gp-and-robust-sample-solution

July 14, 2023

1 Exercise 7

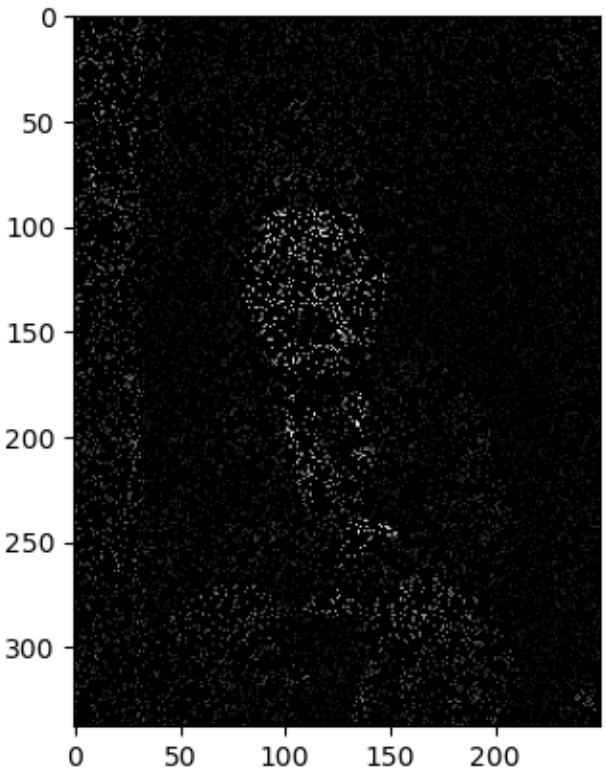
1.1 1) Gaussian process regression

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib
import scipy
from scipy import sparse
from imageio.v3 import imread
```

We first load the grayscale image:

[2]:
```
img = np.float32(imread("cc_90.png"))
```

[3]:
```
_ = plt.imshow(img, cmap='gray')
```



Next, we specify the kernel function:

[6]:
```
def kernel(x1, x2, gamma=1, h=1):
    r = np.sqrt(np.sum((x1-x2)**2, axis=-1)) / h
    return np.exp(-r**gamma)
```

[171]:
```
# naive implementation to build A (not sparse)
def build_A(X, gamma=1, h=1):
    N_train = X.shape[0]
    A = np.zeros((N_train, N_train))
    for i in range(N_train-1):
        A[i+1,(i+1):] = kernel(X[i], X[(i+1):,], gamma=gamma, h=h)
    A += A.T
    np.fill_diagonal(A, 1)
    return A
```

[195]:
```
# sparse implementation
def build_A_sparse(X, gamma=1, h=1, truncation=1e-6):
    N_train = X.shape[0]
    A_sparse = sparse.lil_array((N_train, N_train))
    for i in range(N_train):
        row = kernel(X[i], X, gamma=gamma, h=h)
```

```
        trunc = (row > truncation).nonzero()  # cut off is kernel value is small
        A_sparse[i, trunc] = row[trunc]
    return A_sparse.tocoo()
```

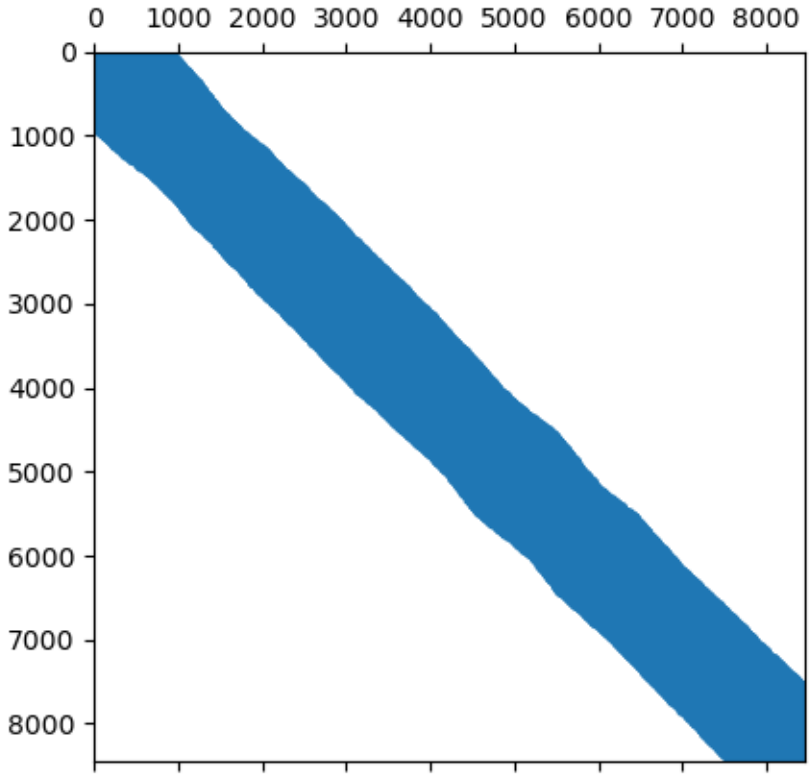
[196]:
```
gamma = 1.5
h = 4
```

[197]:
```
%%time
A = build_A_sparse(X_train, gamma=gamma, h=h)
```

```
CPU times: user 1.93 s, sys: 27.7 ms, total: 1.96 s
Wall time: 1.96 s
```

We can plot the sparsity pattern of the matrix:

[198]:
```
_ = plt.spy(A)
```



If we wanted to construct the complete vector b, we could use the following:

[177]:
```
def build_b(X_test, X_train, gamma=1, h=20, truncation=1e-2):
    b = sparse.lil_array((X_test.shape[0], X_train.shape[0]))
    for j in range(len(X_train)):
        column = kernel(X_test, X_train[j], gamma=gamma, h=h)
```

```
        trunc = (column > truncation).nonzero()
        b[trunc, j] = column[trunc]
    return b.tocoo()
```

[178]:
```
%%time
b = build_b(X_test, X_train, gamma=gamma, h=h)
```

```
CPU times: user 12.8 s, sys: 73.7 ms, total: 12.8 s
Wall time: 12.8 s
```

It turns out that looping over the test points is faster and doesn't require storing the matrix, so this is what we use here:

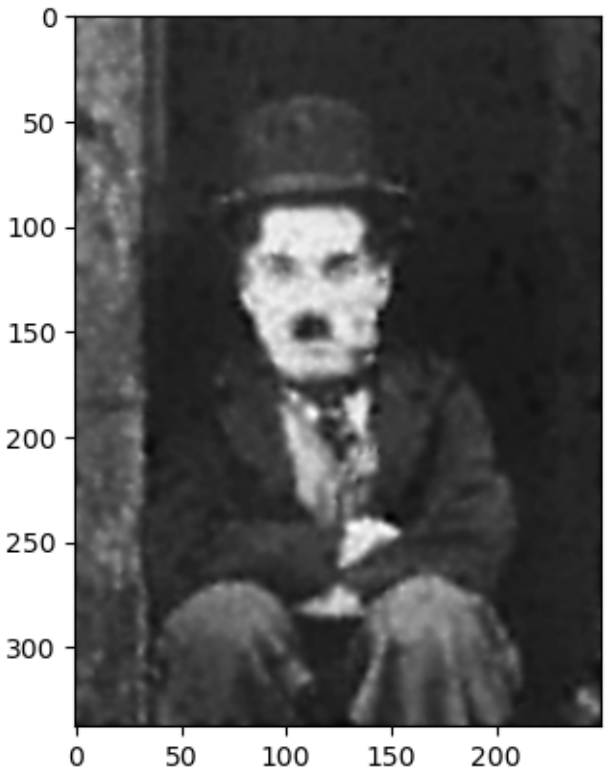
[206]:
```
def construct_image(img, gamma, h, sigma=1, truncation=1e-6, nwr=False):
    # split into "train" and "test" instances
    train_inds = img.nonzero()
    test_inds = (img == 0).nonzero()
    X_train = np.stack(train_inds).T
    Y_train = img[train_inds]
    X_test = np.stack(test_inds).T
    Y_pred = np.empty(X_test.shape[0])
    if nwr:
        # Nadaraya-Watson regression
        print("Computing predictions...")
        for i in range(X_test.shape[0]):
            b = kernel(X_test[i], X_train, gamma=gamma, h=h)
            Y_pred[i] = (b @ Y_train) / np.sum(np.abs(b), axis=-1)
    else:
        # Gaussian process regression
        print("Constructing matrix A...", end=" ")
        A = build_A_sparse(X_train, gamma=gamma, h=h, truncation=truncation)
        print(f"Sparsity: {A.count_nonzero()/np.prod(A.shape)**2}")
        sigma_2 = sigma ** 2
        print("Calculating z...")
        A_ = A + sparse.eye(A.shape[0], format='coo') * sigma_2
        # z = sparse.linalg.spsolve(A_, Y_train)
        l = u = 2000
        ab = np.zeros((l + u + 1, A_.shape[0]))
        for i, d in enumerate(range(u, -(l+1), -1)):
            if d > 0:
                ab[i, d:] = A_.diagonal(d)
            elif d == 0:
                ab[i] = A_.diagonal(d)
            else:
                ab[i, :d] = A_.diagonal(d)
        z = scipy.linalg.solve_banded((l, u), ab, Y_train)
        print("Computing predictions...")
        for i in range(X_test.shape[0]):
```

```
            b = kernel(X_test[i], X_train, gamma=gamma, h=h)
            Y_pred[i] = b @ z
        # b = build_b(X_test, X_train, gamma=gamma, h=h, truncation=truncation)
        # Y_pred = b @ z
    complete_img = img.copy()
    complete_img[test_inds] = Y_pred
    print("Done!")
    return complete_img
```

[207]:
```
%%time
full_img = construct_image(img, gamma=1.5, h=4, sigma=0.1)
```

```
Constructing matrix A… Sparsity: 2.603180615040242e-10
Calculating z…
Computing predictions…
Done!
CPU times: user 28.6 s, sys: 2.87 s, total: 31.4 s
Wall time: 19.9 s
```

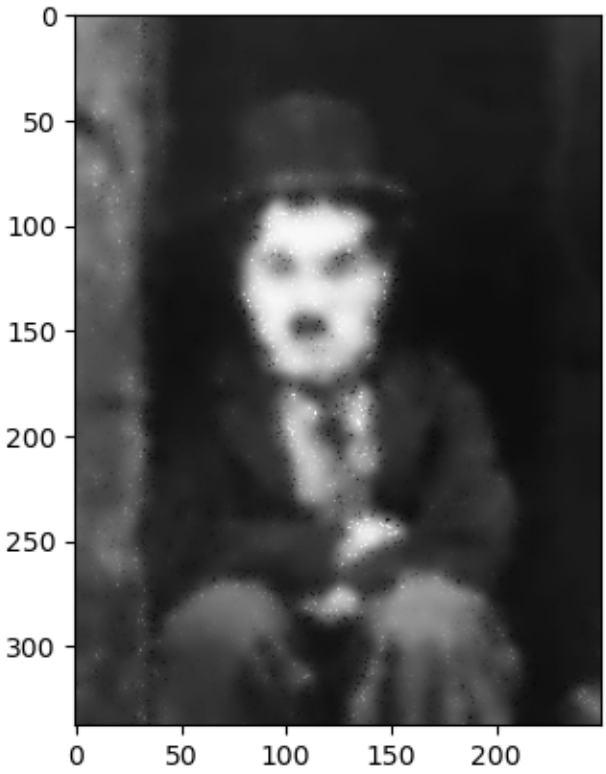
[208]:
```
_ = plt.imshow(full_img, cmap='gray')
```



[209]:
```
%%time
full_img = construct_image(img, gamma=1.5, h=4, sigma=0, nwr=True)
```

```
Computing predictions…
Done!
CPU times: user 12.9 s, sys: 0 ns, total: 12.9 s
Wall time: 12.9 s
```

[210]:
```
_ = plt.imshow(full_img, cmap='gray')
```



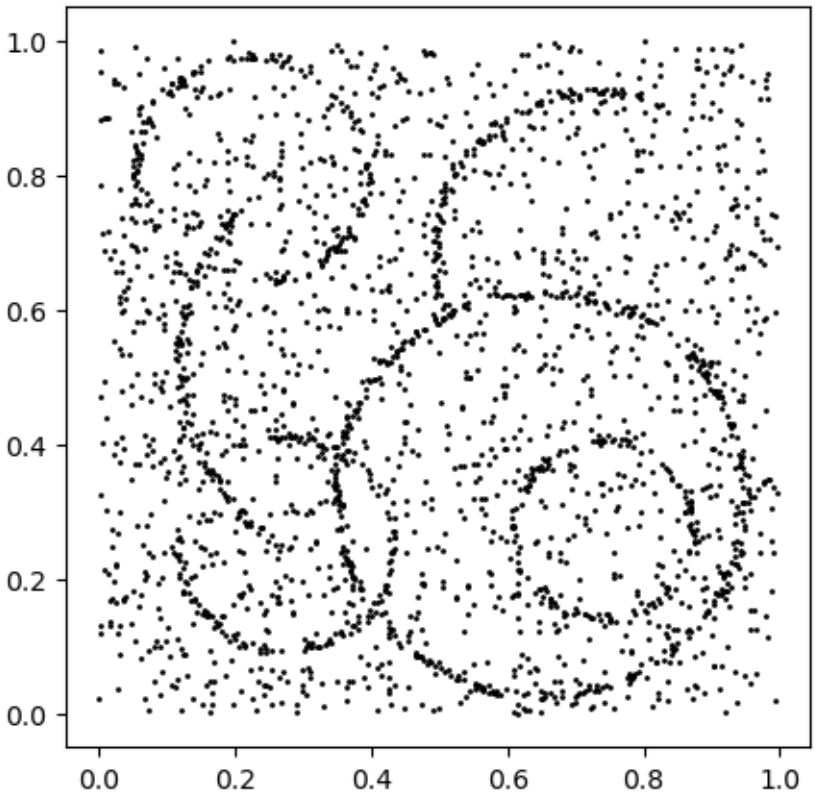
Nadaraya-Watson regression is slightly faster, but as constructing (or looping over) the vector b is the expensive part compared to constructing A and z, the difference is not that pronounced. Gaussian process regression gives more detail for the same kernel parameters.

1.2 2) Fitting Circles

First, we load the data set and visualize it.

[2]:
```
data = np.load("circles.npy")
```

[3]:
```
plt.figure(figsize=(5, 5))
_ = plt.scatter(data[:,0], data[:,1], marker="o", s=1, color="black")
```



Just by looking at it, we can identify segments of 6 different circles.

1.2.1 2.1) RANSAC

[4]:
```
def get_circle(points):
    """Calculate center and radius given three points"""
    A = np.concatenate([2 * points, np.ones((3, 1))], axis=-1)
    b = -np.sum(points ** 2, axis=-1)
    x = np.linalg.solve(A, b)
    center = np.array([-x[0], -x[1]])
    r = np.sqrt(np.sum(center**2)-x[2])
    return center, r
```

[5]:
```
def dist_to_circle(x, center, r):
    """Calculate the euclidean distance of points x to a circle"""
    return np.abs(np.sqrt(np.sum((x-center)**2, axis=-1))-r)
```

[6]:
```
def ransac_circle(data, T, epsilon=1e-2, inliers_relative=False):
    best_inliers = []
    best_circle = ((0, 0), 1)
```

```
    for r in range(T):
        # randomly chooes three points
        inds = np.random.choice(range(data.shape[0]), 3, replace=False)
        points = data[inds]
        # get circumcircle
        center, r = get_circle(points)
        # calculate euclidean distance for each point
        dists = dist_to_circle(data, center, r)
        # obtain inlier indices
        inliers = (dists < epsilon).nonzero()[0]
        if len(inliers) > len(best_inliers):
            # highest inlier count -> store
            best_inliers = inliers
            best_circle = center, r
    return best_inliers, best_circle
```

By looking at the data, we can guess the inlier fraction as roughly 10%.

[7]:
```
gamma = 0.1
alpha = 0.99
T = int(np.ceil(np.log(1-alpha) / np.log(1-gamma**3)))
print(f"T={T:.0f}")
```

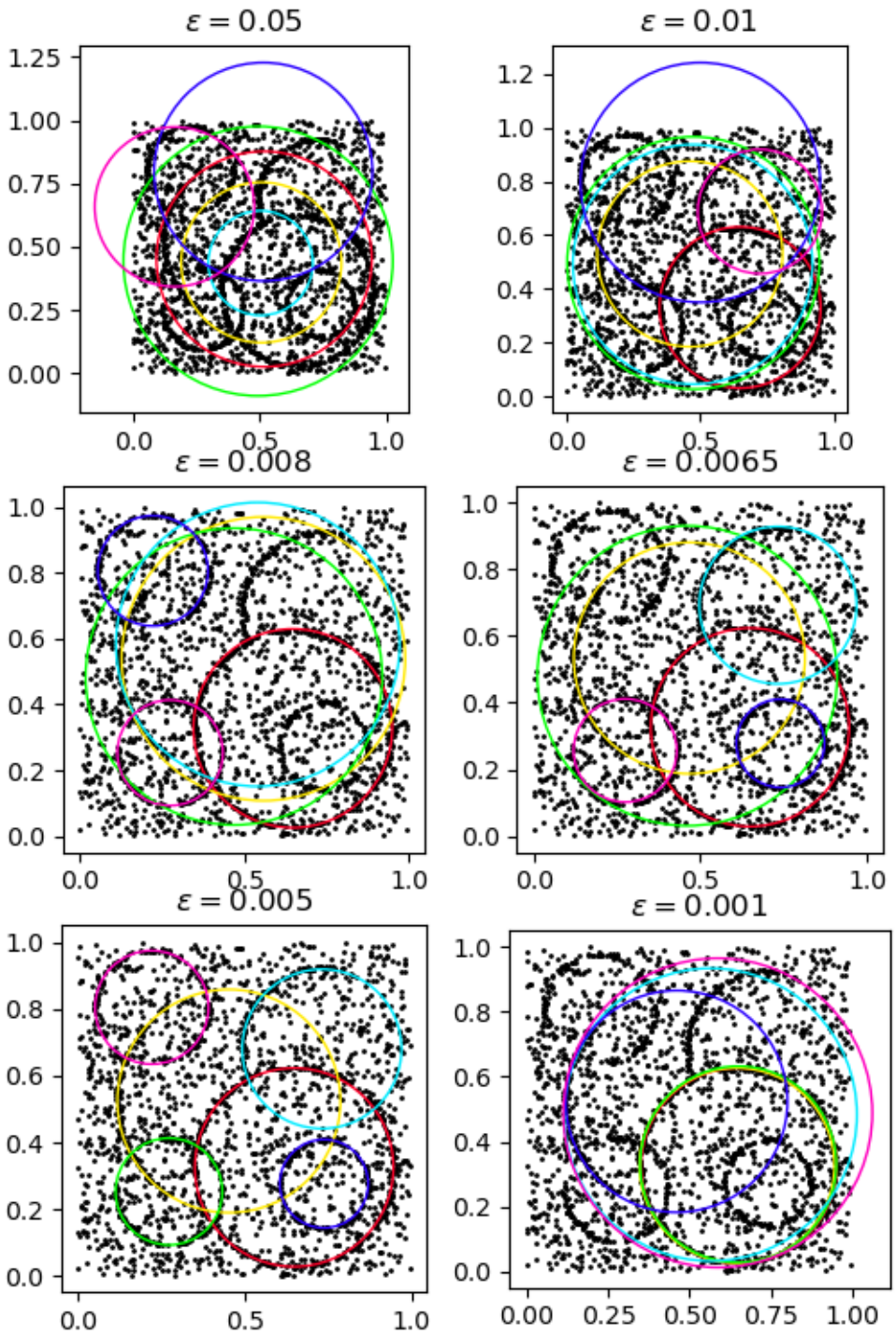
```
T=4603
```

[8]:
```
cmap = matplotlib.colormaps['gist_rainbow']  # set colormap for plotting circles
```

[9]:
```
def fit_circles_ransac(data, T=5000, N=6, epsilon=1e-2, ax=None):
    circles = []

    # prepare plotting
    if ax is None:
        ax = plt.gca()
    ax.scatter(data[:,0], data[:,1], marker="o", s=1, color='black')

    cur_data = data
    for i in range(N):
        # find circle
        inlier_inds, (c, r) = ransac_circle(cur_data, T, epsilon)
        circles.append((c, r))
        # remove inliers
        cur_data = np.delete(cur_data, inlier_inds, axis=0)

        # plot the circle
        circle = plt.Circle(c, radius=r, fill = False, color=cmap(i/(N-1))) #␣
 ↪Create a circle
        ax.add_patch(circle) # Add it to the plot
```

```
        ax.set_aspect('equal', 'box')
    return circles
```

[10]:
```
np.random.seed(2023)  # make cell reproducible
fig, axs = plt.subplots(3, 2, figsize=(6, 9))
epsilons = [5e-2, 1e-2, 8e-3, 6.5e-3, 5e-3, 1e-3]
for i, ax in enumerate(axs.flat):
    ax.set(title=f"$\\epsilon={epsilons[i]}$")
    fit_circles_ransac(data, epsilon=epsilons[i], ax=ax)
```

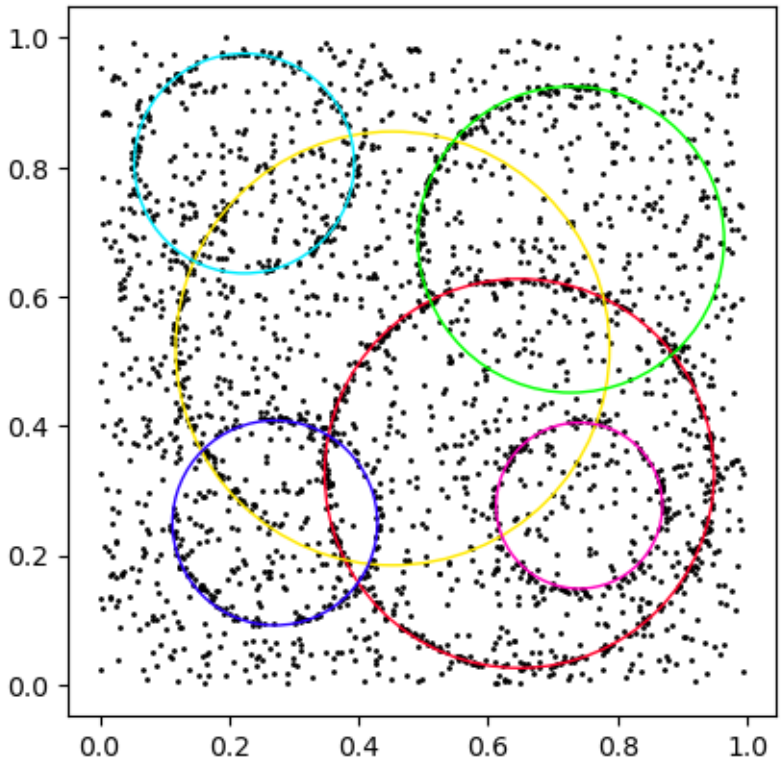


We can see different modes depending on ϵ. For large ϵ, we can observe that the circles just cover as much area as possible, as this leads to the highest number of inliers. For smaller number of ϵ, the results are better, but we can still observe a large circle that does not correspond to a true circle. For even smaller numbers, we see that the same circle is found multiple times, as not all inliers

are included. We could try to tune the scoring function, for example by evaluating the number of inliers divided by the circumference.

A good option here seems to be $\epsilon = 5 \cdot 10^{-3}$, so we use that to obtain the circles for the next tasks:

[11]:
```
circles = fit_circles_ransac(data, epsilon=5e-3)
```

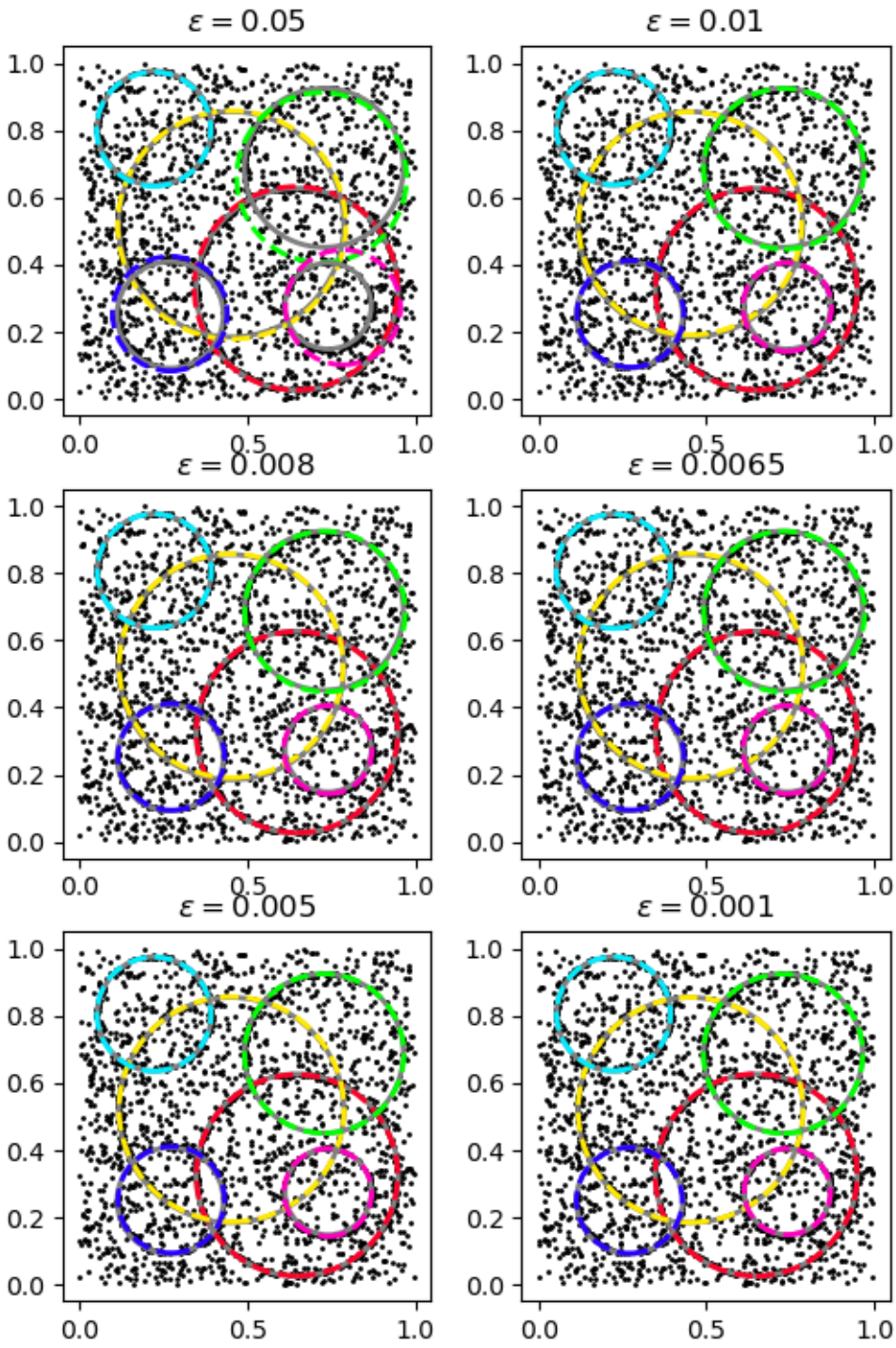


1.3 2.2) Non-linear least squares with the Levenberg-Marquardt algorithm

[12]:
```
def fit_circles_lm(data, circles, T=5000, epsilon=1e-2, max_iter=100, ax=None):
    circles_lm = []

    # prepare plotting
    if ax is None:
        ax = plt.gca()
    ax.scatter(data[:,0], data[:,1], marker="o", s=1, color="black")

    for i in range(len(circles)):
        c, r = circles[i]

        for iter in range(max_iter):
            # select inliers for optimization
            inliers = data[dist_to_circle(data, c, r) < epsilon]
```

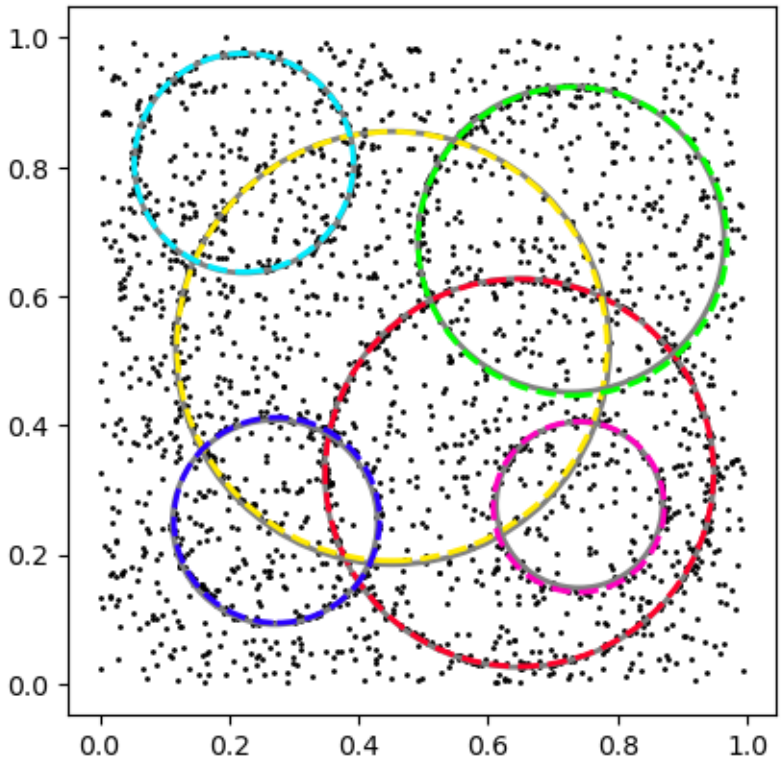
```
            res = scipy.optimize.least_squares(
                lambda p: np.linalg.norm(inliers - p[:2], axis=-1) - p[2],
                (c[0], c[1], r),
                method='lm'
            )

            c = res.x[:2]
            r = res.x[2]

            new_inliers = data[dist_to_circle(data, c, r) < epsilon]

            if len(new_inliers) == len(inliers) and \
                    np.all(inliers == new_inliers):
                # no change in inliers -> stop optimization
                break
            inliers = new_inliers

        circles_lm.append((c, r))

        # plot circles
        circle = plt.Circle(circles[i][0], radius=circles[i][1], fill = False,
                            color="gray", linewidth=2)
        ax.add_patch(circle)

        circle = plt.Circle(c, radius=r, fill = False, color=cmap(i/
↪(len(circles)-1)),
                            linestyle="dashed", linewidth=2)
        ax.add_patch(circle)
        ax.set_aspect('equal', 'box')
    return circles_lm
```

[13]:
```
np.random.seed(2023)  # make cell reproducible
fig, axs = plt.subplots(3, 2, figsize=(6, 9))
epsilons = [5e-2, 1e-2, 8e-3, 6.5e-3, 5e-3, 1e-3]
for i, ax in enumerate(axs.flat):
    ax.set(title=f"$\\epsilon={epsilons[i]}$")
    circles_lm = fit_circles_lm(data, circles, epsilon=epsilons[i], ax=ax)
```



As the optimization is conditional on the inliers selected by the RANSAC algorithm, the optimization doesn't change a lot. Here, we chose an iterative approach to allow for more adaptation. Intuitively, more changes are possile when ϵ is larger. If it is chosen too large, outliers will affect the fit.

[14]:
```
_ = fit_circles_lm(data, circles, epsilon=1e-2)
```



1.4 2.3) Algebraic distance

We are looking for the solution to the equation

$$\tilde{y} - \tilde{x} \cdot \beta = (x_1 - c_1)^2 + (x_2 - c_2)^2 - r^2 \tag{1}$$

$$x_1^2 + x_2^2 - x_1\beta_1 - x_2\beta_2 - \beta_3 = x_1^2 + x_2^2 - 2x_1c_1 + c_1^2 - 2x_2c_2 + c_2^2 - r^2. \tag{2}$$

By comparing the coefficients, it follows that:

$$2c_1 = \beta_1 \Leftrightarrow c_1 = \beta_1/2 \tag{3}$$

$$2c_2 = \beta_2 \Leftrightarrow c_2 = \beta_2/2 \tag{4}$$

$$c_1^2 + c_2^2 - r^2 = -\beta_3 \Leftrightarrow r^2 = \beta_3 + \frac{1}{4}(\beta_1^2 + \beta_2^2) \tag{5}$$

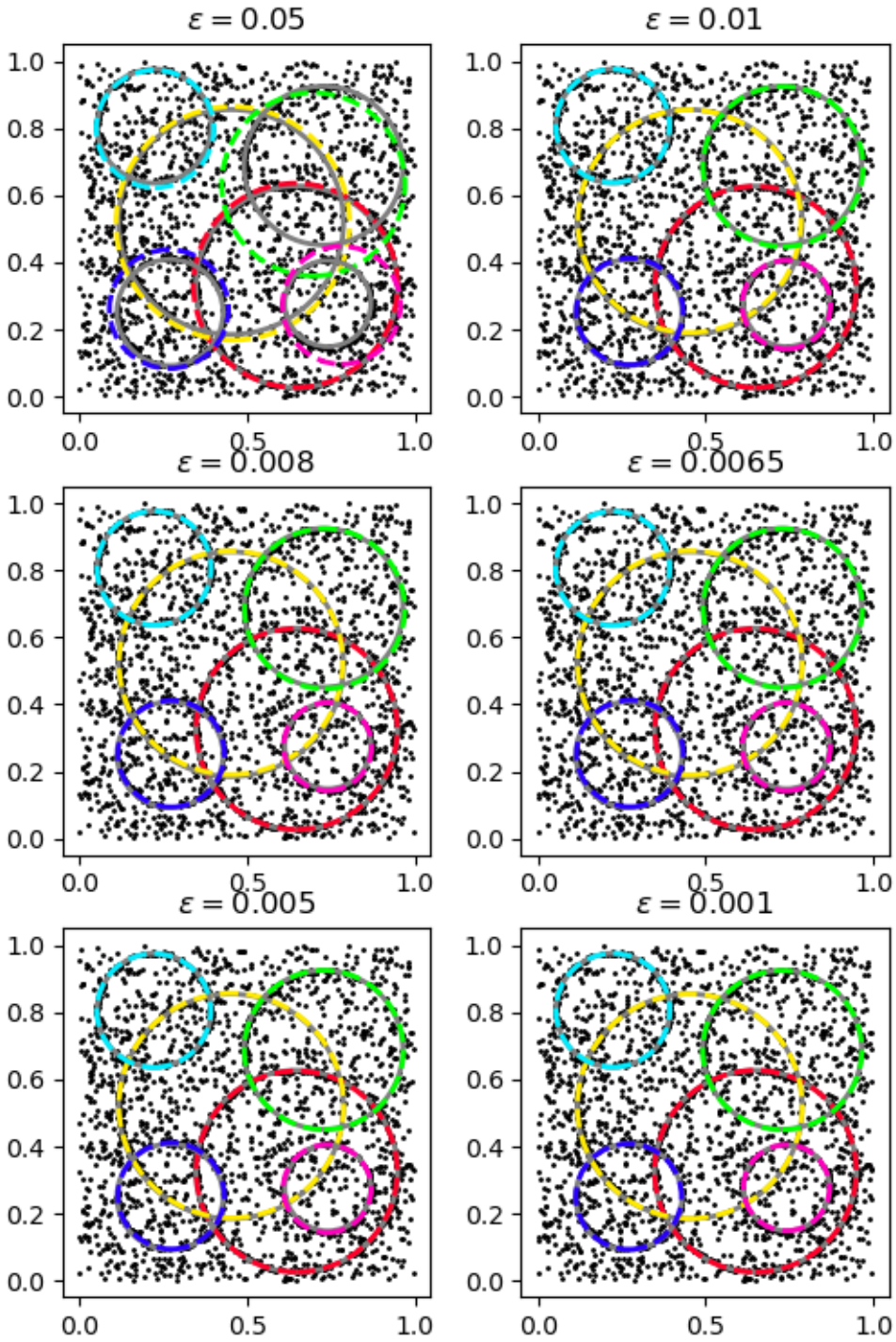
$$\tag{6}$$

[15]:
```
def fit_circles_ad(data, circles, T=5000, epsilon=1e-2, max_iter=100, ax=None):
    circles_ad = []

    # prepare plotting
    if ax is None:
```

```
        ax = plt.gca()
    ax.scatter(data[:,0], data[:,1], marker="o", s=1, color="black")

    for i in range(len(circles)):
        c, r = circles[i]

        for iter in range(max_iter):
            # select inliers for optimization
            inliers = data[dist_to_circle(data, c, r) < epsilon]

            x_tilde = np.concatenate([inliers, np.ones((inliers.shape[0], 1))],␣
↪axis=-1)
            y_tilde = np.sum(inliers**2, axis=-1)
            res = scipy.optimize.least_squares(
                lambda p: y_tilde - x_tilde @ p,
                (c[0], c[1], r)
            )

            c = 0.5*res.x[:2]
            r = np.sqrt(res.x[2] + 0.25*(res.x[0]**2 + res.x[1]**2))

            new_inliers = data[dist_to_circle(data, c, r) < epsilon]

            if len(new_inliers) == len(inliers) and \
                    np.all(inliers == new_inliers):
                # no change in inliers -> stop optimization
                break

            inliers = new_inliers

        circles_ad.append((c, r))

        # plot circles
        circle = plt.Circle(circles[i][0], radius=circles[i][1], fill = False,
                            color="gray", linewidth=2)
        ax.add_patch(circle)

        circle = plt.Circle(c, radius=r, fill = False, color=cmap(i/
↪(len(circles)-1)),
                            linestyle="dashed", linewidth=2)
        ax.add_patch(circle)
        ax.set_aspect('equal', 'box')
    return circles_ad
```

[16]:
```
np.random.seed(2023)  # make cell reproducible
fig, axs = plt.subplots(3, 2, figsize=(6, 9))
epsilons = [5e-2, 1e-2, 8e-3, 6.5e-3, 5e-3, 1e-3]
```

```
for i, ax in enumerate(axs.flat):
    ax.set(title=f"$\\epsilon={epsilons[i]}$")
    circles_ad = fit_circles_ad(data, circles, epsilon=epsilons[i], ax=ax)
```



Again, as the optimization is conditional on the inliers selected by the RANSAC algorithm, the optimization doesn't change a lot.

[17]:
```
_ = fit_circles_ad(data, circles, epsilon=5e-3)
```