
# Pattern Recognition Homework (2)

## Bayesian Classification, Quadratic Multiclass Classification & Non-Parametric Density Estimation

***Instructed by Dr. Zohreh Azimifar***

Abbas Mehrbanian

*abbas.mrbn@gmail.com, Stu ID: 40130935*

Reza Tahmasebi

*saeed77t@gmail.com, Stu ID: 40160957*

# Table of Contents

# 1 Bayesian Classification

Bayesian classification is a probabilistic approach to learning and inference based on a different view of what it means to learn from data, in which probability is used to represent uncertainty about the relationship being learnt. Before we have seen any data, our prior opinions about what the true relationship might be are expressed in a probability distribution. After we look at the data, our revised opinions are captured by a posterior distribution. Bayesian learning can produce the probability distributions of the quantities of interest, and make the optimal decisions by reasoning about these probabilities together with observed data.

## 1.1 Parametric Models

Discriminant analysis (DA). In the DA, objects are separated into classes, minimizing the variance within the class and maximizing the variance between classes, and finding the linear combination of the original variables (directions). These directions are called discriminant functions and their number is equal to that of classes minus one. Separations between classes are hyperplanes and the allocation of a given object within one of the classes is based on a maximum likelihood discriminant rule. DA can be considered qualitative calibration methods, and they are the most used methods in authenticity. Instead of calibrating for a continuous variable, calibration is performed for group membership (categories). The resulting models are evaluated by their predictive ability to predict new and unknown samples. The most used algorithm for DA is described below.

### 1.1.1 Linear discriminant analysis

Linear discriminant analysis (LDA) is a simple classification method, mathematically robust, and often produces robust models, whose accuracy is as good as more complex methods. LDA assumes that the various classes collecting similar objects (from a given area) are described by multivariate normal distributions having the same covariance but different location of centroids within the variable domain.

### 1.1.2 Quadratic discriminant analysis

Quadratic discriminant analysis (QDA) is a general discriminant function with quadratic decision boundaries which can be used to classify data sets with two or more classes.

QDA has more predictability power than LDA but it needs to estimate the covariance matrix for each class.

## 1.2 LDA Implementation

In the first part of project, we implemented the quadratic discriminant analysis (QDA) to classify our datasets.

In the first step we computed the parameters including sigma, means and priors computing means and priors is pretty much same as our previous project implementation which we calculate them per each label. *compute_priors* function accepts labels of our data set and an array of existing classes then computes prior for each class and finally returns an array of size equal to number of classes containing priors. *compute_means* function works in pretty much the same way with only difference that here we accept features as an input too to calculate features data number per each class. This function returns an array of size equal to number of classes containing means for each class.

The aim of LDA is to maximize the between-class variance and minimize the within-class variance, through a linear discriminant function, under the assumption that data in every class are described by a Gaussian probability density function with the same covariance. To compute sigma[1] we used the *cov* function from numpy library which returns estimated covariance matrix, given data and weights. The *compute_sigma* function first calculates x demean (x – mean) for given dataset and returns the covariance of x demean. Covariance indicates the level to which two variables vary together.

To calculate probabilities, we implemented two functions named *compute_probabilities* & *compute_probability*. *compute_probability* calculates probability estimation of given data for given parameters related to a signle class which are sigma, mean and prior. Note that the sigma is same for both classes in LDA which can also be seen in *compute_probabilities* implementation.

The probability is calculated by implementing the following equation. Since all the classes have the same covariance matrix, the quadratic discriminant becomes as follows:

[1] Aka covariance matrix

$$P_{i\ LDA}(x) = g_i(x) = \frac{-1}{2}(x-\mu_i)^T\Sigma^{-1}(x-\mu_i) + \log\left(P(\omega_i)\right)$$

Then we use *compute_probabilities* function to call *compute_probability* for each class with its related parameters and store computed probabilities, in each call, inside a $numpy$ array and return it in the end.

### 1.2.1 Decision boundary

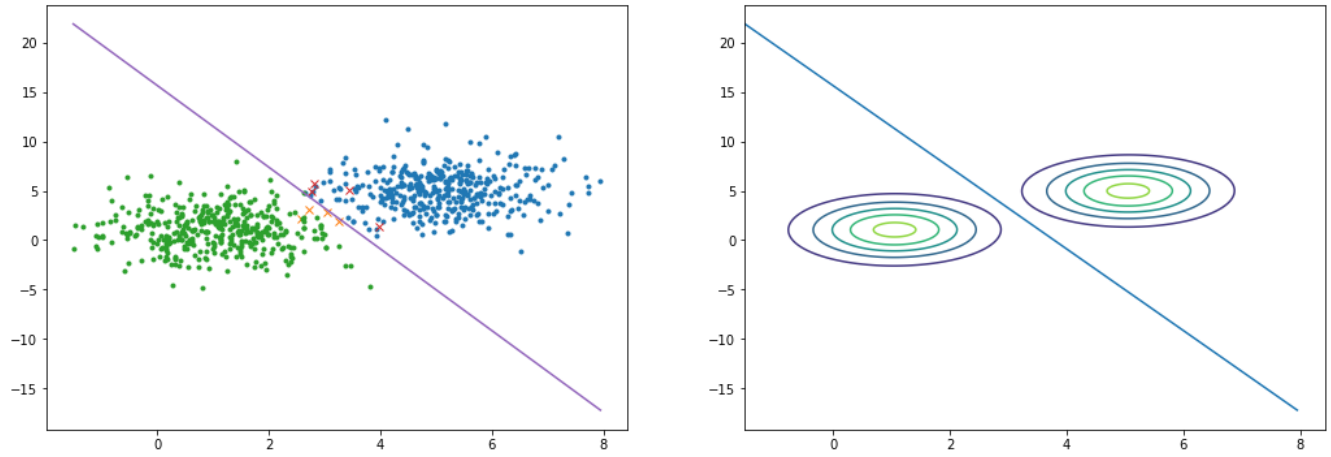
To calculate decision boundary which is a line in this case, we implemented a function that solves the following equation that we learned from class lectures and plots it to given figure.

$$\underbrace{X^T\Sigma^{-1}\left(\mu_i-\mu_j\right)}_{a} + \underbrace{\frac{1}{2}{\mu_i}^T\Sigma^{-1}\mu_i - \frac{1}{2}{\mu_j}^T\Sigma^{-1}\mu_j + \log\left(\frac{P(y=i)}{P(y=j)}\right)}_{b} = 0$$

This implementation is done in *plot_dec_boundary_line* function. This function is later used to plot decision boundary in our scatter and contour pdf plots.

### 1.2.2 Results

We plotted estimated PDFs, Contour estimated PDFs for each dataset separately using the methods discussed in previous section. For scatter plot we represented misclassified samples with a different color and shape.



*(a)* *(b)*

*Figure 1: (a) Train data (1) - Scatter with decision boundary. (b) Train data (1) - 2D Contour estimated PDF with decision boundary*

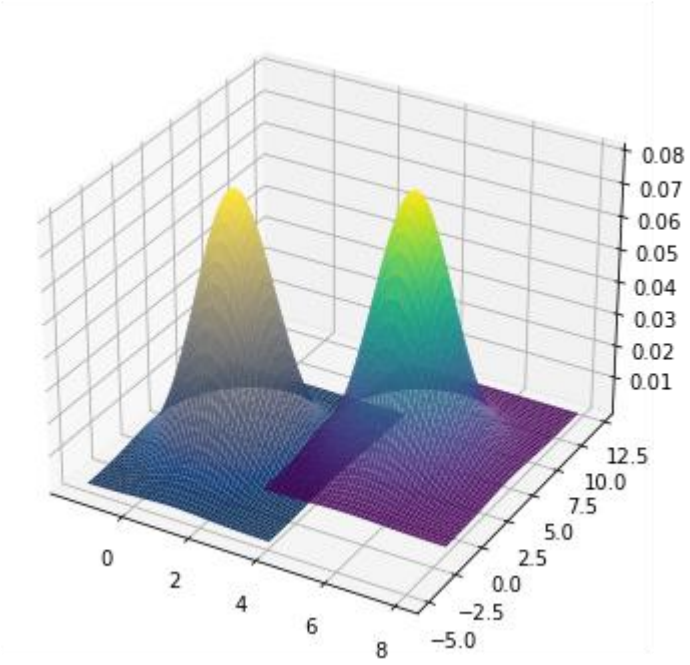
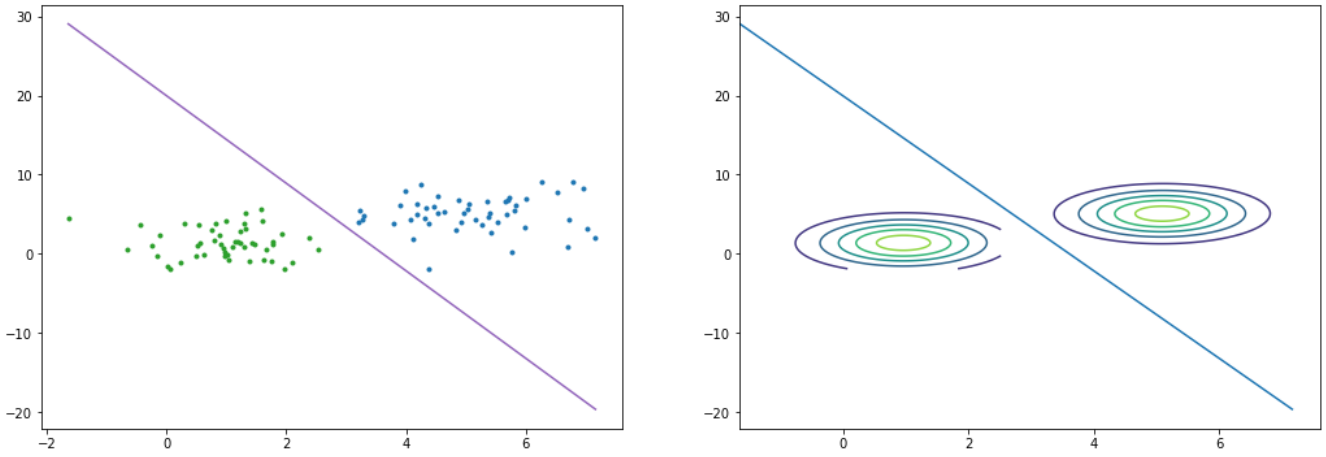



*Figure 2: Train data (1) - 3D estimated PDF*




*(a)* *(b)*

*Figure 3: (a) Test data (1) - Scatter with decision boundary. (b) Test data (1) - 2D Contour estimated PDF with decision boundary*

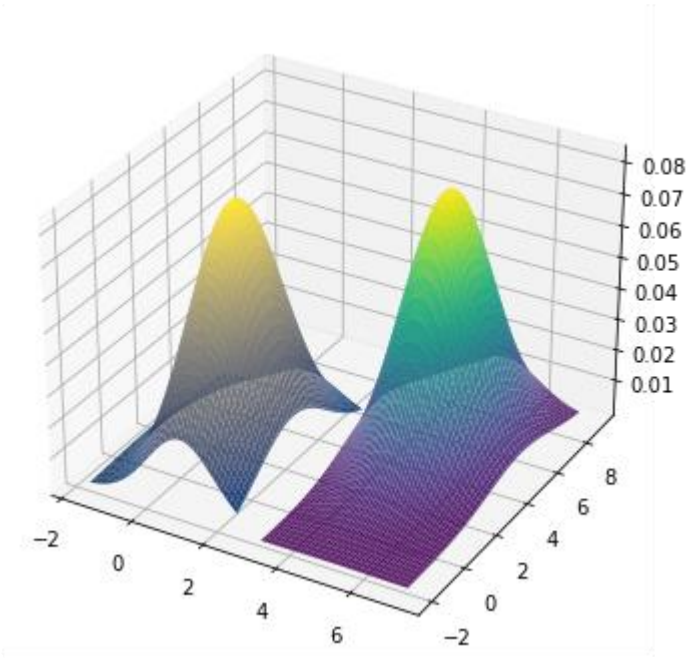
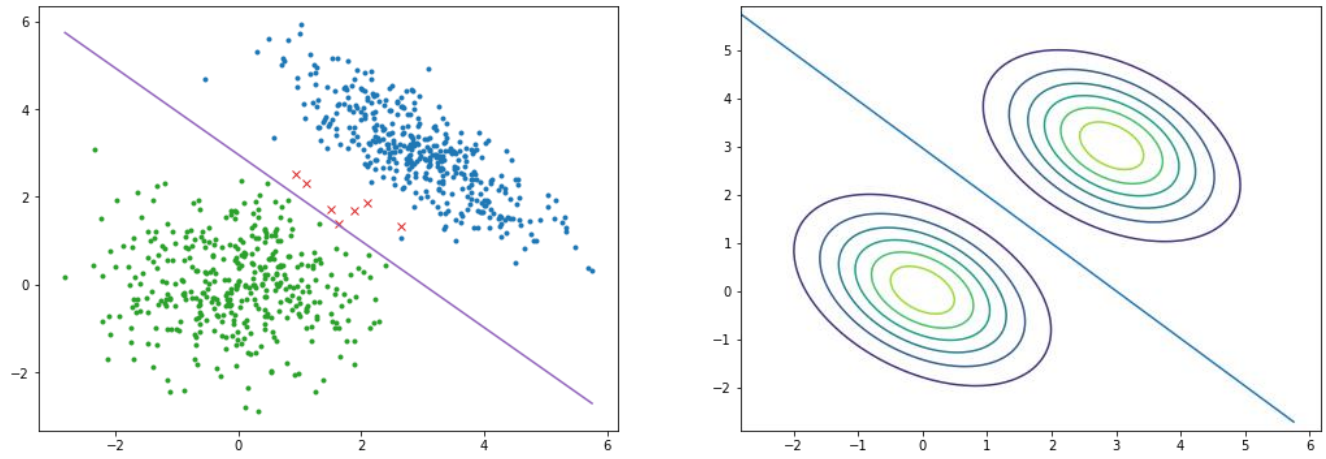



*Figure 4: Test data (1) - 3D estimated PDF*



*(a)* *(b)*

*Figure 5: (a) Train data (2) - Scatter with decision boundary. (b) Train data (2) - 2D Contour estimated PDF with decision boundary*

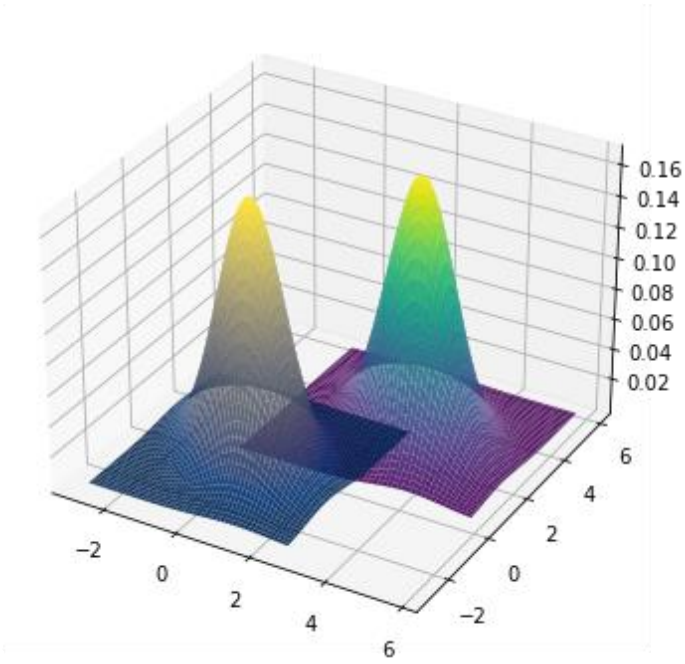
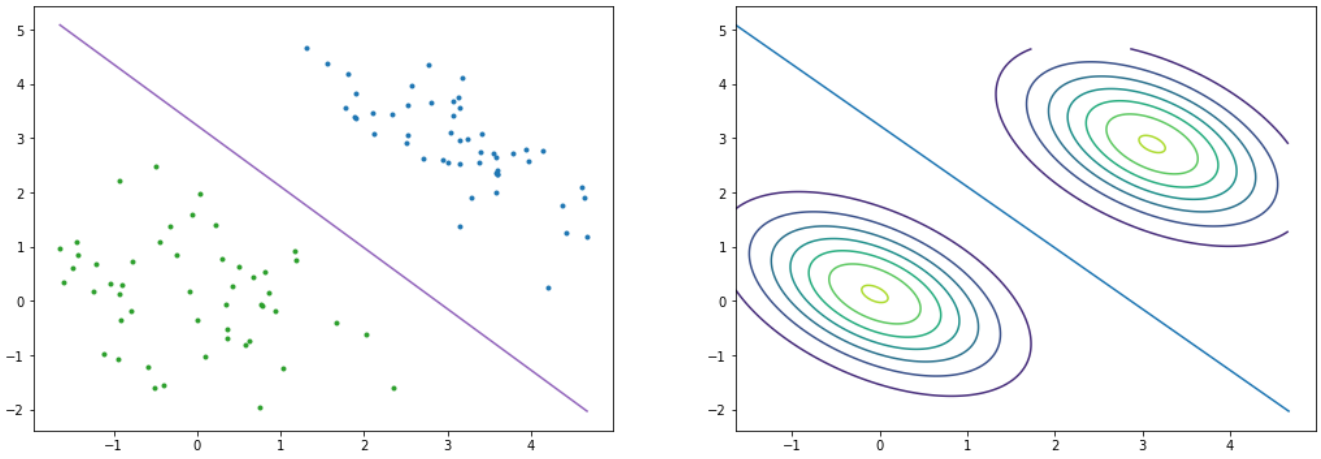



*Figure 6: Train data (2) - 3D estimated PDF*



*(a)* *(b)*

*Figure 7: (a) Test data (2) - Scatter with decision boundary. (b) Test data (2) - 2D Contour estimated PDF with decision boundary*

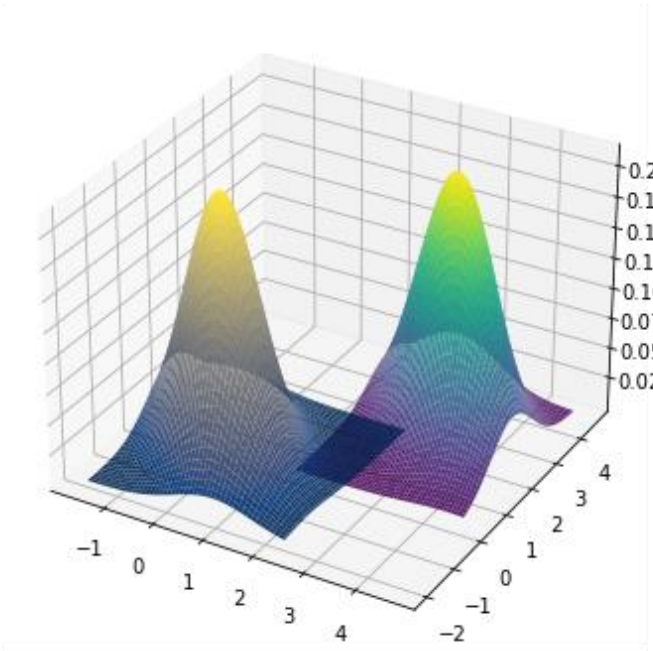



*Figure 8: Test data (2) - 3D estimated PDF*

The accuracies, precisions, recalls and f1 scores for each data set is presented in the following tables. These results were calculated in the same way as previous project (naïve bayes) using calculated confusion matrix.

| Class | Recall | Precision | F1 Score |
|---|---|---|---|
| 0 | 99.0% | 99.0% | 99.0% |
| 1 | 99.0% | 99.0% | 99.0% |
| **Accuracy** | | 99.0% | |

*Table 1: Train dataset (1) results*

| Class | Recall | Precision | F1 Score |
|---|---|---|---|
| 0 | 100% | 100% | 100% |
| 1 | 100% | 100% | 100% |
| **Accuracy** | | 100% | |

*Table 2: Test dataset (1) results*

| Class | Recall | Precision | F1 Score |
|---|---|---|---|
| 0 | 100% | 98.28% | 99.13% |
| 1 | 98.25% | 100% | 99.12% |
| **Accuracy** | | 99.12% | |

*Table 3: Train dataset (2) results*

| Class | Recall | Precision | F1 Score |
|---|---|---|---|
| 0 | 100% | 100% | 100% |
| 1 | 100% | 100% | 100% |
| **Accuracy** | | 100% | |

*Table 4: Test dataset (2) results*

By comparing f1 (b) and f2 (b) we can conclude that gaussian distribution of BC-Train1 dataset is asymmetric, since it has a more likely oval shape. Also, the gaussian distribution of BC-Train2 dataset is symmetric, since it has a more likely circular shape.

## 1.3 QDA Implementation

For the second part of project, we implement the quadratic discriminant analysis (QDA) to classify our datasets.

### 1.3.1 Data generation

In this part we have to generate two datasets for three classes each with 500 samples from three given Gaussian distributions as described in instructions file for each dataset.

To do such task we wrote a function named *generate_dataset* which accepts the following inputs: means, sigmas, samples size for each class, save path as a string. And then creates random values for each class of dataset with the help of *np.random.multivariate_normal* from *numpy* library. This helper function selects random samples from a given multivariate normal distribution. And finally, it saves the generated dataset as a csv file in given path.

Not that we only call this function once and for later runs we simply load out data the same way we did for previous projects.

### 1.3.2 QDA vs. LDA

There a few things which makes the implementation of QDA a bit different from LDA that we'll discuss in this part.

As we know, in the first step, we need to calculate initial parameters which were means, covariance matrixes(sigmas) and priors. The methods used to calculate means and priors are same as what was implemented for LDA. QDA, like LDA, is based on the hypothesis that the probability density distributions are multivariate normal but, in this case, the dispersion is not the same for all of the classes. It follows that the classes differ for the position of their centroid and also for the variance–covariance matrix. To calculate sigma for each class separately we implemented *compute_sigmas* function which computes covariance similar to what we did in *compute_sigma* from LDA but this time for each class separately.

Another difference is in the *compute_probability* and *compute_probabilities* functions. Remember for *compute_probabilities* in LDA implementation we passed a single sigma

to *compute_probability* function in each call. But this times since we have difference sigma value for each class, we pass sigma related to belonging the class.

Since each class has a different covariance matrix, the quadratic discriminant equation becomes as follows:

$$P_{i\ QDA}(x) = g_i(x) = \frac{-1}{2}(x-\mu_i)^T\Sigma^{-1}(x-\mu_i) - \frac{1}{2}\log\left(\left|\Sigma_i\ \right|\right) + \log\left(P(\omega_i)\right)$$

This equation is used to implement our *compute_probability* for QDA.

### 1.3.3 Decision boundary

Unlike LDA here the decision boundary is quadratic. To calculate decision boundary which is a line in this case, first we calculated coefficients in the equation below which we learned from class lectures:

$$\log\left(\frac{P(y=i)}{P(y=j)}\right) - \frac{1}{2}[\,X^T\left(\Sigma_i^{-1} - \Sigma_j^{-1}\right)X + \mu_i{}^T\Sigma^{-1}\mu_i - \mu_j{}^T\Sigma^{-1}\mu_j - 2X^T(\Sigma_i^{-1}\mu_i - \Sigma_j^{-1}\mu_j) = 0$$

$$X^TaX + b^TX + c = 0$$

After finding the coefficients we another function named *solve_quadratic_eq* that we implemented to solve the equation above. This implementation is a part of *plot_decision_boundary_line* which plots the calculated dicision boundary an on plot with given parameters: plot object, features, labels, labels values, means, priors and sigmas of each class.

### 1.3.4 Datasets comparison

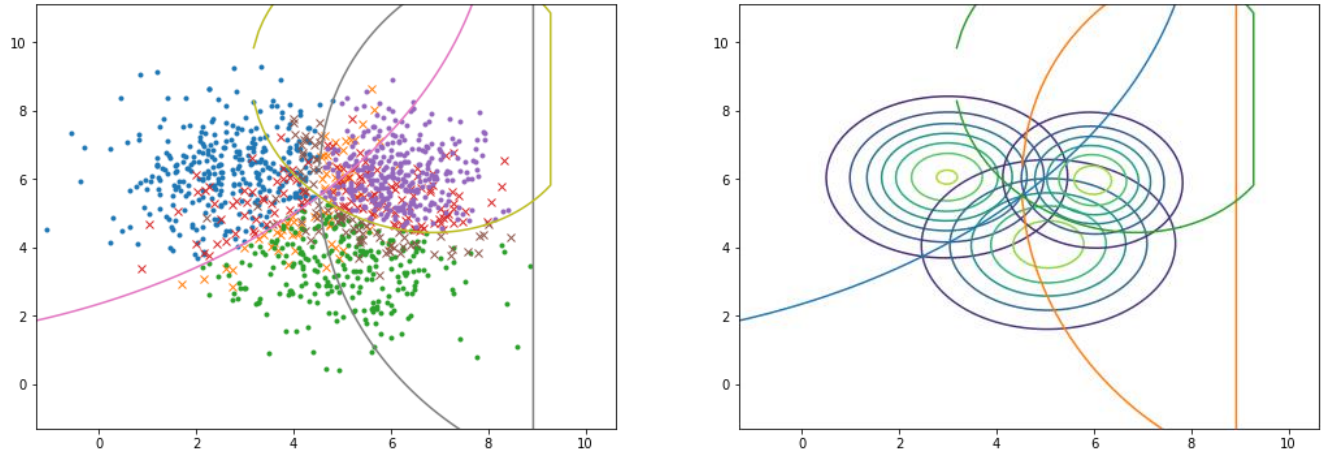
Our first dataset is diagonal, and the second dataset is nondiagonal; we can already see the difference in the contour 2D pdf plots for both datasets; thus, the gaussian distribution of the second dataset will be asymmetric and it'll have a bit rotation since it's none-diagonal. It will be harder to fit on the second data set with an estimated Gaussian distribution. This matter is trivial since the accuracy achieved for the second dataset is less than that achieved for the first. The result is provided in the next section.

We already discussed that we used only 1 covariance matrix in LDA, if we calculate sigma for both classes in the LDA datasets their value will be pretty close to each other that resulted in using LDA with one single sigma and a linear decision boundary. But as

discussed in X the QDA the covariance matrices are difference for each class which resulted in quadratic decision boundrays.

### 1.3.5 Results

We plotted estimated PDFs, Contour estimated PDFs for each dataset separately using the methods discussed in previous section. For scatter plot we represented misclassified samples with a different color and shape.



*(a)* *(b)*

*Figure 9: (a) Train data (1) - Scatter with decision boundary. (b) Train data (1) - 2D Contour estimated PDF with decision boundary*

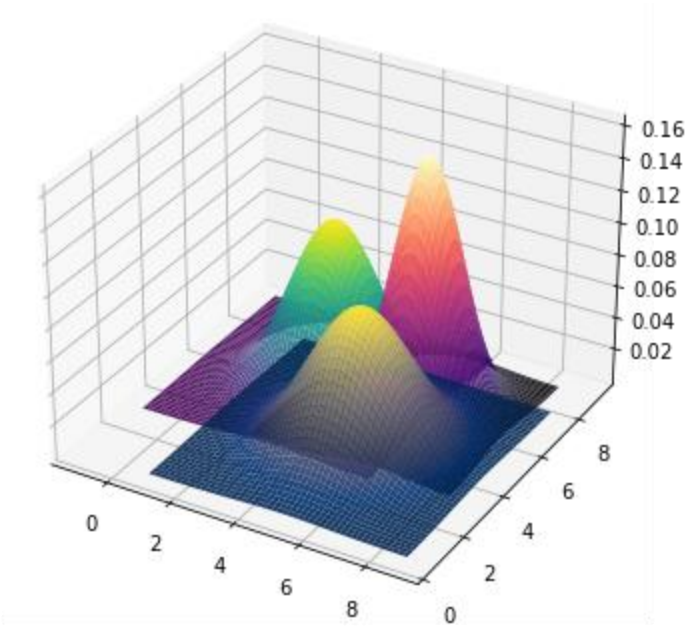
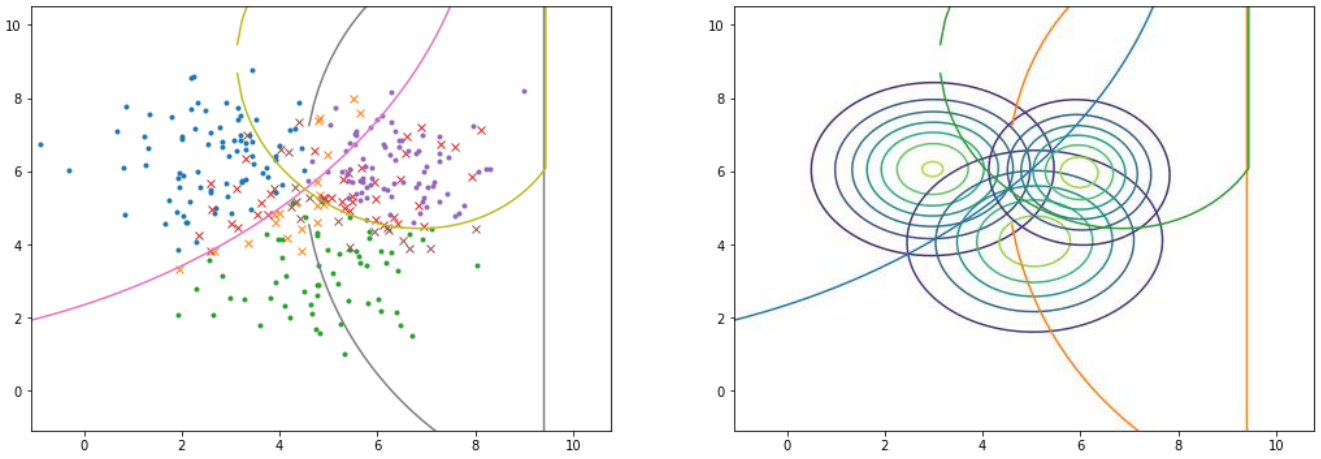



*Figure 10: Train data (1) - 3D estimated PDF*



*(a)* *(b)*

*Figure 11: (a) Test data (1) - Scatter with decision boundary. (b) Test data (1) - 2D Contour estimated PDF with decision boundary*

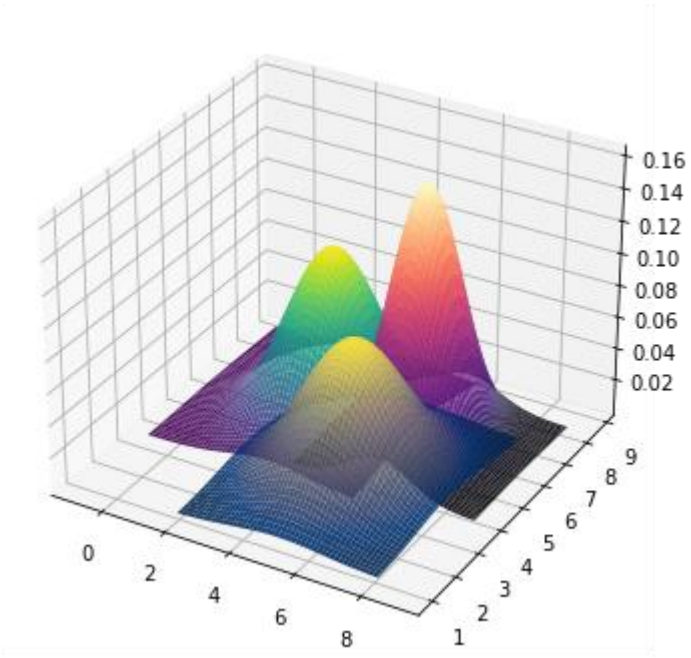
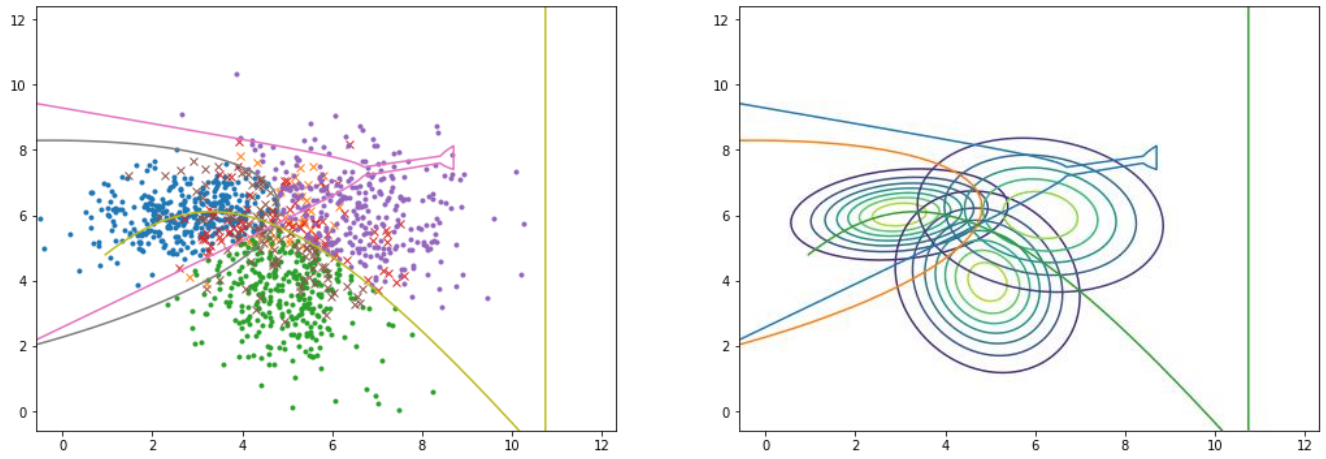



*Figure 12: Test data (1) - 3D estimated PDF*



*(a)* *(b)*

*Figure 13: (a) Train data (2) - Scatter with decision boundary. (b) Train data (2) - 2D Contour estimated PDF with decision boundary*

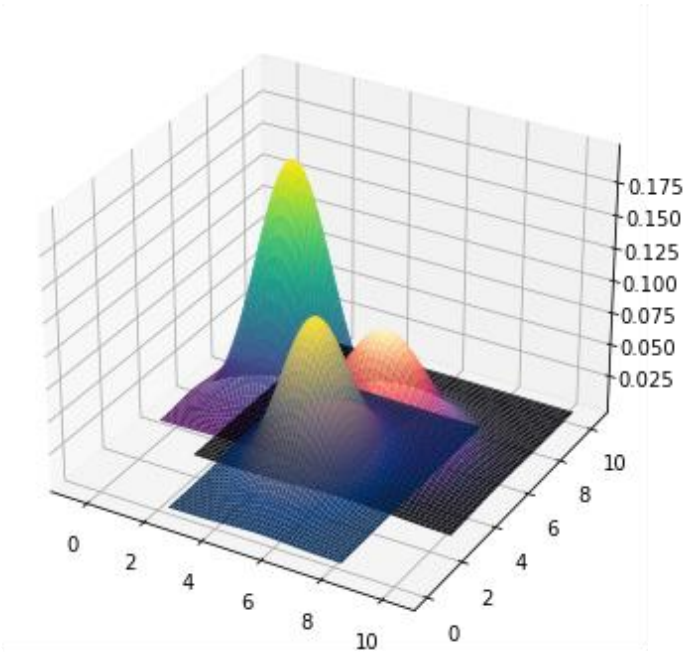
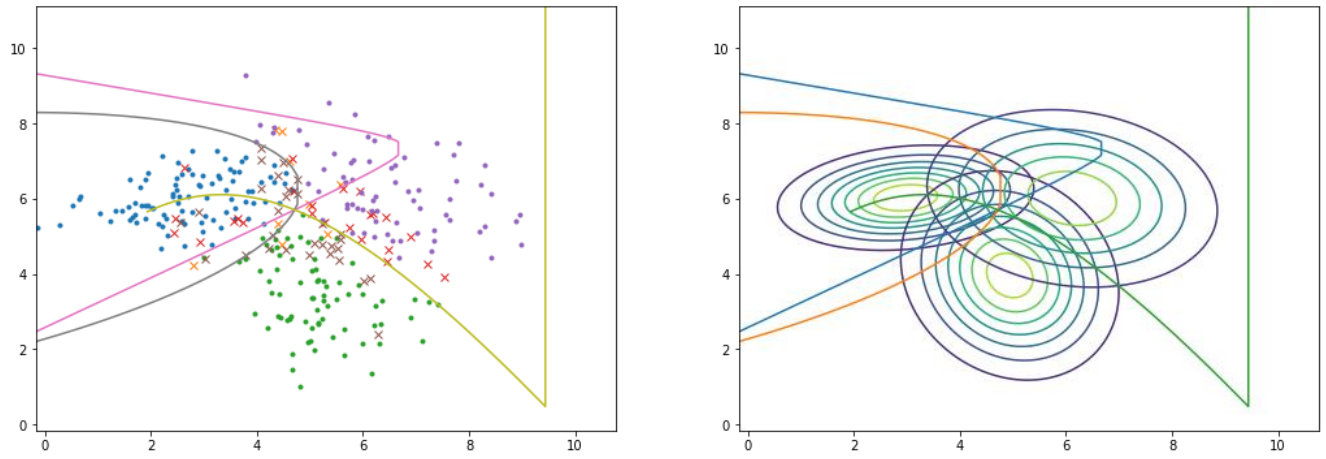



*Figure 14: Train data (2) - 3D estimated PDF*



*(a)* *(b)*

*Figure 15: (a) Test data (2) - Scatter with decision boundary. (b) Test data (2) - 2D Contour estimated PDF with decision boundary*

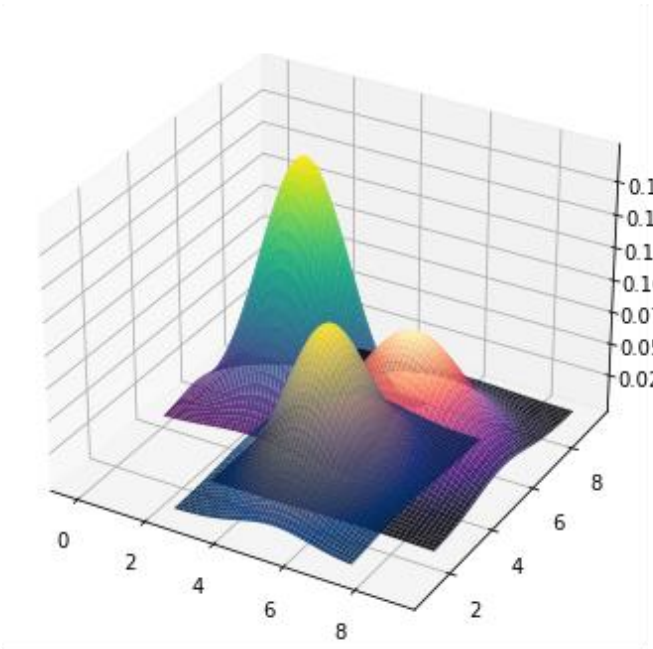



*Figure 16: Test data (2) - 3D estimated PDF*

The accuracies, precisions, recalls and f1 scores for each data set is presented in the next page tables. Again, these results were calculated in the same way as previous project (naïve bayes) using calculated confusion matrix.

| Class | Recall | Precision | F1 Score |
|---|---|---|---|
| 0 | 82.25% | 82.66% | 82.46% |
| 1 | 63.0% | 73.04% | 67.65% |
| 2 | 80.75% | 70.68% | 75.38% |
| **Accuracy** | | 75.33% | |

*Table 5: QDA dataset (1) train results*

| Class | Recall | Precision | F1 Score |
|---|---|---|---|
| 0 | 80.0% | 77.67% | 78.82% |
| 1 | 58.0% | 76.32% | 65.91% |
| 2 | 86.0% | 71.07% | 77.83% |
| **Accuracy** | | 74.67% | |

*Table 6: QDA dataset (1) test results*

| Class | Recall | Precision | F1 Score |
|---|---|---|---|
| 0 | 89.5% | 83.06% | 86.16% |
| 1 | 78.25% | 81.51% | 79.85% |
| 2 | 76.5% | 79.48% | 77.96% |
| **Accuracy** | | 81.42% | |

*Table 7: QDA dataset (2) train results*

| Class | Recall | Precision | F1 Score |
|---|---|---|---|
| 0 | 92.0% | 77.97% | 84.4% |
| 1 | 80.0% | 86.02% | 82.9% |
| 2 | 76.0% | 85.39% | 80.42% |
| **Accuracy** | | 82.67% | |

*Table 8: QDA dataset (2) test results*

# 2 Non-Parametric Density Estimation

## 2.1 Implementation

### 2.1.1 Histogram

To plot histograms based on given h values we implemented a function named *plot_histograms*. This function gets data and h (bandwidth) as arguments. it calculates the minimum and maximum amounts of data, then gets each feature edge with the *np.arange* method and uses the *np.histogram2d* to calculate the data of our histogram.

After all above, the function uses the *np.meshgrid* method to set the X and Y that we want to use to plot the histogram.

The function uses *bar3d*, *colorbar* and *hist* from the *matplotlib* library to plot 3D, color base 3D and 2D histograms, respectively, using the edges and the bins calculated before.

To calculate the density for the unseen data, we need to see in which range it falls and how much data are in that range. Then it multiplies by the product of the total data number and the interval length. In the following formula, h is the interval length, n is the number of train data, and $v_k$ is equal to the number of data is placed in the same interval with a test sample x. We calculate this for each sample in the test data

$$f(x) = \frac{v_k}{n(h_1 h_2 \dots h_d)} \; for\ every\ sample\ x$$

To implement this estimator, we wrote two functions the first one which is *hist_estimate*. This function accepts trained data, test data and h as inputs. it first calculates histogram parameters using *histogram2d* function using train data and h and later uses *estimate_density* function which is the implementation of equation above to calculate estimation for every data point based on test datasets. And it finally returns an array of estimations.

### 2.1.2 KDE Estimator

Our implementation of KDE estimator consists of 4 functions. We implemented the Kernel density estimator equation in form of a function named *KDE* to calculate probabilities ($P_{KDE}(x)$).

$$P_{KDE}(x) = \frac{1}{Nh^D} \sum_{i=1}^{N} k\left(\frac{x - x^{(i)}}{h}\right)$$

This function accepts input data, h which is bandwidth (or smoothing parameter), 2d coordinate matrix from coordinate data vector, sigma and kernel type which is a string to specify using kernel.

Since we are using two different kernel types, we implemented a helper function named *kernel_function*, which calculates $k\left(\frac{x-x^{(i)}}{h}\right)$ part of equation for each data point in 2d coordinate matrix from coordinate data vector. A very popular method for performing multivariate density estimation is the product kernel which is the same method used in this function. This function calls either *gaussian_kernel* or *parzan_window* based on the kernel_type value. As already mentioned, two gaussian and parzan window kernels are implemented in form of functions: *gaussian_kernel* and *parzan_window*.

*parzan_window* function simply accepts a calculated value of $\left(\frac{x-x^{(i)}}{h}\right)$ as $u$ and a bandwidth value which is 0.5 by default and returns 1 if $|u| < 0.5$ and else returns 0.

*gaussian_kernel* function accepts and real number which is a calculated value of $\left(\frac{x-x^{(i)}}{h}\right)$ and also a sigma value. And the it returns a value calculated by the following equation:

$$K(x) = \frac{1}{\sigma^2\sqrt{2\pi}} e^{\frac{-1}{2}(\frac{x}{\sigma})^2}$$

### 2.1.3 KNN Estimator

Our implementation of KNN estimator consists of 2 functions. we implemented this estimator as a function named *KNN* which accepts 2d coordinate matrix created from coordinate data vector and number of neighbors.

$$P(x) = \frac{k}{NV} = \frac{k}{N\, c_D R_k^D(x)}$$

- $R_k(x)$ is the distance between the estimation point and its k-th closest neighbor.
- $c_D$ is the volume of the unit sphere in D dimensions, which is equal to:

$$c_D = \frac{\pi^{D/2}}{\left(\frac{D}{2}\right)!}$$

In our case D = 2 so the equation that we implemented for *KNN* function is as follows:

$$P(x) = \frac{k}{N\pi R_k^2(x)}$$

To calculate $R_k(x)$ we wrote a helper function named *cal_distance* which uses the norm-2 function in numpy (*linalg.norm*) which calculates the Euclidean distance between the estimation point and its k-th closest neighbor. This function accepts 2d coordinate matrix from coordinate features vector, k as number of closest neighbors and a single data point. And finally returns an array of distances as we already discussed.

### 2.1.4 KDE with K-Fold cross validation

K-fold cross-validation is defined as a method for estimating the performance of a model on unseen data. This technique is recommended to be used when the data is scarce and there is an ask to get a good estimate of training and generalization error thereby understanding the aspects such as underfitting and overfitting. This technique is used for hyperparameter tuning such that the model with the most optimal value of hyperparameters can be trained. It is a resampling technique without replacement. The advantage of this approach is that each example is used for training and validation (as part of a test fold) exactly once. This yields a lower-variance estimate of the model performance than the holdout method. As mentioned earlier, this technique is used because it helps to avoid overfitting, which can occur when a model is trained using all of the data. By using k-fold cross-validation, we are able to "test" the model on k different data sets, which helps to ensure that the model is generalizable.

First we randomize the dataset using *shuffle* function from np.random library. Then make the folds set of your data by calling a custum helper function named get_folds we wrote. This function accepts the dataset and a number of folds and creates fold sets using numpy *split* function which cuts the given dataset into k datasets.

Now that we have our fold sets, for each value of h we calculate estimated values using *KDE* function discussed in section 2.1.2 and true density value calculated using *multivariate_normal* function from *spicy* library to true values from a multivariate normal distribution for given prior, mean and sigma. We used equal prior for each class. Since we had 3 classes, we used 1/3.

We chose Euclidean distance as an error calculation criterion here. since we have both true values and estimated ones using KDE, we used *linalg.norm* function from numpy to calculate it for each fold and appended it to an array.

After calculating the error for 5 folds we take an average from them and compare this average error to minimum error we found so far and if it's lower, we simply replace it.

## 2.2 Results

### 2.2.1 True PDFs

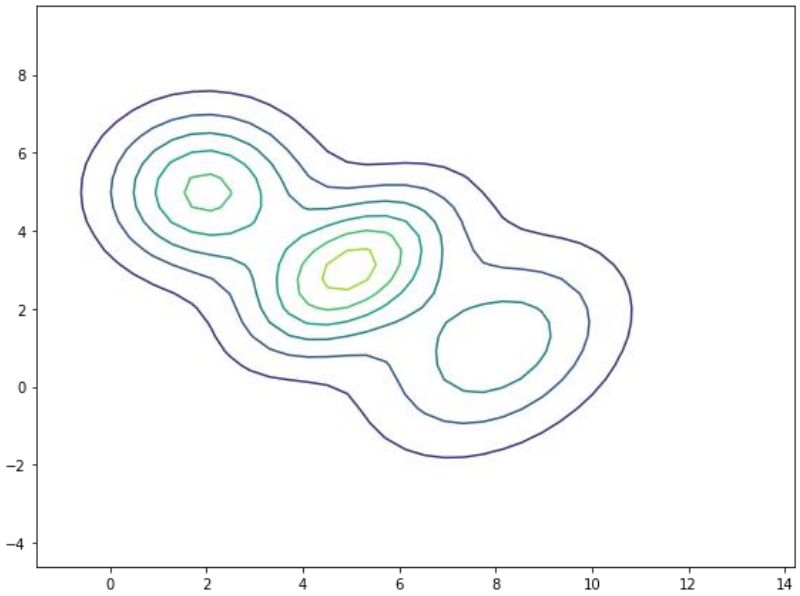



*Figure 17: True 2D contour PDF*

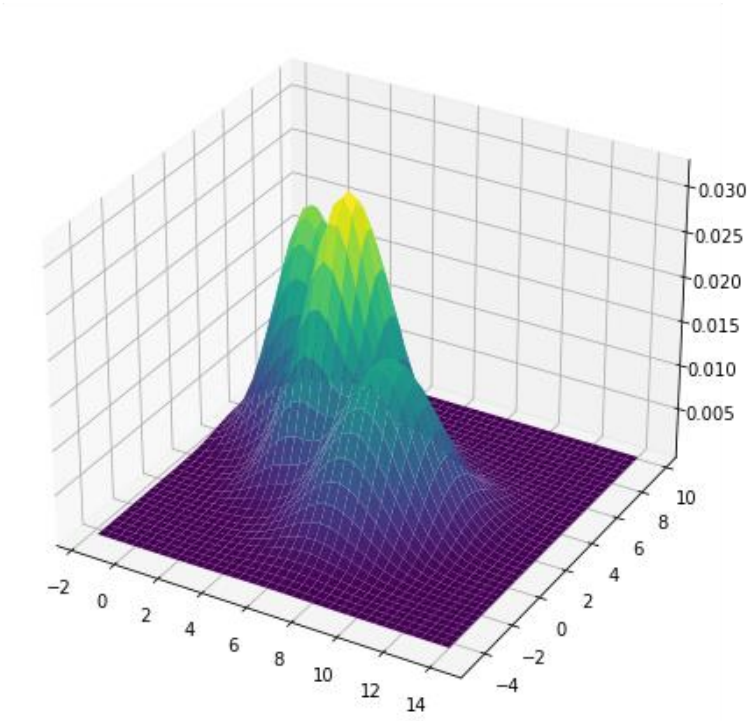
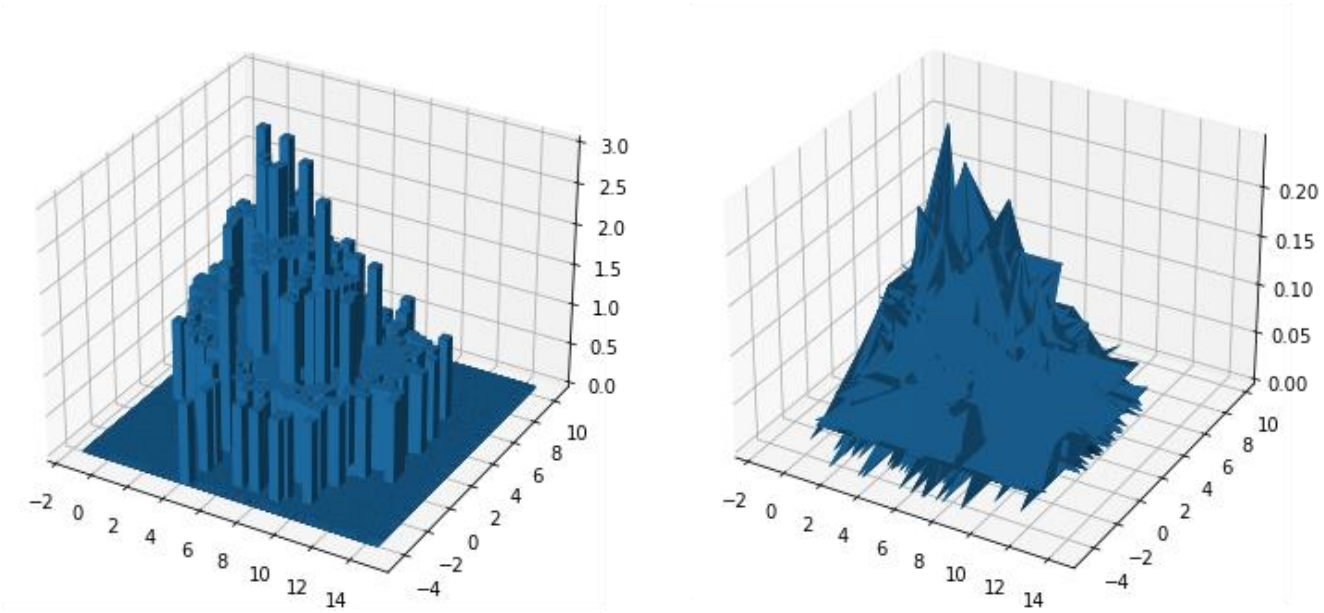


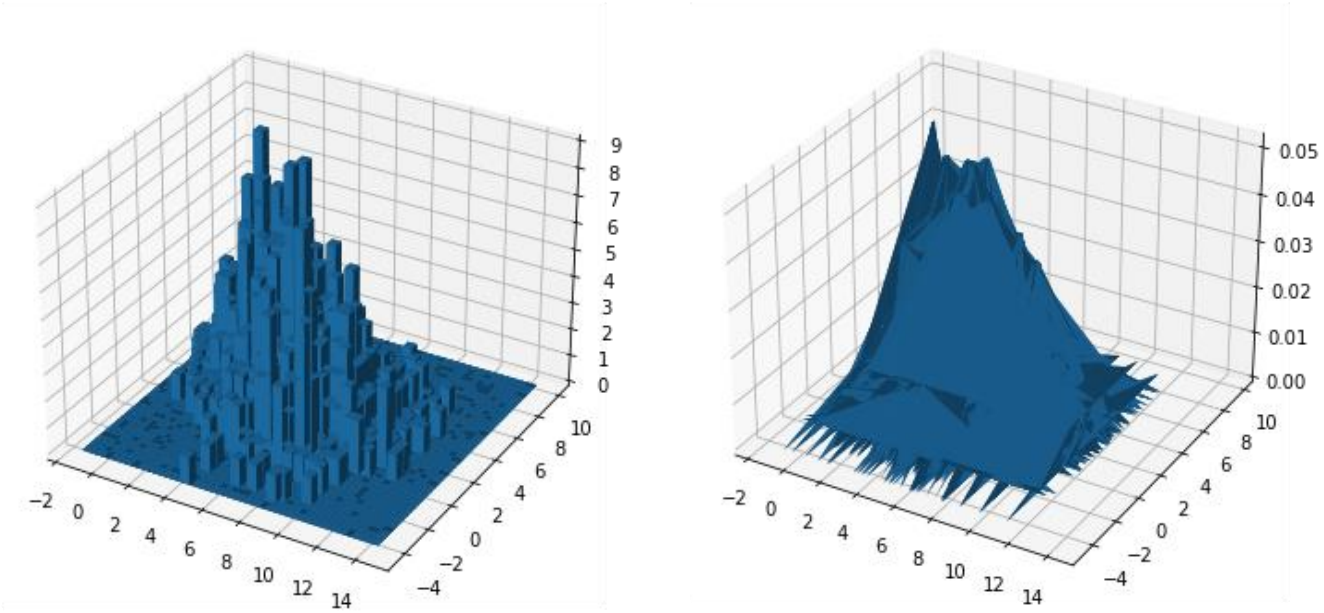

*Figure 18: True 3D PDF*

### 2.2.2 Histogram



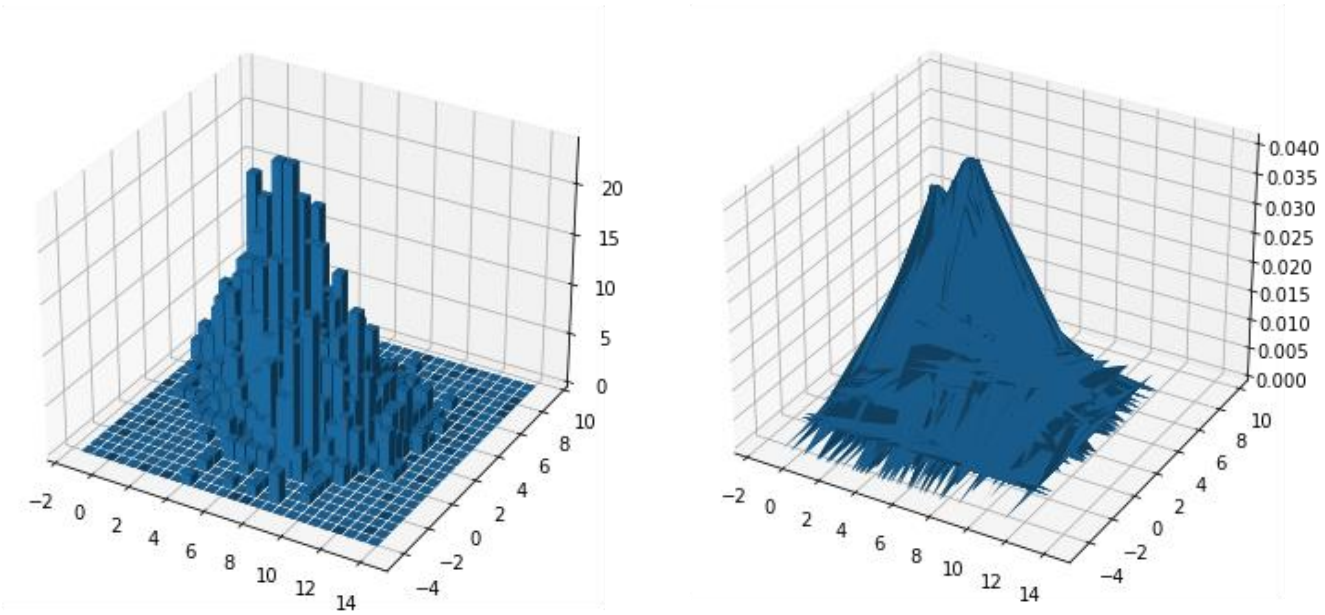

*(a)* *(b)*

*Figure 19: (a) Histogram with h = 0.09 (train data). (b) Estimated PDF (test data)*




*(a)* *(b)*

*Figure 20: (a) Histogram with h = 0.3 (train data). (b) Estimated PDF (test data)*



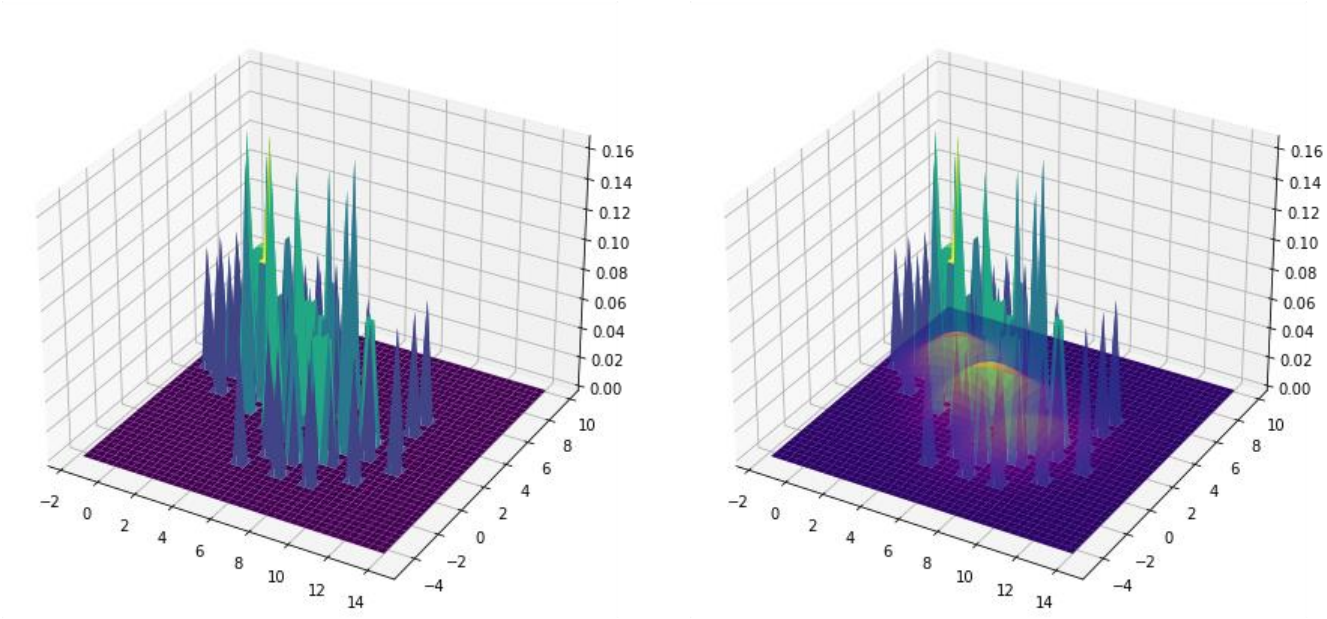

*(a)* *(b)*

*Figure 21: (a) Histogram with h = 0.6 (train data). (b) Estimated PDF (test data)*

It can be seen from the results that as the number of bins increases, the continuity of data space also decreases. So, we might have more bins that are empty.

The closest estimation on test dataset compared to true pdfs is achieved with h = 0.6. We can see as the number of bin decreases(lower value of h) we achieved better results.

### 2.2.3 KDE (Parzan window)




*(a)* *(b)*

*Figure 22: KDE (Parzen Window) 3D PDF with h = 0.09 (a) estimated. (b) estimated vs. true*

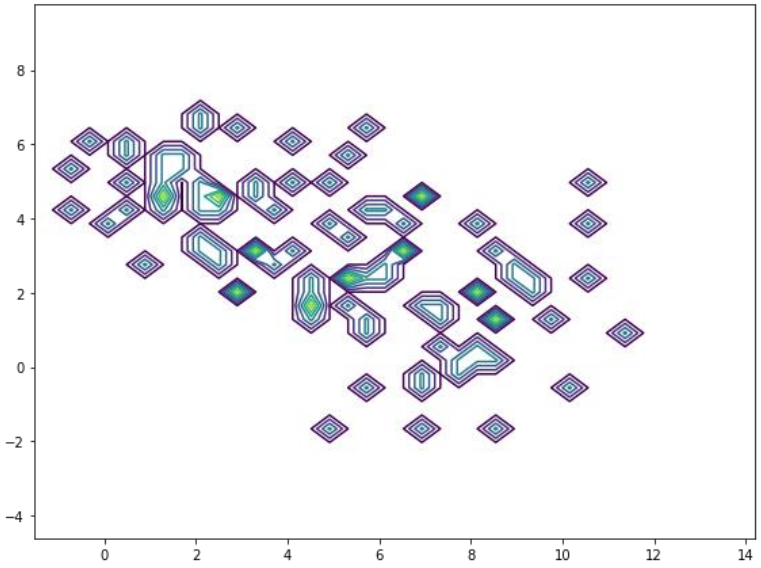
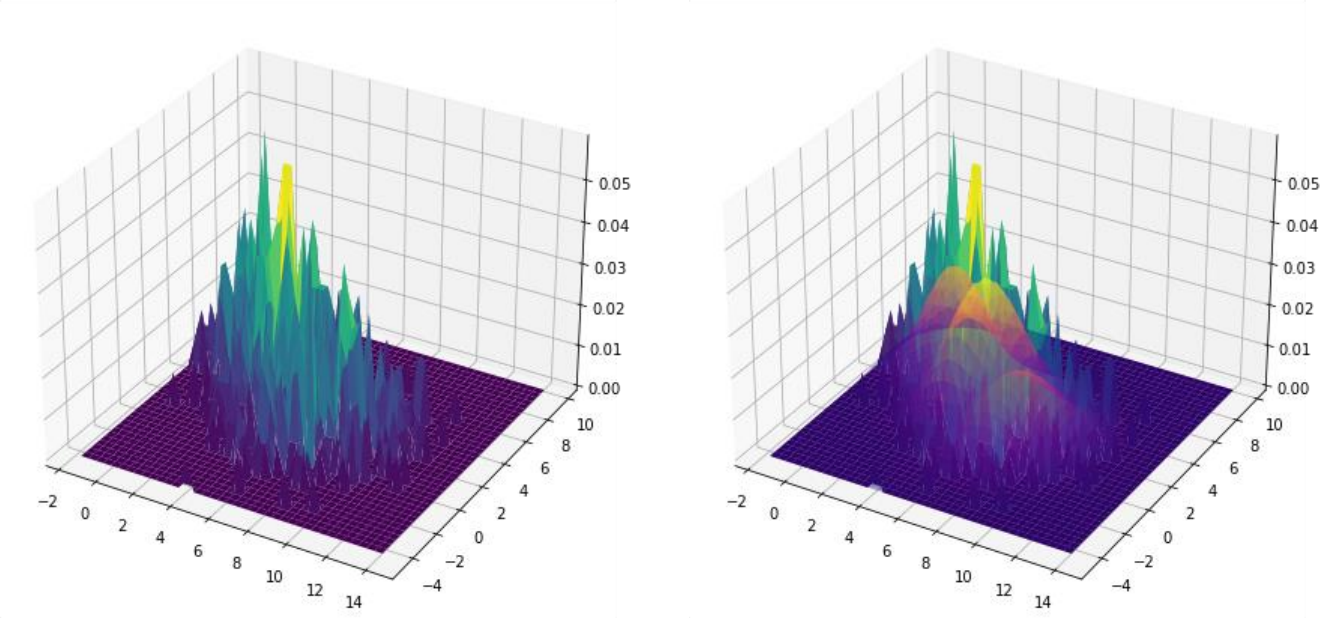


*Figure 23: KDE (Parzen Window) 2D contour estimated PDF with h = 0.09*




*(a)* *(b)*

*Figure 24: KDE (Parzen Window) 3D PDF with h = 0.3 (a) estimated. (b) estimated vs. true*

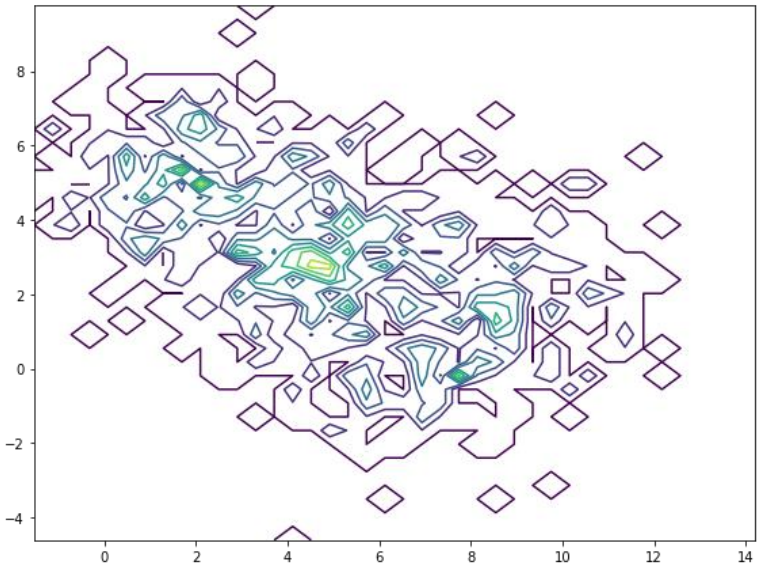
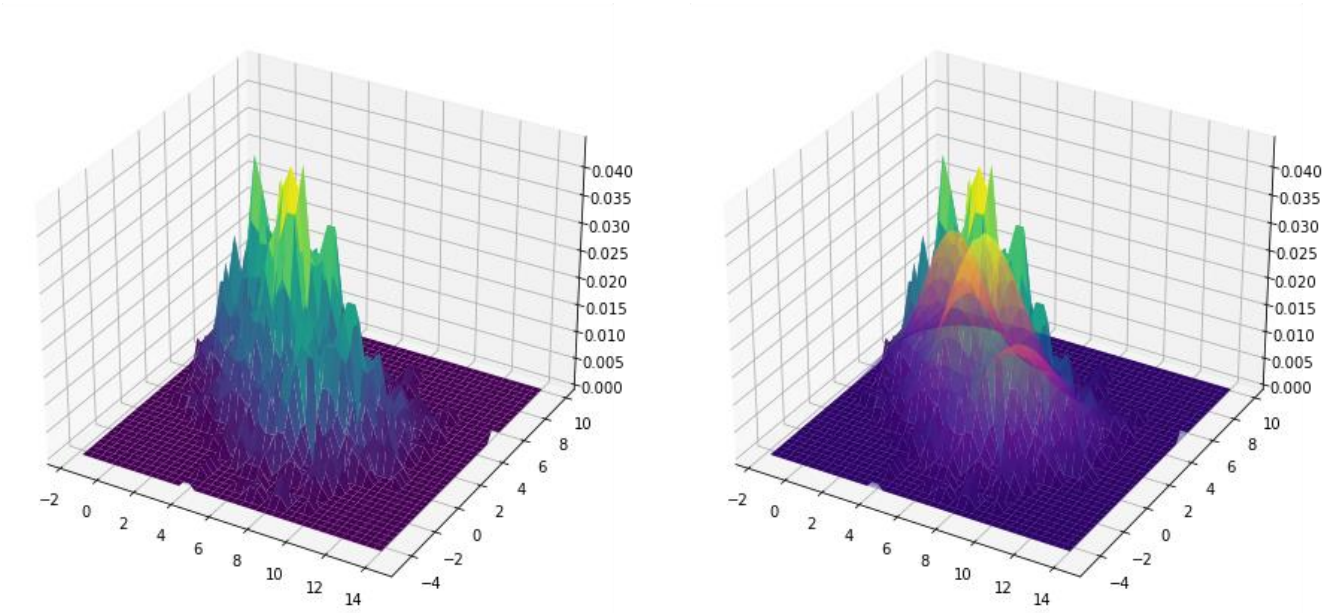



*Figure 25: KDE (Parzen Window) 2D contour estimated PDF with h = 0.3*




*(a)* *(b)*

*Figure 26: KDE (Parzen Window) 3D PDF with h = 0.3 (a) estimated. (b) estimated vs. true*

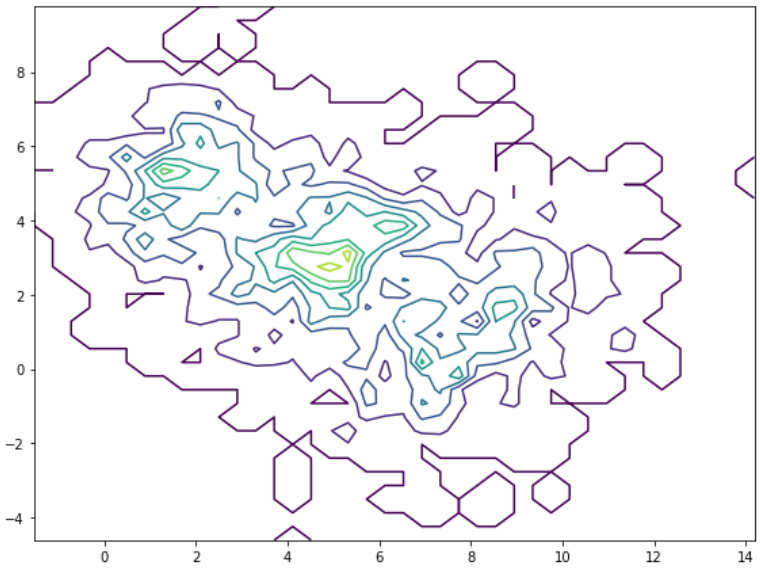
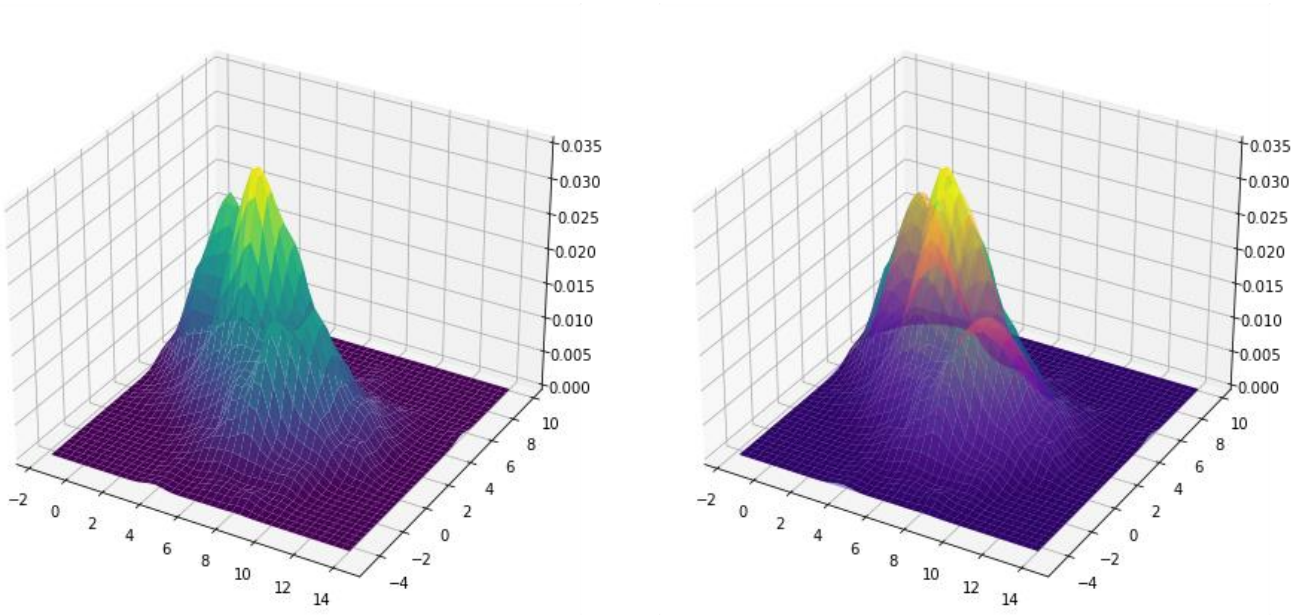


*Figure 27: KDE (Parzen Window) 2D contour estimated PDF with h = 0.6*

Using Parzen window as kernel for KDE, according to the results, the estimation for h = 0.6 showed better performance than the other bandwidths. The results using Parzan window as kernel for KDE seems to be spiky like KNN which we'll see in next sections.

### 2.2.4 KDE (Gaussian Kernel)




*(a)* *(b)*

*Figure 28: KDE (Gaussian Kernel) 3D PDF with h = 0.09 & σ= 0.2 (a) estimated. (b) estimated vs. true*

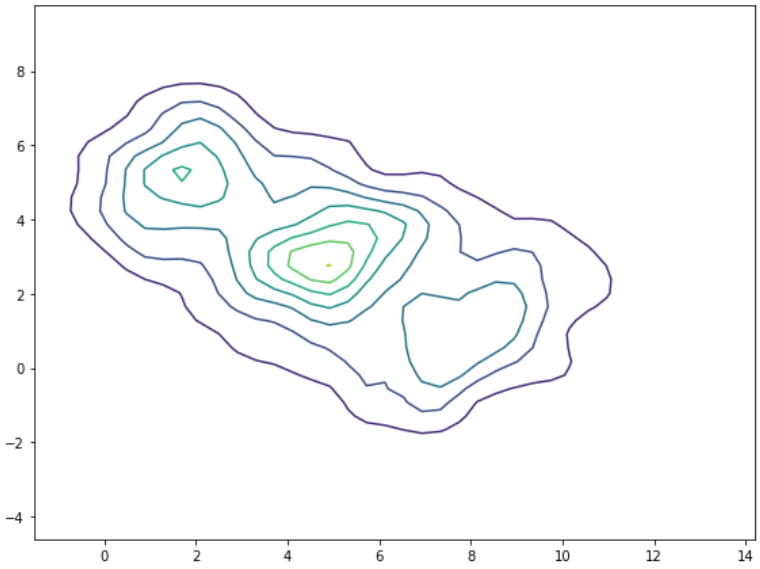
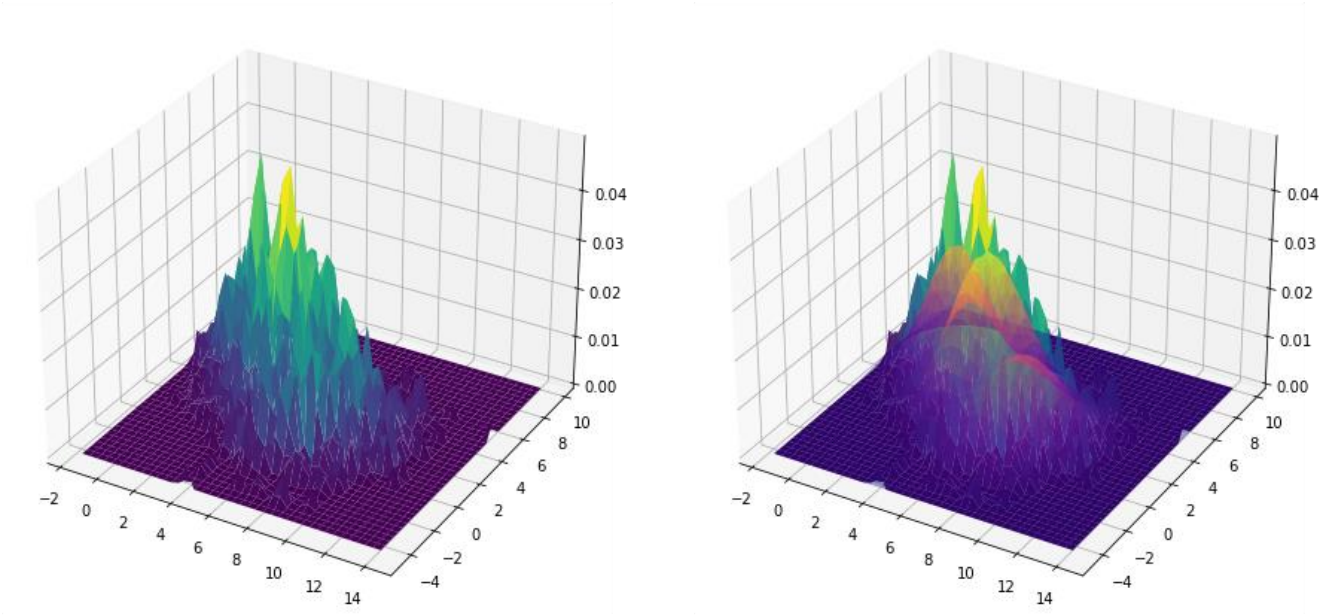


*Figure 29: KDE (Gaussian Kernel) 2D contour estimated PDF with h = 0.09 & σ = 0.2*




*(a)* *(b)*

*Figure 30: KDE (Gaussian Kernel) 3D PDF with h = 0.09 & σ=0.6 (a) estimated. (b) estimated vs. true*

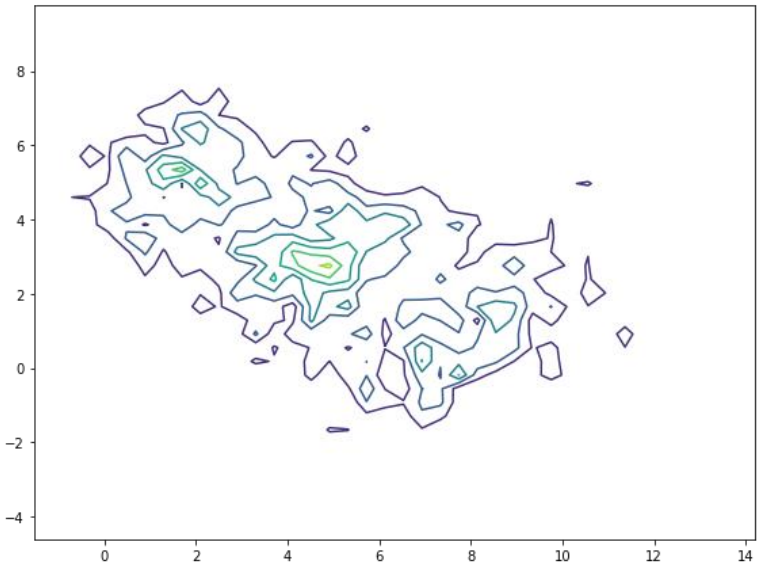
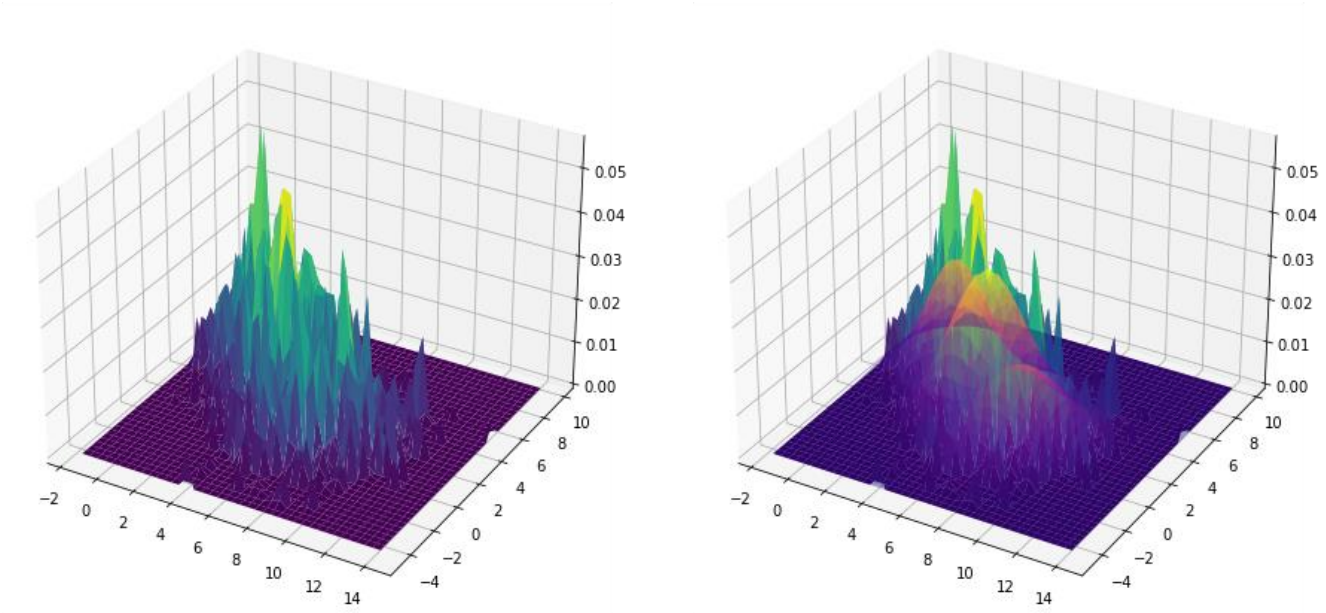



*Figure 31: KDE (Gaussian Kernel) 2D contour estimated PDF with h = 0.09 & σ = 0.6*




*(a)* *(b)*

*Figure 32: KDE (Gaussian Kernel) 3D PDF with h = 0.09 & σ =0.9 (a) estimated. (b) estimated vs. true*

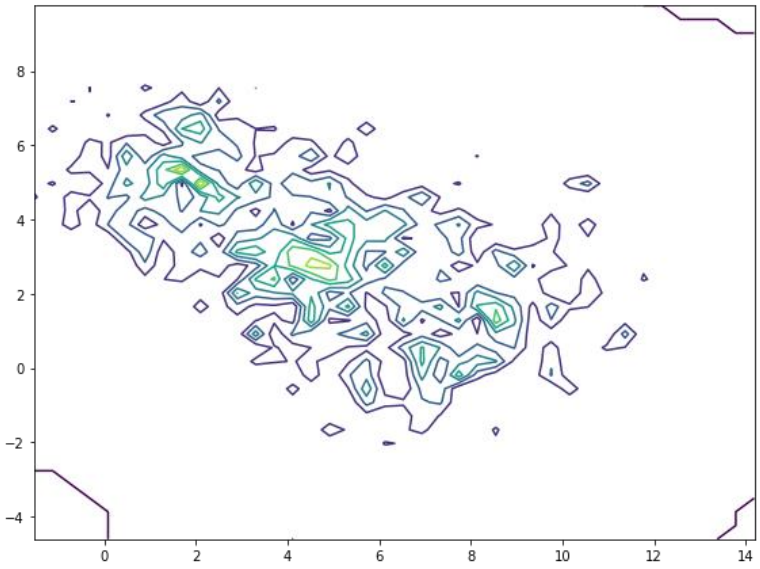
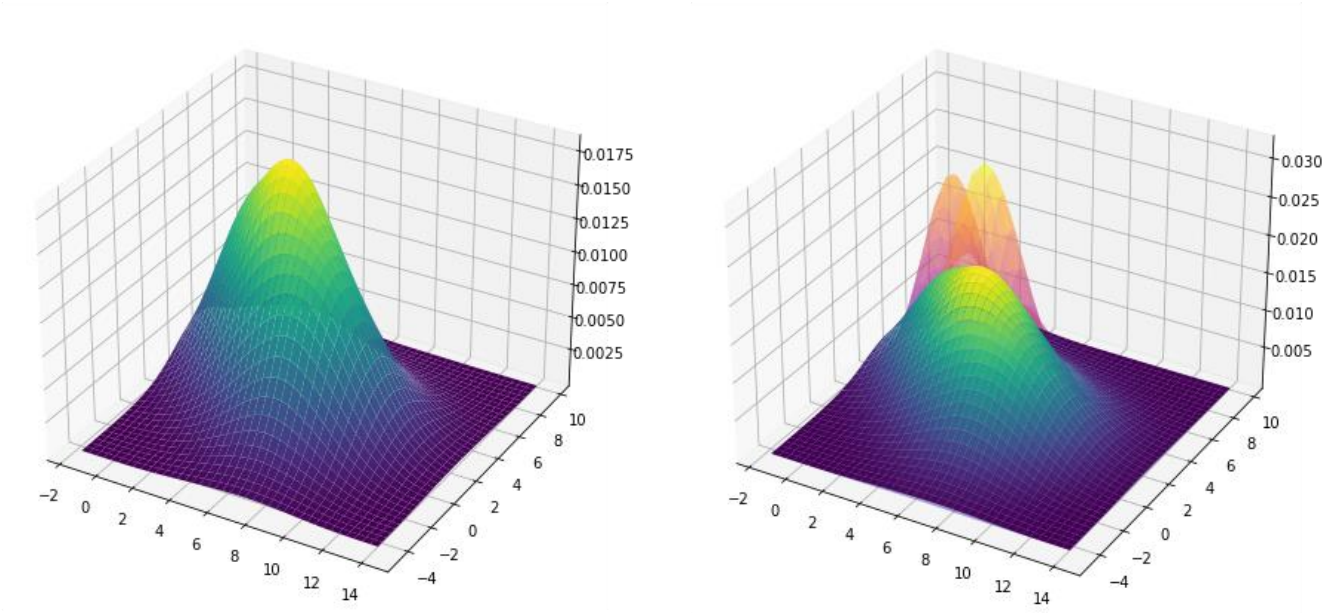



*Figure 33: KDE (Gaussian Kernel) 2D contour estimated PDF with h = 0.09 & σ = 0.9*




*(a)* *(b)*

*Figure 34: KDE (Gaussian Kernel) 3D PDF with h = 0.3 & σ = 0.2 (a) estimated. (b) estimated vs. true*

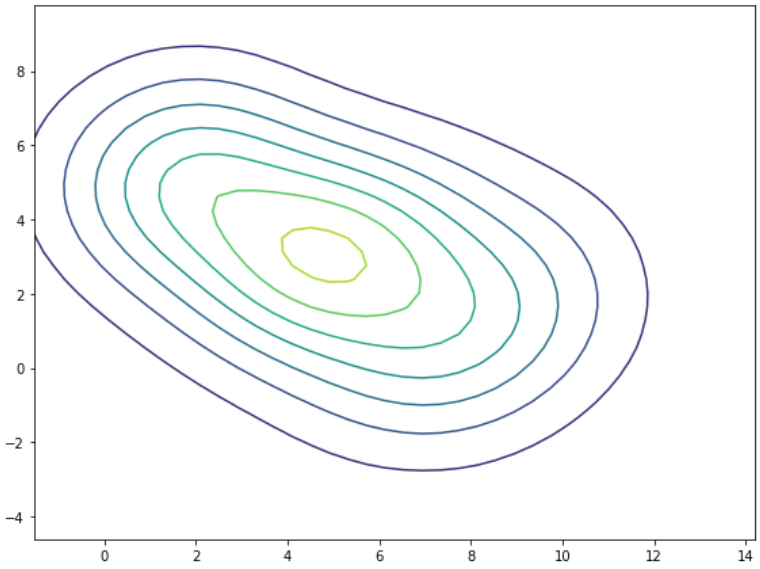
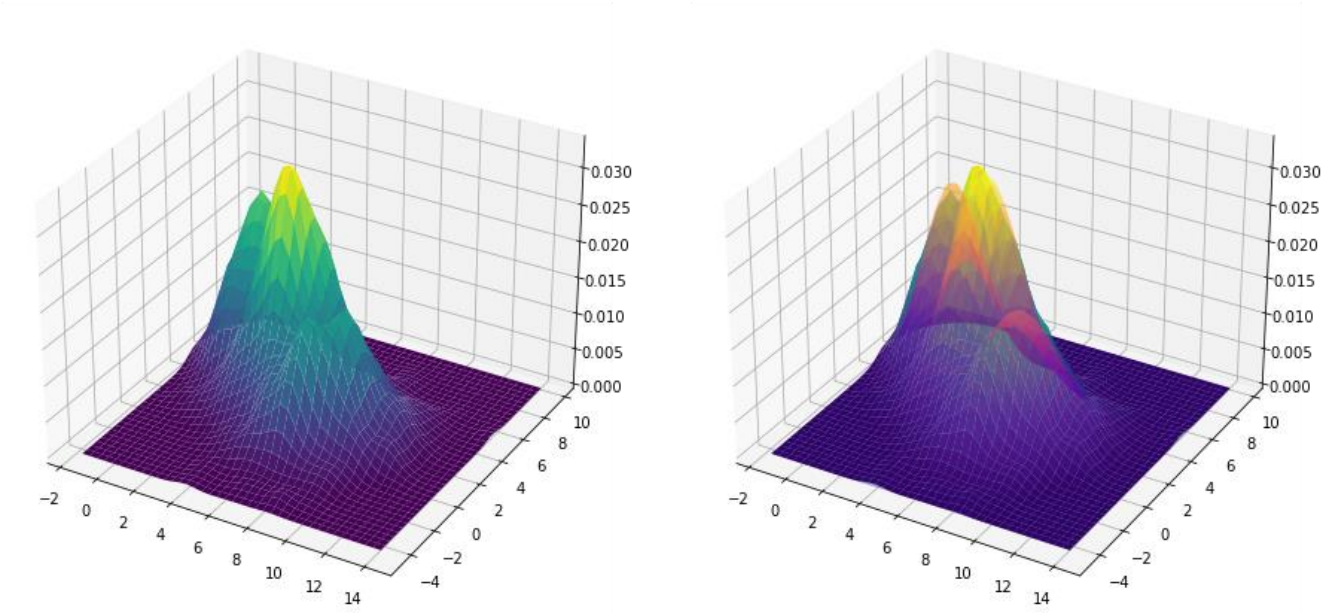


*Figure 35: KDE (Gaussian Kernel) 2D contour estimated PDF with h = 0.3 & σ = 0.2*




*(a)* *(b)*

*Figure 36: KDE (Gaussian Kernel) 3D PDF with h = 0.3 & σ =0.6 (a) estimated. (b) estimated vs. true*

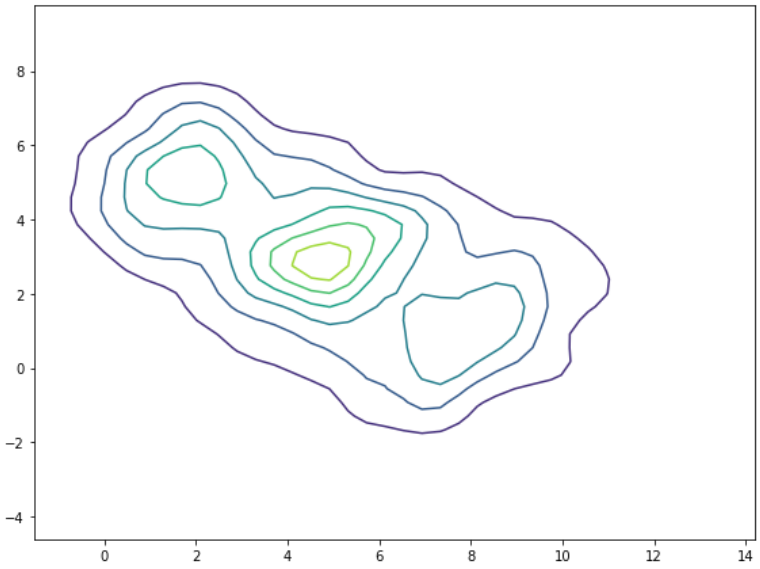
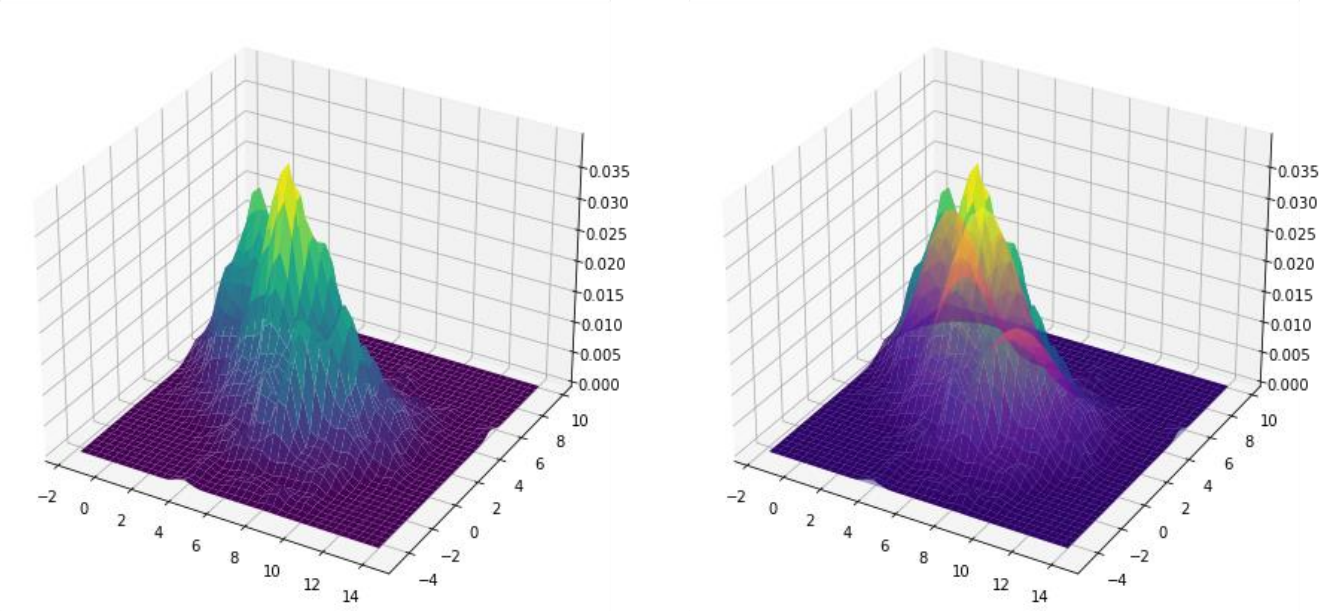



*Figure 37: KDE (Gaussian Kernel) 2D contour estimated PDF with h = 0.3 & σ = 0.6*




*(a)* *(b)*

*Figure 38: KDE (Gaussian Kernel) 3D PDF with h = 0.3 & σ =0.9 (a) estimated. (b) estimated vs. true*

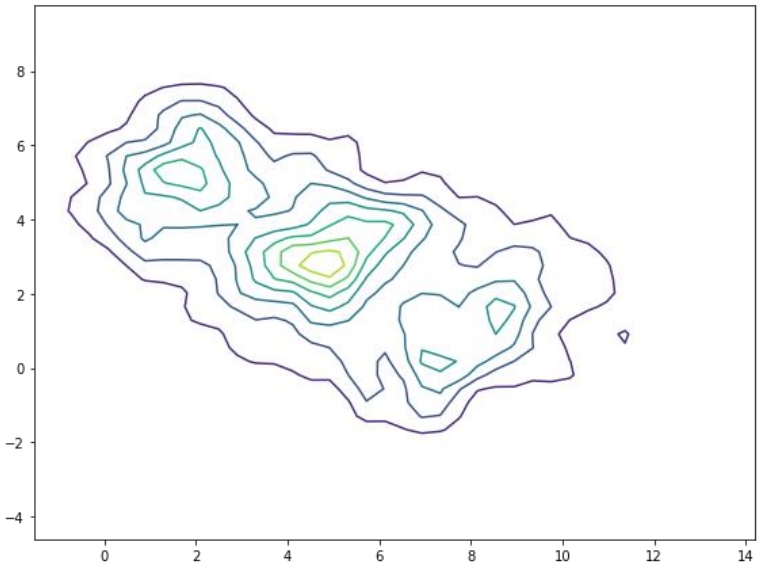
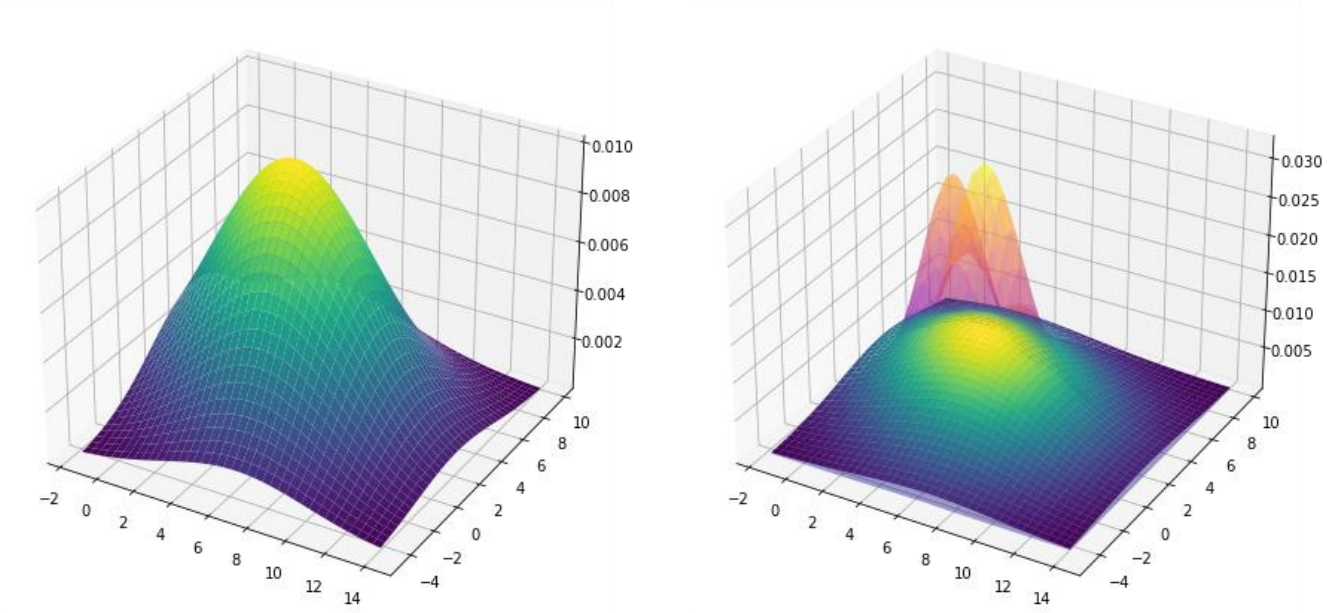



*Figure 39: KDE (Gaussian Kernel) 2D contour estimated PDF with h = 0.3 & σ = 0.9*




*(a)* *(b)*

*Figure 40: KDE (Gaussian Kernel) 3D PDF with h = 0.6 & σ =0.2 (a) estimated. (b) estimated vs. true*

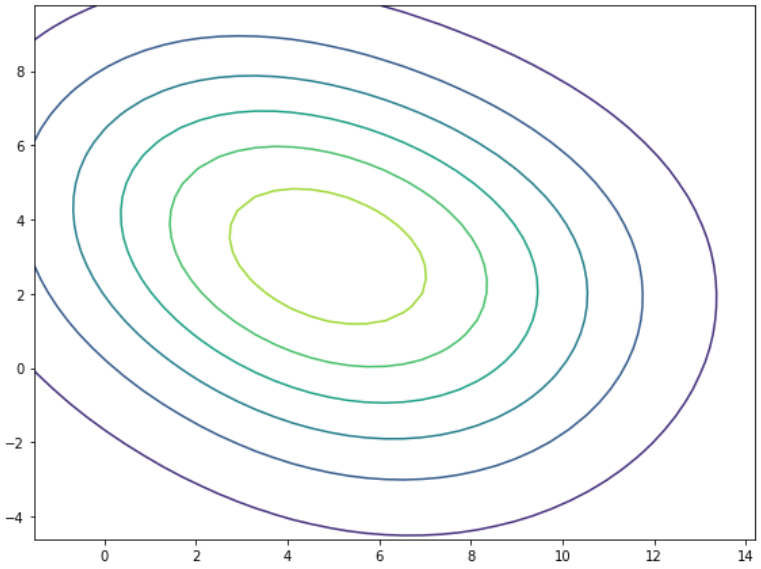
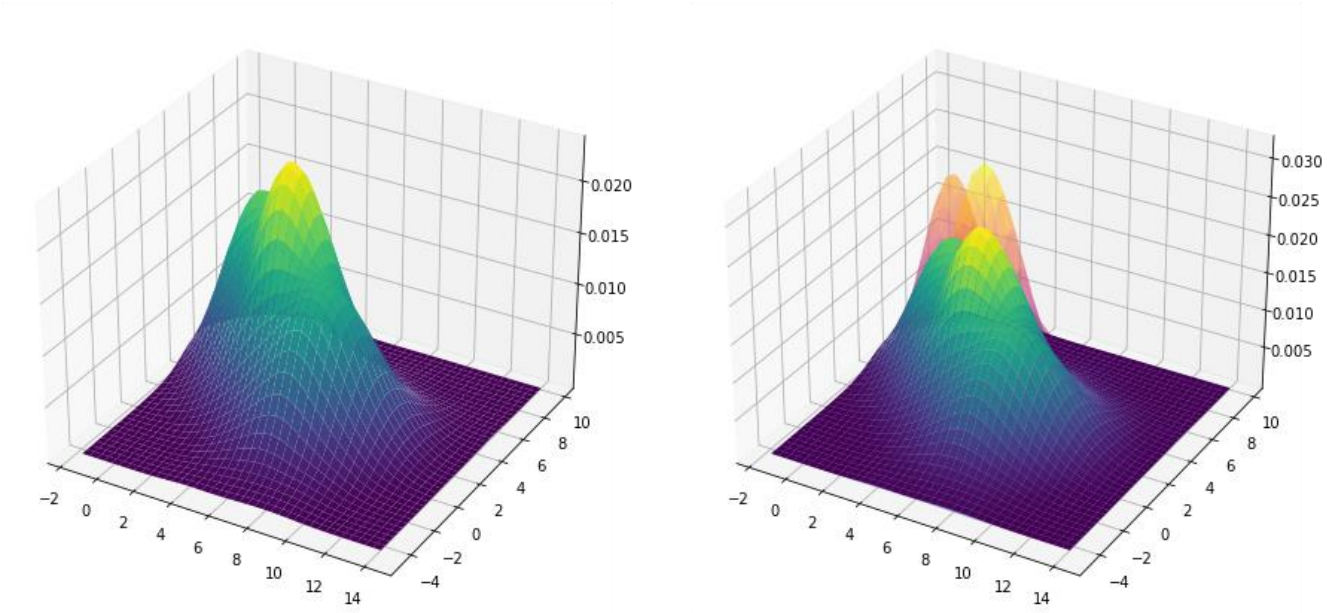


*Figure 41: KDE (Gaussian Kernel) 2D contour estimated PDF with h = 0.6 & σ = 0.2*




*(a)* *(b)*

*Figure 42: KDE (Gaussian Kernel) 3D PDF with h = 0.6 & σ =0.6 (a) estimated. (b) estimated vs. true*

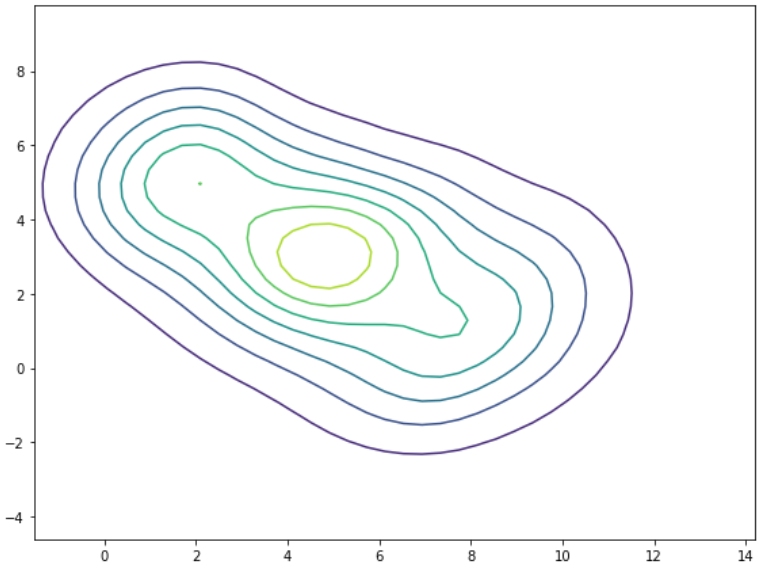
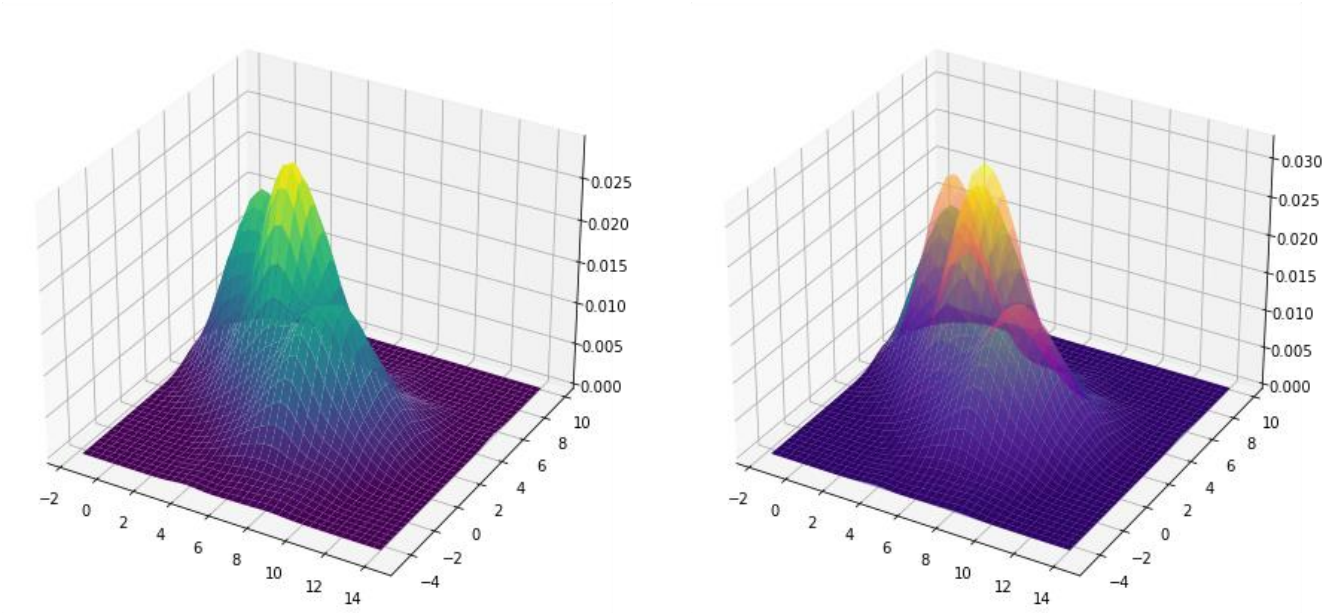


*Figure 43: KDE (Gaussian Kernel) 2D contour estimated PDF with h = 0.6 & σ = 0.6*




*(a)* *(b)*

*Figure 44: KDE (Gaussian Kernel) 3D PDF with h = 0.6 & σ =0.9 (a) estimated. (b) estimated vs. true*

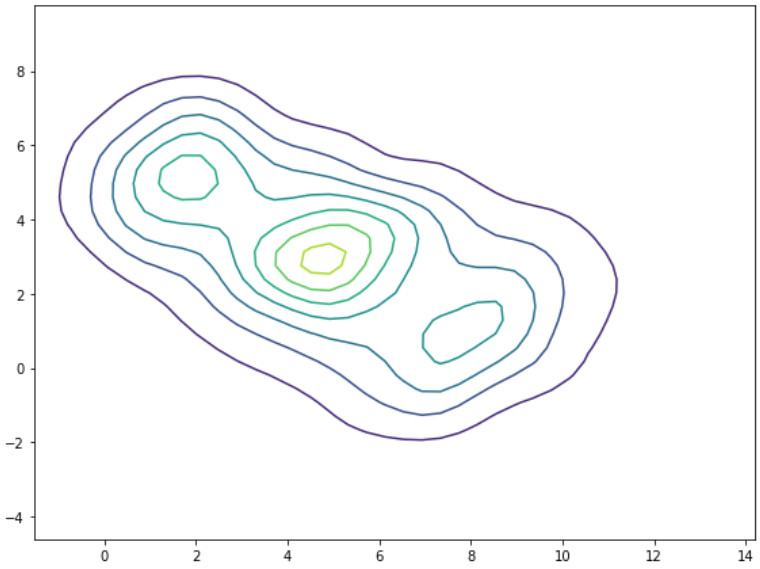
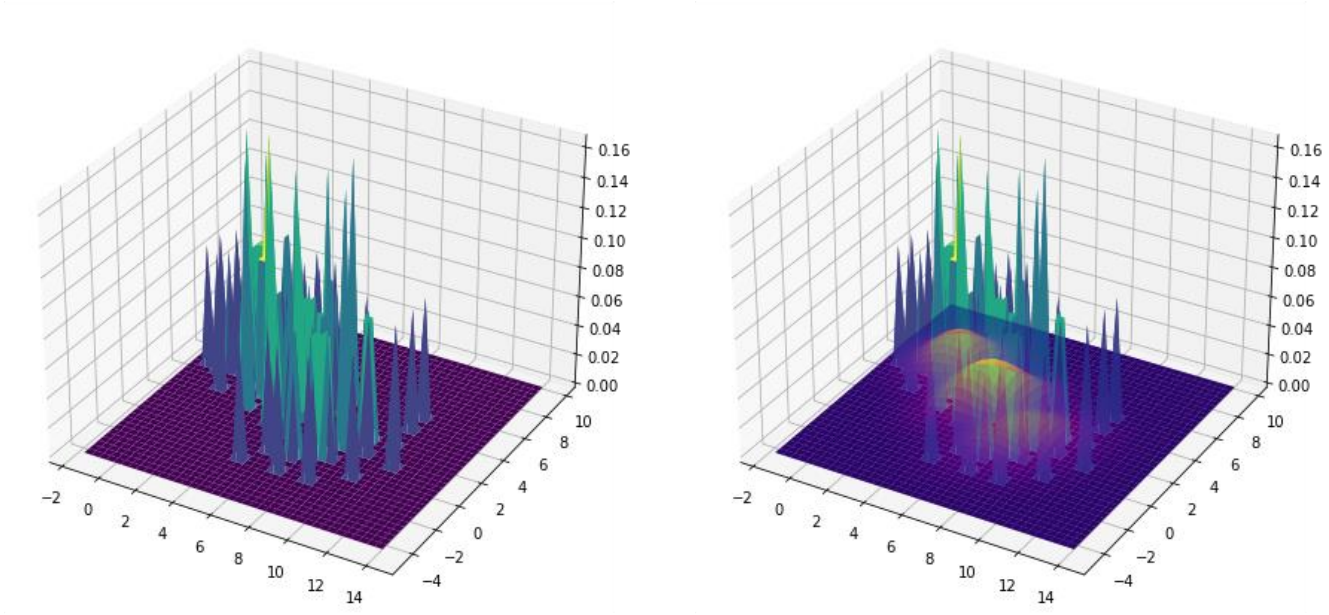


*Figure 45: KDE (Gaussian Kernel) 2D contour estimated PDF with h = 0.6 & σ = 0.9*

Overall, for KDE with gaussian kernel the following inputs had the best estimation: h = 0.6 & σ =0.9, h = 0.09 & σ =0.2, h = 0.3 & σ =0.6. The worst results achieved were for cases: h = 0.6 & σ =0.6, h = 0.6 & σ =0.2. it seems in cases that h > σ the results worsen.

### 2.2.5 KNN




*(a)* *(b)*

*Figure 46: KNN 3D PDF with k = 1 (a) estimated. (b) estimated vs. true*

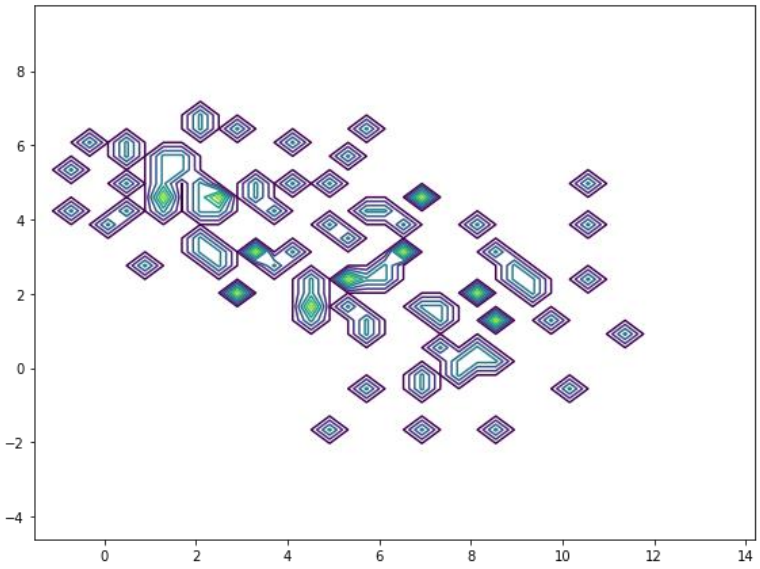
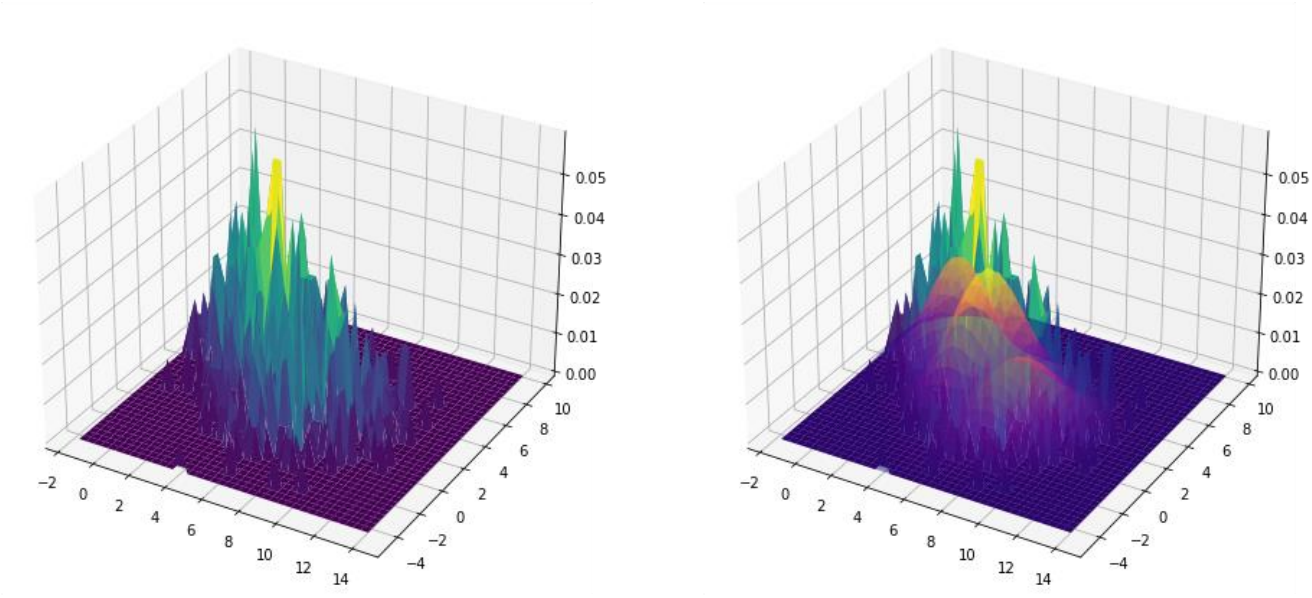


*Figure 47: KNN 2D contour estimated PDF k =1*




*(a)* *(b)*

*Figure 48: KNN 3D PDF with k = 9 (a) estimated. (b) estimated vs. true*

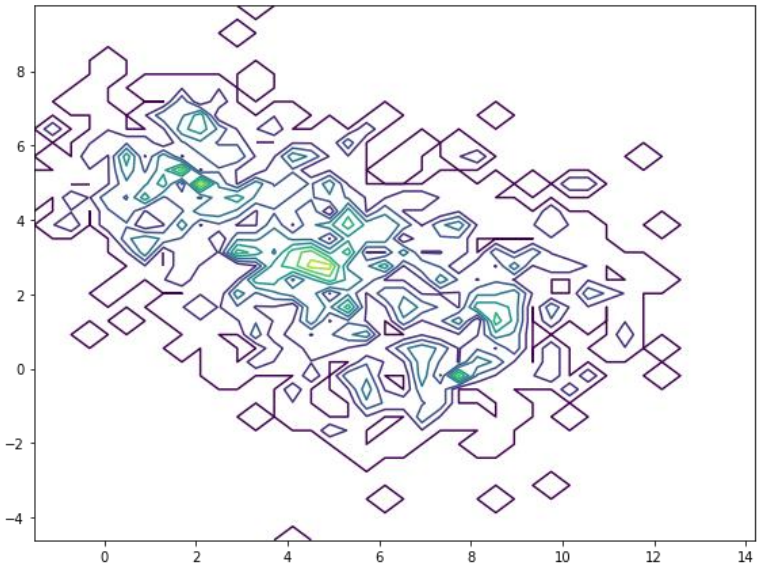
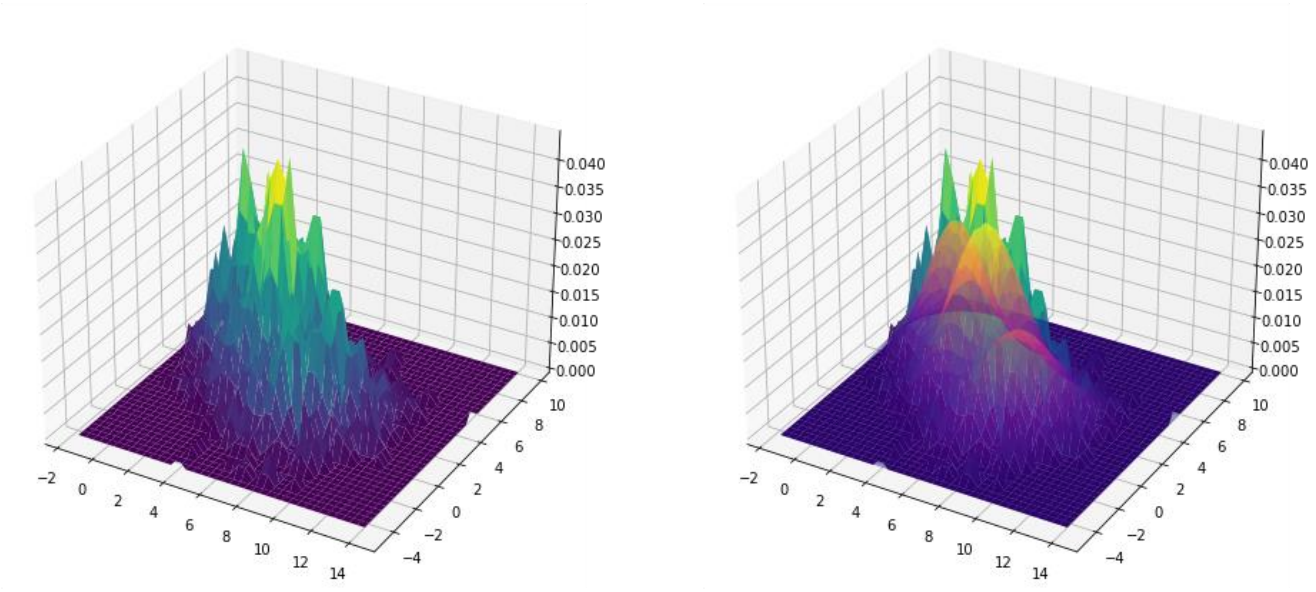



*Figure 49: KNN 2D contour estimated PDF k = 9*




*(a)* *(b)*

*Figure 50: KNN 3D PDF with k = 99 (a) estimated. (b) estimated vs. true*

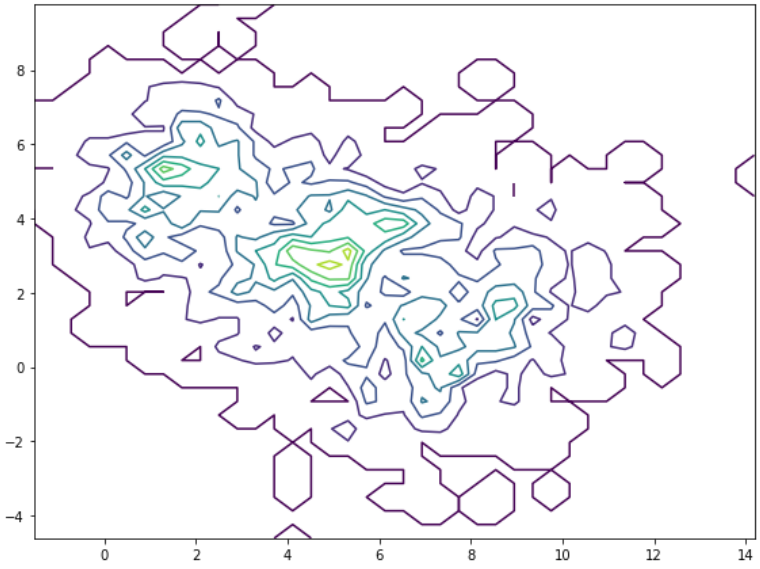


*Figure 51: KNN 2D contour estimated PDF k = 99*

Best result achieved with k = 99. Overall KNN performed poorly in compare to other methods like KDE. The estimation achieved by KNN as can been seen in 3d pdfs plots is so spiky.

### 2.2.6 KDE with K-Fold cross validation

The results for running KDE (with gaussian kernel) with $\sigma = 0.6$ using 5-Fold cross validation is shown in Table 9.

| h | Average error |
|---|---|
| 0.09 | 0.21261 |
| 0.3 | 0.07 |
| 0.6 | 0.05569 |

*Table 9: KDE with 5-Fold cross validation results*

### 2.2.7 Conclusion

It can be clearly said that the best results achieved by KDE with gaussian kernel.