Name: Honda Li Class: CSE 546 SPRING 2021 HW3A

A1: Conceptual Questions

A.1.a

Decraese σ. This makes the function $\exp\left(-\frac{\|u-v\|_2^2}{2\sigma^2}\right)$ thinner, make the inner product between different points more distinct.

A.1.b

True. It's an non-convex objective function, assuming thta deep means more than one hidden layer of course and assuming that activation function is not linear.

A.1.c

Yes it is. This ties to the fact that the Loss function of the Neural net is not convex, and if we start near all zeros, that means the first few iterations of the Gradient descent will aways get the same gradient, hugely limiting weight configuration of the model.

A.1.d

Flase, the what gives non linear decision boundary is the number of layers and the number of neurons in the layers, both ReLU and the Sigmoid functions is using linear decision boundary under the hood, **it's a lot of linear decision bounary combined together**.

A.1.e

A.2: Kernels And Bootstrap

Give na vector whose n th components paramterized by n is given by:

$$\frac{1}{\sqrt{n!}}\exp\left(\frac{-x^2}{2}\right)x^n$$

where x is one dimensional, and the feature mapping function $\phi(x)$ is an infinite dimensional function. Then:

$$\begin{aligned}
\langle\phi(x),\phi(y)\rangle &= \sum_{n=1}^{\infty}\phi_n(x)\phi_n y \qquad \text{(A.2.1)}\\
&= \sum_{n=1}^{\infty}\frac{1}{\sqrt{n!}}\exp\left(-\frac{x^2}{2}\right)x^n\frac{1}{\sqrt{n!}}\exp\left(-\frac{y^2}{2}\right)y^n\\
&= \sum_{n=1}^{\infty}\frac{(xy)^n}{n!}\exp\left(-\frac{x^2+y^2}{2}\right)\\
&= \exp\left(-\frac{x^2+y^2}{2}\right)\sum_{n=1}^{\infty}\frac{(xy)^n}{n!}\\
&= \exp\left(-\frac{x^2+y^2}{2}\right)\exp\left(xy\right)\\
&= \exp\left(-\frac{x^2+y^2}{2}+\frac{2xy}{2}\right)\\
&= \exp\left(\frac{-(x-y)^2}{2}\right)
\end{aligned}$$

And this is the RBF kernel for a scalar, in the 1d case.

A.3: Kernel Ridge Regression

For problem A.3, this is the core routine I used for the whole problem, filename: "kernel_ridge_regression"

```
### This is a script for CSE 546 SPRING 2021, HW3, A.3
### Implementing the kernel ridge regression and visualize some stuff.
### Author: Hongda Li

import numpy as np
from scipy import linalg
import matplotlib.pyplot as plt

linspace = np.linspace
randn = np.random.randn
pinvh = linalg.pinvh
inv = linalg.inv
eye = np.eye
mean = np.mean
std = np.std

class KernelRidge:

    def __init__(this, regularizer_lambda, kernelfunc: callable):
        """

        :param regularizer_lambda:
        :param kernelfunc:
            Takes in the WHOLE traning matrix and compute the kenrnel matrix K.

        """
        this.Lambda = regularizer_lambda
        this.KernelMatrix = None
        this.X = None
        this.Kernel = kernelfunc
        this.Alpha = None
        this.Bias = None

    @property
    def w(this):
        if this.X is None: return None
        return this.X.T@this.Alpha

    def fit(this, X, y):
        """

        :param x:
        :param y:
        :return:
        """
        assert type(X) is np.ndarray and type(y) is np.ndarray, "X, y, must be numpy array"
        assert X.ndim == 2 and y.ndim <= 2
        Warn = "X, y dimension problem"
        if y.ndim == 2:
            assert y.shape[0] == X.shape[0], Warn
            assert y.shape[1] == 1, Warn
        else:
            assert y.shape[0] == X.shape[0], Warn
            y = y[:, np.newaxis]
        assert X.shape[0] >= 1, "Need more than just one sample. "
        # Standardized.
        this.X = X
        n, d = X.shape
        Lambda = this.Lambda
        K = this.Kernel(this.X, this.X)
        assert K.ndim == 2 and K.shape[0] == K.shape[1] and K.shape[0] == n, \
```

```
            "your␣kernel␣function␣implemention␣is␣wrong,␣kernel␣matrix␣is␣having␣the␣wrong␣shape"
        assert np.all(np.abs(K-K.T) < 1e-5), "kernel␣matrix␣is␣not␣symmetric."
        this.KernelMatrix = K
        # get the bias

        # get the alpha.
        this.Alpha = pinvh(K + Lambda*eye(n))@y


    def predict(this, Xtest):
        assert this.X is not None, "Can't␣predict␣when␣not␣trained␣yet.␣"
        Xtrain = this.X
        return this.Kernel(Xtest, Xtrain)@this.Alpha



def main():
    def SimpleTest():
        N = 100
        w, b = 1, 0
        x = linspace(-1, 1, N)
        eps = randn(N)*0.1
        y = w*x + b + eps
        X = x[:, np.newaxis]
        def KernelFunc(X, Y):
            return X@Y.T
        Model = KernelRidge(regularizer_lambda=0.01, kernelfunc=KernelFunc)
        Model.fit(X, y)
        Yhat = Model.predict(X)
        plt.plot(x, y)
        plt.plot(x, Yhat)
        plt.show()
    SimpleTest()


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

A.3.a

To implement, we will need to take care of the process of solving for the best parameter α, and we will also need to careful about the offset. So what we are going to train is on the zero mean data, and then the prediction made from the model will have to add back the offset from the training set of course.
Here is basically what we had from the sections:

$$\begin{aligned}
&\frac{1}{2}\nabla_x[\|K\alpha - y\|_2^2 + \lambda\alpha^T K\alpha] = 0 \qquad \text{(A.3.a)}\\
&\implies K^T(K\alpha - y) + \lambda K\alpha = 0\\
&\qquad K(K\alpha - y) + \lambda K\alpha = 0\\
&\qquad KK\alpha - Ky + \lambda K\alpha = 0\\
&\qquad KK\alpha + \lambda K\alpha = Ky\\
&\qquad K\alpha + \lambda\alpha = y\\
&\qquad \alpha = (K + \lambda I)^{-1}y
\end{aligned}$$

Where, K and y are from the training set. And in this case, the predictor can be computed via: $K_{\text{test,train}}\alpha$ where the $K_{\text{test,train}}$ is computed via:

$$K_{i,j} = \langle \phi(X_{\text{test}}[i,:]), \phi(X_{\text{train}}[:,j]) \rangle$$

To implement the method, I used the following tricks:

1. All the questions from a, b, c, d, e, are running on a fixed sample. When I do the black box optimization, and grid search for hyper parameter, I want to make the whole algorithm deterministic, given a certain sample set.

2. I used a black box optimization to improve the grid search algorithm, The algorithm is SHGO, Simplicial Homology Global Optimization. It's used as a bounded non-convex optimization algorithm as a subroutine for the grid search algorithm.

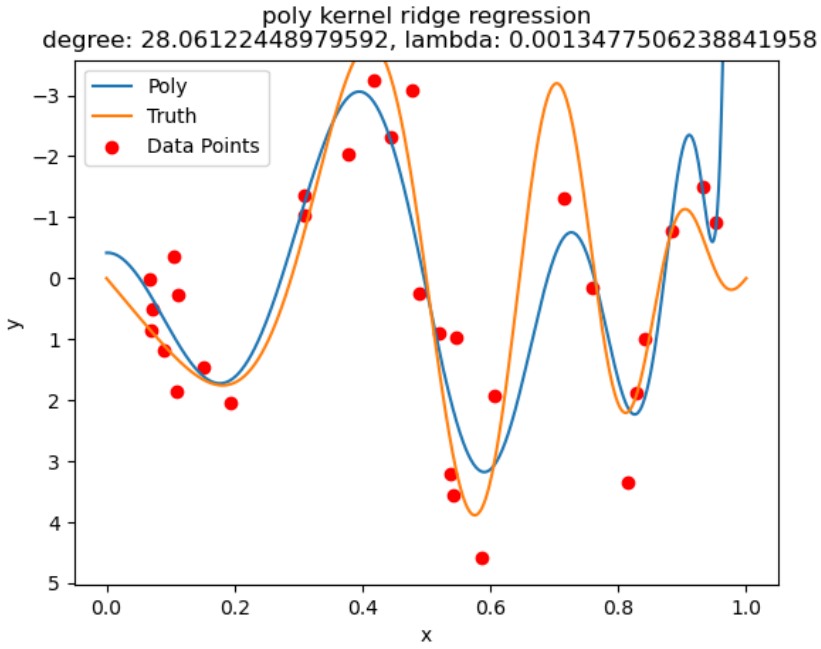
Note: That hyperparameter search algorithm I wrote runs forever. **Note**: For 30 samples, there are a lot of variance on the models, and here is one of the hyper parameter identified by my algorithm:

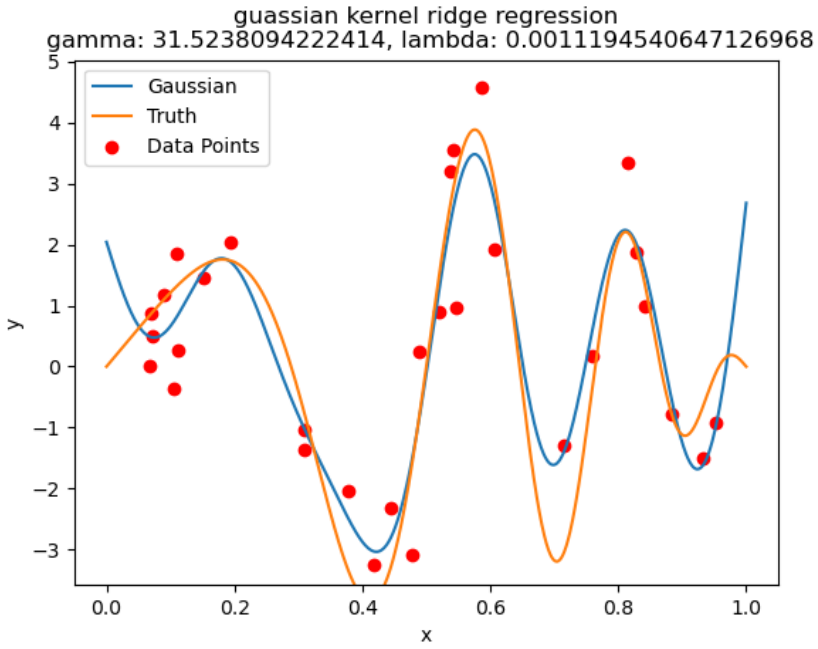
1. For poly kernel: $d = 28.06$, $\lambda = 0.0013477$.

2. For the KBF kernel: $\gamma = 31.5238$, $\lambda = 0.001119$.

A.3.b

This is the plot I had for the particular identified hyper parameters:



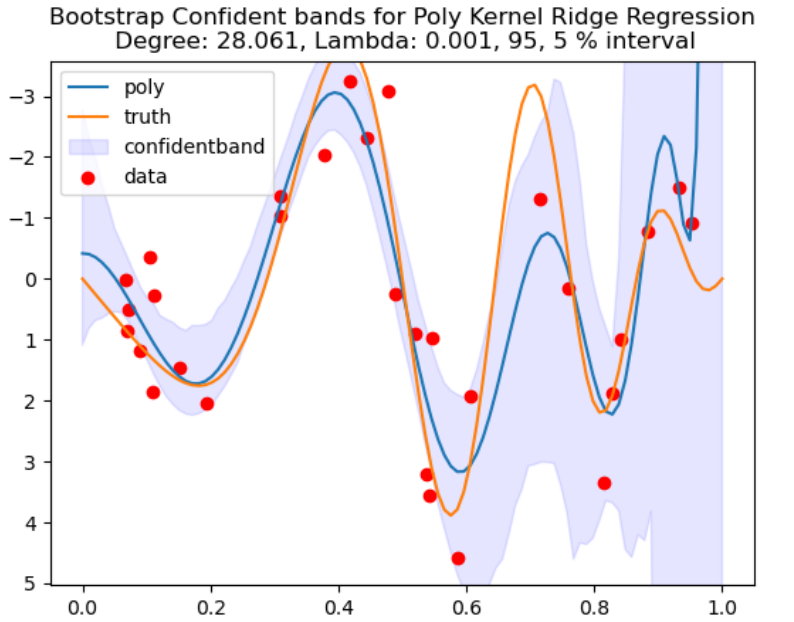


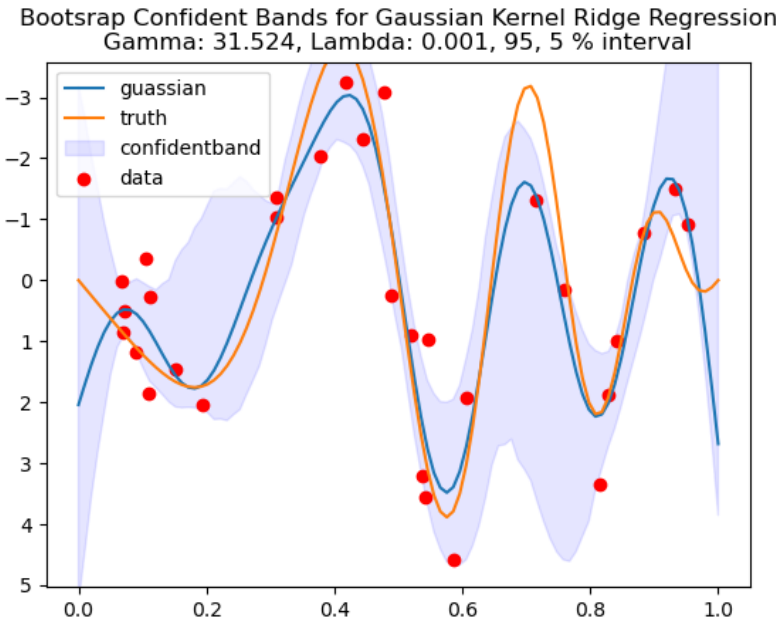
A.3.c

For building the boopstrap, this is something about the implementation that I think it's worth noting:

1. I am still using the same set of samples that produces the hyperparameters.
2. Because of huge variance for the poly model at the boundary of the data set, I have to scale the graph so that we can see what is happening, instead of zooming out like crazy.

Here is the graph for both the model:

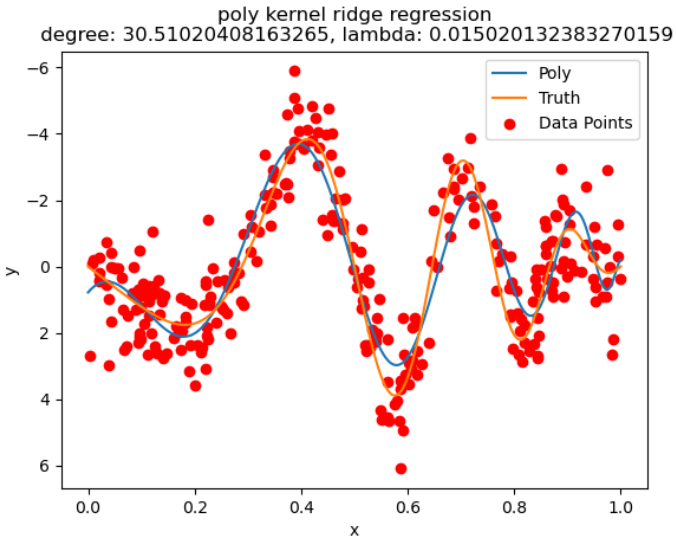


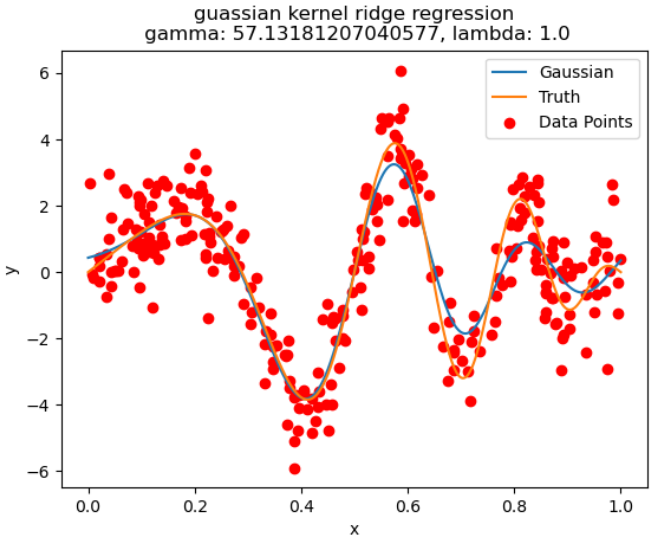


And there are a lot of variance. Very dependent on the sample too.

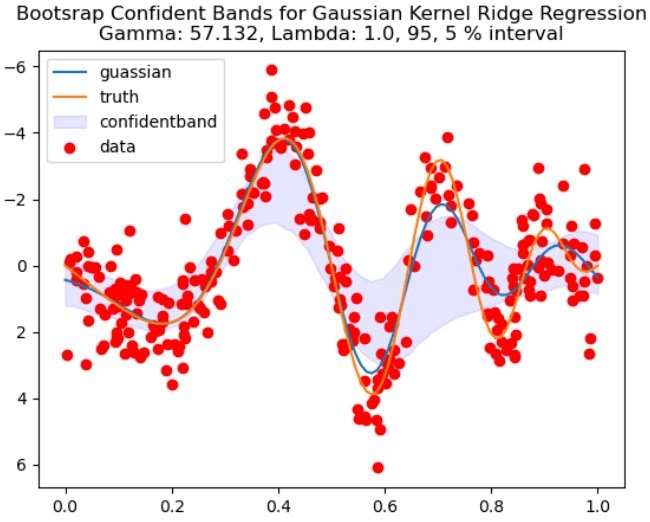
A.3.d

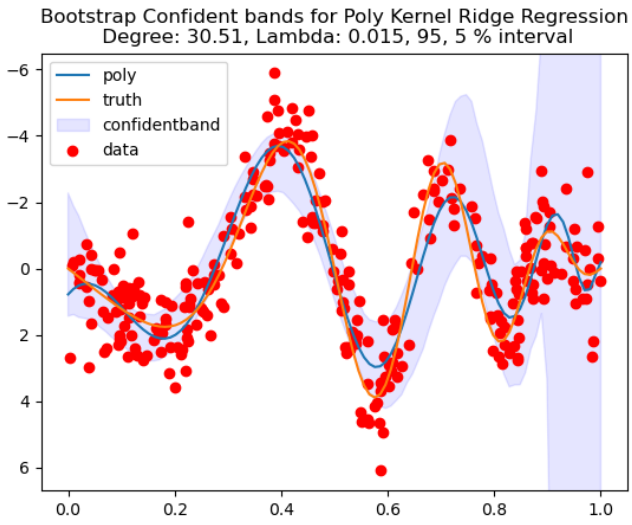
This is the repeated experiement with a sample size of 300 and a 10 Fold cross validation:





And this is the confident bands I placed on these models:





A.3.e

A.3.Code

This is the main driver code I used for all the questions, it's quiet complicated so I have put it in the end of the section.
File name: "A3.py"

```
### This is the script that produce plots and data for A3
### This is for CSE 546 SPRING 2021, HW3.
### Author: Hongda Alto Li
### Requries: kernel_ridge_regression.py


import numpy as np
cos, sin, pi = np.cos, np.sin, np.pi
rand, randn, randint = np.random.rand, np.random.randn, np.random.randint
norm = np.linalg.norm
zeros = np.zeros
mean = np.mean
sum = np.sum
min = np.min
max = np.max
linspace = np.linspace
logspace = np.logspace
percentile = np.percentile
var = np.var
from kernel_ridge_regression import KernelRidge
from sklearn.metrics.pairwise import rbf_kernel, polynomial_kernel
from sklearn.model_selection import KFold
import matplotlib.pyplot as plt
from scipy.optimize import minimize, shgo
from scipy.optimize import Bounds

### Some constant for the whole script:



def RBFKernel(X, Y, gamma):
    """
        Use this kernel to take the inner products between the columns of X, Y
    :param x:
    :return:
```

```
    """

    return rbf_kernel(X, Y, gamma=gamma)


def MyPolyKernel(X, Y, d):
    """
        Use this kernel to take the inner product between the columns of the X, Y
        matrix.
    :param x:
    :return:
    """
    if Y is None: X = Y
    return polynomial_kernel(X, Y, gamma=1, degree=d)


def CrossValErrorEstimate(X, y, regualarizer, kernelfunc, split=None,param_norm=False):
    Errors = []
    AlphaNorm = []
    if split is None:
        split = X.shape[0]
    kf = KFold(n_splits=split, shuffle=False) # Make it deterministic given the same training
        sample.

    for TrainIdx, TestIdx in kf.split(X):
        Model = KernelRidge(regularizer_lambda=regualarizer, kernelfunc=kernelfunc)
        Model.fit(X[TrainIdx], y[TrainIdx])
        yhat = Model.predict(X[TestIdx]).reshape(-1)
        Error = (sum(yhat - y[TestIdx])**2)/len(TestIdx)
        Errors.append(Error)
        AlphaNorm.append(norm(Model.w, np.inf))
    if param_norm:
        return mean(Errors), min(AlphaNorm)
    return mean(Errors)


def GenerateXY(n):
    f = lambda x: 4 * sin(pi * x) * cos(6 * pi * x ** 2)
    x = rand(n)
    x.sort()
    y = f(x) + randn(n)
    return x[:, np.newaxis], y


def main(n=30, KfoldSplit=30):
    X, y = GenerateXY(n)   #  THIS IS SHARED! FOR ALL
    f = lambda x: 4 * sin(pi * x) * cos(6 * pi * x ** 2)
    def PolyKernelHypertune():
        def GetError(deg, l):
            Kernefun = lambda x, y: MyPolyKernel(x, y, deg)
            Error = CrossValErrorEstimate(X,
                                          y,
                                          regualarizer=l,
                                          kernelfunc=Kernefun,
                                          split=KfoldSplit)
            return Error
        BestError = float("inf")
        Best = None
        for Deg in np.linspace(7, 31):
            Result = shgo(lambda x: GetError(Deg, x),
                bounds=[(0, 0.05)],
                n=100,
                sampling_method="simplicial",
                options={"f_tol": 1e-8}
                          )
            if Result.fun < BestError:
                print(f"Poly␣Kernel␣Best␣error␣update␣for␣deg:␣{Deg},␣Lambda:␣{Result.x},␣Error:␣
                    {Result.fun}")
```

```
            BestError = Result.fun
            Best = (Deg, Result.x[0])
        else:
            print(f"failed␣at:␣deg:␣{Deg},␣Lambda:␣{Result.x[0]},␣Error:␣{Result.fun}")

    # Result = shgo(lambda x: GetError(x, 0),
    #      bounds=[(1, 100)],
    #      n=200, sampling_method="simplicial",
    #               options={"f_tol": 1e-8, "disp": True})
    # print(f"SHGO Optimization Results: {Result}")
    return Best

def GaussianKernelHypertune():
    # Grid search, Fix the training sample
    # X, y = GenerateXY()
    Xs = X.reshape(-1)
    Distance = []
    for II in range(Xs.size):
        for JJ in range(II + 1, Xs.size):
            Distance.append(1/(norm(Xs[II] - Xs[JJ])**2))
    GammaLower, GammaHigher = \
        percentile(Distance, 25), percentile(Distance, 75)
    print(f"Hypertune␣Gamma␣kernel␣search␣range:␣{GammaLower,␣GammaHigher}")
    def GetError(gamma, l):
        l, gamma = abs(l), abs(gamma)
        Kernelfun = lambda x, y: RBFKernel(x,y, gamma=gamma)
        Error = CrossValErrorEstimate(X,
                                      y,
                                      regualarizer=l,
                                      kernelfunc=Kernelfun,
                                      split=KfoldSplit
                                      )
        return Error
    # GRID SEARCH INITIAL GUESS
    Result = shgo(lambda x: GetError(x[0], x[1]),
                  bounds=[(GammaLower, GammaHigher), (0, 1)],
                  n=500, sampling_method='sobol',
                  options={"f_tol": 1e-4, "disp": True})
    print("Optimization␣results:␣")
    print(Result)
    print(f"Guassian␣Bestparams:␣{Result.x}")
    return Result.x
# ============ Hyper Param! ================================================
GaussianBest = [20, 0.058]
PolyBest = [19, 0.05]

GaussianBest = GaussianKernelHypertune()
PolyBest = PolyKernelHypertune()

print(f"guassian␣kernel␣best␣is:␣[gamma,␣lambda]␣{GaussianBest}")
print(f"Poly␣kernel␣best␣is:␣[deg,␣lambda]␣{PolyBest}")

def DrawPolyModel():
    x = linspace(0, 1, 1000)
    Model = KernelRidge(regularizer_lambda=PolyBest[1],
                        kernelfunc=
                        lambda X, Y: MyPolyKernel(X, Y, PolyBest[0]))
    Model.fit(X, y)
    plt.ylim([max(y)*1.1, min(y)*1.1])
    plt.plot(x, Model.predict(x[:, np.newaxis]).reshape(-1))
    plt.plot(x, f(x))
    plt.scatter(X.reshape(-1), y, c="red")
    plt.title(f"poly␣kernel␣ridge␣regression\n␣"
              f"degree:␣{PolyBest[0]},␣lambda:␣{PolyBest[1]}")
    plt.xlabel("x")
    plt.ylabel("y")
    plt.legend(["Poly", "Truth", "Data␣Points"])
    plt.savefig("A3b-poly.png")
```

```
    plt.show()
    return Model

BestPolyModel = DrawPolyModel()

def DrawGuassianModel():
    x = linspace(0, 1, 1000)
    Model = KernelRidge(regularizer_lambda=GaussianBest[1],
                        kernelfunc=
                        lambda X, Y: RBFKernel(X, Y, GaussianBest[0])
                        )
    Model.fit(X, y)
    plt.ylim([min(y)*1.1, max(y)*1.1])
    plt.plot(x, Model.predict(x[:, np.newaxis]).reshape(-1))
    plt.plot(x, f(x))
    plt.scatter(X.reshape(-1), y, c="red")
    plt.title(f"guassian␣kernel␣ridge␣regression\n␣"
              f"gamma:␣{GaussianBest[0]},␣lambda:␣{GaussianBest[1]}")
    plt.xlabel("x")
    plt.ylabel("y")
    plt.legend(["Gaussian", "Truth", "Data␣Points"])
    plt.savefig("A3b-gauss.png")
    plt.show()
    return Model

BestGaussianModel = DrawGuassianModel()

# ------------------------------------------------------------------------
# Boopstrap and estimating the confident interval.

def BoopStraping(Kernelfunc, Lambda, x):
    # Given the kernel func, produce the confidence interval.
    BagOfModels = []
    UpperPercentile = []
    LowerPercentile = []
    print("Boopstraping␣fitting␣the␣model")
    for _ in range(300):
        Indices = randint(0, n, 30)
        XTild, Ytild = X[Indices], y[Indices]
        Model = KernelRidge(
            kernelfunc=Kernelfunc,
            regularizer_lambda=Lambda
        )
        Model.fit(XTild, Ytild)
        BagOfModels.append(Model)
    ModelPredictions = np.array(
        [Model.predict(x[:, np.newaxis]).reshape(-1) for Model in BagOfModels])

    for II in range(ModelPredictions.shape[1]):
        UpperPercentile.append(percentile(ModelPredictions[:, II], 95))
        LowerPercentile.append(percentile(ModelPredictions[:, II], 5))

    return UpperPercentile, LowerPercentile

def BoopStrapModelDifference(GaussModel, PolyModel):
    m = 1000
    X, y = GenerateXY(m)
    AllMeanDiff = []
    print("solving␣A3(e)")
    for _ in range(300):
        Idx = randint(0, m, m)
        Idx.sort()
        Idx = np.unique(Idx)
        Xtild, ytild = X[Idx], y[Idx]
        yhat1 = GaussModel.predict(Xtild).reshape(-1)
        yhat2 = PolyModel.predict(Xtild).reshape(-1)
        Var1 = var(yhat1 - ytild)
        Var2 = var(yhat2 - ytild)
```

```
            AllMeanDiff.append(Var2 - Var1)
            print(f"Bootstrap␣sample:␣{_},␣The␣difference␣of␣MSE␣is:␣{AllMeanDiff[-1]}")
        UpperPercentile, LowerPercentile =\
            percentile(AllMeanDiff, 95), percentile(AllMeanDiff, 5)
        return LowerPercentile, UpperPercentile


    Xgrid = linspace(0, 1, 100)
    UpperPercentile, LowerPercentile = BoopStraping(
        lambda x, y: MyPolyKernel(x, y, PolyBest[0]),
        PolyBest[1], Xgrid
    )
    # Plot the polynomial Boop strap,
    plt.title("Bootstrap␣Confident␣bands␣for␣Poly␣Kernel␣Ridge␣Regression\n␣"
              f"Degree:␣{round(PolyBest[0],␣3)},␣Lambda:␣{round(PolyBest[1],␣3)},␣95,␣5␣%␣
                  interval")
    # plt.plot(Xgrid, UpperPercentile)
    # plt.plot(Xgrid, LowerPercentile)
    plt.ylim([max(y) * 1.1, min(y) * 1.1])
    plt.plot(Xgrid, BestPolyModel.predict(Xgrid[:, np.newaxis]))
    plt.plot(Xgrid, f(Xgrid))
    plt.fill_between(Xgrid, UpperPercentile, LowerPercentile, color='b', alpha=.1)
    plt.scatter(X.reshape(-1), y, c="red")
    plt.legend(["poly", "truth", "confidentband", "data"])
    plt.savefig("Poly-boopstraped.png")
    plt.show()
    ## Plot the Gaussian Boopstrap
    UpperPercentile, LowerPercentile = BoopStraping(
        lambda x, y: RBFKernel(x, y, GaussianBest[0]),
        GaussianBest[1], Xgrid
    )
    plt.title("Bootsrap␣Confident␣Bands␣for␣Gaussian␣Kernel␣Ridge␣Regression\n"
              f"Gamma:␣{round(GaussianBest[0],␣3)},␣Lambda:␣{round(GaussianBest[1],␣3)},␣95,␣5␣%␣
                  interval")
    # plt.plot(Xgrid, UpperPercentile)
    # plt.plot(Xgrid, LowerPercentile)
    plt.ylim([max(y) * 1.1, min(y) * 1.1])
    plt.plot(Xgrid, BestGaussianModel.predict(Xgrid[:, np.newaxis]))
    plt.plot(Xgrid, f(Xgrid))
    plt.fill_between(Xgrid, UpperPercentile, LowerPercentile, color='b', alpha=.1)
    plt.scatter(X.reshape(-1), y, c="red")
    plt.legend(["guassian", "truth", "confidentband", "data"])
    plt.savefig("gaussian-boopstraped.png")
    plt.show()

    # A3 Part (e). Additional Boopstrap to compare the models.
    if n == 300 and KfoldSplit == 10:
        print("For␣A3␣(e)␣we␣are␣going␣to␣use␣the␣idea␣of␣"
              "boopstrap␣to␣find␣the␣confidence␣interval␣of␣the"
              "␣best␣MSE␣model␣minus␣the␣Poly␣Model␣")

        LowerPercentile, UpperPercentile = \
            BoopStrapModelDifference(BestGaussianModel, BestPolyModel)
        print(f"The␣Upper␣and␣lower␣bound␣of␣the␣confidence␣interval␣is:"
              f"␣{LowerPercentile,␣UpperPercentile}")
        with open("goodplots/A3e.txt", "a") as file:
            file.write(f"Upper,␣Lower␣bound:␣{LowerPercentile,␣UpperPercentile}")



if __name__ == "__main__":
    import os
    print(f"cwd:␣{os.getcwd()}")
    main(n=300, KfoldSplit=10)
    #main()
```