Report for exercise 3 from group C

Tasks addressed: 4
Authors: Aleksi Kääriäinen (03795252)
Danqing Chen (03766464)
Julia Xu (03716599)
Joseph Alterbaum (03724310)

Last compiled: 2024–09–27

The work on tasks was divided in the following way:

Aleksi Kääriäinen (03795252)	Task 1	25%
	Task 2	25%
	Task 3	25%
	Task 4	25%
Danqing Chen (03766464)	Task 1	25%
	Task 2	25%
	Task 3	25%
	Task 4	25%
Julia Xu (03716599)	Task 1	25%
	Task 2	25%
	Task 3	25%
	Task 4	25%
Joseph Alterbaum (03724310)	Task 1	25%
	Task 2	25%
	Task 3	25%
	Task 4	25%

Report on task 1, Principal component analysis

1 Principal Component Analysis

The first task captures the use of an algorithm called *Principal Component Analysis (PCA)*. There are three main parts in this task consisting of implementing the algorithm, then applying it to a given image data, and finally visualizing and analyzing a new and bigger data set.

In the end of this chapter, a quick feedback about this exercise's workload and achievements are stated.

1.1 Part I: Implementing PCA

The first part of this task is split into six steps to perform the principal component analysis and plot its results. The data set that is being analyzed is given in `pca_dataset.txt`.

Firstly, the text file is being loaded to be able to handle its contents. Secondly, the mean of the data is computed which is then later used to get the center point of the data set. Thirdly, the data set is centered by its mean which is then taken to perform the singular value decomposition (SVD). The built-in function `numpy.linalg.svd()` is used with a parameter `full_matrices` set to `False` to compute the SVD. After that, the principal components correspond with the rows of the resulting matrix `V_t`.

In step five, the results are plotted: Figure 1 shows the plotted data set with directions of the principal components. The lines are calculated by getting the center point of the data set (origin of direction lines) and the coordinates of the principal components (PC) found in the variable `V_t`. The lines are then elongated by a factor of two for better visualization.

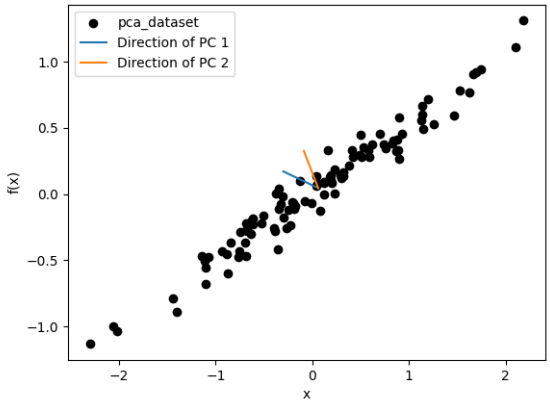


Figure 1: Data set with directions of the two principal components.

Finally, in the last step, the energies of two principal components are calculated which correspond to the percentage of the total energy also known as explained variance (see formula 1).

$$\text{Percentage of Total Energy of (only) i-th PC} = \left(\frac{1}{\text{trace}(S^2)}{\sigma_i}^2\right) 100 \tag{1}$$

where:

S = matrix with singular values in diagonal
σ_i = i-th singular value

Table 1 shows the values of the calculated energies. The first energy number is very high whereas the second one is closer to zero which leads to the interpretation that the first principal component has a much higher impact on the data than the second one. This is also confirmed by the plotted results in figure 1 where it is seen that the first component (x-coordinate) has a wider value range than the y-coordinate.

PC	Energy
1	99.302
2	0.698

Table 1: Energies of first two principal components (rounded to three decimals).

1.2 Part II: Applying PCA to Image

The second part of the task applies the algorithm of PCA to a given image data set. The assignment is to plot the raccoon image in different truncations and evaluate their information and energy loss compared to the amount of PCs used.

1.2.1 Visualization of Raccoon Image in Different Truncations

Figure 2 shows the final visualization of the given raccoon image (`scipy.misc.face(gray=True)`) in various settings. The images are constructed by truncating the matrices resulted from SVD of the original data in the implemented method `utils.reconstruct_data_using_truncated_svd()`. The most vital part in this implementation is understanding the right shapes of the different matrices for correct truncation and matrix multiplication.

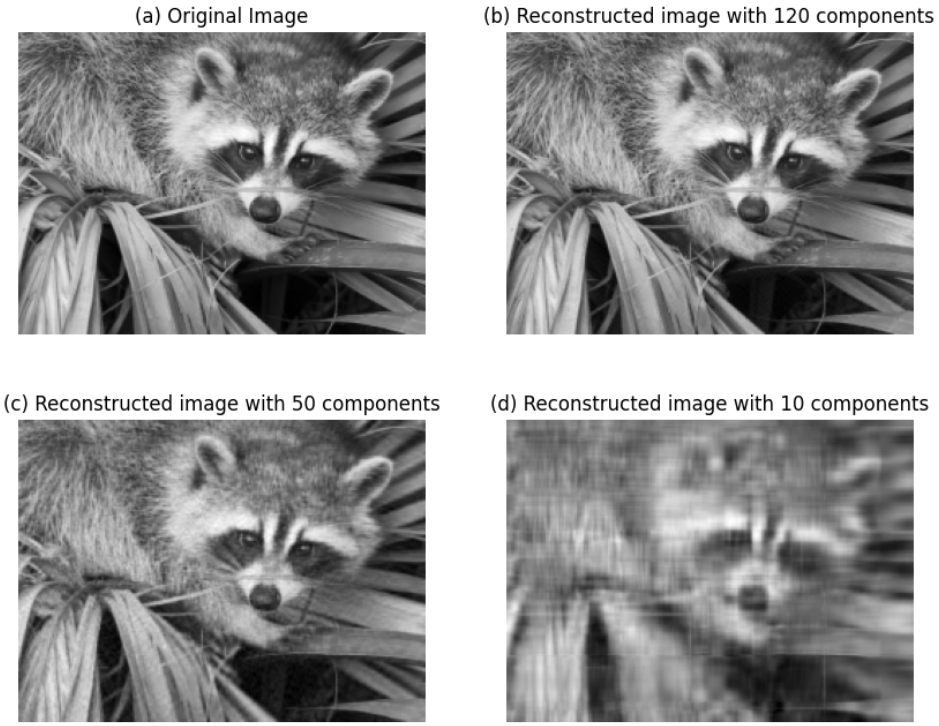


Figure 2: Visualization of reconstructions of the image with (a) all, (b) 120, (c) 50 and (d) 10 principal components.

1.2.2 Evaluation and Interpretation of Components vs. Loss

The information loss in picture (d) of figure 2 is clearly noticeable by the naked human eye. One could argue that picture (c) already shows some kind of slight blurriness. This is however rather unnoticeable compared to

picture (d). Therefore the number for visible information loss is **10 components**.

The number of principal components that lead to an energy loss through truncation smaller than 1% is **26** which is drastically lower than the amount of components in the original image. Since the computed value is between the thresholds of pictures (c) and (d) in figure 2 (50 <26 <10), this result also corresponds to the above observed conclusions concerning the information loss seen by the naked human eye. Ten components would be too less for not compromising the visualization and 50 components would be more than actually needed to get a desired energy loss of smaller 1%.

1.3 Part III: Visualization and Analysis of Data

The third part of this task deals with a different data set again. Here, the focus lies on plotting trajectories from a given data set introducing coordinates of two pedestrians.

1.3.1 Visualization of the path of the first two pedestrians in 2D space

The given method `utils.visualize_traj_two_pedestrians()` is used to visualize the path of the first two pedestrians in 2D space . It takes two 2D arrays as arguments each containing the coordinates for a pedestrian. There is a helper function `utils.extract_positions()` implemented which returns two 2D arrays to extract the right coordinates of pedestrian 1 and 2 from the given data set `data_DMAP_PCA_vadere.txt`. The basic idea of that helper method is to flatten the loaded data set and extract the first 2000 entries for pedestrian 1 and the following 2000 entries for pedestrian 2, and to then save the corresponding x- and y-coordinates in their respective 2D arrays.

Figure 3 shows the visualization of the two pedestrian based on the original data given in above mentioned data file.

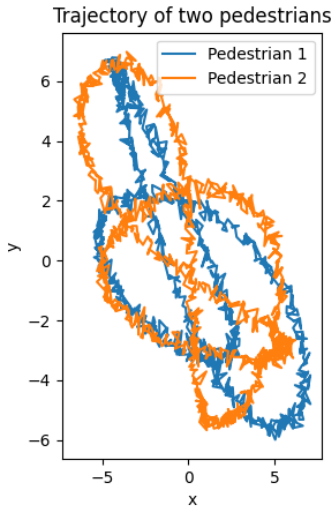


Figure 3: Trajectory of pedestrians 1 and 2 based on the original data.

After visualizing the original data set, it can be seen that the plot has various overlapping points. To get a clearer visualization of the pedestrians' trajectories the data is linearly decomposed by applying the PCA algorithm. After computing SVD on the original extracted data, the data is being reconstructed by truncating SVD and retaining only two principal components. This leads to a low-rank approximation of $rank(A) = 2$ with A being the reconstructed data matrix.

This reconstructed data set is then being used to extract the coordinates of the pedestrian again and then to plot their respective trajectories. Figure 4 shows the visualization on the linearly decomposed data set.

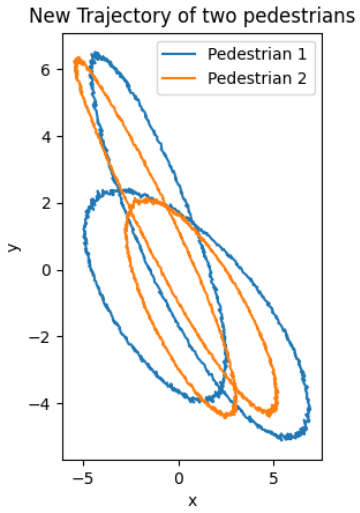


Figure 4: Trajectory of pedestrians 1 and 2 based on the changed data.

1.3.2 Evaluation

By using the method `utils.compute_cumulative_energy()` the computed total energy of two principal components is calculated resulting in **84.93** (rounded by two decimals). Therefore it is not enough to have two principle components to capture most of the energy (greater 90%) of the data set. The number of needed principal components is calculated by the function `utils.compute_num_components_capturing_threshold_energy()` and results in **three necessary PCs** to capture most of the energy.

1.4 Task Feedback

1.4.1 (a) an estimate on how long it took you to implement and test the method

Part I of this task took approximately **five hours** to finish. Most of that time is due to understanding the right structure of the data and due to wrong interpretation of total energy at the beginning. The correct implementation itself was quite fast then after one has understood the definitions and structure of the data.

Part II took longer than the previous part and took approximately of **eleven hours**. This is the part were most of the implementation needed to be done in the file `utils.py`. The most time consuming task was once again implementing the methods concerning *energy* which is due to the already mentioned misunderstanding in the beginning.

Part III was the quickest to finish and took roughly **three hours**.

[Exkursion]: Since a bonus from task 2 (bonus (3) datafold software) was done by the same person working on this task its feedback is placed here as well. The bonus took roughly **four to five hours** which is mostly due to reading into the topic of Diffusion Maps to get a better grasp of understanding and looking into some of the implementation of the code provided by the software which was asked to be used in this bonus task.

1.4.2 (b) how accurate you could represent the data and what measure of accuracy you used

Accuracy in data representation is differently defined here. Whilst in part one and two of this task the data is clearly visible by the human eye, the last part of this task shows challenges in transparent visualization. It can be seen that the lines of the trajectories become slimmer, thinner, and even less overlapping (orange plot).

1.4.3 (c) what you learned about both the data set and the method (which is probably different from what the machine learned)

Applying PCA to two-dimensional data as in the first part of this task is much easier and less complex compared to applying the algorithm on higher-dimensional data. Images deal with a larger dimension which is why matrices from SVD need to be truncated to get a low-rank approximation. This however also leads to a more significant

change in visualization when using different amounts of principal components which were seen in the second part of this task.

Report on task 2, Diffusion Maps

2 Diffusion Maps

2.1 Implementation of Diffusion Maps

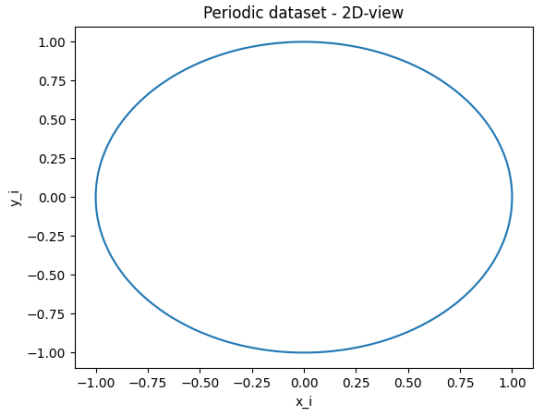
As suggested in the lecture we implemented the Diffusion Map-algorithm according to the paper by Berry et al. [1]. The corresponding code can be found in the module *utils.py* within the folder "*src/task2_diffsuion_maps*" of the repository. Apart from the SciPy-class *scipy.spatial.KDTree* and standard NumPy-packages for matrix operations and eigenvalue-decompositions no further libraries were used.
Firstly, we computed the distance matrix D by indexing all entries of the data matrix X with a KD-tree using the SciPy-library *scipy.spatial.KDTree*. Based on this KD-tree on X we can then create a sparse distance matrix, disregarding all neighbours with a distance greater than the threshold, by using the *sparse_distance_matrix*-method of the KD-Tree on itself. Subsequently, the normalised kernel-matrix K is computed by setting the epsilon value, computing the unnormalised kernel matrix W, corresponding to a Gaussian kernel (cp. [1]), and normalising it to acquire W. Finally, the wanted eigenvalues and eigenvectors, corresponding to the value of the eigenfunctions for each of the N samples, are obtained.

2.2 Part 1 – Diffusion Maps & Fourier analysis

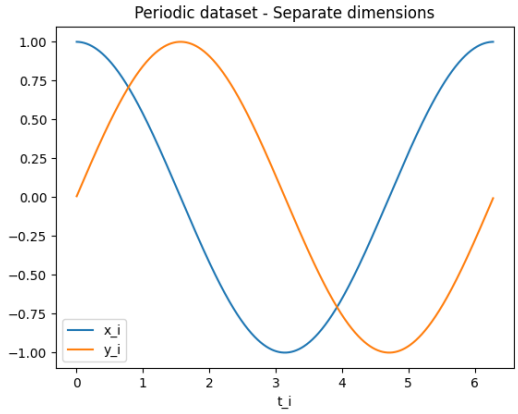
2.2.1 Results

After initialisation of 1000 samples of the periodic dataset with the *create_periodic_dataset*-function of the module "*src/task2_diffusion_maps/utils.py*", the structure of the dataset can be explored. This is achieved by plotting the samples once in the 2-dimensional embedding given by the specification, cp. Fig. 5a, and once separated for each dimension, cp. Fig. 5b. In the first plot we can see that the dataset represents a circle in 2-dimensional Euclidean space. Subsequently, the second visualisation shows that the 2D-circle corresponds to a sine- and cosine-function in the respective dimensions.

Figure 5: Periodic dataset specified in Task 2.1



(a) Visualisation of the dataset in 2 dimensions (x & y)



(b) Visualisation of the dataset in separated dimensions

Additionally, the task was to compute the first 5 eigenfunctions of the data. For this, we used the function *diffusion_map* in the module "*src/task2_diffusion_maps/utils.py*". While the 0-th or constant eigenfunction is normally not of interest, we included it in the visualisation since it becomes important for the comparison to Fourier analysis when considering the data as a signal in the next subsection. The plot of the first five eigenfunctions over the parameter t_i can be seen in Fig. 6. One result from the plot is that the eigenfunctions follow a specific pattern: All odd eigenfunctions are negative sine-functions over t_i with increasing frequencies (cp. ϕ_1, ϕ_3 & ϕ_5) while all even eigenfunctions are negative cosine-functions (cp. ϕ_2 & ϕ_4). This is interesting

because it corresponds to the given embedding of the manifold where the first dimension (x_i in Fig. 5b) resulted in a cosine-function and the second dimension (y_i in Fig. 5b) in a sine-function over t_i.

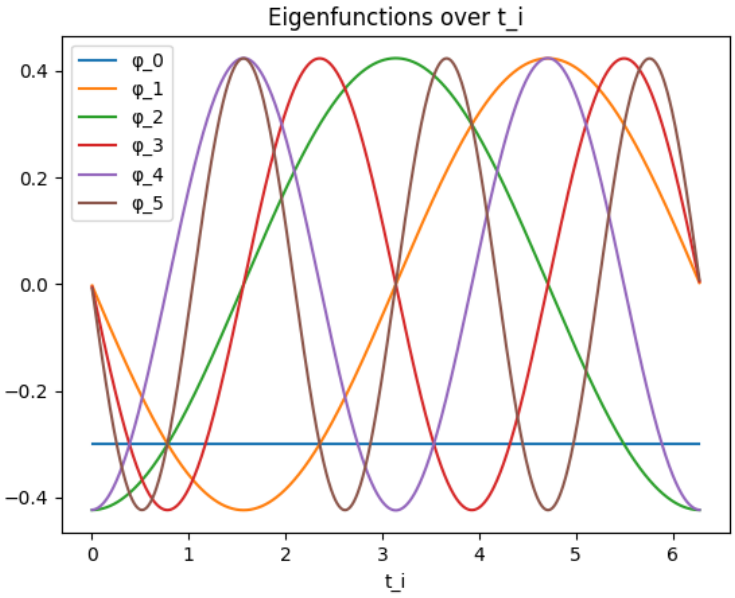


Figure 6: First 5 eigenfunctions of X (& 0th/Constant eigenfunction)

2.2.2 Bonus: Comparison to Fourier-analysis of signal X

While Diffusion Maps are an algorithm to find optimal non-linear coordinates to represent a manifold in a new embedding, Fourier analysis aims at representing general functions by sums of trigonometric functions (or coordinates), e.g., sine and cosine. One possible way of Fourier analysis is the expansion in a Fourier series which can be done in exponential form (N goes to infinity):

$$F(x) = \sum_{k=-N}^{N} A_k \exp(i2\pi \frac{k}{P} x)$$

where i is the imaginary number, A_k are coefficients and $\frac{k}{P}$ is the frequency. An alternative way is the sine-cosine-formulation (again theoretically N goes to infinity):

$$F(x) = A_0 + \sum_{k=1}^{N} (A_k \sin(2\pi \frac{k}{P} x) + B_k \cos(2\pi \frac{k}{P} x))$$

The relation between these two formulations can be retrieved by looking at Euler's formula.
When interpreting our data X as a function or signal over the parameter t_i the objectives of Diffusion Maps and Fourier analysis become very similar: finding an optimal representation of the behaviour as a combination of a theoretically infinite number of new components/coordinates. The second form of Fourier series consisting of a constant term and a sum of sine- and cosine-functions is now very similar to the result we found when looking at Fig. 6. The 0-th eigenfunction is constant while the rest is either a sine or a cosine. Another similarity is the increasing frequency in the eigenfunctions from Diffusion Maps which corresponds to the frequency $\frac{k}{P}$ in the Fourier series which also rises with increasing k.

2.3 Part 2 – The Swiss Roll dataset

2.3.1 5000 data points

Firstly, we obtained the Swiss Roll-dataset with the *make_swiss_roll*-function from the *sklearn.datasets*-library. In order to inspect the 5000 samples we created a 3D-scatter-plot of the data which can be seen in Fig. 7. Accordingly, the Swiss Roll-dataset is a 2-dimensional manifold, because it can locally be approximated by a 2D-plane, which is given as a 3-dimensional embedding in 3D-Euclidean space. The visualisation shows that it is not a linear 2D-manifold however since a 2D-plane does not globally approximate its shape. Therefore, highly likely a PCA with 2 components will not yield good results.

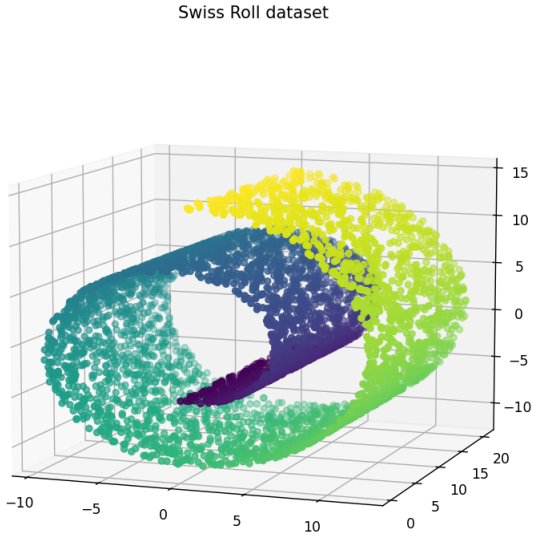
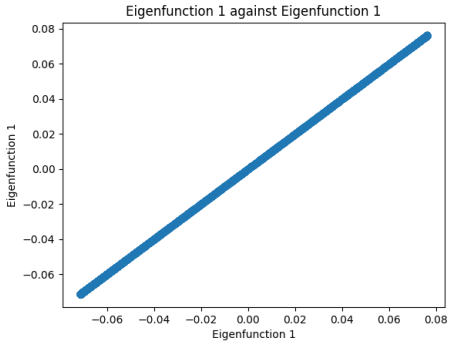


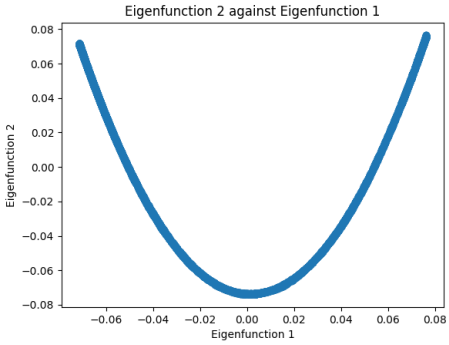
Figure 7: 5000 samples from the Swiss Roll-dataset in a scatter plot

Afterwards, we obtained the first 10 eigenfunctions of the dataset and disregarded the 0-th eigenfunction since it was constant over all samples. One important point to mention here is the performance of our Diffusion Map implementation. 5000 datapoints in 3 dimensions result in a 15000 x 15000 distance matrix and subsequently in an eigenwert-problem of the same size. Since our implementation computed a sparse distance matrix but did not store it as such, our implementation takes a few minutes to return the first ten eigenvectors on a lower grade laptop. While testing on a Desktop-PC resulted in return times of a few seconds, the storage of matrices in a sparse format, such as compressed row storage (CSR), would improve the performance significantly.
Plots of the first 10 eigenfunctions against the first eigenfunction ϕ_1 can be seen in Fig. 8. For ϕ_l to be a function of ϕ_1, every value of the latter has to map to exactly one value of the former. In the case were we plot ϕ_1 vs. ϕ_1, cp. Fig. 8a, it is obviously a linear function of itself. The second eigenfunction seems to be a quadratic function of ϕ_1 (Fig. 8c) and the third exhibits a cubic relation (Fig. 8c). In contrast, the first l where this is not the case anymore is the fourth eigenfunction (l = 4), which can be seen in 8d. This is especially visible in the region where ϕ_1 is bigger than 0.04, where much more than one value of ϕ_4 maps to a value in ϕ_1. While the fourth eigenfunction is globally still resembling a function of ϕ_1, ϕ_5 at the latest is not a function of ϕ_1 anymore.

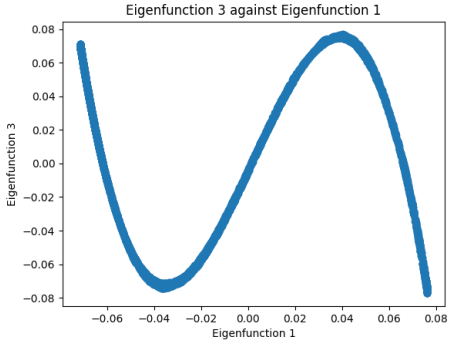
Figure 8: Plots of the first 10 eigenfunctions of the Swiss Roll-dataset with 5000 samples



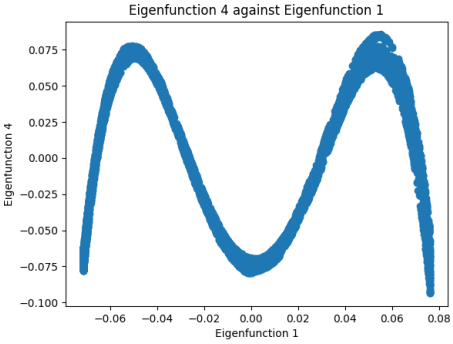
(a) Eigenfunction 1 vs. Eigenfunction 1



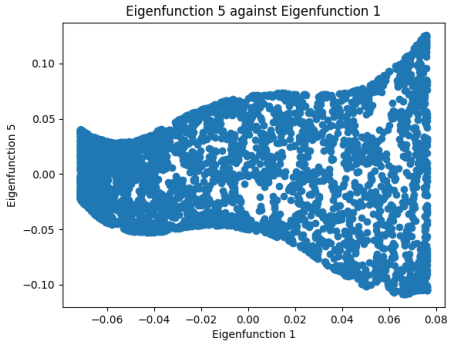
(b) Eigenfunction 2 vs. Eigenfunction 1



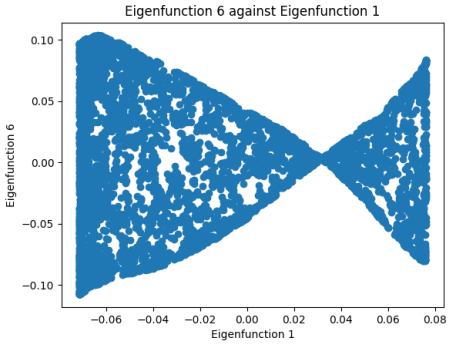
(c) Eigenfunction 3 vs. Eigenfunction 1



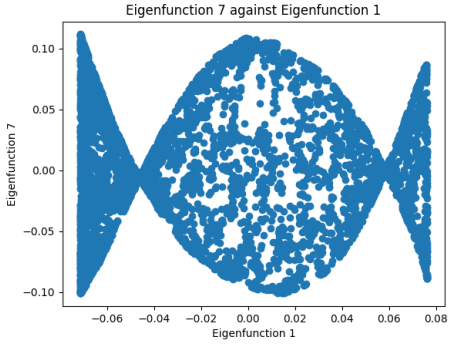
(d) Eigenfunction 4 vs. Eigenfunction 1



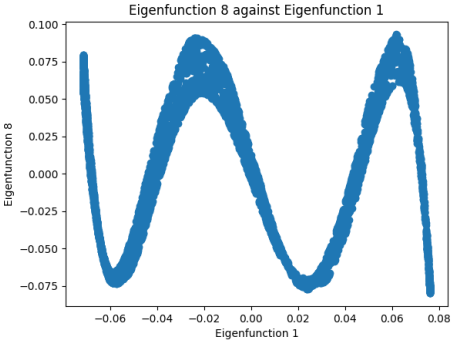
(e) Eigenfunction 5 vs. Eigenfunction 1



(f) Eigenfunction 6 vs. Eigenfunction 1

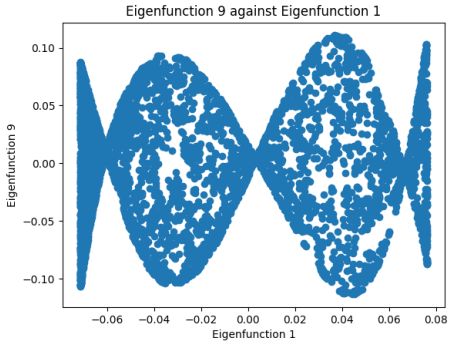


(g) Eigenfunction 7 vs. Eigenfunction 1

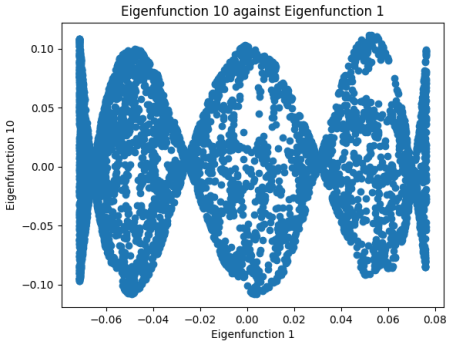


(h) Eigenfunction 8 vs. Eigenfunction 1

Figure 9: Continuation of Fig. 8



(a) Eigenfunction 9 vs. Eigenfunction 1



(b) Eigenfunction 10 vs. Eigenfunction 1

After running our code from Task 1 on the Swiss Roll-dataset we obtain the three principal components along their singular values or energies. The singular values are as follows:

$$\begin{pmatrix} 870.0405436 & 506.10724435 & 454.35203792 \end{pmatrix}$$

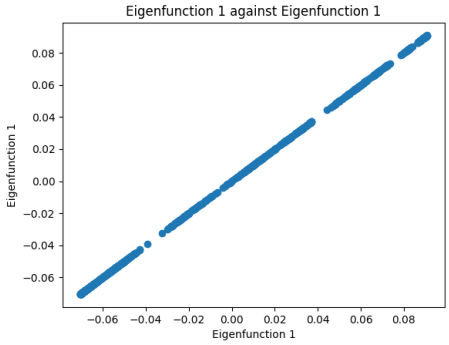
The corresponding singular vectors or principal components are the columns in the following matrix:

$$\begin{pmatrix} 0.21019498 & -0.49653967 & -0.84217957 \\ 0.97706153 & 0.13681351 & 0.16319571 \\ 0.0341884 & -0.85716417 & 0.51390732 \end{pmatrix}$$

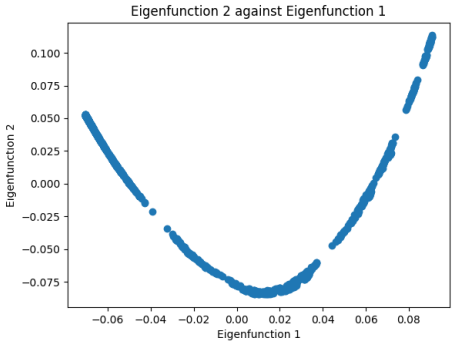
And finally the accumulated energy of the first two principal components amounts to **83.073**% of the total energy in the dataset with 5000 samples. Equivalently, 16.927% of the energy is associated to the third principal component.

Two principal components cannot represent the data properly because while the Swiss Roll is a 2D-manifold (approximated by a 2D-plane locally) it is not a 2D-plane globally. Since PCA is a linear technique finding the Euclidean coordinate system in N dimensions (N $\leq$ #dimensions of the given embedding) along which the variance in the data is maximal it can be seen as an N-dimensional hyperplane approximating the data. The Swiss Roll-dataset however is not a hyperplane, cp. Fig. 7, and therefore contains rather strong variance along all possible dimensions of the given Euclidean embedding space which can be observed when looking at the singular values of the PCA corresponding to the variance. An intuitive measure of how much variance one drops when reducing dimensionality with choosing less principal components is the relative cumulative energy compared to the given embedding. In our case we drop roughly 17% of the energy when trying to reduce dimensionality to 2 dimensions. In contrast, Diffusion Maps are not constrained to linear relations and therefore can approximate the data by non-linear N-dimensional shapes or equivalently put non-linear coordinates. This allows Diffusion Maps to find embeddings with less dimensions while at the same time keeping a higher fraction of the information about the manifold compared to PCA. Moreover, in the case where the data already describes an N-dimensional hyperplane Diffusion Maps can also find linear dependencies.

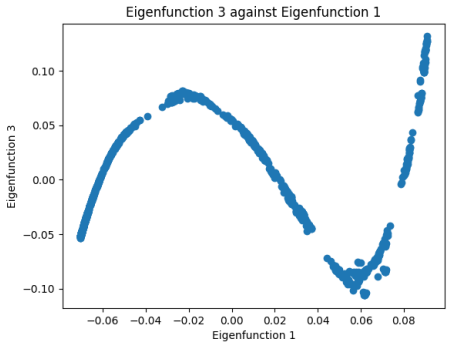
Figure 10: Plots of the first 10 eigenfunctions of the Swiss Roll-dataset with 1000 samples



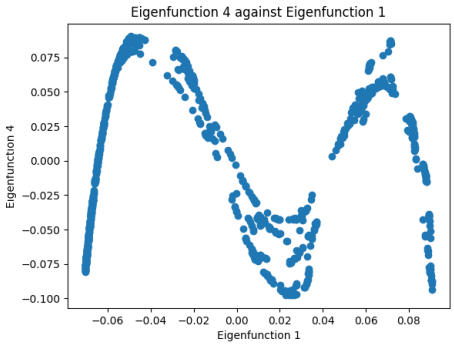
(a) Eigenfunction 1 vs. Eigenfunction 1



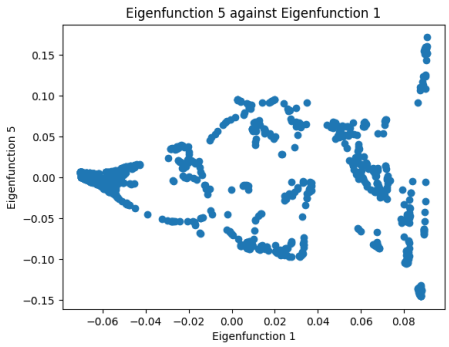
(b) Eigenfunction 2 vs. Eigenfunction 1



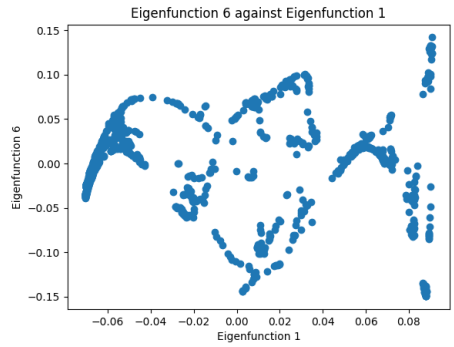
(c) Eigenfunction 3 vs. Eigenfunction 1



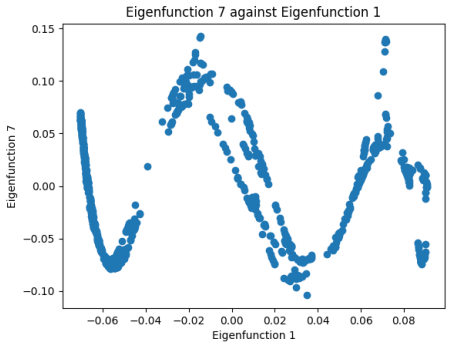
(d) Eigenfunction 4 vs. Eigenfunction 1



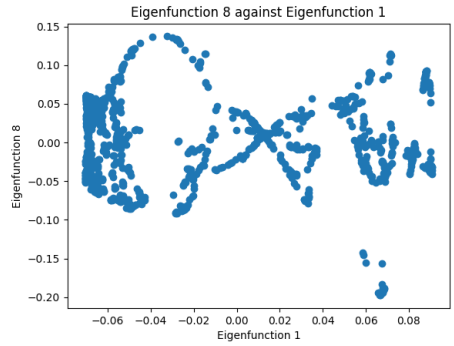
(e) Eigenfunction 5 vs. Eigenfunction 1



(f) Eigenfunction 6 vs. Eigenfunction 1

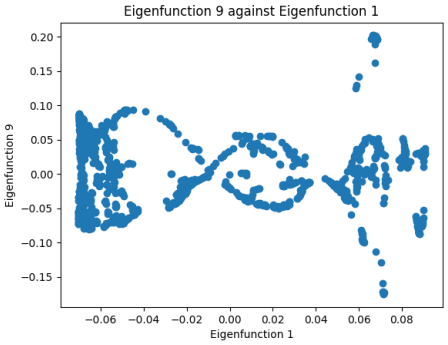


(g) Eigenfunction 7 vs. Eigenfunction 1

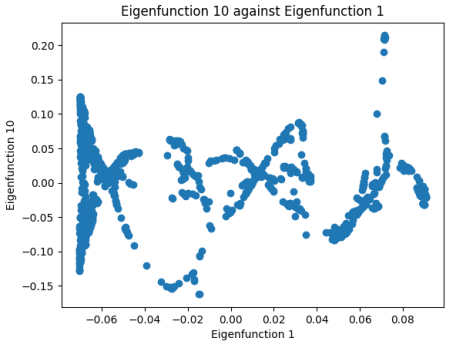


(h) Eigenfunction 8 vs. Eigenfunction 1

Figure 11: Continuation of Fig. 10



(a) Eigenfunction 9 vs. Eigenfunction 1



(b) Eigenfunction 10 vs. Eigenfunction 1

2.3.2 1000 data points

When running the same tests as in the previous chapter with 1000 samples from the Swiss Roll-dataset a few differences occur. The visualisation of the new dataset with 1000 samples looks very similar to the earlier version with 5000 samples seen in Fig. 7 (which is why we did not include it again). In accordance to our expectation in the previous chapter, the computation time for 1000 samples (resulting in a 3000 x 3000 eigenwert-problem) is way faster and does not impose an issue anymore.
The corresponding plots of the first 10 eigenfunctions against the first eigenfunction ϕ_1 can be seen in Fig. 10. Especially the plots of the first 4 eigenfunctions look very similar compared to the plots with 5000 samples although for example the third eigenfunction ϕ_3 seems to be the negative of before. Another observation among all plots is the low density of samples which becomes more visible for the higher eigenfunctions. While some of these still look similar to the corresponding plots with 5000 samples, e.g., ϕ_5 vs. ϕ_1 in Fig. 10e, others are significantly different like ϕ_7 vs. ϕ_1, cp. Fig. 8g (5000 samples) & Fig. 10g (1000 samples). In this case, it even seems that in the run with 1000 samples eigenfunction 7 and eigenfunction 8 were switched compared to previously with 5000 samples. Overall, we observed that when running the analysis multiple times and therefore sampling different datasets of 1000 points the eigenfunctions varied quite strongly. The reason for this is most probably that 1000 data points are not sufficient to securely contain the structure of the Swiss Roll-manifold. This results in stronger differences between the eigenfunctions of multiple datasets containing 1000 samples each. Conceptually, this could be related to the problem of overfitting, where a Machine Learning-model with high capacity starts to memorise the training samples and overfit its parameters. While the first 3-4 eigenfunctions were the same for all different 1000-sample-datasets, the higher eigenfunctions were much more specific for the current sample set. Yet, to prove this proposition a much more intensive testing and mathematical reasoning would have to be provided.
Regarding the number l from which on ϕ_l is no longer a function of ϕ_1, we would propose that it is again $l = 4$ since widespread assignment of multiple values in ϕ_l to one value in ϕ_1 occurs here. However, at some points this is also visible in ϕ_3 & ϕ_2, cp. Fig. 10c around -0.02 & 0.06 as well as Fig. 10b around 0. This is most probably due to the low number of sample sizes which is the reasoning for $l = 4$.

After again running our code from Task 1 on the Swiss Roll-dataset, this time with 1000 samples, we obtain the three principal components along their singular values or energies. The singular values are now as follows:

$$\begin{pmatrix} 395.62555873 & 224.37944569 & 200.87593921 \end{pmatrix}$$

The corresponding singular vectors or principal components are the columns in the following matrix:

$$\begin{pmatrix} -0.24887447 & -0.35673058 & -0.90044699 \\ -0.96581157 & 0.16108806 & 0.20312223 \\ -0.07259135 & -0.92021407 & 0.38462523 \end{pmatrix}$$

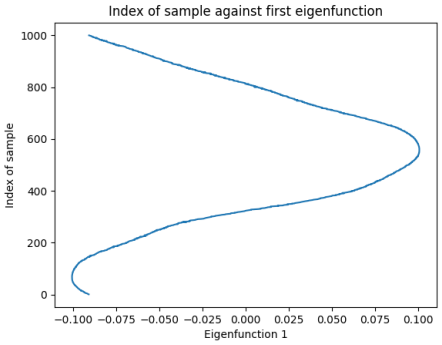
And finally the accumulated energy of the first two principal components amounts to **83.678**% of the total energy in the dataset with 1000 samples. Compared to the dataset with 5000 samples the results are very

similar. The singular values are homogeneously scaled by a factor around 2.2-2.25 for all three and the singular vectors/principal components also show only minor absolute differences of up to +/- 0.1 in each component. The only major difference is that the first singular vector has inverted signs in all components which is due to the singular value decomposition, which does not induce uniqueness of singular vectors but rather has multiple (mirrored) solutions. Additionally, the accumulated energy for two components is almost exactly the same with 83.678% for 1000 samples & 83.073% for 5000 samples.
The most probable reason for this extremely similar result for PCA in both cases is that the overall distribution of the samples is the same. Since PCA mainly relies on variances in the dataset, which are the same if the distribution sampled from is the same, it does not matter how many points are responsible for the variance in the dataset. This only becomes visible when looking at the absolute value of the singular values which scale with the number of samples.

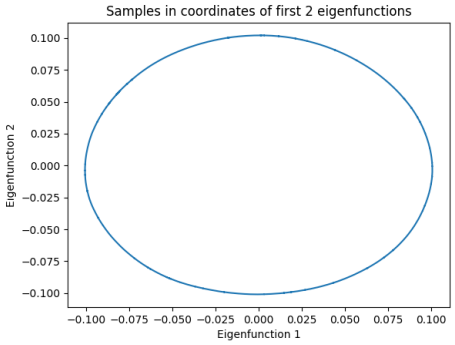
2.4 Part 3 – Diffusion Maps on trajectory data

Our solution to this task can be found under "*src/task2_diffusion_maps/3_trajectory_data.py*". To visualise the trajectories of the first two pedestrians we again used the utility function from Task 1 and obtained the same plot as before, shown in Fig. 3.
A projection into two principal components in PCA corresponds to finding the first two eigenfunctions of the dataset with the Diffusion Maps algorithm. To analyse how many eigenfunctions are necessary for representing the dataset accurately we compute the first three (ϕ_1 - ϕ_3) using the Diffusion Maps implementation and disregard the 0-th eigenfunction since it is constant over all samples. The criterion for accurate representation as given in the exercise sheet is that the samples do not overlap when being represented as coordinates in N eigenfunctions.
Based on this we can check the sufficiency of representation by plotting the dataset samples in coordinates of the first N eigenfunctions and look for intersections in the plot. The visualisation of our results can be seen in Fig. 12, where the first plot is not just 1-dimensional but includes the sample index as well to facilitate understandability. From Fig. 12a we can conclude that 1 dimension is not enough to represent the dataset sufficiently since almost all values of ϕ_1 relate to more than one sample which corresponds to a "1D-intersection". If the plot featured just the sample's coordinate in one dimension (1D-line plot) this would be visible but the plot would be very crowded. In contrast, the plot for 2 dimensions, cp. Fig. 12b, shows no overlaps and therefore the first two eigenfunctions (or components of the Diffusion Maps result) can successfully represent the dataset. In contrast, the first two principal components from PCA were not able to capture most of the energy contained in the data. Additionally, we also visualised the embedding of the dataset in the first three eigenfunctions, visualised in Fig. 12c. Here, again no intersections occur and therefore the dataset is accurately represented, which is an obvious result since looking at the plot from upside-down reconstructs the view with 2 dimensions from Fig. 12b.
In addition, we want to give a possible explanation to why Diffusion Maps can achieve accurate representation with 2 dimensions while PCA cannot. This is most probably due to the non-linear character of Diffusion Maps. While PCA is constrained to finding the optimal N-dimensional hyperplane approximating the dataset, which is problematic when the data is non-linear, Diffusion Maps can introduce a variety of non-linear components/coordinates to find the best representation. In this case, where the trajectory data does not seem to be a hyperplane-like manifold, Diffusion maps can therefore reduce dimensionality of the dataset better (less dimensions) than PCA.

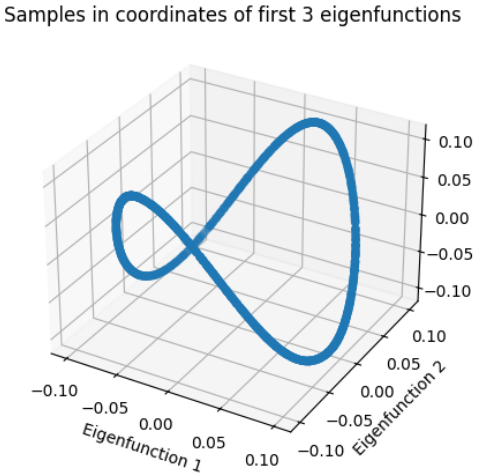
Figure 12: Trajectory data in new coordinates of the eigenfunctions from Diffusion Maps



(a) Index of samples vs. eigenfunction ϕ_1



(b) Samples in coordinates of eigenfunction ϕ_1 & eigenfunction ϕ_2



(c) Samples in coordinates of eigenfunctions ϕ_1, ϕ_2 & ϕ_3

2.5 The Swiss Roll-dataset in the *Datafold*-Software

Running the following code line downloads and installs the `datafold` software.

```
python -m pip install datafold
```

This software already includes methods to create a Diffusion Map. Therefore the Swiss Roll-data set is taken as a parameter to the built-in function to create a Diffusion Map for further calculations.

The `plot_pairwise_eigenvector()` method from `datafold.utils.plot` is used to compute the eigenvectors via the `datafold` software.

Figure 14 shows the result of the first ('first' in a sense as described in sub chapters before) eigenvector compared to different eigenvectors with 5000 samples. Here, the implementation run time was fast enough to not notice a significant delay (in contrast to our own implementation of Diffusion Maps, see sub chapter *2.3 Part 2 – The Swiss Roll dataset*). We compute 11 eigenpairs to get 10 plots of comparing eigenfunction 1 against ten different eigenfunctions.

To achieve this we use the built-in function `dmap = datafold.dynfold.DiffusionMaps.laplace_beltrami()` with a parameter `n_eigenpairs=11` to create a diffusion map with 11 eigenpairs and compute the plots via the `plot_pairwise_eigenvector()` method with the parameter `n=1` which indicates we are comparing all with the first eigenfunction.

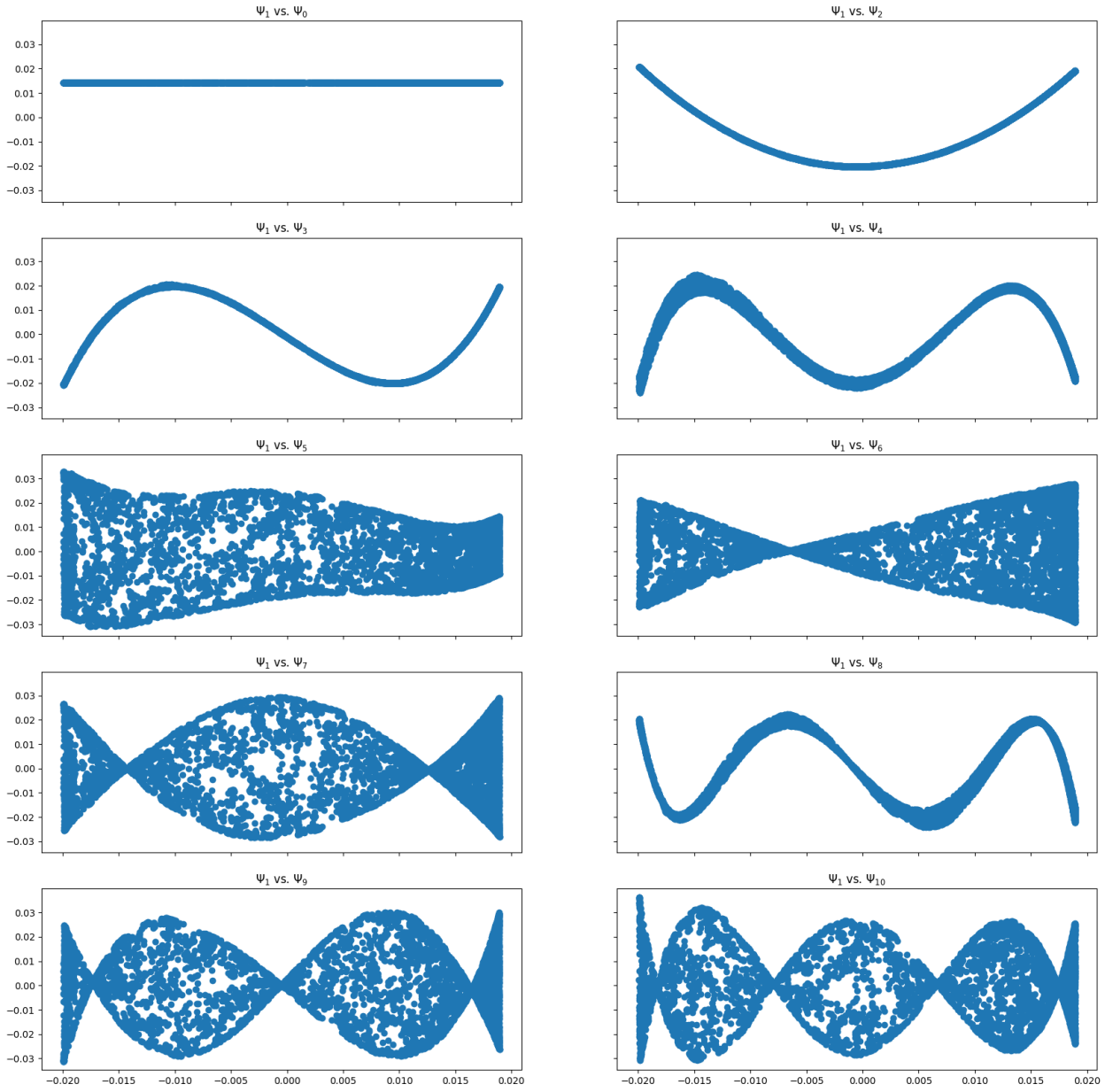


Figure 13: Plots of the first 10 eigenfunctions of the Swiss Roll-data set with 5000 samples

While the plots with 5000 samples show a very similar result compared to the plotting from the own implementation of Diffusion Map. The outcome of from 1000 samples differs quite a lot. Comparing the first two or three eigenvectors seem to still hold the general structure (i.e. quadratic or cubic function) but anything after that resembles less and less from the plots of our own computation. The reason behind this is contemplated to be the same as described in sub chapter *2.3.2 1000 data points*. Multiple runs of this code also lead to different plotted graphs when using 1000 samples. This phenomenon is not observed when using more samples such as 5000. This again validates the suspicion that 1000 samples are not enough data to accurately represent and hold the shape of the original data.

By comparing the time and effort put into our own implementation with the reconstruction of the sub task with the provided downloadable software it is clear that by installing the datafold software reduces the time spent on plotting and computing eigenfunctions by quite a lot.

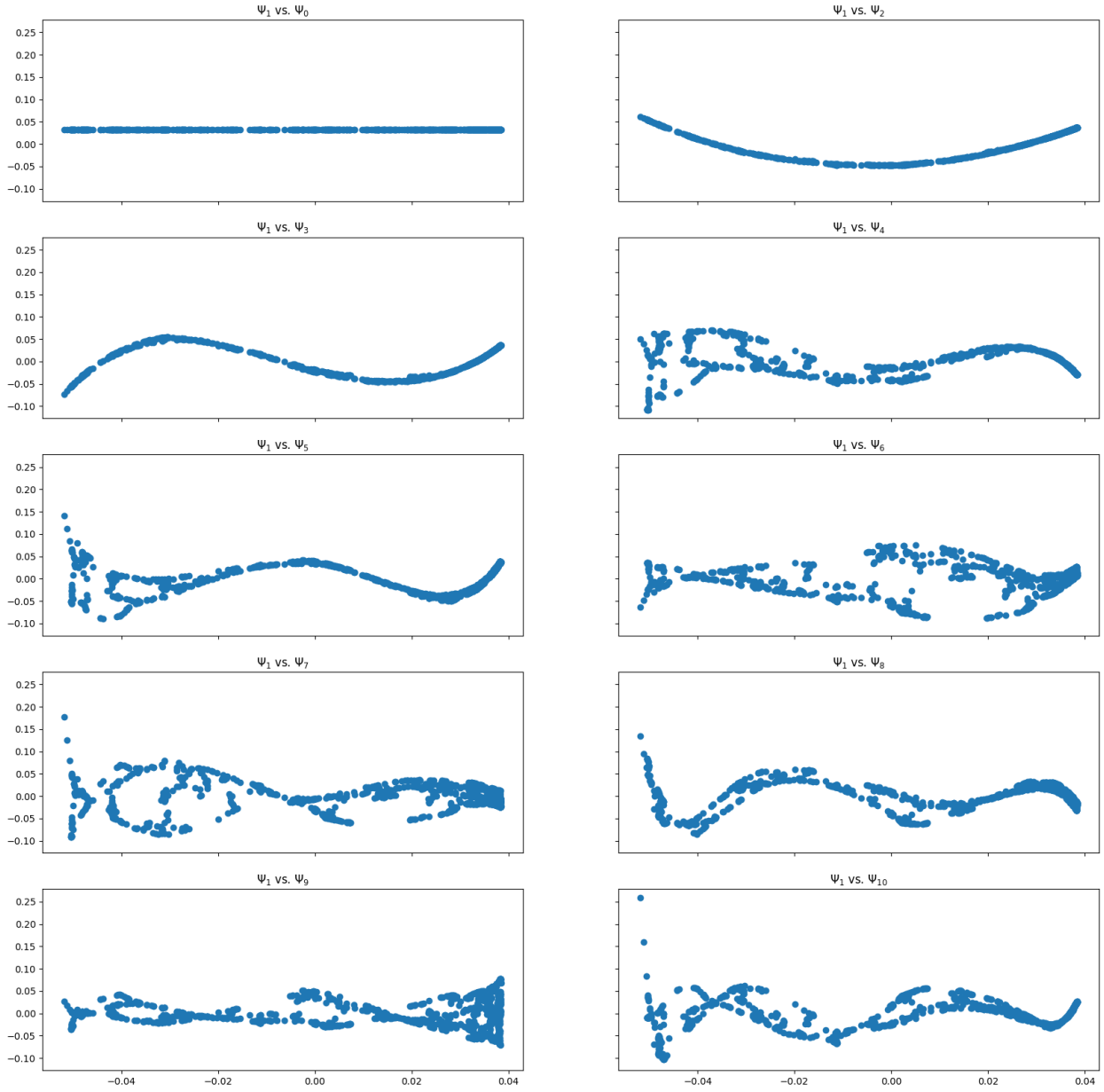


Figure 14: Plots of the first 10 eigenfunctions of the Swiss Roll-data set with 1000 samples

2.6 Additional questions

2.6.1 Time to implement & test method

The implementation of the **Diffusion Maps-algorithm** took approximately **5 hours** where most of the time went into reading and understanding the basics of the algorithm. The actual implementation with NumPy was rather straightforward since clear instructions were given. Implementation of the test in **Task 2.1** was also rather straightforward and required another **2-3 hours** to fully modularise and visualise the results. The most time-intensive analysis was **Task 2.2** which took approximately **6-8 hours** to fully implement, test and visualise the results. This is mostly due to minor changes which were then tested and optimised in multiple iterations. **Task 2.3** was again rather straightforward and its implementation needed **2-3 hours**.

2.6.2 Accuracy of representation & measure of accuracy

In contrast to Task 1 & 3 where representation and corresponding accuracy where used rather straightforward (visual feedback, loss functions), we found it quite difficult to grasp these concepts for Diffusion Maps. One notion on accurately representing the dataset was mentioned in Task 2.3 where the embedding in the new coordinates of the first eigenfunctions should not intersect. However, in Task 2.2 we looked at eigenfunction ϕ_l being a function of ϕ_1 which also corresponds to the number of eigenfunctions or components that are needed to accurately represent the data. Overall, we would state that we were able to represent the data quite accurately since most given datasets where 2- or 3-dimensional embeddings on which we used multiple eigenfunctions (5-10). For the 30-dimensional trajectory dataset from Task 2.3 we know that our representation with 2 & 3 eigenfunctions is accurate since the embeddings do not intersect.

2.6.3 Learnings from dataset & method

Regarding the method we learned that Diffusion Maps are able to obtain a non-linear embedding of the dataset and in consequence also manifold. Since most datasets are not linear this helps a lot in finding lower-dimensional representations and in most cases they perform better at finding those than linear methods, e.g., Principal Component Analysis. Furthermore, reading the paper on Diffusion Maps revealed that in principal other kernels than the exponential kernel, like in this implementation, could be used which would give rise to different eigenfunctions and therefore embeddings. Additionally, the comparison to Fourier analysis of the data in Task 2.1 was an interesting result since it revealed the similarity when looking at continuous signals.
Another big take-away is the understanding of manifolds and the difference in the dimensionality of a manifold and the dimensionality of the respective embedding. The Swiss Roll-dataset as an example is a 2D-manifold since it can be approximated by a 2D-hyperplane locally but in contrast the given embedding was in a 3-dimensional Euclidean space. Looking at the Swiss Roll-dataset with 1000 samples it was also evident that the performance and accuracy of dimensionality reduction is depending on the number of samples given for that manifold. While this was less of an issue for PCA, the results of Diffusion Maps among different datasets with 1000 samples varied quite strongly. Regarding the trajectory dataset it was interesting to see that a high-dimensional embedding (30 dimensions) could be rather well described by 2-3 dimensions which is a big dimensionality reduction.

Report on task 3, Training a Variational Autoencoder on MNIST

3 Variational Autoencoders

This task's goal was to implement a Variational Autoencoder and train it using the MNIST dataset. This task is divided into 3 parts, where the first one discusses the implementation of the VAE, the second about the training the model with 2-dimensional latent space, and the third part compares the results of a two dimensional latent space against the same model trained with 32-dimensional latent space.

3.1 Implementing the VAE

The Variational Autoencoder was implemented using the Python framework PyTorch. It is a fully featured framework for building deep learning models, fitting our use-case perfectly. The framework provides all the needed building blocks to implement a VAE, thus the remaining task was to put the model together.

The VAE is divided into an encoder and a decoder. The encoder takes the input data and represents it in the specified number of latent dimensions. The encoder consists of two hidden layers, both with ReLU activation functions, and outputs the mean and standard deviation of the approximate posterior. The mean layer is an unbounded linear layer with no activation function. Since the mean represents the location of the latent variable distribution in the latent space, we keep it unbounded and linear, so the model can freely learn the underlying mean values.

The standard deviation layer is also a linear layer. We tried softplus, Relu, and no activation function for the standard deviation layer, and decided on not using an activation function on the layer at all, since it produced the best results while training the model.

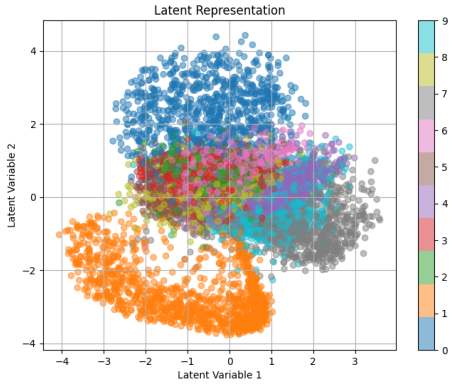
The decoder takes the latent representation of the data and tries to represent it in the input dimensions as accurately as possible. It also consists of two hidden layers, both with ReLU activations, and outputs the mean of the distribution, with a sigmoid layer to ensure that the values are in the range $[0, 1]$.

The loss function used in the training is the ELBO-loss, which is the sum of the reconstruction loss and KL-divergence loss terms. In our case, we use binary cross-entropy as the reconstruction loss.

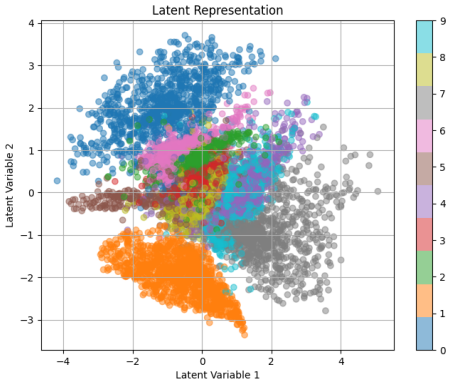
3.2 Training the VAE

The implemented VAE was trained with all the parameters given in the exercise sheet. The full training script can be found in `'./src/task3_vae/1_mnist.py'`. The following plots are obtained by running the script. We assumed that the model would have converged after 100 epochs.

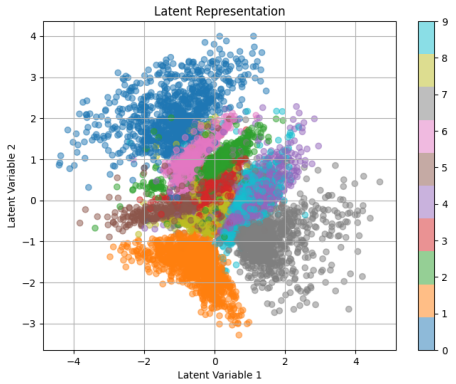
Figure 15: Plots of latent representations



(a) After 1 epoch

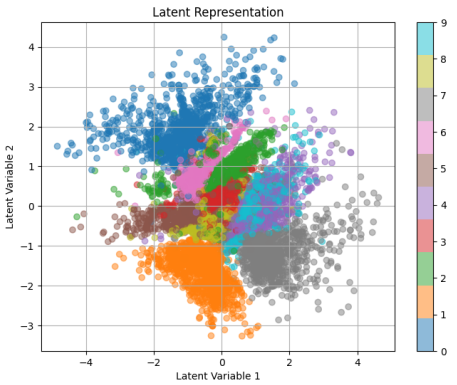


(b) After 5 epochs



(c) After 25 epochs

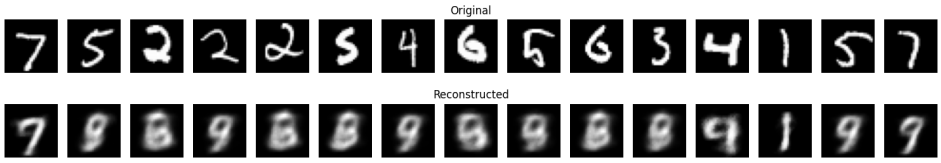
(d) After 50 epochs



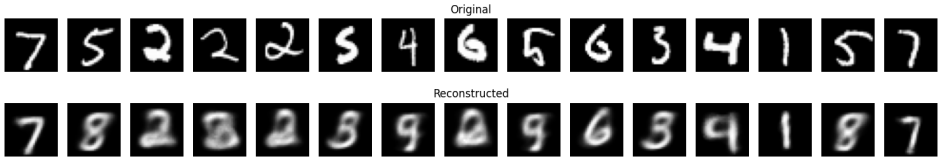
(e) After 100 epochs

The latent representation shows us how well the model is able to recognize the different classes. A well trained model should cluster the different classes. We can see that the classes start to cluster better with each plot, but even after 100 epochs, it can be seen that for example, the model is not able to make a clear distinction between the classes 4 and 9, and 5 and 8, since they overlap heavily in the latent representation scatter plot.

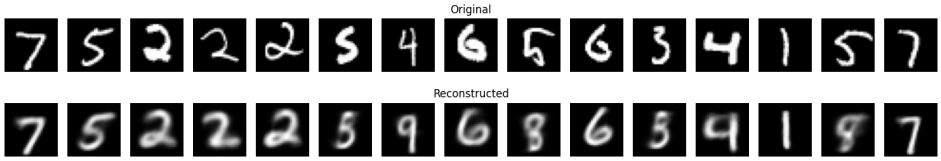
Figure 16: Plots of reconstructed digits



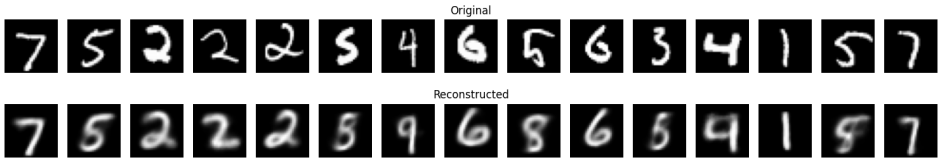
(a) After 1 epoch



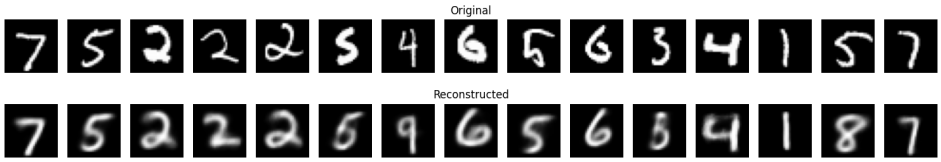
(b) After 5 epochs



(c) After 25 epochs



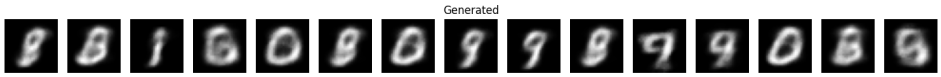
(d) After 50 epochs



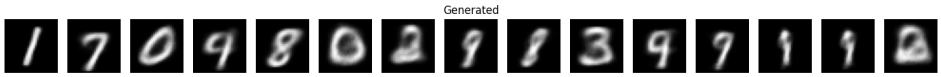
(e) After 100 epochs

The reconstruction of the latent data further confirms that the model is not able to correctly represent the latent data in every case, since in plot 16e there are misclassifications in the 7th and 14th image.

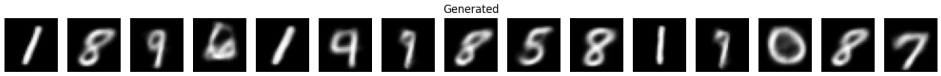
Figure 17: Plots of generated digits



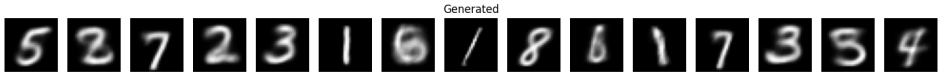
(a) After 1 epoch



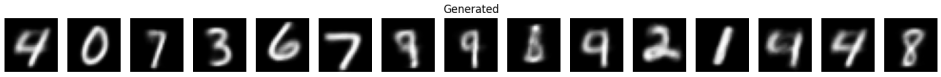
(b) After 5 epochs



(c) After 25 epochs



(d) After 50 epochs



(e) After 100 epochs

In figure 17, the subplots should not be compared column-wise, since in each plot, the generated latent data is obtained from a normal distribution, and the data plotted is thus not the same in the subplots. Rather, one should analyse the blurriness of the images, which reflects the model's uncertainty of the class of the underlying latent data. Once again, it can be noticed that the images depicting a 4, 5, 8 or 9 are much blurrier than the other images, since the model can not make a clear distinction of the classes.

It can be noticed that the generated images are of much poorer quality than the reconstructed digits. We thought that the reason for this phenomenon is that the model architecture might not have sufficient capacity to capture the complexity of the data distribution. If the model is too simplistic or shallow, it may struggle to learn the underlying distribution in the data, leading to poor-quality generated samples.

Figure 18: Plots of generated digits

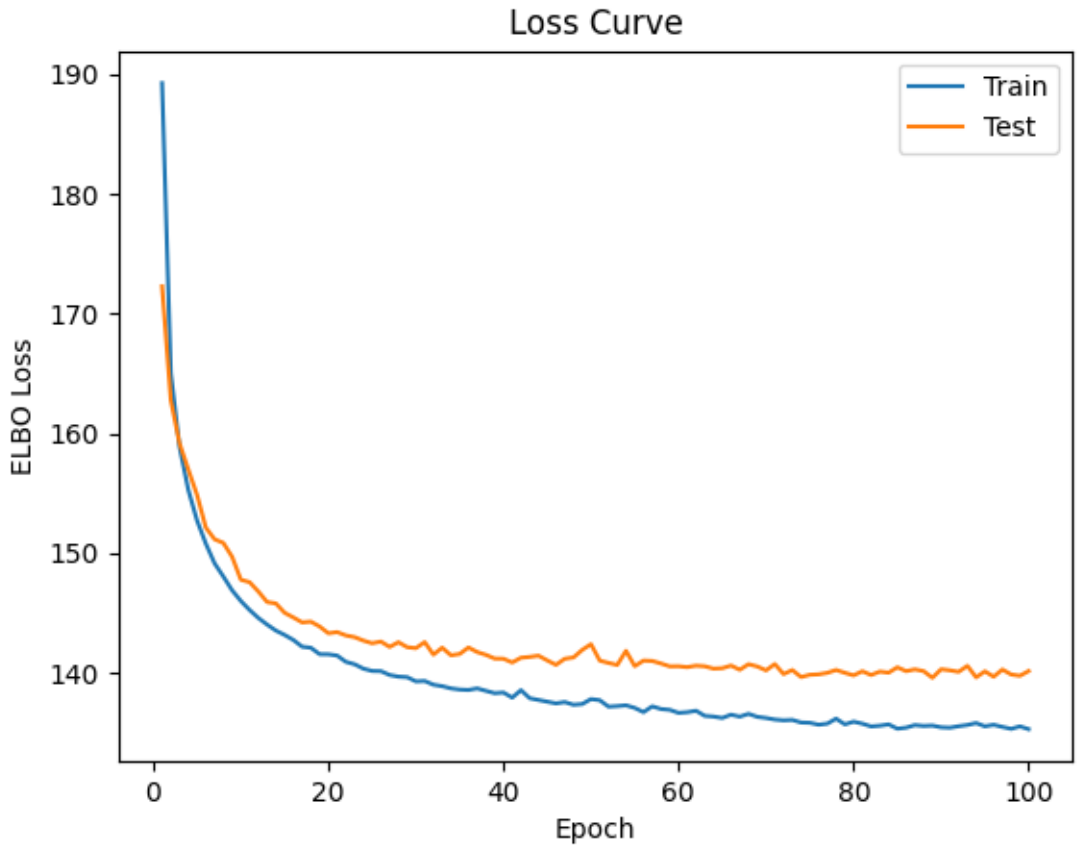


Figure 19: Plot of loss values

Figure 18 shows the values of the loss function as a function of the epochs trained. The model has not necessarily converged at epoch 100, since the training loss curve does not show clear signs of plateauing, but due to hardware limitations causing time constraints, the training was stopped at epoch 100.

Implementing the model and testing the methods took us about 3 days in total. The latent representation, image reconstruction, image generation, and the ELBO-loss curve all give insight to how well the model is performing. Interpreting the plots, we think that we were able to represent the data fairly accurately, since the model is able to reconstruct the images reasonably well, and the image generation is able to produce images with recognizable digits. The ELBO-loss curve also shows that the model is steadily learning the distribution in the data, without overfitting or underfitting.

3.3 Train the VAE using a 32-dimensional latent space and do the following experiments after the optimisation converged

Note: this section is completely done in Google Colab since we don't have enough GPU to do the training locally.

3.3.1 Compare 15 generated digits with the results in 3.2

Motivation The motivation behind this task is to gain a deeper understanding of how the dimensionality of the latent space in a VAE affects its ability to learn and generate data. By comparing models with different latent space sizes, we aim to observe the trade-offs between the complexity of the latent space and the quality of the generated outputs. This exploration is crucial for applications where the balance between model complexity

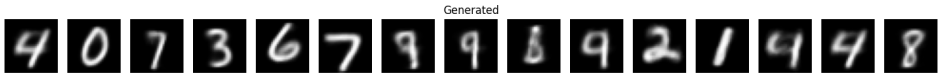


Figure 21: The generated number after 100 epochs from section 3.2

and performance is key, such as in image generation, anomaly detection, and feature extraction. Additionally, understanding the impact of latent space dimensionality can provide insights into how to optimize VAEs for various real-world applications.

Figure 20: Plots of generated digits with 32-dimensional latent space

Observation The comparison of the generated numbers from the Variational Autoencoder (VAE) using latent dimensions of 2 21 and 32 **??** revealed distinct differences in the quality and clarity of the images produced. Specifically, the images generated with a latent dimension of 32 appeared blurrier and more compromised compared to those generated with a latent dimension of 2 (eg: the number 6, 2, 8, especially).

Analysis

1. Latent Dimension Size and Image Clarity

- **Latent Dimension 2:**
  - The VAE with a latent dimension of 2 generated images that were relatively clearer and more defined. This suggests that with a smaller latent space, the model was able to focus on capturing the most significant features of the data, resulting in sharper images.
- **Latent Dimension 32:**
  - The images produced with a latent dimension of 32 were blurrier. This could be due to the larger latent space allowing the model to capture more noise and less relevant features, leading to a decrease in image quality.

2. Model Complexity and Overfitting

A larger latent dimension increases the complexity of the model. While this can theoretically allow the model to capture more intricate details, it also raises the risk of overfitting. In this case, the blurrier images suggest that the VAE might have overfit to noise in the training data, leading to poorer generalization in the generated images.

3. Information Bottleneck

The information bottleneck principle in VAEs posits that the latent space should compress the input data to its most essential features. With a smaller latent dimension (2), the model is forced to distill the data into its core components, which can lead to clearer outputs. In contrast, a larger latent dimension (32) might dilute this effect by allowing too many less essential features to be represented.

4. Trade-off Between Latent Dimension Size and Reconstruction Quality

There is a trade-off between the size of the latent dimension and the quality of the image reconstruction. A very small latent dimension can lead to loss of important details, while a very large latent dimension can lead to capturing too much noise. The blurrier images with a latent dimension of 32 indicate that the model has not found the optimal balance between these factors.

Possible strategy for improvement

1. **Regularization Techniques:**
   - To improve the quality of images generated with a larger latent dimension, consider implementing regularization techniques such as dropout or batch normalization. This can help mitigate overfitting and reduce noise in the generated images.
2. **Tuning Latent Dimension:**
   - Experiment with intermediate latent dimensions (e.g., 10, 16) to find a more optimal balance between capturing essential features and avoiding overfitting. This can help achieve clearer image generation.
3. **Data Augmentation:**
   - Enhancing the training dataset through data augmentation can help the VAE learn more robust features, potentially improving the quality of generated images even with larger latent dimensions.
4. **Improved Loss Function:**
   - Investigate and potentially modify the loss function used during training. Adding perceptual loss or using techniques like GANs (Generative Adversarial Networks) in conjunction with VAEs might improve the perceptual quality of the generated images.

By addressing these factors, it is possible to improve the performance of the VAE and generate clearer images even with higher-dimensional latent spaces. e performance of the VAE and generate clearer images even with higher-dimensional latent spaces.

3.3.2 Compare the loss curve with the one in 4

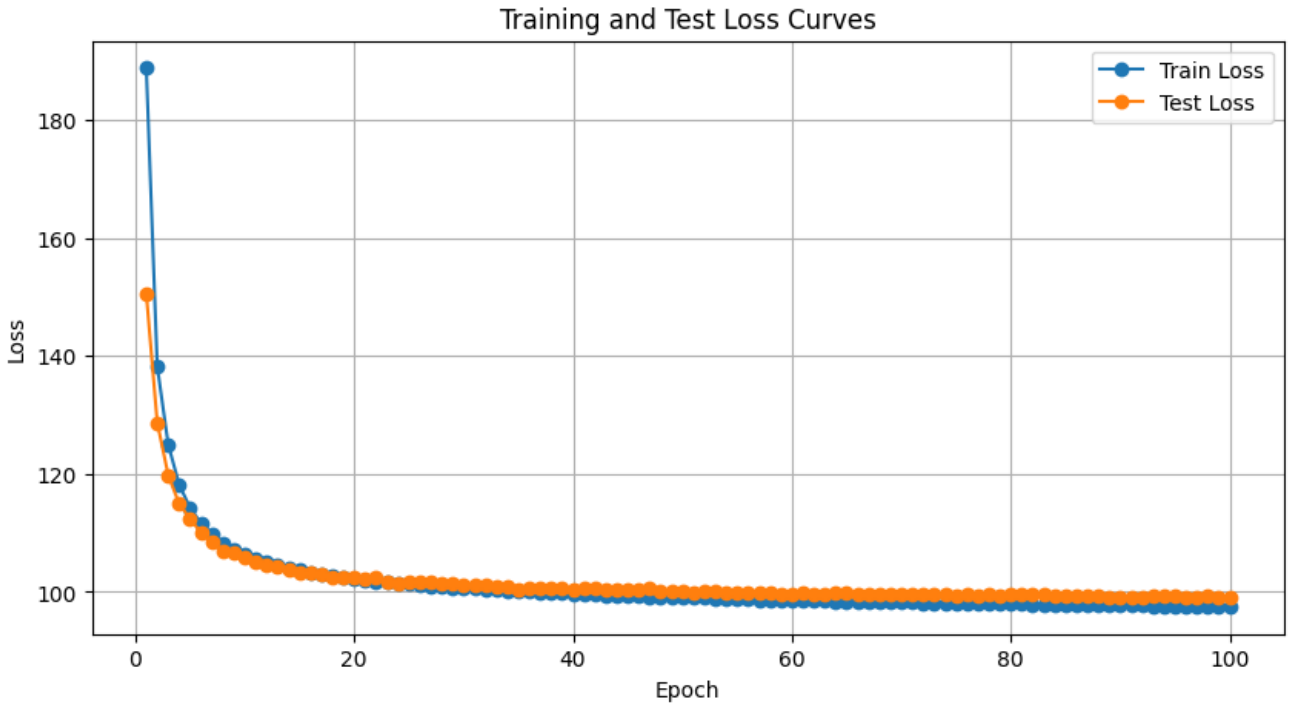


Figure 22: Loss curve of VAE with 32-dimensional latent space

Observations

- **Loss Curve for Latent Dimension 2 (22):**

  - The training loss decreases steadily and converges to a lower value compared to the test loss.
  - The test loss follows a similar trend to the training loss but remains consistently higher, indicating a slight generalization gap.

- **Loss Curve for Latent Dimension 32 (Figure 22):**
  - Both the training and test losses decrease rapidly initially and then plateau at around 100 epochs.
  - The training and test losses converge to similar values, suggesting that the model has managed to generalize well to the test data.

Conclusion

- **Latent Dimension 2:**
  - The VAE with a latent dimension of 2 shows a slight generalization gap, indicating that the model is slightly overfitting to the training data.
  - The lower convergence of the training loss compared to the test loss suggests that the model is more focused on capturing the most significant features of the data, resulting in clearer images but with a risk of underfitting the test data.
- **Latent Dimension 32:**
  - The VAE with a latent dimension of 32 demonstrates good generalization, as indicated by the convergence of training and test losses to similar values.
  - The rapid initial decrease and plateauing of the loss suggest that the model captures a more comprehensive range of features, but this comes at the cost of increased noise, leading to blurrier images.

This task taught us that it is possible to extract meaningful information from a dataset while reducing the dimensionality of the dataset. This was achieved by connecting two neural networks, the encoder and the decoder, that tried to learn the non-linear functions $f(x) = z$, where $\dim z \ll \dim x$, and $g(z) = x$ respectively. Learning the underlying distributions from the original data is not straight-forward at all, and a lot of work has to be put into the hyperparameter tuning to be able to represent the data in the latent dimension accurately.

Report on task 4, Fire Evacuation Planning for the MI Building

4 Task 4, Fire Evacuation Planning for the MI building

Task Description Due to the increasing number of students at the Technical University of Munich, the re-evaluation plan for the MI building needs to be reconsidered. An important piece of information is the distribution of people within the MI building, denoted as $p(x)$.

In a hypothetical scenario, the fire department decided to track 100 random students and employees during the busiest hour on different days. The idea is to use this data for learning $p(x)$. As a first experiment, the fire department wants to estimate the number of people that is critical for the building. To simplify the task, it defined a sensitive area in front of the main entrance marked by the orange rectangle (130/70 & 150/50) where the number of people should not exceed 100.

Note that task 4 is completely done on google colab since we don't have enough GPU locally.

4.1 Visualize the FireEvac Dataset

- Download the FireEvac dataset (training: 3000 x-y-positions; testing: 600 x-y-positions).
- Create a scatter plot to visualize the dataset (see 23).

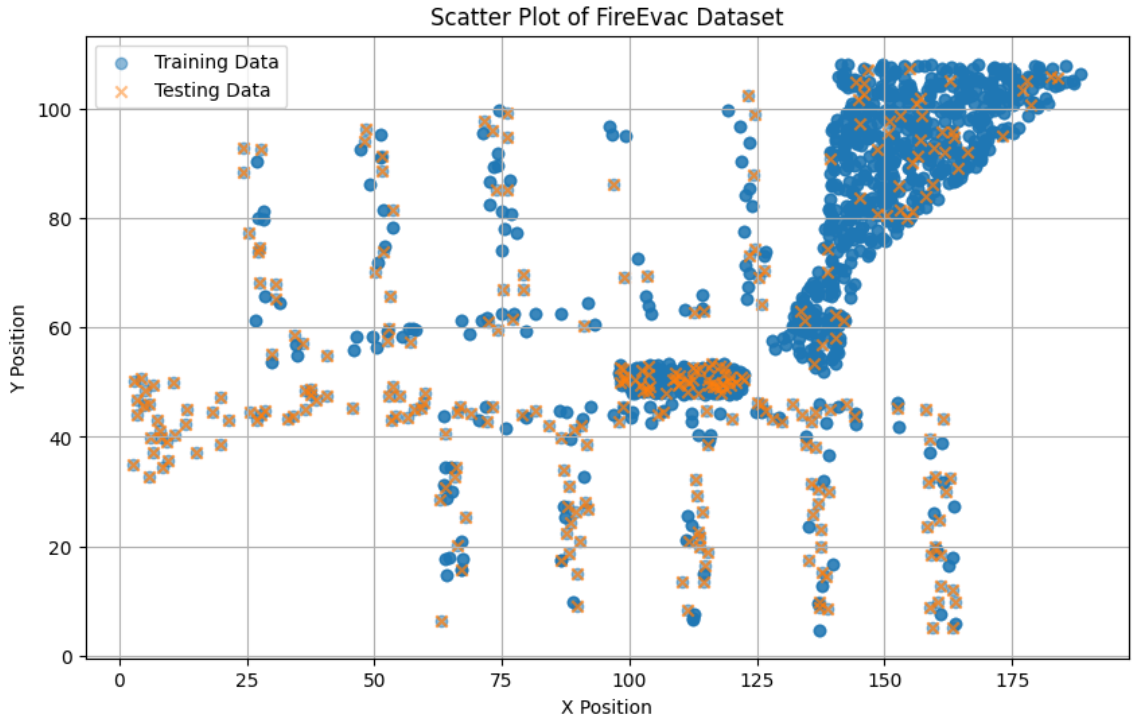


Figure 23: Scatterplot of the FireEvac Dataset

4.2 Train a VAE on the FireEvac Data

- Train a Variational Autoencoder (VAE) on the FireEvac data to learn $p(x)$.
- Reuse your VAE implementation from the previous task, adjusting the number of input and output neurons, and possibly the number of layers.

Note 1: The observation space for this dataset is two-dimensional (instead of 784 in MNIST).

- Although it is a low-dimensional dataset, it is not a simple one. You may need to train the VAE for quite a few epochs, or tune hyperparameters.
- Another good idea is to rescale the input data x to a range of $[-1, 1]$ before training the model.

Note 2: In former semesters, an architecture of 2-64-64-2 for the encoder and decoder worked well, i.e., a two-dimensional latent space and 64 neurons in each of two hidden layers, with a learning rate of 0.0005 (Adam optimizer) and a batch size of 64, training for 200 epochs. The generated data may still not look perfect, but reconstruction works quite well on the test set.

We did as is suggested and scaled the input data accordingly (see 24, 25)

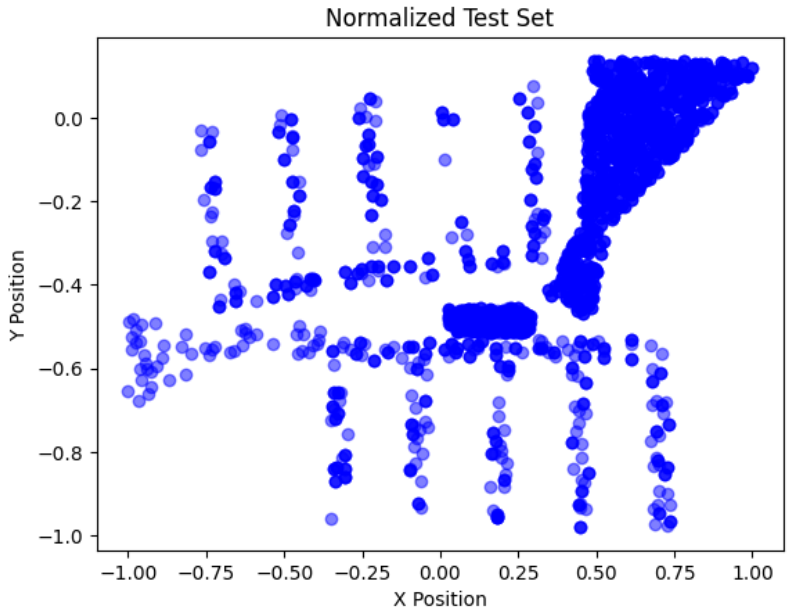


Figure 24: Scatterplot of the normalized train set

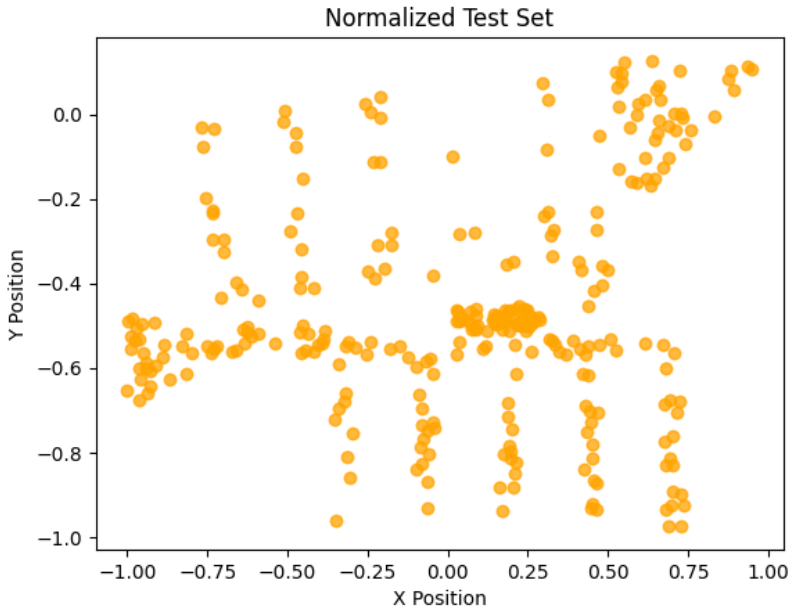


Figure 25: Scatterplot of the normalized test set

We also followed the instruction and adjusted the architecture of the encoder and decoder accordingly, used the learning rate of 0.005, Adam optimizer and the batch size of 64, training epoch of 200. We also provided the code here.

VAE Training Parameters

```
import torch
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset

# Training parameters
input_dim = 2  # (x, y) positions
latent_dim = 2
hidden_dim = 64
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
learning_rate = 0.0005
num_epochs = 200
batch_size = 64

# Initialize model, optimizer, and dataloader
model = VAE(d_in=input_dim, d_latent=latent_dim, d_hidden_layer=hidden_dim, device=
    device).to(device)
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
train_loader = DataLoader(TensorDataset(train_data), batch_size=batch_size, shuffle=True
    )
```

4.3 Make a scatter plot of the reconstructed test set

The result of the reconstruction using VAE can be found in 26.

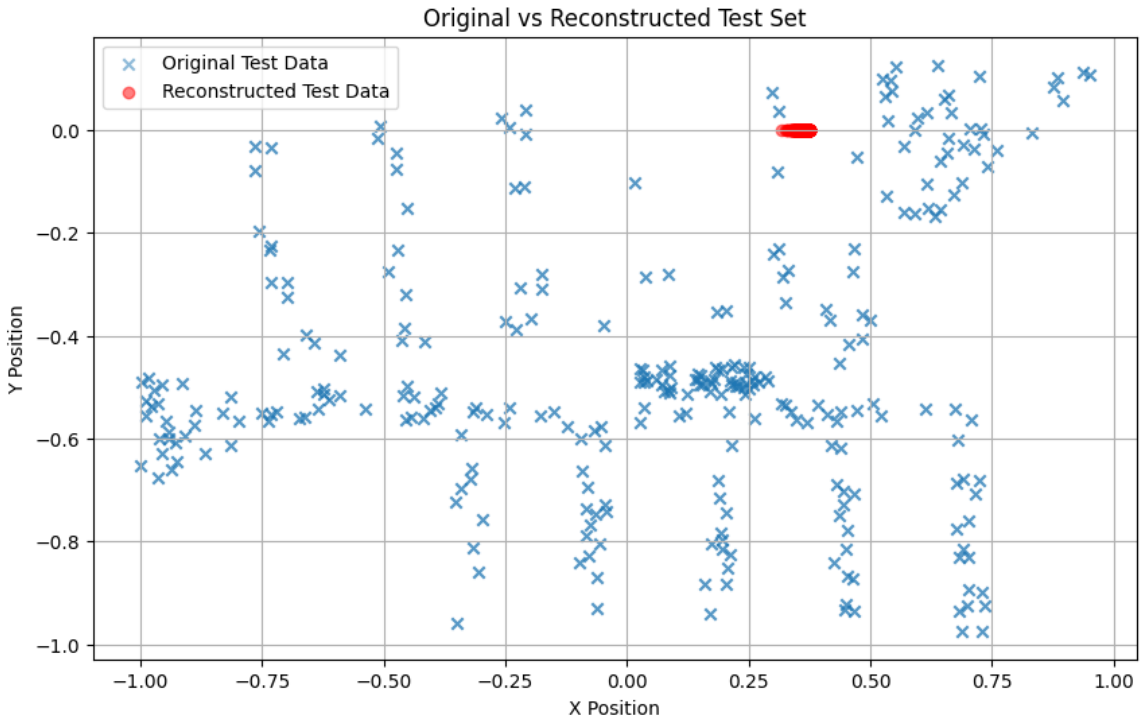


Figure 26: Reconstructed test set

We also zoomed in on the reconstructed test set scatter plot, to have a closer look for better analysis (see 27)

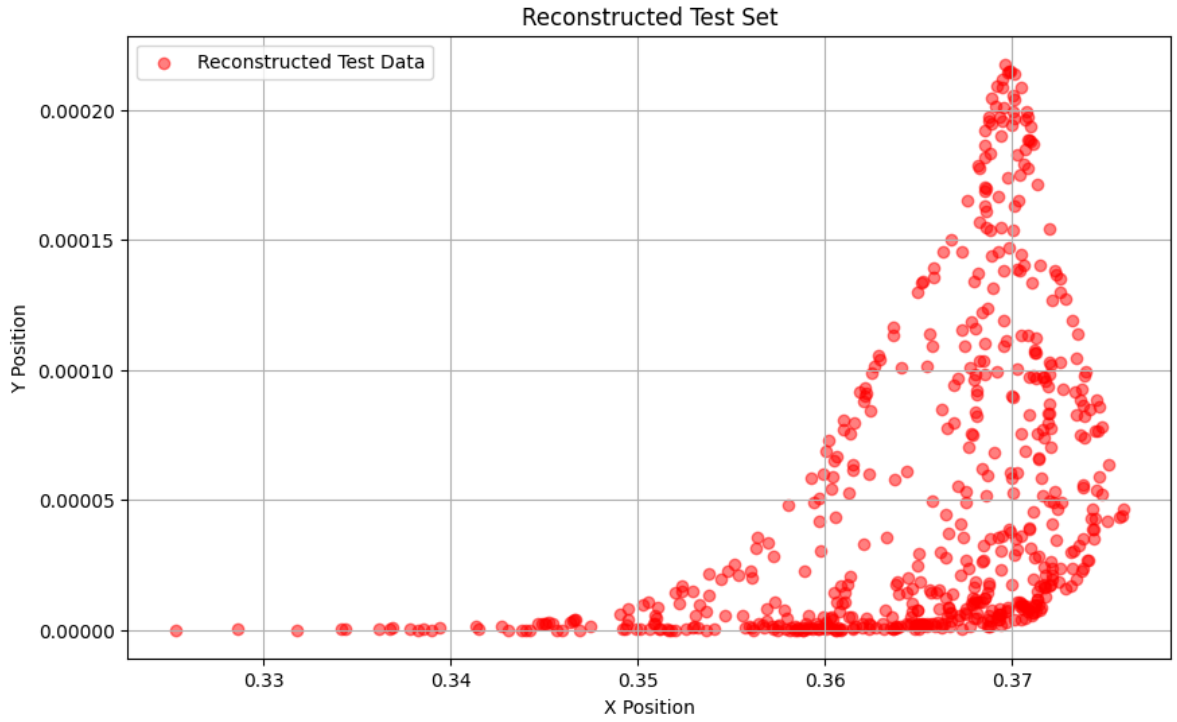


Figure 27: Reconstructed test set when zoomed in

4.3.1 Observations

Projection to a Single Point

The reconstructed values of the test set converged to a single red dot on the plot. This behavior indicates that the VAE mapped every input to the same location, which was not anticipated.

The red dot was positioned outside the predefined sensitive area. This was unexpected, as we hypothesized that the VAE would produce slightly perturbed versions of the original positions.

4.4 Make a scatter plot of 1000 generated samples.

As before we made the scatter plot for both the generated sample compared to the FireEvac dataset, and the zoomed in scatter plot (28, 30).

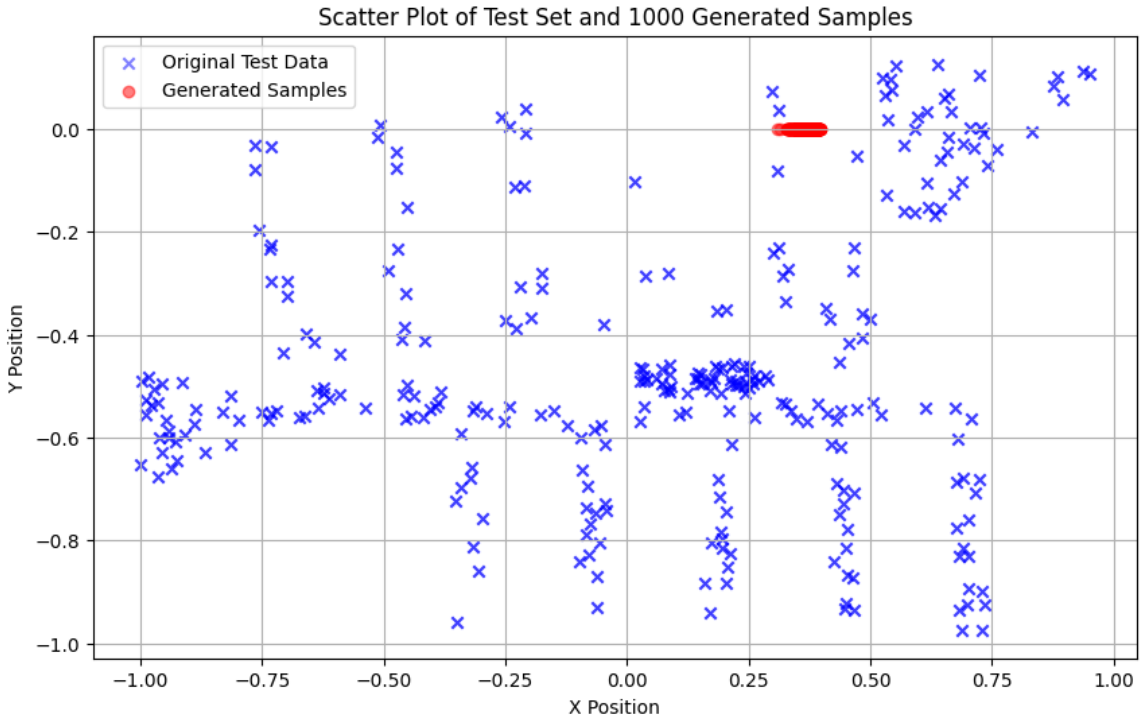


Figure 28: Scatter plot of 1000 generated sample

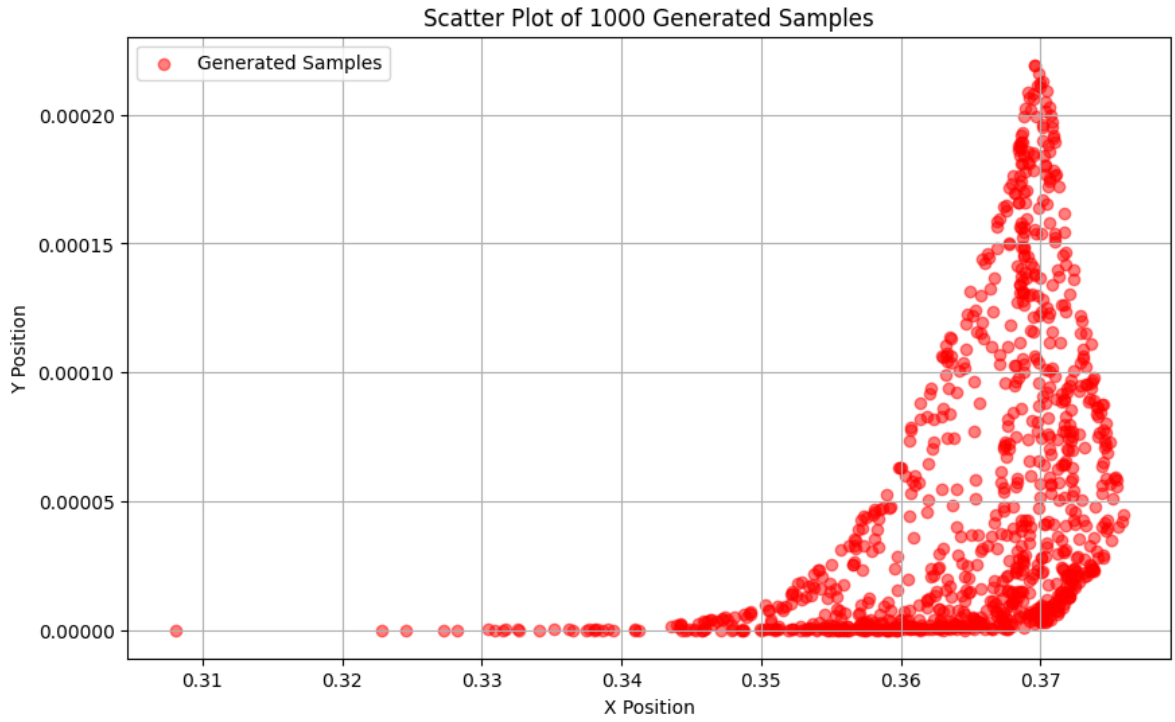


Figure 29: Scatter plot of 1000 generated sample when zoomed in

4.5 Generate data to estimate the critical number of people for the MI building

The scatter plot with the approximate main entrance are illustrated in **??**

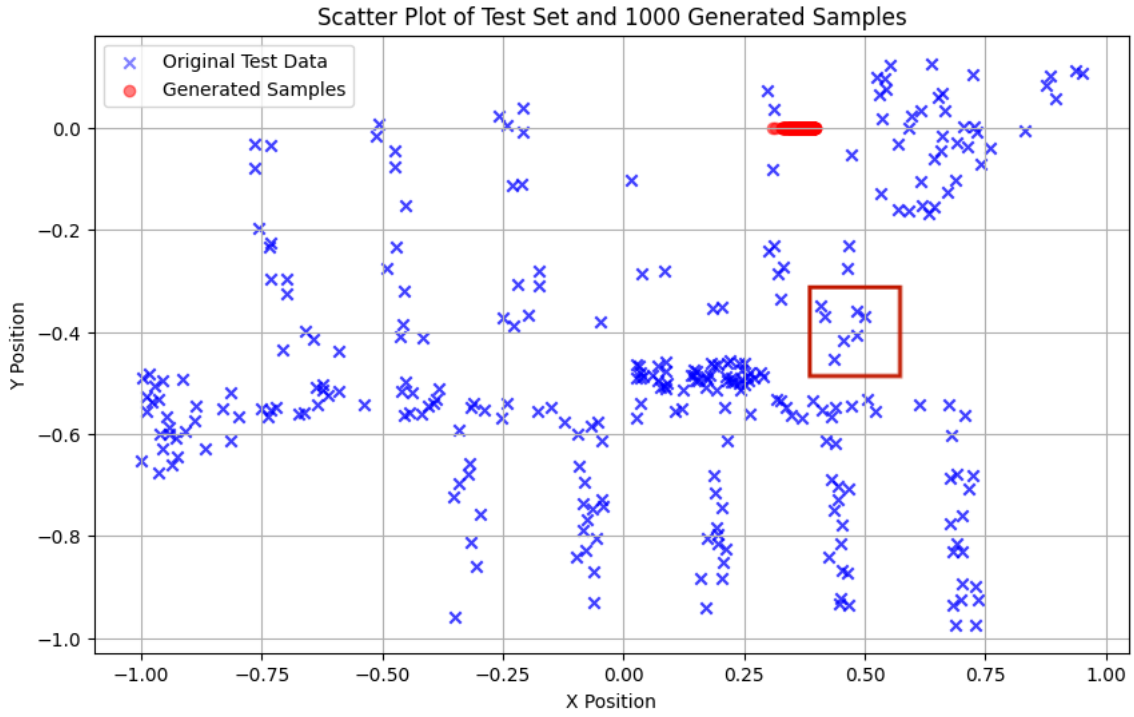


Figure 30: Scatter plot of generated sample, representing people, annotated with the position of main entrance

As we can see from section 4.4, the generated samples does not have any overlap with the critical area (main entrance), so we increased the generated sample to 100000, and the result is still, that there are no overlap between the main entrance (critical area) and the generated sample.

subsectionAnalysis

The peculiar result of all reconstructed points collapsing to a single point led to several key insights:

Latent Space Representation The VAE's latent space collapsed to a single point, indicating a potential issue with the capacity of the latent space or the regularization term (KL divergence) dominating the reconstruction loss.This collapse suggests that the model might not have been trained long enough, or the balance between the reconstruction loss and the KL divergence was not optimal, leading to an underfit model.

Impact of Hyperparameters The learning rate and the number of epochs are crucial in training VAEs. A higher learning rate might have caused the model to converge too quickly to a poor local minimum. Increasing the number of epochs and adjusting the learning rate schedule could help in achieving a better balance between the reconstruction accuracy and the latent space regularization.

Data Normalization

Rescaling the input data to the range of [-1, 1] was a necessary step; however, it also highlights the sensitivity of VAEs to data preprocessing. Different normalization techniques, such as standardization (z-score normalization), might affect the VAE's performance differently and should be explored.

Model Complexity The chosen model architecture (2-64-64-2) might have been too simplistic for capturing the complexities of the FireEvac dataset. Exploring deeper architectures or architectures with different activation functions (e.g., Leaky ReLU, ELU) could provide more flexibility and better performance.

4.5.1 Conclusion

The VAE's behavior of projecting all inputs to a single mean position indicates several potential limitations: - The latent space might have collapsed due to insufficient training or an inappropriate balance between the reconstruction loss and KL divergence. - The chosen hyperparameters and model architecture might not have been optimal for this particular dataset.

4.6 We have to report:

4.6.1 An estimate on how long it took you to implement and test the method:

The implementation of this task (task4) took about a day, including the analysis of the outcome and the coding.

4.7 How accurate you could represent the data and what measure of accuracy you used

: We were able to accurately visualize the data and make the scatterplot of it, but the interpretation of the result of projection to a single point proves to be challenging.

4.8 What you learned about both the dataset and the method (which is probably different from what the machine learned)

About the Dataset

- **Complex Distribution:** The FireEvac dataset, representing x-y positions of people in the MI building, is a low-dimensional yet complex distribution. The data shows specific areas with higher densities, indicating popular spots within the building.
- **Sensitive Areas:** The dataset includes critical zones, such as the area in front of the main entrance, which need special attention during evacuation planning. These areas should not exceed a certain number of people to ensure safety.
- **Variability:** The positions vary significantly, reflecting real-world scenarios (in this case, the MI building layout) where people are distributed unevenly. This variability is crucial for realistic modeling and evacuation planning.

About the Method

- **Challenges with VAE:** Training the Variational Autoencoder (VAE) on the FireEvac dataset revealed some limitations. The reconstructed test set and generated samples often collapsed onto a single point, indicating that the model failed to learn the distribution adequately.
- **Hyperparameter Sensitivity:** The VAE's performance was highly sensitive to hyperparameter settings. Even with extensive tuning, achieving good reconstruction and generation quality was challenging.
- **Training Dynamics:** The training process, which involved rescaling the input data to a range of [-1, 1], did not converge as expected. The model's failure to represent the data distribution accurately suggests that additional techniques, such as different architectures, regularization, or alternative training strategies, might be necessary.
- **Limitations of the Method:** The observed issues highlight limitations in using a VAE for this particular dataset. The method may require significant modifications or a different approach altogether to handle the complexity and variability of the data effectively.

References

[1] T. Berry, J. R. Cressman, Z. Greguric-Ferencek, and T. Sauer. Time-Scale Separation from Diffusion-Mapped Delay Coordinates. *SIAM Journal on Applied Dynamical Systems*, 12(2): 618649, January 2013.