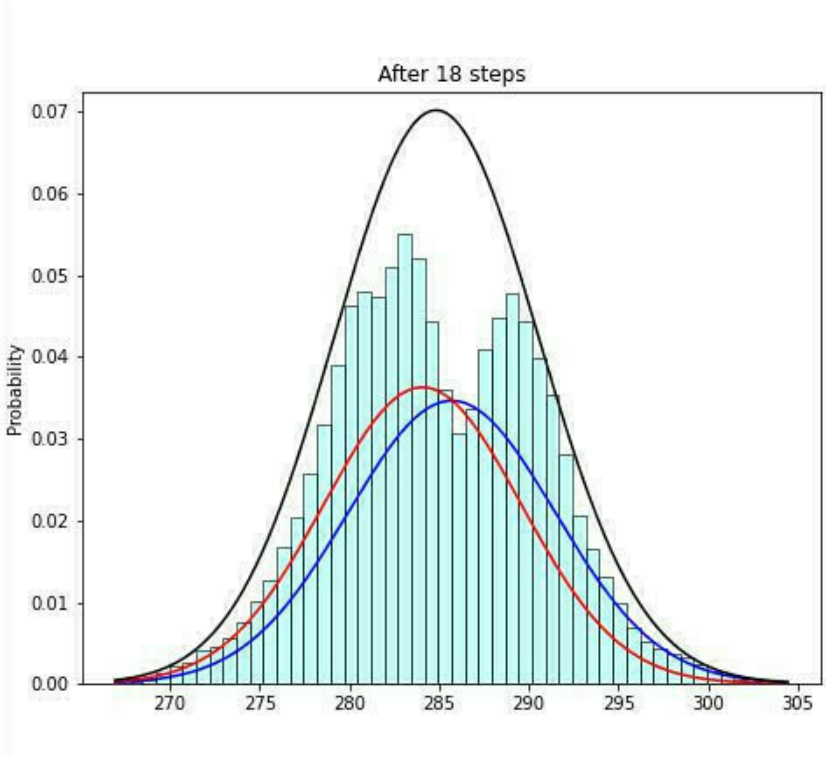

Figure 10: Example of bad initialization for EM algorithm.

variance and weight of each cluster's distributions. More formally, $\forall k \leq N$, we denote C_k the k-th cluster, and we get :

$$\mu_k^{bary} = \frac{1}{|C_k|} \sum_{k' \in C_k} \mu_{k'} \tag{4.16}$$

$$\sigma_k^{bary} = \frac{1}{|C_k|} \sum_{k' \in C_k} \sigma_{k'} \tag{4.17}$$

$$w_k^{bary} = \frac{1}{|C_k|} \sum_{k' \in C_k} w_{k'} \tag{4.18}$$

$$\tag{4.19}$$

Note that the clusters are computed over 10 runs of the k-means algorithms with different initialization and the best clusters are kept.

One can see in Figure 12 that the distributions of each run lands on a different local optimum. For the *dew point*, the distributions are very close to each other, and with more run they would have probably converged toward the same 4 unique distributions. Oppositely, the 3 distributions of the *temperature* do not converge toward the same parameters depending on their initialization. In this case, doing such an average over the distribution allows us to avoid the bad convergence.

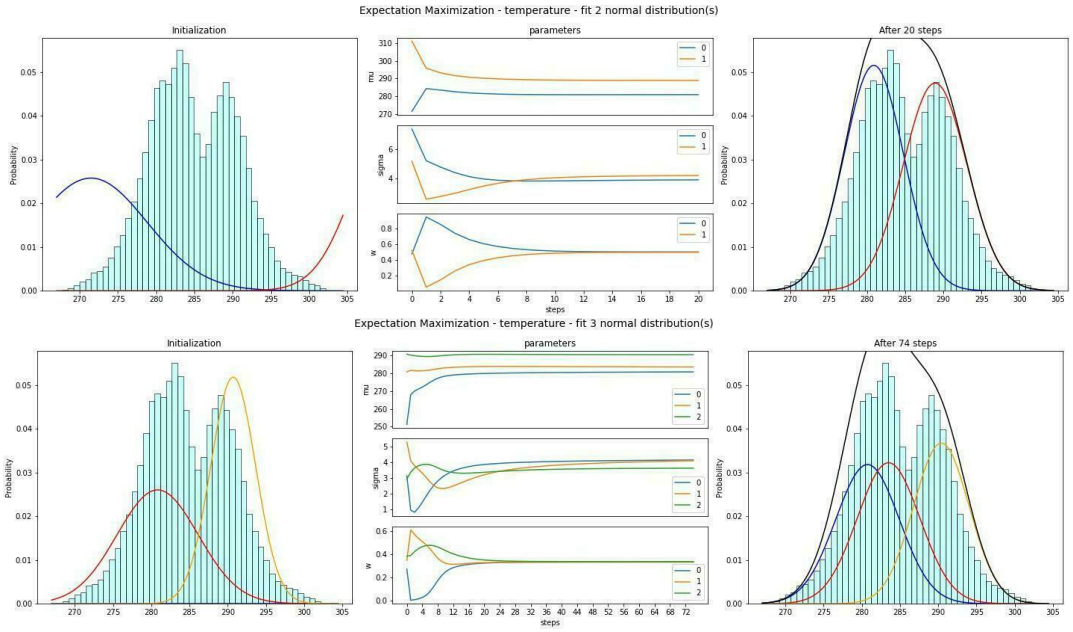

Figure 11: Fitting 2 and 3 normal distribution(s) to the *temperature* histogram from the database. (Left) Initialization : the histogram is still the one of the original feature (here the temperature), the curves are plotted from the initialization parameters. (Center) Progression : each latent parameter is plotted in a different graph (top to down : μ, σ, w) in function of the number of iteration until convergence or maximum iterations reached. (Right) Final step : after iterating the E-step and the M-step for long enough, the algorithm returns the latent parameters of the GMM, each plot represent a Normal distribution and the black curve (*Global*) is the GMM itself as weighted sum of the normal distributions.

4.3 EM in 2 dimensions

This last algorithm is the 2 dimensional version of the EM algorithm. We use part of the article by toward-data-science [twd,] to implement this algorithm. The algorithm is the same as in 1 dimension, except that $\mu \in \mathbb{R}^2$, $\Sigma \in \mathbb{R}^{2\times 2}$: we iterate through the E and M steps until the log-likelihood does not evolve anymore or the maximum number of iterations is reached.

We plot the histogram of two features at the same time, and the algorithm aims at fitting k normal distributions to the histogram. One can see an example of the histogram with the *humidity* and the *dew point* with 2 Normal distributions fitted in Figure 13.

4.4 Results

The EM algorithm is very dependent on the initialization of the latent parameters. It is very important to run it multiple time with different initial parameters to have an

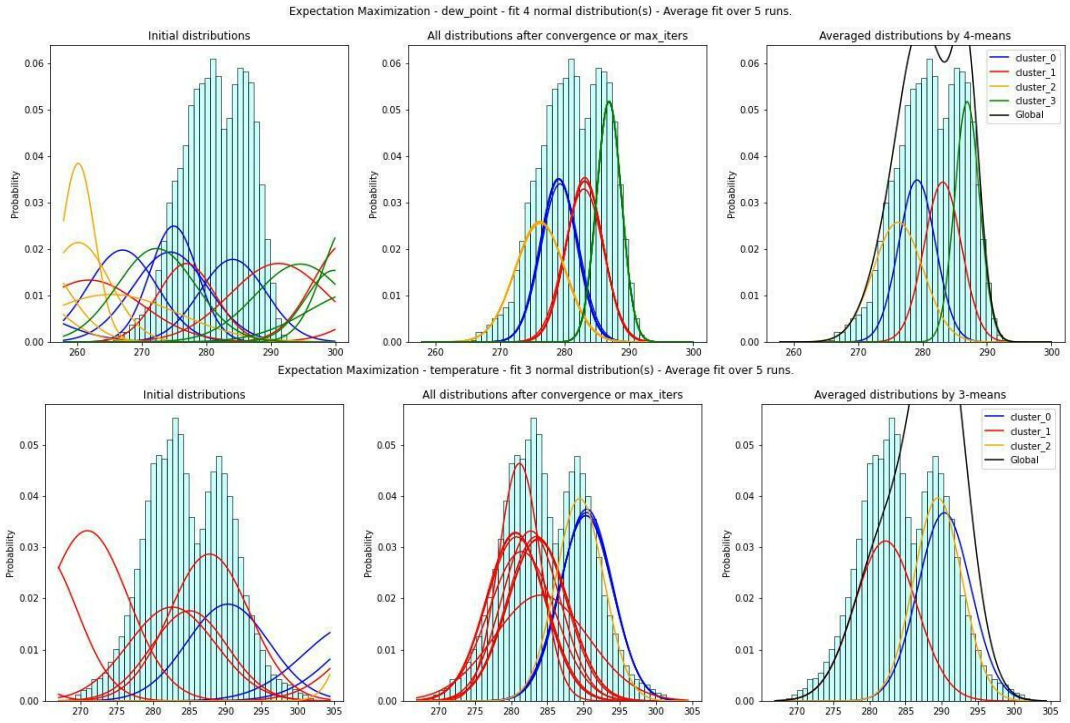

Figure 12: Fitting 4 normal distributions to the *dew point* (above) and 3 normal distributions to the *temperature* (below), over 5 runs, and averaging by doing a barycenter of the distributions of each cluster. (Right) Initialization. (Center) Final step of each run. (Right) Barycenter distribution.

idea of the actual good fit. We implement a solution for this issue through a k-means algorithm. This method takes a lot more time as we need to run the algorithm many times.

When the algorithm start with a good initialization, the result can be pretty impressive, as it manages to converge with very few steps (always without an order of magnitude of 10) and it gives a very good approximation of the distribution, in 1 or 2 dimensions.

Averaging the distributions is not as effective as we thought it would be. Indeed, when all the distributions converged toward the same N parameters, there are no needs to compute an average. Oppositely, when some converge toward bad local maxima, the average is impacted a lot by the bad them.

The EM-2D algorithm shows great results if initialized with the good values, and one can see that the log-likelihood converges pretty fast (Figure 13

4.5 Going further

We are going to list, first, ways to improve the algorithms we propose in this paper. In a second time, we will discuss of possible use of such an algorithm.

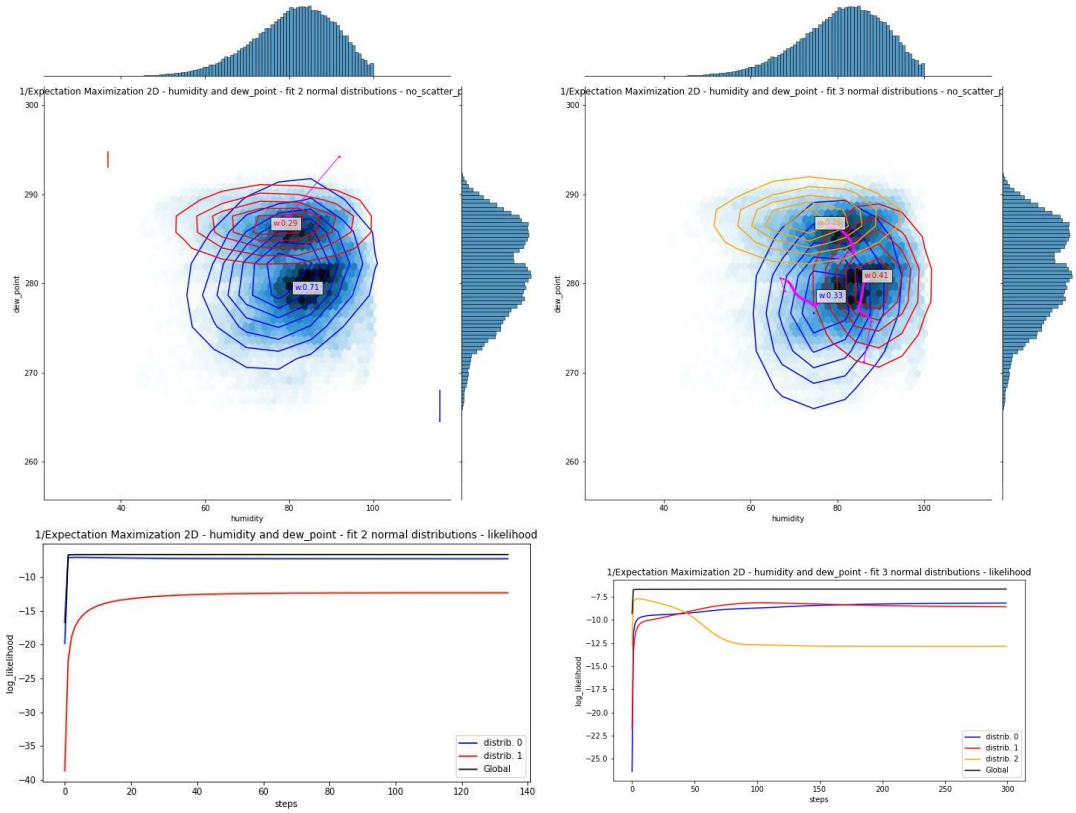

Figure 13: (Above) Histogram of the *Humidity* (x-axis) and the *dew point* (y-axis) with 2 (above) and 3 (below) normal distributions fitted by the 2D-EM algorithm. The magenta line is the location of the distributions over the iterations. The $\boxed{w = weight}$ in the center of each distribution is the weight of each distribution.
(Below) the corresponding log-likelihood.

An issue we face is that when we try to fit too many distributions, or that one tends to be very "thin", some σ converges toward 0 ($\det(\Sigma) \to 0$ in 2D). Such a convergence highlights that the GMM is not suited to fit the distribution, or that the initial parameters are not good enough. It could be interesting to just remove the corresponding distribution from the algorithm instead of fixing a minimum value for, σ as we do. note that the Gaussian distribution is not always adapted, as one can see in Figure 14 with the *current precipitation*.

Related to the previous one, a second improvement could be to be able to fit other distributions than Normal once. For example, in the same figure, one can imagine that one or multiple exponential distributions would be more adapted.

About the averaging over k clusters, we think it could be interesting to identify very bad local maxima and to delete the corresponding distributions before running the k-means algorithm. We can actually do this by computing the likelihood of each distribution and just removing the outliers (the highest once). Another improvement would be to force a minimum number (or even a same number) of individuals by clusters.

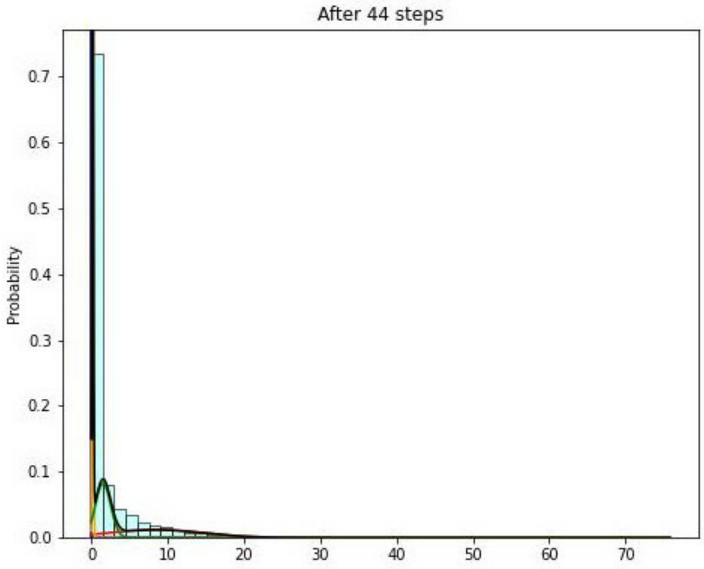

Figure 14: Example of distribution for which a GMM is not adapted. Feature : *current precipitation*.

Indeed, as one can see in Figure 12, when looking at the *temperature* plots, the clusters 0 and 2 have way less individuals to average from than the cluster 1. Also, another way of averaging could be to use optimal transport barycenter [Bonneel et al., 2016]. Even though it is not necessary with Gaussian distributions as their barycenter has a closed form, it could be interesting if mixing different distributions of if using other distributions in general. We think that another way around could be to use the median instead of the average over the distributions' parameters inside a cluster.

About the EM-2D algorithm, we think it could be worth it to implement a similar averaging method to be able to run the algorithm with different distributions.

Now, let us expose a way of using the algorithm. When retrieving the latent parameters from the underlying distribution of multiple features (let's say N features), one has the probability $p(X_{1,\dots,N})$. We could use this acquired knowledge to either perform imputation of missing value or prediction. Let's call Y either the missing value or the variable to be predicted. Using EM, one knows the joint probability $p(X_{1,\dots,N}, Y)$. Thus, they can estimate Y using $p(Y|X_{1,\dots,N}) = \dfrac{p(X_{1,\dots,N}, Y)}{p(X_{1,\dots,N})}$.

Another way of using the latent parameters found by the EM algorithm is to keep only the GMM parameters to describe the data. That way, one can describe a continuous distribution carrying significantly fewer parameters than with the original dataset. As EM is very sensitive to its initialization, one would have to be careful before doing such compression, it would require some improvements, and strong guaranties, possibly provided by the solutions explained above.

One can find more example of the 1D and 2D algorithms in the Appendix section, see Figure 25 (1D average example) and Figure 26 (2D examples).

5 Variational Auto-Encoder

In this section, we will explain what a Variational Auto-Encoder (VAE) architecture is, what differs from a classical Auto-Encoder (AE), and how we can apply it to our problem.

5.1 Dimensionality reduction and Auto-Encoders

Dimensionality reduction is a process used in Machine Learning to lower the number of features that describes a given data set. It can be done to speed up the computation, to increase correlation, or for visualization purposes; either by selecting the features to keep from the existing ones or project them to a new lower-dim space. There exist a large number of dimensionality reduction algorithms (linear/non-linear, deterministic/probabilistic, ...), some of which are more suitable for a certain type of data than others.

Auto-Encoders [Kramer, 1991] belong to the non-linear deterministic ones. It is composed of an "Encoder" (denoted E) that compresses the data down to a low dimension vector and a "Decoder" (denoted D) that computes the inverse projection, back to the original space (Fig. 15). The aim is to reconstruct the input as good as possible, $\hat{X} = D(E(X))$, by minimizing a loss l (usually Mean Squared Error) such that $\hat{X} \rightarrow X$; which comes down to finding E^* and D^* that satisfy $(E^*, D^*) = arg\min_{E,D}[l(X, D(E(X)))]$. Depending on the input distribution, the architecture of the encoder/decoder (usually symmetric with Deep Neural Networks) and the dimensionality of the latent space, the encoding can be lossless (perfect reconstruction) or lossy (inexact reconstruction). The latter can be desired in the case of denoising, for example.

The major drawback with classical Auto-Encoders is the lack of regularity in the encoded space. This is a problem for data generation : a point randomly sampled from the latent space is not guaranteed to lead to a meaningful result when decoded. Indeed, nothing in the Auto-Encoder architecture or during training forces the latent space to be well-organised. This is the issue that Variational Auto-Encoders seek to address.

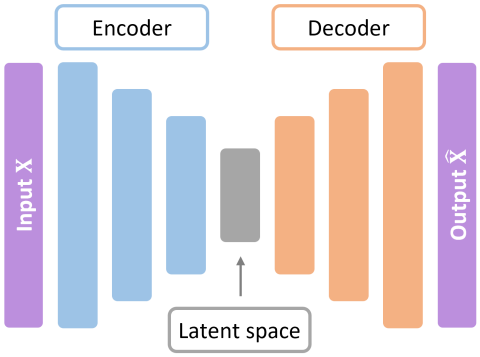

Figure 15: Auto-Encoder architecture

5.2 Variational Auto-Encoders

Variational Auto-Encoders were introduced in 2014 by [Kingma and Welling, 2014] as a way to do an efficient approximate posterior inference even when the posterior distribution is intractable. The architecture differ from classical Auto-Encoders in that the latent space is now a distribution, generally a Normal one, from which data points are randomly sampled to feed the decoder $z \sim \mathcal{N}(\mu, \sigma^2)$ (Fig. 16). Therefore, the model is now non-deterministic : for a fixed set of weights, the same input will lead to a (slightly) different output every time.

The minimization problem is also different, the loss function is composed of two terms : the reconstruction error (same as classical Auto-Encoders) and a regularization term on the latent space to ensure that it follows the chosen prior distribution. In practice, the prior is a Normal distribution and the regularization is the KL-divergence between the two :

$$KL(q_\phi(z|x) \;||\; p(z)) = -\frac{1}{2}\sum_i [\sigma_i + \mu_i^2 - 1 - log(\sigma_i)]$$

The total loss can be written :

$$\begin{aligned} loss &= reconstruction\ \ loss + regularization\ \ loss \\ &= \|X - D(E(X))\|^2 + KL(\mathcal{N}(\mu, \sigma^2) \;||\; \mathcal{N}(0,1)) \end{aligned} \tag{5.1}$$

It encourages the encodings to be distributed evenly around the center of the latent space and penalize the model if they are spread in specific regions. Hence, the regularization guaranty the continuity and the completeness of the latent space, i.e. decoded points are similar if their encoded vectors are similar and every decoded point sampled from the latent space is meaningful. Figure 17 allows to visualize well such properties (images from the MNIST dataset of 28x28 pixels were projected down to 2 dimensions using a VAE, and after training, some points were sampled and projected back).

However, training the model with backpropagation is not possible in this configuration because the sampling operation in the latent space is probabilistic and therefore

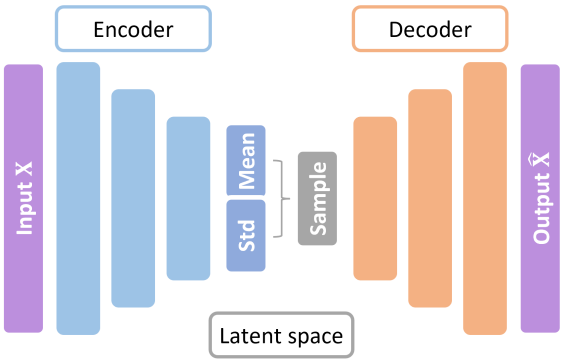

Figure 16: Variational Auto-Encoder architecture

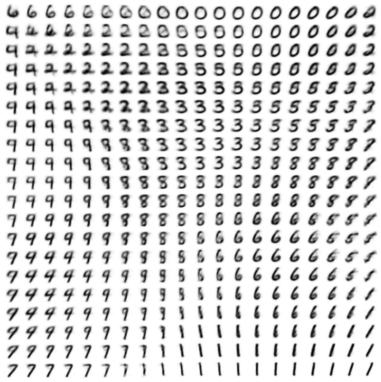

Figure 17: MNIST projection from a 2D manifold

not differentiable. To overcome this problem, the authors used what they called the "Reparameterization trick" (Fig. 18). The key idea is to shift the stochastic node outside the path of the gradient by keeping μ and σ fixed and using another variable $\epsilon \sim \mathcal{N}(0, 1)$. That way, $z = \mu + \sigma * \epsilon$ is perfectly equivalent to $z \sim \mathcal{N}(\mu, \sigma^2)$, but the gradient can flow through the two (now deterministic) parameters.

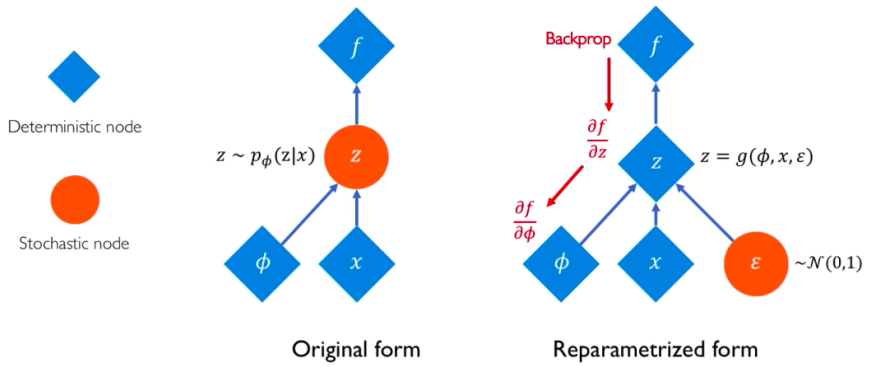

Figure 18: Reparametrization Trick (source : [Sayak, 2020])

5.3 Experimentation

For our experimentation, we implemented a Variational Auto-Encoders in TensorFlow and applied it to the 10-features weather dataset. Due to time constrains, we only tried models that are composed of a single layer of fully connected ("Dense") neurons (one for the encoder and one for the decoder). We studied the effects of two parameters: the size of the latent space and the number of neurons in the fully connected layer. For the former, we projected the data to 2, 5, 10 and 15 dimensions ; while for the latter, we tried with 4, 8, 16, 32, 64 and 128 neurons. The dataset was randomly split into

a training set (of 74,000 points) and a validation set (of 8,000 points) to evaluate the reconstruction capacity of the models and check for overfitting.

5.4 Results

Figure 19 shows the VAEs training curves for all parameter combinations. Note that the KL-divergence plots are computed between the original distribution and the reconstructed distributions (not from the latent space). From left to right, the number of latent dimensions is increasing, and we can clearly see that the reconstruction loss (MSE) and the KL-divergence are getting smaller and smaller overall. With 10 and 15 latent dim (and a sufficient number of neurons), the error is so small that the data points are correctly reconstructed. However, latent spaces with fewer dimensions (2 and 5) are not able to encode the data in a lossless way. This is not so surprising, our dataset is already a low dimension one with little room for compression. In this case, using a VAE to try to compress the data is pointless but can help reduce the complexity of the distributions nonetheless. Representing the dataset using only Normal distributions is an improvement over non-standard complex ones. We can also note that the higher the number of neurons, the lower the loss ; until a certain point (above 64) where the improvement plateaus. With 4 and 8 neurons, the model is not capable of reconstructing the data, no matter the number of latent dimensions. None of the validation loss ever increased, meaning that the models are far from overfitting (even 2x128 neurons is way too little to learn a complex distribution of nearly 80,000 points)

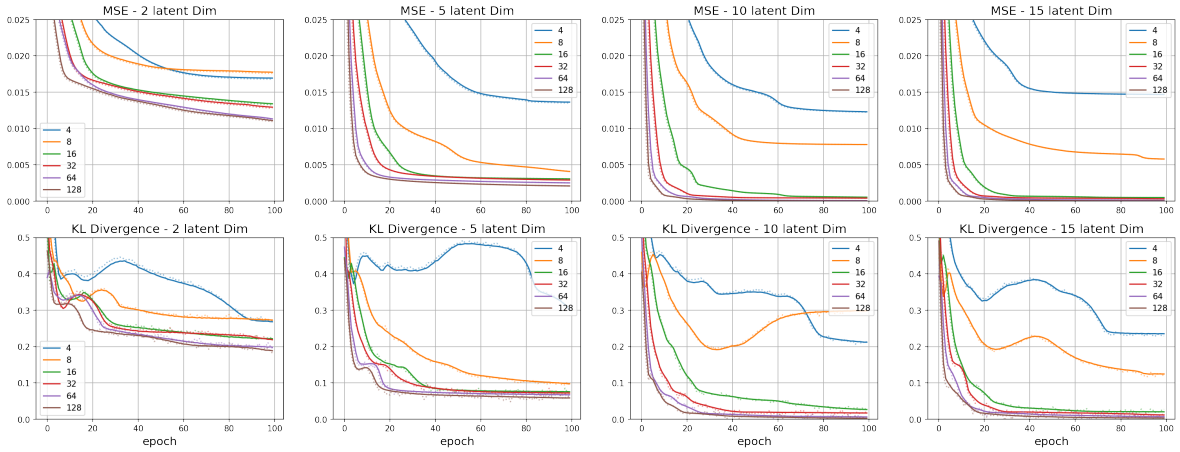

Figure 19: Training curves of VAEs of multiple sizes and multiple latent dimensions
Top row : Mean Squared Error | Y-axis : KL-divergence

Figures 20 and 21 show respectively the reconstruction of the validation set and 2,000 points sampled from the learned distribution with a 10 dimensional latent space, that or each feature and each number of neurons tested. As we saw earlier, when the model is too small, it cannot learn to project the data properly. The effect of the model

size on the quality of the output is best seen on the second feature ("month"), where the lines become sharper and sharper to represent the categorical distribution. The sampling, however, is not great ; it follows very loosely the training distribution (this is probably due to an error in the implementation that we could not investigate before the due date). (The other plots for 2, 5, and 15 latent dim are available in the appendix)

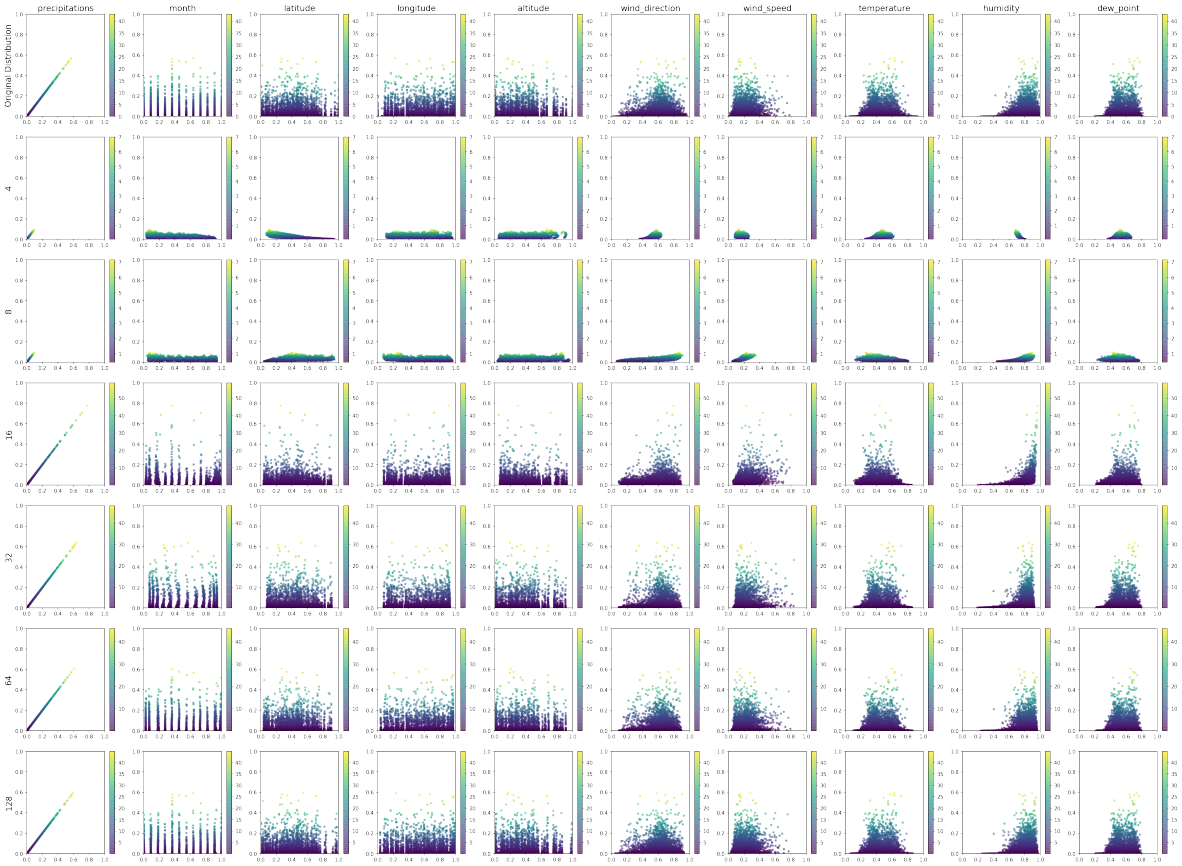

Figure 20: Reconstruction of the validation set for a 10 dimensions latent space
X-axis : individual features | Y-axis : Number of neurons in the encoder/decoder

Finally, figure 22 plots the final values at the end of training of the MSE and KL divergence along with the number of parameters in each model. As we saw earlier, this confirms that models with fewer than 16 neurons or a latent dimension lower than 10, are not able to learn the distribution and the greater the number, the lower the loss. On the third plot, we can see that using a single layer for the encoder and decoder allows for a linear growth in log scale and adding dimensions has a non-negligible but small impact on the parameters size.

To go further, we could try to project the data to other dimensions of the latent space (9, 8, 7 or 6 dim) and analyze the trend of the reconstruction loss. We could also try with larger networks and more layers to see if it can counter-balance the lossy

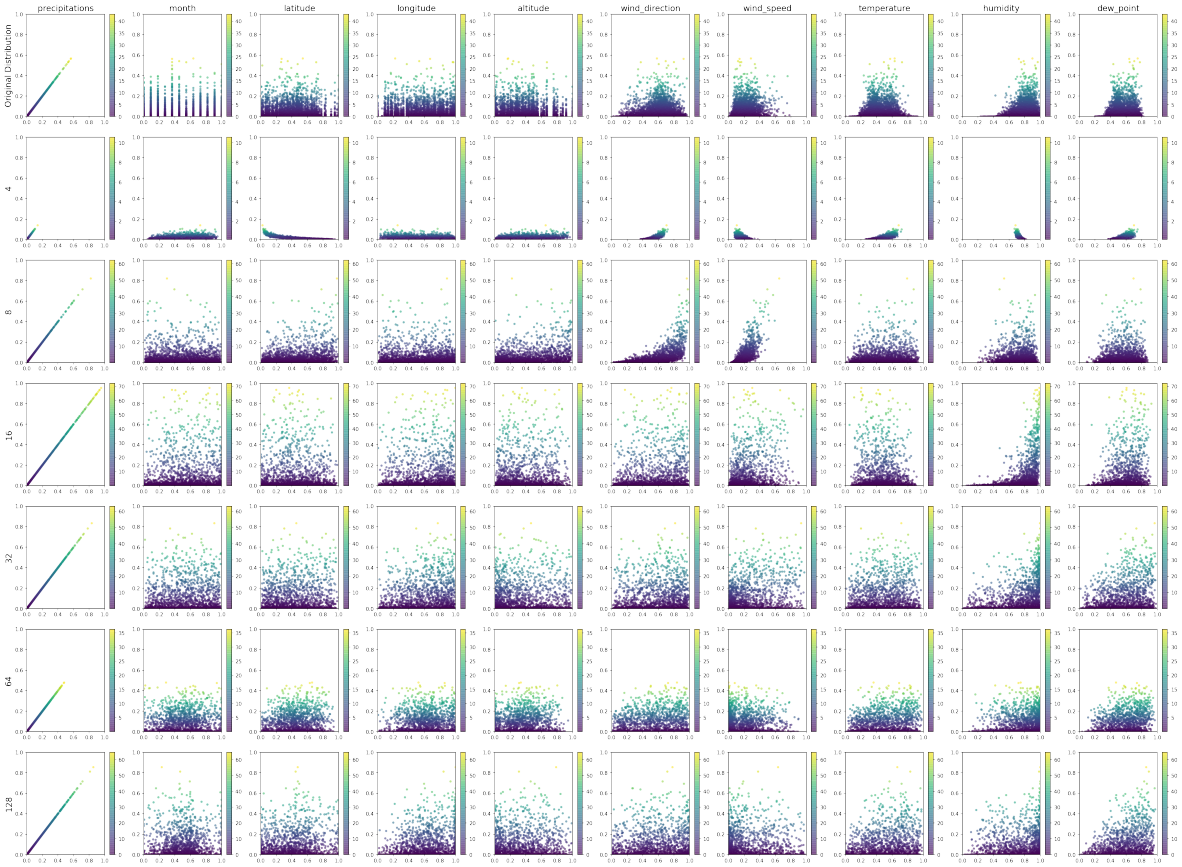

Figure 21: Sampled data from a 10 dimensions latent space
X-axis : individual features | Y-axis : Number of neurons in the encoder/decoder

compression (the blue and the orange curves in fig. 22 are still decreasing at 128 neurons).

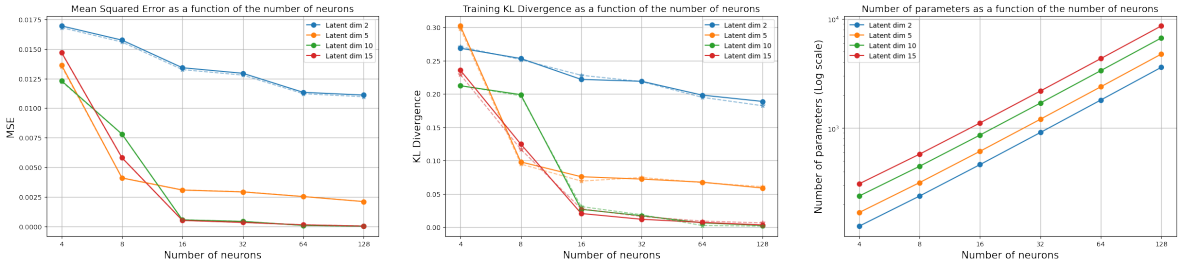

Figure 22: Mean Squared Error, KL-divergence score and Number of Parameters as a function of the number of neurons

6 Conclusion

In conclusion, we first saw that we can find the parameters of a normal distribution with Gibbs sampling, and create new samples that follow the distributions of our features.

In a second part, probabilistic models (in our case NUTS), appears to be efficient to impute missing data, but is computationally slow for large dataset like ours. Moreover, when sampling predictions from the posterior, the algorithm is sensitive to correlation in our case, thus if the data distribution has been modified by imputations, it will be reported on the posterior and thus on the sampling.

In a third part, we saw the efficiency and weaknesses of the Expectation Maximization algorithm. We saw that the EM algorithm is very sensitive to initial conditions. When initialized with the good parameters, the EM algorithm is very efficient en converges within the order of magnitude of 10 iterations, both in 1 and 2 dimensions. We saw some ways to improve the algorithms we proposed and possible ways to use the acquired information.

Finally, we saw that Variational Auto-Encoders improve on classical Auto-Encoders by adding a regularized prior to the latent space, which allows it to have the required regularity for generating new samples. Our dataset is already too low dimensional to be compressed using this method but can help reduce the complexity of the distribution by modeling it using only Normal distributions.

References

[bay, a] Bayesian imputation.

[bay, b] Bayesian regression using numpyro.

[twd,] Towarddatascience.

[Bingham et al., 2018] Bingham, E., Chen, J. P., Jankowiak, M., Obermeyer, F., Pradhan, N., Karaletsos, T., Singh, R., Szerlip, P., Horsfall, P., and Goodman, N. D. (2018). Pyro: Deep Universal Probabilistic Programming. *arXiv preprint arXiv:1810.09538*.

[Bonneel et al., 2016] Bonneel, N., Peyré, G., and Cuturi, M. (2016). Wasserstein barycentric coordinates: histogram regression using optimal transport. *ACM Trans. Graph.*, 35(4):71–1.

[Hoffman et al., 2014] Hoffman, M. D., Gelman, A., et al. (2014). The no-u-turn sampler: adaptively setting path lengths in hamiltonian monte carlo. *J. Mach. Learn. Res.*, 15(1):1593–1623.

[Kingma and Welling, 2014] Kingma, D. P. and Welling, M. (2014). Auto-encoding variational bayes.

[Kramer, 1991] Kramer, M. A. (1991). Nonlinear principal component analysis using autoassociative neural networks.

[Moon, 1996] Moon, T. K. (1996). The expectation-maximization algorithm. *IEEE Signal processing magazine*, 13(6):47–60.

[Phan et al., 2019] Phan, D., Pradhan, N., and Jankowiak, M. (2019). Composable effects for flexible and accelerated probabilistic programming in numpyro. *arXiv preprint arXiv:1912.11554*.

[Sayak, 2020] Sayak, P. (2020). Reparameterization trick in variational autoencoders.

[Stuart Geman, 1984] Stuart Geman, D. G. (1984). Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-6:721 – 741.

Appendix

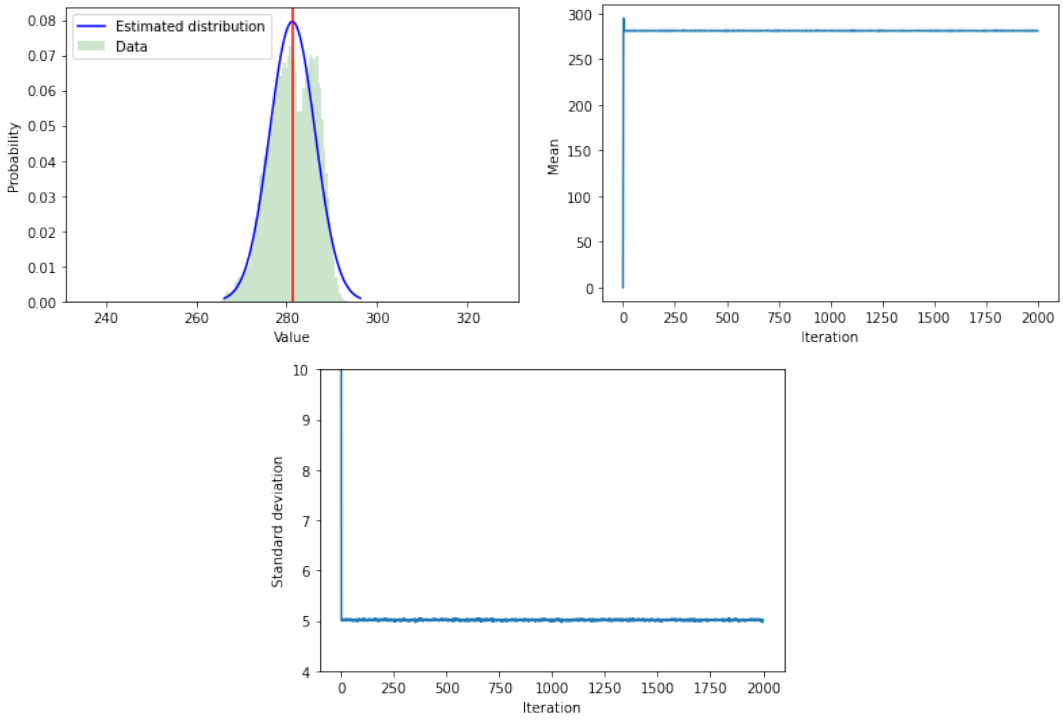

Figure 23: Gibbs sampling : find μ and σ on dew point distribution

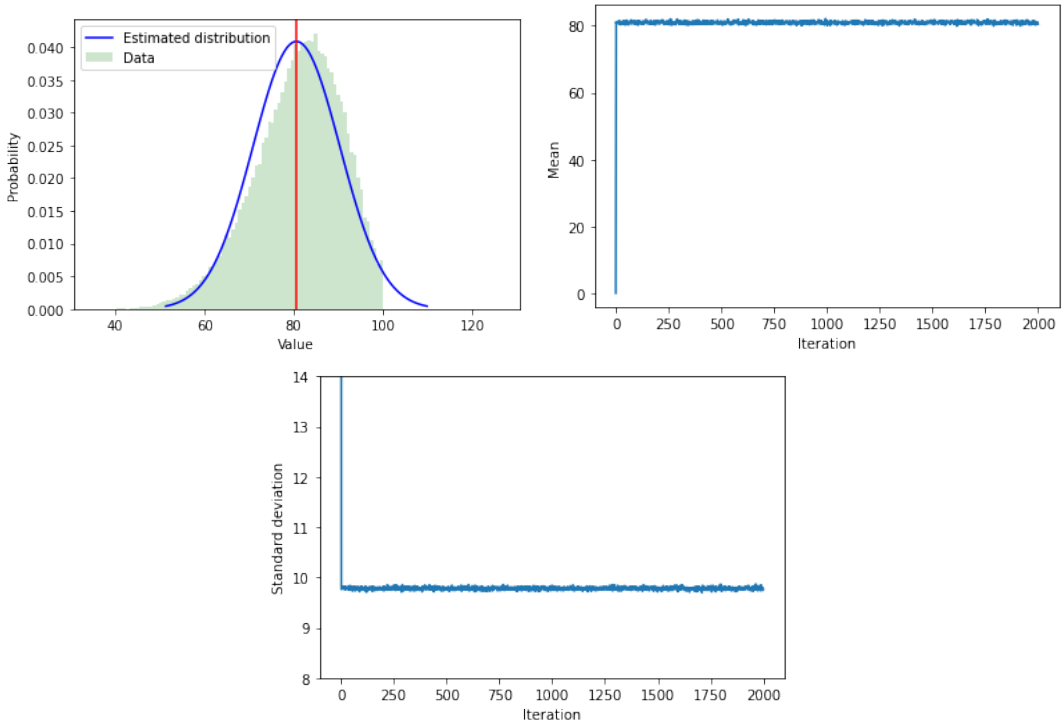

Figure 24: Gibbs sampling : find μ and σ on humidity distribution

```
# p(x_0|x_1)
mu = mean[0] + (cov[0,1] * (new_point[1] - mean[1]))/cov[1,1]
sigma = np.sqrt(cov[0, 0]-(cov[0,1]**2)/cov[1,1])
new_point[0] = np.random.randn()*sigma + mu

# p(x_1|x_0)
mu = mean[1] + (cov[1,0] * (new_point[0] - mean[0]))/cov[0,0]
sigma = np.sqrt(cov[1, 1]-(cov[1,0]**2)/cov[0,0])
new_point[1] = np.random.randn()*sigma + mu
```

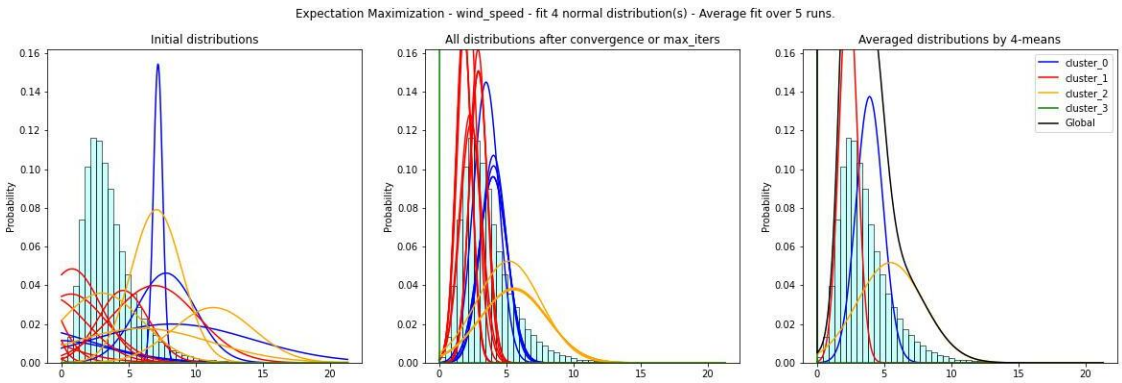

Figure 25: Example of distribution where the averaging is very useful and works well.

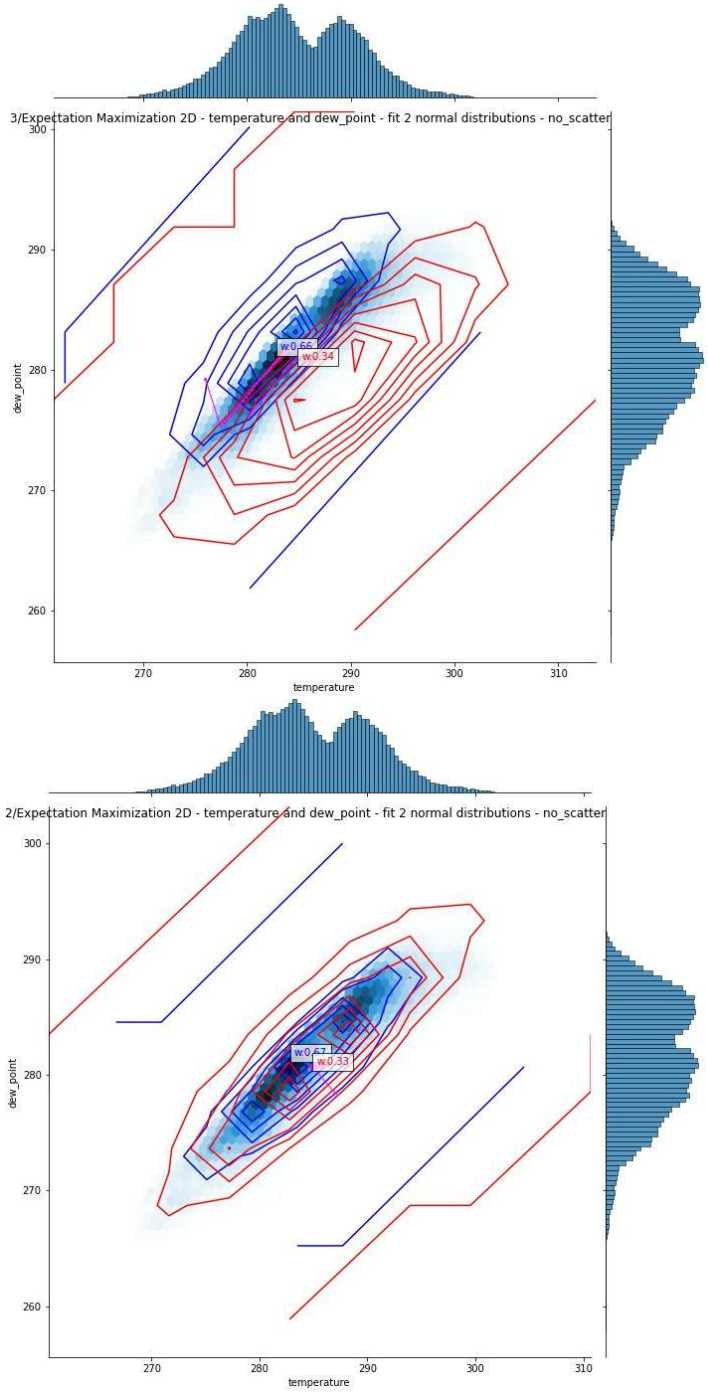

Figure 26: Example of 2D histogram where the first run of the EM algorithm (above) didn't give good results but the second one (below) describes the distribution very accurately.

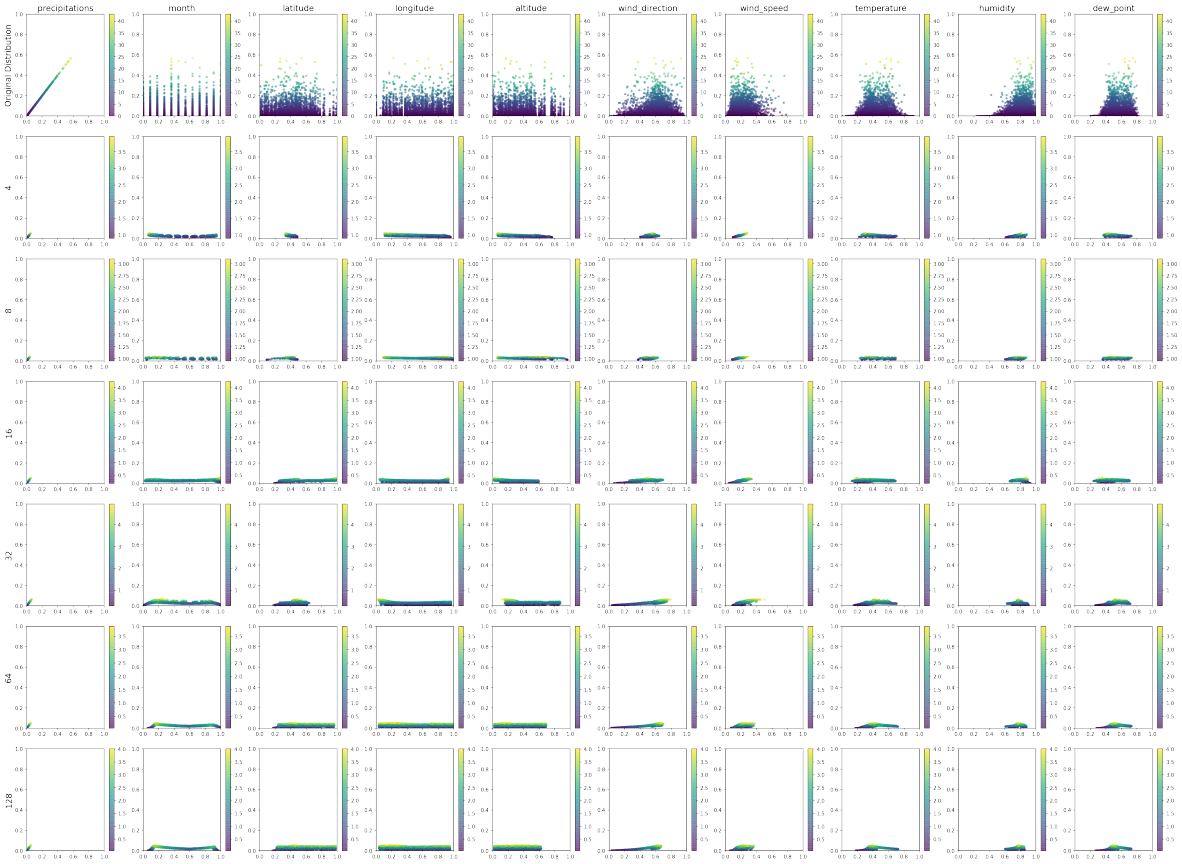

Figure 27: Reconstruction of the validation set for a 2 dimensions latent space
X-axis : individual features | Y-axis : Number of neurons in the encoder/decoder

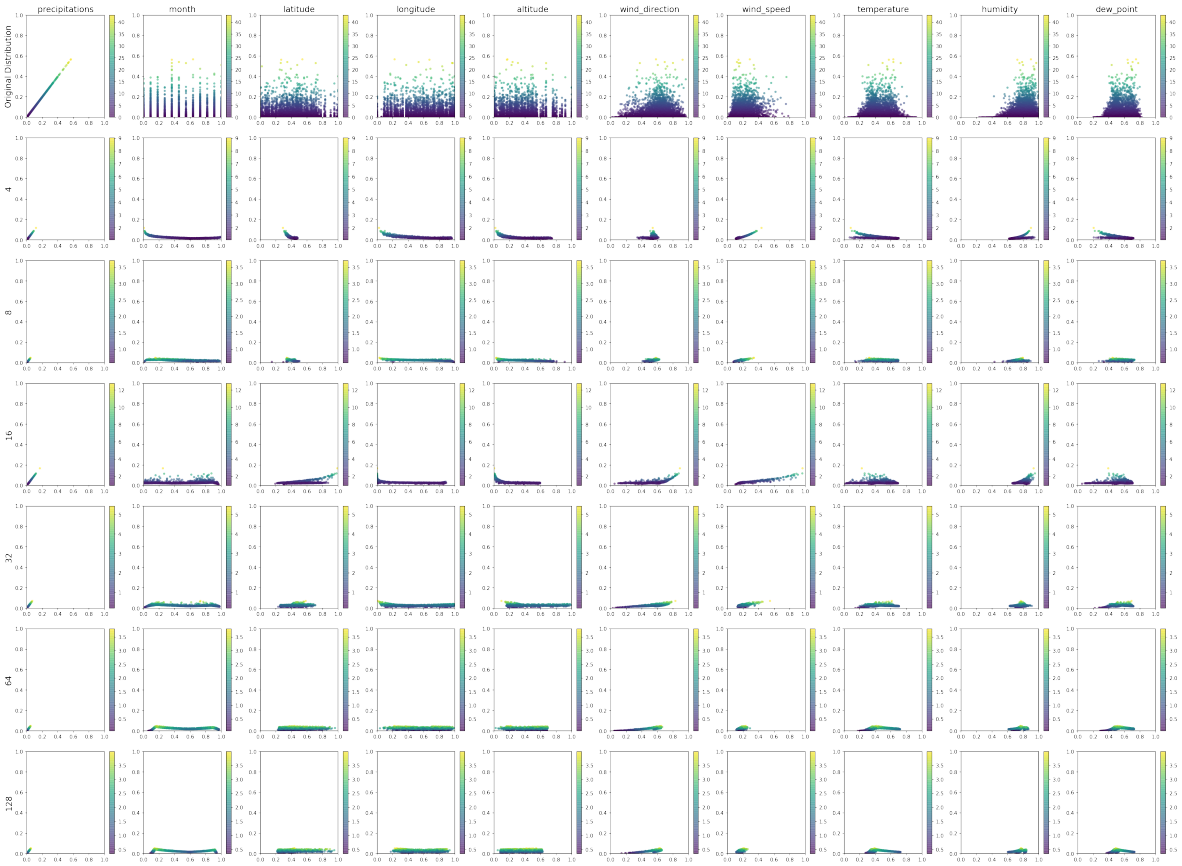

Figure 28: Sampled data from a 2 dimensions latent space
X-axis : individual features | Y-axis : Number of neurons in the encoder/decoder

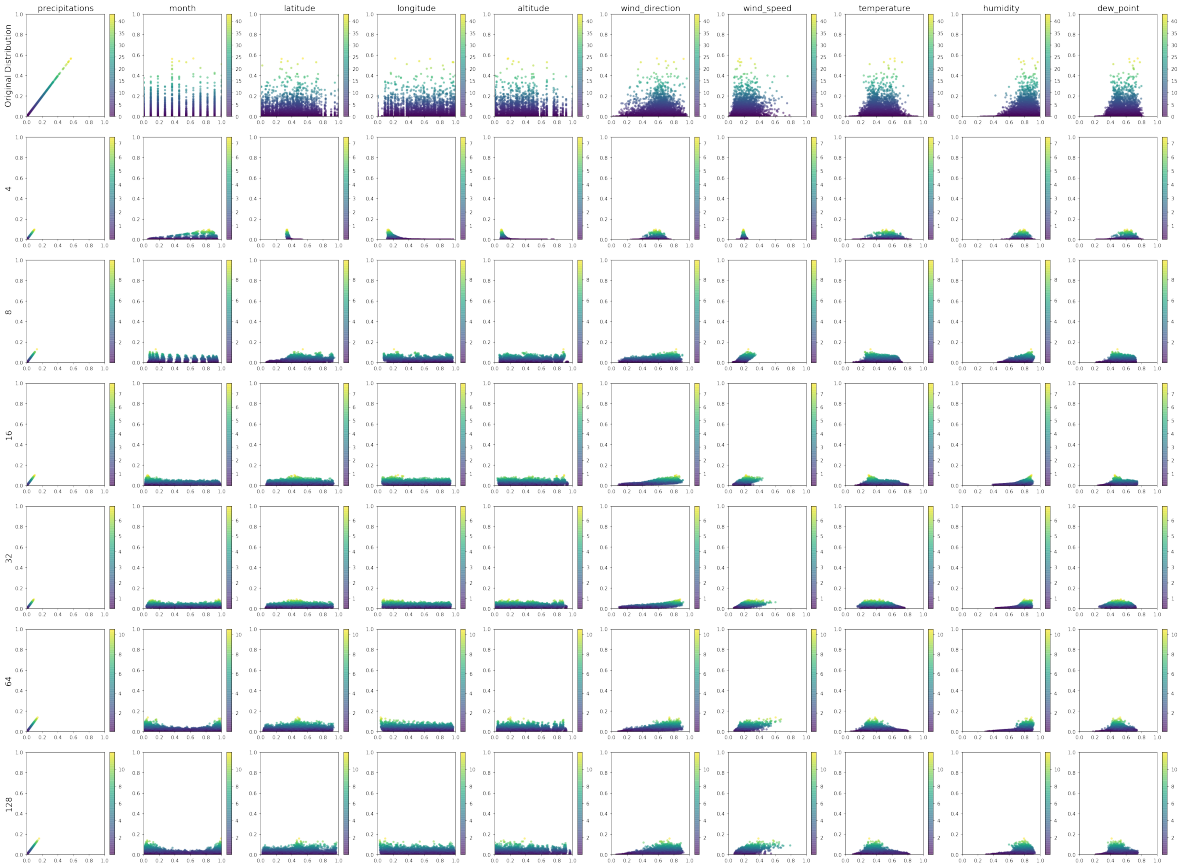

Figure 29: Reconstruction of the validation set for a 5 dimensions latent space
X-axis : individual features | Y-axis : Number of neurons in the encoder/decoder

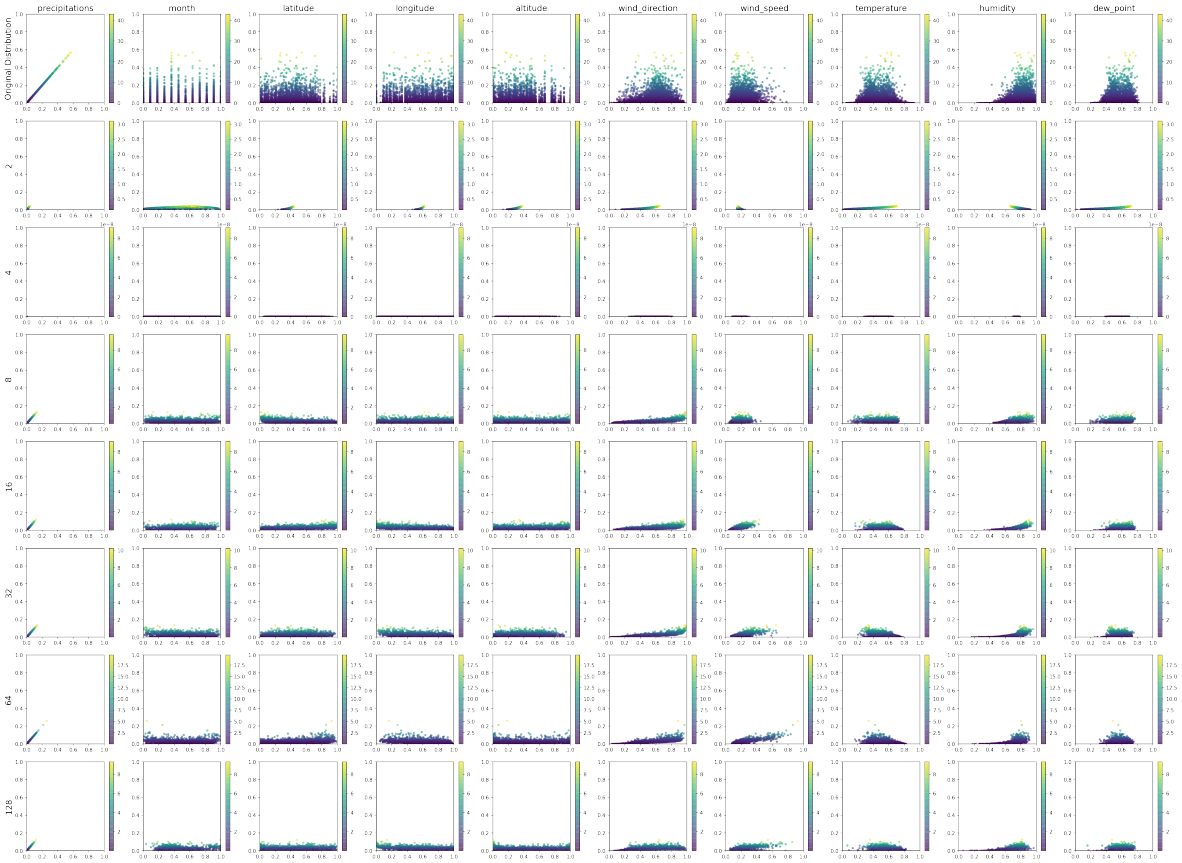

Figure 30: Sampled data from a 5 dimensions latent space
X-axis : individual features | Y-axis : Number of neurons in the encoder/decoder

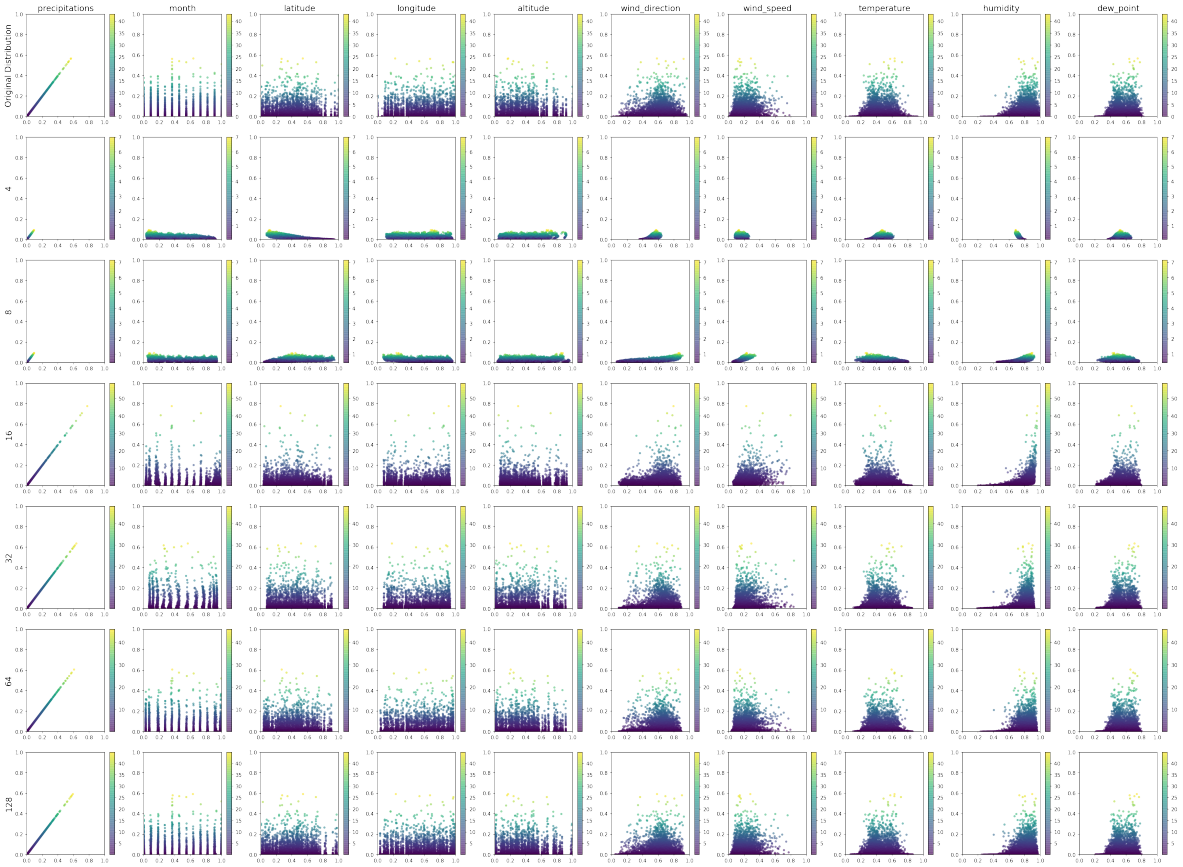

Figure 31: Reconstruction of the validation set for a 10 dimensions latent space
X-axis : individual features | Y-axis : Number of neurons in the encoder/decoder

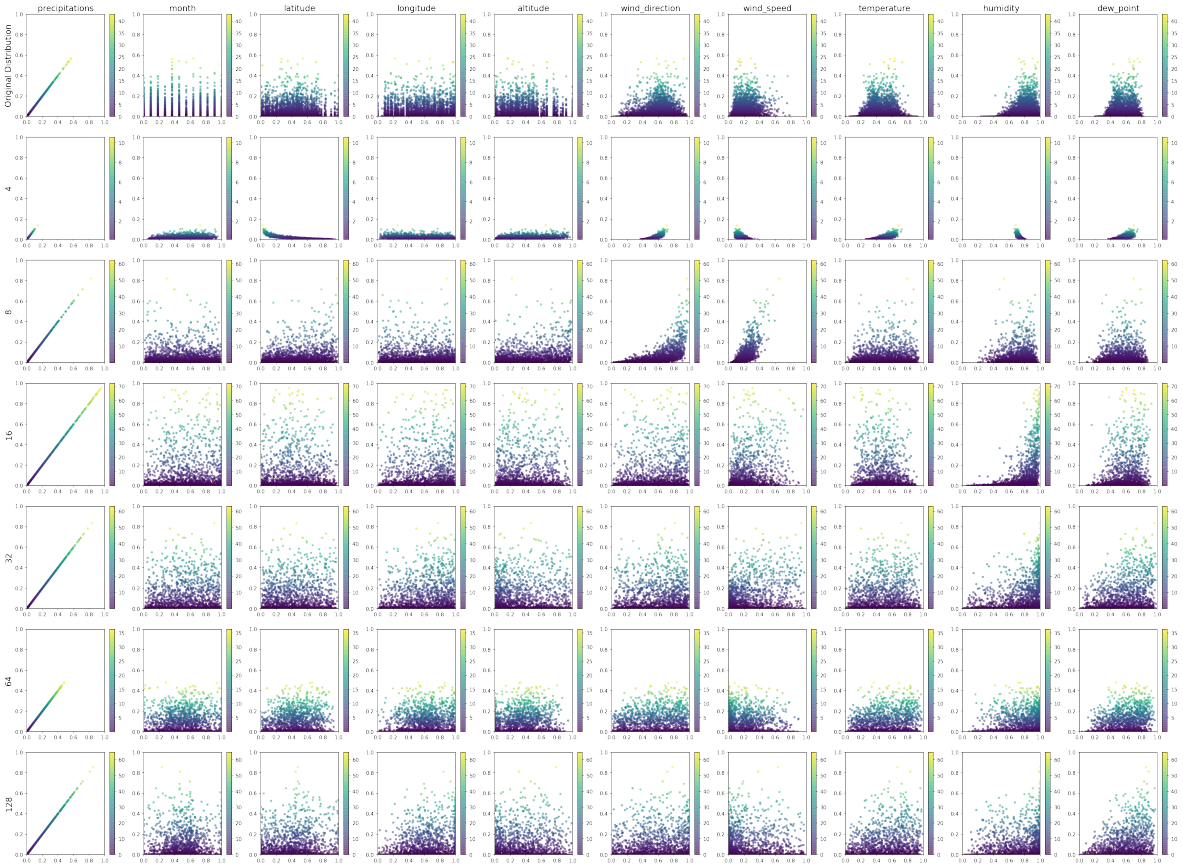

Figure 32: Sampled data from a 10 dimensions latent space
X-axis : individual features | Y-axis : Number of neurons in the encoder/decoder

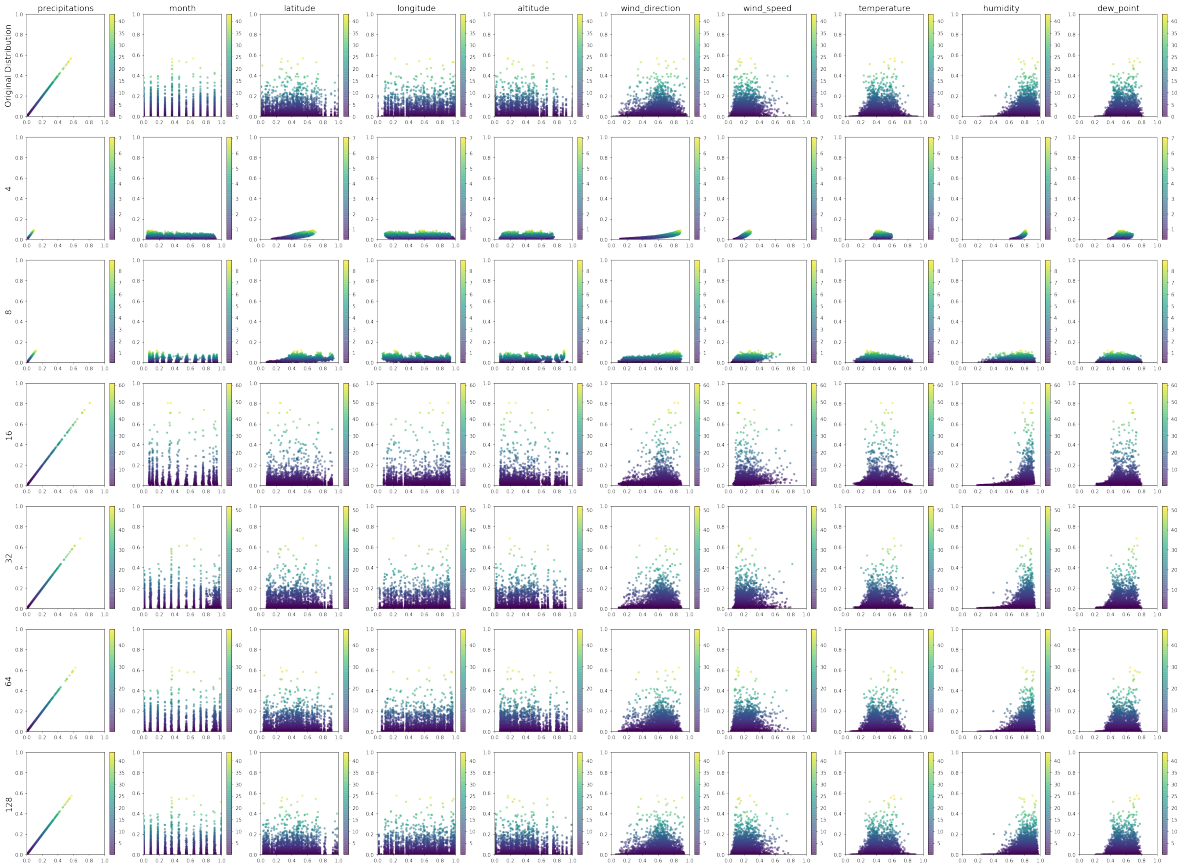

Figure 33: Reconstruction of the validation set for a 15 dimensions latent space
X-axis : individual features | Y-axis : Number of neurons in the encoder/decoder

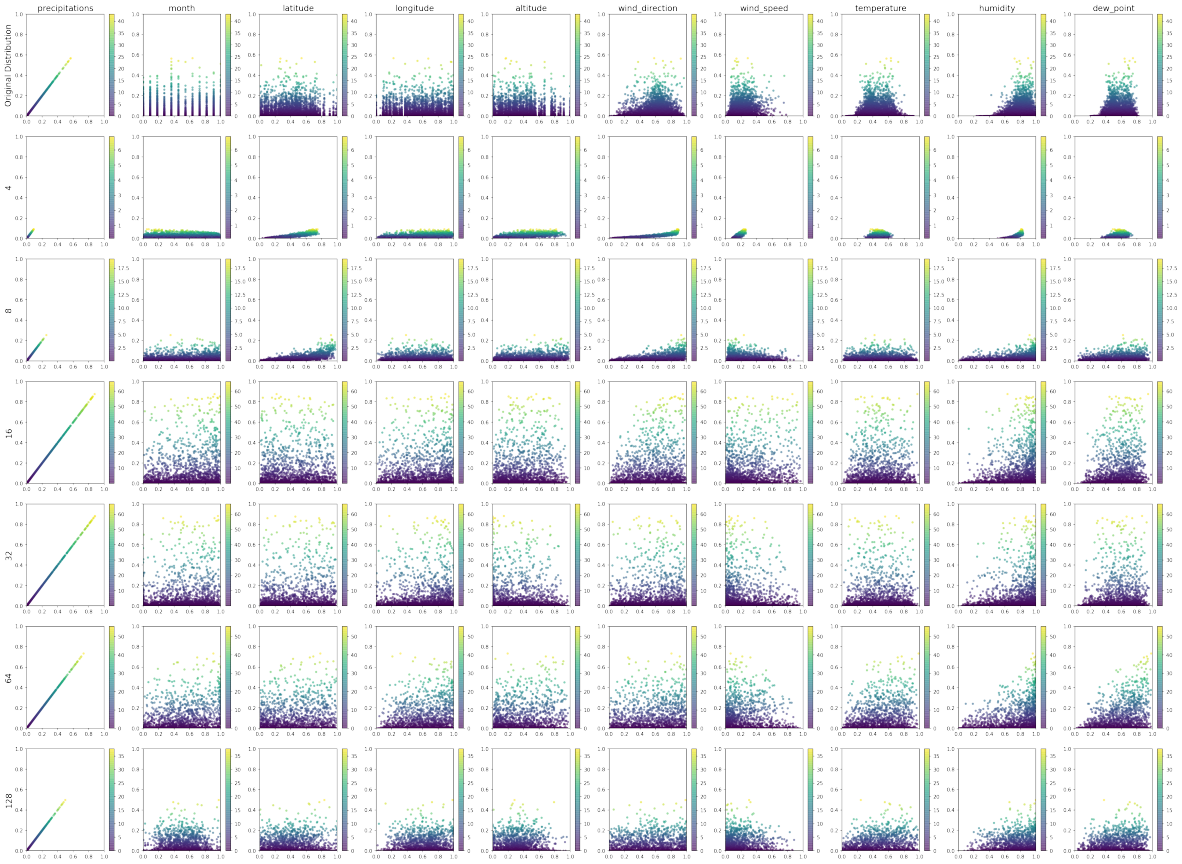

Figure 34: Sampled data from a 15 dimensions latent space
X-axis : individual features | Y-axis : Number of neurons in the encoder/decoder