CS 189 Introduction to Machine Learning
Summer 2018

HW4

Your self-grade URL is http://eecs189.org/self_grade?question_ids=1_1,1_2,2_1,2_2,2_3,2_4,2_5,3_1,3_2,3_3,3_4,3_5,3_6,3_7,4_1,4_2,4_3,5_1,5_2,5_3,6.

This homework is due **Monday, July 16 at 11:59pm.**

2 Kernel Ridge Regression: Theory

In ridge regression, we are given a vector $\mathbf{y} \in \mathbb{R}^n$ and a matrix $\mathbf{X} \in \mathbb{R}^{n\times\ell}$, where n is the number of training points and ℓ is the dimension of the raw data points. In most settings we don't want to work with just the raw feature space, so we augment the data points with features and replace $\mathbf{X}$ with $\mathbf{\Phi} \in \mathbb{R}^{n\times d}$, where $\boldsymbol{\phi}_i^\top = \boldsymbol{\phi}(\mathbf{x}_i) \in \mathbb{R}^d$. Then we solve a well-defined optimization problem that involves the matrix $\mathbf{\Phi}$ and $\mathbf{y}$ to find the parameters $\mathbf{w} \in \mathbb{R}^d$. Note the problem that arises here. If we have polynomial features of degree at most p in the raw ℓ dimensional space, then there are $d = \binom{\ell+p}{p}$ terms that we need to optimize, which can be very, very large (much larger than the number of training points n). Wouldn't it be useful, if instead of solving an optimization problem over d variables, we could solve an equivalent problem over n variables (where n is potentially much smaller than d), and achieve a computational runtime independent of the number of augmented features? As it turns out, the concept of kernels (in addition to a technique called the kernel trick) will allow us to achieve this goal.

(a) (Dual perspective of the kernel method) In lecture, you saw a derivation of kernel ridge regression involving Gaussians and conditioning. There is also a pure optimization perspective that uses Lagrangian multipliers to find the dual of the ridge regression problem. First, we could rewrite the original problem as

$$\begin{aligned} \underset{\mathbf{w},\mathbf{r}}{\text{minimize}} \quad & \frac{1}{2}\left[\|\mathbf{r}\|_2^2 + \lambda\|\mathbf{w}\|_2^2\right] \\ \text{subject to} \quad & \mathbf{r} = \mathbf{X}\mathbf{w} - \mathbf{y}. \end{aligned}$$

Show that the solution of this is equivalent to

$$\min_{\mathbf{w},\mathbf{r}} \max_{\boldsymbol{\alpha}} L(\mathbf{w},\mathbf{r},\boldsymbol{\alpha}) := \min_{\mathbf{w},\mathbf{r}} \max_{\boldsymbol{\alpha}} \left[\frac{1}{2}\|\mathbf{r}\|_2^2 + \frac{\lambda}{2}\|\mathbf{w}\|_2^2 + \boldsymbol{\alpha}^\top(\mathbf{r} - \mathbf{X}\mathbf{w} + \mathbf{y})\right], \tag{1}$$

where $L(\mathbf{w},\mathbf{r},\boldsymbol{\alpha})$ is the Lagrangian function.

Solution: The Lagrangian is

$$L(\mathbf{w},\mathbf{r},\boldsymbol{\alpha}) = \frac{1}{2}\|\mathbf{r}\|_2^2 + \frac{\lambda}{2}\|\mathbf{w}\|_2^2 + \boldsymbol{\alpha}^\top(\mathbf{r} - \mathbf{X}\mathbf{w} + \mathbf{y}). \tag{2}$$

We can then argue the equivalence of the two optimization problems in the following way (the argument is similar to the one you used in the "Geometry of Ridge Regression" problem in HW2): If $\mathbf{w}$ and $\mathbf{r}$ in the outer optimization problem are chosen such that $\mathbf{r} = \mathbf{X}\mathbf{w} - \mathbf{y}$, the two optimization problems coincide. If $\mathbf{r} \neq \mathbf{X}\mathbf{w} - \mathbf{y}$, say for index i we have $\mathbf{r}_i \neq (\mathbf{X}\mathbf{w} - \mathbf{y})_i$, by driving α_i to ∞ or $-\infty$, the inner maximum will be ∞ and therefore points with $\mathbf{r} \neq \mathbf{X}\mathbf{w} - \mathbf{y}$ will not obtain the outer minimum, which means the constraint ends up being satisfied for the solution.

(b) Using the minmax theorem[1], we can swap the min and max (think about what the order of min and max means here and why it is important):

$$\min_{\mathbf{w},\mathbf{r}} \max_{\boldsymbol{\alpha}} L(\mathbf{w}, \mathbf{r}, \boldsymbol{\alpha}) = \max_{\boldsymbol{\alpha}} \min_{\mathbf{w},\mathbf{r}} L(\mathbf{w}, \mathbf{r}, \boldsymbol{\alpha}). \tag{3}$$

Argue that the solution for α of the right hand side of (2) is equal to

$$\arg\min_{\boldsymbol{\alpha}} \left[\frac{1}{2}\boldsymbol{\alpha}^\top(\mathbf{K} + \lambda\mathbf{I})\boldsymbol{\alpha} - \lambda\boldsymbol{\alpha}^\top\mathbf{y}\right] \text{ where } \mathbf{K} = \mathbf{X}\mathbf{X}^\top \in \mathbb{R}^{n\times n}. \tag{4}$$

You can do this by setting the appropriate partial derivative of the Lagrangian L to zero. This is often call *the Lagrangian dual problem* of the original optimization problem.

Solution: To get the solution of the inner minimization problem, we solve $\boldsymbol{\nabla}_{\mathbf{w},\mathbf{r}} L(\mathbf{w}, \mathbf{r}) = 0$. Writing the partial derivatives out, we get

$$\frac{\partial L}{\partial \mathbf{w}}(\mathbf{w}, \mathbf{r}, \boldsymbol{\alpha}) = 0 \implies \lambda\mathbf{w} - \mathbf{X}^\top\alpha = 0 \implies \mathbf{w}^* = \frac{1}{\lambda}\mathbf{X}^\top\boldsymbol{\alpha} \tag{5}$$

$$\frac{\partial L}{\partial \mathbf{r}}(\mathbf{w}, \mathbf{r}, \boldsymbol{\alpha}) = 0 \implies \mathbf{r} + \boldsymbol{\alpha} = 0 \implies \mathbf{r}^* = -\boldsymbol{\alpha}. \tag{6}$$

Plugging them into the Lagrangian, we get

$$L(\mathbf{w}^*, \mathbf{r}^*, \boldsymbol{\alpha}) = \frac{1}{2}\|\boldsymbol{\alpha}\|_2^2 + \frac{1}{2\lambda}\left\|\mathbf{X}^\top\boldsymbol{\alpha}\right\|_2^2 + \boldsymbol{\alpha}^\top(-\boldsymbol{\alpha} - \frac{1}{\lambda}\mathbf{X}\mathbf{X}^\top\mathbf{w} + \mathbf{y}) \tag{7}$$

$$= -\frac{1}{2}\|\boldsymbol{\alpha}\|_2^2 - \frac{1}{2\lambda}\boldsymbol{\alpha}^\top\mathbf{X}\mathbf{X}^\top\boldsymbol{\alpha} + \boldsymbol{\alpha}\mathbf{y} \tag{8}$$

and therefore the dual problem is $\max_{\boldsymbol{\alpha}} L(\mathbf{w}^*, \mathbf{r}^*, \boldsymbol{\alpha})$, which is equivalent to

$$\min_{\boldsymbol{\alpha}} \left[\frac{1}{2}\boldsymbol{\alpha}^\top(\mathbf{K} + \lambda\mathbf{I})\boldsymbol{\alpha} - \lambda\boldsymbol{\alpha}^\top\mathbf{y}\right],$$

where $\mathbf{K} = \mathbf{X}\mathbf{X}^\top \in \mathbb{R}^{n\times n}$.

(c) **Finally, prove that the optimal $\mathbf{w}^*$ can be computed using**

$$\mathbf{w}^* = \mathbf{X}^\top(\mathbf{K} + \lambda\mathbf{I})^{-1}\mathbf{y}. \tag{9}$$

[1] https://www.wikiwand.com/en/Minimax_theorem

Solution: By setting the gradient of the dual objective to zero, we get the equation

$$(\mathbf{K} + \lambda \mathbf{I})\boldsymbol{\alpha} - \lambda \boldsymbol{\alpha}^\top \mathbf{y} = 0. \tag{10}$$

for the minimizer $\boldsymbol{\alpha}$. Therefore the solution of the dual problem is

$$\boldsymbol{\alpha} = \lambda \left(\mathbf{K} + \lambda \mathbf{I}\right)^{-1} \mathbf{y}. \tag{11}$$

Using the relationship from Equation 5, we have

$$\mathbf{w}^* = \mathbf{X}^\top \left(\mathbf{K} + \lambda \mathbf{I}\right)^{-1} \mathbf{y}. \tag{12}$$

(d) (Polynomial Regression from a kernelized view) In this part, we will show that polynomial regression with a particular Tiknov regularization is the same as kernel ridge regression with a polynomial kernel for second-order polynomials. Recall that a degree 2 polynomial kernel function on $\mathbb{R}^d$ is defined as

$$K(\mathbf{x}_i, \mathbf{x}_j) = (1 + \mathbf{x}_i{}^\top \mathbf{x}_j)^2, \tag{13}$$

for any $\mathbf{x}_i, \mathbf{x}_j \in \mathbb{R}^\ell$. Given a dataset $(\mathbf{x}_i, y_i)$ for $i = 1, 2, \ldots, n$, **show the solution to kernel ridge regression is the same as the regularized least square solution to polynomial regression (with unweighted monomials as features) for $p = 2$ given the right choice of Tikhonov regularization for the polynomial regression.** That is, show for any new point $\mathbf{x}$ given in the prediction stage, both methods give the same prediction $\hat{y}$ with the same training data. **What is the Tikhonov regularization matrix here?**

Hint: You may or may not use the following matrix identity:

$$\mathbf{A}(a\mathbf{I}_d + \mathbf{A}^\top \mathbf{A})^{-1} = (a\mathbf{I} + \mathbf{A}\mathbf{A}^\top)^{-1}\mathbf{A}, \tag{14}$$

for any matrix $\mathbf{A} \in \mathbb{R}^{n \times d}$ and any positive real number a.

Solution: Define a vector-valued function from $\mathbb{R}^2 \to \mathbb{R}^6$ such that

$$\boldsymbol{\phi}(a) = (1, a_1^2, a_2^2, \sqrt{2}a_1, \sqrt{2}a_2, \sqrt{2}a_1 a_2)^\top$$

for $a = (a_1, a_2)^\top$.

Define a matrix in $\mathbb{R}^{n \times 6}$ such that

$$\boldsymbol{\Phi} = \begin{bmatrix} \boldsymbol{\phi}(x_1)^\top \\ \boldsymbol{\phi}(x_2)^\top \\ \vdots \\ \boldsymbol{\phi}(x_n)^\top \end{bmatrix} \tag{15}$$

We observe that $K(x, y) = \boldsymbol{\phi}(x)^\top \boldsymbol{\phi}(y)$. For a kernel matrix $\mathbf{K} \in \mathbb{R}^{n \times n}$ with $K_{ij} = K(x_i, x_j)$, we have

$$K_{ij} = \boldsymbol{\phi}(x_i)^\top \boldsymbol{\phi}(x_j) = (\boldsymbol{\Phi}\boldsymbol{\Phi}^\top)_{ij}. \tag{16}$$

That is

$$\mathbf{K} = \boldsymbol{\Phi}\boldsymbol{\Phi}^\top.$$

Recall the solution to Kernel ridge regression is a function f with

$$\begin{aligned} f(x) &= \sum_{i=1}^{N} \boldsymbol{\alpha}_i K(x_i, x) \\ &= \sum_{i=1}^{n} \boldsymbol{\alpha}_i \boldsymbol{\phi}(x_i)^\top \boldsymbol{\phi}(x) \\ &= \boldsymbol{\alpha}^\top \boldsymbol{\Phi} \boldsymbol{\phi}(x), \end{aligned}$$

where

$$\boldsymbol{\alpha} = (K + \lambda n \mathbf{I}_n)^{-1} \mathbf{y}. \tag{17}$$

Therefore, we can write $f(x)$ as

$$f(x) = \mathbf{y}^\top (\boldsymbol{\Phi}\boldsymbol{\Phi}^\top + \lambda n \mathbf{I}_n)^{-1} \boldsymbol{\Phi} \boldsymbol{\phi}(x). \tag{18}$$

For polynomial regression, define a vector-valued function from $\mathbb{R}^2 \to \mathbb{R}^6$ such that

$$\tilde{\boldsymbol{\phi}}(a) = (1, a_1^2, a_2^2, a_1, a_2, a_1 a_2)^\top$$

for $a = (a_1, a_2)^\top$.

Define a matrix in $\mathbb{R}^{n \times 6}$ such that

$$\tilde{\boldsymbol{\Phi}} = \begin{bmatrix} \tilde{\boldsymbol{\phi}}(x_1)^\top \\ \tilde{\boldsymbol{\phi}}(x_2)^\top \\ \vdots \\ \tilde{\boldsymbol{\phi}}(x_n)^\top \end{bmatrix} \tag{19}$$

Observe the relationship between $\boldsymbol{\phi}$ and $\tilde{\boldsymbol{\phi}}$: We have

$$\tilde{\boldsymbol{\phi}}(x) = \mathbf{D}\boldsymbol{\phi}(x), \tilde{\boldsymbol{\Phi}} = \boldsymbol{\Phi}\mathbf{D}, \tag{20}$$

for a diagonal matrix $\mathbf{D} \in \mathbb{R}^{6 \times 6}$, with

$$\mathbf{D} = \text{diag}(1, 1, 1, 1/\sqrt{2}, 1/\sqrt{2}, 1/\sqrt{2}).$$

A polynomial regression is nothing but replacing linear feature X by $\tilde{\boldsymbol{\phi}}(X) \in \mathbb{R}^6$ and add a Tikhonov regularization over the parameters $w \in \mathbb{R}^6$. Recall in a previous homework, we've shown it has a closed form solution

$$\mathbf{w} = (\tilde{\boldsymbol{\Phi}}^\top \tilde{\boldsymbol{\Phi}} + \boldsymbol{\Lambda})^{-1} \tilde{\boldsymbol{\Phi}}^\top Y, \tag{21}$$

for a polynomial regression with Tikhonov regularization matrix $\boldsymbol{\Lambda} \in \mathbb{R}^{d \times d}$. Let $\boldsymbol{\Lambda}$ be a diagonal matrix defined by

$$\boldsymbol{\Lambda} = \text{diag}(\lambda n, \lambda n, \lambda n, \lambda n/2, \lambda n/2, \lambda n/2) = \mathbf{D}(\lambda n \mathbf{I}_6)\mathbf{D}. \tag{22}$$

The predictor produced by Tikhonov regression is

$$\begin{aligned}
g(x) &= \mathbf{w}^\top \tilde{\boldsymbol{\phi}}(x) \\
&= [(\tilde{\mathbf{\Phi}}^\top \tilde{\mathbf{\Phi}} + \mathbf{\Lambda})^{-1} \tilde{\mathbf{\Phi}}^\top \mathbf{y}]^\top \tilde{\boldsymbol{\phi}}(x) \\
&= \mathbf{y}^\top \tilde{\mathbf{\Phi}} (\tilde{\mathbf{\Phi}}^\top \tilde{\mathbf{\Phi}} + \mathbf{\Lambda})^{-1} \tilde{\boldsymbol{\phi}}(x) \\
&= \mathbf{y}^\top \mathbf{\Phi} \mathbf{D} (\mathbf{D} \mathbf{\Phi}^\top \mathbf{\Phi} \mathbf{D} + \mathbf{D}(\mathbf{D}^{-1} \mathbf{\Lambda} \mathbf{D}^{-1}) \mathbf{D})^{-1} \mathbf{D} \boldsymbol{\phi}(x) \\
&= \mathbf{y}^\top \mathbf{\Phi} \mathbf{D} \mathbf{D}^{-1} (\mathbf{\Phi}^\top \mathbf{\Phi} + (\mathbf{D}^{-1} \mathbf{\Lambda} \mathbf{D}^{-1}))^{-1} \mathbf{D}^{-1} \mathbf{D} \boldsymbol{\phi}(x) \\
&= \mathbf{y}^\top \mathbf{\Phi} (\mathbf{\Phi}^\top \mathbf{\Phi} + (\mathbf{D}^{-1} \mathbf{\Lambda} \mathbf{D}^{-1}))^{-1} \boldsymbol{\phi}(x) \\
&= \mathbf{y}^\top \mathbf{\Phi} (\mathbf{\Phi}^\top \mathbf{\Phi} + \lambda n \mathbf{I}_6)^{-1} \boldsymbol{\phi}(x) \\
&= \mathbf{y}^\top (\mathbf{\Phi} \mathbf{\Phi}^\top + \lambda n \mathbf{I}_n)^{-1} \mathbf{\Phi} \boldsymbol{\phi}(x) = f(x),
\end{aligned}$$

where the last equation follows from the hint. Hence, we have shown the equivalence between the two predictors.

(e) In general, for any polynomial regression with pth order polynomial on $\mathbb{R}^\ell$ with an appropriately specified Tikhonov regression, we can show the equivalence between it and kernel ridge regression with a polynomial kernel of order p. **Comment on the computational complexity of doing least squares for polynomial regression with this Tikhonov regression directly and that of doing kernel ridge regression in the training stage.** (That is, the complexity of finding $\boldsymbol{\alpha}$ and finding $\mathbf{w}$.) **Compare with the computational complexity of actually doing prediction as well.**

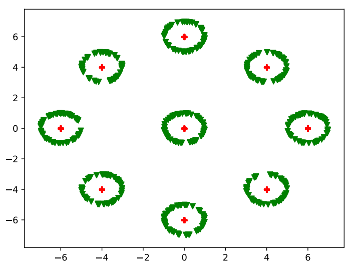
Solution: In the polynomial regression with Tikhonov regularization, for any data point (x_i, y_i), computing its polynomial features of order p takes $O(d^p)$. The complexity of solving least square is $O(d^{3p} + d^p n)$. The total complexity is $O(d^{3p} + d^p n)$.

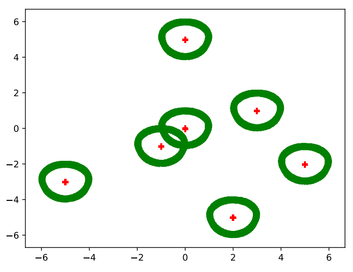
In the kernel ridge regression, the complexity of computing the kernel matrix is $O(n^2 d \log p)$. The complexity of getting $\boldsymbol{\alpha}$ after that is $O(n^3)$. The total complexity is $O(n^3 + n^2 d \log p)$. It only has a log dependence on p but cubic dependence on n. Kernel ridge regression is preferred when p is large.

3 Kernel Ridge Regression: Practice

In the following problem, you will implement Polynomial Ridge Regression and its kernel variant Kernel Ridge Regression, and compare them with each other. You will be dealing with a 2D regression problem, i.e., $\mathbf{x}_i \in \mathbb{R}^2$. We give you three datasets, `circle.npz` (small dataset), `heart.npz` (medium dataset), and `asymmetric.npz` (large dataset). In this problem, we choose $y_i \in \{-1, +1\}$, so you may view this question as a classification problem. Later on in the course we will learn about logistic regression and SVMs, which can solve classification problems much better and can also leverage kernels.

(a) **Use `matplotlib.pyplot` to visualize all the datasets and attach the plots to your report**. Label the points with different y values with different colors and/or shapes. You are only allow to use `numpy.*` and `numpy.linalg.*` in the following questions.


(a) `circle.npz`


(b) `asymmetric.npz`

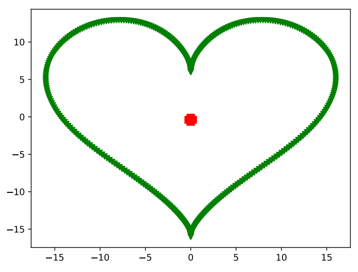

(c) `heart.npz`

Figure 1: Dataset visualization

Solution:

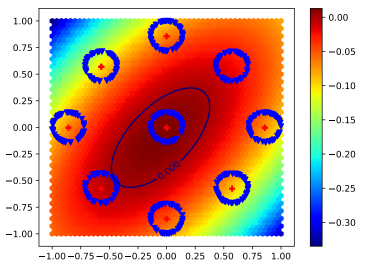
See Figure 1.

(b) **Implement polynomial ridge regression** (non-kernelized version that you should already have implemented in your previous homework) **to fit the datasets `circle.npz`, `asymmetric.npy`, and `heart.npz`**. Use the first 80% data as the training dataset and the last 20% data as the validation dataset. **Report both the average training squared loss and the average validation squared for polynomial order** $p \in \{1, \dots, 16\}$. Use the regularization term $\lambda = 0.001$ for all p. **Visualize your result and attach the heatmap plots for the learned predictions over the entire 2D domain for** $p \in \{2, 4, 6, 8, 10, 12\}$ **in your report.** You can start with the code from previous homeworks.

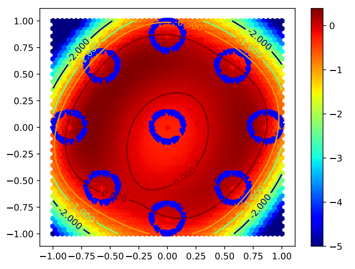
Solution:

See Figure 2, 3, and 4. The error can be found in next part. If you directly use the code from homework 2, **you may find that your result is slightly different from the error here due to the *difference of the constant terms* used in the polynomial**, e.g., feature x_1x_2 vs feature $2x_1x_2$, but your plot should be similar.

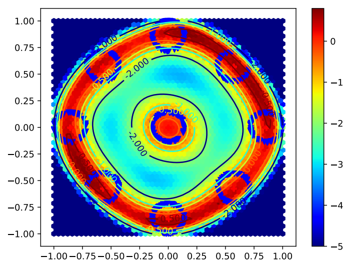
```
#!/usr/bin/env python3

import matplotlib.pyplot as plt
import numpy as np
from matplotlib import cm

# data = np.load('circle.npz')
# data = np.load('heart.npz')
data = np.load('asymmetric.npz')

SPLIT = 0.8
X = data["x"]
y = data["y"]
X /= np.max(X)  # normalize the data

n_train = int(X.shape[0] * SPLIT)
X_train = X[:n_train:, :]
X_valid = X[n_train:, :]
y_train = y[:n_train]
```



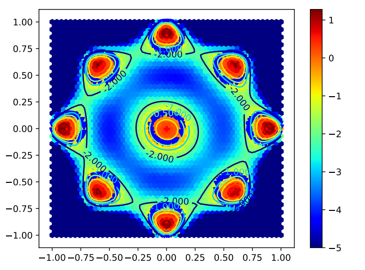
(a) $p = 2$



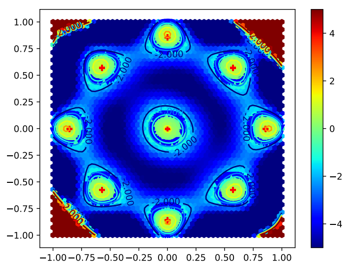
(b) $p = 4$



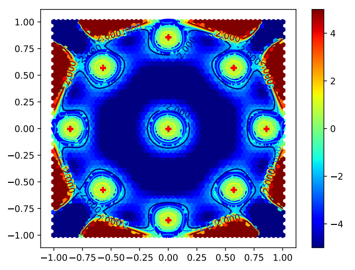
(c) $p = 6$



(d) $p = 8$

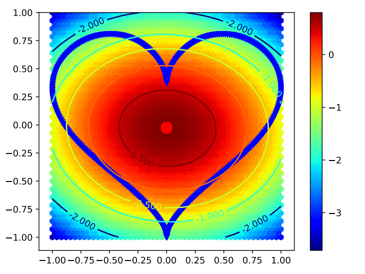


(e) $p = 10$

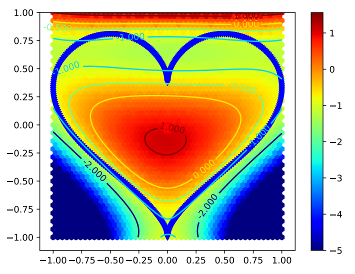


(f) $p = 12$

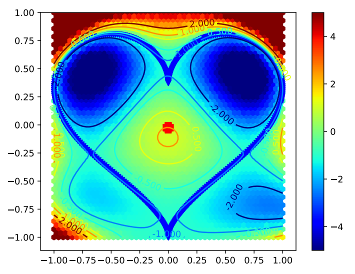
Figure 2: Heat map of circle.npz



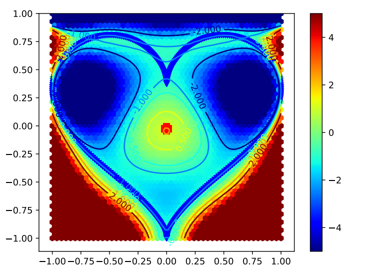
(a) $p = 2$



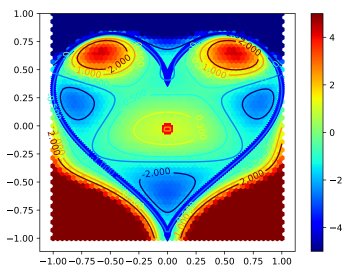
(b) $p = 4$



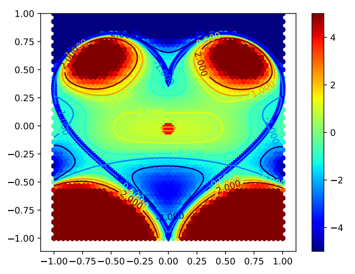
(c) $p = 6$



(d) $p = 8$

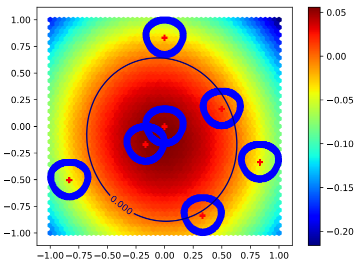


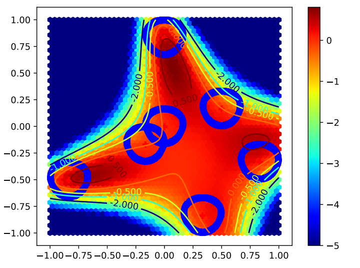
(e) $p = 10$

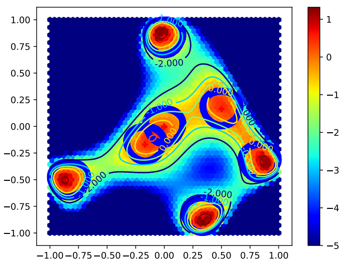


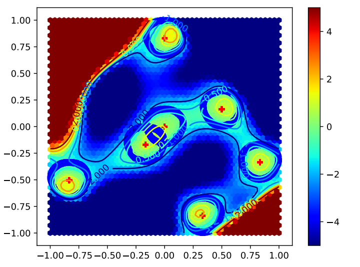
(f) $p = 12$

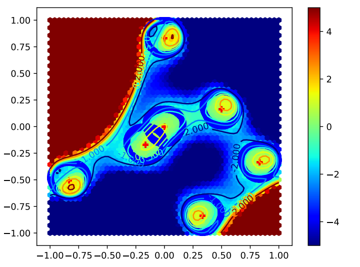
Figure 3: Heat map of heart.npz


(a) $p = 2$


(b) $p = 4$


(c) $p = 6$


(d) $p = 8$


(e) $p = 10$

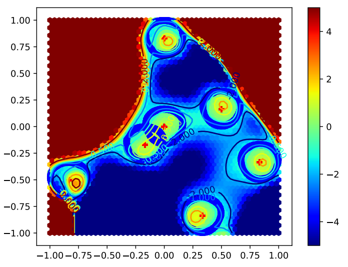

(f) $p = 12$

Figure 4: Heat map of asymmetric.npz

```
y_valid = y[n_train:]

LAMBDA = 0.001


def lstsq(A, b, lambda_=0):
    return np.linalg.solve(A.T @ A + lambda_ * np.eye(A.shape[1]), A.T @ b)


def heatmap(f, fname=False, clip=5):
    # example: heatmap(lambda x, y: x * x + y * y)
    # clip: clip the function range to [-clip, clip] to generate a clean plot
    #   set it to zero to disable this function

    xx0 = xx1 = np.linspace(np.min(X), np.max(X), 72)
    x0, x1 = np.meshgrid(xx0, xx1)
    x0, x1 = x0.ravel(), x1.ravel()
    z0 = f(x0, x1)

    if clip:
        z0[z0 > clip] = clip
        z0[z0 < -clip] = -clip

    plt.hexbin(x0, x1, C=z0, gridsize=50, cmap=cm.jet, bins=None)
    plt.colorbar()
    cs = plt.contour(
        xx0, xx1, z0.reshape(xx0.size, xx1.size), [-2, -1, -0.5, 0, 0.5, 1, 2], cmap=cm.jet)
    plt.clabel(cs, inline=1, fontsize=10)

    pos = y[:] == +1.0
    neg = y[:] == -1.0
    plt.scatter(X[pos, 0], X[pos, 1], c='red', marker='+')
    plt.scatter(X[neg, 0], X[neg, 1], c='blue', marker='v')
    if fname:
        plt.savefig(fname)
    plt.show()


def assemble_feature(x, D):
    from scipy.special import binom
    xs = []
    for d0 in range(D + 1):
        for d1 in range(D - d0 + 1):
            # non-kernel polynomial feature
            # xs.append((x[:, 0]**d0) * (x[:, 1]**d1))
            # # kernel polynomial feature
            xs.append((x[:, 0]**d0) * (x[:, 1]**d1) * np.sqrt(binom(D, d0) * binom(D - d0, d1
    ↪ )))
    return np.column_stack(xs)


def main():
    for D in range(1, 17):
        Xd_train = assemble_feature(X_train, D)
        Xd_valid = assemble_feature(X_valid, D)
        w = lstsq(Xd_train, y_train, LAMBDA)
        error_train = np.average(np.square(y_train - Xd_train @ w))
        error_valid = np.average(np.square(y_valid - Xd_valid @ w))
        print("p = {:2d}   train_error = {:10.6f}  validation_error = {:10.6f}  cond = {:14.6
    ↪ f}".
              format(D, error_train, error_valid,
                     np.linalg.cond(Xd_valid.T @ Xd_valid + np.eye(Xd_valid.shape[1]))))
        if D in [2, 4, 6, 8, 10, 12]:
            fname = "result/asym%02d.pdf" % D
            heatmap(lambda x, y: assemble_feature(np.vstack([x, y]).T, D) @ w, fname)


if __name__ == "__main__":
    main()
```

(c) **Implement kernel ridge regression to fit the datasets `circle.npz`, `heart.npz`, and optionally (due to the computational requirements), `asymmetric.npz`.** Use the polynomial kernel $K(\mathbf{x}_i, \mathbf{x}_j) = (1+\mathbf{x}_i^\top \mathbf{x}_j)^p$. Use the first 80% data as the training dataset and the last 20% data as the validation dataset. **Report both the average training squared loss and the average validation squared loss for polynomial order $p \in \{1, \ldots, 16\}$.** Use the regularization term $\lambda = 0.001$ for all p. The sample code for generating heatmap plot is included in the start kit. **For `circle.npz`, also report the average training squared loss and validation squared loss for polynomial order $p \in \{1, \ldots, 24\}$ when you use only the first 15% data as the training dataset and the rest 85% data as the validation dataset.** Based on the error, **comment on when you want to use a high-order polynomial in linear/ridge regression.**

Solution:
You can see that when you training data is not enough, i.e., in the case when you only use 15% of the training data, you can easily overfit your training data if you use a high-order polynomial. When you have enough training data, i.e., in the case you are using the 80% of the training data, the overfitting is more unlikely. Therefore, you want to use a high-order polynomial only when you have enough training data to avoid the overfitting problem. The average error here is

```
########## 80% Training Data ################
####### circle.npz #######
p =  1   train_error =   0.997088  validation_error =   0.997579  cond =       3.885463
p =  2   train_error =   0.995537  validation_error =   1.001056  cond =      40.439621
p =  3   train_error =   0.992699  validation_error =   1.019356  cond =     230.817918
p =  4   train_error =   0.943011  validation_error =   0.997941  cond =     437.187915
p =  5   train_error =   0.935539  validation_error =   1.029308  cond =     804.009794
```

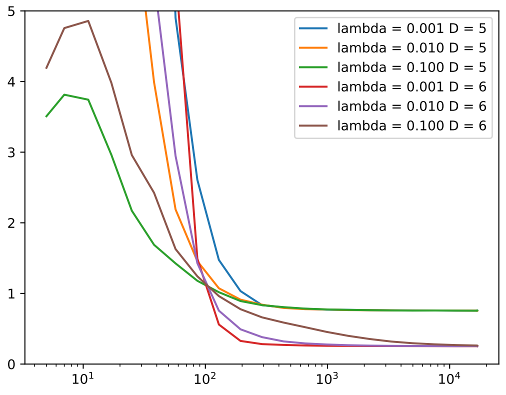
```
p =  6   train_error =   0.511241  validation_error =   0.547531  cond =     1307.933645
p =  7   train_error =   0.507592  validation_error =   0.549927  cond =     2159.011214
p =  8   train_error =   0.086389  validation_error =   0.101056  cond =     3630.740079
p =  9   train_error =   0.081809  validation_error =   0.097989  cond =     6230.776776
p = 10   train_error =   0.043086  validation_error =   0.054167  cond =    10920.048093
p = 11   train_error =   0.013966  validation_error =   0.018290  cond =    19529.648519
p = 12   train_error =   0.008685  validation_error =   0.011348  cond =    35549.340362
p = 13   train_error =   0.006517  validation_error =   0.008556  cond =    65983.294010
p = 14   train_error =   0.003665  validation_error =   0.004821  cond =   123976.972506
p = 15   train_error =   0.001912  validation_error =   0.002475  cond =   234627.222155
p = 16   train_error =   0.001400  validation_error =   0.001797  cond =   446625.921685
###### heart.npz ######
p =  1   train_error =   0.962643  validation_error =   0.959952  cond =        6.646302
p =  2   train_error =   0.236718  validation_error =   0.189837  cond =       26.941658
p =  3   train_error =   0.115481  validation_error =   0.090813  cond =      217.010014
p =  4   train_error =   0.012163  validation_error =   0.009089  cond =      348.834425
p =  5   train_error =   0.003759  validation_error =   0.002975  cond =      638.648596
p =  6   train_error =   0.002294  validation_error =   0.001613  cond =     1262.823064
p =  7   train_error =   0.001441  validation_error =   0.001056  cond =     2554.245128
p =  8   train_error =   0.000665  validation_error =   0.000428  cond =     5222.932534
p =  9   train_error =   0.000305  validation_error =   0.000202  cond =    10754.752173
p = 10   train_error =   0.000189  validation_error =   0.000138  cond =    22259.613418
p = 11   train_error =   0.000139  validation_error =   0.000114  cond =    46259.310324
p = 12   train_error =   0.000111  validation_error =   0.000097  cond =    96458.107873
p = 13   train_error =   0.000093  validation_error =   0.000084  cond =   201706.212544
p = 14   train_error =   0.000081  validation_error =   0.000075  cond =   422842.117216
p = 15   train_error =   0.000072  validation_error =   0.000068  cond =   888359.857996
p = 16   train_error =   0.000064  validation_error =   0.000062  cond =  1870033.835947
###### asymmetric.npz ######
p =  1   train_error =   0.999989  validation_error =   1.000194  cond =        4.303603
p =  2   train_error =   0.998260  validation_error =   1.000176  cond =       82.880736
p =  3   train_error =   0.991565  validation_error =   0.991388  cond =      559.928514
p =  4   train_error =   0.828692  validation_error =   0.822373  cond =     4924.555570
p =  5   train_error =   0.758986  validation_error =   0.748816  cond =    15783.658385
p =  6   train_error =   0.263368  validation_error =   0.241398  cond =    36482.622481
p =  7   train_error =   0.218690  validation_error =   0.195606  cond =    73065.066532
p =  8   train_error =   0.140721  validation_error =   0.120891  cond =   148442.373823
p =  9   train_error =   0.120781  validation_error =   0.102239  cond =   303228.309085
p = 10   train_error =   0.109520  validation_error =   0.092603  cond =   623400.268355
p = 11   train_error =   0.095645  validation_error =   0.081190  cond =  1289425.566871
p = 12   train_error =   0.083126  validation_error =   0.070826  cond =  2682742.562813
p = 13   train_error =   0.069519  validation_error =   0.059635  cond =  5613779.945180
p = 14   train_error =   0.052339  validation_error =   0.044942  cond =  11813079.998338
p = 15   train_error =   0.037785  validation_error =   0.032575  cond =  24993651.532068
p = 16   train_error =   0.029511  validation_error =   0.025690  cond =  53158174.199813
######### Just using 15% Training Data ################
####### circle.npz #######
p =  1   train_error = 0.977122  validation_error = 1.017212  cond =  154347.326799
p =  2   train_error = 0.965179  validation_error = 1.040716  cond =  188799.151210
p =  3   train_error = 0.935814  validation_error = 1.083452  cond =  260636.616808
p =  4   train_error = 0.828087  validation_error = 1.220925  cond =  388234.123476
p =  5   train_error = 0.808276  validation_error = 1.294004  cond =  605958.721676
p =  6   train_error = 0.465600  validation_error = 0.731820  cond =  974938.119166
p =  7   train_error = 0.418462  validation_error = 0.701896  cond = 1604147.948302
p =  8   train_error = 0.094915  validation_error = 0.326256  cond = 2690114.807338
p =  9   train_error = 0.064552  validation_error = 0.979804  cond = 4592713.085243
p = 10   train_error = 0.054649  validation_error = 2.273410  cond = 7981356.922646
p = 11   train_error = 0.036871  validation_error = 3.763307  cond = 14136597.558594
p = 12   train_error = 0.019774  validation_error = 1.865602  cond = 26239673.362870
p = 13   train_error = 0.009580  validation_error = 0.104549  cond = 49619782.252457
p = 14   train_error = 0.005777  validation_error = 0.372263  cond = 94594909.390382
p = 15   train_error = 0.004199  validation_error = 0.544182  cond = 181457265.287672
p = 16   train_error = 0.002995  validation_error = 0.436762  cond = 349803221.168144
p = 17   train_error = 0.001924  validation_error = 0.705161  cond = 677043148.807441
p = 18   train_error = 0.001210  validation_error = 1.518994  cond = 1314776445.035100
p = 19   train_error = 0.000851  validation_error = 3.576013  cond = 2560349372.861672
p = 20   train_error = 0.000678  validation_error = 7.938049  cond = 4997765669.676615
p = 21   train_error = 0.000571  validation_error = 16.370187  cond = 9775415811.240183
p = 22   train_error = 0.000483  validation_error = 32.763564  cond = 19153899435.104542
p = 23   train_error = 0.000405  validation_error = 62.110989  cond = 37587428504.160706
p = 24   train_error = 0.000344  validation_error = 103.845313  cond = 73859595026.545380
```

```python
#!/usr/bin/env python3

import matplotlib.pyplot as plt
import numpy as np
import scipy.special
from matplotlib import cm

# data = np.load('circle.npz')
data = np.load('heart.npz')
# data = np.load('asymmetric.npz')

SPLIT = 0.80
X = data["x"]
y = data["y"]
```

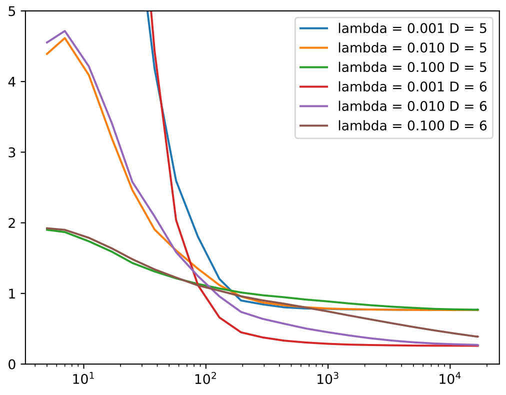
```
X /= np.max(X)  # normalize the data

n_train = int(X.shape[0] * SPLIT)
X_train = X[:n_train:, :]
X_valid = X[n_train:, :]
y_train = y[:n_train]
y_valid = y[n_train:]

LAMBDA = 0.001


def poly_kernel(X, XT, D):
    return np.power(X @ XT + 1, D)


def rbf_kernel(X, XT, sigma):
    XXT = -2 * X @ XT
    XXT += np.sum(X * X, axis=1, keepdims=True)
    XXT += np.sum(XT * XT, axis=0, keepdims=True)
    return np.exp(-XXT / (2 * sigma * sigma))


def heatmap(f, fname=False, clip=5):
    # example: heatmap(lambda x, y: x * x + y * y)
    # clip: clip the function range to [-clip, clip] to generate a clean plot
    #   set it to zero to disable this function

    xx0 = xx1 = np.linspace(np.min(X), np.max(X), 72)
    x0, x1 = np.meshgrid(xx0, xx1)
    x0, x1 = x0.ravel(), x1.ravel()
    z0 = f(x0, x1)

    if clip:
        z0[z0 > clip] = clip
        z0[z0 < -clip] = -clip

    plt.hexbin(x0, x1, C=z0, gridsize=50, cmap=cm.jet, bins=None)
    plt.colorbar()
    cs = plt.contour(
        xx0, xx1, z0.reshape(xx0.size, xx1.size), [-2, -1, -0.5, 0, 0.5, 1, 2], cmap=cm.jet)
    plt.clabel(cs, inline=1, fontsize=10)

    pos = y[:] == +1.0
    neg = y[:] == -1.0
    plt.scatter(X[pos, 0], X[pos, 1], c='red', marker='+')
    plt.scatter(X[neg, 0], X[neg, 1], c='blue', marker='v')
    if fname:
        plt.savefig(fname)
    plt.show()


def main():
    for D in range(1, 16):
        # polynomial kernel
        K = poly_kernel(X_train, X_train.T, D) + LAMBDA * np.eye(X_train.shape[0])
        coeff = np.linalg.solve(K, y_train)
        error_train = np.average(np.square(y_train - poly_kernel(X_train, X_train.T, D) @
    ↪ coeff))
        error_valid = np.average(np.square(y_valid - poly_kernel(X_valid, X_train.T, D) @
    ↪ coeff))
        print("p = {:2d}   train_error = {:7.6f}  validation_error = {:7.6f}  cond = {:14.6f}
    ↪ ".
              format(D, error_train, error_valid, np.linalg.cond(K)))
        # heatmap(lambda x, y: poly_kernel(np.column_stack([x, y]), X_train.T, D) @ coeff)
        # if D in [2, 4, 6, 8, 10, 12]:
        #     fname = "result/poly%02d.pdf" % D
        #     heatmap(lambda x, y: poly_kernel(np.column_stack([x, y]), X_train.T, D) @ coeff
    ↪ , fname)

    for sigma in [10, 3, 1, 0.3, 0.1, 0.03]:
        K = rbf_kernel(X_train, X_train.T, sigma) + LAMBDA * np.eye(X_train.shape[0])
        coeff = np.linalg.solve(K, y_train)
        error_train = np.average(
            np.square(y_train - rbf_kernel(X_train, X_train.T, sigma) @ coeff))
        error_valid = np.average(
            np.square(y_valid - rbf_kernel(X_valid, X_train.T, sigma) @ coeff))
        print("sigma = {:6.3f} train_error = {:7.6f} validation_error = {:7.6f} cond = {:14.6
    ↪ f}".
              format(sigma, error_train, error_valid, np.linalg.cond(K)))
        heatmap(
            lambda x, y: rbf_kernel(np.column_stack([x, y]), X_train.T, sigma) @ coeff,
            fname="heart_RBF0_%4f.pdf" % sigma)


if __name__ == "__main__":
    main()
```

(d) (Diminishing influence of the prior with growing amount of data) With increasing of amount of data, the prior (from the statistical view) and regularization (from the optimization view) will be washed away and become less and less important. Sample the training data from the first 80% data from `asymmetric.npz` and use the data from the last 20% data for validation. **Make a plot whose x axis is the amount of the training data and y axis is the validation squared loss of the non-kernelized ridge regression algorithm. Optionally, repeat the same for kernel ridge regression.** Include 6 curves for hyper-parameters $\lambda \in \{0.0001, 0.001, 0.01\}$ and $p = \{5, 6\}$. Your plot should demonstrate that with same p, the validation squared loss will converge with enough data, regardless of the choice of λ and/or the regularizer matrix. You can use log plot on x axis for clearity and you need to resample the data multiple times for the given p, λ, and the amount of training data in order to get a smooth curve.

Solution: See Figure 5.



(a) Kernel Ridge Regression



(b) Non-Kernel Ridge Regression

Figure 5: Plot for Diminishing Influence of the Prior

```python
#!/usr/bin/env python3

import matplotlib.pyplot as plt
import numpy as np
from matplotlib import cm
from scipy import interpolate

# data = np.load('circle.npz')
# data = np.load('heart.npz')
data = np.load('asymmetric.npz')

SPLIT = 0.8
X = data["x"]
y = data["y"]
X /= np.max(X)  # normalize the data

index = np.arange(X.shape[0])
np.random.shuffle(index)
X = X[index, :]
y = y[index]

n_train = int(X.shape[0] * SPLIT)
X_train = X[:n_train:, :]
X_valid = X[n_train:, :]
y_train = y[:n_train]
y_valid = y[n_train:]

LAMBDA = 0.001


def lstsq(A, b, lambda_=0):
    return np.linalg.solve(A.T @ A + lambda_ * np.eye(A.shape[1]), A.T @ b)


def heatmap(f, fname=False, clip=True):
    # example: heatmap(lambda x, y: x * x + y * y)
    xx = yy = np.linspace(np.min(X), np.max(X), 72)
    x0, y0 = np.meshgrid(xx, yy)
    x0, y0 = x0.ravel(), y0.ravel()
    z0 = f(x0, y0)

    if clip:
        z0[z0 > 5] = 5
        z0[z0 < -5] = -5

    plt.hexbin(x0, y0, C=z0, gridsize=50, cmap=cm.jet, bins=None)
    plt.colorbar()
    cs = plt.contour(
        xx, yy, z0.reshape(xx.size, yy.size), [-2, -1, -0.5, 0, 0.5, 1, 2], cmap=cm.jet)
    plt.clabel(cs, inline=1, fontsize=10)

    pos = y[:] == +1.0
    neg = y[:] == -1.0
    plt.scatter(X[pos, 0], X[pos, 1], c='red', marker='+')
    plt.scatter(X[neg, 0], X[neg, 1], c='blue', marker='v')
    if fname:
        plt.savefig(fname)
    plt.show()


def assemble_feature(x, D):
    from scipy.special import binom
    xs = []
    for d0 in range(D + 1):
        for d1 in range(D - d0 + 1):
            # non-kernel polynomial feature
            xs.append((x[:, 0]**d0) * (x[:, 1]**d1))
            # kernel polynomial feature
            # xs.append((x[:, 0]**d0) * (x[:, 1]**d1) * np.sqrt(binom(D, d0) * binom(D - d0,
    ↪ d1)))
    return np.column_stack(xs)


def main():
    plt.xscale('log')
    LAMBDA = 0.01
    for D in [5, 6]:
        Xd_train = assemble_feature(X_train, D)
        Xd_valid = assemble_feature(X_valid, D)
        for LAMBDA in [0.001, 0.01, 0.1]:
            print(LAMBDA)
            pltx = [int(1.5**x) for x in range(4, 300) if 1.5**x < n_train] + [n_train]
            plty = []
            for n_sampl in pltx:
                error = []
                time = max(int(40000 / n_sampl), 1)
                for ttt in range(time):
                    idx = np.random.randint(n_train, size=n_sampl)
                    Xd_sampl = Xd_train[idx, :]
                    y_sampl = y_train[idx]
                    w = lstsq(Xd_sampl, y_sampl, LAMBDA)
                    error_valid = np.average(np.square(y_valid - Xd_valid @ w))
                    error.append(error_valid)
                plty.append(np.average(error))
            plt.plot(pltx, plty, label="lambda = %.3f D = %d" % (LAMBDA, D))
    plt.ylim([0, 5])
    plt.legend()
    plt.show()


if __name__ == "__main__":
    main()
```

(e) A popular kernel function that is widely used in various kernelized learning algorithms is called the radial basis function kernel (RBF kernel). It is defined as

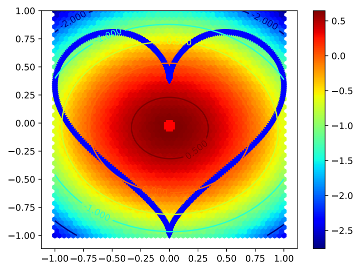
$$K(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}'\|_2^2}{2\sigma^2}\right). \tag{23}$$

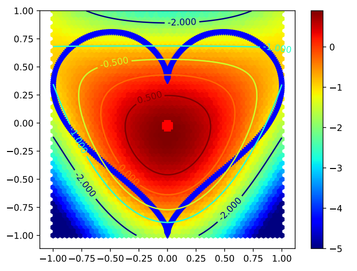
Implement the RBF kernel function for kernel ridge regression to fit the dataset `heart.npz`. Use the regularization term $\lambda = 0.001$. **Report the average squared loss, visualize your result and attach the heatmap plots for the fitted functions over the 2D domain for $\sigma \in \{10, 3, 1, 0.3, 0.1, 0.03\}$ in your report.** You may want to vectorize your kernel functions to speed up your implementation. **Comment on the effect of σ.**

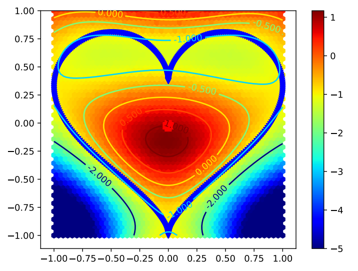
Solution:
The average fitting error is

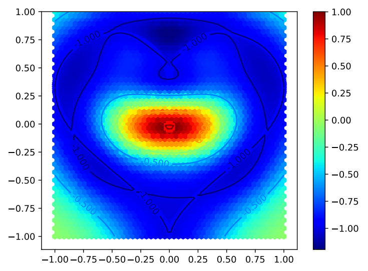
```
sigma = 10.000 train_error = 0.279653 validation_error = 0.224638 cond =  800690.695468
sigma =  3.000 train_error = 0.119629 validation_error = 0.082379 cond =  778537.061196
sigma =  1.000 train_error = 0.005872 validation_error = 0.004201 cond =  648473.876828
sigma =  0.300 train_error = 0.000053 validation_error = 0.000050 cond =  469873.484420
sigma =  0.100 train_error = 0.000000 validation_error = 0.000000 cond =  442247.855472
sigma =  0.030 train_error = 0.000000 validation_error = 0.000078 cond =  291224.335632
```

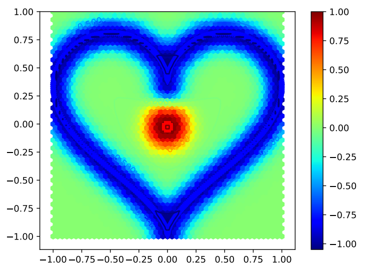
The heat map can be found in Figure 6 for $\sigma \in \{10, 3, 1, 0.3, 0.1, 0.03\}$. As we see, the larger σ, the more data the kernel averages over and the more blurry the image of the heatmap gets. The previous code from kernel regression includes the implementation of RBF kernel.


(a) $\sigma = 10$


(b) $\sigma = 3$


(c) $\sigma = 1$


(d) $\sigma = 0.3$


(e) $\sigma = 0.1$

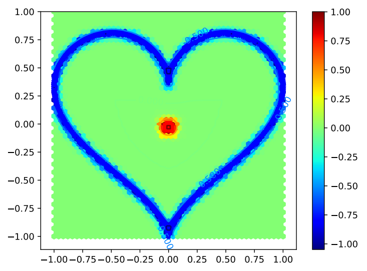

(f) $\sigma = 0.03$

Figure 6: Heatmap of `heart.npz`

(f) For polynomial ridge regression, **which of your implementation is more efficient, the kernelized one or the non-kernelized one?** For RBF kernel, **explain whether it is possible to implement it in the non-kernelized ridge regression. Summarize when you prefer the kernelized to the non-kernelized ridge regression.**

Solution: For polynomial regression, our non-kernelized method is more efficient in practice. This matches our prediction, where the computational complexity of non-kernelized method is smaller when $d \ll n$. However, this problem is in 2D space. When the feature space is in $\mathbb{R}^{20}$ for example, using a $d = 10$ polynomial can be impractical for non-kernelized method.

In summary, we want to use kernelized method when the feature space is much larger than the number pf samples, or when the dimension of the feature space is infinite like in RBF, in which it is impossible to use the non-kernelized method.

(g) Disable the `clip` option in the provided `heatmap` function and redraw the heatmap plots for the functions learned by the polynomial kernel and RBF kernel. Experiment on the provided datasets and **describe one potential problem of the polynomial kernel related to what you see here.** Does the RBF kernel have such problem? **Compute, compare, comment, and attach the heatmap plots of the polynomial kernel and the RBF kernel on `heart.npz` dataset.**

Solution: Figure 7 shows the comparison between polynomial kernel (left) and RBF kernel (right) with similar fitting error. One most observable problem is that the polynomial kernel does not extrapolate well, i.e., you will see absurdly large number outside the domain of training data. On the other hand, RBF kernel extrapolates much nicely compared to the polynomial kernel.

4 Ridge regression vs. PCA

Assume we are given n training data points $(\mathbf{x}_i, y_i)$. We collect the target values into $\mathbf{y} \in \mathbb{R}^n$, and the inputs into the matrix $\mathbf{X} \in \mathbb{R}^{n\times d}$ where the rows are the $d-$dimensional feature vectors $\mathbf{x}_i^\top$ corresponding to each training point. Furthermore, assume that $\frac{1}{n}\sum_{i=1}^n \mathbf{x_i} = \mathbf{0}$, $n > d$ and $\mathbf{X}$ has rank d.

In this problem we want to compare two procedures: The first is ridge regression with hyperparameter λ, while the second is applying ordinary least squares after using PCA to reduce the feature dimension from d to k (we give this latter approach the short-hand name k-PCA-OLS where k is the hyperparameter).

Notation: The singular value decomposition of $\mathbf{X}$ reads $\mathbf{X} = \mathbf{U\Sigma V}^\top$ where $\mathbf{U} \in \mathbb{R}^{n\times n}$, $\mathbf{\Sigma} \in \mathbb{R}^{n\times d}$ and $\mathbf{V} \in \mathbb{R}^{d\times d}$. We denote by $\mathbf{u}_i$ the n-dimensional column vectors of $\mathbf{U}$ and by $\mathbf{v}_i$ the $d-$dimensional column vectors of $\mathbf{V}$. Furthermore the diagonal entries $\sigma_i = \Sigma_{i,i}$ of $\mathbf{\Sigma}$ satisfy $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_d > 0$. For notational convenience, assume that $\sigma_i = 0$ for $i > d$.

(a) It turns out that the ridge regression optimizer (with $\lambda > 0$) in the $\mathbf{V}$-transformed coordinates

$$\widehat{\mathbf{w}}_{\text{ridge}} = \arg\min_{\mathbf{w}} \|\mathbf{XVw} - \mathbf{y}\|_2^2 + \lambda\|\mathbf{w}\|_2^2$$

has the following expression:

$$\widehat{\mathbf{w}}_{\text{ridge}} = \text{diag}(\frac{\sigma_i}{\lambda + \sigma_i^2})\mathbf{U}^\top\mathbf{y}. \tag{24}$$

Use $\widehat{y}_{test} = \mathbf{x}_{test}^\top\mathbf{V}\widehat{\mathbf{w}}_{\text{ridge}}$ to denote the resulting prediction for a hypothetical $\mathbf{x}_{test}$. Using (24) and the appropriate scalar $\{\beta_i\}$, this can be written as:

$$\widehat{y}_{test} = \mathbf{x}_{test}^\top \sum_{i=1}^d \mathbf{v}_i\beta_i\mathbf{u}_i^\top\mathbf{y}. \tag{25}$$

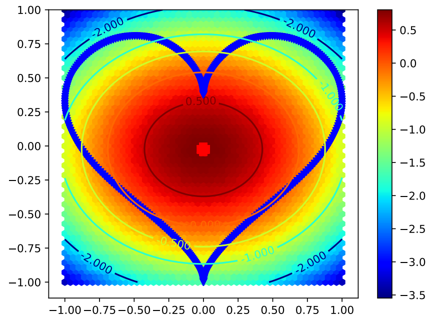
What are the $\beta_i \in \mathbb{R}$ for this to correspond to (24) from ridge regression?

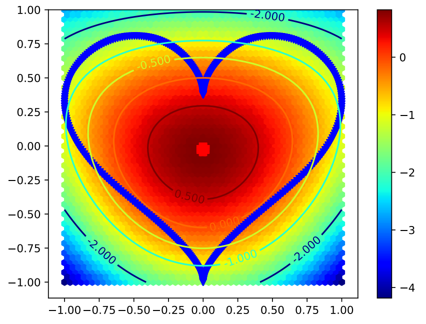
Solution:

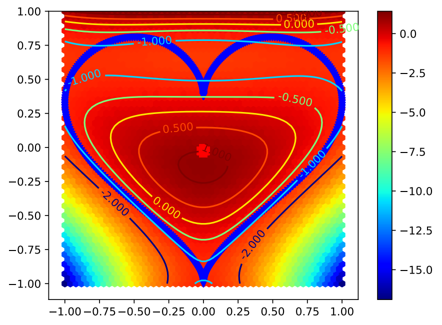
The resulting prediction for ridge reads

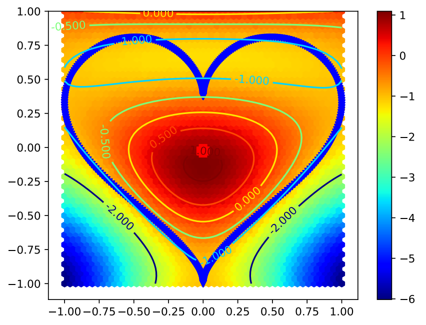
$$\hat{\mathbf{y}}_{\text{ridge}} = \mathbf{x}^\top\mathbf{V}\,\text{diag}\,(\frac{\sigma_i}{\lambda+\sigma_i^2})\mathbf{U}^\top\mathbf{y}$$
$$= \mathbf{x}^\top\sum_{i=1}^d \frac{\sigma_i}{\lambda+\sigma_i^2}\mathbf{v}_i\mathbf{u}_i^\top\mathbf{y}$$

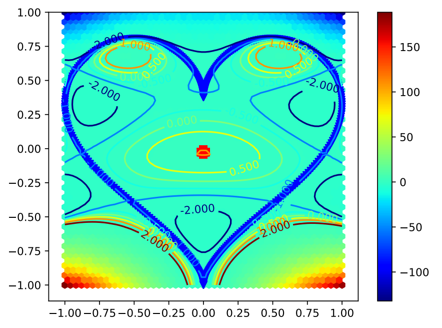
Therefore we have $\beta_i = \frac{\sigma_i}{\lambda+\sigma_i^2}$ for $i = 1, \ldots, d$.


(a) $p = 2$


(b) $\sigma = 3$


(c) $p = 4$


(d) $\sigma = 1$


(e) $p = 12$

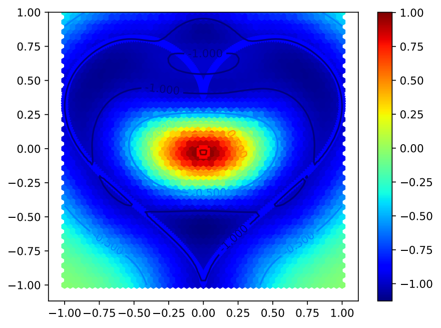

(f) $\sigma = 0.3$

Figure 7: Heatmap of `heart.npz` when the fitting error is similar between polynomial kernel and RBF kernel, with $p = 2, 4, 12$ (left) and $\sigma = 3, 1, 0.3$ (right). Notice the range on the right colorbar.

(b) Suppose that we do k-PCA-OLS — i.e. ordinary least squares on the reduced k-dimensional feature space obtained by projecting the raw feature vectors onto the $k < d$ principal components of the covariance matrix $\mathbf{X}^\top\mathbf{X}$. Use $\widehat{y}_{test}$ to denote the resulting prediction for a hypothetical $\mathbf{x}_{test}$,

It turns out that the learned k-PCA-OLS predictor can be written as:

$$\widehat{y}_{test} = \mathbf{x}_{test}^\top \sum_{i=1}^{d} \mathbf{v}_i \beta_i \mathbf{u}_i^\top \mathbf{y}. \tag{26}$$

Give the $\beta_i \in \mathbb{R}$ coefficients for k-PCA-OLS. Show work.

Hint 1: some of these β_i will be zero. Also, if you want to use the compact form of the SVD, feel free to do so if that speeds up your derivation.

Hint 2: some inspiration may be possible by looking at the next part for an implicit clue as to what the answer might be.

Solution: The OLS on the k-PCA-reduced features reads

$$\min_{\mathbf{w}} \|\mathbf{X}\mathbf{V}_k\mathbf{w} - \mathbf{y}\|_2^2$$

where the columns of $\mathbf{V}_k$ consist of the first k eigenvectors of $\mathbf{X}$.

In the following we use the compact form SVD, that is note that one can write

$$\begin{aligned}\mathbf{X} &= \mathbf{U}\mathbf{\Sigma}\mathbf{V}\\ &= \mathbf{U}_d\mathbf{\Sigma}_d\mathbf{V}\end{aligned}$$

where $\mathbf{\Sigma}_d = \operatorname{diag}(\sigma_i)$ for $i = 1,\dots,d$ and $\mathbf{U}_d$ are the first d columns of $\mathbf{U}$. In general we use the notation $\mathbf{\Sigma}_k = \operatorname{diag}(\sigma_i)$ for $i = 1,\dots,k$.

Apply OLS on the new matrix $\mathbf{X}\mathbf{V}_k$ to obtain

$$\begin{aligned}\widehat{\mathbf{w}}_{\text{PCA}} &= [(\mathbf{X}\mathbf{V}_k)^\top(\mathbf{X}\mathbf{V}_k)]^{-1}(\mathbf{X}\mathbf{V}_k)^\top\mathbf{y}\\ &= [\mathbf{V}_k^\top\mathbf{V}\mathbf{\Sigma}_d^2\mathbf{V}^\top\mathbf{V}_k]^{-1}\mathbf{V}_k^\top\mathbf{X}^\top\mathbf{y}\\ &= \mathbf{\Sigma}_k^{-1}\mathbf{U}_k^\top\mathbf{y} = \widetilde{\mathbf{\Sigma}}_k^{-1}\mathbf{U}^\top\mathbf{y}\end{aligned}$$

where $\widetilde{\mathbf{\Sigma}}_k = \begin{pmatrix}\mathbf{\Sigma}_k & 0\end{pmatrix}$

The resulting prediction for PCA reads (note that you need to project it first!)

$$\begin{aligned}\widehat{\mathbf{y}}_{\text{PCA}} &= \mathbf{x}^\top\mathbf{V}_k\widehat{\mathbf{w}}_{\text{PCA}}\\ &= \mathbf{x}^\top\mathbf{V}_k\mathbf{\Sigma}_k^{-1}\mathbf{U}_k^\top\mathbf{y}\\ &= \mathbf{x}^\top\sum_{i=1}^{k}\frac{1}{\sigma_i}\mathbf{v}_i\mathbf{u}_i^\top\mathbf{y}\end{aligned}$$

and hence $\beta_i = \frac{1}{\sigma_i}$ if $i \le k$ and $\beta_i = 0$ for $i = k+1,\dots,d$.

(c) For the following part, $d = 5$. The following $\boldsymbol{\beta} := (\beta_1, \ldots, \beta_5)$ (written out to two significant figures) are the results of OLS (i.e. what we would get from ridge regression in the limit $\lambda \to 0$), λ-ridge-regression, and k-PCA-OLS for some $\mathbf{X}, \mathbf{y}$ (identical for each method) and $\lambda = 100, k = 3$. **For each of the three sub-parts below, determine if it is possible for the given β to be generated by k-PCA-OLS and explain your reasoning.**

We hope this helps you intuitively see the connection between these three methods.

Hint: The singular values of $\mathbf{X}$ *are* $\sigma_1 = 100$, $\sigma_2 = 10$, $\sigma_3 = 2$, $\sigma_4 = 0.1$, $\sigma_5 = 0.01$.

(i) $\boldsymbol{\beta} = (0.01, 0.1, 0.5, 10, 100)$

(ii) $\boldsymbol{\beta} = (0.01, 0.05, 0.02, 0, 0)$

(iii) $\boldsymbol{\beta} = (0.01, 0.1, 0.5, 0, 0)$

Solution: NEW SOLUTION

Impossible, Impossible, Possible.

Given the singular values and the solution to the previous problem, for k-OLS-PCA we have $\beta_i = 0$ for $i > k$. For all others $\beta_i = \frac{1}{\sigma_i}$.

OLD SOLUTION

Ridge, 3-PCA-OLS, OLS.

Reasoning: The prediction for OLS is the same as for PCA with $k = d$.

$$\hat{\mathbf{y}}_{OLS} = \mathbf{x}^\top \sum_{i=1}^{d} \frac{1}{\sigma_i} \mathbf{v}_i \mathbf{u}_i^\top \mathbf{y}$$

Putting all pieces together, we can thus see that PCA does "hard shrinkage" or "hard cutoff" (i.e. sets to zero) of the last $k+1, \ldots, d$ coefficients β_i, whereas ridge regression does "soft shrinkage" (i.e. shrinks towards zero) of the coefficients.

5 Kernel PCA

In lectures, discussion, and homework, we learned how to use PCA to do dimensionality reduction by projecting the data to a subspace that captures most of the variability. This works well for data that is roughly Gaussian shaped, but many real-world high dimensional datasets have underlying low-dimensional structure that is not well captured by linear subspaces. However, when we lift the raw data into a higher-dimensional feature space by means of a nonlinear transformation, the underlying low-dimensional structure once again can manifest as an approximate subspace. Linear dimensionality reduction can then proceed. As we have seen in class so far, kernels are an alternate way to deal with these kinds of nonlinear patterns without having to explicitly deal with the augmented feature space. This problem asks you to discover how to apply the "kernel trick" to PCA.

Let $\mathbf{X} \in \mathbb{R}^{n\times\ell}$ be the data matrix, where n is the number of samples and ℓ is the dimension of the raw data. Namely, the data matrix contains the data points $\mathbf{x}_j \in \mathbb{R}^\ell$ as rows

$$\mathbf{X} = \begin{pmatrix} \mathbf{x}_1^\top \\ \mathbf{x}_2^\top \\ \vdots \\ \mathbf{x}_n^\top \end{pmatrix} \in \mathbb{R}^{n\times\ell}. \tag{27}$$

(a) **Compute $\mathbf{X}\mathbf{X}^\top$ in terms of the singular value decomposition $\mathbf{X} = \mathbf{U}\Sigma\mathbf{V}^\top$ where $\mathbf{U} \in \mathbb{R}^{n\times n}$, $\Sigma \in \mathbb{R}^{n\times\ell}$ and $\mathbf{V} \in \mathbb{R}^{\ell\times\ell}$.** Notice that $\mathbf{X}\mathbf{X}^\top$ is the matrix of pairwise Euclidean inner products for the data points. **How would you get $\mathbf{U}$ if you only had access to $\mathbf{X}\mathbf{X}^\top$?**

Solution: By plugging in the compact SVD decomposition $\mathbf{X} = \mathbf{U}\Sigma\mathbf{V}^\top$ and using $\mathbf{U}^\top\mathbf{U} = \mathbf{I}$ we get

$$\mathbf{X}^\top\mathbf{X} = \mathbf{V}\Sigma\mathbf{U}^\top\mathbf{U}\Sigma\mathbf{V}^\top = \mathbf{V}\Sigma^2\mathbf{V}^\top.$$

Similarly with $\mathbf{V}^\top\mathbf{V} = \mathbf{I}$ we get

$$\mathbf{X}\mathbf{X}^\top = \mathbf{U}\Sigma\mathbf{V}^\top\mathbf{V}\Sigma\mathbf{U}^\top = \mathbf{U}\Sigma^2\mathbf{U}^\top.$$

Notice from the last line that $\mathbf{U}$ are the eigenvectors of $\mathbf{X}\mathbf{X}^\top$ with eigenvalues $\sigma_1^2, \sigma_2^2, \ldots, \sigma_\ell^2$ where $\sigma_1, \sigma_2, \ldots, \sigma_d$ are the singular values of $\mathbf{X}$ and can therefore be computed by performing an eigendecomposition of $\mathbf{X}\mathbf{X}^\top$.

(b) Given a new test point $\mathbf{x}_{test} \in \mathbb{R}^{\ell}$, one central use of PCA is to compute the projection of $\mathbf{x}_{test}$ onto the subspace spanned by the k top singular vectors $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k$.

Express the scalar projection $z_j = \mathbf{v}_j^\top \mathbf{x}_{test}$ onto the j-th principal component as a function of the inner products

$$\mathbf{X}\mathbf{x}_{test} = \begin{pmatrix} \langle \mathbf{x}_1, \mathbf{x}_{test} \rangle \\ \vdots \\ \langle \mathbf{x}_n, \mathbf{x}_{test} \rangle \end{pmatrix}. \tag{28}$$

Assume that all diagonal entries of $\mathbf{\Sigma}$ are nonzero and non-increasing, that is $\sigma_1 \geq \sigma_2 \geq \cdots > 0$.

Hint: Express $\mathbf{V}^\top$ in terms of the singular values $\mathbf{\Sigma}$, the left singular vectors $\mathbf{U}$ and the data matrix $\mathbf{X}$. If you want to use the compact form of the SVD, feel free to do so.

Solution: By multiplying the compact SVD $\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$ on both sides with $\mathbf{U}^\top$, we get $\mathbf{U}^\top\mathbf{X} = \mathbf{\Sigma}\mathbf{V}^\top$ and multiplying both sides of the new equation with $\mathbf{\Sigma}^{-1}$, we obtain

$$\mathbf{V}^\top = \mathbf{\Sigma}^{-1}\mathbf{U}^\top\mathbf{X}.$$

Therefore we get

$$z_j = \mathbf{v}_j^\top \mathbf{x}_{test} = \frac{1}{\sigma_j}\mathbf{u}_j^\top \mathbf{X}\mathbf{x}_{test}$$

.

(c) How would you define kernelized PCA for a general kernel function $k(\mathbf{x}_i, \mathbf{x}_j)$ (to replace the Euclidean inner product $\langle \mathbf{x}_i, \mathbf{x}_j \rangle$)? For example, the RBF kernel $k(\mathbf{x}_i, \mathbf{x}_j) = \exp\left(-\frac{\|\mathbf{x}_i - \mathbf{x}_j\|^2}{\delta^2}\right)$.

Describe this in terms of a procedure which takes as inputs the training data points $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n \in \mathbb{R}^\ell$ and the new test point $\mathbf{x}_{test} \in \mathbb{R}^\ell$, and outputs the analog of the previous part's z_j coordinate in the kernelized PCA setting. You should include how to compute U from the data, as well as how to compute the analog of $\mathrm{X}\mathbf{x}_{test}$ from the previous part.

Invoking the SVD or computing eigenvalues/eigenvectors is fine in your procedure, as long as it is clear what matrix is having its SVD or eigenvalues/eigenvectors computed. The kernel $k(\cdot, \cdot)$ can be used as a black-box function in your procedure as long as it is clear what arguments it is being given.

Solution: For kernelizing PCA, we replace inner products $\langle \mathbf{x}_i, \mathbf{x}_j \rangle$ with $k(\mathbf{x}_i, \mathbf{x}_j)$ and $\langle \mathbf{x}_i, \mathbf{x}_{test} \rangle$ with $k(\mathbf{x}_i, \mathbf{x}_{test})$, the procedure is then:

(a) Obtain the vectors $\mathbf{u}_j$ as eigenvectors from the eigendecomposition of the kernelized counterpart of the Gram matrix: $\mathbf{K} \in \mathbb{R}^{n \times n}$ with $\mathbf{K}_{ij} = k(\mathbf{x}_i, \mathbf{x}_j)$. The eigenvalues should be sorted in decreasing order. They are all non-negative real numbers because of the properties of kernels — the K matrix must be positive semi-definite.

(b) Kernelize the inner products $z_j = \frac{1}{\sigma_j}\mathbf{u}_j^\top \mathbf{X}\mathbf{x}_{test}$ from the previous part by using:

$$z_j = \frac{1}{\sigma_j}\mathbf{u}_j^\top \begin{pmatrix} k(\mathbf{x}_1, \mathbf{x}_{test}) \\ k(\mathbf{x}_2, \mathbf{x}_{test}) \\ \vdots \\ k(\mathbf{x}_n, \mathbf{x}_{test}) \end{pmatrix}, \tag{29}$$

where the σ_j are the square roots of the eigenvalues for the martix K above generated by using the kernel on all pairs of training points. Because these are non-negative real numbers, the square root is well defined.

6 Your Own Question

Write your own question, and provide a thorough solution.

Writing your own problems is a very important way to really learn the material. The famous "Bloom's Taxonomy" that lists the levels of learning is: Remember, Understand, Apply, Analyze, Evaluate, and Create. Using what you know to create is the top-level. We rarely ask you any HW questions about the lowest level of straight-up remembering, expecting you to be able to do that yourself. (e.g. make yourself flashcards) But we don't want the same to be true about the highest level.

As a practical matter, having some practice at trying to create problems helps you study for exams much better than simply counting on solving existing practice problems. This is because thinking about how to create an interesting problem forces you to really look at the material from the perspective of those who are going to create the exams.

Besides, this is fun. If you want to make a boring problem, go ahead. That is your prerogative. But it is more fun to really engage with the material, discover something interesting, and then come up with a problem that walks others down a journey that lets them share your discovery. You don't have to achieve this every week. But unless you try every week, it probably won't happen ever.