Question 1. Getting Started

Read through this page carefully. You may typeset your homework in latex or submit neatly handwritten/scanned solutions. Please start each question on a new page. Delivera

1. Submit a PDF of your writeup, **with an appendix for your code**, to assignment on Gradescope, "¡¡title¿¿ Write-Up". If there are graphs, include those graphs in the
2. If there is code, submit all code needed to reproduce your results, "¡¡title¿¿ Code".
3. If there is a test set, submit your test set evaluation results, "¡¡title¿¿ Test Set".

After you've submitted your homework, watch out for the self-grade form.

(a) Who else did you work with on this homework? In case of course events, just describe the group. How did you work on this homework? Any comments about the homework?

I worked with Weiran Liu. I fixed the Latex issue first and then review some of the materials we learned and then started working.

(b) Please copy the following statement and sign next to it. We just want to make it *extra* clear so that no one inadverdently cheats.

I certify that all solutions are entirely in my words and that I have not looked at another student's solutions. I have credited all external sources in this write up.

I certify that all solutions are entirely in my words and that I have not looked at another student's solutions. I have credited all external sources in this write up. ——Huanjie Sheng

Question 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 μ and covariance matrix Σ). 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 $\mathbb{X}$ 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 $\vec{x}_1, \ldots, \vec{x}_n \in \mathbb{R}^d$ from X given Σ and μ.**

$$P(\Sigma, \vec{\mu}|\vec{x}_1, \ldots, \vec{x}_n) = \frac{1}{(\sqrt{2\pi|\Sigma|})^n} e^{\frac{1}{2}\sum_{i=1}^n ((\vec{x_i}-\vec{\mu})^\top \Sigma^{-1}(\vec{x_i}-\vec{\mu}))}$$

$$\begin{aligned}
& \ln P(\Sigma, \vec{\mu}|\vec{x}_1, \ldots, \vec{x}_n) \\
= \; & -n\ln(\sqrt{2\pi|\Sigma|}) + \tfrac{1}{2}\textstyle\sum_{i=1}^n ((\vec{x_i}-\mu)^\top\Sigma^{-1}(\vec{x_i}-\mu)) \\
= \; & -\tfrac{n}{2}(\ln(2\pi) - \ln|\Sigma|) + \tfrac{1}{2} tr((\tilde{X} - \vec{1}\vec{\mu}^\top)\Sigma^{-1}(\tilde{X} - \vec{1}\vec{\mu}^\top)^\top) \\
= \; & -\tfrac{n}{2}(\ln(2\pi) - \ln|\Sigma|) + \tfrac{1}{2}(tr(\tilde{X}\Sigma^{-1}\tilde{X}^\top) - tr(\vec{1}\vec{\mu}^\top\Sigma^{-1}\tilde{X}^\top) + tr(\vec{1}\vec{\mu}^\top\Sigma^{-1}\vec{\mu}\vec{1}^\top) - tr(\tilde{X}\Sigma^{-1}\vec{\mu}\vec{1}^\top))
\end{aligned}$$

$$\text{where } \tilde{X} = \begin{bmatrix} \vec{x_1} & \vec{x_2} & \ldots & \vec{x_n} \end{bmatrix}$$

(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.

We find the mean μ first:

$$\begin{aligned}
& \nabla_{\vec{\mu}} \ln P(\Sigma, \vec{\mu}|\vec{x}_1, \ldots, \vec{x}_n) \\
= \; & 0 + 0 + \tfrac{1}{2}(0 - \Sigma^{-1}\tilde{X}^\top\vec{1} + 2\Sigma^{-1}\vec{\mu}\vec{1}^\top\vec{1} - (\tilde{X}\Sigma^{-1})^\top\vec{1}) \\
= \; & \Sigma^{-1}\vec{\mu}\vec{1}^\top\vec{1} - \Sigma^{-1}\tilde{X}^\top\vec{1}
\end{aligned}$$

$$\begin{aligned}
\nabla_{\vec{\mu}} \ln P(\Sigma, \vec{\mu}|\vec{x}_1, \ldots, \vec{x}_n) &= 0 \\
\Sigma^{-1}\vec{\mu}\vec{1}^\top\vec{1} - \Sigma^{-1}\tilde{X}^\top\vec{1} &= 0 \\
\Sigma^{-1}(\vec{\mu}\vec{1}^\top\vec{1} - \tilde{X}^\top\vec{1}) &= 0 \\
\vec{\mu}\vec{1}^\top\vec{1} &= \tilde{X}^\top\vec{1} \\
\vec{\mu} &= \tfrac{1}{n}\tilde{X}^\top\vec{1}
\end{aligned}$$

Now we look at the variance Σ:

$$\begin{aligned}
& \nabla_\Sigma \ln P(\Sigma, \vec{\mu}|\vec{x}_1, \ldots, \vec{x}_n) \\
= \; & 0 + \tfrac{n}{2}\tfrac{1}{|\Sigma|}|\Sigma|(\Sigma^{-1})^\top + \tfrac{1}{2}(\Sigma^{-1}\tilde{X}^\top\tilde{X}\Sigma^{-1} - \Sigma^{-1}\vec{\mu}\vec{1}^\top\tilde{X}\Sigma^{-1} + \Sigma^{-1}\vec{\mu}\vec{1}^\top\vec{1}\vec{\mu}^\top\Sigma^{-1} - \Sigma^{-1}\tilde{X}^\top\vec{1}\vec{\mu}^\top\Sigma^{-1}) \\
= \; & \tfrac{n}{2}\Sigma^{-1} + \tfrac{1}{2}(\Sigma^{-1}(\tilde{X}^\top - \vec{\mu}\vec{1}^\top)(\tilde{X} - \vec{1}\vec{\mu}^\top)\Sigma^{-1})
\end{aligned}$$

$$\begin{aligned}
\nabla_\Sigma \ln P(\Sigma, \vec{\mu}|\vec{x}_1, \ldots, \vec{x}_n) &= 0 \\
n\Sigma^{-1} + \Sigma^{-1}(\tilde{X}^\top - \vec{\mu}\vec{1}^\top)(\tilde{X} - \vec{1}\vec{\mu}^\top)\Sigma^{-1} &= 0 \\
\Sigma^{-1}(n\mathbb{I} + (\tilde{X}^\top - \vec{\mu}\vec{1}^\top)(\tilde{X} - \vec{1}\vec{\mu}^\top)\Sigma^{-1}) &= 0 \\
(\tilde{X}^\top - \vec{\mu}\vec{1}^\top)(\tilde{X} - \vec{1}\vec{\mu}^\top) &= n\Sigma \\
\tfrac{1}{n}(\tilde{X}^\top - \vec{\mu}\vec{1}^\top)(\tilde{X} - \vec{1}\vec{\mu}^\top) &= \Sigma
\end{aligned}$$

We substitute the solution to μ above:

$$\begin{aligned}
\Sigma &= \tfrac{1}{n}(\tilde{X}^\top - \vec{\mu}\vec{1}^\top)(\tilde{X} - \vec{1}\vec{\mu}^\top) \\
\Sigma &= \tfrac{1}{n}(\tilde{X}^\top - \tfrac{1}{n}\tilde{X}^\top\vec{1}\vec{1}^\top)(\tilde{X} - \vec{1}(\tfrac{1}{n}\tilde{X}^\top\vec{1})^\top) \\
\Sigma &= \tfrac{1}{n}(\tilde{X}^\top - \tfrac{1}{n}\tilde{X}^\top\vec{1}\vec{1}^\top)(\tilde{X} - \tfrac{1}{n}\vec{1}\vec{1}^\top\tilde{X})
\end{aligned}$$

We can see that the MLE solution to multivariate Gaussian distribution is just the combination of the MLE solutions to all univariate Gaussian distributions.

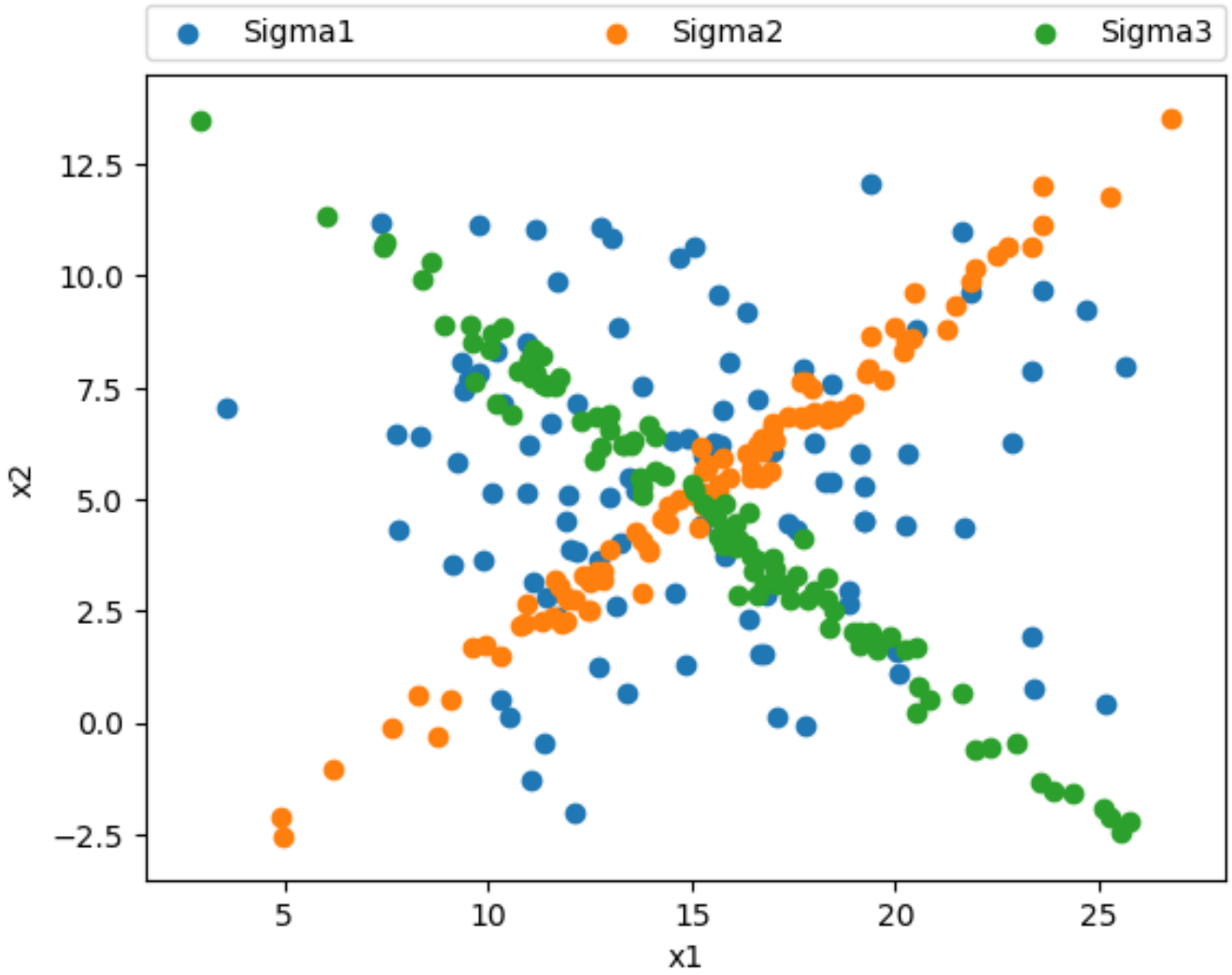
(c) Use the following code to sample from a two-dimensional Multivariate Gaussian and plot the samples:

```
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:

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

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.



$$\Sigma_1 = \begin{bmatrix} 20 & 0 \\ 0 & 10 \end{bmatrix} \quad \hat{\mu} = \begin{bmatrix} 14.99571712 \\ 5.45150579 \end{bmatrix} \quad \hat{\Sigma} = \begin{bmatrix} 20.88795883 & -0.35242617 \\ -0.35242617 & 10.31880934 \end{bmatrix}$$

$$\Sigma_2 = \begin{bmatrix} 20 & 14 \\ 14 & 10 \end{bmatrix} \quad \hat{\mu} = \begin{bmatrix} 15.66296604 \\ 5.41663562 \end{bmatrix} \quad \hat{\Sigma} = \begin{bmatrix} 18.4216983 & 13.05870625 \\ 13.05870625 & 9.42714322 \end{bmatrix}$$

$$\Sigma_3 = \begin{bmatrix} 20 & -14 \\ -14 & 10 \end{bmatrix} \quad \hat{\mu} = \begin{bmatrix} 15.48017663 \\ 4.61543441 \end{bmatrix} \quad \hat{\Sigma} = \begin{bmatrix} 21.44889434 & -15.20031873 \\ -15.20031873 & 10.98160867 \end{bmatrix}$$

```
import numpy as np
import matplotlib.pyplot as plt
import os


def bmatrix(a):
"""Returns a LaTeX bmatrix
Retrieved from https://stackoverflow.com/questions/17129290/numpy-2d-and-1d-array-to-latex-bmatrix
:a: numpy array
:returns: LaTeX bmatrix as a string
"""
if len(a.shape) > 2:
raise ValueError('bmatrix can at most display two dimensions')
lines = str(a).replace('[', '').replace(']', '').splitlines()
```

```
rv = [r'\begin{bmatrix}']
rv += [' ' + ' & '.join(l.split()) + r'\\' for l in lines]
rv += [r'\end{bmatrix}']
return '\n'.join(rv)


mu = [15, 5]
all_sigmas = np.asarray(([[20, 0], [0, 10]], [[20, 14], [14, 10]], [[20, -14], [-14, 10]]))
np.random.seed(0)
n = 100
vec_one = np.ones((n, 1))
for i in range(len(all_sigmas)):
sigma = all_sigmas[i]
samples = np.random.multivariate_normal(mu, sigma, size=n)

e_mu = samples.T.dot(vec_one) / n
e_sigma = (samples.T - e_mu.dot(vec_one.T)).dot(samples - vec_one.dot(e_mu.T)) / n

print('\[')
print('\Sigma_' + str(i + 1) + '=')
print(bmatrix(sigma))
print('\hat{\mu}=')
print(bmatrix(e_mu))
print('\hat{\Sigma}=')
print(bmatrix(e_sigma))
print('\]\n')
plt.scatter(samples[:, 0], samples[:, 1], label='Sigma' + str(i + 1))

lgd = plt.legend(bbox_to_anchor=(0., 1.02, 1., .102), loc=3, ncol=3, mode="expand", borderaxespad=0.)
plt.xlabel('x1')
plt.ylabel('x2')
plt.show(block=False)
filename = 'Figure_2c.png'
save_path = os.path.join('.', filename)
plt.savefig(save_path, bbox_extra_artists=(lgd,), bbox_inches='tight')
```

Question 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 $\vec{x} \in \mathbb{R}^d$ be a d-dimensional vector and $Y \in \mathbb{R}$ be a one-dimensional random variable. Assume a linear model: $Y = \vec{x}^\top \vec{w} + Z$ where $Z \in \mathbb{R}$ is a standard Gaussian random variable $Z \sim N(0,1)$ and $\vec{w} \in \mathbb{R}^d$ is a d-dimensional Gaussian random vector $\vec{w} \sim N(0, \mathbf{\Sigma})$. $\mathbf{\Sigma}$ is a known symmetric positive definite covariance matrix. Note that $\vec{w}$ is independent of the observation noise. **What is the conditional distribution of Y given $\vec{x}$ and $\vec{w}$?**

As we learned in the lecture:

$$Y = \vec{x}^\top \vec{w} + Z$$

Because $\vec{x}$ and $\vec{w}$ are given, we can treat them as constants:

$$Y \sim N(\vec{x}^\top \vec{w}, 1)$$

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

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

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

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

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

(1)Let $X = \begin{bmatrix} \vec{x_1}^\top \\ \vec{x_2}^\top \\ \vdots \\ \vec{x_n}^\top \end{bmatrix}$

$$\begin{aligned} P(\vec{w}|(X,\vec{Y})) &\propto Ce^{(Y-X\vec{w})^\top(Y-X\vec{w})+\vec{w}^\top\Sigma^{-1}\vec{w}} \\ &\propto Ce^{Y^\top Y-\vec{w}^\top X^\top Y-Y^\top X\vec{w}+\vec{w}^\top X^\top X\vec{w}+\vec{w}^\top\Sigma^{-1}\vec{w}} \\ &\propto C'e^{\frac{1}{2}\vec{w}^\top(\top X^\top X+\Sigma^{-1})\vec{w}-Y^\top X\vec{w}} \end{aligned}$$

Here C abd C' are some constants based on the training data. Now we can use the hint above:

$$P(\vec{w}|(X,Y)) \propto N((X^\top X+\Sigma^{-1})^{-1}X^\top Y,?)$$

Oops, the hint doesn't tell me what the variance is. Let's take a look the standard multivariate Gaussian distribution PDF:

$$P(X=x)=\frac{1}{\sqrt{2\pi|\Sigma|}}e^{\frac{1}{2}(x-\mu)^\top\Sigma^{-1}(x-\mu)}$$

We notice that the coefficient of the quadratic term is the inverse of the covariance matrix. Can we you this without proving it? Yes, of course. Therefore, we have:

$$P(\vec{w}|(X,Y)) \propto N((X^\top X+\Sigma^{-1})^{-1}X^\top Y,(X^\top X+\Sigma^{-1})^{-1})$$

(2) The MAP estimator of w is given by the mean/median of the posterior distribution:

$$\hat{w}=(X^\top X+\Sigma^{-1})^{-1}X^\top Y$$

(3) Recall what Tikhonov regularization is

$$\vec{w}=\arg\min_{\vec{w}}\frac{1}{2}\|\vec{y}-\mathbb{X}\vec{w}\|_2^2+\lambda\|\Gamma\vec{w}\|_2^2$$

And we have derived and solved $\vec{w}$ to be:

$$\hat{\vec{w}}=(\mathbb{X}^\top\mathbb{X}+2\lambda\Gamma^\top\Gamma)^{-1}\mathbb{X}^\top\vec{y}$$

You can see that the formulas are very similar. Because we know Σ is a positive semidefinite symmetric matrix and so does its inverse Σ^{-1}, we can always find a decomposition such that $\Sigma^{-1}=A^\top A$ and $A=\sqrt{2\lambda}\Gamma$. If the off-diagonal entries of the covariance matrix Σ are all zeros and all the diagonal entries have the same value, Tikhonov regularization can be simplified into ridge regression. In other words, if all $\vec{w}$s are i.i.d. Gaussian, Tikhonov regularization becomes simple ridge regression.

(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. $\mathbb{Z}$ is no longer $N(\vec{0},\mathbb{I}_n)$ but instead distributed as $N(\mu_z,\mathbf{\Sigma}_z)$ for some mean μ_z and some covariance $\mathbf{\Sigma}_z$ (still independent of the parameter $\vec{w}$). **Derive the posterior distribution of $\vec{w}$ 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 $\mathbb{Z}$ as a function of a standard normal Gaussian vector $\vec{V}\sim N(\vec{0},\mathbb{I}_n)$ and use the result in (b) for an equivalent model of the form $\widetilde{\mathbb{Y}}=\widetilde{\mathbb{X}}\vec{w}+\vec{V}$.

We can write $\mathbb{Z}$ as a function of a standard normal Gaussian vector $\vec{V} \sim N(\vec{0}, \mathbb{I}_n)$. How do we achieve this? Whitening! Recall that if $\vec{V} \sim N(0,1)$ then $A\vec{V} \sim N(0, AA^T)$ Let's decompose $\mathbf{\Sigma}_z$

$$\mathbf{\Sigma}_z = \Gamma_z^\top \Gamma_z \quad \mathbf{\Sigma}_z^{-1} = \Gamma_z^{-1} \Gamma_z^{-\top}$$

Now we can apply the following transformation on $\vec{V}$:

$$\mathbb{Z} = \Gamma_z^\top \vec{V} + \mu_z$$

$$\vec{V} = \Gamma_z^{-\top}(\mathbb{Z} - \mu_z)$$

We go back and take a look at our original formula:

$$\begin{aligned} Y &= X\vec{w} + \mathbb{Z} \\ Y - \mu_z &= X\vec{w} + \mathbb{Z} - \mu_z \\ \Gamma_z^{-\top}(Y - \mu_z) &= \Gamma_z^{-\top} X\vec{w} + \Gamma_z^{-\top}(\mathbb{Z} - \mu_z) \\ \widetilde{\mathbb{Y}} &= \widetilde{\mathbb{X}}\vec{w} + \vec{V} \end{aligned}$$

$$\text{where} \quad \widetilde{\mathbb{Y}} = \Gamma_z^{-\top}(Y - \mu_z) \quad , \quad \widetilde{\mathbb{X}} = \Gamma_z^{-\top} X$$

Here we know the solution to $\vec{w}$ using $\widetilde{\mathbb{X}}$ and $\widetilde{\mathbb{Y}}$ because $\vec{V} \sim N(\vec{0}, \mathbb{I})$. Copy the solution to part (b) down here:

$$P(\vec{w}|(\widetilde{\mathbb{X}}, \widetilde{\mathbb{Y}})) \sim N((\widetilde{\mathbb{X}}^\top \widetilde{\mathbb{X}} + \Sigma^{-1})^{-1} \widetilde{\mathbb{X}}^\top \widetilde{\mathbb{Y}}, (\widetilde{\mathbb{X}}^\top \widetilde{\mathbb{X}} + \Sigma^{-1})^{-1})$$

Calculate the variance first because it's simpler:

$$\begin{aligned} & (\widetilde{\mathbb{X}}^\top \widetilde{\mathbb{X}} + \Sigma^{-1})^{-1} \\ = \; & ((\Gamma_z^{-\top} X)^\top \Gamma_z^{-\top} X + \Sigma^{-1})^{-1} \\ = \; & (X^\top \Gamma_z^{-1} \Gamma_z^{-\top} X + \Sigma^{-1})^{-1} \\ = \; & (X^\top (\Gamma_z^\top \Gamma_z)^{-1} X + \Sigma^{-1})^{-1} \\ = \; & (X^\top \Sigma_z^{-1} X + \Sigma^{-1})^{-1} \end{aligned}$$

Now we do the second part of the mean:

$$\begin{aligned} & \widetilde{\mathbb{X}}^\top \widetilde{\mathbb{Y}} \\ = \; & X^\top \Gamma_z^{-1} \Gamma_z^{-\top} (Y - \mu_z) \\ = \; & X^\top (\Gamma_z^\top \Gamma_z)^{-1} (Y - \mu_z) \\ = \; & X^\top \Sigma_z^{-1} (Y - \mu_z) \end{aligned}$$

Finally, we got the solution! Just copy and paste:

$$P(\vec{w}|(X,Y)) \sim N((X^\top \Sigma_z^{-1} X + \Sigma^{-1})^{-1} X^\top \Sigma_z^{-1} (Y - \mu_z), (X^\top \Sigma_z^{-1} X + \Sigma^{-1})^{-1})$$

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

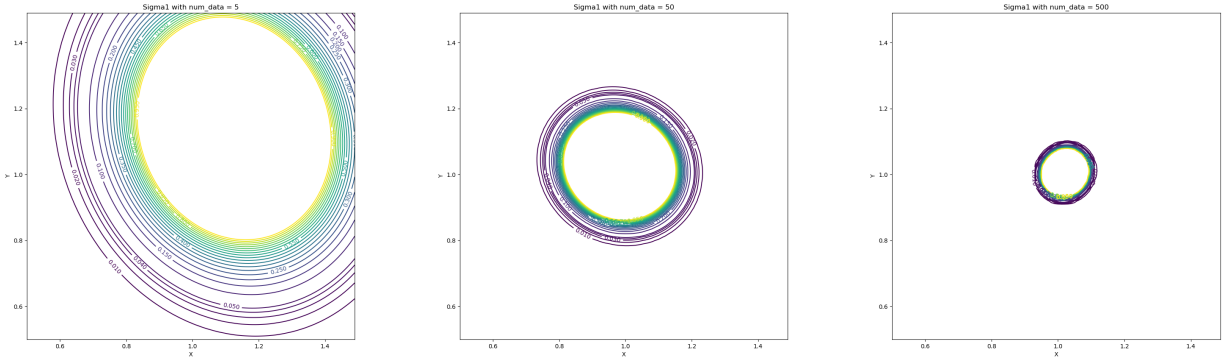
$$\mathbf{\Sigma}_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}; \quad \mathbf{\Sigma}_2 = \begin{bmatrix} 1 & 0.25 \\ 0.25 & 1 \end{bmatrix}; \quad \mathbf{\Sigma}_3 = \begin{bmatrix} 1 & 0.9 \\ 0.9 & 1 \end{bmatrix};$$
$$\mathbf{\Sigma}_4 = \begin{bmatrix} 1 & -0.25 \\ -0.25 & 1 \end{bmatrix}; \quad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

Under the priors above, the coordinates of the (random) vector $\vec{w}$ 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.

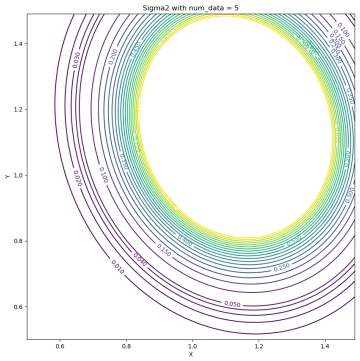
Using the starter code, generate data points (in the range [5, 500]) $Y = x_1 + x_2 + Z$ with $x_1, x_2 \sim N(0, 5)$ and $Z \sim N(0, 1)$ as training data (here, the true $\vec{w}$ is thus $\begin{bmatrix} 1 & 1 \end{bmatrix}^T$). Note that the randomness of x_i here is only for the generation of the 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 $\vec{w}$ for various settings of $\mathbf{\Sigma}$ 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?**

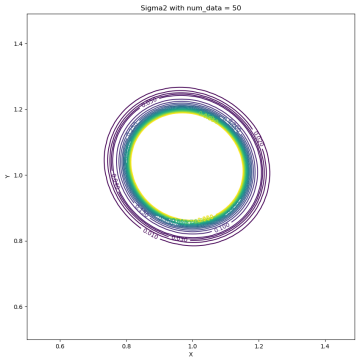
We see that the larger the sample size, the better we can estimate w. However, the prior doesn't affect the result very much. From left to right: Sample size = 5, 50, 500

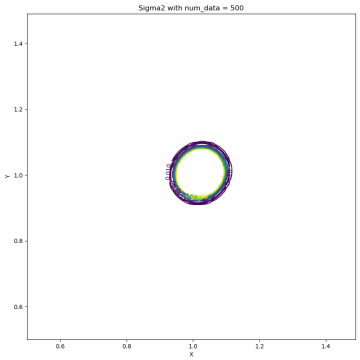
$$\mathbf{\Sigma}_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$



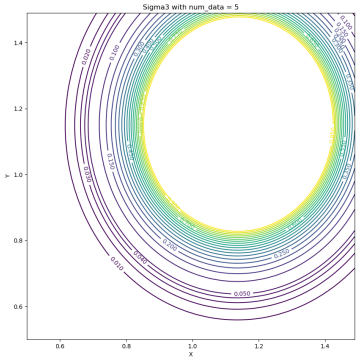
$$\mathbf{\Sigma}_2 = \begin{bmatrix} 1 & 0.25 \\ 0.25 & 1 \end{bmatrix}$$

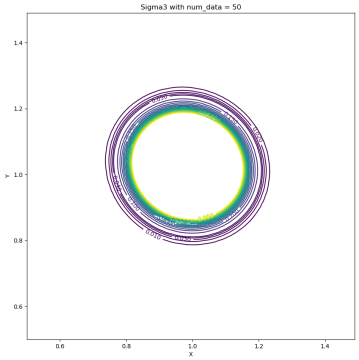


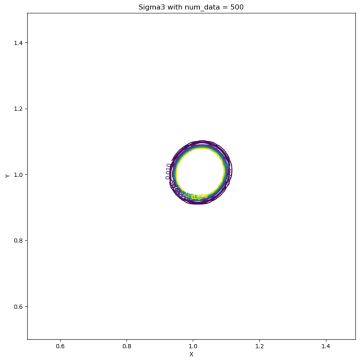




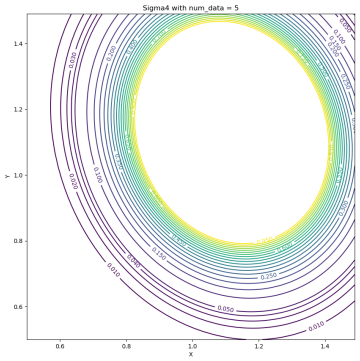
$$\mathbf{\Sigma}_3 = \begin{bmatrix} 1 & 0.9 \\ 0.9 & 1 \end{bmatrix}$$

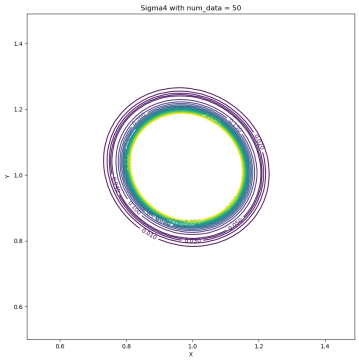


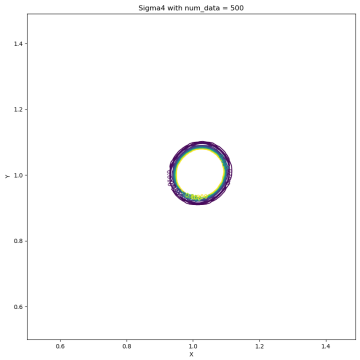




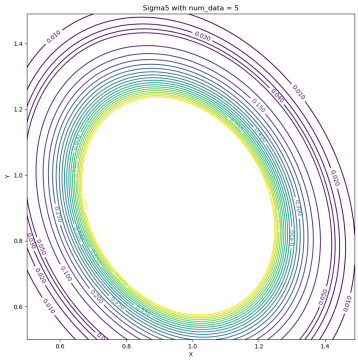
$$\mathbf{\Sigma}_4 = \begin{bmatrix} 1 & -0.25 \\ -0.25 & 1 \end{bmatrix}$$

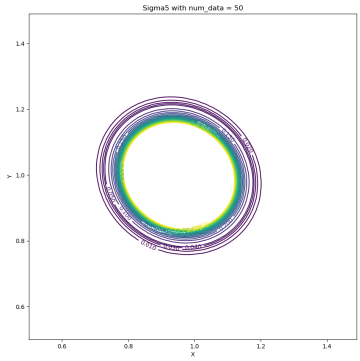


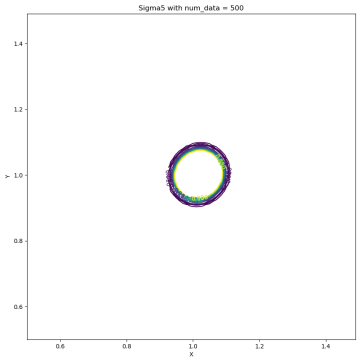




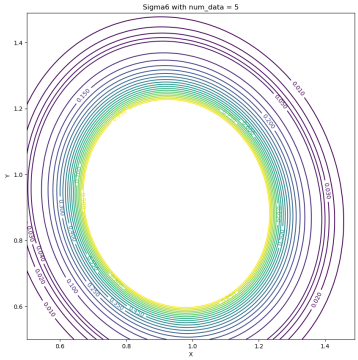
$$\mathbf{\Sigma}_5 = \begin{bmatrix} 1 & -0.9 \\ -0.9 & 1 \end{bmatrix}$$

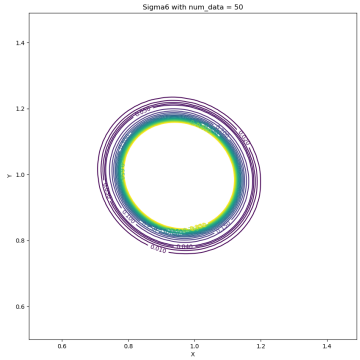


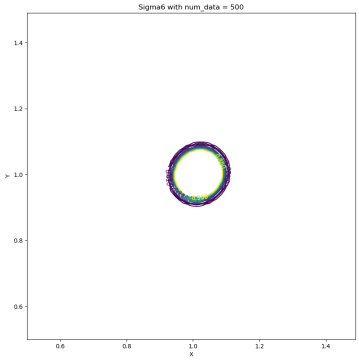




$$\mathbf{\Sigma}_6 = \begin{bmatrix} 0.1 & 0 \\ 0 & 0.1 \end{bmatrix}$$







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

np.random.seed(0)


def generate_data(n):
"""
This function generates data of size n.
"""
# TODO implement this
sigma_x = 5
sigma_z = 1
X = np.random.normal(0, np.sqrt(sigma_x), (n, 2))
Z = np.random.normal(0, np.sqrt(sigma_z), (n, 1))
y = X.dot(np.ones((2, 1))) + Z
return (X, y)


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

```
def compute_mean_var(X, y, Sigma):
"""
This function computes the mean and variance of the posterior
"""
# TODO implement this
sigma = np.linalg.inv(X.T.dot(X) + (np.linalg.inv(Sigma)))
mu = sigma.dot(X.T).dot(Y)
mux = mu[0, 0]
muy = mu[1, 0]
sigmax = np.sqrt(sigma[0, 0])
sigmay = np.sqrt(sigma[1, 1])
sigmaxy = (sigma[0, 1] + sigma[1, 0]) / 2
return mux, muy, sigmax, sigmay, sigmaxy


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 = [str(i) for i in range(1, 6 + 1)]

for num_data in [5, 50, 500]:
X, Y = generate_data(num_data)
for i, Sigma in enumerate(Sigmas):
mux, muy, sigmax, sigmay, sigmaxy = compute_mean_var(X, Y,
Sigma) # TODO compute the mean and covariance of posterior.

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)

# TODO Generate the function values of bivariate normal.
Z = matplotlib.mlab.bivariate_normal(X_grid, Y_grid, sigmax, sigmay, mux, muy, sigmaxy)

# plot
plt.figure(figsize=(10, 10))
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))
plt.savefig('Figure_3d_Sigma' + names[i] + '_num_data_{}.png'.format(num_data))
```

(e) (Influence of Priors) Generate n training data samples from $Y = x_1 + x_2 + Z$ where again $x_1, x_2 \sim N(0, 5)$ and $Z \sim N(0, 1)$ 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 $\widehat{\vec{w}}$ of the parameter $\vec{w}$ (our prediction for Y given a new data point x_i is then $\widehat{Y} = \widehat{w}_1 x_1 + \widehat{w}_2 x_2$). Specifically, considering $\widehat{\vec{w}}$ now fixed, the expected error for a randomly drawn Y given the true (but unknown) parameter vector $\vec{w} = (1, 1)^\top$ is equal to $\mathbb{E}_Z(\|Y - \widehat{Y}\|^2) = 5(\widehat{w}_1 - 1)^2 + 5(\widehat{w}_2 - 1)^2 + 1$. We call this the *theoretical average test error*.

In practice, the expectation with respect to the true conditional distribution of Y given $\vec{w}$ cannot be computed since the true $\vec{w}$ 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, $\hat{Y}_i = \vec{x}_i^{\top}\vec{\widehat{w}}$ where $\vec{x}_i \in \mathbb{R}^2$ and $\vec{\widehat{w}}$ in your model is the solution to the least square problem with Tikhonov regularization given the training data.

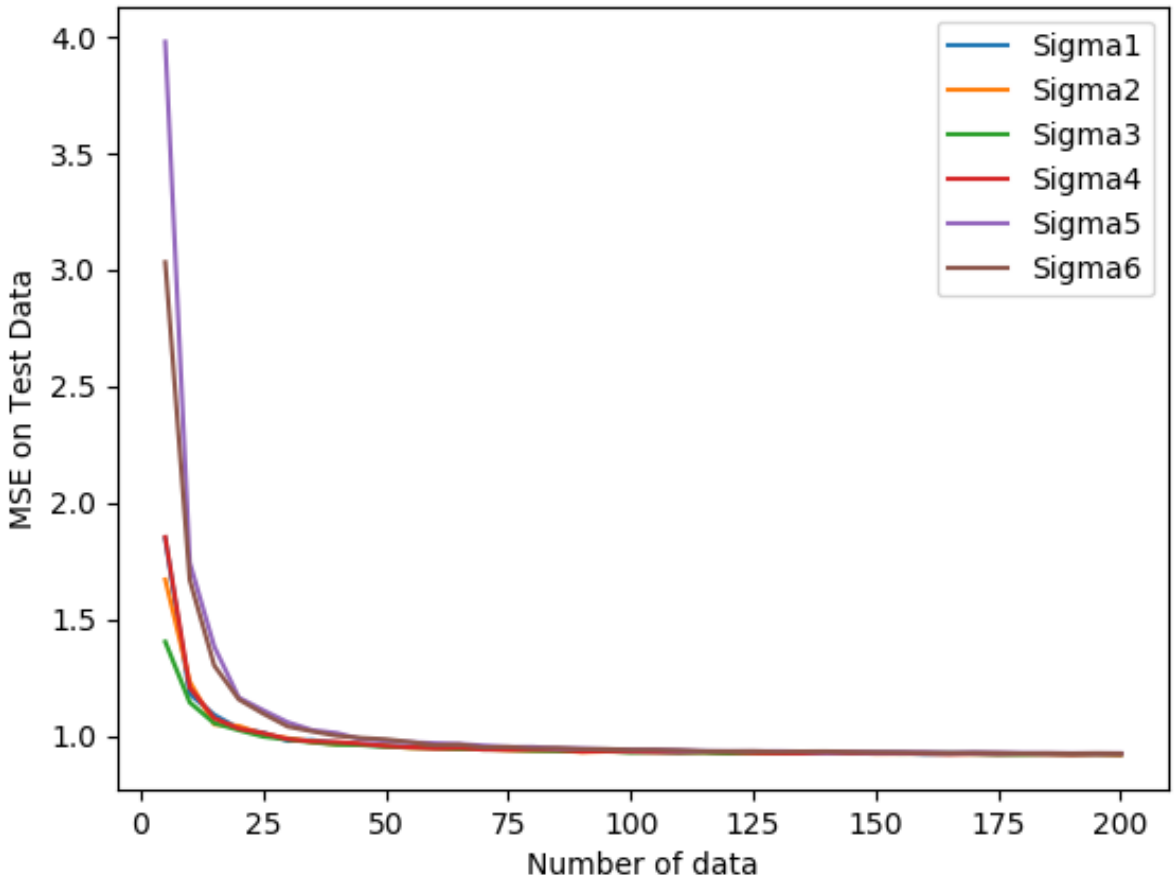
Generate a test set of 500 **data points** (x_i, Y_i) **from the above model. Plot the empirical and theoretical mean square error between** $\hat{Y}_i$ **and** Y_i **over the test data with respect to the size of training data** n **(increase** n **from** 5 **to** 200 **in increments of** 5**).**

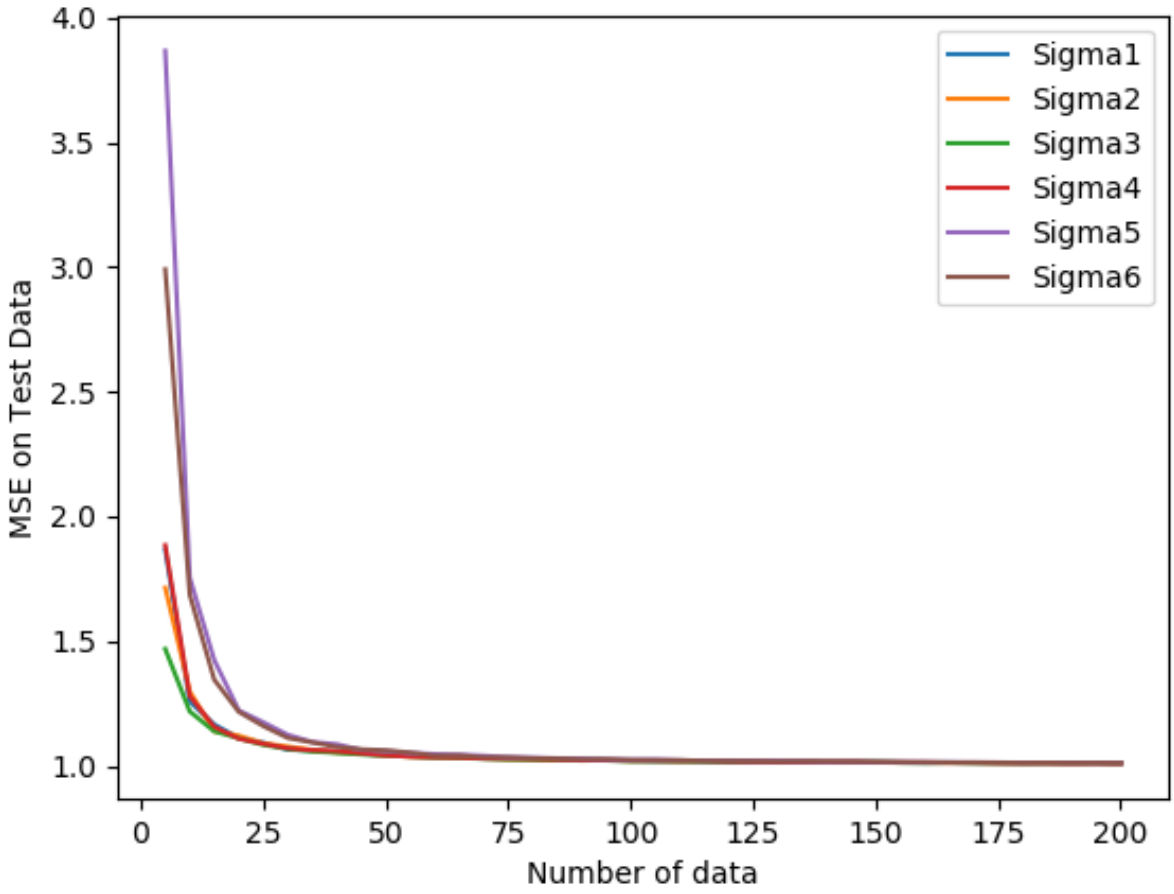
Note: If we just plotted both the empirical and theoretical average test errors with respect to the amount of training data for one "round" or training data generation, the results would still look jagged. In order to give a quantitive statement about the test error with respect to the training data n with a "smoother" plot, what we really want to know is the expectation of the theoretical average test error with respect to $\vec{w}$ and the training samples Y_i, i.e. $\mathbb{E}_{Y_1,\ldots,Y_n}\mathbb{E}_Z(\|Y - \widehat{Y}\|^2)$ (note that in this term only $\widehat{Y}$ depends on $Y_1, \ldots, Y_n$ whereas Y is an independent fresh sample). Consequently, as an approximation, it is worth replicating this 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** $\vec{w} = (1, 1)^{\top}$**? Describe how the influence of different priors changes with the number of data points.**

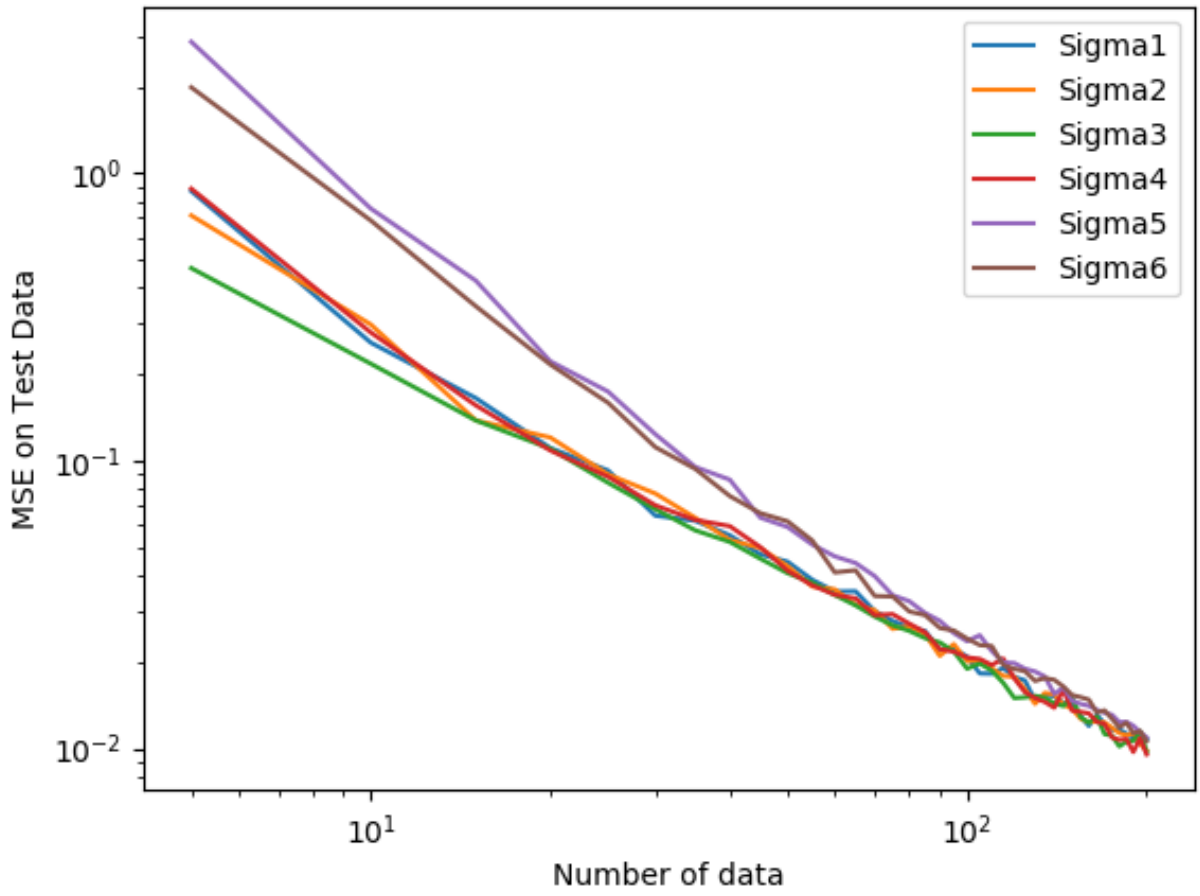
(1) We can see the difference between different prior diminishes as the amount of training data increases.

(2) A prior with a strong positive correlation is considered as a "good" prior and our "good" prior is $Sigma_3$. I choose them because they gave us small errors when the sample size is small.

(3) The influence of different priors decreases with the number of data points


Sigma1
Sigma2
Sigma3
Sigma4
Sigma5
Sigma6
MSE on Test Data
Number of data
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0
25
50
75
100
125
150
175
200


4.0
3.5
3.0
2.5
2.0
1.5
1.0
MSE on Test Data
0
25
50
75
100
125
150
175
200
Number of data
Sigma1
Sigma2
Sigma3
Sigma4
Sigma5
Sigma6




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

np.random.seed(0)
w = [1.0, 1.0]
n_test = 100
n_trains = np.arange(5, 205, 5)
n_trails = 500

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)]

sigma_x = 5
sigma_z = 1


def generate_data(n):
"""
This function generates data of size n.
"""
# TODO implement this
X = np.random.normal(0, np.sqrt(sigma_x), (n, 2))
Z = np.random.normal(0, np.sqrt(sigma_z), (n, 1))
y = X.dot(np.ones((2, 1))) + Z
return (X, y)
```

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


def compute_mse(X, Y, w):
"""
This function computes MSE given data and estimated w.
"""
# TODO implement this
mse = np.mean((Y - X.dot(w)) ** 2)
return mse


def compute_theoretical_mse(w):
"""
This function computes theoretical MSE given estimated w.
"""
theoretical_mse = sigma_x * np.sum((w - 1) ** 2) + sigma_z
# TODO implement this
return theoretical_mse


# 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):
# TODO implement the mses and theoretical_mses
X_train, y_train = generate_data(n_train)
w = tikhonov_regression(X_train, y_train, Sigma)
mses[i, j] = compute_mse(X_test, y_test, w)
theoretical_mses[i, j] = 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('Figure_3e_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('Figure_3e_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('Figure_3e_log_theoretical_MSE.png')
```

Question 4. Kernel Ridge Regression: Theory

In ridge regression, we are given a vector $\vec{y} \in \mathbb{R}^n$ and a matrix $\vec{X} \in \mathbb{R}^{n \times \ell}$, 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 $\vec{X}$ with $\boldsymbol{\Phi} \in \mathbb{R}^{n \times d}$, where $\boldsymbol{\phi}_i^\top = \boldsymbol{\phi}(\vec{x}_i) \in \mathbb{R}^d$. Then we solve a well-defined optimization problem that involves the matrix $\boldsymbol{\Phi}$ and $\vec{y}$ to find the parameters $\vec{w} \in \mathbb{R}^d$. 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

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

Show that the solution of this is equivalent to

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

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

First, we change the constraint condition to $\vec{r} - \vec{X}\vec{w} - \vec{y} = 0$

Second, Lagrangian told us, for a problem like this one we can always find a $\boldsymbol{\alpha}$ such that:

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

While it looks different to what we want to show because it's just a necessary condition but not sufficient. Recall the solution to HW02 Q2a.

Formal derivation: Given a matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$ and a vector $\mathbf{y} \in \mathbb{R}^n$, define the optimization problem

$$\begin{aligned} &\text{minimize } \|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2 \\ &\text{subject to } \|\mathbf{w}\|_2^2 \le \beta^2. \end{aligned} \tag{2}$$

In the first step we show that the above problem is equivalent (i.e. it has the same solution) as the following one:

$$\min_{\mathbf{w}} \max_{\lambda \ge 0} \|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2 + \lambda \cdot \underbrace{(\|\mathbf{w}\|_2^2 - \beta^2)}_{\Delta(\mathbf{w})} \tag{3}$$

We see this as follows:

$$\max_{\lambda \geq 0} \|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2 + \lambda \cdot \Delta(\mathbf{w}) = \|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2 + \begin{cases} \infty & \text{if } \Delta(\mathbf{w}) > 0 \\ 0 & \text{if } \Delta(\mathbf{w}) \leq 0. \end{cases}$$

In the last step we used that if $\Delta(\mathbf{w}) > 0$, the term $\lambda\Delta(\mathbf{w})$ can be made arbitrarily large by making λ large. On the other hand if $\Delta(\mathbf{w}) \leq 0$, the maximization problem is solved by $\lambda = 0$.

It turns out that something called strong duality[1] holds here, which means we can exchange the minimization and maximization in (3) without changing the resulting optimal objective function. We get

$$\max_{\lambda \geq 0} \min_{\mathbf{w}} \|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2 + \lambda \cdot (\|\mathbf{w}\|_2^2 - \beta^2). \tag{4}$$

For every β, the maximum will be attained at some $\lambda^*(\beta)$ and this establishes the equivalence of the constrained problem (2) with the unconstrained problem

$$\min_{\mathbf{w}} \|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2 + \lambda^*(\beta) \cdot (\|\mathbf{w}\|_2^2 - \beta^2). \tag{5}$$

i.e. There is a $\lambda^*(\beta)$ so that the problem has the same resulting $\mathbf{w}$ solution as if we had solved the original constrained optimization problem.

I'm happy to use this solution and argue that this is just another application of Lagrangian function here. Therefore, we have:

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

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

$$\min_{\vec{w},\vec{r}} \max_{\boldsymbol{\alpha}} L(\vec{w}, \vec{r}, \boldsymbol{\alpha}) = \max_{\boldsymbol{\alpha}} \min_{\vec{w},\vec{r}} L(\vec{w}, \vec{r}, \boldsymbol{\alpha}). \tag{2}$$

Argue that the right hand side is equal to

$$\arg\min_{\boldsymbol{\alpha}} \left[\frac{1}{2} \boldsymbol{\alpha}^\top (\mathbb{K} + \lambda \mathbb{I}) \boldsymbol{\alpha} - \lambda \boldsymbol{\alpha}^\top \vec{y} \right] \text{ where } \mathbb{K} = \vec{X}^\top \vec{X} \in \mathbb{R}^{n \times n}. \tag{3}$$

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.

When we set the derivative of Lagrangian function to zero, we will get our desired solution. $\nabla L(\vec{w}, \vec{r}, \boldsymbol{\alpha}) = 0$

$$\begin{cases} \nabla_{\vec{r}} L(\vec{w}, \vec{r}, \boldsymbol{\alpha}) = \vec{r} + \boldsymbol{\alpha} = 0 \\ \nabla_{\vec{w}} L(\vec{w}, \vec{r}, \boldsymbol{\alpha}) = \lambda \vec{w} + \vec{X}^\top \boldsymbol{\alpha} = 0 \end{cases}$$

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

Substitute the above equalities into the original equation gives us

$$\arg\max_{\boldsymbol{\alpha}} \left[\frac{1}{2}\boldsymbol{\alpha}^\top\boldsymbol{\alpha} + \frac{\lambda}{2}\frac{1}{\lambda^2}\boldsymbol{\alpha}^\top\vec{X}\vec{X}^\top\boldsymbol{\alpha} + \boldsymbol{\alpha}^\top(-\boldsymbol{\alpha} - \frac{1}{\lambda}\vec{X}\vec{X}^\top\boldsymbol{\alpha} + \vec{y})\right]$$

$$\arg\max_{\boldsymbol{\alpha}} \left[-\frac{1}{2}\boldsymbol{\alpha}^\top\boldsymbol{\alpha} - \frac{1}{2\lambda}\boldsymbol{\alpha}^\top\vec{X}\vec{X}^\top\boldsymbol{\alpha} + \boldsymbol{\alpha}^\top\vec{y}\right]$$

$$\arg\min_{\boldsymbol{\alpha}} \left[\frac{\lambda}{2}\boldsymbol{\alpha}^\top\boldsymbol{\alpha} + \frac{1}{2}\boldsymbol{\alpha}^\top\vec{X}\vec{X}^\top\boldsymbol{\alpha} - \lambda\boldsymbol{\alpha}^\top\vec{y}\right]$$

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

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

$$\vec{w}^* = \vec{X}\left(\mathbb{K} + \lambda\mathbb{I}\right)^{-1}\vec{y}. \tag{4}$$

We take the derivative of the optimization function above:

$$\begin{aligned} & \nabla\tfrac{1}{2}\boldsymbol{\alpha}^\top(\mathbb{K} + \lambda\mathbb{I})\boldsymbol{\alpha} - \lambda\boldsymbol{\alpha}^\top\vec{y} \\ = \;& (\mathbb{K} + \lambda\mathbb{I})\boldsymbol{\alpha} - \lambda\vec{y} \end{aligned}$$

Setting the gradient to zero gives us:

$$\boldsymbol{\alpha}^* = (\mathbb{K} + \lambda\mathbb{I})^{-1}\lambda\vec{y}$$

Recall the above equalities in part (b) by setting the gradient of Lagrangian function to zero:

$$\vec{w}^* = \frac{1}{\lambda}\vec{X}^\top\boldsymbol{\alpha}^* = \frac{1}{\lambda}\vec{X}^\top(\mathbb{K} + \lambda\mathbb{I})^{-1}\lambda\vec{y}$$

Therefore, we have:

$$\vec{w}^* = \vec{X}^\top\left(\mathbb{K} + \lambda\mathbb{I}\right)^{-1}\vec{y}$$

(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

$$K(\vec{x}_i, \vec{x}_j) = (1 + \vec{x}_i^\top\vec{x}_j)^2, \tag{5}$$

for any $\vec{x}_i, \vec{x}_j \in \mathbb{R}^d$. Given a dataset $(\vec{x}_i, y_i)$ 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 $d = 2$ given the right choice of Tikhonov regularization for the polynomial regression.** That is, show for any new point $\vec{x}$ 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:

$$\vec{A}(a\mathbb{I}_d + \vec{A}^\top \vec{A})^{-1} = (a\mathbb{I} + \vec{A}\vec{A}^\top)^{-1}\vec{A}, \tag{6}$$

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

Copy and paste the solution to Tikhonov regularization here (As is pointed out on Piazza, the constant 2 is removed here which is different from HW02):

$$\hat{\vec{w}} = (\mathbb{X}^\top \mathbb{X} + \lambda \Gamma^\top \Gamma)^{-1} \mathbb{X}^\top \vec{y}$$

Compared to the solution above:

$$\vec{w}^* = \vec{X}^\top \left(\vec{X}\vec{X}^\top + \lambda \mathbb{I}\right)^{-1} \vec{y}$$

For d=2, we have

$$\begin{aligned}
& K(\vec{x}_i, \vec{x}_j) \\
= \;& (1 + \vec{x}_i^\top \vec{x}_j)^2 \\
= \;& (1 + (x_{i1}x_{j1} + x_{i2}x_{j2}))^2 \\
= \;& x_{i1}^2 x_{j1}^2 + x_{i2}^2 x_{j2}^2 + 2x_{i1}x_{j1}x_{i2}x_{j2} + 2x_{i1}x_{j1} + 2x_{i2}\vec{x}_{j2} + 1 \\
= \;& \begin{bmatrix} x_{i1}^2 & x_{i2}^2 & \sqrt{2}x_{i1}x_{i2} & \sqrt{2}x_{i1} & \sqrt{2}x_{i2} & 1 \end{bmatrix} \begin{bmatrix} x_{j1}^2 \\ x_{j2}^2 \\ \sqrt{2}x_{j1}x_{j2} \\ \sqrt{2}x_{j1} \\ \sqrt{2}x_{j2} \\ 1 \end{bmatrix}
\end{aligned}$$

Let's denote Φ_x^* to be the polynomial feature matrix of one data point, and we have:

$$\Phi_x^* = \begin{bmatrix} x_1^2 \\ x_2^2 \\ \sqrt{2}x_1x_2 \\ \sqrt{2}x_1 \\ \sqrt{2}x_2 \\ 1 \end{bmatrix}$$

Compare it to the polynomial feature matrix without weights Φ_x of one data point:

$$\Phi_x = \begin{bmatrix} x_1^2 \\ x_2^2 \\ x_1x_2 \\ x_1 \\ x_2 \\ 1 \end{bmatrix} \quad \Phi_x^* = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & \sqrt{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & \sqrt{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & \sqrt{2} & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1^2 \\ x_2^2 \\ x_1x_2 \\ x_1 \\ x_2 \\ 1 \end{bmatrix}$$

Let's denote the diagonal matrix above as D. We then take a look at the full polynomial feature matrix with n data points:

$$\boldsymbol{\Phi}_{\boldsymbol{x}}^* = \begin{bmatrix} \Phi_{x1}^{*\top} \\ \Phi_{x2}^{*\top} \\ \vdots \\ \Phi_{xn}^{*\top} \end{bmatrix} \quad \boldsymbol{\Phi}_{\boldsymbol{x}} = \begin{bmatrix} \Phi_{x1}^{\top} \\ \Phi_{x2}^{\top} \\ \vdots \\ \Phi_{xn}^{\top} \end{bmatrix}$$

Apply the linear transformation above gives us:

$$\boldsymbol{\Phi}_{\boldsymbol{x}}^{*} = \begin{bmatrix} \Phi_{x1}^{\top} D^{\top} \\ \Phi_{x2}^{\top} D^{\top} \\ \vdots \\ \Phi_{xn}^{\top} D^{\top} \end{bmatrix} = \boldsymbol{\Phi}_{\boldsymbol{x}} D$$

Here $\boldsymbol{\Phi}_{\boldsymbol{x}} \in \mathbb{R}^{n \times 6}$ and $D \in \mathbb{R}^{6 \times 6}$. It should be noted that $\boldsymbol{\Phi}_{\boldsymbol{x}}$ is a stack of row feature vectors so the position of D is changed from the left to the right. Let's just put those transformations together to avoid confusion:

$$\Phi_x^* = D\Phi_x \quad \boldsymbol{\Phi}_{\boldsymbol{x}}^{*} = \boldsymbol{\Phi}_{\boldsymbol{x}} D$$
$$\Phi_x^{*\top} = \Phi_x^{\top} D \quad \boldsymbol{\Phi}_{\boldsymbol{x}}^{*\top} = D\boldsymbol{\Phi}_{\boldsymbol{x}}{}^{\top}$$

Now, let's substitute this equality into the prediction for a new point $\vec{x}$ which has a polynomial feature vector Φ_x (Φ_x is only one data point so $\Phi_x^{\top}$ is our regular feature matrix with one data point and $\Phi_x^{*\top} w^*$ is the predictor which is a scalar). We also denote the polynomial feature vector of y to be Φ_y. Recall the predictor for kernel ridge regression is given by:

$$\begin{aligned}
& \Phi_x^{*\top} \vec{w*} \\
= \; & \Phi_x^{\top} D \boldsymbol{\Phi}_{\boldsymbol{x}}{}^{*\top} \left(\boldsymbol{\Phi}_{\boldsymbol{x}}{}^{*} \boldsymbol{\Phi}_{\boldsymbol{x}}{}^{*\top} + \lambda \mathbb{I} \right)^{-1} \Phi_y \\
= \; & \Phi_x^{\top} D D \boldsymbol{\Phi}_{\boldsymbol{x}}{}^{\top} \left(\boldsymbol{\Phi}_{\boldsymbol{x}} D D \boldsymbol{\Phi}_{\boldsymbol{x}}{}^{\top} + \lambda \mathbb{I} \right)^{-1} \Phi_y \\
= \; & \Phi_x^{\top} D \left(D \boldsymbol{\Phi}_{\boldsymbol{x}}{}^{\top} \boldsymbol{\Phi}_{\boldsymbol{x}} D + \lambda \mathbb{I} \right)^{-1} D \boldsymbol{\Phi}_{\boldsymbol{x}}{}^{\top} \Phi_y \\
= \; & \Phi_x^{\top} \left(D^{-1} D \boldsymbol{\Phi}_{\boldsymbol{x}}{}^{\top} \boldsymbol{\Phi}_{\boldsymbol{x}} D D^{-1} + D^{-1} \lambda \mathbb{I} D^{-1} \right)^{-1} \boldsymbol{\Phi}_{\boldsymbol{x}}{}^{\top} \Phi_y \\
= \; & \Phi_x^{\top} \left(\boldsymbol{\Phi}_{\boldsymbol{x}}{}^{\top} \boldsymbol{\Phi}_{\boldsymbol{x}} + D^{-1} \lambda \mathbb{I} D^{-1} \right)^{-1} \boldsymbol{\Phi}_{\boldsymbol{x}}{}^{\top} \Phi_y \\
= \; & \Phi_x^{\top} \left(\boldsymbol{\Phi}_{\boldsymbol{x}}{}^{\top} \boldsymbol{\Phi}_{\boldsymbol{x}} + 2\lambda \Gamma^{\top} \Gamma \right)^{-1} \boldsymbol{\Phi}_{\boldsymbol{x}}{}^{\top} \Phi_y
\end{aligned}$$

Therefore, we can get the Tikhonov matrix:

$$D^{-1} \lambda \mathbb{I} D^{-1} = \lambda \Gamma^{\top} \Gamma$$

We just need to set $\Gamma = D^{-1}$. That is:

$$\Gamma = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{\sqrt{2}}{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{\sqrt{2}}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{\sqrt{2}}{2} & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

(e) In general, for any polynomial regression with pth order polynomial on $\mathbb{R}^d$ 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 $\vec{w}$.) **Compare with the computational complexity of actually doing prediction as well.**

(1) The time complexity of doinig Tikhonov regression directly:
Get polynomial feature matrix $\Rightarrow O(d^p)$
Matrix Multiplication$\Rightarrow O(d^{2p}n)$
Matrix inversion $\Rightarrow O(d^{3p})$
Overall time complexity $\Rightarrow O(d^{3p} + d^{2p}n)$

(2) The time complexity of doing kernel ridge regression instead:
Use kernel function to calculate $\Phi_x\Phi_x^\top \Rightarrow O(n^2(d + logp))$
Matrix inversion $\Rightarrow O(n^3)$
Overall time complexity $\Rightarrow O(n^3 + n^2(d + logp))$

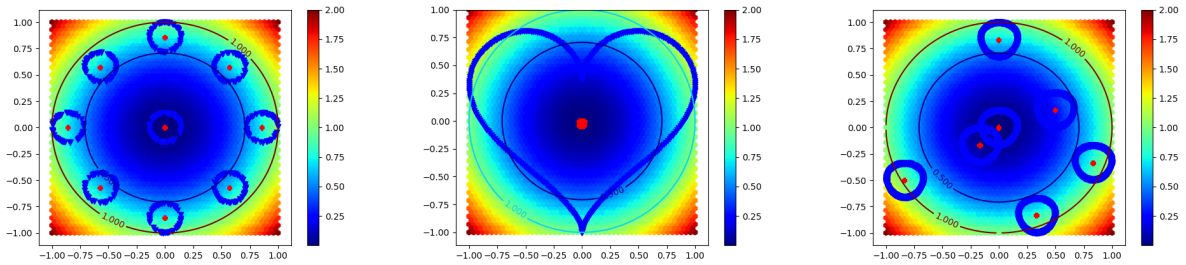
(3) The time complexity of doing prediction using Tikhonov regression directly:
Get polynomial feature matrix for one new data point $\Rightarrow O(d^p)$
Matrix multiplication to get the predictor $\Rightarrow O(d^p)$
Overall time complexity $\Rightarrow O(d^p)$

(4) The time complexity of doing prediction using kernel ridge regression instead:
Use kernel function to calculate $\Phi_x^\top {\boldsymbol{\Phi}_{\boldsymbol{x}}}^{*\top}$ for one new data point $\Rightarrow O(n(d + logp))$
Matrix multiplication to get the predictor $\Rightarrow O(n)$
Overall time complexity $\Rightarrow O(n(d + logp))$
We prefer using kernel when n is small but d and p are large.

Question 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., $\vec{x}_i \in \mathbb{R}^2$. 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.



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

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

# choose the data you want to load
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
```

```
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 > clip] = clip
z0[z0 < -clip] = -clip

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')
plt.show()


def main():
# example usage of heatmap
heatmap(lambda x, y: x * x + y * y)


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

(b) **Implement polynomial ridge regression** (non-kernelized version that you should already have implemented in your previous homework) **to fit all three datasets**. Use the first 80% data as the training dataset and the last 20% data as the validation dataset. **Report both the average training error and the average validation error 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 only for** `asymmetric.npz` **(as you have already done for the other two datasets) over the 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.

(Please be aware that the vector is auto-folded due the width limit of the page) The polynomial order is from 1 to 16.

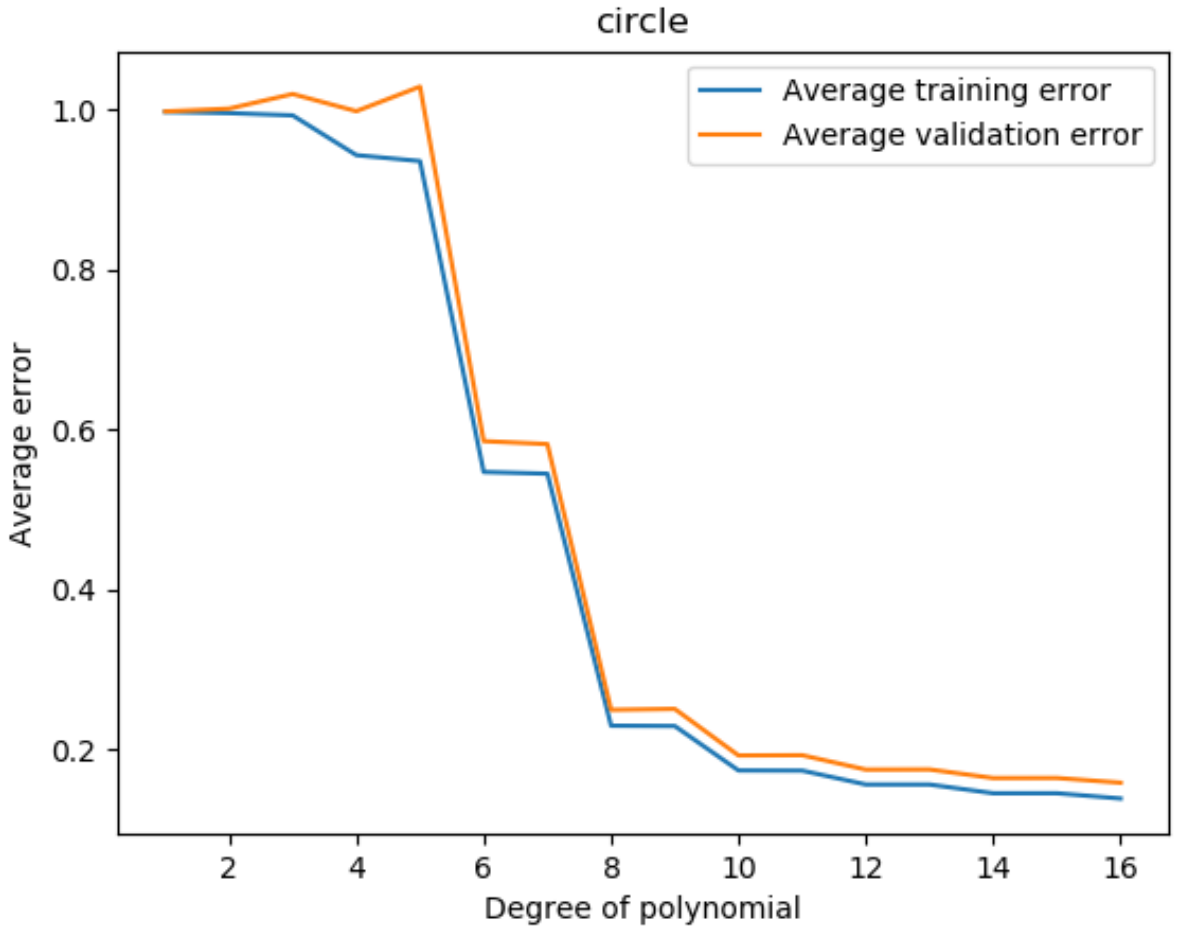
circle

Average training error:

$$\begin{bmatrix} 0.99708761332 & 0.99553683019 & 0.99269893794 & 0.94301144809 & 0.93554984849 \\ 0.547155137 & 0.5450153735 & 0.23019031846 & 0.22976015144 & 0.17427302935 \\ 0.17407314779 & 0.15672280433 & 0.1566710619 & 0.14578655478 & 0.14578368356 \\ 0.1391981828 \end{bmatrix}$$

Average validation error:

$$\begin{bmatrix} 0.9975785898 & 1.00105611443 & 1.01935012248 & 0.99791404283 & 1.02859658708 \\ 0.58568759327 & 0.58200710751 & 0.24999004454 & 0.25113476414 & 0.19299805979 \\ 0.19329666071 & 0.17533526394 & 0.17548995194 & 0.16474477919 & 0.16481623672 \\ 0.15895198216 \end{bmatrix}$$



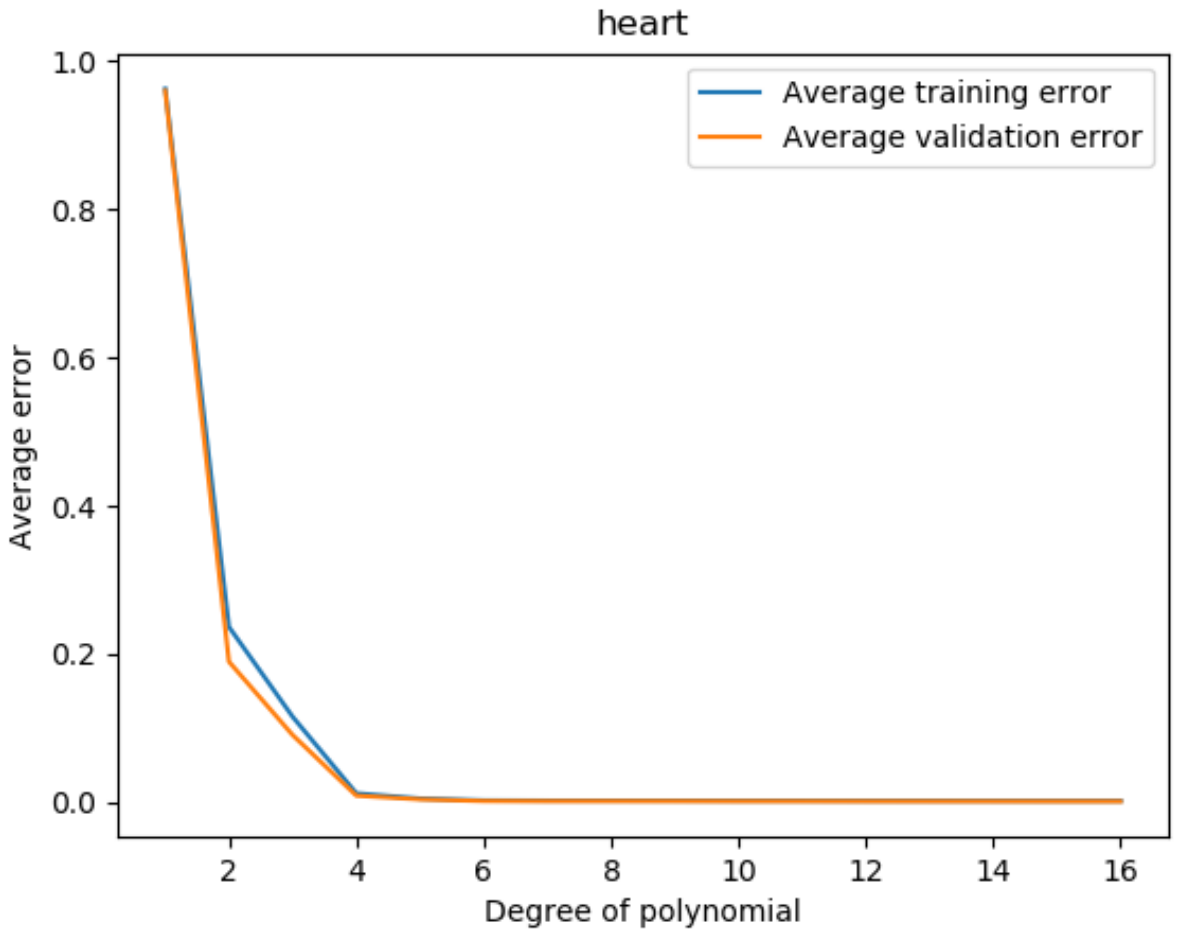
heart

Average training error:

$$\begin{bmatrix} 0.96264303387 & 0.23671821257 & 0.11548077719 & 0.01216926948 & 0.00516037453 \\ 0.00262962803 & 0.00237796468 & 0.0023538032 & 0.00232107088 & 0.00219334634 \\ 0.00218360499 & 0.00209044689 & 0.00207021839 & 0.0020360911 & 0.00200519507 \\ 0.00199829694 \end{bmatrix}$$

Average validation error:

$$\begin{bmatrix} 0.95995232727 & 0.18983714833 & 0.090801213 & 0.00912261506 & 0.00410240958 \\ 0.00185798904 & 0.00164415368 & 0.00163965427 & 0.00160872638 & 0.00150035109 \\ 0.00148787711 & 0.00141364432 & 0.00139519851 & 0.00137072205 & 0.00134404497 \\ 0.00134015086 \end{bmatrix}$$

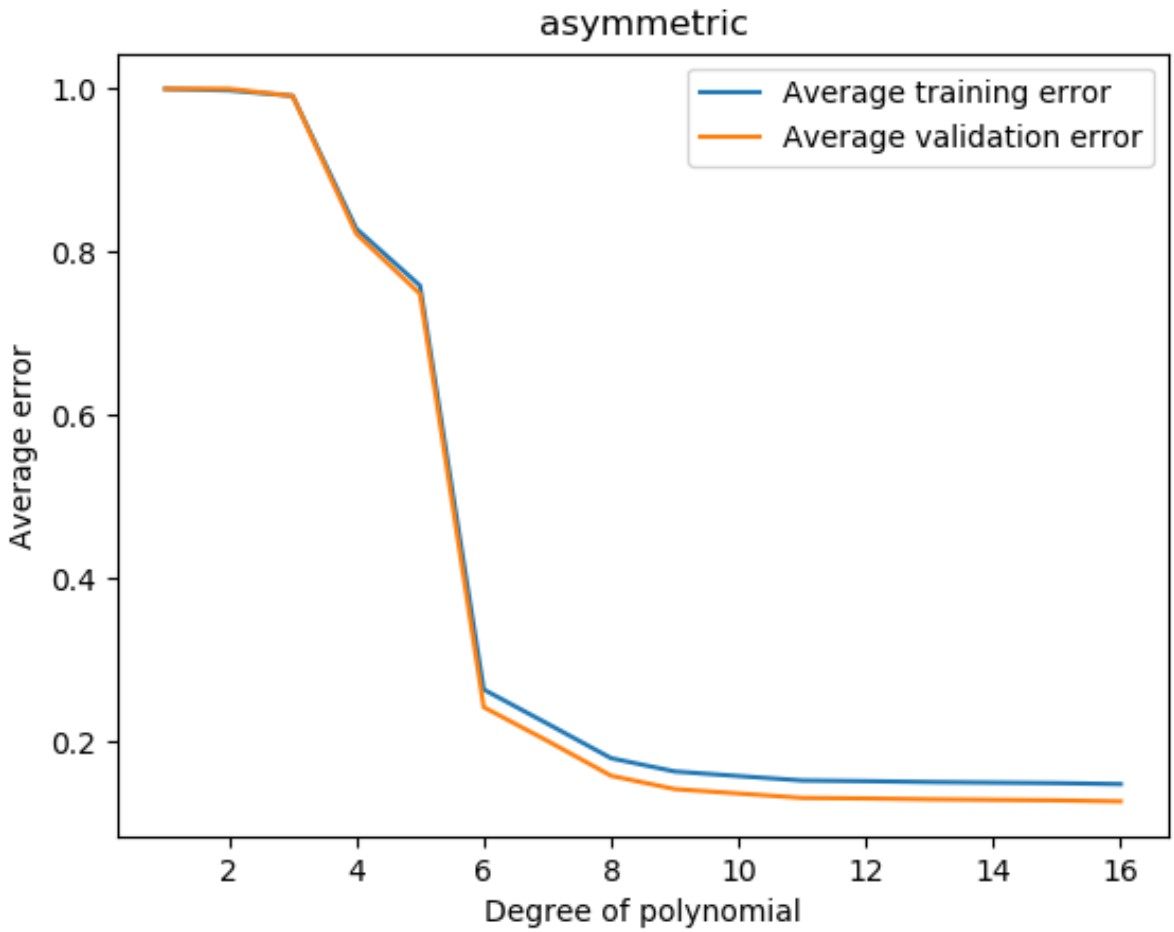


asymmetric
Average training error:

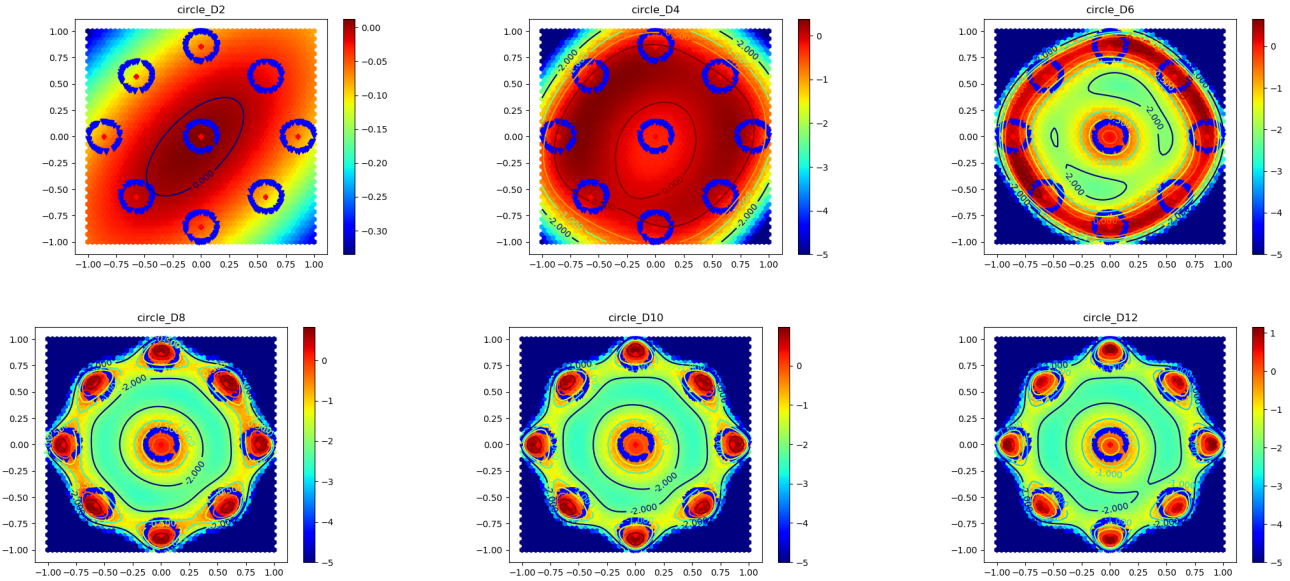
$$\begin{bmatrix} 0.99998894733 & 0.99825960859 & 0.99156494313 & 0.82869215058 & 0.75898687368 \\ 0.26403992227 & 0.22219460539 & 0.17985325231 & 0.16358087734 & 0.15797733433 \\ 0.15256240975 & 0.15173648517 & 0.15066193745 & 0.14994973815 & 0.14929491859 \\ 0.14813412684 \end{bmatrix}$$

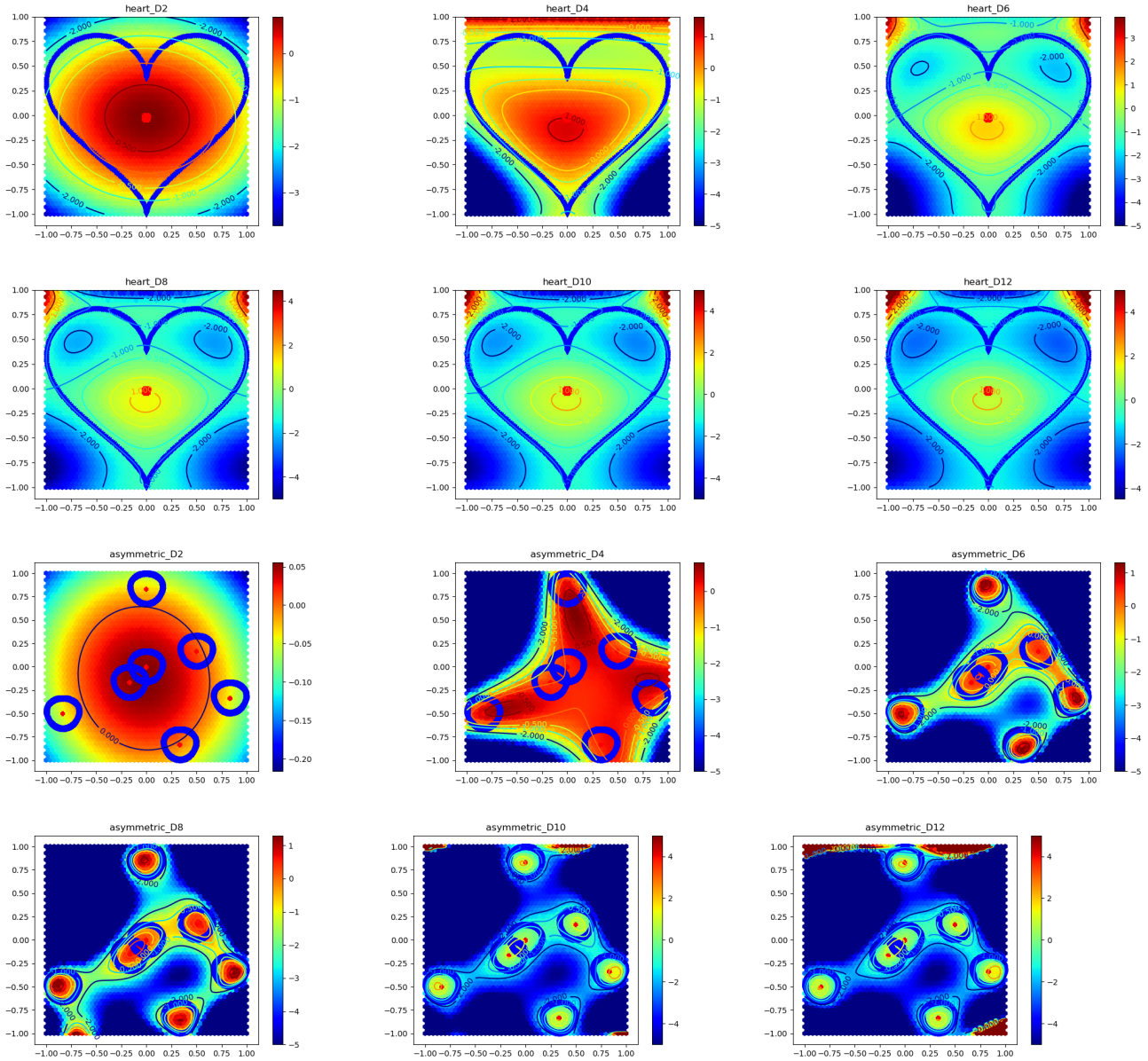
Average validation error:

$$\begin{bmatrix} 1.00019385473 & 1.00017621399 & 0.99138842516 & 0.82236887751 & 0.74881111802 \\ 0.24239777871 & 0.20113932036 & 0.15834715798 & 0.14199340088 & 0.13662301479 \\ 0.13119849463 & 0.13051903373 & 0.12951624491 & 0.1288266216 & 0.12817375839 \\ 0.12694373237 \end{bmatrix}$$



Now let's visualize the results. Because lambda function is garbage, I replaced it.






```
import matplotlib.pyplot as plt
from matplotlib import cm
import numpy as np
import scipy.io as spio

SPLIT = 0.8
KD = 16 # max D = 16
LAMBDA = 0.001

data_names = ('circle', 'heart', 'asymmetric')

def bmatrix(a):
"""Returns a LaTeX bmatrix
Retrieved from https://stackoverflow.com/questions/17129290/numpy-2d-and-1d-array-to-latex-bmatrix
:a: numpy array
:returns: LaTeX bmatrix as a string
"""
if len(a.shape) > 2:
raise ValueError('bmatrix can at most display two dimensions')
lines = str(a).replace('[', '').replace(']', '').splitlines()
```

```
rv = [r'\begin{bmatrix}']
rv += [' ' + ' & '.join(l.split()) + r'\\' for l in lines]
rv += [r'\end{bmatrix}']
return '\n'.join(rv)

def ridge(A, b, lambda_):
# Make sure data are centralized
return np.linalg.solve(A.T.dot(A) + lambda_ * np.eye(A.shape[1]), A.T.dot(b))


def get_error(X, w, y):
return np.mean((X.dot(w) - y) ** 2)


def fit(D, lambda_, train_x, train_y, validation_x, validation_y):
# YOUR CODE TO COMPUTE THE AVERAGE ERROR PER SAMPLE
errors = np.asarray([0.0, 0.0])
poly_train_x = polynomial(train_x, D)
w = ridge(poly_train_x, train_y, lambda_)
errors[0] = get_error(poly_train_x, w, train_y)
errors[1] = get_error(polynomial(validation_x, D), w, validation_y)
return errors, w


def polynomial(poly_data, degree):
if poly_data.ndim != 2:
pass
nvar = poly_data.shape[1]
npoints = poly_data.shape[0]
results = np.ones((npoints, comb(degree + nvar, degree)))
if degree == 0:
return results
start = 0
counts = np.zeros(nvar, dtype=int)
cur_index = 1
cur_len = 1
for d in range(1, degree + 1):
last_len = cur_len
cur_start = 0
cur_len = 0
for i in range(0, nvar):
for j in range(start + cur_start, start + last_len):
results[:, cur_index] = poly_data[:, i] * results[:, j]
cur_index += 1
temp = counts[i]
counts[i] = last_len - cur_start
cur_start += temp
cur_len += counts[i]
start = start + last_len

return results


def get_polyvalue(x0, y0, coefficient, degree):
N = len(y0)
result = np.sum(polynomial(np.column_stack([x0, y0]), degree) * np.vstack([coefficient] * N), axis=1)
return result

def comb(n, k):
result = 1
for i in range(k):
result *= (n - i)
for i in range(k):
```

```
result /= (k - i)
return int(np.round(result))


def main():
np.set_printoptions(precision=11)

for i_database, name_database in enumerate(data_names):
# choose the data you want to load
data = np.load(name_database + '.npz')
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:]
Etrain = np.zeros(KD)
Evalid = np.zeros(KD)
print(r'\textbf{' + name_database + r'}\\')
for D in range(KD):
# print(D + 1)
[Etrain[D], Evalid[D]], weights = fit(D + 1, LAMBDA, X_train, y_train, X_valid, y_valid)
heatplt = heatmap(X, y, weights, D + 1, clip=5)
heatplt.title(name_database + '_D' + str(D+1))
heatplt.savefig('Figure_5b_heatmap_' + name_database + '_D' + str(D+1) + '.png')

print('Average training error:')
print('\[')
print(bmatrix(Etrain))
print('\]')
print('Average validation error:')
print('\[')
print(bmatrix(Evalid))
print('\]')
print('\includegraphics[width=1.0\textwidth]{Figure_5b_mse_'+ name_database + r'}\\')

plt.figure()
plt.plot(np.linspace(1, KD, KD), Etrain, label='Average training error')
plt.plot(np.linspace(1, KD, KD), Evalid, label='Average validation error')
plt.xlabel('Degree of polynomial')
plt.ylabel('Average error')
plt.title(name_database)
plt.legend()
plt.savefig('Figure_5b_mse_'+ name_database +'.png')

def heatmap(X, y, coef, degree, 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

xx = yy = np.linspace(np.min(X), np.max(X), 72)
x0, y0 = np.meshgrid(xx, yy)
x0, y0 = x0.ravel(), y0.ravel()
z0 = get_polyvalue(x0, y0, coef, degree)
# print(z0)

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

```
plt.figure()
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')
# plt.show()
return plt

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

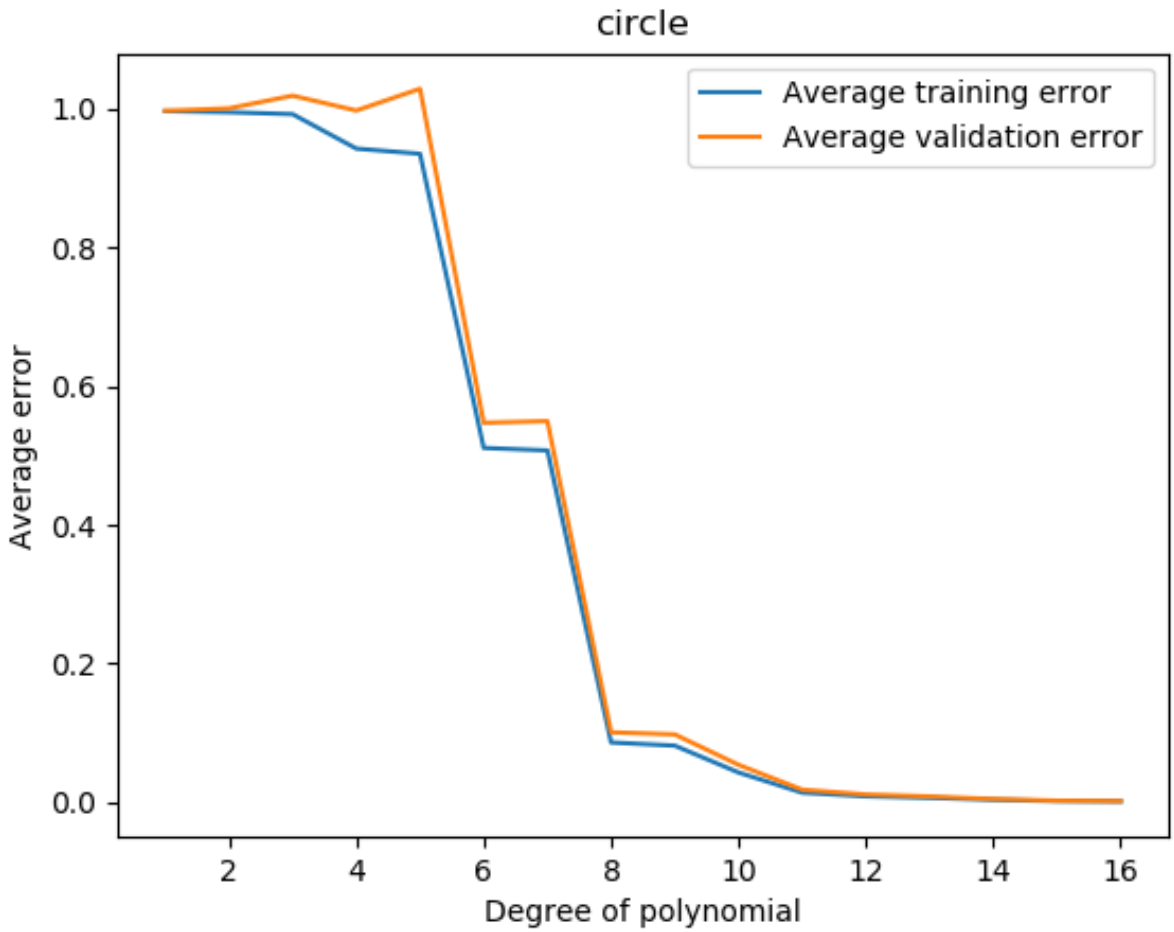
(c) **Implement kernel ridge regression to fit the datasets `circle.npz`, `asymmetric.npy`, and `heart.npz`.** Use the polynomial kernel $K(\vec{x}_i, \vec{x}_j) = (1 + \vec{x}_i^\top \vec{x}_j)^p$. Use the first 80% data as the training dataset and the last 20% data as the validation dataset. **Report both the average training error, and the average validation error 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.** The sample code for generating heatmap plot is included in the start kit. **For `circle.npz`, also report the average training error and validation error 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 to use a high-order polynomial in linear/ridge regression.**

I will cheat here by using our conclusion in Q4 and converting the problem into Tikhonov regularization: **circle**
Average training error:

$$\begin{bmatrix} 0.99708761332 & 0.99553683018 & 0.99269893719 & 0.94301126322 & 0.93553881067 \\ 0.51124143362 & 0.50759247898 & 0.08638915813 & 0.08180925266 & 0.0430864829 \\ 0.01396641171 & 0.0086847439 & 0.00651656971 & 0.00366509033 & 0.00191170212 \\ 0.00139993861 & & & & \end{bmatrix}$$

Average validation error:

$$\begin{bmatrix} 0.9975785898 & 1.00105619555 & 1.01935608633 & 0.9979414392 & 1.02930778276 \\ 0.54753103979 & 0.54992737354 & 0.1010558177 & 0.09798907416 & 0.05416663018 \\ 0.01828999698 & 0.01134838351 & 0.00855594446 & 0.00482119954 & 0.00247543849 \\ 0.00179713389 & & & & \end{bmatrix}$$



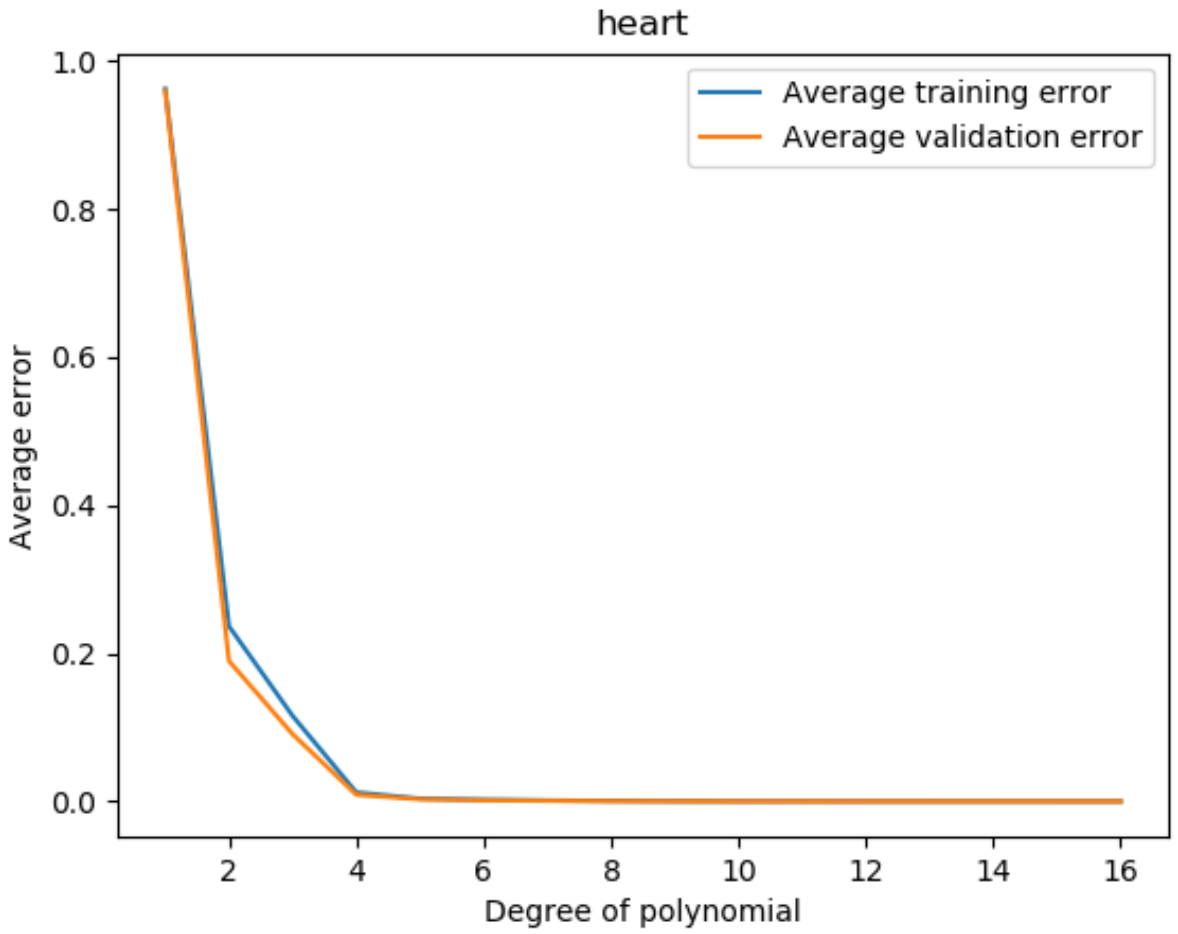
heart

Average training error:

$$\begin{bmatrix} 9.62643033874e-01 & 2.36718212573e-01 & 1.15480763896e-01 \\ 1.21631328661e-02 & 3.75913955820e-03 & 2.29395081004e-03 \\ 1.44087706457e-03 & 6.65027636982e-04 & 3.05465896121e-04 \\ 1.88558125011e-04 & 1.39349161546e-04 & 1.10966037041e-04 \\ 9.32048506155e-05 & 8.11449300007e-05 & 7.18264454175e-05 \\ 6.39087469644e-05 & & \end{bmatrix}$$

Average validation error:

$$\begin{bmatrix} 9.59952327266e-01 & 1.89837299481e-01 & 9.08127666321e-02 \\ 9.08936200226e-03 & 2.97530863355e-03 & 1.61294508137e-03 \\ 1.05595300378e-03 & 4.28480728581e-04 & 2.02467563623e-04 \\ 1.38336762240e-04 & 1.14430617218e-04 & 9.66655064697e-05 \\ 8.41703116840e-05 & 7.51603037493e-05 & 6.80934203027e-05 \\ 6.18938495270e-05 & & \end{bmatrix}$$

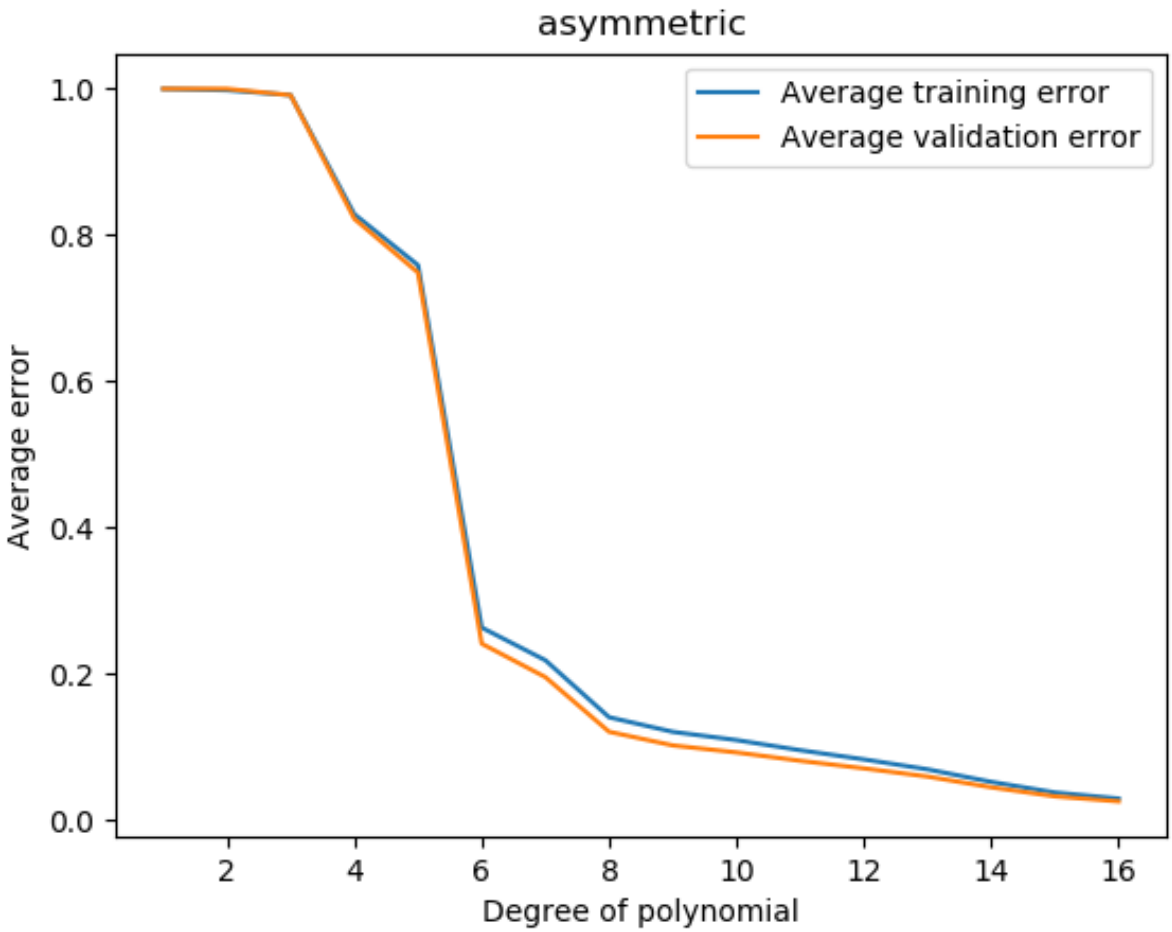


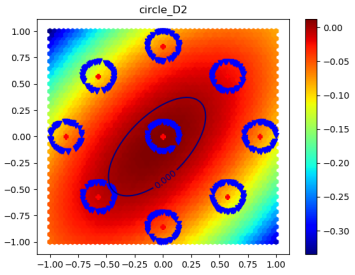
asymmetric
Average training error:

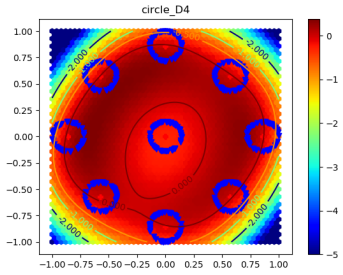
$$\begin{bmatrix} 0.99998894733 & 0.99825960859 & 0.99156494312 & 0.82869197963 & 0.75898567628 \\ 0.26336785333 & 0.2186902709 & 0.14072114116 & 0.12078076844 & 0.10952039431 \\ 0.09564509907 & 0.08312637272 & 0.06951922449 & 0.05233904001 & 0.03778454925 \\ 0.02951062101 & & & & \end{bmatrix}$$

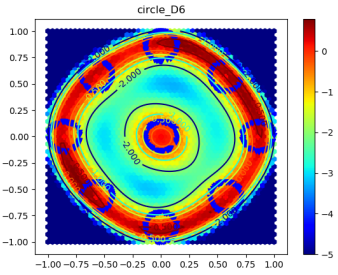
Average validation error:

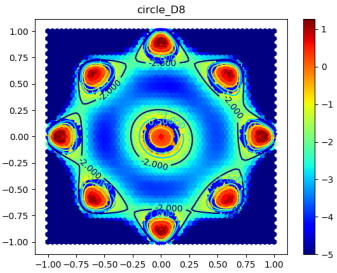
$$\begin{bmatrix} 1.00019385473 & 1.00017621383 & 0.99138849997 & 0.82237278408 & 0.74881648724 \\ 0.24139774186 & 0.19560635716 & 0.12089080238 & 0.10223905153 & 0.09260297403 \\ 0.0811899801 & 0.07082649485 & 0.05963545759 & 0.04494200208 & 0.03257504477 \\ 0.02568978698 & & & & \end{bmatrix}$$

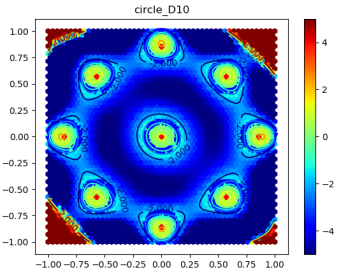

asymmetric
Average training error
Average validation error
Average error
Degree of polynomial
1.0
0.8
0.6
0.4
0.2
0.0
2
4
6
8
10
12
14
16

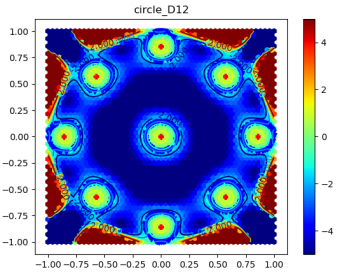

circle_D2

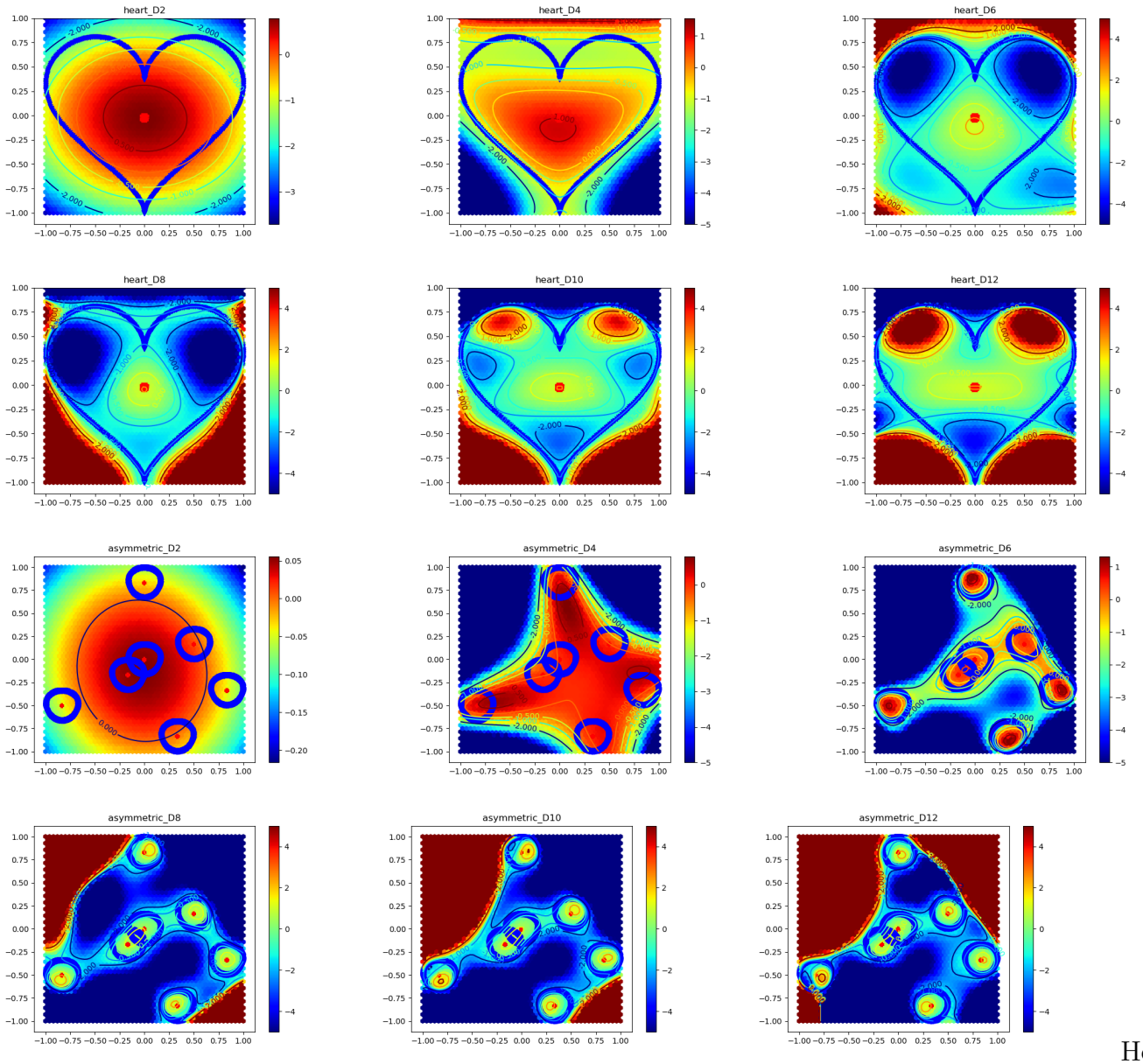

circle_D4


circle_D6


circle_D8


circle_D10


circle_D12



Here is the code using kernel ridge regression:

```
import matplotlib.pyplot as plt
from matplotlib import cm
import numpy as np
import scipy.io as spio

SPLIT = 0.8
KD = 16 # max D = 16
LAMBDA = 0.001

data_names = ('circle', 'heart', 'asymmetric')

def bmatrix(a):
"""Returns a LaTeX bmatrix
Retrieved from https://stackoverflow.com/questions/17129290/numpy-2d-and-1d-array-to-latex-bmatrix
:a: numpy array
:returns: LaTeX bmatrix as a string
"""
if len(a.shape) > 2:
raise ValueError('bmatrix can at most display two dimensions')
```

```
lines = str(a).replace('[', '').replace(']', '').splitlines()
rv = [r'\begin{bmatrix}']
rv += [' ' + ' & '.join(l.split()) + r'\\' for l in lines]
rv += [r'\end{bmatrix}']
return '\n'.join(rv)


def kernel_ridge(gram_matrix, b, lambda_):
# Make sure data are centralized
return np.linalg.solve(gram_matrix + lambda_ * np.eye(gram_matrix.shape[1]), b)


def get_error(X, alpha, y):
return np.mean((X.dot(alpha) - y) ** 2)


def fit(D, lambda_, train_x, train_y, validation_x, validation_y):
# YOUR CODE TO COMPUTE THE AVERAGE ERROR PER SAMPLE
errors = np.asarray([0.0, 0.0])
alpha = kernel_ridge(kernel(train_x, train_x, D), train_y, lambda_)
errors[0] = get_error(kernel(train_x, train_x, D), alpha, train_y)
errors[1] = get_error(kernel(validation_x, train_x, D), alpha, validation_y)
return errors, alpha


def kernel(phi_x, phi_z, degree):
nx = phi_x.shape[0]
nz = phi_z.shape[0]
gram_matrix = np.zeros((nx, nz))
for i in range(nx):
xi = phi_x[i,:]
for j in range(nz):
xj = phi_z[j,:]
gram_matrix[i, j] = (1 + xi.T.dot(xj)) ** degree
return gram_matrix


def get_kernelvalue(x0, y0, X_train, coefficient, degree):
N = len(y0)
result = np.sum(kernel(np.column_stack([x0, y0]), X_train, degree) * np.vstack([coefficient] * N), axis=1)
return result


def main():
np.set_printoptions(precision=11)

for i_database, name_database in enumerate(data_names):
# choose the data you want to load
data = np.load(name_database + '.npz')
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:]
Etrain = np.zeros(KD)
Evalid = np.zeros(KD)
print(r'\textbf{' + name_database + r'}\\')
for D in range(KD):
# print(D + 1)
```

```
[Etrain[D], Evalid[D]], weights = fit(D + 1, LAMBDA, X_train, y_train, X_valid, y_valid)
heatplt = heatmap(X, y, X_train, weights, D + 1, clip=5)
heatplt.title(name_database + '_D' + str(D+1))
heatplt.savefig('Figure_5c_heatmap_' + name_database + '_D' + str(D+1) + '.png')
heatplt.close()

heatplt = heatmap(X, y, X_train, weights, D + 1, clip=0)
heatplt.title(name_database + '_D' + str(D+1))
heatplt.savefig('Figure_5c_heatmap_noclip_' + name_database + '_D' + str(D+1) + '.png')
heatplt.close()


print('Average training error:')
print('\[')
print(bmatrix(Etrain))
print('\]')
print('Average validation error:')
print('\[')
print(bmatrix(Evalid))
print('\]')
print('\includegraphics[width=1.0\textwidth]{Figure_5c_mse_'+ name_database + r'}\\')
plt.figure()
plt.plot(np.linspace(1, KD, KD), Etrain, label='Average training error')
plt.plot(np.linspace(1, KD, KD), Evalid, label='Average validation error')
plt.xlabel('Degree of polynomial')
plt.ylabel('Average error')
plt.title(name_database)
plt.legend()
plt.savefig('Figure_5c_mse_'+ name_database +'.png')

def heatmap(X, y, X_train, coef, degree, 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

xx = yy = np.linspace(np.min(X), np.max(X), 72)
x0, y0 = np.meshgrid(xx, yy)
x0, y0 = x0.ravel(), y0.ravel()
z0 = get_kernelvalue(x0, y0, X_train, coef, degree)
# print(z0)

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

plt.figure()
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')
# plt.show()
return plt

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

And the code using Tikhonov regularization. I'm being honest here so don't take points off, please.

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

SPLIT = 0.8
KD = 16 # max D = 16
LAMBDA = 0.001

data_names = ('circle', 'heart', 'asymmetric')


def bmatrix(a):
"""Returns a LaTeX bmatrix
Retrieved from https://stackoverflow.com/questions/17129290/numpy-2d-and-1d-array-to-latex-bmatrix
:a: numpy array
:returns: LaTeX bmatrix as a string
"""
if len(a.shape) > 2:
raise ValueError('bmatrix can at most display two dimensions')
lines = str(a).replace('[', '').replace(']', '').splitlines()
rv = [r'\begin{bmatrix}']
rv += [' ' + ' & '.join(l.split()) + r'\\' for l in lines]
rv += [r'\end{bmatrix}']
return '\n'.join(rv)


def poly_coefs(degree):
"""
This function returns the pascal triangle at level degree
:return: a numpy array with all polynomial coefficients
"""
results = np.zeros(degree + 1)
for i in range(degree + 1):
results[i] = comb(degree, i)
return results


def get_kernel_coefs(nvar, degree):
"""
calculate the diagonal entries of kernel method to create Tikhonov matrix
:param nvar:
:param degree:
:return:
"""
results = np.zeros(comb(degree + nvar, degree))
poly_scale_coefs = poly_coefs(degree)
pointer = 0
for idegree in range(degree + 1):
results[pointer: pointer + idegree + 1] = \
poly_coefs(idegree) * poly_scale_coefs[idegree]
pointer += idegree + 1
return results


def ridge(A, b, lambda_, nvar, degree):
"""This function only works if there are only two variables
"""
# Make sure data are centralized
return np.linalg.solve(A.T.dot(A) + lambda_ * np.diag(1 / get_kernel_coefs(nvar, degree)), A.T.dot(b))


def get_error(X, w, y):
return np.mean((X.dot(w) - y) ** 2)
```

```
def fit(D, lambda_, train_x, train_y, validation_x, validation_y):
# YOUR CODE TO COMPUTE THE AVERAGE ERROR PER SAMPLE
errors = np.asarray([0.0, 0.0])
poly_train_x = polynomial(train_x, D)
w = ridge(poly_train_x, train_y, lambda_, train_x.shape[1], D)
errors[0] = get_error(poly_train_x, w, train_y)
errors[1] = get_error(polynomial(validation_x, D), w, validation_y)
return errors, w


def polynomial(poly_data, degree):
if poly_data.ndim != 2:
pass
nvar = poly_data.shape[1]
npoints = poly_data.shape[0]
results = np.ones((npoints, comb(degree + nvar, degree)))
if degree == 0:
return results
start = 0
counts = np.zeros(nvar, dtype=int)
cur_index = 1
cur_len = 1
for d in range(1, degree + 1):
last_len = cur_len
cur_start = 0
cur_len = 0
for i in range(0, nvar):
for j in range(start + cur_start, start + last_len):
results[:, cur_index] = poly_data[:, i] * results[:, j]
cur_index += 1
temp = counts[i]
counts[i] = last_len - cur_start
cur_start += temp
cur_len += counts[i]
start = start + last_len

return results


def get_polyvalue(x0, y0, coefficient, degree):
N = len(y0)
result = np.sum(polynomial(np.column_stack([x0, y0]), degree) * np.vstack([coefficient] * N), axis=1)
return result


def comb(n, k):
result = 1
for i in range(k):
result *= (n - i)
for i in range(k):
result /= (k - i)
return int(np.round(result))


def main():
np.set_printoptions(precision=11)

for i_database, name_database in enumerate(data_names):
# choose the data you want to load
data = np.load(name_database + '.npz')
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:]
Etrain = np.zeros(KD)
Evalid = np.zeros(KD)
print(r'\textbf{' + name_database + r'}\\')
for D in range(KD):
# print(D + 1)
[Etrain[D], Evalid[D]], weights = fit(D + 1, LAMBDA, X_train, y_train, X_valid, y_valid)

heatplt = heatmap(X, y, weights, D + 1, clip=5)
heatplt.title(name_database + '_D' + str(D + 1))
heatplt.savefig('Figure_5c_heatmap_' + name_database + '_D' + str(D + 1) + '.png')
heatplt.close()

heatplt = heatmap(X, y, weights, D + 1, clip=0)
heatplt.title(name_database + '_D' + str(D + 1))
heatplt.savefig('Figure_5c_heatmap_noclip_' + name_database + '_D' + str(D + 1) + '.png')
heatplt.close()

print('Average training error:')
print('\[')
print(bmatrix(Etrain))
print('\]')
print('Average validation error:')
print('\[')
print(bmatrix(Evalid))
print('\]')
print('\includegraphics[width=1.0\textwidth]{Figure_5c_mse_' + name_database + r'}\\')

plt.figure()
plt.plot(np.linspace(1, KD, KD), Etrain, label='Average training error')
plt.plot(np.linspace(1, KD, KD), Evalid, label='Average validation error')
plt.xlabel('Degree of polynomial')
plt.ylabel('Average error')
plt.title(name_database)
plt.legend()
plt.savefig('Figure_5c_mse_' + name_database + '.png')


def heatmap(X, y, coef, degree, 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

xx = yy = np.linspace(np.min(X), np.max(X), 72)
x0, y0 = np.meshgrid(xx, yy)
x0, y0 = x0.ravel(), y0.ravel()
z0 = get_polyvalue(x0, y0, coef, degree)
# print(z0)

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

plt.figure()
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')
# plt.show()
return plt


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

Now we look at circle.npz with a smaller training dataset. We get much smaller training errors but relatively large validation error. This is because we don't have enough training data to support our high dimensional polynomial model. Therefore, it's not wise to use a high dimensional polynomial model when we don't have enough data to support it. It's even worse if we can see the data have relatively simple patterns.

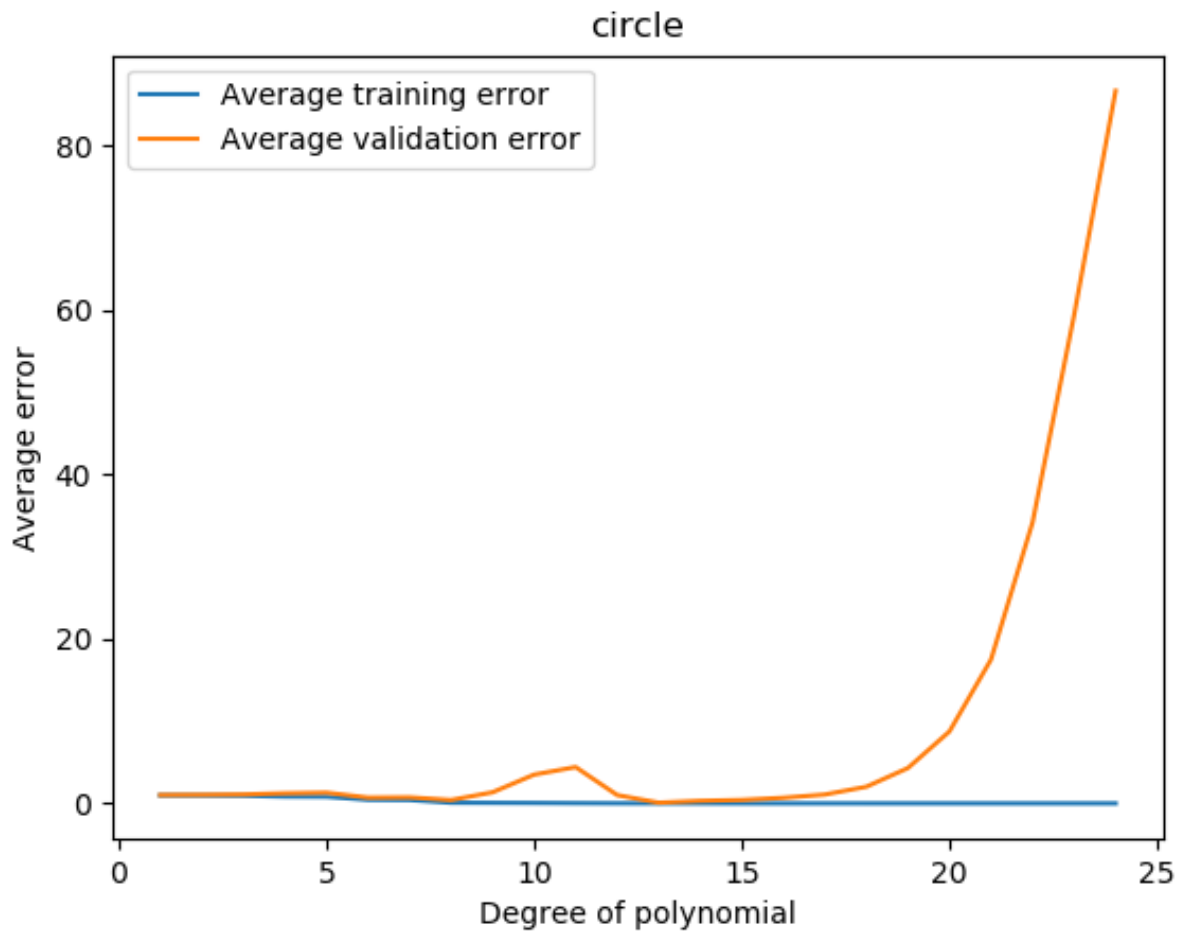
circle

Average training error:

$$\begin{bmatrix} 9.77121664814e-01 & 9.65178657315e-01 & 9.35813766962e-01 & 8.28075515529e-01 \\ 8.08157155561e-01 & 4.39518520060e-01 & 4.15687670430e-01 & 7.60984805621e-02 \\ 6.14620298158e-02 & 4.72646028211e-02 & 2.68066553304e-02 & 1.12711971542e-02 \\ 5.48169428340e-03 & 3.76945512729e-03 & 2.77101802470e-03 & 1.79787398014e-03 \\ 1.07083584030e-03 & 7.07762097847e-04 & 5.57948096548e-04 & 4.78298149246e-04 \\ 4.13846415368e-04 & 3.54101440695e-04 & 3.03356180692e-04 & 2.63196366860e-04 \end{bmatrix}$$

Average validation error:

$$\begin{bmatrix} 1.01721224532e+00 & 1.04072424153e+00 & 1.08347711127e+00 & 1.22307711851e+00 \\ 1.29959394697e+00 & 7.01293104483e-01 & 7.32313383821e-01 & 4.08354110889e-01 \\ 1.34810515977e+00 & 3.49447940150e+00 & 4.41760318237e+00 & 1.00886738346e+00 \\ 8.09952397964e-02 & 3.06176090266e-01 & 4.16934889574e-01 & 6.79290703205e-01 \\ 1.07478644051e+00 & 2.02341817582e+00 & 4.29800055612e+00 & 8.74665368523e+00 \\ 1.74941378996e+01 & 3.41564329847e+01 & 5.93395898714e+01 & 8.66455562781e+01 \end{bmatrix}$$




```
import matplotlib.pyplot as plt
from matplotlib import cm
import numpy as np
import scipy.io as spio

SPLIT = 0.8
KD = 16 # max D = 16
LAMBDA = 0.001

data_names = ('circle', 'heart', 'asymmetric')

def bmatrix(a):
"""Returns a LaTeX bmatrix
Retrieved from https://stackoverflow.com/questions/17129290/numpy-2d-and-1d-array-to-latex-bmatrix
:a: numpy array
:returns: LaTeX bmatrix as a string
"""
if len(a.shape) > 2:
raise ValueError('bmatrix can at most display two dimensions')
lines = str(a).replace('[', '').replace(']', '').splitlines()
rv = [r'\begin{bmatrix}']
rv += [' ' + ' & '.join(l.split()) + r'\\' for l in lines]
rv += [r'\end{bmatrix}']
return '\n'.join(rv)

def ridge(A, b, lambda_):
```

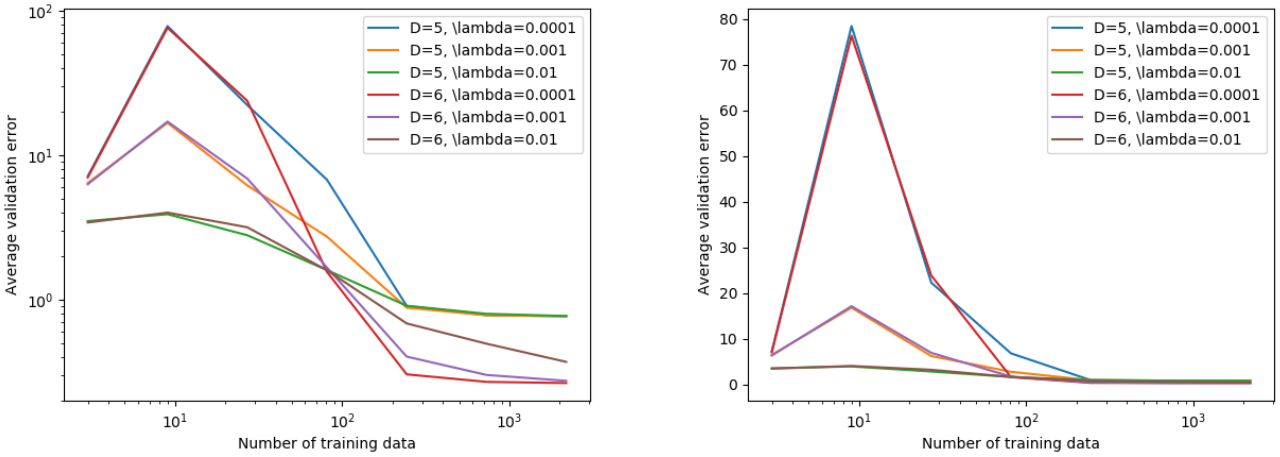
```
# Make sure data are centralized
return np.linalg.solve(A.T.dot(A) + lambda_ * np.eye(A.shape[1]), A.T.dot(b))


def get_error(X, w, y):
return np.mean((X.dot(w) - y) ** 2)


def fit(D, lambda_, train_x, train_y, validation_x, validation_y):
# YOUR CODE TO COMPUTE THE AVERAGE ERROR PER SAMPLE
errors = np.asarray([0.0, 0.0])
poly_train_x = polynomial(train_x, D)
w = ridge(poly_train_x, train_y, lambda_)
errors[0] = get_error(poly_train_x, w, train_y)
errors[1] = get_error(polynomial(validation_x, D), w, validation_y)
return errors, w


def polynomial(poly_data, degree):
if poly_data.ndim != 2:
pass
nvar = poly_data.shape[1]
npoints = poly_data.shape[0]
results = np.ones((npoints, comb(degree + nvar, degree)))
if degree == 0:
return results
start = 0
counts = np.zeros(nvar, dtype=int)
cur_index = 1
cur_len = 1
for d in range(1, degree + 1):
last_len = cur_len
cur_start = 0
cur_len = 0
for i in range(0, nvar):
for j in range(start + cur_start, start + last_len):
results[:, cur_index] = poly_data[:, i] * results[:, j]
cur_index += 1
temp = counts[i]
counts[i] = last_len - cur_start
cur_start += temp
cur_len += counts[i]
start = start + last_len

return results


def get_polyvalue(x0, y0, coefficient, degree):
N = len(y0)
result = np.sum(polynomial(np.column_stack([x0, y0]), degree) * np.vstack([coefficient] * N), axis=1)
return result

def comb(n, k):
result = 1
for i in range(k):
result *= (n - i)
for i in range(k):
result /= (k - i)
return int(np.round(result))


def main():
np.set_printoptions(precision=11)
```

```
for i_database, name_database in enumerate(data_names):
# choose the data you want to load
data = np.load(name_database + '.npz')
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:]
Etrain = np.zeros(KD)
Evalid = np.zeros(KD)
print(r'\textbf{' + name_database + r'}\\')
for D in range(KD):
# print(D + 1)
[Etrain[D], Evalid[D]], weights = fit(D + 1, LAMBDA, X_train, y_train, X_valid, y_valid)
# heatplt = heatmap(X, y, weights, D + 1, clip=5)
# heatplt.title(name_database + '_D' + str(D+1))
# heatplt.savefig('Figure_5b_heatmap_' + name_database + '_D' + str(D+1) + '.png')

print('Average training error:')
print('\[')
print(bmatrix(Etrain))
print('\]')
print('Average validation error:')
print('\[')
print(bmatrix(Evalid))
print('\]')
print('\includegraphics[width=1.0\textwidth]{Figure_5b_mse_'+ name_database + r'}\\')

plt.figure()
plt.plot(np.linspace(1, KD, KD), Etrain, label='Average training error')
plt.plot(np.linspace(1, KD, KD), Evalid, label='Average validation error')
plt.xlabel('Degree of polynomial')
plt.ylabel('Average error')
plt.title(name_database)
plt.legend()
plt.savefig('Figure_5c2_mse_'+ name_database +'.png')

def heatmap(X, y, coef, degree, 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

xx = yy = np.linspace(np.min(X), np.max(X), 72)
x0, y0 = np.meshgrid(xx, yy)
x0, y0 = x0.ravel(), y0.ravel()
z0 = get_polyvalue(x0, y0, coef, degree)
# print(z0)

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

plt.figure()
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)
```

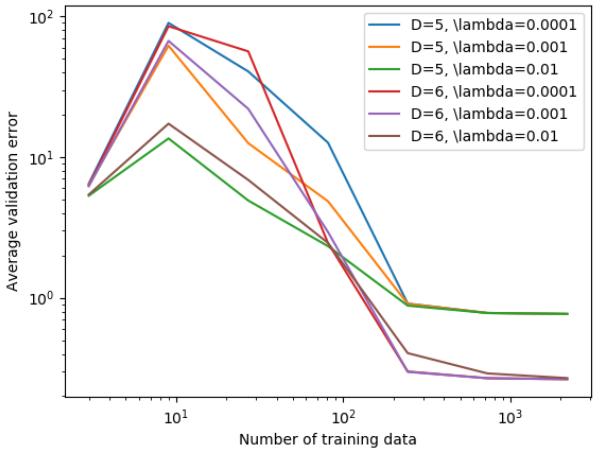
```
pos = y[:] == +1.0
neg = y[:] == -1.0
plt.scatter(X[pos, 0], X[pos, 1], c='red', marker='+')
plt.scatter(X[neg, 0], X[neg, 1], c='blue', marker='v')
# plt.show()
return plt

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

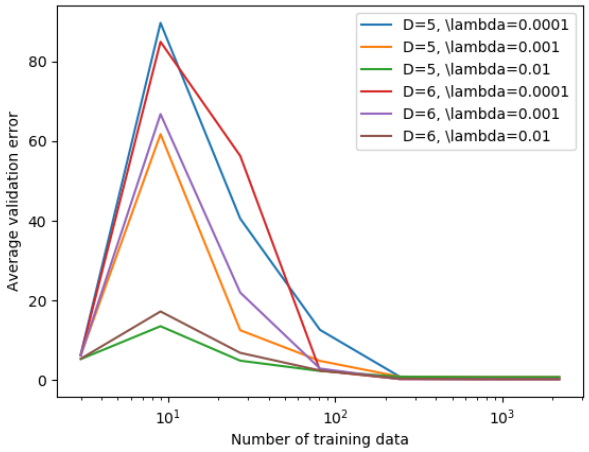
(d) (Diminishing influence of the prior with growing amount of data) With increasing of amount of data, the prior (from the statistical view) and regularization (from the optimization view) will be washed away and become less and less important. Sample the training data from the first 80% data from `asymmetric.npz` and use the data from the last 20 % data **Make a plot whose x axis is the amount of the training data and y axis is the validation error of the non-kernelized ridge regression algorithm. 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 error will converge with enough data, regardless of the choice of λ and the regularizer. 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.

It shows that large sample size will wash out the effect of λ.
Non-kernelized ridge regression:



The following are the graphs of kernel ridge regression:






```
import math
import matplotlib.pyplot as plt
import numpy as np

SPLIT = 0.15
KD = [5, 6] # max D = 16
LAMBDA = [0.0001, 0.001, 0.01]
np.random.seed(0)
data_names = ('asymmetric',)


def bmatrix(a):
"""Returns a LaTeX bmatrix
Retrieved from https://stackoverflow.com/questions/17129290/numpy-2d-and-1d-array-to-latex-bmatrix
:a: numpy array
:returns: LaTeX bmatrix as a string
"""
if len(a.shape) > 2:
raise ValueError('bmatrix can at most display two dimensions')
lines = str(a).replace('[', '').replace(']', '').splitlines()
rv = [r'\begin{bmatrix}']
rv += [' ' + ' & '.join(l.split()) + r'\\' for l in lines]
rv += [r'\end{bmatrix}']
return '\n'.join(rv)


def ridge_none_kernel(A, b, lambda_):
# Make sure data are centralized
return np.linalg.solve(A.T.dot(A) + lambda_ * np.eye(A.shape[1]), A.T.dot(b))


def get_error_none_kernel(X, w, y):
return np.mean((X.dot(w) - y) ** 2)


def fit_none_kernel(D, lambda_, train_x, train_y, validation_x, validation_y):
# YOUR CODE TO COMPUTE THE AVERAGE ERROR PER SAMPLE
errors = np.asarray([0.0, 0.0])
poly_train_x = polynomial(train_x, D)
w = ridge_none_kernel(poly_train_x, train_y, lambda_)
errors[0] = get_error_none_kernel(poly_train_x, w, train_y)
errors[1] = get_error_none_kernel(polynomial(validation_x, D), w, validation_y)
return errors, w
```

```
def polynomial(poly_data, degree):
if poly_data.ndim != 2:
pass
nvar = poly_data.shape[1]
npoints = poly_data.shape[0]
results = np.ones((npoints, comb(degree + nvar, degree)))
if degree == 0:
return results
start = 0
counts = np.zeros(nvar, dtype=int)
cur_index = 1
cur_len = 1
for d in range(1, degree + 1):
last_len = cur_len
cur_start = 0
cur_len = 0
for i in range(0, nvar):
for j in range(start + cur_start, start + last_len):
results[:, cur_index] = poly_data[:, i] * results[:, j]
cur_index += 1
temp = counts[i]
counts[i] = last_len - cur_start
cur_start += temp
cur_len += counts[i]
start = start + last_len

return results


def comb(n, k):
result = 1
for i in range(k):
result *= (n - i)
for i in range(k):
result /= (k - i)
return int(np.round(result))


def kernel_ridge(gram_matrix, b, lambda_):
# Make sure data are centralized
return np.linalg.solve(gram_matrix + lambda_ * np.eye(gram_matrix.shape[1]), b)


def get_error_kernel(X, alpha, y):
return np.mean((X.dot(alpha) - y) ** 2)


def fit_kernel(D, lambda_, train_x, train_y, validation_x, validation_y):
# YOUR CODE TO COMPUTE THE AVERAGE ERROR PER SAMPLE
errors = np.asarray([0.0, 0.0])
alpha = kernel_ridge(kernel(train_x, train_x, D), train_y, lambda_)
# no need to calculate training error for this question
# errors[0] = get_error(kernel(train_x, train_x, D), alpha, train_y)
errors[1] = get_error_kernel(kernel(validation_x, train_x, D), alpha, validation_y)
return errors, alpha


def kernel(phi_x, phi_z, degree):
nx = phi_x.shape[0]
nz = phi_z.shape[0]
gram_matrix = np.zeros((nx, nz))
for i in range(nx):
xi = phi_x[i, :]
```

```
for j in range(nz):
xj = phi_z[j, :]
gram_matrix[i, j] = (1 + xi.T.dot(xj)) ** degree
return gram_matrix


def main():
np.set_printoptions(precision=11)

for i_database, name_database in enumerate(data_names):
# choose the data you want to load
data = np.load(name_database + '.npz')
X = data["x"]
y = data["y"]
X /= np.max(X) # normalize the data
n_train = int(X.shape[0] * SPLIT)
# X_train_all = X[:n_train:, :]
X_valid = X[n_train:, :]
# y_train_all = y[:n_train]
y_valid = y[n_train:]

replicates = 10
log_base = 3
n_train_10fold = int(math.log(n_train, log_base))
x_spec = np.zeros(n_train_10fold)
Evalid_kernel = np.zeros((n_train_10fold, len(KD), len(LAMBDA)))
Evalid_none_kernel = np.zeros((n_train_10fold, len(KD), len(LAMBDA)))

for iSample in range(n_train_10fold):
print(iSample + 1, ' of ', n_train_10fold)
this_sample_size = log_base ** (iSample + 1)
x_spec[iSample] = this_sample_size
# Etrain = np.zeros((len(KD),len(LAMBDA)))
for iReplicate in range(replicates):
indexes = np.asarray(np.random.choice(n_train, this_sample_size), dtype=int)
X_train = X[indexes, :]
y_train = y[indexes]
for iD, D in enumerate(KD):
# print(D + 1)
for ilambda, lmbda in enumerate(LAMBDA):
[_, temp], _ = fit_kernel(D, lmbda, X_train, y_train, X_valid, y_valid)
Evalid_kernel[iSample, iD, ilambda] += temp
[_, temp], _ = fit_none_kernel(D, lmbda, X_train, y_train, X_valid, y_valid)
Evalid_none_kernel[iSample, iD, ilambda] += temp

Evalid_kernel /= replicates
Evalid_none_kernel /= replicates
# plt.plot(np.linspace(1, KD, KD), Etrain, label='Average training error')
plt.figure()
for iD, D in enumerate(KD):
for ilambda, lmbda in enumerate(LAMBDA):
plt.semilogx(x_spec, Evalid_kernel[:, iD, ilambda], label='D=' + str(D) + ', \lambda=' + str(lmbda))
plt.xlabel('Number of training data')
plt.ylabel('Average validation error')
plt.legend()
plt.savefig('Figure_5d_kernel_semilogx_' + name_database + '.png')
plt.close()

plt.figure()
for iD, D in enumerate(KD):
for ilambda, lmbda in enumerate(LAMBDA):
plt.loglog(x_spec, Evalid_kernel[:, iD, ilambda], label='D=' + str(D) + ', \lambda=' + str(lmbda))
plt.xlabel('Number of training data')
```

```
plt.ylabel('Average validation error')
plt.legend()
plt.savefig('Figure_5d_kernel_loglog_' + name_database + '.png')
plt.close()

plt.figure()
for iD, D in enumerate(KD):
for ilambda, lmbda in enumerate(LAMBDA):
plt.semilogx(x_spec, Evalid_none_kernel[:, iD, ilambda], label='D=' + str(D) + ', \lambda=' + str(lmbda))
plt.xlabel('Number of training data')
plt.ylabel('Average validation error')
plt.legend()
plt.savefig('Figure_5d_nokernel_semilogx_' + name_database + '.png')
plt.close()

plt.figure()
for iD, D in enumerate(KD):
for ilambda, lmbda in enumerate(LAMBDA):
plt.loglog(x_spec, Evalid_none_kernel[:, iD, ilambda], label='D=' + str(D) + ', \lambda=' + str(lmbda))
plt.xlabel('Number of training data')
plt.ylabel('Average validation error')
plt.legend()
plt.savefig('Figure_5d_nokernel_loglog_' + name_database + '.png')
plt.close()


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

$$K(\vec{x}, \vec{x}') = \exp\left(-\frac{\|\vec{x} - \vec{x}'\|_2^2}{2\sigma^2}\right). \tag{7}$$

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 error, 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**.**

heart

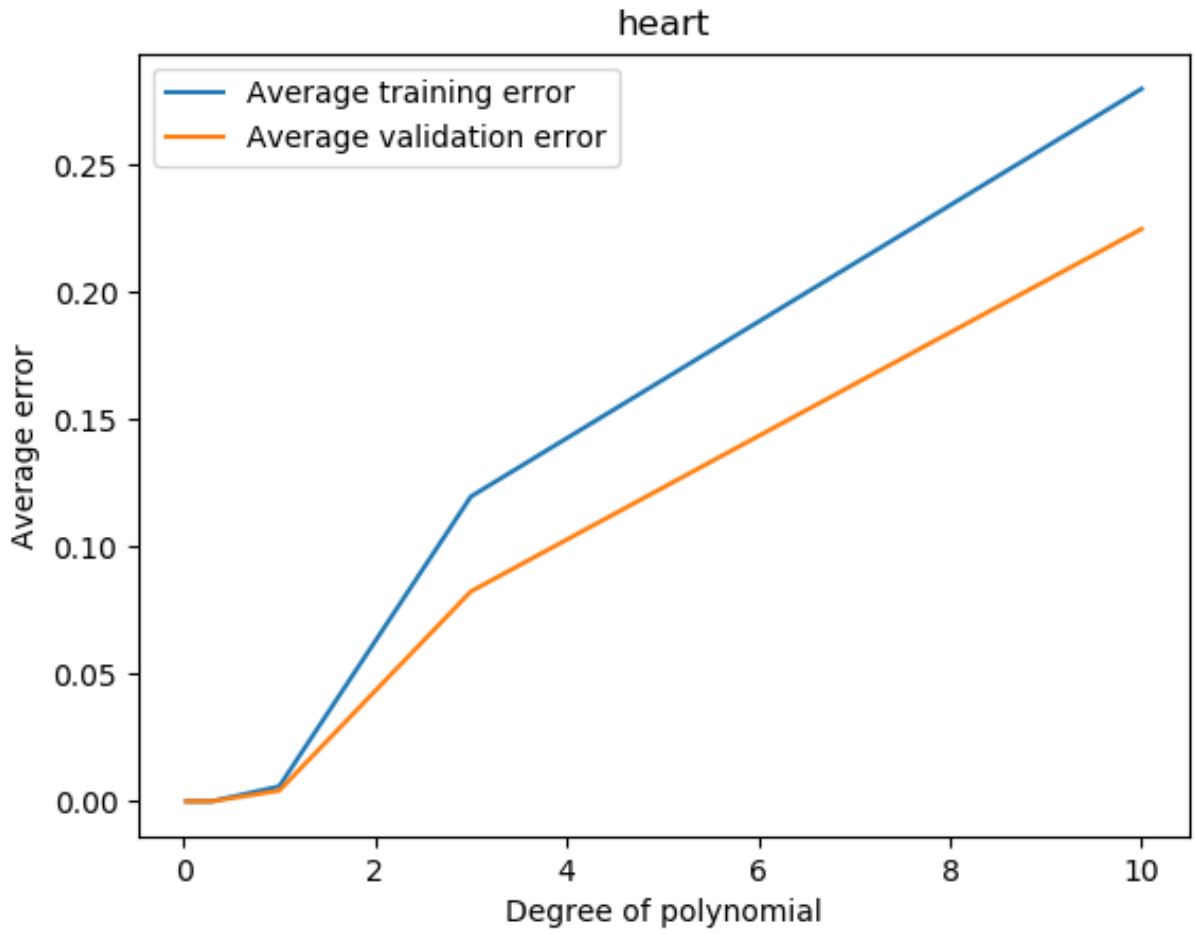
Degree of polynomial here should read σ

Average training error:

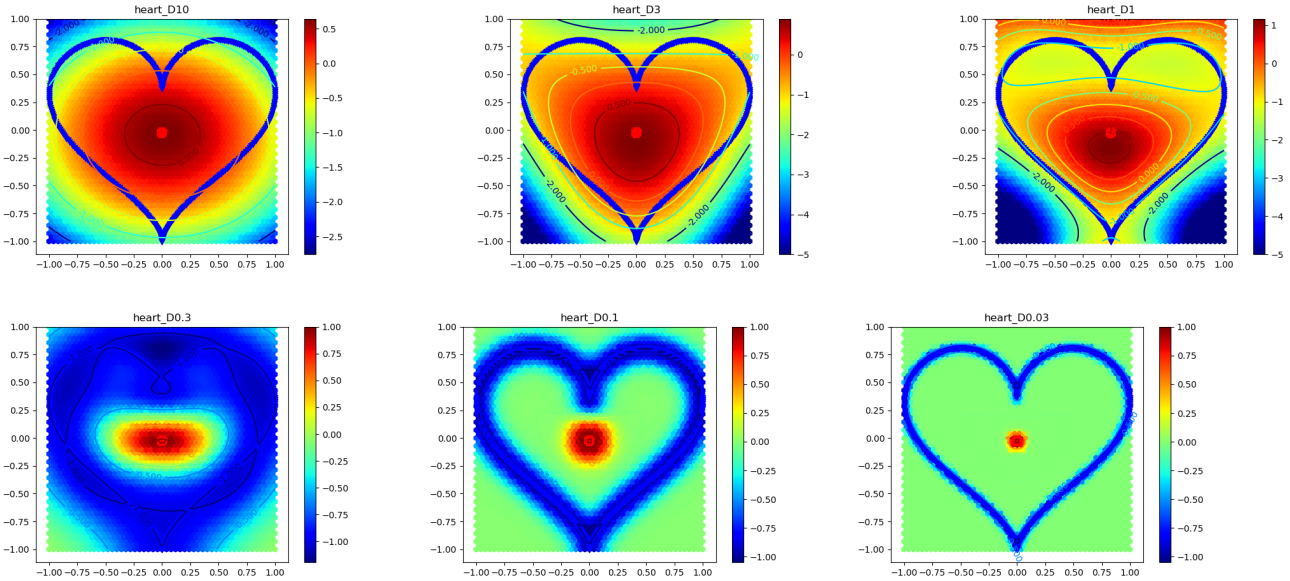
$$\begin{bmatrix} 2.79652764795e-01 & 1.19629185940e-01 & 5.87217076866e-03 & 5.27549858671e-05 \\ 8.49292997250e-08 & 2.47081162311e-07 \end{bmatrix}$$

Average validation error:

$$\begin{bmatrix} 2.24638432866e-01 & 8.23787147171e-02 & 4.20109547686e-03 & 4.98793115613e-05 \\ 1.14997881045e-07 & 7.76855004079e-05 \end{bmatrix}$$



Now we visualize our result:



We can see that as σ decreases, the hot area shrinks making the peak sharper. Therefore, σ controls the spread of predictor.

```
import matplotlib.pyplot as plt
from matplotlib import cm
```

```
import numpy as np
import scipy.io as spio

SPLIT = 0.8
LAMBDA = 0.001
SIGMA = [10, 3, 1, 0.3, 0.1, 0.03]

data_names = ('heart', )

def bmatrix(a):
"""Returns a LaTeX bmatrix
Retrieved from https://stackoverflow.com/questions/17129290/numpy-2d-and-1d-array-to-latex-bmatrix
:a: numpy array
:returns: LaTeX bmatrix as a string
"""
if len(a.shape) > 2:
raise ValueError('bmatrix can at most display two dimensions')
lines = str(a).replace('[', '').replace(']', '').splitlines()
rv = [r'\begin{bmatrix}']
rv += [' ' + ' & '.join(l.split()) + r'\\' for l in lines]
rv += [r'\end{bmatrix}']
return '\n'.join(rv)

def kernel_ridge(gram_matrix, b, lambda_):
# Make sure data are centralized
return np.linalg.solve(gram_matrix + lambda_ * np.eye(gram_matrix.shape[1]), b)


def get_error(X, alpha, y):
return np.mean((X.dot(alpha) - y) ** 2)


def fit(sigma, lambda_, train_x, train_y, validation_x, validation_y):
# YOUR CODE TO COMPUTE THE AVERAGE ERROR PER SAMPLE
errors = np.asarray([0.0, 0.0])
alpha = kernel_ridge(gaussian_kernel(train_x, train_x, sigma), train_y, lambda_)
errors[0] = get_error(gaussian_kernel(train_x, train_x, sigma), alpha, train_y)
errors[1] = get_error(gaussian_kernel(validation_x, train_x, sigma), alpha, validation_y)
return errors, alpha


def gaussian_kernel(phi_x, phi_z, sigma):
nx = phi_x.shape[0]
nz = phi_z.shape[0]
gram_matrix = np.zeros((nx, nz))
for i in range(nx):
xi = phi_x[i, :]
for j in range(nz):
xj = phi_z[j, :]
gram_matrix[i, j] = np.sum((xi - xj) ** 2)
gram_matrix = np.exp(- gram_matrix/(2 * sigma * sigma))
return gram_matrix


def get_kernelvalue(x0, y0, X_train, coefficient, degree):
N = len(y0)
result = np.sum(gaussian_kernel(np.column_stack([x0, y0]), X_train, degree) * np.vstack([coefficient] *
    N), axis=1)
return result


def main():
np.set_printoptions(precision=11)
```

```
len_sigma = len(SIGMA)
for i_database, name_database in enumerate(data_names):
# choose the data you want to load
data = np.load(name_database + '.npz')
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:]
Etrain = np.zeros(len_sigma)
Evalid = np.zeros(len_sigma)
print(r'\textbf{' + name_database + r'}\\')
for iSigma, sigma in enumerate(SIGMA):
# print(D + 1)
[Etrain[iSigma], Evalid[iSigma]], weights = fit(sigma, LAMBDA, X_train, y_train, X_valid, y_valid)
heatplt = heatmap(X, y, X_train, weights, sigma, clip=5)
heatplt.title(name_database + '_D' + str(sigma))
heatplt.savefig('Figure_5e_heatmap_' + name_database + '_sigma=' + str(sigma) + '.png')
heatplt = heatmap(X, y, X_train, weights, sigma, clip=0)
heatplt.title(name_database + '_D' + str(sigma))
heatplt.savefig('Figure_5e_heatmap_noclip_' + name_database + '_sigma=' + str(sigma) + '.png')

print('Average training error:')
print('\[')
print(bmatrix(Etrain))
print('\]')
print('Average validation error:')
print('\[')
print(bmatrix(Evalid))
print('\]')
print('\includegraphics[width=1.0\textwidth]{Figure_5e_mse_'+ name_database + r'}\\')
plt.figure()
plt.plot(SIGMA, Etrain, label='Average training error')
plt.plot(SIGMA, Evalid, label='Average validation error')
plt.xlabel('Degree of polynomial')
plt.ylabel('Average error')
plt.title(name_database)
plt.legend()
plt.savefig('Figure_5e_mse_'+ name_database +'.png')


def heatmap(X, y, X_train, coef, degree, 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

xx = yy = np.linspace(np.min(X), np.max(X), 72)
x0, y0 = np.meshgrid(xx, yy)
x0, y0 = x0.ravel(), y0.ravel()
z0 = get_kernelvalue(x0, y0, X_train, coef, degree)
# print(z0)

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

plt.figure()
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')
# plt.show()
return plt

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

(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.**

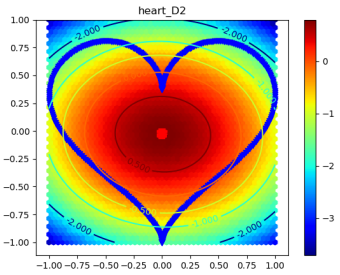
If the training sample size is large, non-kernelized one is better; If the polynomial degree is high, kernelized one is better.

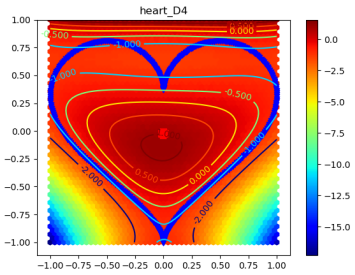
It's not possible to implement the non-kernelized version of RBF kernel because it has infinite polynomial degree.

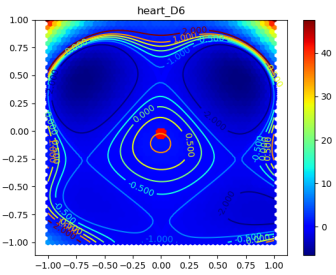
(g) Disable the `clip` option in the provided `heatmap` function and redraw the heatmap plots for the functions learned by the polynomial kernel and RBF kernel. Experiment on the provided datasets and **describe one potential problem of the polynomial kernel related to what you see here.** Does the RBF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

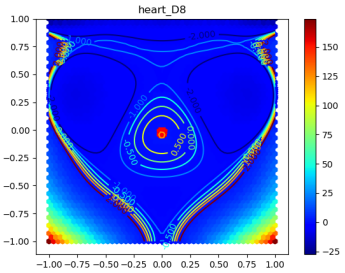
Polynomial kernel is not stable because it gives crazy predictions for the range that is not covered by the training data. On the other hand, the prediction given by RBF kernel is more stable and converge to a relatively expected solution.

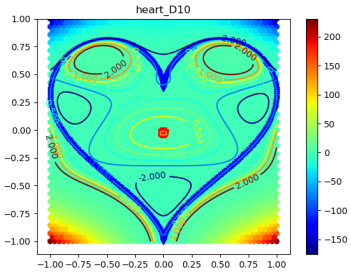
Polynomial kernel of heart

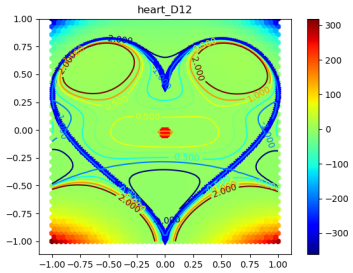




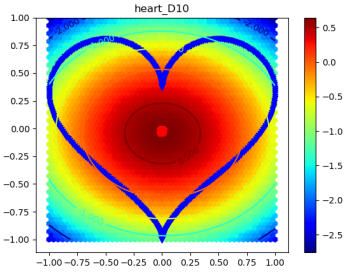


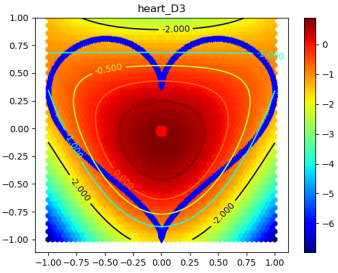


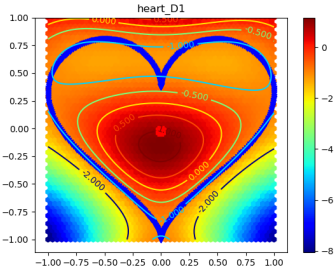


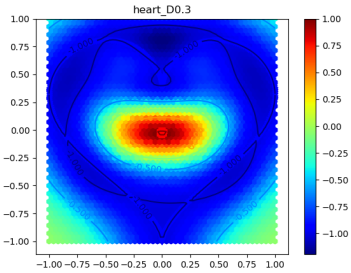


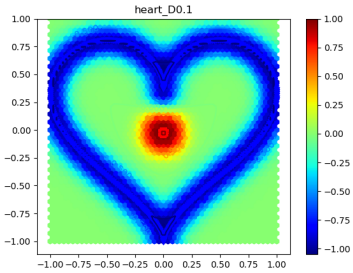
RBF kernel of heart

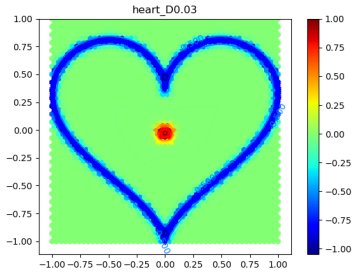












Question 6. Your Own Question

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.

I'm a biology student who has never taken EE127. Therefore, my goal here is to understand what Lagrangian multiplier is. I found a pretty good source explaining the problem using a geometric example (`http://www.eng.newcastle.edu.au/eecs/cdsc/books/cce/Slides/Duality.pdf`). However, this is not sufficient to be my own question. What should I do? I'm not interested in proving it. Let's do an example then.
Consider the problem:

$$\begin{aligned} \min \quad & x_1^2 + 2x_2^2 + 2x_1 + 8x_2 \\ \text{subjected to} \quad & -x_1 - 2x_2 + 10 \leq 10 \\ & x \geq 0 \end{aligned}$$

Let's write our the dual function:

$$L(x_1, x_2, \lambda) = x_1^2 + 2x_2^2 + 2x_1 + 8x_2 + \lambda(-x_1 - 2x_2 + 10)$$

Now we need to check Slater's condition for strong duality of the problem (`https://en.wikipedia.org/wiki/Slater%27s_condition`). We notice that the coefficients in the first equation are all positive and therefore it's a convex function. Two constraints are all linear functions. Then we converted the original primal problem into its dual problem in the dual function where

$$\min_{x \geq 0} \max_{\lambda \geq 0} L(x_1, x_2, \lambda) = \max_{\lambda \geq 0} \min_{x \geq 0} L(x_1, x_2, \lambda)$$

Then let's take the derivatives of Lagrangian equation:

$$\begin{cases} \nabla_{x_1} L(x_1, x_2, \lambda) = 2x_1 + 2 - \lambda = 0 \\ \nabla_{x_2} L(x_1, x_2, \lambda) = 4x_2 + 8 - 2\lambda = 0 \\ \nabla_{\lambda} L(x_1, x_2, \lambda) = 10 - x_1 - 2x_2 = 0 \end{cases}$$

Solving this gives us:

$$\begin{cases} x_1 = 4 \\ x_2 = 3 \\ \lambda = 10 \end{cases}$$

We substitute this into our solution gives us:

$$x_1^2 + 2x_2^2 + 2x_1 + 8x_2 = 66$$

We can also check and make sure this is a convex optimization problem by taking the second derivatives. However, I'm too lazy to show that.
I like to take derivatives!
The original problem of this question can be found here. (`http://homepages.rpi.edu/~mitchj/handouts/ldeg/ldeg.html`)