CS 189 Fall 2018 — Introduction to Machine Learning — HW3

This homework is due **Friday, September 21st at 10pm.**

2 Properties of kernels

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.

For a function k to be a valid kernel, it suffices to show either of the following conditions is true:

1. k has an inner product representation: $\exists\ \Phi : \mathbb{R}^d \to \mathcal{H}$, where $\mathcal{H}$ is some (possibly infinite-dimensional) inner product space such that $\forall \mathbf{x}_i, \mathbf{x}_j \in \mathbb{R}^d,\ k(\mathbf{x}_i, \mathbf{x}_j) = \langle \Phi(\mathbf{x}_i), \Phi(\mathbf{x}_j)\rangle$. The map Φ is called a *feature map* of the kernel k.

2. For every sample $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n \in R^d$, the Gram matrix

$$\mathbf{K} = \begin{bmatrix} k(\mathbf{x}_1,\mathbf{x}_1) & \cdots & k(\mathbf{x}_1,\mathbf{x}_n) \\ \vdots & k(\mathbf{x}_i,\mathbf{x}_j) & \vdots \\ k(\mathbf{x}_n,\mathbf{x}_1) & \cdots & k(\mathbf{x}_n,\mathbf{x}_n) \end{bmatrix}$$

   is positive semidefinite.

(a) Show that if the second condition holds, then for any finite set of vectors, $\mathcal{X} = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n\}$, in $\mathbb{R}^d$ there exists a feature map $\Phi_{\mathcal{X}}$ that maps the finite set $\mathcal{X}$ to $\mathbb{R}^n$ such that, for all $\mathbf{x}_i$ and $\mathbf{x}_j$ in $\mathcal{X}$, we have $k(\mathbf{x}_i, \mathbf{x}_j) = \langle \Phi_{\mathcal{X}}(\mathbf{x}_i), \Phi_{\mathcal{X}}(\mathbf{x}_j)\rangle$.

Solution: The Gram matrix of the data is a symmetric matrix: $\mathbf{K}_{ij} = k(\mathbf{x}_i, \mathbf{x}_j)$. This matrix admits an diagonoalization

$$\mathbf{K} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^\top,$$

where U is an orthogonal matrix with columns denoted by $\mathbf{u}_i$ and $\Lambda = \operatorname{diag}(\lambda_1, \lambda_2, \dots, \lambda_n)$ a diagonal matrix. The entries of Λ are non-negative because the Gram matrix is positive semi-definite. Therefore, we can define $\Phi_{\mathcal{X}}(\mathbf{x}_i) = (U\Lambda^{1/2})_i^\top$, the i-th column of $(U\Lambda^{1/2})^\top$. Then, by construction, we have $k(\mathbf{x}_i, \mathbf{x}_j) = \langle \Phi_{\mathcal{X}}(\mathbf{x}_i), \Phi_{\mathcal{X}}(\mathbf{x}_j)\rangle$.

(b) Show that when $k\colon \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ is a valid kernel, for all vectors $\mathbf{x}_1, \mathbf{x}_2 \in \mathbb{R}^d$ we have

$$k(\mathbf{x}_1, \mathbf{x}_2) \leq \sqrt{k(\mathbf{x}_1, \mathbf{x}_1)k(\mathbf{x}_2, \mathbf{x}_2)}.$$

Show how the classical Cauchy-Schwarz inequality is a special case.

Solution: The Gram matrix of two points must be positive semi-definite:

$$\begin{bmatrix} k(\mathbf{x}_1, \mathbf{x}_1) & k(\mathbf{x}_1, \mathbf{x}_2) \\ k(\mathbf{x}_2, \mathbf{x}_1) & k(\mathbf{x}_2, \mathbf{x}_2) \end{bmatrix} \succeq 0.$$

Therefore the determinant of this matrix must be non-negative. Since $k(\mathbf{x}_1, \mathbf{x}_2) = k(\mathbf{x}_2, \mathbf{x}_1)$, we get that

$$k(\mathbf{x}_1, \mathbf{x}_1)k(\mathbf{x}_2, \mathbf{x}_2) - k(\mathbf{x}_1, \mathbf{x}_2)^2 \geq 0.$$

Now the conclusion follows by simple algebraic manipulations.

We can recover the classic Cauchy-Schwarz inequality ($\langle \mathbf{x}_1, \mathbf{x}_2\rangle \leq \|\mathbf{x}_1\|_2\|\mathbf{x}_2\|_2$) by choosing k to be the linear kernel: $k(\mathbf{x}_1, \mathbf{x}_2) = \langle \mathbf{x}_1, \mathbf{x}_2\rangle$.

(c) Suppose k_1 and k_2 are valid kernels with feature maps $\Phi_1\colon \mathbb{R}^d \to \mathbb{R}^p$ and $\Phi_2\colon \mathbb{R}^d \to \mathbb{R}^q$ respectively, for some finite positive integers p and q. Construct a feature map for the product of the two kernels in terms of Φ_1 and Φ_2, i.e. construct Φ_3 such that for all $\mathbf{x}_1, \mathbf{x}_2 \in \mathbb{R}^d$ we have

$$k(\mathbf{x}_1, \mathbf{x}_2) = k_1(\mathbf{x}_1, \mathbf{x}_2)k_2(\mathbf{x}_1, \mathbf{x}_2) = \langle \Phi_3(\mathbf{x}_1), \Phi_3(\mathbf{x}_2)\rangle.$$

Solution:

We have

$$\begin{aligned} k_1(\mathbf{x}_1, \mathbf{x}_2)k_2(\mathbf{x}_1, \mathbf{x}_2) &= \langle \Phi_1(\mathbf{x}_1), \Phi_1(\mathbf{x}_2)\rangle\langle \Phi_2(\mathbf{x}_1), \Phi_2(\mathbf{x}_2)\rangle \\ &= \operatorname{tr}\left(\Phi_1(\mathbf{x}_1)^\top\Phi_1(\mathbf{x}_2)\Phi_2(\mathbf{x}_2)^\top\Phi_2(\mathbf{x}_1)\right) \\ &= \operatorname{tr}\left(\Phi_2(\mathbf{x}_1)\Phi_1(\mathbf{x}_1)^\top\Phi_1(\mathbf{x}_2)\Phi_2(\mathbf{x}_2)^\top\right). \end{aligned}$$

Therefore we can construct a feature map Φ_3 which maps $\mathbf{x}$ into $\mathbb{R}^{p\times q}$. More precisely, we define

$$\Phi_3(\mathbf{x}) = \Phi_1(\mathbf{x})\Phi_2(\mathbf{x})^\top.$$

Hence the product of two kernels is a valid kernel.

Recall that the inner product between two matrices $A, B \in R^{p\times q}$ is defined to be

$$\langle A, B\rangle = \operatorname{tr}\left(A^\top B\right) = \sum_{i=1}^{p}\sum_{j=1}^{q} A_{ij}B_{ij}.$$

3 Understanding specific kernels

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

$$k(\mathbf{x}_i, \mathbf{x}_j) = (1 + \mathbf{x}_i{}^\top \mathbf{x}_j)^p, \tag{1}$$

for any $\mathbf{x}_i, \mathbf{x}_j \in \mathbb{R}^\ell$. Assuming $\ell = 2$ and $p = 2$ and given a dataset $(\mathbf{x}_i, y_i)_{i=1,\dots,n}$, show the solution to kernel ridge regression is the same as the regularized least square solution to polynomial regression (with unweighted monomials as features) given the right choice of 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.

(Hint: you should consider a regularization term of the form $\|Mw\|_2^2$ for some matrix M. Such regularization is called *Tikhonov regularization*.)

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

$$\boldsymbol{\phi}(a) = (1, a_1^2, a_2^2, \sqrt{2}a_1, \sqrt{2}a_2, \sqrt{2}a_1a_2)^\top$$

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

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

$$\boldsymbol{\Phi} = \begin{bmatrix} \boldsymbol{\phi}(x_1)^\top \\ \boldsymbol{\phi}(x_2)^\top \\ \vdots \\ \boldsymbol{\phi}(x_n)^\top \end{bmatrix} \tag{2}$$

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

$$K_{ij} = \boldsymbol{\phi}(x_i)^\top\boldsymbol{\phi}(x_j) = (\boldsymbol{\Phi}\boldsymbol{\Phi}^\top)_{ij}. \tag{3}$$

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

$$\boldsymbol{\alpha} = (K + \lambda n \mathbf{I}_n)^{-1}\mathbf{y}. \tag{4}$$

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

$$f(x) = \mathbf{y}^\top(\mathbf{\Phi}\mathbf{\Phi}^\top + \lambda n \mathbf{I}_n)^{-1}\mathbf{\Phi}\boldsymbol{\phi}(x). \tag{5}$$

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

$$\tilde{\boldsymbol{\phi}}(a) = (1, a_1^2, a_2^2, a_1, a_2, a_1a_2)^\top$$

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

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

$$\tilde{\mathbf{\Phi}} = \begin{bmatrix} \tilde{\boldsymbol{\phi}}(x_1)^\top \\ \tilde{\boldsymbol{\phi}}(x_2)^\top \\ \vdots \\ \tilde{\boldsymbol{\phi}}(x_n)^\top \end{bmatrix} \tag{6}$$

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

$$\tilde{\boldsymbol{\phi}}(x) = \mathbf{D}\boldsymbol{\phi}(x), \tilde{\mathbf{\Phi}} = \mathbf{\Phi}\mathbf{D}, \tag{7}$$

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 $w \in \mathbb{R}^6$. Then, similarly to the solution of reguralired least squares, we get the closed form solution

$$\mathbf{w} = (\tilde{\mathbf{\Phi}}^\top\tilde{\mathbf{\Phi}} + \boldsymbol{M})^{-1}\tilde{\mathbf{\Phi}}^\top Y, \tag{8}$$

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

$$\boldsymbol{M} = \text{diag}(\lambda n, \lambda n, \lambda n, \lambda n/2, \lambda n/2, \lambda n/2) = \mathbf{D}(\lambda n\mathbf{I}_6)\mathbf{D}, \tag{9}$$

for the same parameter λ used in the regularization of the kernel ridge regression.

Then, the regression model produced by Tikhonov regression is

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

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

(b) In general, for any polynomial regression with pth order polynomial on $\mathbb{R}^\ell$ with an appropriately specified Tikhonov regularization, 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 the 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 $O(\ell^p)$. The complexity of solving least square is $O(\ell^{3p} + \ell^{2p}n)$. The total complexity is $O(\ell^{3p} + \ell^{2p}n)$.

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

Once the coefficients are found, making a prediction with the polynomial regression model takes $O(\ell^p)$ time, while making predictions with the kernel regression model takes time $O(n(\ell + \log p))$ time. As before, kernel regression is preferred when p is large.

(c) Show that the function k defined by $k(x_1, x_2) = \exp\left(\frac{x_1 x_2}{\gamma^2}\right)$ for all $x_1, x_2 \in \mathbb{R}$ is a valid kernel. Comment on the relation between polynomial regression and kernel regression with the kernel k. What effect does γ have on the importance of different features in the resulting regression model? (Hint: consider the Taylor series expansion of the function e^z.)

Solution: Recall that the Taylor expansion of e^z is

$$e^z = 1 + z + \frac{z^2}{2} + \sum_{j=3}^{\infty} \frac{z^j}{j!}.$$

Therefore, we can construct an infinite-dimensional feature map Φ such that

$$x \xrightarrow{\Phi} \left(1, \frac{x}{\gamma}, \frac{x^2}{\gamma^2\sqrt{2}}, \dots, \frac{x^j}{\gamma^j\sqrt{j!}}, \dots\right).$$

Then,

$$\langle \Phi(x_1), \Phi(x_2) \rangle = \sum_{i=1}^{\infty} \Phi(x_1)_i \Phi(x_2)_i = \exp\left(\frac{x_1 x_2}{\gamma^2}\right) = k(x_1, x_2).$$

From studying the feature map we see that as the parameter γ increases the high-degree features become less relevant for the kernel value between to data points, making behave more like a low degree polynomial kernel. As γ decreases to zero the opposite effect occurs.

(d) Consider the function $k\colon (0,1) \times (0,1) \to \mathbb{R}$ defined by $k(x_1, x_2) = \min\{x_1, x_2\}$. Prove that k is a valid kernel (Hint: write k as the integral of a product of two simple functions and then prove that its Gram matrices are positive semi-definite).

Now, consider a training set $\{(x_i, y_i)\}_{i=1,\dots,n}$ with $y_i \in \mathbb{R}$ and distinct points x_i in $(0,1)$. Show that if we ran kernel regression without regularization on this data set, we would obtain zero training error. More precisely, find explicit coefficients α_j, in terms of the training data, such that for all points (x_i, y_i) in the training set we have

$$\sum_{j=1}^{n} \alpha_j \min\{x_j, x_i\} = y_i.$$

Solution:

Let us consider the indicator functions

$$1_z(t) = \begin{cases} 1 & \text{if } t \le z, \\ 0 & \text{if } t > z. \end{cases}$$

Then,

$$\min\{x_1, x_2\} = \int_0^1 1_{x_1}(t) 1_{x_2}(t) dt.$$

Now, we need to show that for every $x_1, x_2, \dots x_n \in (0,1)$ the corresponing Gram matrix $\mathbf{K}$ is positive semi-definite. For every $\mathbf{v} \in \mathbb{R}^n$ we have

$$\begin{aligned}
\mathbf{v}^\top \mathbf{K} \mathbf{v} &= \sum_{i,j=1}^{n} v_i v_j \min\{x_i, x_j\} \\
&= \sum_{i,j=1}^{n} v_i v_j \int_0^1 1_{x_i}(t) 1_{x_j}(t) dt \\
&= \int_0^1 \sum_{i,j=1}^{n} v_i v_j 1_{x_i}(t) 1_{x_j}(t) dt \\
&= \int_0^1 \left(\sum_{i}^{n} v_i 1_{x_i}(t) \right) \left(\sum_{j}^{n} v_j 1_{x_j}(t) \right) dt \\
&= \int_0^1 \left(\sum_{i}^{n} v_i 1_{x_i}(t) \right)^2 dt \ge 0,
\end{aligned}$$

where the third equality follows by the linearity of integration.

For the second part of the problem, since the data points are distinct, we can assume without loss of generality that $x_1 < x_2 < \dots < x_n$ (we can just reorder the data points). A vector $\boldsymbol{\alpha}$ satisfies $\mathbf{K}\boldsymbol{\alpha} = \mathbf{y}$ if for every i we have

$$\sum_{j=1}^{n} \alpha_j \min\{x_j, x_i\} = y_i.$$

By the ordering assumption these linear constraints on $\boldsymbol{\alpha}$ can be rewritten as

$$\sum_{j=1}^{i-1} \alpha_j x_j + x_i \sum_{j=i}^{n} \alpha_j = y_i$$

(if $i = 1$, the first sum is defined to be zero).

Now, we substract the equation corresponding to y_{i-1} from the equation corresponding to y_i. We obtain

$$(x_i - x_{i-1}) \sum_{j=i}^{n} \alpha_j = y_i - y_{i-1}.$$

Since $x_i \neq x_{i+1}$, for every i from 1 to n (where x_0 and y_0 are defined to be zero) we obtain

$$\sum_{j=i}^{n} \alpha_j = \frac{y_i - y_{i-1}}{x_i - x_{i-1}}. \tag{10}$$

From these equations we see that

$$\alpha_n = \frac{y_n - y_{n-1}}{x_n - x_{n-1}},$$
$$\alpha_i = \frac{y_i - y_{i-1}}{x_i - x_{i-1}} - \frac{y_{i+1} - y_i}{x_{i+1} - x_i}, \; \forall i \in \{1, \ldots, n-1\}$$

satisfy the equations (10). Since equations (10) are linearly independent it follows that the original system of linear equestions $\mathbf{K}\boldsymbol{\alpha} = \mathbf{y}$ is also satisfied (one can also check this fact through some algebraic manipulations).

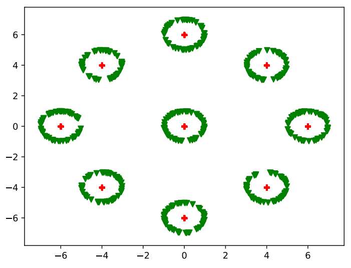
(e) Let k be a valid kernel with positive definite Gram matrices and let us consider a training set $\{(\mathbf{x}_i, y_i)\}_{i=1,\ldots,n}$ with $\mathbf{x}_i \in \mathbb{R}^d$ and $y_i \in \mathbb{R}$ for all i, and with all vectors $\mathbf{x}_i$ distinct. What training error should we expect if we ran kernel regression without regularization on this data set? Justify your answer.

Solution: We expect to see training error zero. When we perform kernel regression without regularization we obtain coefficients $\boldsymbol{\alpha} = \mathbf{K}^{-1}\mathbf{y}$ since the Gram matrix $\mathbf{K}$ is positive definite and therefore invertible. Since the predictions of the resulting regression model on the training set are $\mathbf{K}\boldsymbol{\alpha}$, it follows that training error will be zero.

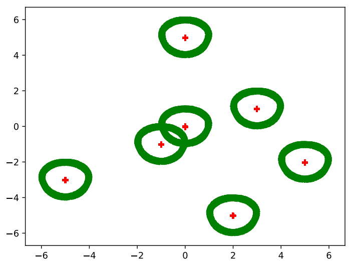
4 Kernel Ridge Regression: Practice

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.

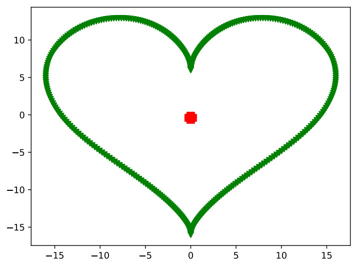
You are only allowed to use `numpy.*`, `numpy.linalg.*`, and `matplotlib` in the following questions. Make sure to include plots and results in your writeups.



(a) `circle.npz`



(b) `asymmetric.npz`



(c) `heart.npz`

Figure 1: Dataset visualization

(a) Use `matplotlib` 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.

Solution:

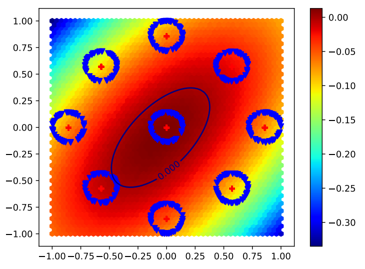
See Figure 1.

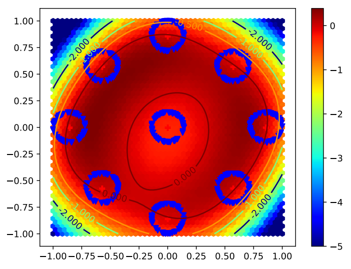
(b) Implement polynomial ridge regression (non-kernelized version that you should already have implemented in your previous homework) to fit the datasets `circle.npz`, `asymmetric.npz`, and `heart.npz`. Use the first 80% data as the training dataset and the last 20% data as the validation dataset. Report 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 homework 2.

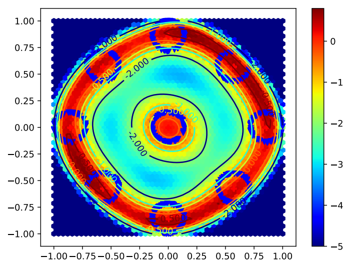
Solution:

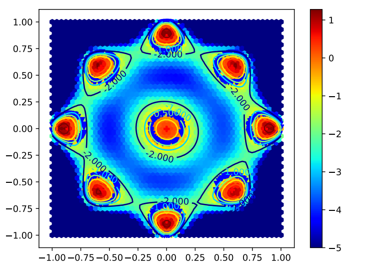
See Figure 2, 3, and 4. The error can be found in next part. If you directly use the code from homework 2, you may find that your result is slightly different from the error here due to the *difference of the constant terms* used in the polynomial, e.g., feature x_1x_2 vs feature $2x_1x_2$, but your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

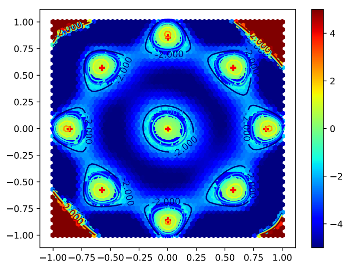
```
#!/usr/bin/env python3

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

```


(a) $p = 2$


(b) $p = 4$


(c) $p = 6$


(d) $p = 8$


(e) $p = 10$

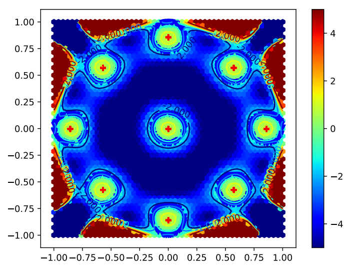
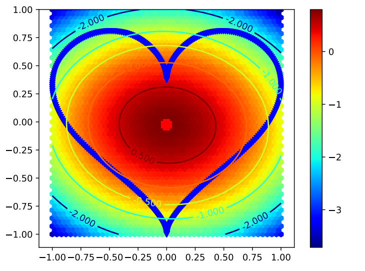
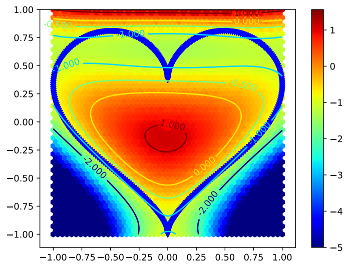
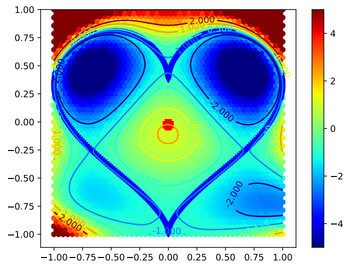
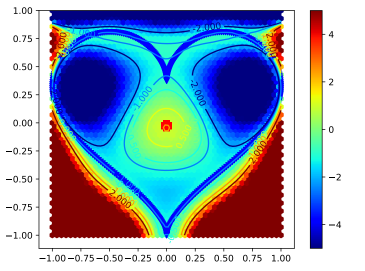

(f) $p = 12$

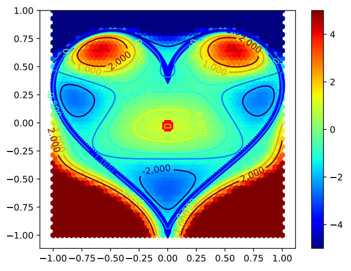
Figure 2: Heat map of circle.npz


(a) $p = 2$


(b) $p = 4$


(c) $p = 6$


(d) $p = 8$


(e) $p = 10$

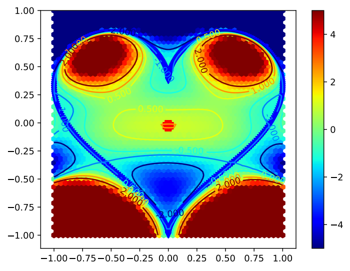
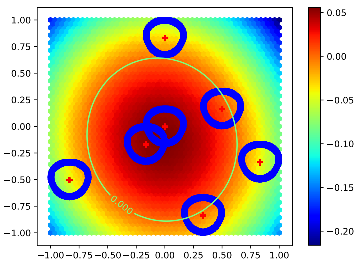
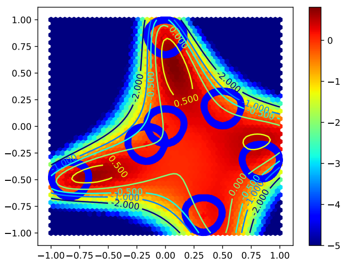
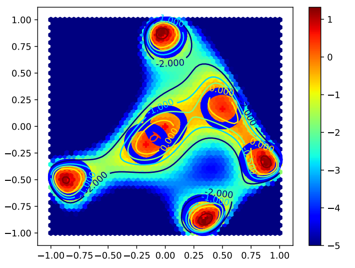
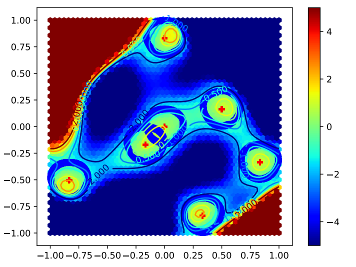

(f) $p = 12$

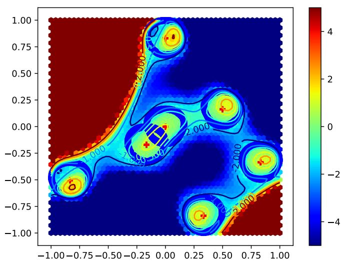
Figure 3: Heat map of heart.npz


(a) $p = 2$


(b) $p = 4$


(c) $p = 6$


(d) $p = 8$


(e) $p = 10$

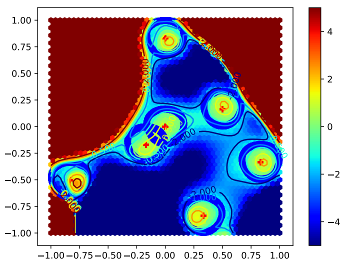

(f) $p = 12$

Figure 4: Heat map of asymmetric.npz

```
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=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 ###############
```

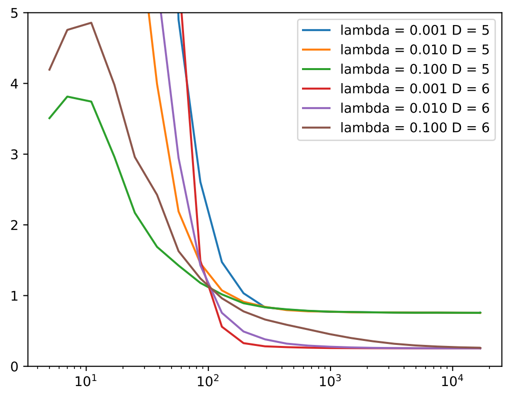
```
####### circle.npz #######
p =  1   train_error =   0.997088  validation_error =   0.997579  cond =       3.885463
p =  2   train_error =   0.995537  validation_error =   1.001056  cond =      40.439621
p =  3   train_error =   0.992699  validation_error =   1.019356  cond =     230.817918
p =  4   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')
```

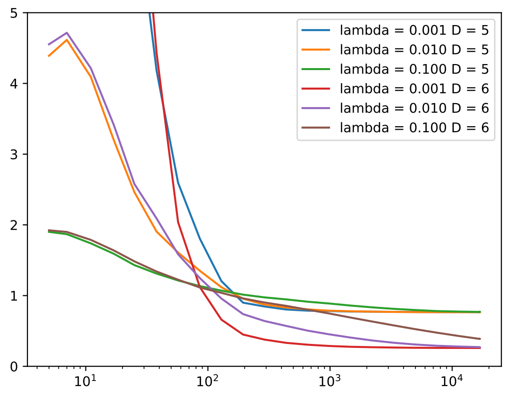
```python

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) With increasing of amount of data, the gains from regularization diminish. 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. 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. You can use log plot on x axis for clarity 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

```
#!/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

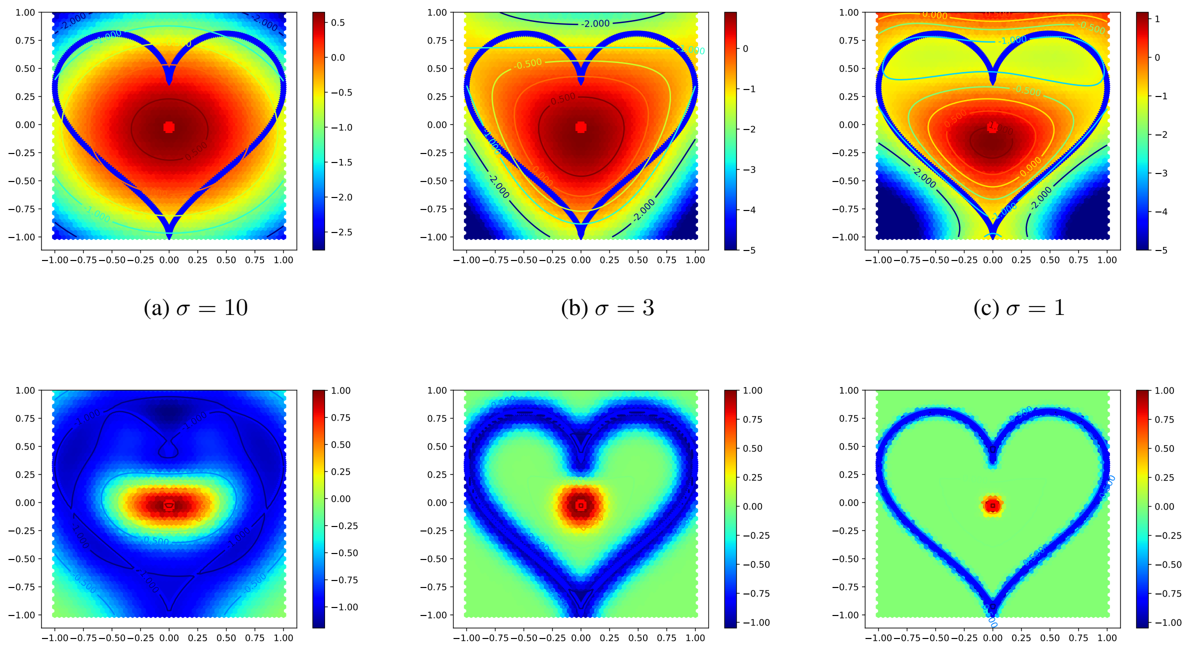
$$K(\mathbf{x},\mathbf{x}') = \exp\left(-\frac{\|\mathbf{x}-\mathbf{x}'\|_2^2}{2\sigma^2}\right). \tag{11}$$

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.

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`