CS 189:289A Introduction to Machine Learning
Fall 2021 Jennifer Listgarten and Jitendra Malik

HW4

Due 11/01 at 11:59pm

- We prefer that you typeset your answers using LaTeX or other word processing software. If you haven't yet learned LaTeX, one of the crown jewels of computer science, now is a good time! Neatly handwritten and scanned solutions will also be accepted for the written questions.
- In all of the questions, **show your work**, not just the final answer.

Deliverables:

1. Submit a PDF of your homework to the Gradescope assignment entitled "HW4 Write-Up". **Please start each question on a new page.** If there are graphs, include those graphs in the correct sections. **Do not** put them in an appendix. We need each solution to be self-contained on pages of its own.
   - In your write-up, please state with whom you worked on the homework. This should be on its own page and should be the first page that you submit.
   - In your write-up, please copy the following statement and sign your signature next to it. (Mac Preview and FoxIt PDF Reader, among others, have tools to let you sign a PDF file.) We want to make it *extra* clear so that no one inadvertently cheats. *"I certify that all solutions are entirely in my own words and that I have not looked at another student's solutions. I have given credit to all external sources I consulted."*
   - **Replicate all your code in an appendix**. Begin code for each coding question in a fresh page. Do not put code from multiple questions in the same page. When you upload this PDF on Gradescope, *make sure* that you assign the relevant pages of your code from appendix to correct questions.

1 Kernels

For a function $k(x_i, x_j)$ 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 $\forall x_i, x_j \in \mathbb{R}^d,\ k(x_i, x_j) = \langle \Phi(x_i), \Phi(x_j) \rangle$.
2. For every sample $x_1, x_2, \ldots, x_n \in R^d$, the kernel matrix

$$K = \begin{bmatrix} k(x_1, x_1) & \cdots & k(x_1, x_n) \\ \vdots & k(x_i, x_j) & \vdots \\ k(x_n, x_1) & \cdots & k(x_n, x_n) \end{bmatrix}$$

   is positive semidefinite.

   Starting with part (c), you can use either condition (1) or (2) in your proofs.

(a) Show that the first condition implies the second one, i.e. if $\forall x_i, x_j \in \mathbb{R}^d,\ k(x_i, x_j) = \langle \Phi(x_i), \Phi(x_j) \rangle$ then the kernel matrix K is PSD.

Solution: $\forall a \in \mathbb{R}^n, a^T K a = \sum_{i,j} a_i a_j k(x_i, x_j) = \sum_j a_j \langle \sum_i a_i \Phi(x_i), \Phi(x_j) \rangle = \langle \sum_i a_i \Phi(x_i), \sum_j a_j \Phi(x_j) \rangle \geq 0$

(b) Show that if the second condition holds, then for any finite set of vectors, $\mathcal{X} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$, 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 kernel 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 $\Lambda = \mathrm{diag}(\lambda_1, \lambda_2, \ldots, \lambda_n)$ a diagonal matrix. The entries of Λ are non-negative because the kernel 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 $k(\mathbf{x}_i, \mathbf{x}_j) = \langle \Phi_{\mathcal{X}}(\mathbf{x}_i), \Phi_{\mathcal{X}}(\mathbf{x}_j) \rangle$.

(c) Given a positive semidefinite matrix A, show that $k(x_i, x_j) = x_i^\top A x_j$ is a valid kernel.

Solution: We can show k admits a valid inner product representation:

$$k(x_i, x_j) = x_i^\top A x_j = x_i^\top P D^{1/2} D^{1/2} P^\top x_j = \langle D^{1/2} P^\top x_i, D^{1/2} P^\top x_j \rangle = \langle \Phi(x_i), \Phi(x_j) \rangle$$

where $\Phi(x) = D^{1/2} P^\top x$

(d) Give a counterexample that shows why $k(x_i, x_j) = x_i^\top (\mathrm{rev}(x_j))$ (where $\mathrm{rev}(x)$ reverses the order of the components in x) is *not* a valid kernel.

Solution: A counterexample: We have that $k((-1, 1), (-1, 1)) = -2$, but this is invalid since if k is a valid kernel then $\forall x,\ k(x, x) = \langle \Phi(x), \Phi(x) \rangle \geq 0$.

(e) 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 kernel 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: $k(\mathbf{x}_1, \mathbf{x}_2) = \langle \mathbf{x}_1, \mathbf{x}_2 \rangle$.

(f) 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

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

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

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

Solution:

We have

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

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.

2 Kernel Ridge Regression: Theory

(a) As we already know, the following procedure gives us the solution to ridge regression in feature space:

$$\arg\min_{\mathbf{w}} \|\mathbf{\Phi w} - \mathbf{y}\|_2^2 + \lambda\|\mathbf{w}\|_2^2 \tag{1}$$

Recall from Homework 1 that the solution to ridge regression is given by

$$\hat{\mathbf{w}} = (\mathbf{\Phi}^\top\mathbf{\Phi} + \lambda I_d)^{-1}\mathbf{\Phi}^\top y$$

Show that we can rewrite $\hat{\mathbf{w}}$ as

$$\hat{\mathbf{w}} = \mathbf{\Phi}^\top(\mathbf{\Phi\Phi}^\top + \lambda I_n)^{-1} y$$

You may have previously seen this in a past discussion.

Solution:

$$\begin{aligned}
(\mathbf{\Phi}^\top\mathbf{\Phi} + \lambda I_d)^{-1}\mathbf{\Phi}^\top &= (\mathbf{\Phi}^\top\mathbf{\Phi} + \lambda I_d)^{-1}\mathbf{\Phi}^\top(\mathbf{\Phi\Phi}^\top + \lambda I_n)(\mathbf{\Phi\Phi}^\top + \lambda I_n)^{-1} \\
&= (\mathbf{\Phi}^\top\mathbf{\Phi} + \lambda I_d)^{-1}(\mathbf{\Phi}^\top\mathbf{\Phi\Phi}^\top + \lambda\mathbf{\Phi}^\top)(\mathbf{\Phi\Phi}^\top + \lambda I_n)^{-1} \\
&= (\mathbf{\Phi}^\top\mathbf{\Phi} + \lambda I_d)^{-1}(\mathbf{\Phi}^\top\mathbf{\Phi} + \lambda I_d)\mathbf{\Phi}^\top(\mathbf{\Phi\Phi}^\top + \lambda I_n)^{-1} \\
&= \mathbf{\Phi}^\top(\mathbf{\Phi\Phi}^\top + \lambda I_n)^{-1} \\
\implies \hat{\mathbf{w}} &= (\mathbf{\Phi}^\top\mathbf{\Phi} + \lambda I_d)^{-1}\mathbf{\Phi}^\top y \\
&= \mathbf{\Phi}^\top(\mathbf{\Phi\Phi}^\top + \lambda I_n)^{-1} y
\end{aligned}$$

(b) The prediction for a test point $\mathbf{x}$ is given by $\phi(\mathbf{x})^\top\hat{\mathbf{w}}$, where $\hat{\mathbf{w}}$ is the solution to (1). In this part we will show how $\phi(\mathbf{x})^\top\hat{\mathbf{w}}$ can be computed using only the kernel $K(\mathbf{x}_i, \mathbf{x}_j) = \phi(\mathbf{x}_i)^\top\phi(\mathbf{x}_j)$. Denote the following object:

$$\mathbf{k}(\mathbf{x}) := [k(\mathbf{x}, \mathbf{x}_1), k(\mathbf{x}, \mathbf{x}_2), \ldots, k(\mathbf{x}, \mathbf{x}_n)]^\top$$

Using the result from part (a), show that

$$\phi(\mathbf{x})^\top\hat{\mathbf{w}} = \mathbf{k}(\mathbf{x})^\top(\mathbf{K} + \lambda I)^{-1}\mathbf{y}.$$

In other words, if we define $\hat{\alpha} := (\mathbf{K} + \lambda I)^{-1}\mathbf{y}$, then

$$\phi(\mathbf{x})^\top\hat{\mathbf{w}} = \sum_{i=1}^{n} \alpha_i K(\mathbf{x}, \mathbf{x}_i)$$

— our prediction is a linear combination of kernel functions at different data points.

Solution: From above we know that

$$\hat{\mathbf{w}} = \mathbf{\Phi}^\top(\mathbf{\Phi\Phi}^\top + \lambda I)^{-1} y.$$

Now we recognize that $(\mathbf{\Phi}\mathbf{\Phi}^\top)_{ij} = \phi(\mathbf{x}_i)^\top \phi(\mathbf{x}_j)$, and thus, $\mathbf{\Phi}\mathbf{\Phi}^\top = \mathbf{K}$. Thus,

$$\begin{aligned}\phi(\mathbf{x})^\top \hat{\mathbf{w}} &= \phi(\mathbf{x})^\top \mathbf{\Phi}^\top (\mathbf{K} + \lambda I)^{-1} y. \\ &= \mathbf{k}(\mathbf{x})^\top (\mathbf{K} + \lambda I)^{-1} y \\ &= \sum_{i=1}^{n} \alpha_i K(\mathbf{x}, \mathbf{x}_i).\end{aligned}$$

(c) We will now consider kernel functions that do not directly correspond to a finite-dimensional featurization of the input points. For simplicity, we will stick to a scalar underlying raw input x. (The same style of argument can help you understand the vector case as well.) Consider the radial basis function (RBF) kernel function

$$k(x, z) = \exp\left(-\frac{\|x - z\|^2}{2\sigma^2}\right),$$

for some fixed hyperparameter σ. It turns out that this kernel does not correspond to any finite-dimensional featurization $\phi(x)$. However, there exists a series $\phi_d(x)$ of d-dimensional features, such that $\phi_d(x)^T \phi_d(z)$ converges as $d \to \infty$ to $k(x, z)$. Using Taylor expansions, find $\phi_d(x)$. *(Hint: focus your attention on the Taylor expansion of $e^{\frac{xz}{\sigma^2}}$.)*

Solution: We can rewrite $k(x, z)$ as

$$k(x, z) = e^{-x^2/(2\sigma^2)} e^{-z^2/(2\sigma^2)} e^{xz/\sigma^2}.$$

Now observe that, by the Taylor expansion,

$$e^{xz/\sigma^2} = 1 + \frac{xz}{\sigma^2} + \frac{(xz)^2}{\sigma^4 \cdot 2!} + \frac{(xz)^3}{\sigma^6 \cdot 3!} + \cdots$$

We can rewrite this as the inner product of

$$\begin{bmatrix} 1 & \frac{x}{\sigma} & \frac{x^2}{\sigma^2 \sqrt{2!}} & \frac{x^3}{\sigma^3 \sqrt{3!}} \cdots \end{bmatrix}^T,$$

and

$$\begin{bmatrix} 1 & \frac{z}{\sigma} & \frac{z^2}{\sigma^2 \sqrt{2!}} & \frac{z^3}{\sigma^3 \sqrt{3!}} \cdots \end{bmatrix}^T.$$

Truncating to just d terms and substituting back into our expression for $k(x, z)$, we see that

$$k(x, z) \approx \phi_d(x)^T \phi_d(z),$$

where

$$\phi_d(x) = e^{-x^2/(2\sigma^2)} \begin{bmatrix} 1 & \frac{x}{\sigma} & \frac{x^2}{\sigma^2 \sqrt{2!}} & \cdots \frac{x^{d-1}}{\sigma^{d-1} \sqrt{(d-1)!}} \end{bmatrix}^T,$$

with equality achieved in the limit as $d \to \infty$.

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, the labels are actually discrete $y_i \in \{-1, +1\}$, so in practice you should probably use a different model such as kernel SVMs, kernel logistic regression, or neural networks. The use of ridge regression here is for your practice and ease of coding.

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) 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) to fit the datasets `circle.npz`, `asymmetric.npz`, and `heart.npz`. The data is already shuffled. 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 $p \in \{2, 4, 6, 8, 10, 12\}$. 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. Code for generating polynomial features and heatmap plots is included for your convenience.

Solution:

```
Dataset circle
p =  2   train_error =   0.995537  validation_error =   1.001056
p =  4   train_error =   0.943011  validation_error =   0.997914
p =  6   train_error =   0.547155  validation_error =   0.585688
p =  8   train_error =   0.230190  validation_error =   0.249990
p = 10   train_error =   0.174273  validation_error =   0.192998
p = 12   train_error =   0.156723  validation_error =   0.175335
Dataset heart
p =  2   train_error =   0.236718  validation_error =   0.189837
p =  4   train_error =   0.012169  validation_error =   0.009123
p =  6   train_error =   0.002630  validation_error =   0.001858
p =  8   train_error =   0.002354  validation_error =   0.001640
p = 10   train_error =   0.002193  validation_error =   0.001500
p = 12   train_error =   0.002090  validation_error =   0.001414
Dataset asymmetric
```

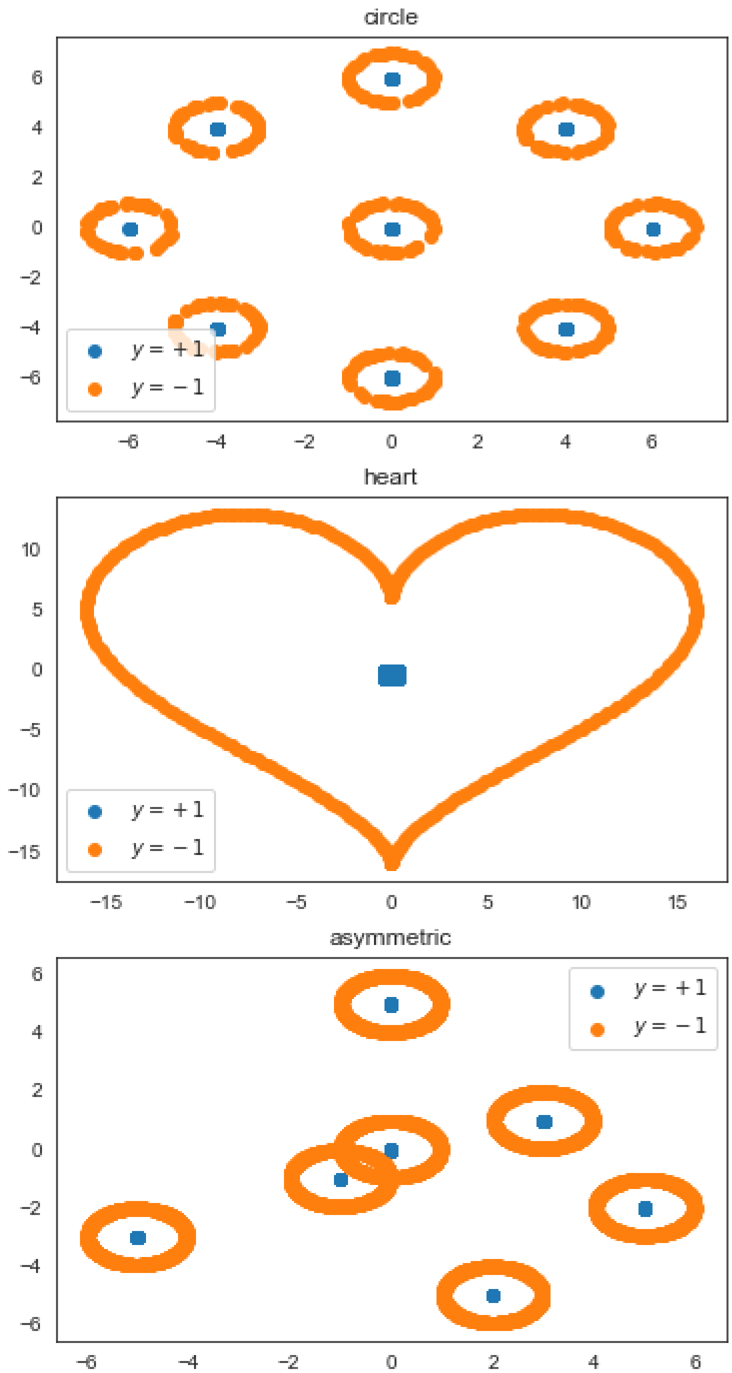



Figure 1: Dataset visualization

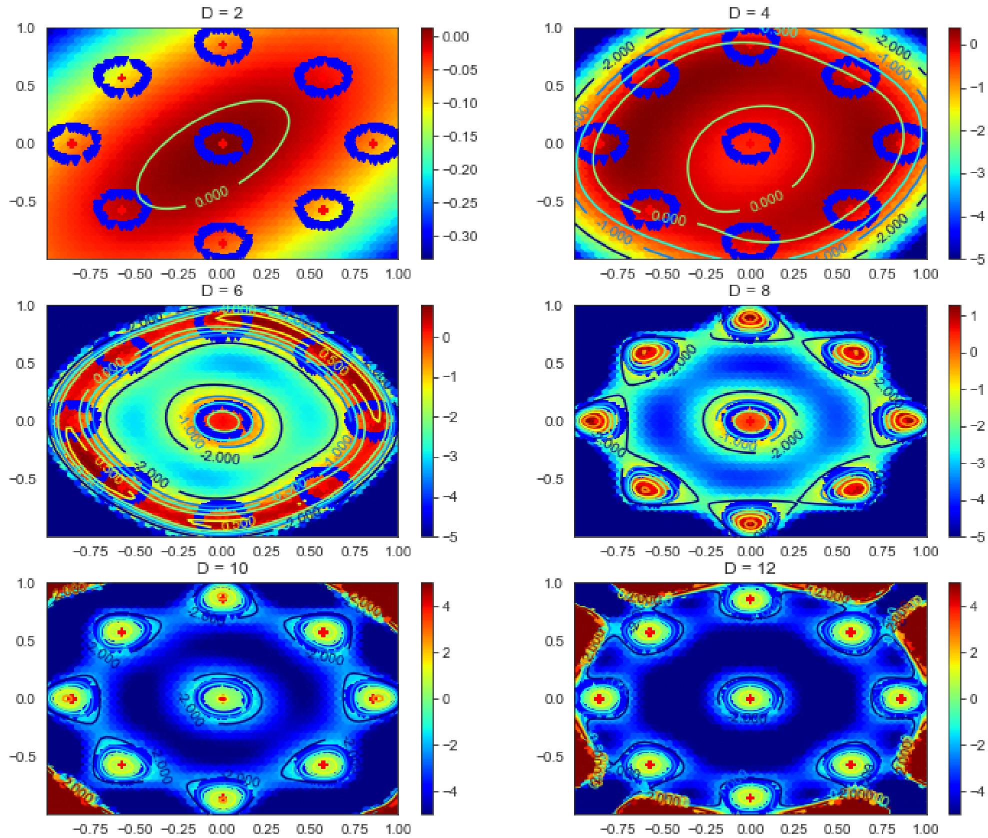


Figure 2: Heat map of circle.npz

```
p =  2   train_error =   0.998260  validation_error =   1.000176
p =  4   train_error =   0.828692  validation_error =   0.822369
p =  6   train_error =   0.264040  validation_error =   0.242398
p =  8   train_error =   0.179853  validation_error =   0.158347
p = 10   train_error =   0.157977  validation_error =   0.136623
p = 12   train_error =   0.151736  validation_error =   0.130519
```

See Figure 2, 3, and 4. The error can be found in next part.

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

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

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

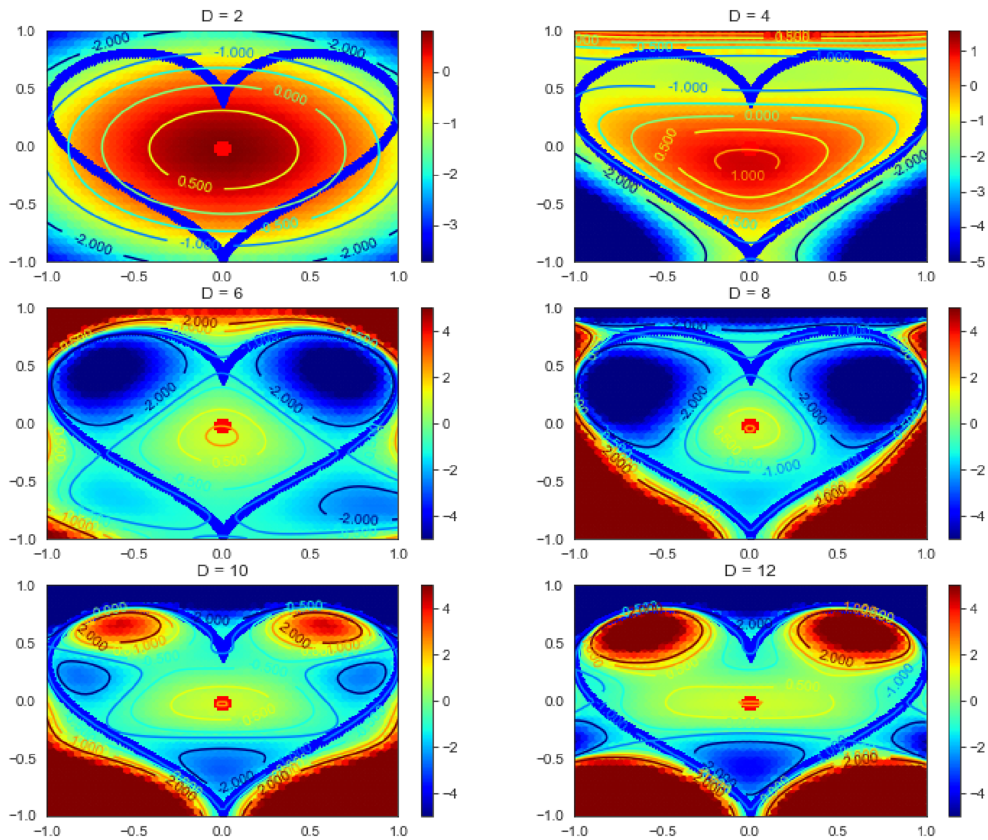
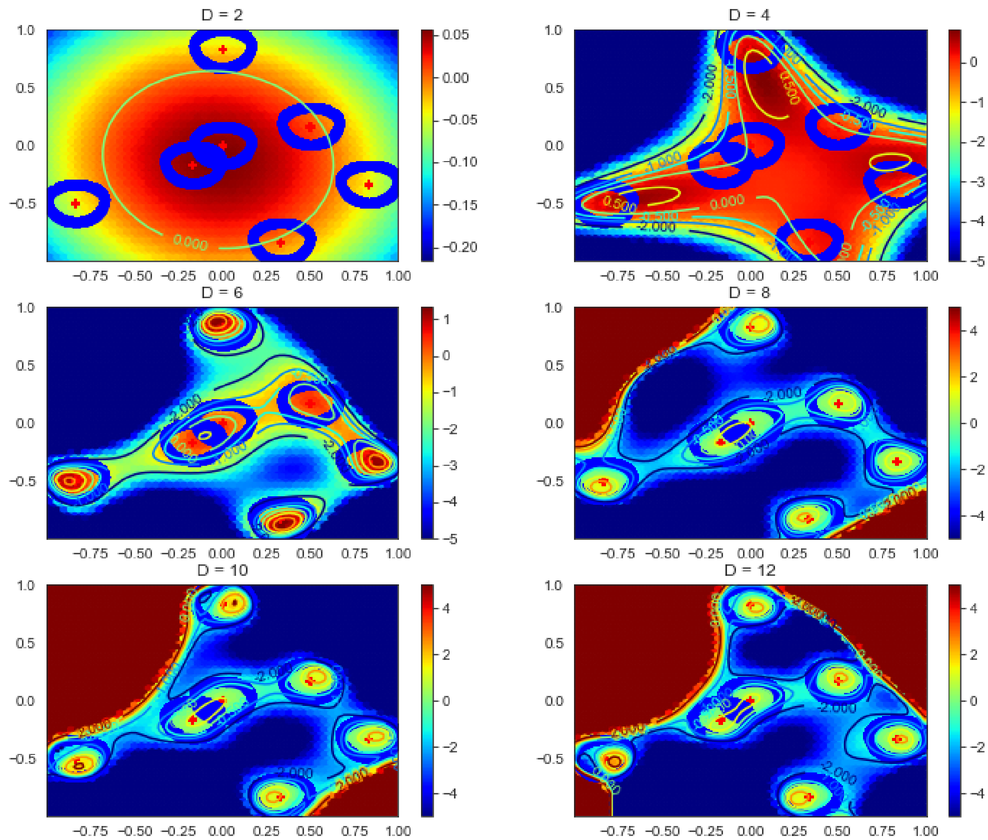



Figure 3: Heat map of heart.npz



(a) $p = 2$

Figure 4: Heat map of asymmetric.npz


```python
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):
    """Create a vector of polynomial features up to order D from x"""
    from scipy.special import binom
    xs = []
    for d0 in range(D + 1):
        for d1 in range(D - d0 + 1):
            xs.append((x[:, 0]**d0) * (x[:, 1]**d1))
    poly_x = np.column_stack(xs)
    return poly_x

def main():
    for ds in ['circle', 'heart', 'asymmetric']:
        data = np.load(f'{ds}.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]
        y_valid = y[n_train:]



        LAMBDA = 0.001
        isubplot = 0
        fig = plt.figure(figsize=[12,10])
        for D in range(1, 17):
            ### start poly_nonkernel ###
            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))
```

```python
            cond = np.linalg.cond(Xd_valid.T @ Xd_valid + np.eye(Xd_valid.shape[1]))
            ### end poly_nonkernel ###
            if D in [2, 4, 6, 8, 10, 12]:
                isubplot += 1
                plt.subplot(3,2,isubplot)
                heatmap(lambda x, y: assemble_feature(np.vstack([x, y]).T, D) @ w)
                plt.title("D = %d" % D)
            print("p = {:2d}   train_error = {:10.6f}  validation_error = {:10.6f}  cond = {:14.6f}".
                  format(D, error_train, error_valid, cond))
        fig.savefig(f"./result/{ds}_non_kernel.png")

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. 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. Lastly, comment on which of polynomial versus kernel ridge regression runs faster, and why.

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. For this problem, polynomial ridge regression runs faster than kernel ridge regression, because the number of data points is greater than the number of dimensions with the polynomial basis. 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)
```

```python
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 = {:14.6f}".
              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) 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

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

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 5 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.

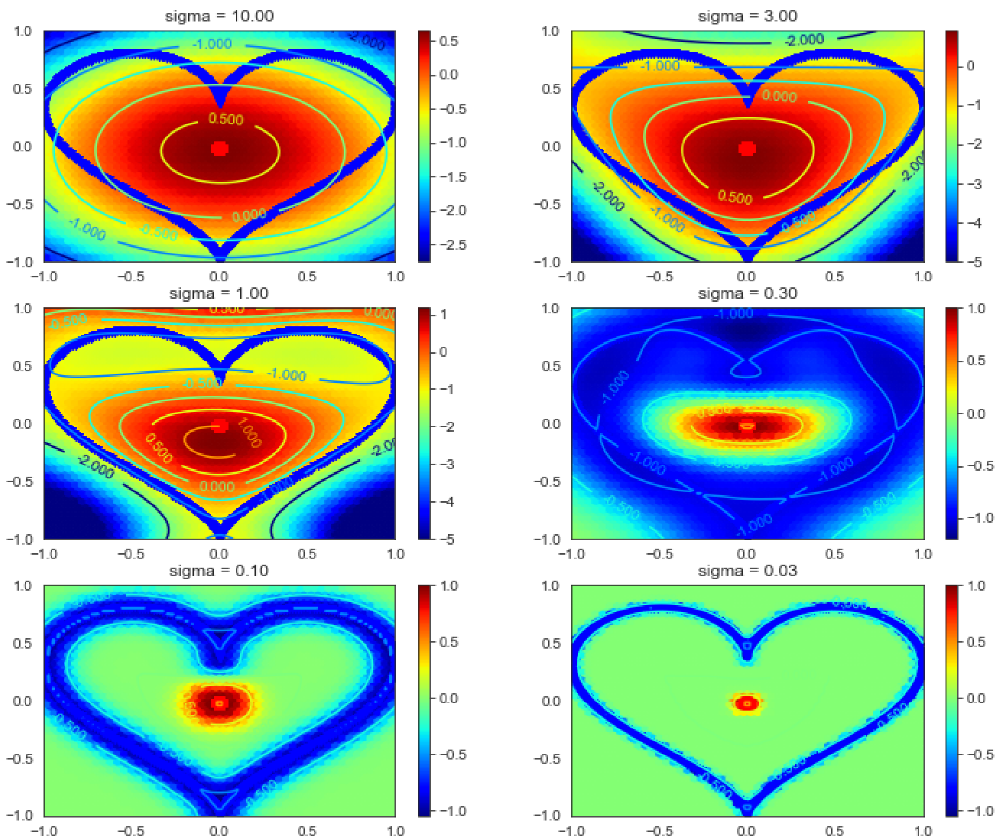


Figure 5: Heatmap of `heart.npz`

4 Eigenfaces

In this question we will perform and analyze PCA on a dataset consisting of images of faces. Each datapoint, $\mathbf{x}$, consists of a flattened $62x47$ pixel image (i.e. $\mathbf{x} \in \mathbb{R}^{2914}$). The dataset can be downloaded using the *sklearn* library as follows:

```
dataset = sklearn.datasets.fetch_lfw_people()
X = dataset['data']
```

This may take a few minutes to run. The file sizes are 250Mb so be sure that you have enough space on your computer to store the images. The *sklearn* library should only be used for downloading the data. For the rest of the problem you may use *matplotlib*, *numpy.**, *numpy.linalg.eig* and *numpy.linalg.svd*.

(a) Plot the first 20 images to get familiar with the dataset.

Note: when plotting the images, be sure to reshape them to be a matrix of size 62 x 47. Images can be plotted with matplotlib.pyplot.imshow. The argument cmap=matplotlib.pyplot.cm.gray provides the best colormap to view the images

Solution:

```
import sklearn
import sklearn.datasets
import numpy as np
import matplotlib.pyplot as plt

def show_image(image, h=62, w=47):
    """Helper function to plot a single image"""
    plt.imshow(image.reshape((h, w)), cmap=plt.cm.gray)
    plt.xticks(())
    plt.yticks(())

def show_images(images, titles=None, n_row=3, n_col=4, h=62, w=47):
    """Helper function to plot a gallery of images"""
    plt.figure(figsize=(1.8 * n_col, 2.4 * n_row))
    plt.subplots_adjust(bottom=0, left=.01, right=.99, top=.90, hspace=.35)
    if titles is None:
        titles = ["" for _ in images]
    for i in range(min(n_row * n_col, len(images))):
        plt.subplot(n_row, n_col, i + 1)
        plt.imshow(images[i].reshape((h, w)), cmap=plt.cm.gray)
        plt.title(titles[i], size=12)
        plt.xticks(())
        plt.yticks(())
```

```
dataset = sklearn.datasets.fetch_lfw_people()
X = dataset['data']
show_images(X[:20], n_row=4, n_col=5)
```

(b) Recall that in order to perform PCA, we must first center our data. Compute the average face of the dataset, center the data, and plot the average face.

Solution:

```
X_mean = X.mean(0)
X = (X - X_mean)
show_image(X_mean)
```

Happy Halloween!

(c) Perform PCA on the dataset. Plot the first 20 images reconstructed after being projected onto the top 10 principal components (PCs). Do the same after projecting onto the top 100 PCs and the top 1000 PCs.

Solution:

```
def project_and_reconstruct(X, V, r):
    X_r = X @ (V[:, :r])
    X_recon = X_r @ V.T[:r, :]
    return X_recon

lam, V = np.linalg.eig(X.T @ X)

r = 10
X_recon = project_and_reconstruct(X, V, r=r)
fig = show_images(X_recon[:20], n_row=4, n_col=5)

r = 100
X_recon = project_and_reconstruct(X, V, r=r)
fig = show_images(X_recon[:20], n_row=4, n_col=5)

r = 1000
X_recon = project_and_reconstruct(X, V, r=r)
fig = show_images(X_recon[:20], n_row=4, n_col=5)
```

Figure 6: 10 PCs

Figure 7: 100 PCs

Figure 8: 1000 PCs

(d) For this dataset, we refer to the PCs as “eigenfaces”. Plot the top 20 eigenfaces.

Solution:

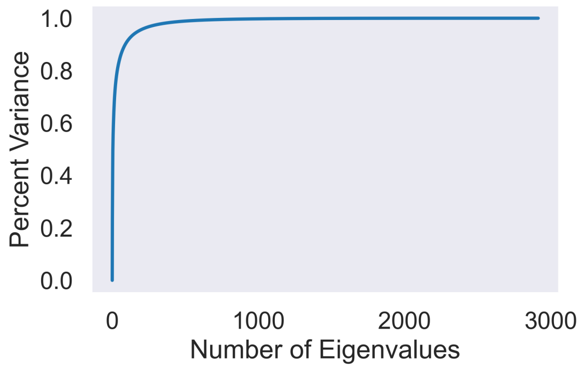
```
fig = show_images(V.T[:20], n_row=4, n_col=5)
```

(e) Recall from lecture that we can compute the percent variance explained by a certain number of PCs by using the eigenvalues of the covariance matrix. Plot the percent variance explained as a function of the number of PCs and determine how many PCs are needed to explain 95% of the variance.

Solution: The first 177 PCs are needed to explain 95% of the variance.

```
def perc_variance(lam, r):
    return (lam[:r] / lam.sum()).sum()
perc_vars = np.asarray([perc_variance(lam, r) for r in range(X.shape[1])])
rank = np.asarray([1 + i for i in range(len(perc_vars))])

fig, ax = plt.subplots()
ax.plot(rank, perc_vars)
ax.set(ylabel="Percent Variance", xlabel="Number of Eigenvalues")
```



(f) Use the first 80% of the dataset as your training set and the remaining 20% as the test set. We will use the training set to compute the PCs and we will evaluate our reconstruction loss on both the training and test set. For the following number of PCs, $[10, 20, 50, 100, 500, 1000, 2914]$, perform PCA using the training set and compute the average reconstruction loss for both the training and test set. Plot the error for both the training and test set as a function of the number of PCs.

Solution:

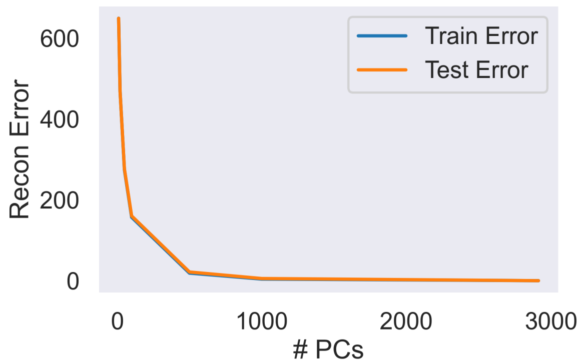
```
n, d = X.shape
X_train = X[:int(n*0.8)]
X_test = X[int(n*0.8):]
lam_train, V_train = np.linalg.eig(X_train.T @ X_train)

train_errors = []
test_errors = []
dims = [10, 20, 50, 100, 500, 1000, d]
for r in dims:
    X_train_recon = project_and_reconstruct(X_train, V_train, r)
    X_train_error = np.mean((X_train - X_train_recon)**2)

    X_test_recon = project_and_reconstruct(X_test, V_train, r)
    X_test_error = np.mean((X_test - X_test_recon)**2)

    train_errors.append(X_train_error)
    test_errors.append(X_test_error)

fig, ax = plt.subplots()
ax.plot(dims, train_errors, label="Train Error")
ax.plot(dims, test_errors, label="Test Error")
ax.legend()
```


Train Error
Test Error
Recon Error
600
400
200
0
0
1000
2000
3000
PCs