Exercise 7

Deadline: 14.07.2023, 16:00

This exercise is devoted to Gaussian processes and robust regression. The required data can be found at `https://tinyurl.com/HD-EML-ex07-material-zip` and as external link on MaMPF.

Regulations

Please hand in your solution as a Jupyter notebook `gp-and-robust.ipynb`, accompanied with `gp-and-robust.html`. **It is important that you stick to these file naming conventions!** Zip all files into a single archive `ex06.zip` and upload this file to MaMPF before the given deadline. Moreover, please set your **Anzeigename**/**display name** and **Name in Uebungsgruppen**/**name in tutorials** in MaMPF to your real name, which should be identical to your name in `muesli` and make sure you **join the submission** of your team via the invitation code before the submission deadline. Check out `https://mampf.blog/handing-in-homework-assignments` for instructions.

1 Gaussian process regression (20 points)

We learned in the lecture that Gaussian processes with a radial basis function kernel can interpolate or approximate functions whose values are known at arbitrary locations ("scattered data"). Here, we want to use this to reconstruct all missing pixels in the grayscale image `cc_90.png` (in `https://tinyurl.com/HD-EML-ex07-material-zip`). The features X_i are the pixel coordinates, and the response y_i is the corresponding grayvalue. Pixels with grayvalue $= 0$ are considered missing and shall be replaced with their regressed values. Since about 90% of the pixels are missing, not much is to be seen in `cc_90.png`.

We will use the generalized exponential kernel with hyper-parameters $\gamma \in (0, 2]$ and h:

$$K(\mathbf{x}, \mathbf{x}'; \gamma, h) = \exp\left(-r(\mathbf{x}, \mathbf{x}', h)^\gamma\right) \tag{1}$$

$$r(\mathbf{x}, \mathbf{x}'; h) = \sqrt{\frac{\|\mathbf{x} - \mathbf{x}'\|^2}{h^2}} \tag{2}$$

Recall from the lecture that we collected the pairwise similarities between the given (training) points in a matrix $\mathbf{A}$ with elements

$$\mathbf{A}_{ii'} = K(\mathbf{x}_i, \mathbf{x}_{i'}; \gamma, h) \tag{3}$$

The similarity of a test point $\widetilde{\mathbf{x}}$ with the training set is collected in the vector $\mathbf{b}$ with elements

$$\mathbf{b}_i = K(\mathbf{x}_i, \widetilde{\mathbf{x}}; \gamma, h) \tag{4}$$

The mean of the predicted grayvalue $\hat{y}$ at location $\widetilde{\mathbf{x}}$ is calculated from the given grayvalues $\mathbf{y}$ as

$$\hat{y} = \mathbf{b}^T \cdot (\mathbf{A} + \sigma^2 \cdot \mathbf{I})^{-1} \cdot \mathbf{y} \tag{5}$$

where σ^2 is the noise variance of the $\mathbf{y}$ and $\mathbf{I}$ the unit matrix. If you implement this formula naively, your program will probably be very slow. Use the following tricks to speed-up the execution:

- Cut off the kernel function at a sensibly large radius (i.e. set it to zero when the similarity gets reasonably small) to make $\mathbf{A}$ sparse, and use a sparse matrix class to store it (see ex. 4).

- Since $\mathbf{z} = (\mathbf{A}+\sigma^2 \cdot \mathbf{I})^{-1} \cdot \mathbf{y}$ is the same for each test point, it should be pre-computed. However, computing this directly with the matrix inverse is much more expensive than solving the linear system

$$(\mathbf{A} + \sigma^2 \cdot \mathbf{I}) \cdot \mathbf{z} = \mathbf{y} \tag{6}$$

  The difference is especially large for sparse matrices, and you could solve the equation with the function `scipy.sparse.linalg.spsolve`. However, an even bigger speed-up can be achieved by noticing that matrix $\mathbf{A}$ is actually banded (it has $\approx$ 2000 non-zero diagonals). To exploit this, construct the matrix with `coo_matrix.diagonal` and solve the linear system with `scipy.linalg.solve_banded`.
- Avoid loops and use vectorization when possible.

Play with the hyper-parameters γ, h, σ of the model to optimize the quality of the reconstructed image.

A simpler approximation to Gaussian process regression can be achieved by Nadaraya-Watson regression. It avoides the expensive matrix inversion by defining the regression of a test point as:

$$\hat{y} = \frac{\mathbf{b}^T \cdot \mathbf{y}}{\|\mathbf{b}\|_1} \tag{7}$$

Implement this approach as well, and comment on the speed and quality of the two methods.

2 Fitting Circles

In this exercise we will have a closer look at fitting circles to data. The numpy-file `circles.npy` (in `https://tinyurl.com/HD-EML-ex07-material-zip`) contains many pairs of x-y-coordinates, and can be loaded through `data = np.load("circles.npy")`. Visualize the data in a scatter plot to show that the points are arranged in the shape of several circles and circle segments. Pay attention that the axes are scaled identically when plotting the data, otherwise your circles will look like ellipses. How many circles or circle segments would you fit into the data as a human?

2.1 RANSAC (8 Points)

Since the points of each circle are outliers to all other circles, we need a robust regression algorithm to fit the circles. To this end, implement the RANSAC algorithm and specialize it for the fitting of circles:

- For a set number of times T, repeat the following:
  - Randomly choose 3 points and determine their circumcircle, parametrized by its radius and the coordinates of the center. Derive or look-up the formula for the circumcircle, given three points.
  - Classify points as inliers whose Euclidean distance to this circle is less than ϵ (y hyper-parameter). Count the inliers.
  - If the inlier count for this circle is higher than for the best circle so far, save the current circle and its inliers as the new best.
- Fit further circles by deleting all inliers of the last fitted circle from the dataset and repeat the procedure.

Estimate the number of iterations T that is needed, as described in the lecture, using a rough guess of the inlier fraction for a single circle.

Plot all fitted circles on top of the original data and comment on the result. Implement experiments to find out how sensitive the result is to the choice of ϵ (the threshold for the inlier distance) and use a good value for the final run.

Hint: For plotting circles, you can use the following methods:

```
circle = plt.Circle((cx, cy), radius=r, fill=False) # Create a circle
plt.gca().add_patch(circle) # Add it to the plot
```

In the next two sub-tasks, we will further improve the fits of the circles using two different methods. If you were unable to implement RANSAC, then either try to fit a circle manually to the data and get the inliers this way, or create data for a circle + noise and use it for the rest of the exercise.

2.2 Non-linear least squares with the Levenberg-Marquardt algorithm (5 Points)

We can get more accurate models by fitting the circles to *all* of their inliers (instead of just three points). To this end, we minimize the squared Euclidean distances between the circle and its inliers

$$\hat{c}, \hat{r} = \arg\min_{c,r} \sum_i \left(\|x_i - c\|_2 - r \right)^2 \tag{8}$$

Due to the square root in the norm $\|x_i - c\|_2$, this is a non-linear least squares problem. It can be solved by the Levenberg-Marquardt algorithm[1]. Optimize each circle from subtask **??** with the library function `scipy.optimize.least_squares` and use the RANSAC solutions as initial guess. Plot the resulting circles, investigate the influence of the inlier threshold ϵ, and comment on your findings.

2.3 Algebraic Distance (7 Points)

In the old days, when library implementations of the Levenberg-Marquardt algorithm were not yet widely available, an alternative approach reformulated circle fitting as an ordinary least squares problem. This can be achieved by using the squared norm instead of the norm, resulting in an optimization over the algebraic distances:

$$\hat{c}, \hat{r} = \arg\min_{c,r} \sum_i \left(||x_i - c||_2^2 - r^2 \right)^2 \tag{9}$$

This is still non-linear, but can be transformed into a linear least squares problem by the non-linear transformations

$$\widetilde{x}_i = [x_{i1}, x_{i2}, 1] \qquad \widetilde{y}_i = x_{i1}^2 + x_{i2}^2 \tag{10}$$

where x_{i1}, x_{i2} are the coordinates of inlier i. The solution of the OLS problem

$$\hat{\beta} = \arg\min_{\beta} \sum_i \left(\widetilde{y}_i - \widetilde{x}_i \cdot \beta \right)^2 \tag{11}$$

gives a 3-dimensional weight vector $\hat{\beta} = [\beta_1, \beta_2, \beta_3]^T$, and the desired circle parameters $\hat{c}, \hat{r}$ can be obtained by non-linear transformations of $\beta_1, \beta_2, \beta_3$. Derive the formulas for $\hat{c}$ and $\hat{r}$ and implement the method using the pseudo-inverse or the function `scipy.optimize.least_squares`. Again, plot the resulting circles, investigate the influence of the inlier threshold ϵ, and comment on your findings.

[1] Since the Levenberg-Marquardt algorithm was not covered in the lecture this time, you can watch last year's recordings at `https://mampf.mathi.uni-heidelberg.de/media/17998/play`, starting around 36'30