CS 189 Introduction to Machine Learning
Spring 2018 HW4

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

This homework is due **Friday, February 16 at 10pm.**

2 MLE of Multivariate Gaussian

In lecture, we discussed uses of the multivariate Gaussian distribution. We just assumed that we knew the parameters of the distribution (the mean vector $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$). In practice, though, we will often want to estimate μ and Σ from data. (This will come up even beyond regression-type problems: for example, when we want to use Gaussian models for classification problems.) This problem asks you to derive the Maximum Likelihood Estimate for the mean and variance of a multivariate Gaussian distribution.

(a) Let $\mathbf{X}$ have a multivariate Gaussian distribution with mean $\mu \in \mathbb{R}^d$ and covariance matrix $\Sigma \in \mathbb{R}^{d\times d}$. **Write the log likelihood of drawing the n i.i.d. samples $\mathbf{x}_1, \ldots, \mathbf{x}_n \in \mathbb{R}^d$ from X given Σ and μ.**

Solution: First, we calculate the likelihood by separating out the product:

$$\begin{aligned} P(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n|\mu, \Sigma) &= \prod_{i=1}^{n} P(\mathbf{x}_i|\mu, \Sigma) \\ &= \prod_{i=1}^{n} \frac{1}{(2\pi)^{d/2}|\Sigma|^{1/2}} \exp\{-\frac{1}{2}(\mathbf{x}_i - \mu)^\top \Sigma^{-1}(\mathbf{x}_i - \mu)\} \\ &= \frac{1}{(2\pi)^{nd/2}|\Sigma|^{n/2}} \exp\{-\frac{1}{2}\sum_{i=1}^{n}(\mathbf{x}_i - \mu)^\top \Sigma^{-1}(\mathbf{x}_i - \mu)\} \end{aligned}$$

Then we take the log:

$$\log P(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n|\mu, \Sigma) = -\frac{nd}{2}\log 2\pi - \frac{n}{2}\log|\Sigma| - \frac{1}{2}\sum_{i=1}^{n}(\mathbf{x}_i - \mu)^\top \Sigma^{-1}(\mathbf{x}_i - \mu)$$

(b) **Find MLE of μ and Σ.** For taking derivatives with respect to matrices, you may use any formula in "The Matrix Cookbook" without proof. This is a reasonably involved problem part with lots of steps to get to the answer. We recommend students first do the one-dimensional case and then the two-dimensional case to warm up.

Note: Conventions for gradient and derivative in "The Matrix Cookbook" may vary from the conventions we saw in the discussion.

Solution: To find the MLE of μ and Σ, we solve the following problem:

$$\begin{aligned}\hat{\mu}, \hat{\Sigma} &= \arg\max_{\mu,\Sigma} \log P(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n | \mu, \Sigma) \\ &= \arg\max_{\mu,\Sigma} -\frac{nd}{2}\log 2\pi - \frac{n}{2}\log|\Sigma| - \frac{1}{2}\sum_{i=1}^{n}(\mathbf{x}_i - \mu)^\top \Sigma^{-1}(\mathbf{x}_i - \mu) \\ &= \arg\min_{\mu,\Sigma} \sum_{i=1}^{n}(\mathbf{x}_i - \mu)^\top \Sigma^{-1}(\mathbf{x}_i - \mu) + n\log|\Sigma|\end{aligned}$$

Here, we expected many students would simply take the derivatives and set them to zero to find the optimum solution. If that is what you did, it is fine for now and you can give yourself full credit. However, strictly speaking, that is not a correct argument. This is because finding a zero derivative is just finding a critical point, not necessarily an optimal solution. In general, one needs to find all the critical points and check the "boundary" (which in this case would be points on the boundary of the positive semidefinite cone, since that is the domain of $\log|\Sigma|$. Alternatively, we can leverage what we know about convexity (a good optimization course like EECS 127 is really helpful here in building your intuition about this).

Anyway, since the objective above is not jointly convex, so we cannot simply take derivatives and set them to 0! Instead, we decompose the minimization into a nested optimization problem:

$$\min_{\mu,\Sigma} \sum_{i=1}^{n}(\mathbf{x}_i - \mu)^\top \Sigma^{-1}(\mathbf{x}_i - \mu) + \log|\Sigma| = \min_{\Sigma}\min_{\mu} \sum_{i=1}^{n}(\mathbf{x}_i - \mu)^\top \Sigma^{-1}(\mathbf{x}_i - \mu) + n\log|\Sigma|$$

The optimization problem has been decomposed into an inner problem that optimizes for μ given a fixed Σ, and an outer problem that optimizes for Σ given the optimal value $\hat{\mu}$. Let's first solve the inner optimization problem. Given a fixed positive definite Σ (i.e. symmetric with all eigenvalues strictly larger than zero), the objective is transparently convex in μ since it is a classic quadratic.

$$\hat{\mu} = \arg\min_{\mu} \sum_{i=1}^{n}(\mathbf{x}_i - \mu)^\top \Sigma^{-1}(\mathbf{x}_i - \mu) + \log|\Sigma| = \arg\min_{\mu} \sum_{i=1}^{n}(\mathbf{x}_i - \mu)^\top \Sigma^{-1}(\mathbf{x}_i - \mu)$$

So we find the MLE of the mean by differentiating with respect to μ. In discussion 0 we saw that $\frac{\partial x^\top A x}{\partial x} = x^\top(A^\top + A)$. If we denote $(x_i - \mu) = v_i$, we get that:

$$\frac{\partial (x_i - \mu)^\top \Sigma^{-1}(x_i - \mu)}{\partial \mu} = \frac{\partial v_i^\top \Sigma^{-1} v_i}{\partial v_i}\frac{\partial v_i}{\partial \mu} = (x_i - \mu)^\top(\Sigma^{-\top} + \Sigma^{-1})(-1) = -2(x_i - \mu)^\top \Sigma^{-1}$$

Therefore,

$$\frac{\partial}{\partial \mu}\sum_{i=1}^{n}(\mathbf{x}_i - \mu)^\top \Sigma^{-1}(\mathbf{x}_i - \mu) = \sum_{i=1}^{n}\frac{\partial}{\partial \mu}(\mathbf{x}_i - \mu)^\top \Sigma^{-1}(\mathbf{x}_i - \mu) = -2\sum_{i=1}^{n}(\mathbf{x}_i - \mu)^\top \Sigma^{-1}$$

We set the expression equal to 0:

$$\sum_{i=1}^{n} (\mathbf{x}_i - \mu)^\top \Sigma^{-1} = 0 \implies \hat{\mu} = \frac{\sum_{i=1}^{n} \mathbf{x}_i}{n}.$$

Notice that the choice of optimizing mean did not depend on the covariance Σ.

Having solved the inner optimization problem, we now have that

$$\hat{\Sigma} = \arg\min_{\Sigma} \sum_{i=1}^{n} (\mathbf{x}_i - \hat{\mu})^\top \Sigma^{-1} (\mathbf{x}_i - \hat{\mu}) + n \log |\Sigma|$$

We find the MLE of the variance by differentiating with respect to Σ, which requires using Equations 57 and 61 from The Matrix Cookbook.

Acording to Equation 61 we get that:

$$\frac{\partial((\mathbf{x}_i - \hat{\mu})^\top \Sigma^{-1} (\mathbf{x}_i - \hat{\mu}))}{\partial \Sigma} = -\Sigma^{-1} (\mathbf{x}_i - \hat{\mu})(\mathbf{x}_i - \hat{\mu})^\top \Sigma^{-1}$$

From Equation 57 we get that:

$$\frac{\partial \log |\Sigma|}{\partial \Sigma} = \Sigma^{-1}$$

Therefore,

$$\frac{\partial}{\partial \Sigma} \left(\sum_{i=1}^{n} (\mathbf{x}_i - \hat{\mu})^\top \Sigma^{-1} (\mathbf{x}_i - \hat{\mu}) + n \log |\Sigma| \right) = \sum_{i=1}^{n} -\Sigma^{-1} (\mathbf{x}_i - \hat{\mu})(\mathbf{x}_i - \hat{\mu})^\top \Sigma^{-1} + n\Sigma^{-1}$$

We set the expression equal to 0:

$$\sum_{i=1}^{n} -\Sigma^{-1} (\mathbf{x}_i - \hat{\mu})(\mathbf{x}_i - \hat{\mu})^\top \Sigma^{-1} + n\Sigma^{-1} = 0 \implies \hat{\Sigma} = \frac{1}{n} \sum_{i=1}^{n} (\mathbf{x}_i - \hat{\mu})(\mathbf{x}_i - \hat{\mu})^\top$$

It remains to argue that the unique stationary point we found for the optimization problem is indeed a minimum. Since there are no discontinuities to check, we do this by checking the boundary conditions for the objective

$$f(\mu, \Sigma) = \sum_{i=1}^{n} (\mathbf{x}_i - \mu)^\top \Sigma^{-1} (\mathbf{x}_i - \mu) + n \log |\Sigma|.$$

Consider fixed Σ now. For the objective to be finite, we need all eigenvalues of Σ to be positive (if this is not the case, the Σ^{-1} term is not defined). In this case, if one component $\mu_j \to \infty$ or $\mu_j \to -\infty$, the quadratic term goes to ∞.

On the other hand, now consider μ fixed and we vary Σ. Since we are optimizing over all positive semidefinite Σ, we approach the boundaries of the domain if eigenvalues of Σ are going to 0 or ∞. For an eigenvalue λ_i going to 0 or λ_i going to ∞ we have

$$f(\mu, \Sigma) = o\left(\frac{1}{\lambda_i}\right) + o\left(\log(\lambda_i)\right) \to \infty$$

and therefore the solution we found is indeed a minimum.

(c) Use the following code to sample from a two-dimensional Multivariate Gaussian and plot the samples:

```
import numpy as np
import matplotlib.pyplot as plt
mu = [15, 5]
sigma = [[20, 0], [0, 10]]
samples = np.random.multivariate_normal(mu, sigma, size=100)
plt.scatter(samples[:, 0], samples[:, 1])
plt.show()
```

Try the following three values of Σ:

$$\Sigma = \begin{bmatrix} 20 & 0 \\ 0 & 10 \end{bmatrix}; \quad \Sigma = \begin{bmatrix} 20 & 14 \\ 14 & 10 \end{bmatrix}; \quad \Sigma = \begin{bmatrix} 20 & -14 \\ -14 & 10 \end{bmatrix}.$$

Calculate the mean and covariance matrix of these distributions from the samples (that is, implement part (b)). Report your results. Include your code in your write-up. Note: you are allowed to use numpy.

Solution: There is no "correct" answer for the sample mean and variance, since it depends on randomness, but hopefully you got values for the MLE mean and variance that were close to the true mean and variance. If they are very far off, this may indicate an error in your code. The following code works by computing the formula above directly (including the matrix multiplication):

```
sample_mean = np.average(samples, axis=0)
sample_variance = 0.01 * np.sum(np.matrix((samples[i]-sample_mean)).T *
                  np.matrix((samples[i]-sample_mean)) for i in range(100))
```

3 Tikhonov Regularization and Weighted Least Squares

In lecture, you have seen this worked out in one way. In homework 2 we introduced Tikhonov regularization as a generalization of ridge regression. In this problem, we look at Tikhonov regularization from a probabilistic standpoint.

The main goal is to deepen your understanding of how priors and thus the right regularization affect the MAP estimator. First, you will work out how introducing a certain probabilistic prior before maximizing the posterior is equivalent to adding the Tikhonov regularization term: by adding the Tikhonov regularization term, we effectively constrain our optimization space. Similarly, using a probabilistic prior drives our optimization towards solutions that have a high (prior) probability of occurring. In the second half of the problem you will then do some simulations to see how different priors influence the estimator explicitly, as well as how this effect changes as the number of samples grows.

(a) Let $\mathbf{x} \in \mathbb{R}^d$ be a d-dimensional vector and $Y \in \mathbb{R}$ be a one-dimensional random variable. Assume a linear model: $Y = \mathbf{x}^\top \mathbf{w} + Z$ where $Z \in \mathbb{R}$ is a standard Gaussian random variable

$Z \sim N(0,1)$ and $\mathbf{w} \in \mathbb{R}^d$ is a d-dimensional Gaussian random vector $\mathbf{w} \sim N(0, \mathbf{\Sigma})$. $\mathbf{\Sigma}$ is a known symmetric positive definite covariance matrix. Note that $\mathbf{w}$ is independent of the observation noise. **What is the conditional distribution of Y given $\mathbf{x}$ and $\mathbf{w}$?**

Solution: A Gaussian random variable plus a constant is a Gaussian random variable with the mean having that constant added to it and the same variance, so $Y|\mathbf{x}, \mathbf{w} \sim N(\mathbf{x}^\top \mathbf{w}, 1)$. Concretely, we have

$$f(Y|\mathbf{x}, \mathbf{w}) = \frac{1}{\sqrt{2\pi}} \exp\{-\frac{1}{2}(Y - \mathbf{x}^\top \mathbf{w})^2\},$$

the pdf of Gaussian distribution.

(b) (Tikhonov regularization) Let us assume that we are given n training data points $\{(\mathbf{x}_1, Y_1), (\mathbf{x}_2, Y_2), \cdots, (\mathbf{x}_n, Y_n)\}$ which we know are generated i.i.d. according to the model of $(\mathbf{x}, Y)$ in the previous part, i.e. we draw one $\mathbf{w}$ and use this to generate all Y_i given distinct but arbitrary $\{\mathbf{x}_i\}_{i=1}^n$, but the observation noise Z_i varies across the different training points. **Derive the posterior distribution of w given the training data. Based on your result, what is the MAP estimate of w? Comment on how Tikhonov regularization is a generalization of ridge regression from a probabilistic perspective.**

Note: $\mathbf{w}$ and $\mathbf{Y} = (Y_1, Y_2, \dots, Y_n)$ are jointly Gaussian in this problem given $\{\mathbf{x}_i\}_{i=1}^n$.

Hint: (You may or may not find this useful) If the probability density function of a random variable is of the form

$$f(\mathbf{v}) = C \cdot \exp\{-\frac{1}{2}\mathbf{v}^\top \mathbf{A}\mathbf{v} + \mathbf{b}^\top \mathbf{v}\},$$

where C is some constant to make $f(\mathbf{v})$ integrates to 1, then the mean of $\mathbf{v}$ is $\mathbf{A}^{-1}\mathbf{b}$. This can be used to help complete squares if you choose to go that way.

Solution: By Bayes' Theorem, we have

$$f(\mathbf{w}|\mathbf{x}_1, \cdots, \mathbf{x}_n, Y_1, \cdots, Y_n) \propto [\prod_{i=1}^n f(Y_i|\mathbf{x}_i, \mathbf{w})] f(\mathbf{w})$$

The above derivation follows by omitting the constant terms in the denominator which don't depend on $\mathbf{w}$.

$$[\prod_{i=1}^n f(Y_i|\mathbf{x}_i, \mathbf{w})] f(\mathbf{w}) \propto [\prod_{i=1}^n \exp\{-\frac{1}{2}(Y_i - \mathbf{x}_i^\top \mathbf{w})^2\}] \exp\{-\frac{\mathbf{w}^\top \mathbf{\Sigma}^{-1}\mathbf{w}}{2}\}$$

The above follows from writing out the pdf directly.

$$\begin{aligned}[\prod_{i=1}^n \exp\{-\frac{1}{2}(Y_i - \mathbf{x}_i^\top \mathbf{w})^2\}] \exp\{-\frac{\mathbf{w}^\top \mathbf{\Sigma}^{-1}\mathbf{w}}{2}\} &\propto [\exp\{-\frac{1}{2}\sum_{i=1}^n (Y_i - \mathbf{x}_i^\top \mathbf{w})^2\}] \exp\{-\frac{\mathbf{w}^\top \mathbf{\Sigma}^{-1}\mathbf{w}}{2}\} \\ &\propto \exp\{-\frac{1}{2}[\mathbf{w}^T(\mathbf{X}^\top \mathbf{X} + \mathbf{\Sigma}^{-1})\mathbf{w} - 2(\mathbf{X}^\top \mathbf{Y})^\top \mathbf{w}]\},\end{aligned}$$

The above follows from expanding out the squared term. Here we have $\mathbf{X} \in \mathbb{R}^{n\times d}$ and the ith row of $\mathbf{X}$ is $\mathbf{x}_i^\top$ and we have $\mathbf{Y} \in \mathbb{R}^n$ with ith entry Y_i. Therefore the posterior of $\mathbf{w}|\mathbf{x}_1, \cdots, \mathbf{x}_n, Y_1, \cdots, Y_n$ is $N\big((\mathbf{X}^\top\mathbf{X} + \mathbf{\Sigma}^{-1})^{-1}\mathbf{X}^\top\mathbf{Y}, (\mathbf{X}^\top\mathbf{X} + \mathbf{\Sigma}^{-1})^{-1}\big)$. Derived based on the following lemma.

Lemma: If the probability density function of a random variable is of the form

$$f(\mathbf{v}) = C \cdot \exp\{-\frac{1}{2}\mathbf{v}^\top\mathbf{A}\mathbf{v} + \mathbf{b}^\top\mathbf{v}\},$$

where C is some constant to make $f(\mathbf{v})$ integrate to 1, and $\mathbf{A}$ is a symmetric positive definite matrix, then $\mathbf{v}$ is distributed as $N(\mathbf{A}^{-1}\mathbf{b}, \mathbf{A}^{-1})$.

The proof of the lemma is as follows:

$$\begin{aligned}
f(\mathbf{v}) &= C \cdot \exp\Big\{-\frac{1}{2}\mathbf{v}^\top\mathbf{A}\mathbf{v} + \mathbf{b}^\top\mathbf{v}\Big\}, \\
&= C_1 \cdot \exp\Big\{-\frac{1}{2}\Big(\mathbf{v} - (\mathbf{A}^{-1}\mathbf{b})\Big)^\top \mathbf{A}\Big(\mathbf{v} - (\mathbf{A}^{-1}\mathbf{b})\Big) - \frac{1}{2}\mathbf{b}^\top\mathbf{A}^{-1}\mathbf{b}\Big\}, \\
&= C_2 \cdot \exp\Big\{-\frac{1}{2}\Big(\mathbf{v} - (\mathbf{A}^{-1}\mathbf{b})\Big)^\top \mathbf{A}\Big(\mathbf{v} - (\mathbf{A}^{-1}\mathbf{b})\Big)\Big\},
\end{aligned}$$

which is the density of a Gaussian random variable with mean $\mathbf{A}^{-1}\mathbf{b}$ and covariance matrix $\mathbf{A}^{-1}$.

Now that we have the posterior distribution of $\mathbf{w}$, it is simple to find the MAP estimate. The MAP estimate is simply a maximization over the posterior probability. Since we know the posterior of $\mathbf{w}$ is a Gaussian, we have that the MAP estimate is the mean of that Gaussian. So we have that MAP estimate of $\mathbf{w}$ is $(\mathbf{X}^\top\mathbf{X} + \mathbf{\Sigma}^{-1})^{-1}\mathbf{X}^\top\mathbf{Y}$.

In ridge regression, we also assume a Gaussian prior on $\mathbf{w}$. However, we constrain the covariance of the prior to be a simple identity scaled by some constant. Tikhonov regularization generalizes this and allows us to select any covariance matrix for our prior on $\mathbf{w}$. Note in particular that this means our prior can involve non-zero covariance terms, or scale variances in different directions differently (e.g., the first entry of $\mathbf{w}$ may have a smaller variance than the second entry; Tikhonov regularization allows us to encode this knowledge in the problem setup).

(c) We have so far assumed that the observation noise has a standard normal distribution. While this assumption is nice to work with, we would like to be able to handle a more general noise model. In particular, we would like to extend our result from the previous part to the case where the observation noise variables Z_i are no longer independent across samples, i.e. $\mathbf{Z}$ is no longer $N(\mathbf{0}, \mathbb{I}_n)$ but instead distributed as $N(\boldsymbol{\mu}_z, \mathbf{\Sigma}_z)$ for some mean μ_z and some covariance $\mathbf{\Sigma}_z$ (still independent of the parameter $\mathbf{w}$). **Derive the posterior distribution of $\mathbf{w}$ by appropriately changing coordinates.** We make the reasonable assumption that the $\mathbf{\Sigma}_z$ is invertible, since otherwise, there would be some dimension in which there is no noise.

Hint: Write $\mathbf{Z}$ as a function of a standard normal Gaussian vector $\mathbf{V} \sim N(\mathbf{0}, \mathbb{I}_n)$ and use the result in (b) for an equivalent model of the form $\widetilde{\mathbf{Y}} = \widetilde{\mathbf{X}}\mathbf{w} + \mathbf{V}$.

Solution: By changing variables, our goal is to reduce to the previous case — with white observation noises.

We want to define matrix $\mathbf{\Sigma}_z^{1/2}$ such that $\mathbf{\Sigma}_z^{1/2}(\mathbf{\Sigma}_z^{1/2})^\top = \mathbf{\Sigma}_z$. According to eigenvalue decomposition $\mathbf{\Sigma}_z = \mathbf{U}\Delta\mathbf{U}^\top$. Exploiting the spectral theorem's guarantee of orthonormal eigenvalues and all positive eigenvalues to form the diagonal matrix Δ, we get that $\mathbf{\Sigma}_z^{\frac{1}{2}} = \mathbf{U}\Delta^{\frac{1}{2}}$.

Now, we define $\mathbf{V}$ such that $\mathbf{Z} = \boldsymbol{\mu}_z + \mathbf{\Sigma}_z^{\frac{1}{2}}\mathbf{V}$. Then $\mathbf{V} \sim N(\mathbf{0}, \mathbb{I})$ and $\mathbf{V} = \mathbf{\Sigma}_z^{-\frac{1}{2}}(\mathbf{Z} - \boldsymbol{\mu}_z)$.

$$\begin{aligned}
&\mathbf{Y} = \mathbf{X}\mathbf{w} + \mathbf{Z} \\
&\mathbf{Y} - \boldsymbol{\mu}_z = \mathbf{X}\mathbf{w} + \mathbf{Z} - \boldsymbol{\mu}_z \\
&\mathbf{\Sigma}_z^{-\frac{1}{2}}(\mathbf{Y} - \boldsymbol{\mu}_z) = \mathbf{\Sigma}_z^{-\frac{1}{2}}\mathbf{X}\mathbf{w} + \mathbf{\Sigma}_z^{-\frac{1}{2}}(\mathbf{Z} - \boldsymbol{\mu}_z) \\
&\mathbf{\Sigma}_z^{-\frac{1}{2}}(\mathbf{Y} - \boldsymbol{\mu}_z) = \mathbf{\Sigma}_z^{-\frac{1}{2}}\mathbf{X}\mathbf{w} + \mathbf{V}
\end{aligned}$$

So the problem reduces to the previous part by denoting $\mathbf{\Sigma}_z^{-\frac{1}{2}}\mathbf{X}$ as $\widetilde{\mathbf{X}}$ and donting $\widetilde{\mathbf{Y}}$ as $\mathbf{\Sigma}_z^{-\frac{1}{2}}(\mathbf{Y} - \boldsymbol{\mu}_z)$, which yields the posterior of $\mathbf{w}|\mathbf{x}_1, \cdots, \mathbf{x}_n, Y_1, \cdots, Y_n$ as a multivariate Gaussian

$$N\left((\widetilde{\mathbf{X}}^\top\widetilde{\mathbf{X}} + \mathbf{\Sigma}^{-1})^{-1}\widetilde{\mathbf{X}}^\top\widetilde{\mathbf{Y}}, (\widetilde{\mathbf{X}}^\top\widetilde{\mathbf{X}} + \mathbf{\Sigma}^{-1})^{-1}\right)$$

or

$$N\left((\mathbf{X}^\top\mathbf{\Sigma}_z^{-1}\mathbf{X} + \mathbf{\Sigma}^{-1})^{-1}\mathbf{X}^\top\mathbf{\Sigma}_z^{-1}(\mathbf{Y} - \boldsymbol{\mu}_z), (\mathbf{X}^\top\mathbf{\Sigma}_z^{-1}\mathbf{X} + \mathbf{\Sigma}^{-1})^{-1}\right).$$

(d) (Compare the effect of different priors) In this part, you will generate plots that show how different priors on $\mathbf{w}$ affect our prediction of the true $\mathbf{w}$ which generated the data points. Pay attention to how the amount of data used and the choice of prior relative to the true $\mathbf{w}$ we use are related to the final prediction.

Do the following for $\mathbf{\Sigma} = \mathbf{\Sigma}_1, \mathbf{\Sigma}_2, \mathbf{\Sigma}_3, \mathbf{\Sigma}_4, \mathbf{\Sigma}_5, \mathbf{\Sigma}_6$ respectively, where

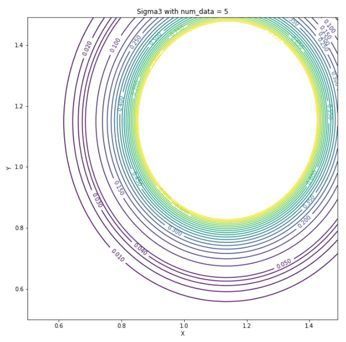
$$\mathbf{\Sigma}_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}; \quad \mathbf{\Sigma}_2 = \begin{bmatrix} 1 & 0.25 \\ 0.25 & 1 \end{bmatrix}; \quad \mathbf{\Sigma}_3 = \begin{bmatrix} 1 & 0.9 \\ 0.9 & 1 \end{bmatrix};$$
$$\mathbf{\Sigma}_4 = \begin{bmatrix} 1 & -0.25 \\ -0.25 & 1 \end{bmatrix}; \quad \mathbf{\Sigma}_5 = \begin{bmatrix} 1 & -0.9 \\ -0.9 & 1 \end{bmatrix}; \quad \mathbf{\Sigma}_6 = \begin{bmatrix} 0.1 & 0 \\ 0 & 0.1 \end{bmatrix}$$

Under the priors above, the coordinates of the (random) vector $\mathbf{w}$ are: (1) independent with large variance, (2) mildly positively correlated, (3) strongly positively correlated, (4) mildly negatively correlated, (5) strongly negatively correlated, and (6) independent with small variances respectively.

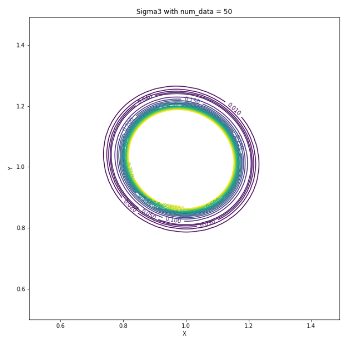
Using the starter code, generate $n \in \{5, 6, \ldots, 500\}$ data points $Y = x_1 + x_2 + Z$ with $x_1, x_2 \sim N(0, 5)$ and $Z \sim N(0, 1)$ as training data (here, the true $\mathbf{w}$ is thus $\begin{bmatrix} 1 & 1 \end{bmatrix}^\top$). Note that the randomness of x_i here is only for the generation of the plot but in our probabilistic model for parameter estimation we consider them as fixed and given. The starter code helps you generate an interactive plot where you can adjust the covariance prior and the number of samples used to calculate the posterior. **Include 6 plots of the contours of the posteriors on w for various settings of Σ and number of data points. Write the covariance prior and number of samples for each plot. What do you observe as the number of data points increases?**

Solution: We can observe that a good prior matters, but enough data will wash away the effect of the prior. A more detailed analysis is given in the next subproblem.

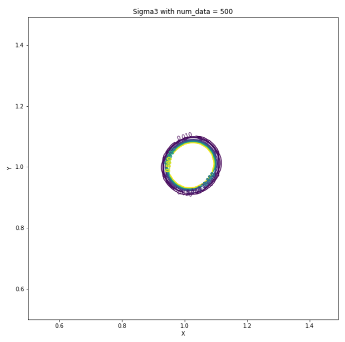
```python
# imports
import matplotlib
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.widgets import Slider
np.random.seed(0)

def generate_data(n):
    """
    This function generates data of size n.
    """
    X = np.random.randn(n,2)*np.sqrt(5)
    Z = np.random.randn(n)
    y = np.sum(X,axis=1) + Z
    return (X,y)

def tikhonov_regression(X,Y,Sigma):
    """
    This function computes w based on the formula of tikhonov_regression.
    """
    return np.linalg.inv((X.T.dot(X)+np.linalg.inv(Sigma))).dot(X.T.dot(Y))

def compute_mean_var(X,y,Sigma):
    """
    This function computes the mean and variance of the posterior
    """
    mean = tikhonov_regression(X,y,Sigma)
    var = np.linalg.inv(X.T.dot(X)+np.linalg.inv(Sigma))
    mux,muy = mean
    sigmax = np.sqrt(var[0,0])
    sigmay = np.sqrt(var[-1,-1])
    sigmaxy = var[0,-1]
    return mux,muy,sigmax,sigmay,sigmaxy

# Define the sigmas and number of samples to use
Sigmas = [np.array([[1,0],[0,1]]), np.array([[1,0.25],[0.25,1]]),
          np.array([[1,0.9],[0.9,1]]), np.array([[1,-0.25],[-0.25,1]]),
          np.array([[1,-0.9],[-0.9,1]]), np.array([[0.1,0],[0,0.1]])]
Num_Sample_Range = [5, 500]

##############################################################

def gen_plot():
    """
    This function refreshes the interactive plot.
    """
    plt.sca(ax)
    plt.cla()
    CS = plt.contour(X_grid, Y_grid, Z, levels =
        np.concatenate([np.arange(0,0.05,0.01),np.arange(0.05,1,0.05)]))
    plt.clabel(CS, inline=1, fontsize=10)
    plt.xlabel('X')
    plt.ylabel('Y')
    plt.title('Sigma'+ names[i] + ' with num_data = {}'.format(num_data))

names = [str(i) for i in range(1,len(Sigmas)+1)]

fig = plt.figure(figsize=(7.5,7.5))
ax = plt.axes()
plt.subplots_adjust(left=0.15, bottom=0.3)

# define the interactive sliders
sigma_ax = plt.axes([0.25, 0.1, 0.65, 0.03])
sample_ax = plt.axes([0.25, 0.15, 0.65, 0.03])
sigma_slider = Slider(sigma_ax, 'Sigma', valmin=0, valmax=len(Sigmas)-1e-5,
                         valinit=0, valfmt="%d")
num_data_slider = Slider(sample_ax, 'Num Samples', valmin=Num_Sample_Range[0],
                         valmax=Num_Sample_Range[1], valinit=Num_Sample_Range[0],
                         valfmt="%d")
sigma_slider.valtext.set_visible(False)
num_data_slider.valtext.set_visible(False)

# initial settings for plot
num_data = Num_Sample_Range[0]; Sigma = Sigmas[0]; i = 0
x = np.arange(0.5, 1.5, 0.01)
y = np.arange(0.5, 1.5, 0.01)
X_grid, Y_grid = np.meshgrid(x, y)
# Generate the function values of bivariate normal.
X, Y = generate_data(num_data)
mux,muy,sigmax,sigmay,sigmaxy = compute_mean_var(X,Y,Sigma)
Z = matplotlib.mlab.bivariate_normal(X_grid,Y_grid, sigmax, sigmay, mux, muy, sigmaxy)

def sigma_update(val):
    """
    This function is called in response to interaction with the Sigma sliding bar.
    """
    global Z, i
    if val != -1:
        i = int(val)
    Sigma = Sigmas[i]
    mux,muy,sigmax,sigmay,sigmaxy = compute_mean_var(X,Y,Sigma)
    Z = matplotlib.mlab.bivariate_normal(X_grid,Y_grid, sigmax, sigmay, mux, muy, sigmaxy)
    gen_plot()

def num_sample_update(val):
    """
    This function is called in response to interaction with the number of samples sliding bar
    ↪ .
    """
    global X, Y, num_data
    max_val = Num_Sample_Range[1]
    min_val = Num_Sample_Range[0]
    r = max_val - min_val
    num_data_ = int(((val - min_val) / r)**2 * r + min_val)
    if num_data == num_data_:
        return
    num_data = num_data_
    X, Y = generate_data(num_data)
    sigma_update(-1)

sigma_slider.on_changed(sigma_update)
num_data_slider.on_changed(num_sample_update)

gen_plot()
plt.show()
```



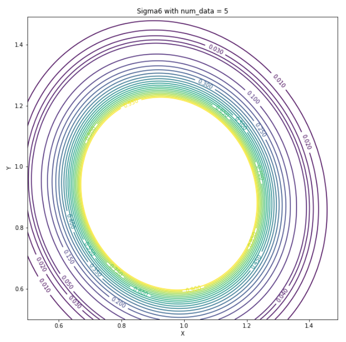
(a) $\mathbf{\Sigma}_3, n = 5$



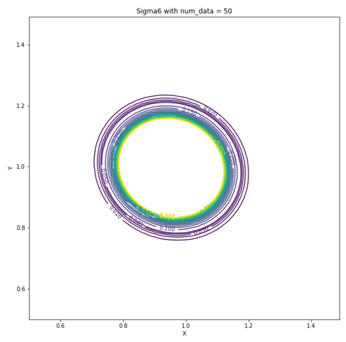
(b) $\mathbf{\Sigma}_3, n = 50$



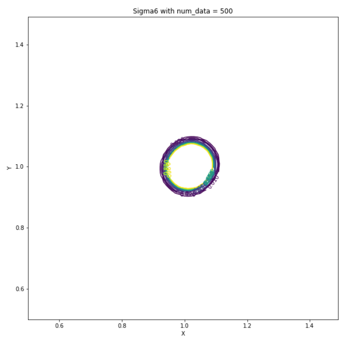
(c) $\mathbf{\Sigma}_3, n = 500$



(d) $\mathbf{\Sigma}_6, n = 5$



(e) $\mathbf{\Sigma}_6, n = 50$



(f) $\mathbf{\Sigma}_6, n = 500$

(e) (Influence of Priors) For our simulations, we will generate n training data samples from $Y = x_1 + x_2 + Z$ where again $x_1, x_2 \sim N(0, 5)$ and $Z \sim N(0, 1)$ (all of them independent of each other) as before. Notice that the true parameters $w_1 = 1, w_2 = 1$ are moderately large and positively correlated with each other. We want to quantitatively understand how the effect of the prior influences the mean square error as we get more training data. This should corroborate the qualitative results you saw in the previous part.

In this case, we could directly compute the "test error" for a given estimator $\widehat{\mathbf{w}}$ of the parameter $\mathbf{w}$ (our prediction for Y given a new data point $\mathbf{x} = (x_1, x_2)^\top$ is then $\widehat{Y} = \widehat{w}_1 x_1 + \widehat{w}_2 x_2$). Specifically, considering $\widehat{\mathbf{w}}$ now fixed, the expected error for a randomly drawn Y given the true (but unknown) parameter vector $\mathbf{w} = (1, 1)^\top$ is equal to $\mathbb{E}_{Z,\mathbf{x}}(Y - \widehat{Y})^2 = 5(\widehat{w}_1 - 1)^2 + 5(\widehat{w}_2 - 1)^2 + 1$. We call this the *theoretical average test error*. Note that here by our choice of definition, the expectation for new test samples is over $\mathbf{x}$ as well, although our estimator is not taking the randomness of $\mathbf{x}$ in the training data into account.

In practice, the expectation with respect to the true conditional distribution of Y given $\mathbf{w}$ cannot be computed since the true $\mathbf{w}$ is unknown. Instead, we are only given a finite amount of samples from the model (which we call the *test set*, which independent of the training data, but identically distributed) so that it is only possible to compute

$$\frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

which we call the *empirical average test error* (also known as MSE). Again, note that here, $\hat{Y}_i = \mathbf{x}_i^\top \widehat{\mathbf{w}}$ where $\mathbf{x}_i \in \mathbb{R}^2$ and $\widehat{\mathbf{w}}$ in your model is the solution to the least square problem with Tikhonov regularization given the training data.

Generate a test set of 500 data points $(\mathbf{x}_i, Y_i)$ from the above model. Plot the empirical and theoretical mean square error between $\hat{Y}_i$ and Y_i over the test data with respect to the size of training data n (increase n from 5 to 200 in increments of 5).

Note: If we just plotted both the empirical and theoretical average test errors with respect to the amount of training data for one "round" or training data, the results would still look jagged. In order to give a quantitive statement about the test error with respect to the training data n with a "smoother" plot, what we really want to know is the expectation of the theoretical average test error with respect to $\mathbf{w}$ and the training samples $(\mathbf{x}_i, Y_i)$, i.e. $\mathbb{E}_{(\mathbf{x}_1,Y_1),\ldots,(\mathbf{x}_n,Y_n)}\mathbb{E}_{Z,\mathbf{x}}(Y - \widehat{Y})^2$ (note that in this term, only $\widehat{Y}$ depends on the training data $(\mathbf{x}_1, Y_1), \ldots, (\mathbf{x}_n, Y_n)$ whereas $(\mathbf{x}, Y)$ is an independent fresh test sample). Consequently, as an approximation, it is worth replicating the entire experiment a few times (say 100 times) to get an empirical estimate of this quantity. (It is also insightful to look at the spread.) **Compare what happens for different priors as the amount of training data increases. Try plotting the theoretical MSE with logarithmic x and y-axes and explain the plot. What constitutes a "good" prior and which of the given priors are "good" choices for our particular $\mathbf{w} = (1,1)^\top$? Describe how the influence of different priors changes with the number of data points.**

Solution: We first plot the test MSE and theoretical MSE in the usual scale. We observe both of them decreases as n increases. But the lines overlap and are indistinguishable. Then we use a log scale for both x and y axis. Interestingly, we observe a linear relationship under the log scale, which follows from the fact that MSE decreases as the reciprocal of n, the number of training samples. And we can distinguish different priors better.

We can see in the log-scaled plot that Σ_3 has the best error for a small amount of training data, so we could say that Σ_3 is the best prior for $\mathbf{w} = (1,1)^\top$, however as we add more training data the influence of the prior washes away and all priors converge to roughly the same error.

We can summarize the quality of the priors as follows:

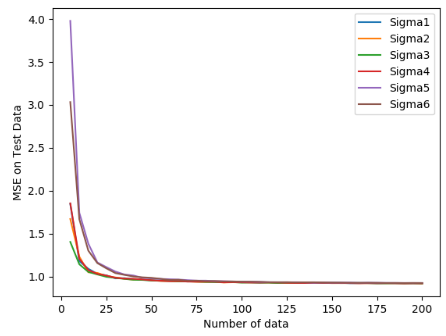
- Σ_3 is the best prior in terms of data efficiency because it asserts that the variables w_1 and w_2 are strongly correlated. This is indeed the case for the ground truth.
- Σ_1, Σ_2 and Σ_4 are doing pretty well in terms of data efficiency. This is because while they assert that the w_1 and w_2 are uncorrelated or mildly positively or negatively correlated, they leave a large amount of uncertainty on the values w_1 and w_2 by having large diagonal entries, so they can easily adapt to the data.
- Σ_5 and Σ_6 need the most data to estimate the parameters well. This is because they either assert the wrong model (w_1 and w_2 are anti-correlated) or are very certain about the wrong model (independent w_1 and w_2). Therefore we need more data to "overrule" the prior.

Three observations:

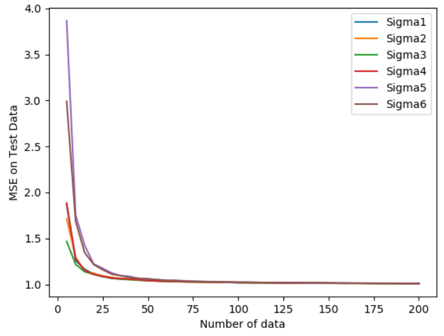
- Enough data will wash away the effect of prior.

- A good prior helps when there are not enough data.
- How to plot is IMPORTANT for scientific observation.

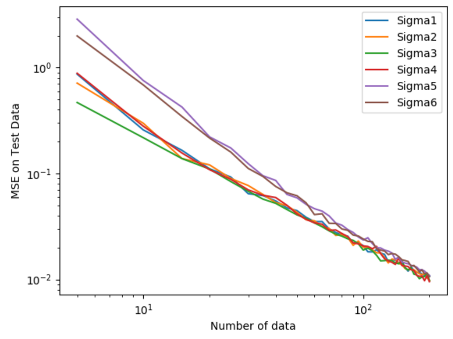
```python
import numpy as np
import matplotlib.pyplot as plt
np.random.seed(0)
w = [1.0,1.0]
n_test = 500
n_trains = np.arange(5,205,5)
n_trails = 100

Sigmas = [np.array([[1,0],[0,1]]), np.array([[1,0.25],[0.25,1]]),
          np.array([[1,0.9],[0.9,1]]), np.array([[1,-0.25],[-0.25,1]]),
          np.array([[1,-0.9],[-0.9,1]]), np.array([[0.1,0],[0,0.1]])]
names = ['Sigma{}'.format(i+1) for i in range(6)]

def generate_data(n):
    """
    This function generates data of size n.
    """
    X = np.random.randn(n,2) * np.sqrt(5)
    Z = np.random.randn(n)
    y = np.sum(X,axis=1) + Z
    return (X,y)

def tikhonov_regression(X,Y,Sigma):
    """
    This function computes w based on the formula of tikhonov_regression.
    """
    return np.linalg.inv((X.T.dot(X)+np.linalg.inv(Sigma))).dot(X.T.dot(Y))

def compute_mse(X,Y, w):
    """
    This function computes MSE given data and estimated w.
    """
    return np.mean((np.squeeze(X.dot(w)) - Y)**2)

def compute_theoretical_mse(w):
    return sum((w-1) ** 2) * 5 + 1

# Generate Test Data.
X_test, y_test = generate_data(n_test)

mses = np.zeros((len(Sigmas), len(n_trains), n_trails))

theoretical_mses = np.zeros((len(Sigmas), len(n_trains), n_trails))

for seed in range(n_trails):
    np.random.seed(seed)
    for i,Sigma in enumerate(Sigmas):
        for j,n_train in enumerate(n_trains):
            X,y = generate_data(n_train) # Generate data.
            w = tikhonov_regression(X,y,Sigma) # Estimate w.
            mses[i,j,seed] = compute_mse(X_test, y_test, w) # Compute MSE.
            theoretical_mses[i,j,seed] = compute_theoretical_mse(w)

# Plot
plt.figure()
for i,_ in enumerate(Sigmas):
    plt.plot(n_trains, np.mean(mses[i],axis = -1),label = names[i])
plt.xlabel('Number of data')
plt.ylabel('MSE on Test Data')
plt.legend()
plt.savefig('MSE.png')

plt.figure()
for i,_ in enumerate(Sigmas):
    plt.plot(n_trains, np.mean(theoretical_mses[i],axis = -1),label = names[i])
plt.xlabel('Number of data')
plt.ylabel('MSE on Test Data')
plt.legend()
plt.savefig('theoretical_MSE.png')

plt.figure()
for i,_ in enumerate(Sigmas):
    plt.loglog(n_trains, np.mean(theoretical_mses[i]-1,axis = -1),label = names[i])
plt.xlabel('Number of data')
plt.ylabel('MSE on Test Data')
plt.legend()
plt.savefig('log_theoretical_MSE.png')
```



(g) Average (over runs) MSE on Test Data. We can see this dropping but it is hard to understand what exactly is going on.



(h) Average (over runs) Theoretical MSE. This reveals that there are still some wiggles — so the wiggles are also coming from variance in our estimator (given that we are only taking the average over a finite number of runs), not random peculiarities of our test data.



(i) Log log plot of average (over runs) theoretical MSE. By looking at a log-log plot, where both axes are on log scale, we can see more clearly the pattern — the quality of the prior impacts where we start with small amounts of training data, but the rate of convergence is given by a power-law.

4 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):

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

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

Argue that the right hand side is equal to

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

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 $\nabla_{\mathbf{w},\mathbf{r}}L(\mathbf{w},\mathbf{r}) = 0$. Writing the partial derivatives out, we get

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

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

Plugging them into the Lagrangian, we get

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

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

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

(c) **Finally, prove that the optimal w^* can be computed using**

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

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

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

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

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

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

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

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

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

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

That is

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

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

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

where

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The predictor produced by Tikhonov regression is

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

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 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.

5 Kernel Ridge Regression: Practice

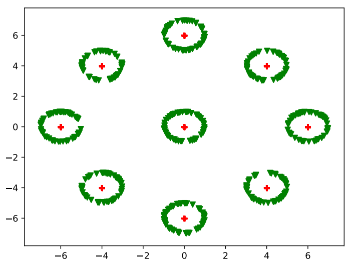
In the following problem, you will implement Polynomial Ridge Regression and its kernel variant Kernel Ridge Regression, and compare them with each other. You will be dealing with a 2D regression problem, i.e., $\mathbf{x}_i \in \mathbb{R}^2$. We give you three datasets, `circle.npz` (small dataset), `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.

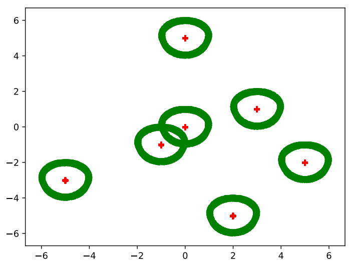
Solution:

See Figure 1.

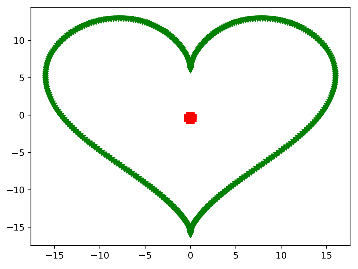
(b) **Implement polynomial ridge regression** (non-kernelized version that you should already have implemented in your previous homework) **to fit the datasets `circle.npz`, `asymmetric.npy`, and `heart.npz`**. Use the first 80% data as the training dataset and the last 20% data as the validation dataset. **Report both the average training squared loss and the average validation squared for polynomial order** $p \in \{1, \ldots, 16\}$. 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, problem 5.



(j) `circle.npz`

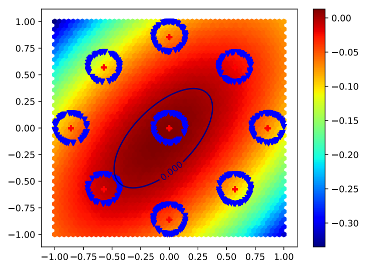


(k) `asymmetric.npz`

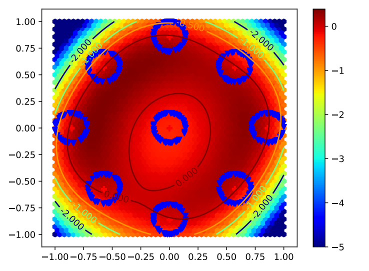


(l) `heart.npz`

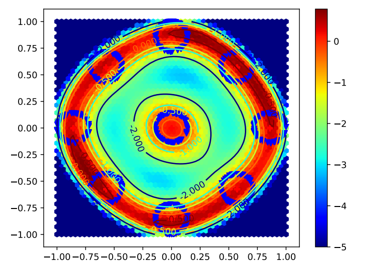
Figure 1: Dataset visualization



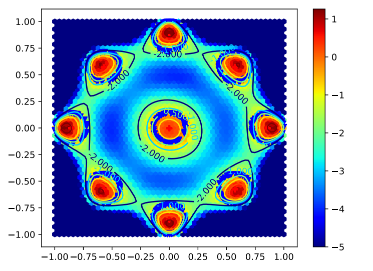
(a) $p = 2$



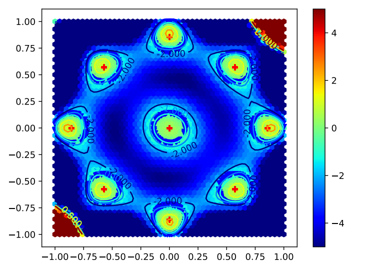
(b) $p = 4$



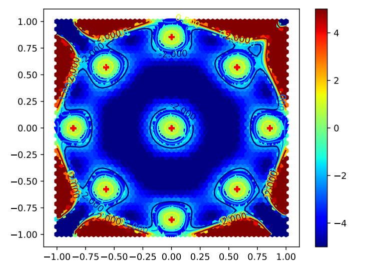
(c) $p = 6$



(d) $p = 8$

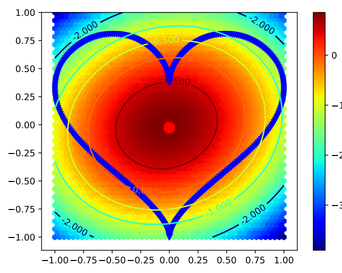


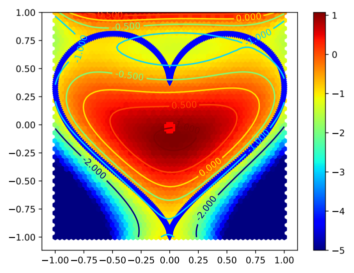
(e) $p = 10$

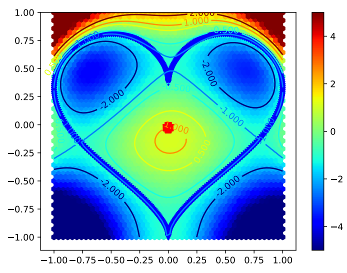


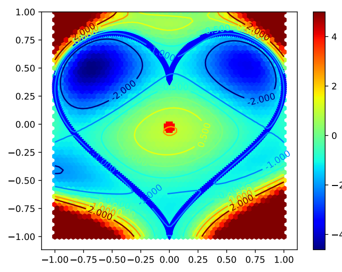
(f) $p = 12$

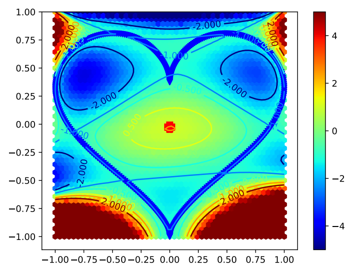
Figure 2: Heat map of circle.npz


(a) $p = 2$


(b) $p = 4$


(c) $p = 6$


(d) $p = 8$


(e) $p = 10$

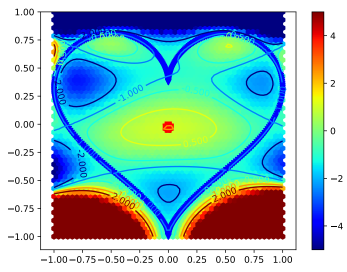
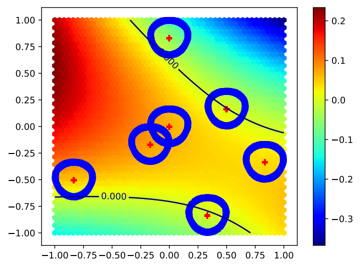
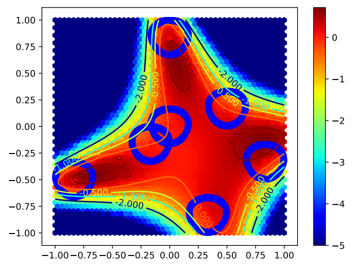
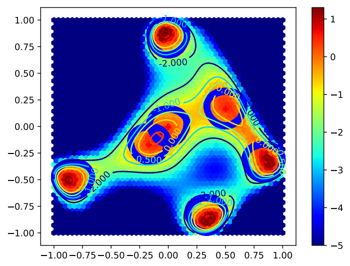
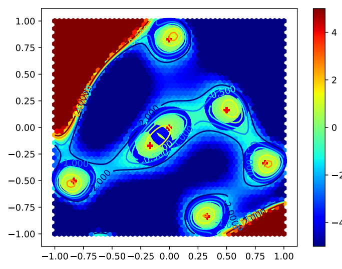

(f) $p = 12$

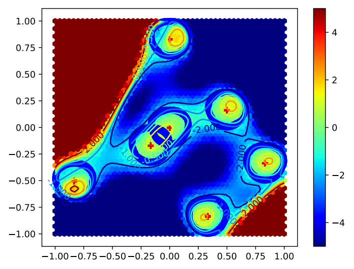
Figure 3: Heat map of heart.npz


(a) $p = 2$


(b) $p = 4$


(c) $p = 6$


(d) $p = 8$


(e) $p = 10$

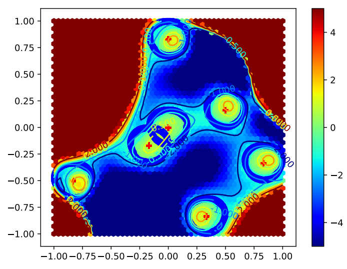

(f) $p = 12$

Figure 4: Heat map of asymmetric.npz

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 in kernel ridge regression due to the *difference of the constant terms* of the polynomial**, but your plot should be similar.

```python
#!/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]
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, 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')
    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("D = {: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 = "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 =  820907.453998
p =  2   train_error = 0.995537  validation_error = 1.001056  cond =  989295.364900
p =  3   train_error = 0.992699  validation_error = 1.019351  cond = 1343199.616779
p =  4   train_error = 0.943011  validation_error = 0.997922  cond = 1972288.839522
p =  5   train_error = 0.935539  validation_error = 1.029207  cond = 3045737.405098
p =  6   train_error = 0.512533  validation_error = 0.548791  cond = 4863407.600553
p =  7   train_error = 0.507675  validation_error = 0.549238  cond = 7955964.187381
p =  8   train_error = 0.088668  validation_error = 0.102742  cond = 13269828.224191
p =  9   train_error = 0.083278  validation_error = 0.098597  cond = 22510772.815029
p = 10   train_error = 0.057527  validation_error = 0.071359  cond = 38786251.862798
p = 11   train_error = 0.020365  validation_error = 0.026058  cond = 67809304.792464
p = 12   train_error = 0.010860  validation_error = 0.014138  cond = 120156706.327969
p = 13   train_error = 0.008167  validation_error = 0.010614  cond = 215511722.324505
p = 14   train_error = 0.005374  validation_error = 0.007011  cond = 390664244.209680
p = 15   train_error = 0.002760  validation_error = 0.003579  cond = 714664126.824285
p = 16   train_error = 0.001716  validation_error = 0.002217  cond = 1317683791.519599
```

```
###### heart.npz ######
p =  1   train_error = 0.962643  validation_error = 0.959952  cond =  803209.748495
p =  2   train_error = 0.236718  validation_error = 0.189836  cond =  840666.818390
p =  3   train_error = 0.115481  validation_error = 0.090806  cond =  939963.690904
p =  4   train_error = 0.012163  validation_error = 0.009084  cond = 1209900.485664
p =  5   train_error = 0.004136  validation_error = 0.003268  cond = 1954958.809849
p =  6   train_error = 0.002388  validation_error = 0.001676  cond = 3689166.714984
p =  7   train_error = 0.001610  validation_error = 0.001161  cond = 7340900.466579
p =  8   train_error = 0.000947  validation_error = 0.000636  cond = 14899859.581048
p =  9   train_error = 0.000438  validation_error = 0.000276  cond = 30564088.440461
p = 10   train_error = 0.000250  validation_error = 0.000169  cond = 63145526.762684
p = 11   train_error = 0.000175  validation_error = 0.000132  cond = 131166168.848124
p = 12   train_error = 0.000137  validation_error = 0.000112  cond = 273641662.796594
p = 13   train_error = 0.000113  validation_error = 0.000097  cond = 572924846.478053
p = 14   train_error = 0.000096  validation_error = 0.000085  cond = 1203162667.273223
p = 15   train_error = 0.000083  validation_error = 0.000076  cond = 2533216571.727164
p = 16   train_error = 0.000074  validation_error = 0.000069  cond = 5345516694.273760
###### 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
```

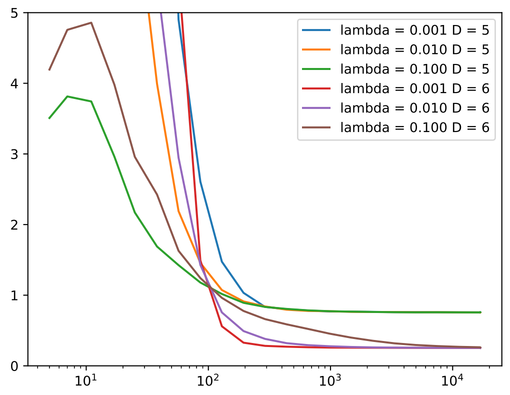
```python
#!/usr/bin/env python3

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:]

```

```python
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, 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')
    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("D = {: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}".
```

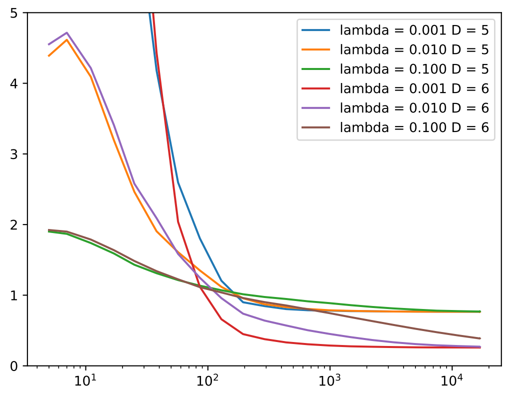
```
                format(sigma, error_train, error_valid, np.linalg.cond(K)))
        # heatmap(
        #     lambda x, y: rbf_kernel(np.column_stack([x, y]), X_train.T, sigma) @ coeff,
        #     fname="result/heart_RBF0_%4f.pdf" % sigma)


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

```
#!/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

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

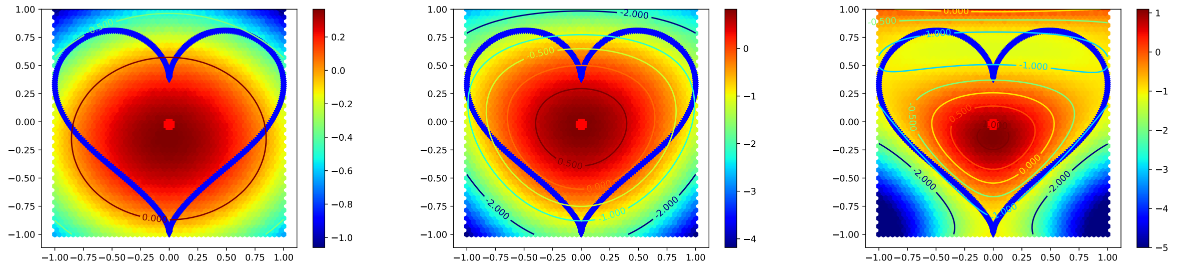
Implement the RBF kernel function for kernel ridge regression to fit the dataset `heart.npz`. Use the regularization term $\lambda = 0.001$. **Report the average squared loss, visualize your result and attach the heatmap plots for the fitted functions over the 2D domain for $\sigma \in \{10, 3, 1, 0.3, 0.1, 0.03\}$ 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 = 0.995883 validation_error = 1.000416 cond =  814877.550220
sigma =  3.000 train_error = 0.993453 validation_error = 1.009423 cond =  765939.001583
sigma =  1.000 train_error = 0.942752 validation_error = 1.023007 cond =  489921.901894
sigma =  0.300 train_error = 0.001661 validation_error = 0.001883 cond =  109532.702805
sigma =  0.100 train_error = 0.000063 validation_error = 0.000160 cond =   57394.488759
sigma =  0.030 train_error = 0.000003 validation_error = 0.003013 cond =   47589.723281
```

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.

(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.**


(a) $\sigma = 10$ (b) $\sigma = 3$ (c) $\sigma = 1$

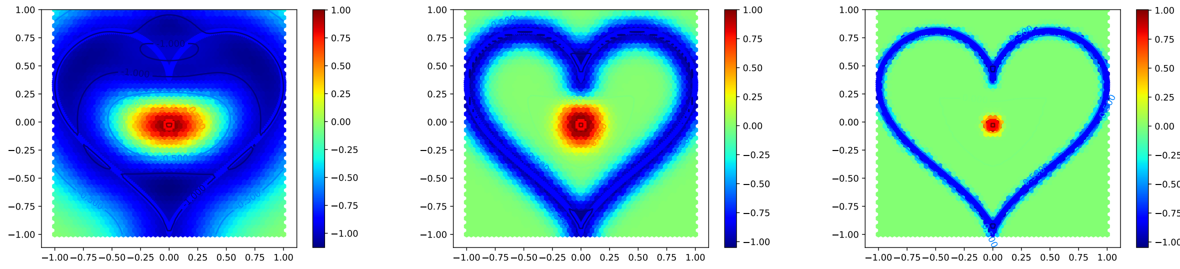

(d) $\sigma = 0.3$ (e) $\sigma = 0.1$ (f) $\sigma = 0.03$

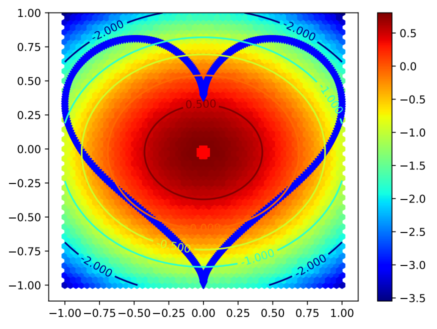
Figure 6: Heatmap of `heart.npz`

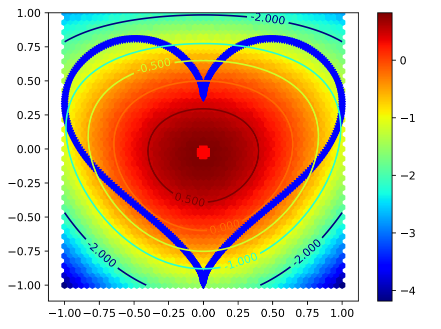
Solution: For polynomial regression, our non-kernelized method is more efficient in practice. This matches our prediction, where the computational complexity of non-kernelized method is smaller when $d \ll n$. However, this problem is in 2D space. When the feature space is in $\mathbb{R}^{20}$ for example,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

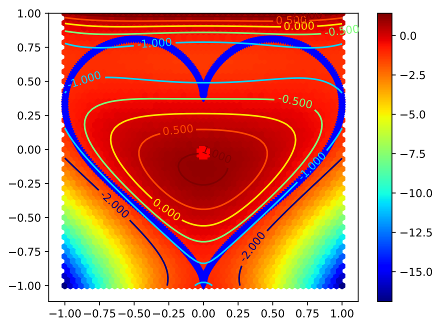
In summary, we want to use kernelized method when the feature space is much larger than the number pf samples, or when the dimension of the feature space is infinite like in RBF, in which it is impossible to use the non-kernelized method.

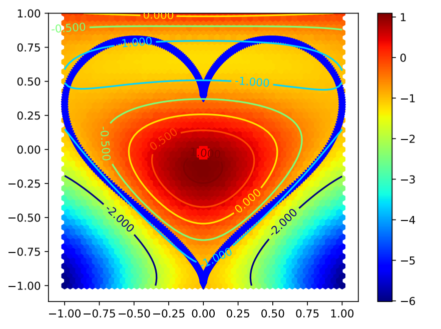
(g) Disable the `clip` option in the provided `heatmap` function and redraw the heatmap plots for the functions learned by the polynomial kernel and RBF kernel. Experiment on the provided datasets and **describe one potential problem of the polynomial kernel related to what you see here.** Does the RBF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

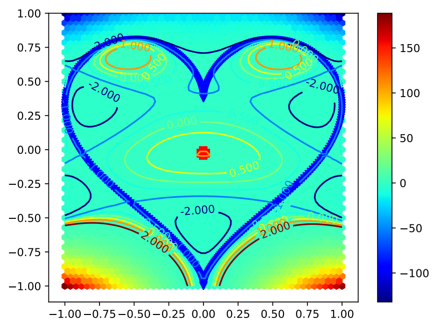
Solution: Figure 7 shows the comparison between polynomial kernel (left) and RBF kernel (right) with similar fitting error. One most observable problem is that the polynomial kernel does not extrapolate well, i.e., you will see absurdly large number outside the domain of training data. On the other hand,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


(a) $p = 2$


(b) $\sigma = 3$


(c) $p = 4$


(d) $\sigma = 1$


(e) $p = 12$

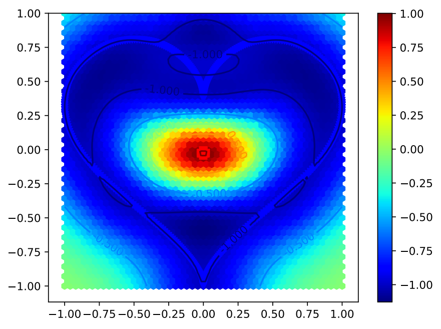

(f) $\sigma = 0.3$

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.

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.