Assignment 1

Xuanjie Fang

Fudan University

16307130335@fudan.edu.cn

Preliminaries

package requirements:

- matplotlib
- numpy

```
#install packages
$pip install matplotlib
$pip install numpy
```

Implementation

- ### histogram

  ```
  def histogram(num=500, bin=100):
      sampled_data = get_data(num)
      plt.hist(sampled_data, normed=True, bins=bin, edgecolor="black",alpha=0.7)
      plt.title('histogram')
      plt.xlabel('x')
      plt.ylabel('f(x)')
      plt.show()
  ```

  Considering the impact of bins on estimation, we can easily make an agreements that we should try to make it that **samples exits in every region** so we can obtain a continuous distributed Image. Hence , the value of bins should be less than the size of data set:

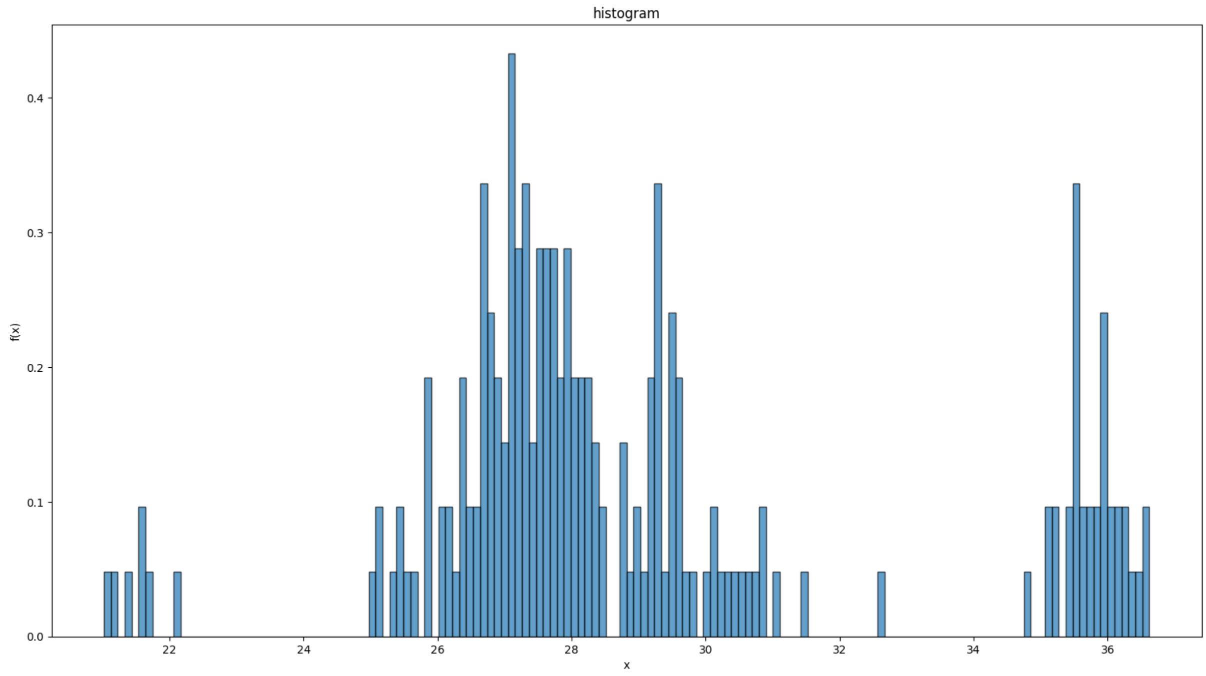
  For example, if we generate 200 samples, it's better to choice 100 as the value of bins rather than 400.

  Of course, the value of bins **should not be too small** since we want to get a more accurate answer.

  What I think a good section for bins is :

  $$50\% \times size(samples) < bins < 90\% \times size(samples)$$

  ```
  $python source.py -m hist -n 200 -b 150
  ```



- **kernel density estimation**

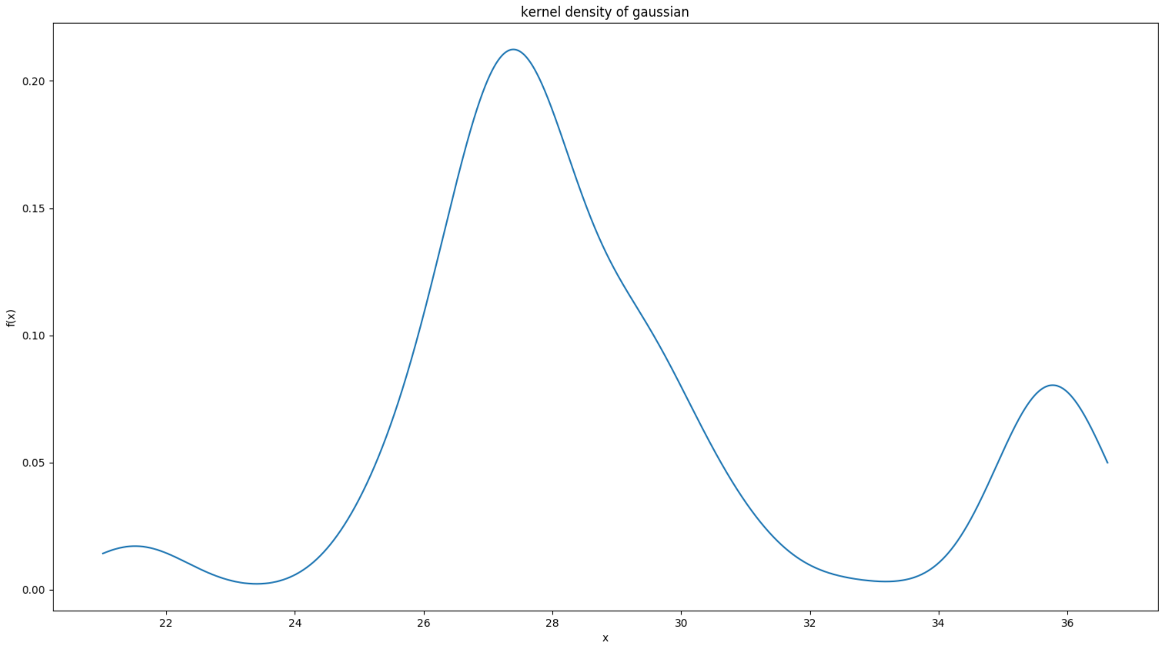
```
def gaussian(x):
    return (1/(sqrt(2*pi))) * exp(-0.5*x*x)

def kernel_density(k=gaussian, h=0.75, num=500):
    sampled_data, min_data, max_data, x_plot = handler_sample(num)
    fx = []
    for i in x_plot:
        px = 0
        for j in sampled_data:
            px += k((j-i)/h)
        px = px / (h*num)
        fx.append(px)
    display(x_plot, fx, "kernel density of gaussian")
```

Using **'mean intergrated squared error'** to determine the best **'h'**:

```
def best_h(sampled_data):
    std = np.std(sampled_data, ddof=1)
    h = 1.06*std*pow(len(sampled_data), -1/5)
    return h
```

I will not explain why this algorithm works here since it's a very common method , but it's clear that `h` should neither be too large nor be too small .

Here is the best estimate achieved with `num_data=100` when **h**=1.4:



- ## k nearest neighbor

```
def k_nearest_neighbor(k=10, num=500):
    sampled_data, min_data, max_data, x_plot = handler_sample(num)
    fx = []
    sum = 0
    for i in x_plot:
        dist = []
        for j in sampled_data:
            dist.append(abs(j-i))
        dist.sort()
        fx.append(k/(2*num*dist[k-1]))
    display(x_plot, fx, "k nearest neighbor")
```

Please show that the nearest neighbor method does not always yield a valid distribution:

$$p(x) = \frac{K}{NV}$$

Where **V** is the volume of a 1-dimensional hypersphere with radius **R**, where in this question **R** is the distance to from **x** to its **K**th nearest neighbor in the data set. Thus, in polar coordinates, if we consider sufficiently large values for the radial coordinate **r**, we have

$$p(x) \propto r^{-1}$$

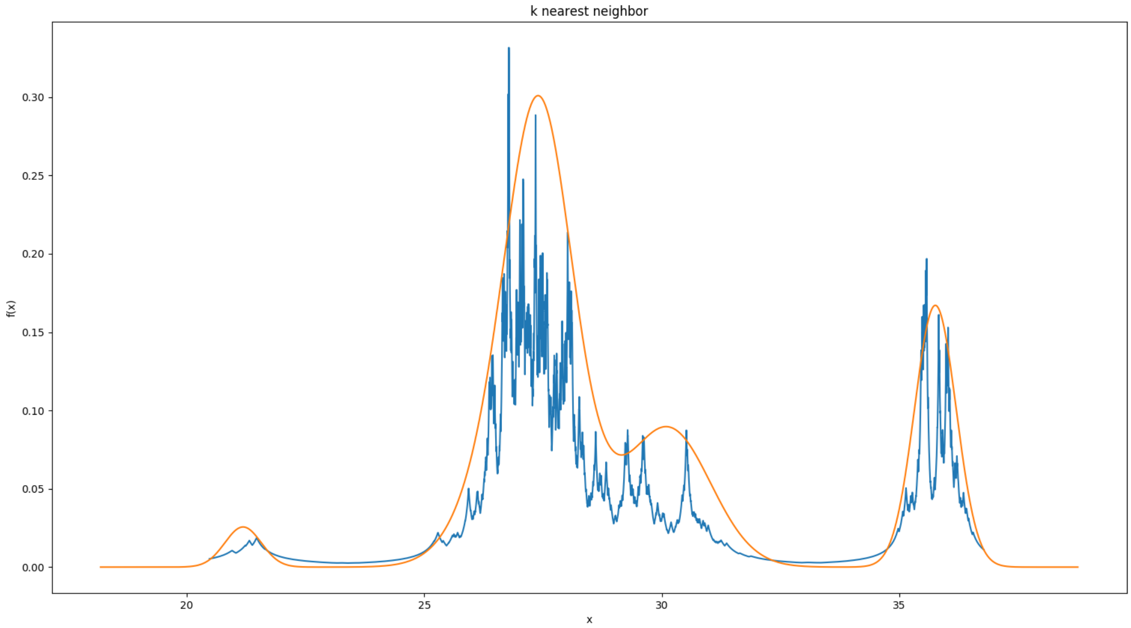
If we consider the intergral or **p(x)** and note that the volume element **dx** can be written as **dr**, we get:

$$\int p(x)dx \propto \int r^{-1}dr$$

Which diverges logarithmically.

And I think the image of 'knn' is very ugly: (data=200)

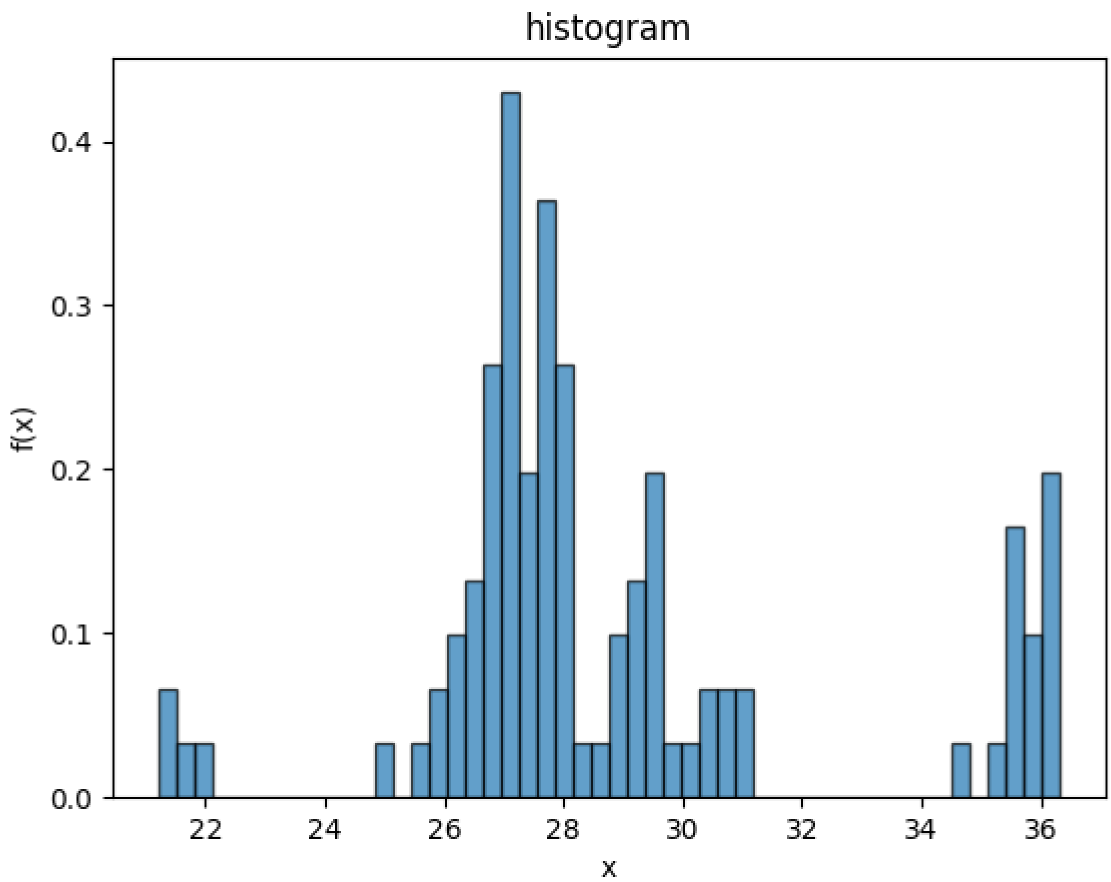
```
$python source.py -m knn -n 200
```



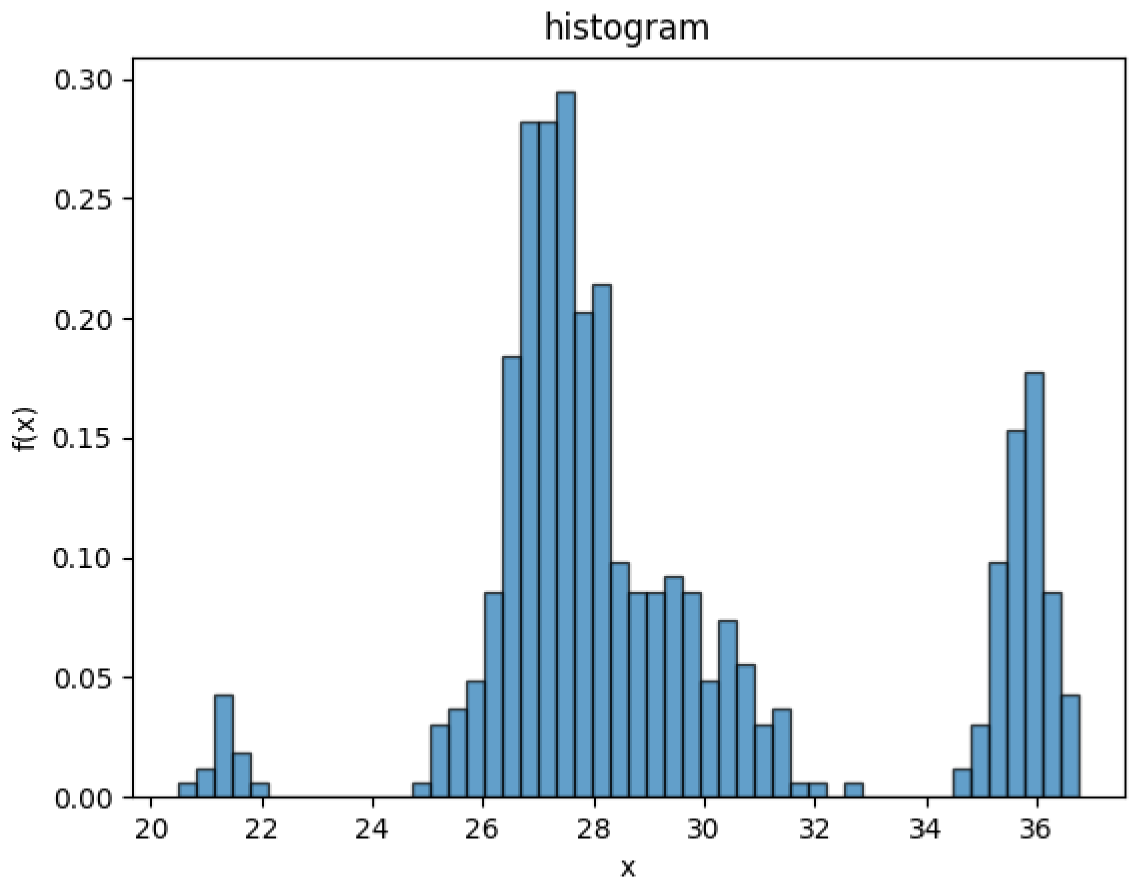
Using

```
def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("--methods", "-m", default="hist", choices=["hist", "kde",
"knn"])
    parser.add_argument("--number", "-n", default=100, type=int)
    parser.add_argument("--bins", "-b", default=50, type=int)
    parser.add_argument("--k_near", "-k", default=10, type=int)
    parser.add_argument("--bandwidth", '-d', default=0.75, type=float)
    args = parser.parse_args()
    if(args.methods == "hist"):
        histogram(args.number, args.bins)
    if(args.methods == "kde"):
        kernel_density(gaussian, args.bandwidth, args.number)
    if(args.methods == "knn"):
        k_nearest_neighbor(args.k_near, args.number)
```

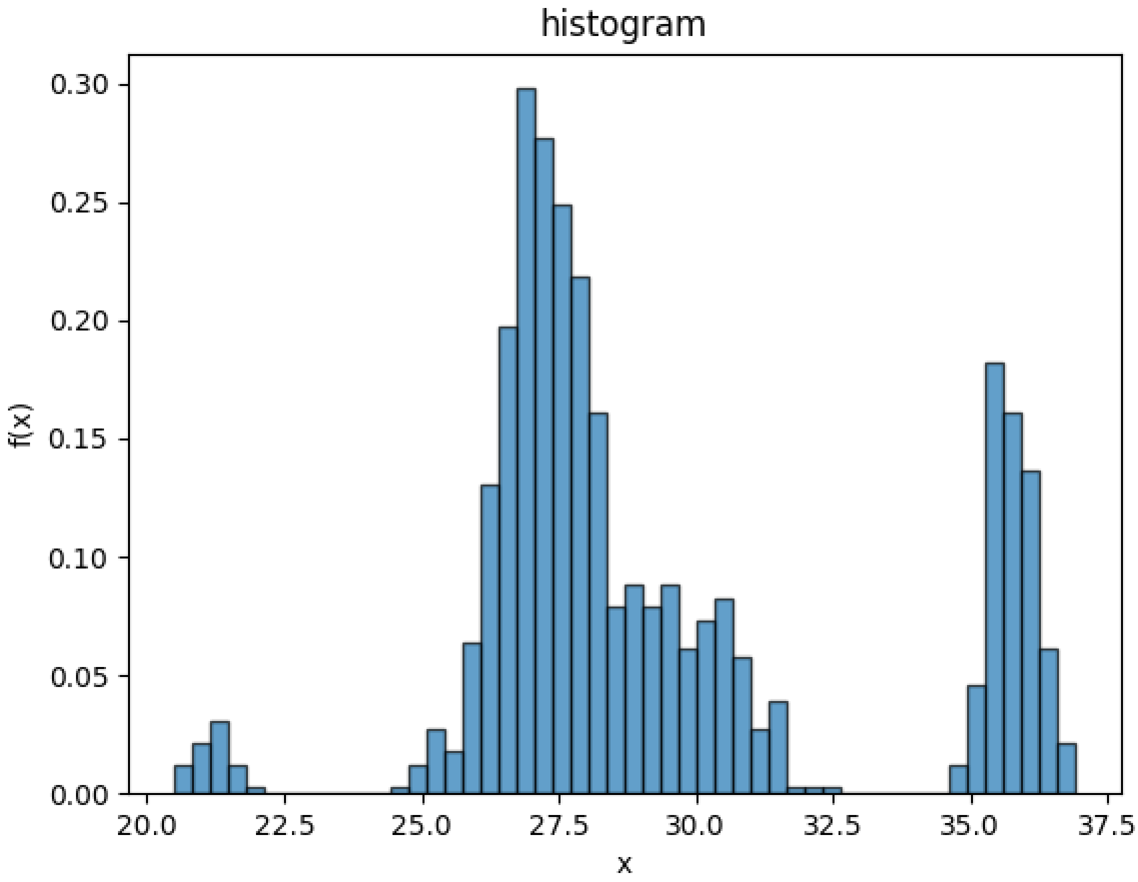
Result



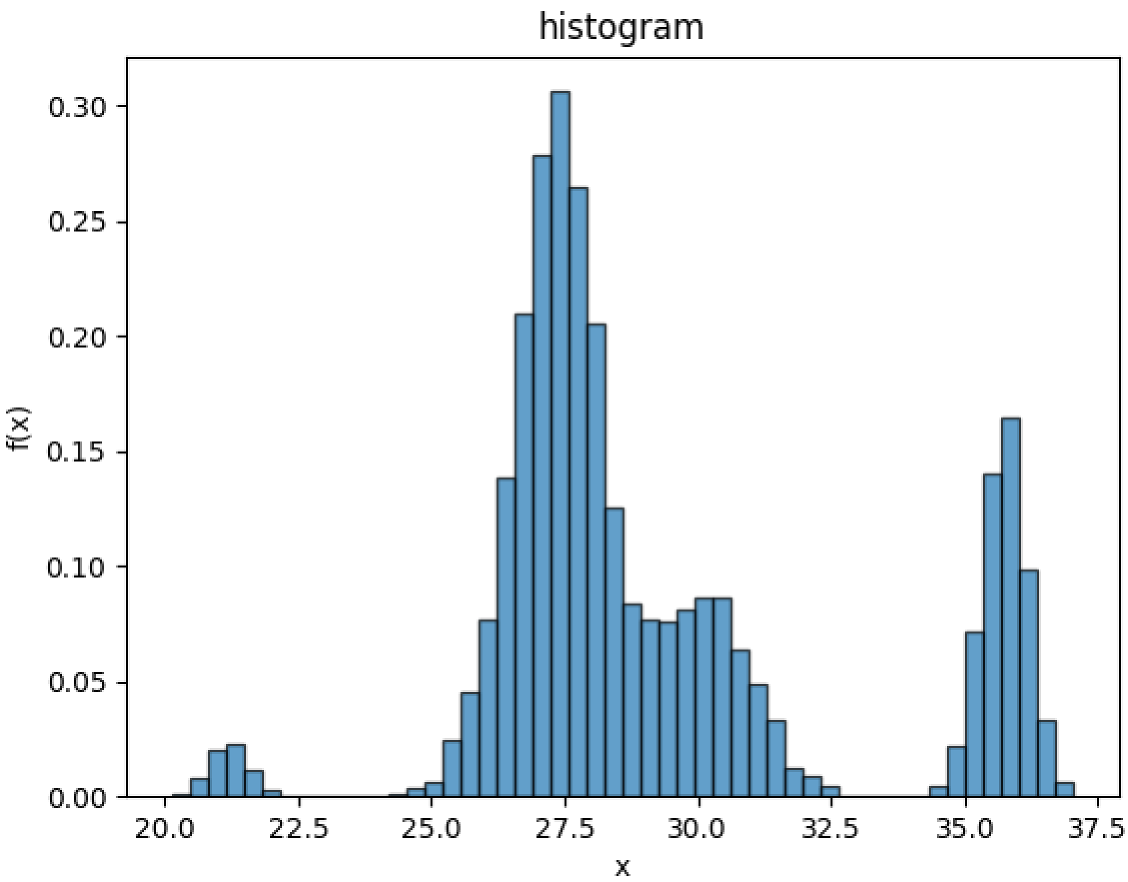
hist_100



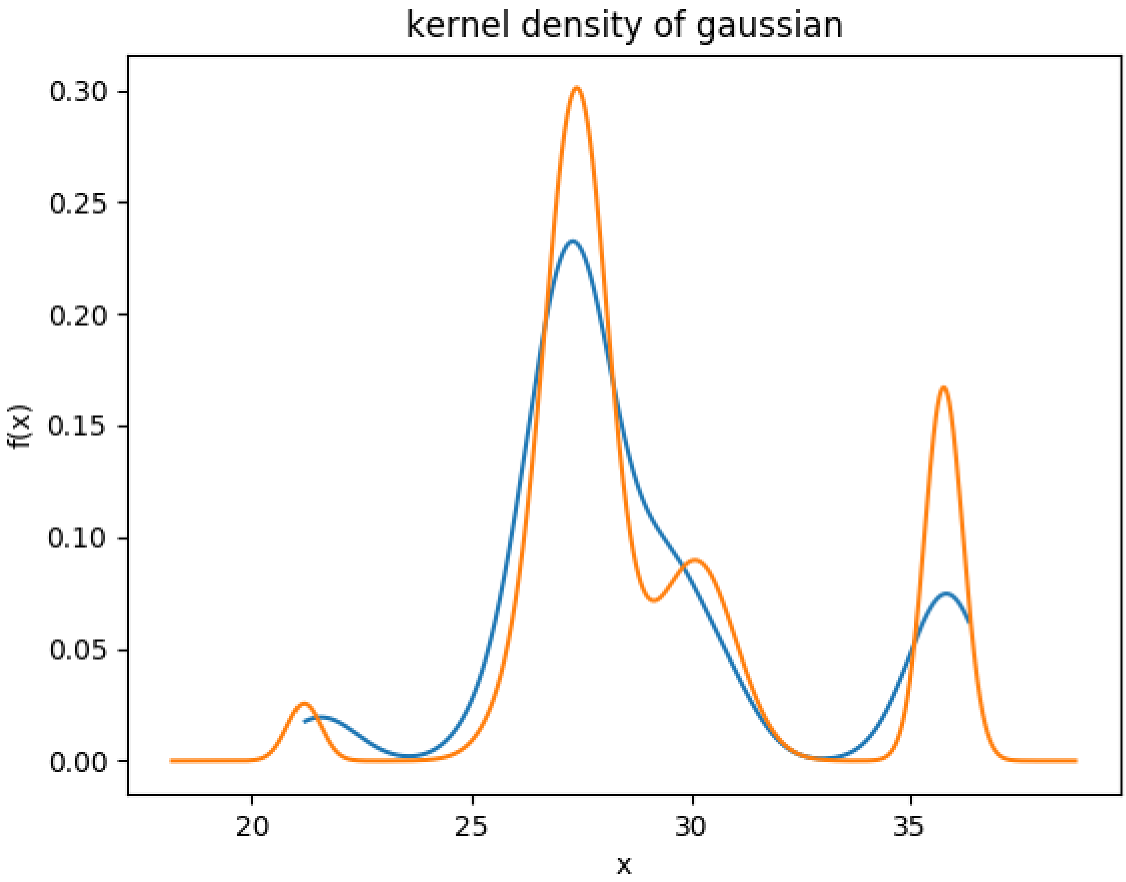
hist_500



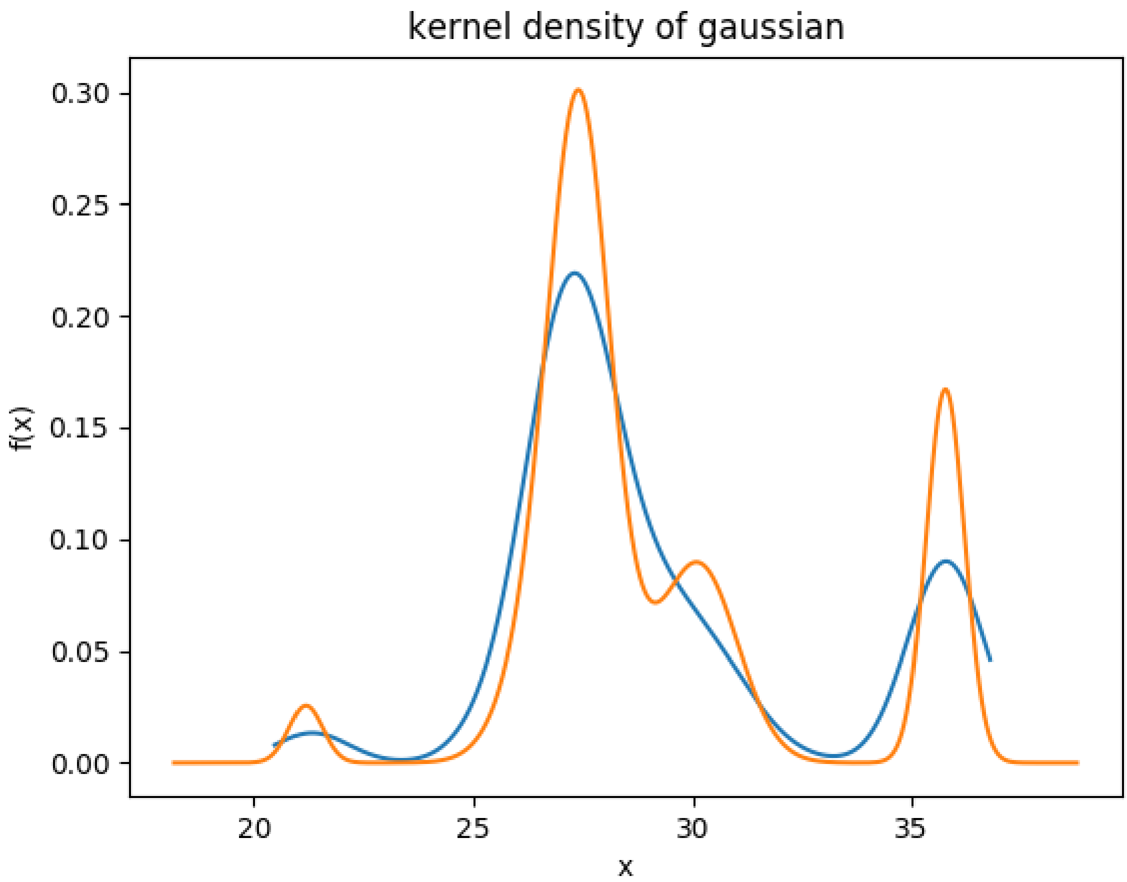
hist_1000



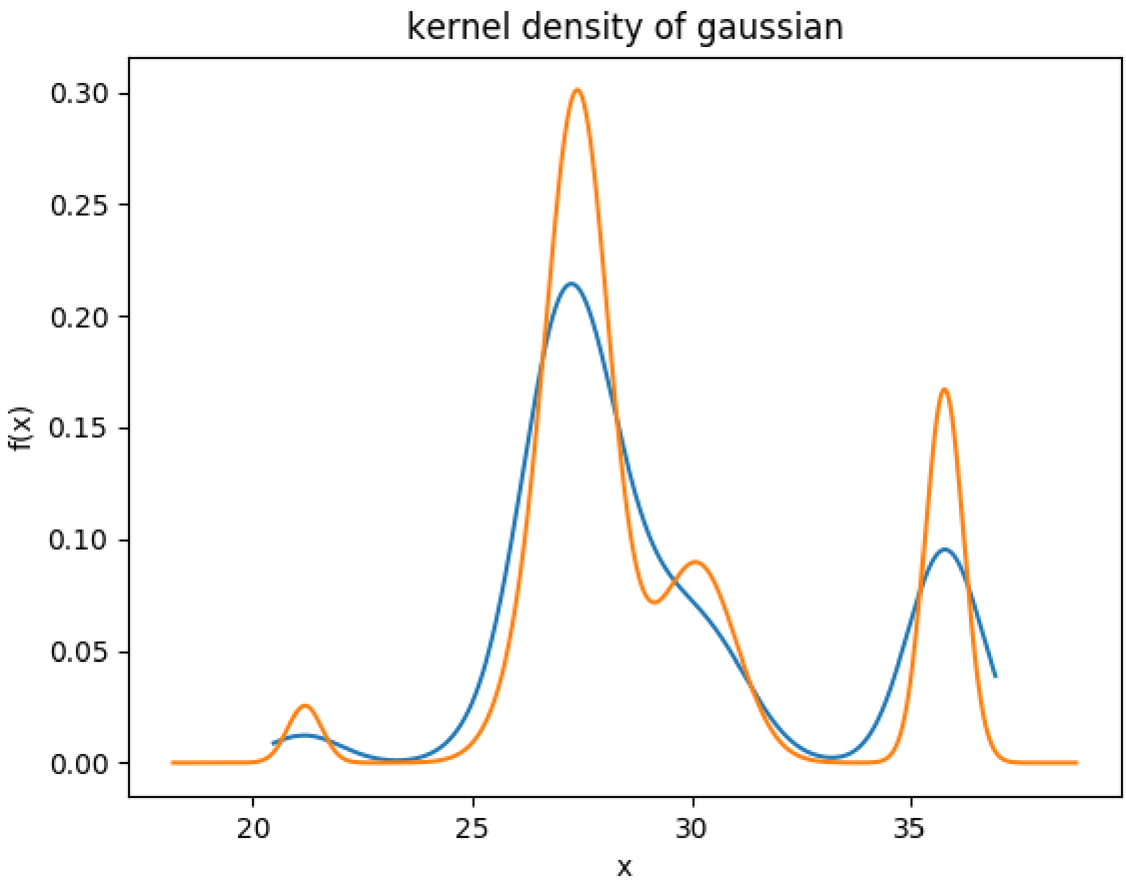
hist_10000



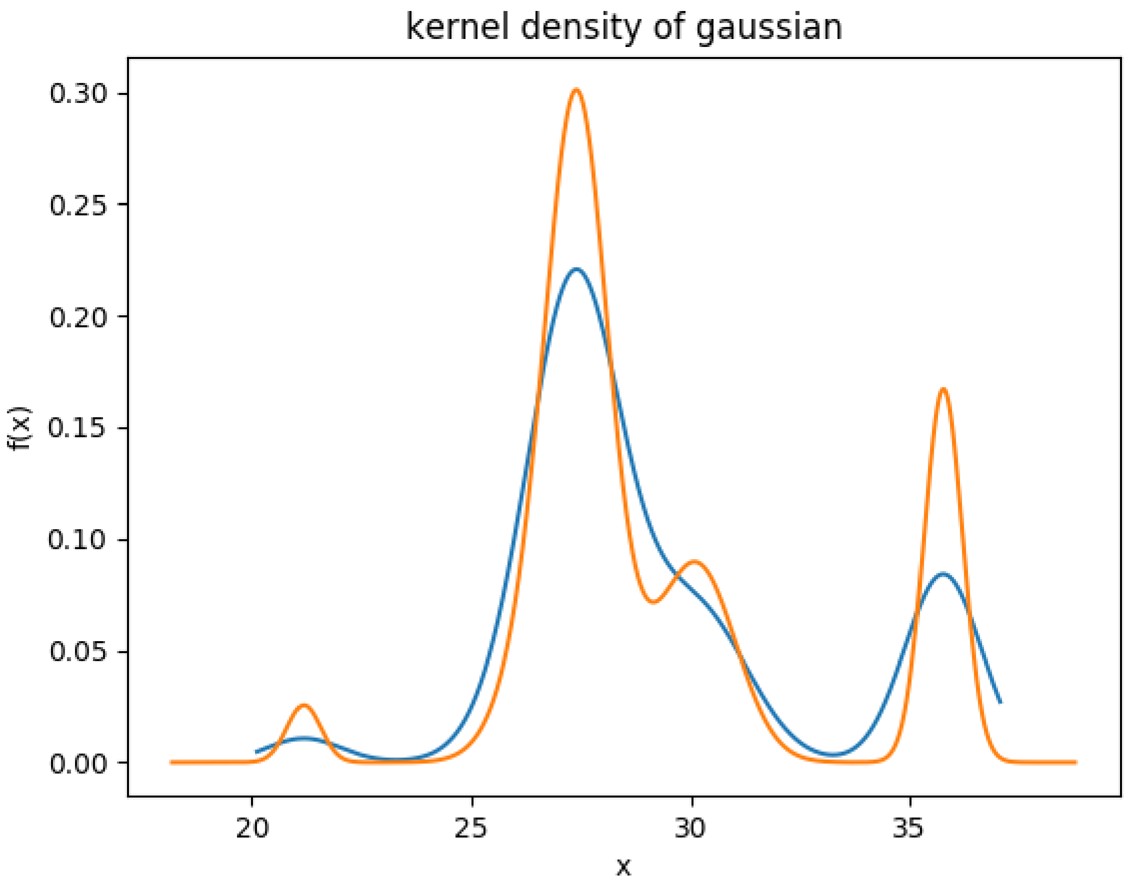
kde_100



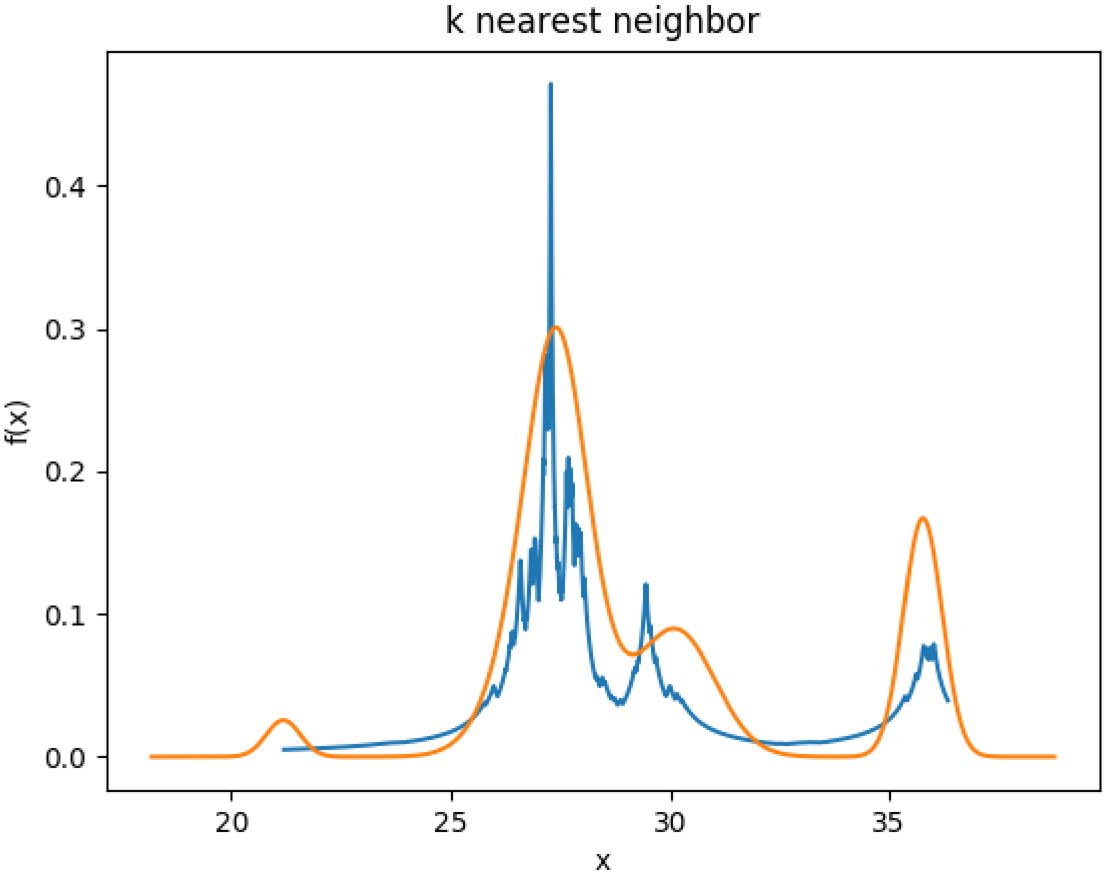
kde_500



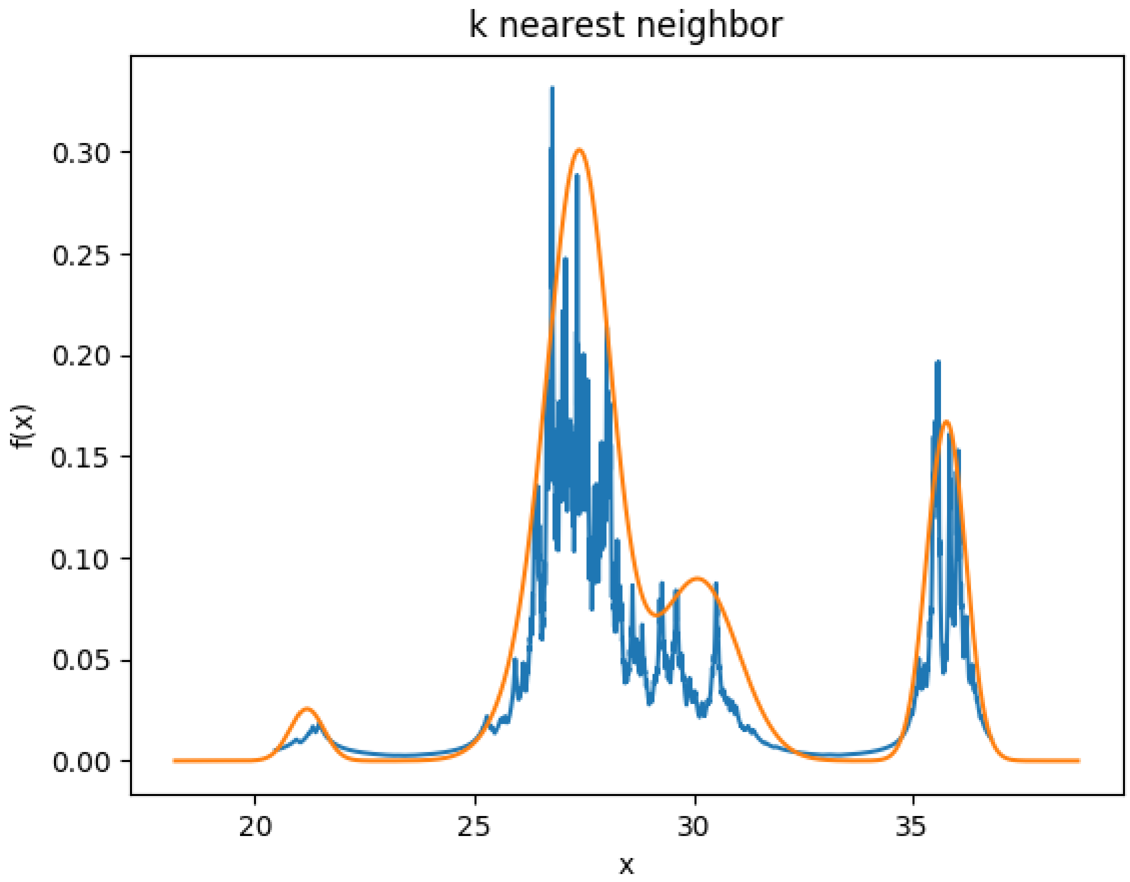
kde_1000



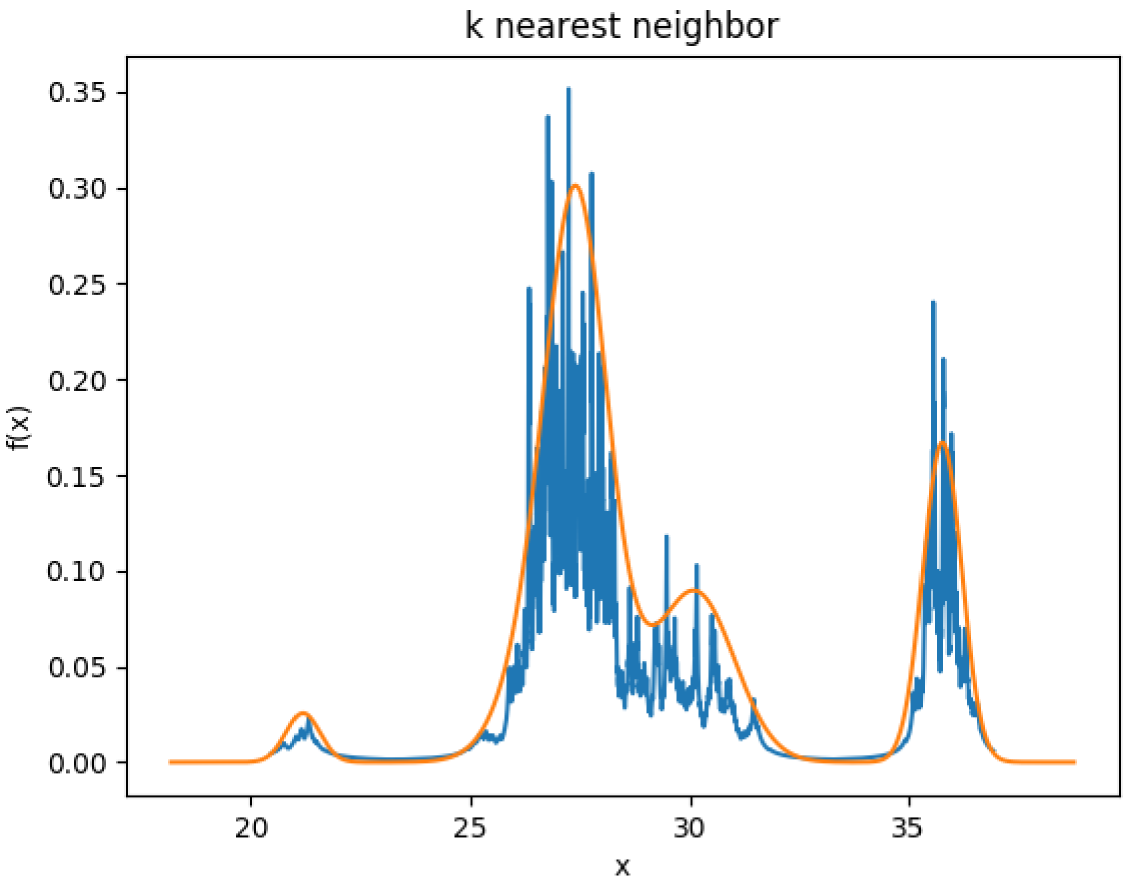
kde_10000



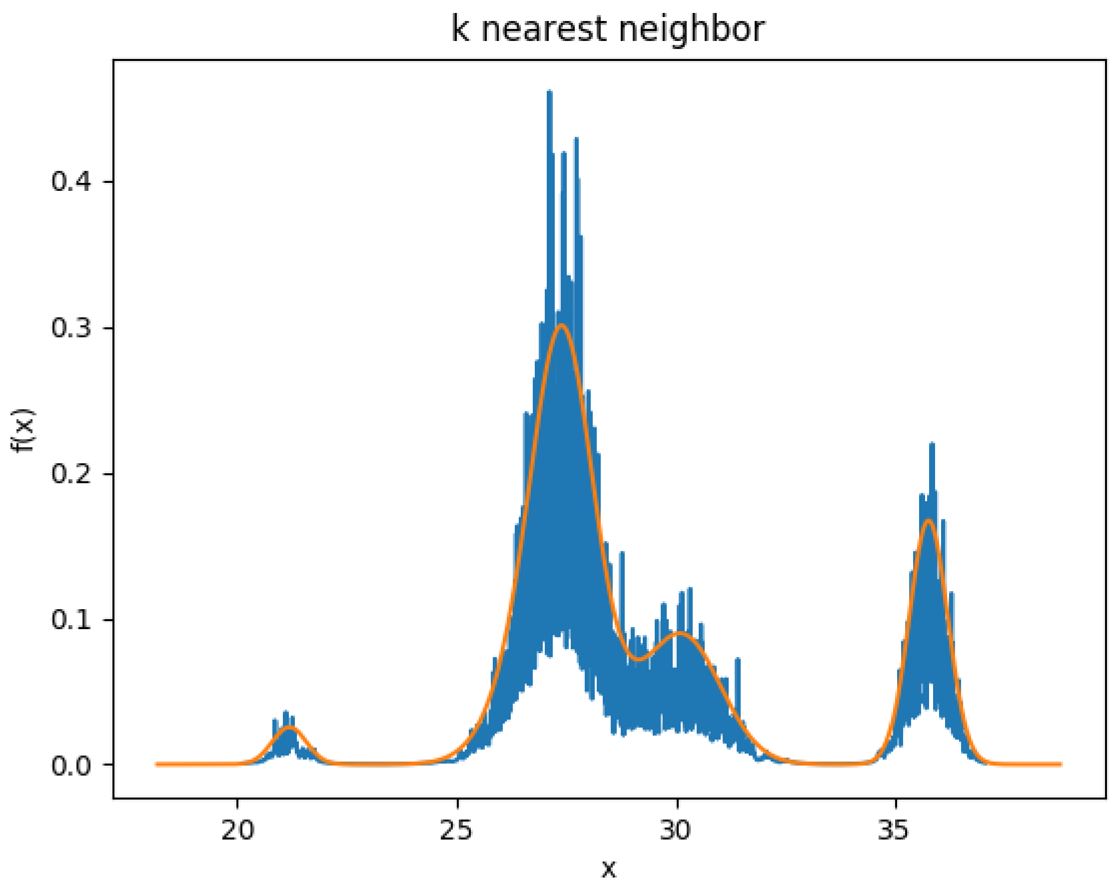
knn_100



knn_500



knn_1000



knn_10000