IFT 6390 Homework 1

Arlie Coles (20121051) and Yue (Violet) Guo (20120727)

September 28, 2018

Problem 1. *Small exercise on probabilities.*

Solution. First, we let:

- the probability of a woman having cancer be $P(C) = 0.015$,
- the probability of a test for cancer being positive, given that a woman has cancer, be $P(T|C) = 0.87$,
- the probability of a test for cancer being positive, given that a woman does not have cancer, be $P(T|\neg C) = 0.096$.

Then, to find the probability that a woman who has received a positive result actually has cancer, $P(C|T)$, we apply Bayes' Rule:

$$P(T|\neg C)P(\neg C) + P(T|C) + P(C) = (0.87 \times 0.015 + 0.096 \times (1 - 0.015)) = 0.10761 \quad (1)$$

$$\begin{aligned} P(C|T) &= \frac{P(T|C)P(C)}{P(T)} \\ &= \frac{P(T|C)P(C)}{P(T|\neg C)P(\neg C) + P(T|C) + P(C)} \\ &= \frac{(0.87)(0.015)}{(0.87 \times 0.015 + 0.096 \times (1 - 0.015))} \\ &= \frac{0.01305}{0.10761} \\ &= 0.12127 = 12.13\% \end{aligned}$$

Therefore, the doctors surveyed ought to have responded with *E) between 10% and 30%*.

Problem 2. *Curse of dimensionality and geometric intuition in higher dimensions.*

Solution.

1. Volume can be generalized in d dimensions as $V = c^d$.

2. Since the probability density is zero anywhere outside the cube, we know that the integral of the probability density function over the volume must be equal to 1:

$$1 = \int_V p(x)dx$$

   And since the probability distribution is uniform, we can move it outside of the integral as it is a constant:

$$\begin{aligned} 1 &= p(x) \int_V dx \\ &= p(x)V \\ &= p(x)c^d \\ \Rightarrow p(x) &= \frac{1}{c^d}. \end{aligned}$$

3. We can define the volume of the interior cube as $(0.94c)^d$. Since the volume of the exterior cube is c^d, we can define the volume of the outer shell as $c^d - (0.94c)^d = c^d(1 - 0.94^d)$.

   Then, the probability of a generated point x falling within the shell is:

$$\begin{aligned} P(\text{shell}) &= \int_V p(x)dx \\ &= \frac{1}{c^d} \int_V dx \\ &= \frac{1}{c^d} c^d(1 - 0.94^d) \\ &= 1 - 0.94^d. \end{aligned}$$

   The probability of a generated point x falling in the smaller interior hypercube is:

$$
\begin{aligned}
P(\text{interior}) &= \int_V p(x)dx \\
&= \frac{1}{c^d}\int_V dx \\
&= \frac{1}{c^d}(0.94^d)(c^d) \\
&= 0.94^d.
\end{aligned}
$$

4. Changing values of d:

$$
\begin{array}{lll}
d = 1: & 1 - 0.94^1 & = 0.06 \\
d = 2: & 1 - 0.94^2 & = 0.1164 \\
d = 3: & 1 - 0.94^3 & = 0.1694 \\
d = 5: & 1 - 0.94^5 & = 0.2661 \\
d = 10: & 1 - 0.94^{10} & = 0.4614 \\
d = 100: & 1 - 0.94^{100} & = 0.9979 \\
d = 1000: & 1 - 0.94^{1000} & \approx 1
\end{array}
$$

5. In higher dimensions, it seems that the distribution of points is much more concentrated around the "edges" of the given "space", which runs counter to intuition in smaller dimensions about uniform distributions, where we suppose that there is not a relationship between location in the space and the value given by the probability density function.

Problem 3. *Parametric Gaussian density estimation vs. Parzen window density estimation*

Solution.

1. (a) An isotropic Gaussian has two parameters:
   - μ, the mean, of dimension d, and
   - σ^2, the variance, of dimension 1. (Note that the covariance matrix Σ is of dimension $d \times d$, where the diagonal terms are all equal to one σ^2 value.)

   (b) For calculating the mean:

$$\mu = \frac{1}{n} \sum_{i=1}^{n} x_i$$

   For calculating the variance:

$$\Sigma = \frac{1}{n} \sum_{i=1}^{n} (x_i - \mu)(x_i - \mu)^T$$

   Then, we arrive at σ^2 by decomposing the resulting matrix (which is all zeros excepting the σ^2's on the diagonal, since the Gaussian is isotropic) into the following form, where the value of σ^2 is apparent:

$$\Sigma = \sigma^2 I$$

   (c) We can describe the complexity-affecting parts of the μ-calculating algorithm like so:

```
for i = 1 to n:                                          \\ O(n)
  add x_i to the running total, which has d components   \\ O(d)
```

   This is $O(nd)$.

   We can describe the complexity-affecting parts of the σ-calculating algorithm like so:

```
for i = 1 to n:                                          \\ O(n)
  subtract mu from x_i, both with d components           \\ O(d)
   repeat the previous operation                         \\ O(d)
   take its transpose                                    \\ O(1)
   multiply the resulting dx1 and 1xd vectors to get a dxd matrix
       \\ O(d^2)
```

This is $O(n(d+d+1+d^2)) = O(n(d+d^2)) \approx O(nd^2)$.

The complexity of calculating both parameters is then $O(nd + nd + nd^2) = O(nd + nd^2) = O(n(d+d^2)) \approx O(nd^2)$.

(d) The probability density function is:

$$\begin{aligned}\hat{p}_{gauss-isotrop}(x) &= \mathcal{N}_{\mu,\sigma^2} \\ &= \frac{1}{(2\pi)^{d/2}\sigma^d} \exp\left(\frac{-1}{2}\frac{||x-\mu||^2}{\sigma^2}\right)\end{aligned}$$

(e) Calculating a prediction from $p(x)$ at a given x only uses one data point, so it is not dependent on n. Subtracting μ from x in the exponent, however, is dependent on d since each has d components. Therefore the complexity is $O(d)$.

2. (a) The "training/learning" phase for the Parzen method consists of loading each training point. We center each kernel Gaussian on each training point, which becomes the μ of the Gaussian, whose σ is predefined.

(b) The probability density function is:

$$\hat{p}_{Parzen}(x) = \sum_{i=1}^{n} \frac{1}{(2\pi)^{d/2}\sigma^d} \exp\left(\frac{-1}{2}\frac{||x-\mu||^2}{\sigma^2}\right)$$

(c) The complexity for calculating a prediction is $O(nd)$, since we must perform the $x - \mu$ subtraction, which deals with d components, n times.

```
for each x_i where i = 1 to n:                    \\ O(n)
  calculate x_i - mu, which has d components       \\ O(d)
   square operation x_i - mu, which has d components    \\ O(d)
```

3. (a) The Parzen approach has a higher capacity/expressivity. Taking the sum over a set of kernels, where the kernels can be close together or far apart depending on the distribution of the data, allows for a highly nuanced density curve to emerge, while the parametric Gaussian approach is restricted to a density curve delimited in shape by the definition of a single Gaussian.

(b) The Parzen approach is also more likely to overfit and memorize noise, for the same reason. Its direct incorporation of the location of each datapoint as summands to the final density curve means that noise will contribute just as well to

the final density curve as the good data, while the parametric Gaussian approach generalizes this effect somewhat by using averages over the whole dataset and conforming to a less-nuanced shape.

(c) *Parameters* are learned by the algorithm from training data, whereas *hyperparameters* are assigned by the user, and validated on training and validation data. For example, in the parametric Gaussian density function questions below, we can optimize our μ parameter by finding the log error and taking derivatives, and this feedback process and learning of the parameter is the use case for the Gaussian modelling framework. By contrast, this Parzen framework involves the user observing the dataset and deciding what is the best σ value by "tuning" its value until a satisfactory result emerges. These types of algorithms must have users to set hyperparameters, because there is no efficient way of optimizing the function iteratively (e.g., in this case, the function is not convex).

4. Diagonal Gaussians

(a) To derive the equation of a diagonal Gaussian density with dimension d, we can start with the general Gaussian density:

$$p(x) = \frac{1}{(2\pi)^{d/2}\sqrt{|\Sigma|}} \exp\left(\frac{-1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right)$$

where

$$\Sigma = \begin{bmatrix} \sigma_{11}^2 & 0 & \dots & 0 \\ 0 & \sigma_{ii}^2 & 0 & 0 \\ \vdots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & a_{dd}^2 \end{bmatrix}$$

and where

$$\Sigma^{-1} = \begin{bmatrix} \frac{1}{\sigma_{11}^2} & 0 & \dots & 0 \\ 0 & \frac{1}{\sigma_{ii}^2} & 0 & 0 \\ \vdots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & \frac{1}{\sigma_{dd}^2} \end{bmatrix}$$

Then, since the only filled positions in the Σ matrix are the diagonals, we can manually represent the matrix multiplication while skipping the calculations in the

matrix multiplication that do not interact with that diagonal:

$$p(x) = \frac{1}{(2\pi)^{d/2}\sqrt{|\Sigma|}} \exp\left(\frac{-1}{2}\sum_{i=1}^{d}\frac{1}{\sigma_{ii}^2}(x_i - \mu_i)^2\right)$$

The above makes use of the fact that a transpose of a matrix symmetric about the diagonal is itself. Then, if we like, we can further represent the constant normalization term in product notation, using the fact that the determinant of a diagonal matrix is equal to the product of the diagonal terms:

$$\begin{aligned} p(x) &= \frac{1}{(2\pi)^{d/2}\prod_{i=1}^{d}\sigma_{ii}^2} \exp\left(\frac{-1}{2}\sum_{i=1}^{d}\frac{1}{\sigma_{ii}^2}(x_i - \mu_i)^2\right) \\ &= (\prod_{i=1}^{d}\frac{1}{(2\pi)^{1/2}\sigma_{ii}}) \exp\left(\frac{-1}{2}\sum_{i=1}^{d}\frac{1}{\sigma_{ii}^2}(x_i - \mu_i)^2\right) \\ &= (\prod_{i=1}^{d}\frac{1}{(2\pi\sigma_{ii}^2)^{1/2}}) \exp\left(\frac{-1}{2}\sum_{i=1}^{d}\frac{1}{\sigma_{ii}^2}(x_i - \mu_i)^2\right) \end{aligned}$$

The parameters of the diagonal Gaussian are:

- μ, the mean, of dimension d, and
- Σ, the covariance matrix, of dimension $d \times d$.

(b) *Proof.* Show that the components of a random vector following a diagonal Gaussian distribution are independent random variables.

First, let us walk through the idea. We would like to prove the independence of the components of a diagonal Gaussian by proving that the product of each normal distribution's density function made up of each component of the random vector is equal to the diagonal Gaussian distribution's density function, namely the random vector's density distribution.

The Gaussian density function for dataset D is defined as:

$$\begin{aligned} p(\boldsymbol{x}) &= \mathcal{N}_{\mu,\Sigma}(\boldsymbol{x}) \\ &= \frac{1}{(2\pi)^{d/2}\sqrt{det(\Sigma)}} \exp\left(-\frac{1}{2}(\boldsymbol{x} - \mu)^T\Sigma^{-1}(\boldsymbol{x} - \mu)\right) \end{aligned}$$

Since we have a diagonal Σ, it can be written in the form of a summation of the diagonal terms in the power, and then transformed into a product of exponents. Note that σ_{ii} here refers to the diagonal terms:

$$p(\boldsymbol{x}) = \frac{1}{(2\pi)^{d/2}\sqrt{det(\Sigma)}} \exp\left(\sum_{i=1}^{d} -\frac{1}{2\sigma_{ii}^2}(x_i - \mu)^2\right)$$
$$= \frac{1}{(2\pi)^{d/2}\sqrt{det(\Sigma)}} \prod_{i=1}^{d} \exp - \left(\frac{1}{2\sigma_{ii}^2}(x_i - \mu)^2\right)$$

As above, we can also express the constant term in a product form, giving us the expression for $p(\boldsymbol{x})$:

$$p(\boldsymbol{x}) = \prod_{i=1}^{d} \frac{1}{(2\pi\sigma_{ii}^2)^{1/2}} \prod_{i=1}^{d} \exp\left(-\frac{1}{2\sigma_{ii}^2}(x_i - \mu)^2\right)$$
$$= \prod_{i=1}^{d} \frac{1}{(2\pi\sigma_{ii}^2)^{1/2}} \exp\left(-\frac{1}{2\sigma_{ii}^2}(x_i - \mu)^2\right)$$

Note that $\frac{1}{(2\pi\sigma_{ii}^2)^{1/2}} \exp\left(\frac{1}{2\sigma_{ii}^2}(x_i - \mu)^2\right)$ is a Gaussian with parameters $\mathcal{N}(\mu, \sigma_{ii})$. Thus, we have:

$$p(\boldsymbol{x}) = \prod_{i=1}^{d} \mathcal{N}(\mu, \sigma_{ii})$$

Therefore, the diagonal Gaussian density function is equal to the product of the individual components' density functions, and the individual components are independent. □

(c) Empirical risk can be written as:

$$\hat{R}(f, D) = \frac{1}{|D|} \sum_{(x,y)\in D} L(f(x), y)$$
$$= \frac{1}{|D|} \sum_{(x,y)\in D} -\log(p(x))$$

Substituting from the previous part, we have:

$$\begin{aligned}
\hat{R}(f, D) &= \frac{1}{|D|} \sum_{(x,y)\in D} -\log\left(\prod_{i=1}^{d} \frac{1}{(2\pi\sigma_{ii}^2)^{1/2}} \exp\left(-\frac{1}{2\sigma_{ii}^2}(x_i-\mu)^2\right)\right) \\
&= \frac{1}{|D|} \sum_{(x,y)\in D} \sum_{i=1}^{d} -\left(\log\left(\frac{1}{(2\pi\sigma_{ii}^2)^{1/2}} \exp\left(-\frac{1}{2\sigma_{ii}^2}(x_i-\mu)^2\right)\right)\right) \\
&= \frac{1}{|D|} \sum_{(x,y)\in D} \sum_{i=1}^{d} -\left(\log\left(\frac{1}{(2\pi\sigma_{ii}^2)^{1/2}}\right) + \log\left(\exp\left(-\frac{1}{2\sigma_{ii}^2}(x_i-\mu)^2\right)\right)\right) \\
&= \frac{1}{|D|} \sum_{(x,y)\in D} \sum_{i=1}^{d} -\log\left(\frac{1}{(2\pi\sigma_{ii}^2)^{1/2}}\right) - \log\left(\exp\left(-\frac{1}{2\sigma_{ii}^2}(x_i-\mu)^2\right)\right) \\
&= \frac{1}{|D|} \sum_{(x,y)\in D} \sum_{i=1}^{d} -\log\left((2\pi\sigma_{ii}^2)^{-1/2}\right) - \left(-\left(\frac{1}{2\sigma_{ii}^2}(x_i-\mu)^2\right)\right) \\
&= \frac{1}{|D|} \sum_{(x,y)\in D} \sum_{i=1}^{d} \frac{1}{2}\log\left(2\pi\sigma_{ii}^2\right) + \frac{1}{2\sigma_{ii}^2}(x_i-\mu)^2 \\
&= \frac{1}{2|D|} \sum_{(x,y)\in D} \sum_{i=1}^{d} \log\left(2\pi\sigma_{ii}^2\right) + \frac{1}{\sigma_{ii}^2}(x_i-\mu)^2
\end{aligned}$$

(d) To find the optimal μ value, we take the derivative with respect to μ:

$$\begin{aligned}
\frac{\partial}{\partial\mu_i} &= \frac{1}{2|D|} \sum_{(x,y)\in D} \sum_{i=1}^{d} \frac{1}{\sigma_{ii}^2} 2(-x_i^{(j)} + \mu_i) \\
&= \frac{1}{|D|} \sum_{(x,y)\in D} \sum_{i=1}^{d} \frac{1}{\sigma_{ii}^2}(-x_i^{(j)} + \mu_i)
\end{aligned}$$

Setting $\frac{\partial}{\partial\mu_{ki}} = 0$, we have:

$$\begin{aligned}
\frac{1}{|D|} \sum_{(x,y)\in D} \sum_{i=1}^{d} \frac{1}{\sigma_{ii}^2}(-x_i^{(j)} + \mu_i) &= 0 \\
\sum_{(x,y)\in D} \sum_{i=1}^{d} \frac{1}{\sigma_{ii}^2} x_i &= \sum_{(x,y)\in D} \sum_{i=1}^{d} \frac{1}{\sigma_{ii}^2}\mu_i \\
\mu_i &= x_i^{(j)}
\end{aligned}$$

Therefore, the optimal μ value is $x_i^{(j)}$.

Now, to find the optimal σ^2 value, we take the derivative with respect to σ_{ii}^2:

$$\begin{aligned}\frac{\partial}{\partial \sigma_{ii}^2} &= \frac{1}{2|D|} \sum_{(x,y)\in D} \sum_{i=1}^{d} \frac{1}{2\pi\sigma_{ii}^2} 2\pi - \frac{1}{\sigma_{ii}^4}(x_i^{(j)} + \mu_i) \\ &= \frac{1}{2|D|} \sum_{(x,y)\in D} \sum_{i=1}^{d} \frac{1}{\sigma_{ii}^2} - \frac{(x_i^{(j)} - \mu_i)}{\sigma_{ii}^4}\end{aligned}$$

Setting $\frac{\partial}{\partial \sigma_{ii}^2}$ to zero, we have:

$$\begin{aligned}\frac{1}{2|D|} \sum_{(x,y)\in D} \sum_{i=1}^{d} \frac{1}{\sigma_{ii}^2} - \frac{(x_i^{(j)} - \mu_i)}{\sigma_{ii}^4} &= 0 \\ \frac{1}{2|D|} \sum_{(x,y)\in D} \sum_{i=1}^{d} \frac{1}{\sigma_{ii}^2} &= \frac{1}{2|D|} \sum_{(x,y)\in D} \sum_{i=1}^{d} \frac{(x_i^{(j)} - \mu_i)^2}{\sigma_{ii}^4} \\ \frac{1}{\sigma_{ii}^2} &= \frac{(x_i^{(j)} - \mu_i)^2}{\sigma_{ii}^4} \\ \sigma_{ii}^4 &= \sigma_{ii}^2 (x_i^{(j)} - \mu_i)^2 \\ \sigma_{ii}^2 &= (x_i^{(j)} - \mu_i)^2\end{aligned}$$

Therefore, the optimal σ_{ii}^2 value is $(x_i - \mu_i)^2$.

Problem 4. *Practical part: density estimation*

Solution.

1. Note to TAs: We have attached the Jupyter Notebook PDF export in this file. All the answers and plots are included. Some parts of our code blocks get cut off due to the restricted margin. We have consulted the professor, and he has stated that is not a problem. The complete code can be found in our Jupyter Notebook '4. Practical Part - Density Estimation.ipynb'

4. Practical Part - Density Estimation

September 28, 2018

1 4. Practical part: Density estimation

IFT 6390 Fall/Automne 2018, HW1, Q4, practical

Arlie Coles (20121051) & Yue(Violet) Guo (20120727)

Note to TAs: these implementations run on Python 3.

1.1 1. Implementation of a Gaussian parametric density estimator

In [18]:
```
import numpy as np
import math

class gaussianDiag:
    def __init__(self, data):
        self.train_data = data
        self.train_mean = 0
        self.train_cov = 0
        self.train_std = 0

        # Hack to get dimension, since input can be confusing:
        #   ex. test_data = [1, 2, 3] is 1D data, not 3D!
        #   but test_data = [[1,2], [2,3], [3,4]] would be 2D data, with 3 data points.
        if isinstance(self.train_data[0], np.float64) or isinstance(self.train_data[0],
            self.dim_d = 1
        else:
            self.dim_d = len(self.train_data[0])

    def train_diag(self):
        '''
        Calculate the mean and generate the covariance matrix
        '''

        # Get the mean
        self.train_mean = np.mean(self.train_data, axis = 0)

        # Get the covariance matrix
        #   We don't have to worry about "making it diagonal" here:
        #   when we predict, we will just do the operations on the diagonal parts of th
```

```
            #   effectively treating all other entries as zero.
            self.train_cov = np.cov(self.train_data)


        def predict_diag(self, test_data):
            '''
            Calculate the const = 1/(2 pi sigma^2 sqrt(det(cov))
            then, sample each point's density ~ 1/sigma^2 (x_i - \mu)^2
            finally, take the negative log
            '''

            # Hack to accomodate 1D data: the determinant of a 1x1 matrix is just that entr
            if self.dim_d == 1:
                det = self.train_cov
            else:
                # The determinant of a diagonal matrix is the product along the diagonal,
                #   and we are taking everything except the diagonal to be zero
                det = self.train_cov.diagonal().prod()

            # Calculate the constant (normalizing) term
            const_term = 1.0/np.multiply((2*math.pi)**(float(self.dim_d)/2.0), np.sqrt(det)

            # Calculate the sum part of the exponent
            sum = 0
            if self.dim_d == 1: # Hack for 1D data: no need to index in, since these "matri
                sum = (1.0/self.train_cov) * (test_data - self.train_mean)**2
            else:
                for i in range(self.dim_d-1):
                    sum += (1.0/self.train_cov[i,i]) * (test_data[i] - self.train_mean[i])*

            # Construct the e^() term
            exp_term = math.e**(-0.5 * sum)

            # Finally, calculate the log density
            density = math.log(const_term * exp_term)
            return density
```

We test the Gaussian estimator with some dummy data:

```
In [19]: data = np.array([[10 ,11,12], [13,10,8]])
         test_point = np.array([5,8,10])

         part2 = gaussianDiag(data)
         part2.train_diag()
         print(part2.predict_diag(test_point))
```

```
-25.29814999749476
```

1.2 2. Implementation of a Parzen density estimator

```
In [20]: class parzenEstimator:
             def __init__(self, data, kernel_stdev):
                 self.train_data = data
                 self.kernel_stdev = kernel_stdev
                 if isinstance(self.train_data[0], np.float64) or isinstance(self.train_data[0],
                     self.d = 1
                 else:
                     self.d = len(self.train_data[0])

             def train(self, train_data):
                 # Loading datapoints was taken care of in the constructor above.
                 # This function creates the lambda function that makes summing easier/faster in
                 # This metaphorically "creates" a Gaussian centered on each datapoint, while le
                 #  the test point open, until the weighting actually occurs at test time.

                 norm_const = 1.0/np.multiply((2*math.pi)**(self.d/2), self.kernel_stdev**self.d
                 self.summand = lambda x, mu : np.multiply(norm_const, np.exp(np.multiply(-0.5,

             def predict(self, test_point):
                 # Sum over the kernels with the datapoint as x
                 density = 0
                 for datapoint in self.train_data:
                     density += self.summand(test_point, datapoint)
                 log_density = np.log(density)
                 return log_density
```

We test the Parzen estimator with some dummy data:

```
In [21]: data = np.array([[2 ,2,3], [2,2,2]])
         test_point = np.array([1,2,3])

         part2 = parzenEstimator(data, 0.5)
         part2.train(data)
         print(part2.predict(test_point))
```

```
-2.315000408525107
```

1.3 3. 1D densities

First, let's open and parse the Iris dataset, selecting just the versicolor class:

```
In [22]: with open('iris.data', 'r') as fp:
             iris_data = fp.readlines()

         # Get just the versicolor class
         versicolor_lines = []
```

```
        for line in iris_data:
            if 'versicolor' in line:
                versicolor_lines.append(line.strip())
```

Then, let's further select just the data for sepal length:

```
In [23]: data_1d = []
         for line in versicolor_lines:
             data_1d.append(float(line.split(',')[0]))
         data_1d.sort()
         data_1d = np.array(data_1d)
```

1.3.1 1D density Part (a)

N.B. A plot of the data points of the subset is aggregated in one single plot with parzen and diagonal gaussian The plot is shown at the end of 1D density code

1.3.2 1D density Part (b)

A plot of the density estimated by the Gaussian estimator:

```
In [24]: # Instantiate and train the estimator
         oneD_param = gaussianDiag(data_1d)
         oneD_param.train_diag()

         def gaussian_full_density(data_1d):
             density_points = []
             for datapoint in data_1d:
                 # Calculate the density at each point
                 density_points.append(oneD_param.predict_diag(datapoint))
             return np.array(density_points)

         density_points_gaussian = gaussian_full_density(data_1d)
```

1.3.3 1D density Part (c)

Calculating the density estimated by the Parzen estimator, with a σ that is too small:

```
In [25]: # The entire density curve here consists of each p(x) for all x in the data,
         # calculated by using (all the dataset - x) as the training data,
         # and x as the test data.

         stdev = 0.1

         def parzen_full_density(data_1d, stdev):
             density_points = []
             for datapoint in data_1d:
                 # First get a list of all the data - the test point
                 for i, train_datapoint in enumerate(data_1d):
```

```
                if np.array_equal(datapoint, train_datapoint): #datapoint == train_datapoin
                    train_data = np.delete(data_1d, i)
                    break

            # Then calculate the weights of all the other points relative to the test point
            estimator = parzenEstimator(train_data, stdev)
            estimator.train(train_data)
            density_points.append(estimator.predict(datapoint))
        return np.array(density_points)

    density_points_small = parzen_full_density(data_1d, stdev)
```

1.3.4 1D density Part (d)

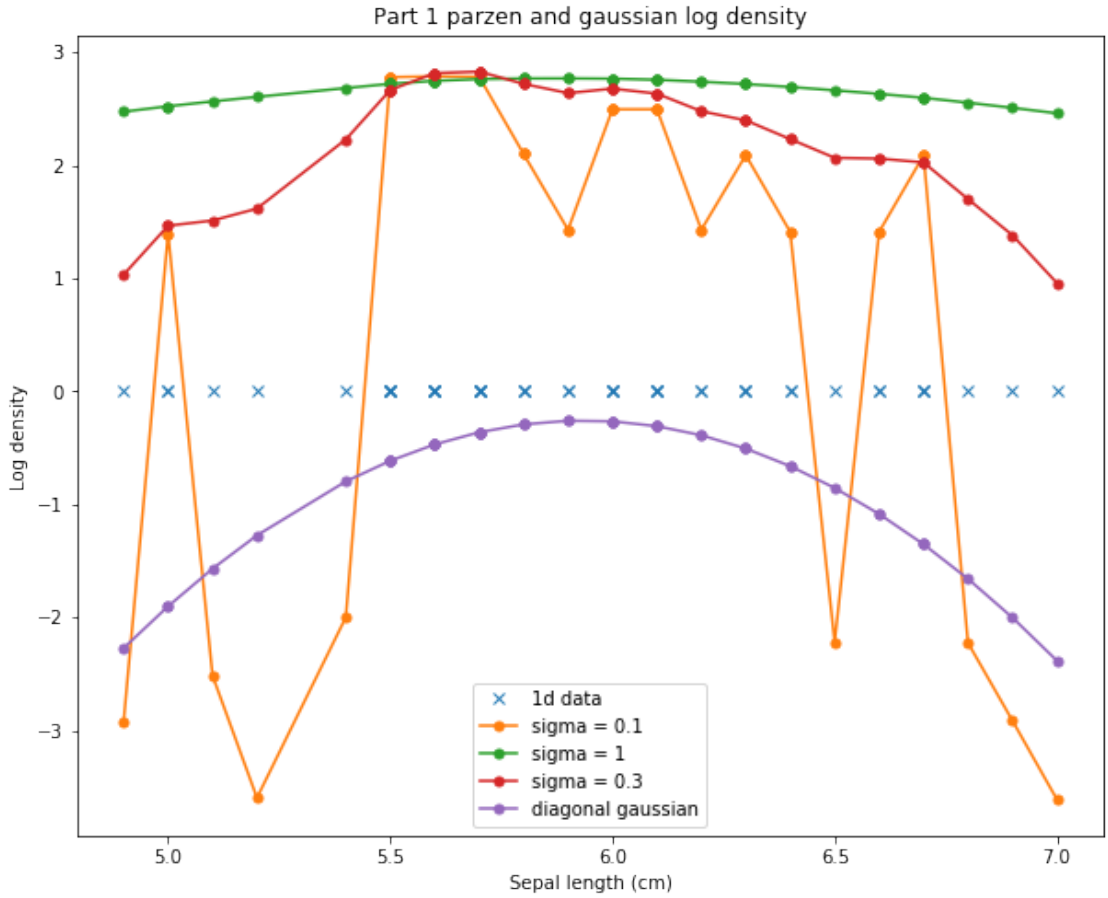
Calculating the density estimated by the Parzen estimator, with a σ that is too big:

```
In [26]: stdev = 1

         density_points_big = parzen_full_density(data_1d, stdev)
```

1.3.5 1D density Part (e)

A plot of the density estimated by the Parzen estimator, after also calculating the density with a σ that is an appropriate size:

```
In [27]: stdev = .3
         density_points_good = parzen_full_density(data_1d, stdev)

In [28]: import matplotlib.pyplot as plt
         # Make plots bigger
         plt.rcParams['figure.figsize'] = [10, 8]

         plt.plot(data_1d, np.zeros_like(data_1d) + 0., 'x', label = "1d data")
         #plt.yticks([])



         plt.plot(data_1d, density_points_small, marker="o", ms=5, label="sigma = 0.1")
         plt.plot(data_1d, density_points_big, marker="o", ms=5, label="sigma = 1")
         plt.plot(data_1d, density_points_good, marker="o", ms=5, label="sigma = {}".format(stde

         # Plot the result
         plt.plot(data_1d, density_points_gaussian, marker="o", ms=5, label ="diagonal gaussian"

         plt.legend(loc='best')
         plt.xlabel('Sepal length (cm)')
         plt.ylabel('Log density')
         plt.title('Part 1 parzen and gaussian log density')
         plt.show()
```



1.3.6 1D density Part (f)

The hyperparamter σ was "tuned" by hand, after a strategy modelled on binary search (i.e. test a midpoint between the two extremes, choosing the best half's endpoints as the new extremes until a satisfactory value was found). 0.3 seems to be an acceptable value because it strikes a balance between modelling where datapoints tend to concentrate while not overfitting and memorizing noise of our very limited dataset.

1.4 4. 2D densities

First, let's get the sepal length and sepal width from the versicolor irises:

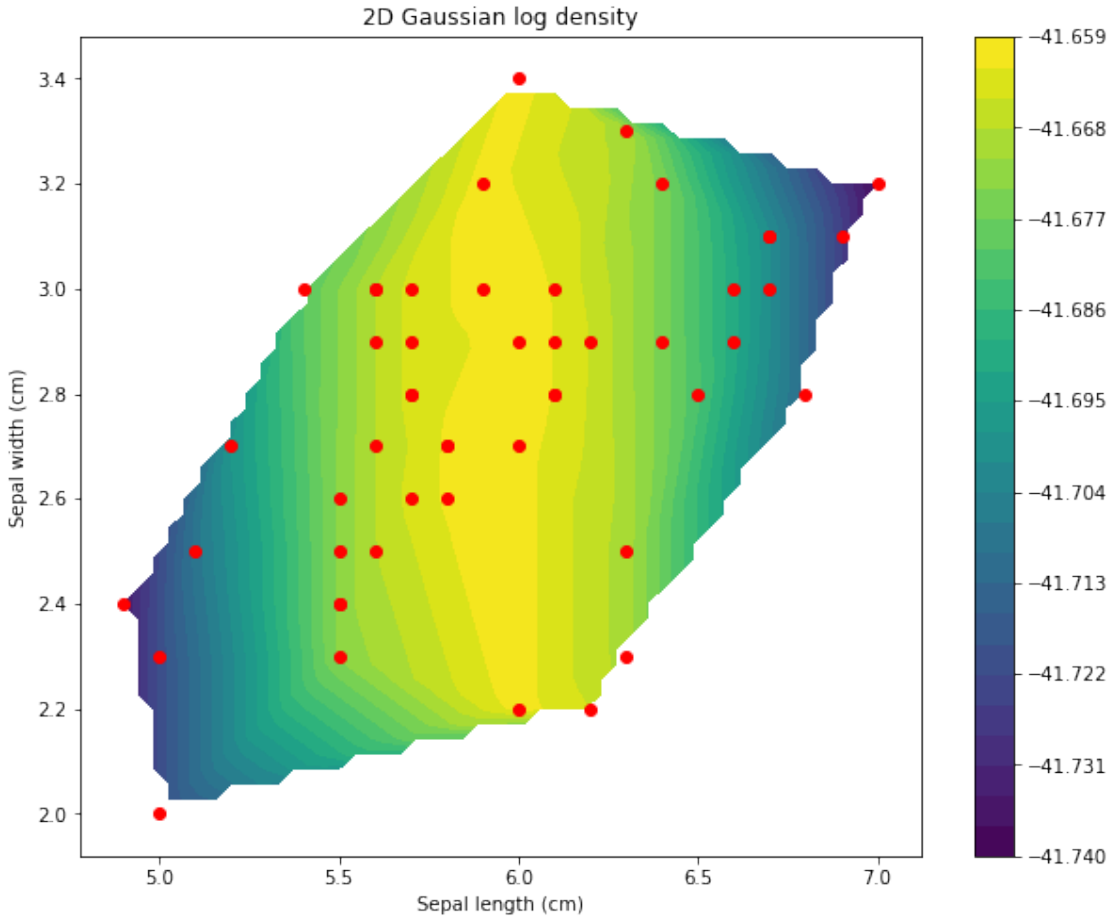
```
In [29]: data_2d = []
         for line in versicolor_lines:
             data_2d.append([float(line.split(',')[0]), float(line.split(',')[1])])

         data_2d = np.array(data_2d)
         sep_length = data_2d[:,0]
         sep_width = data_2d[:,1]
```

1.4.1 2D densities Part (a)

The density estimated by the diagonal Gaussian parametric estimator:

```
In [30]: # Instantiate and train the estimator
         twoD_param = gaussianDiag(data_2d)
         twoD_param.train_diag()

         def gaussian_full_density(data_2d):
             density_points = []
             for datapoint in data_2d:
                 # Calculate the density at each point
                 density_points.append(twoD_param.predict_diag(datapoint))
             return np.array(density_points)

         density_points = gaussian_full_density(data_2d)

         # Plot the scatter points on top of contour
         from matplotlib.mlab import griddata

         xi = np.linspace(min(sep_length), max(sep_length))
         yi = np.linspace(min(sep_width), max(sep_width))
         zi = griddata(sep_length, sep_width, density_points, xi, yi, interp='linear')

         plt.contourf(xi, yi, zi, 30)
         plt.colorbar();

         plt.scatter(sep_length, sep_width, color = 'r')

         plt.title('2D Gaussian log density')
         plt.xlabel('Sepal length (cm)')
         plt.ylabel('Sepal width (cm)')

         plt.show()
```

```
/Users/user/miniconda3/lib/python3.6/site-packages/ipykernel_launcher.py:19: MatplotlibDeprecati
```



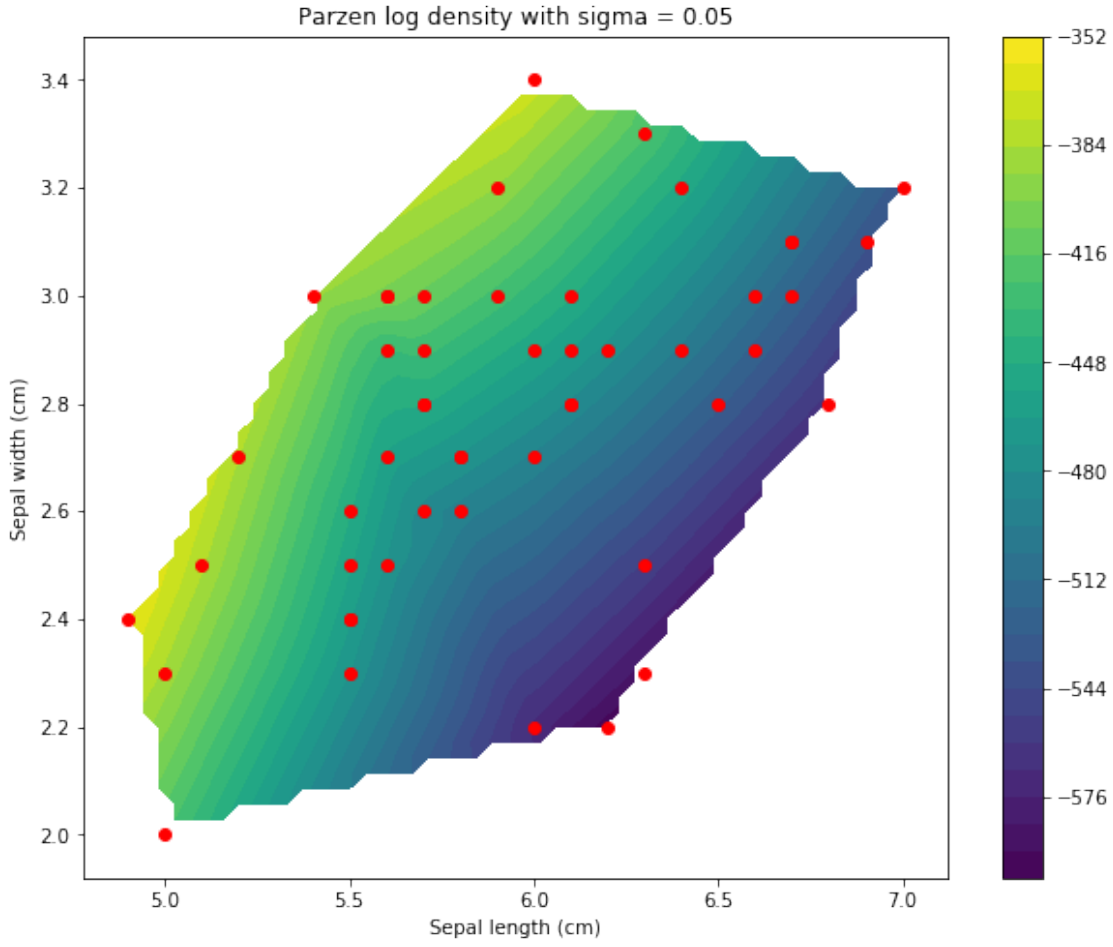
1.4.2 2D densities Part (b)

The density estimated by the Parzen estimator, with a σ that is too small:

```
In [31]: stdev = .05

         density_points = parzen_full_density(data_2d, stdev)

         def plot_parzen(sep_length, sep_width, density_points, stdev):
             # Plot the scatter points on top of contour
             from matplotlib.mlab import griddata

             xi = np.linspace(min(sep_length), max(sep_length))
             yi = np.linspace(min(sep_width), max(sep_width))
             zi = griddata(sep_length, sep_width, density_points, xi, yi, interp='linear')

             plt.contourf(xi, yi, zi, 30)
             plt.colorbar();
```

```
        plt.scatter(sep_length, sep_width,  color = 'r')

        plt.title('Parzen log density with sigma = {}'.format(stdev))
        plt.xlabel('Sepal length (cm)')
        plt.ylabel('Sepal width (cm)')

        plt.show()

    plot_parzen(sep_length, sep_width, density_points, stdev)
```

```
/Users/user/miniconda3/lib/python3.6/site-packages/ipykernel_launcher.py:11: MatplotlibDeprecati
  # This is added back by InteractiveShellApp.init_path()
```

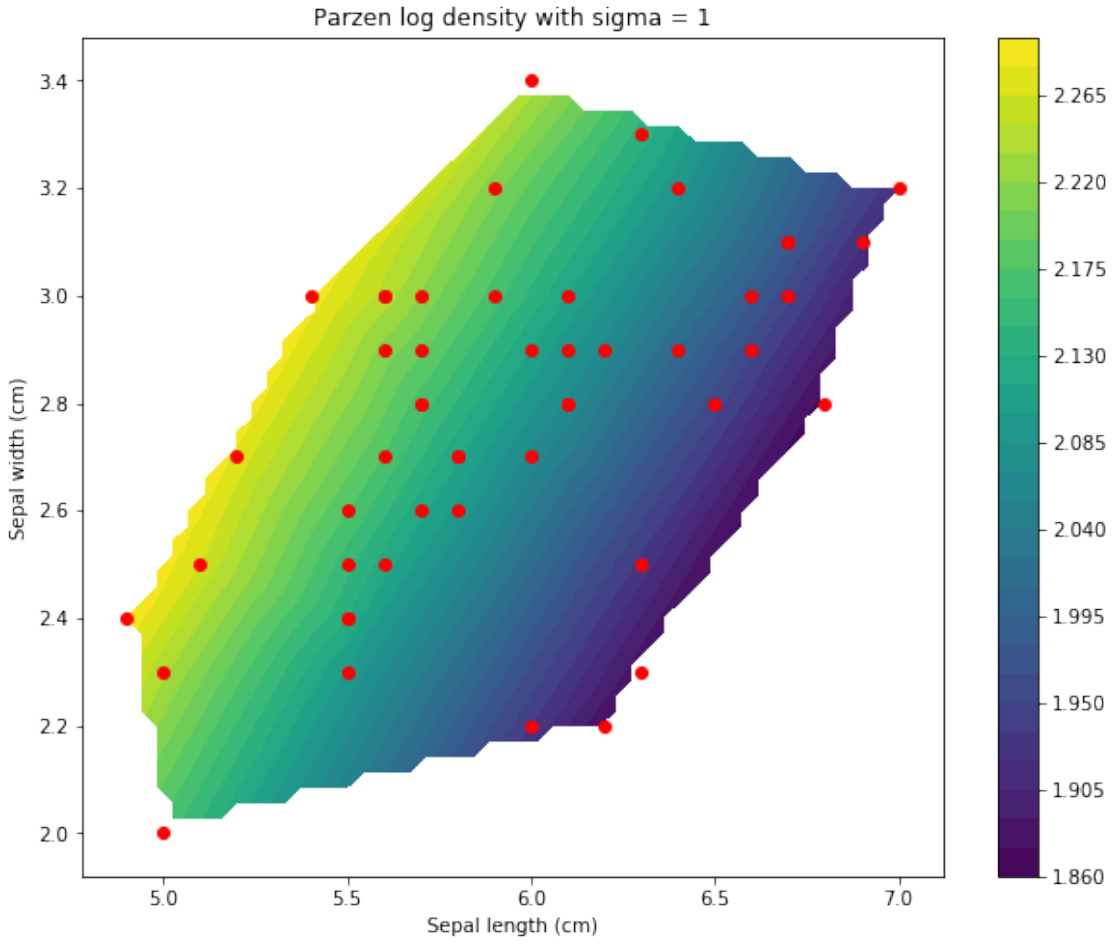


1.4.3 2D densities Part (c)

The density estimated by the Parzen estimator, with a σ that is too big:

```
In [32]: stdev = 1
         density_points = parzen_full_density(data_2d, stdev)

         plot_parzen(sep_length, sep_width, density_points, stdev)
```

```
/Users/user/miniconda3/lib/python3.6/site-packages/ipykernel_launcher.py:11: MatplotlibDeprecati
  # This is added back by InteractiveShellApp.init_path()
```

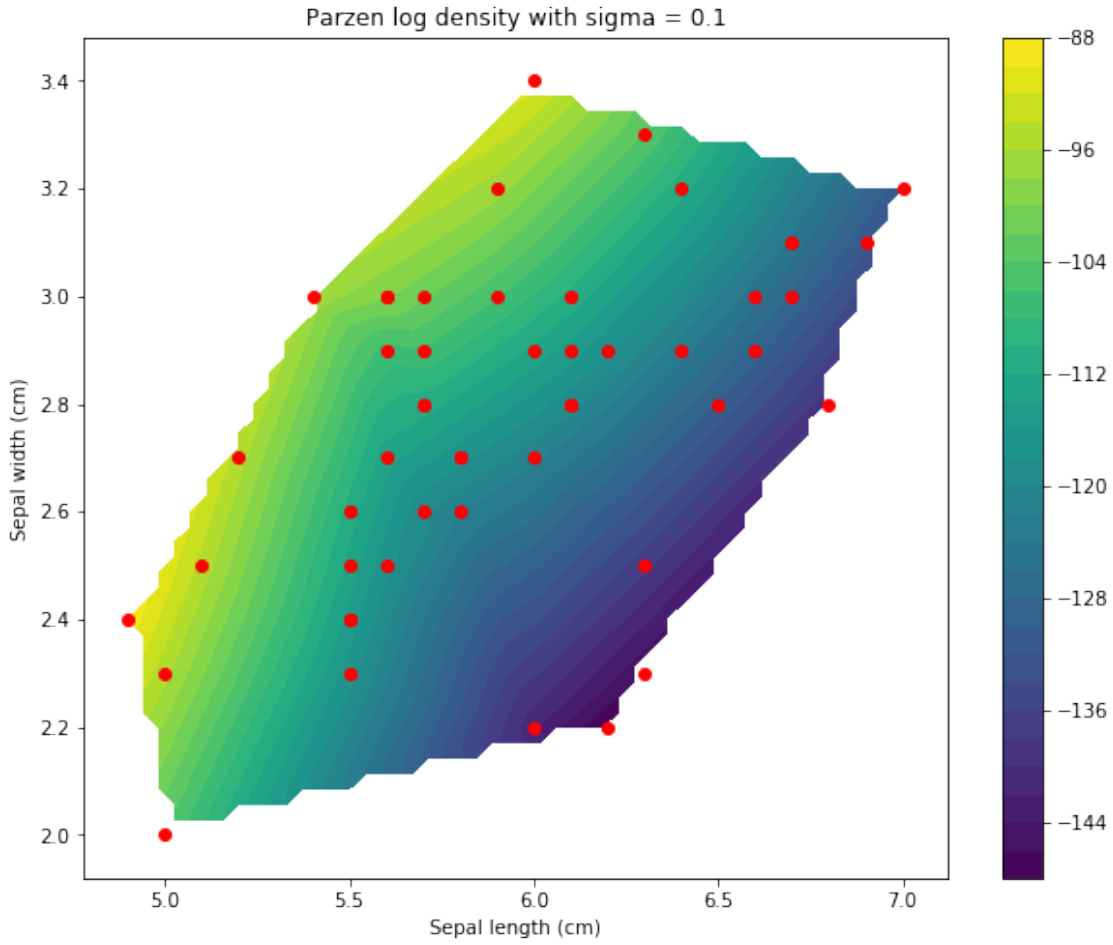


1.4.4 2D densities Part (d)

The density estimated by the Parzen estimator, with a σ that is an appropriate size:

```
In [33]: stdev = 0.1
         density_points = parzen_full_density(data_2d, stdev)

         plot_parzen(sep_length, sep_width, density_points, stdev)
```

```
/Users/user/miniconda3/lib/python3.6/site-packages/ipykernel_launcher.py:11: MatplotlibDeprecati
  # This is added back by InteractiveShellApp.init_path()
```



1.4.5 2D densities Part (f)

The hyperparamter σ was "tuned" by hand using a similar binary-search-like strategy outlined in Part (f) of Question 3. 0.1 seems to be an acceptable value because it captures more distribution detail than $\sigma = 1$, for example, which only models the main "ridge" of densities shown in yellow and red and simplifies it to a straight line shape, but memorizes less noise than $\sigma = 0.05$, which may be overfitting the "dead spot" shown in blue near the bottom-right of the graph. However, it is worth noting that this choice seems more arbitrary than the tuning done in Question 3. Testing values far below 0.05 to see extreme overfitting may be beneficial, but in this implementation is impossible, due to underflow/divide-by-zero errors.