**CUED - Engineering Tripos Part IIB 2024-2025** **Module Coursework**

| Module | 4G5 | Title of report | Prediction of Drug Solubility |
|---|---|---|---|

Date submitted: 19/03/2025

Assessment for this module is ☒ 100% / ☐ 25% coursework

of which this assignment forms 50 %

| UNDERGRADUATE and POST GRADUATE STUDENTS | | | | |
|---|---|---|---|---|
| Name: | Mac Walker | College: | Magdalene | ☐ Undergraduate<br>☒ Post graduate |

Feedback to the student

☐ **See also comments in the text**

| | | Very good | **Good** | Needs improvmt |
|---|---|---|---|---|
| CONTENT | **Completeness, quantity of content:**<br>Has the report covered all aspects of the lab? Has the analysis been carried out thoroughly? | | | |
| | **Correctness, quality of content**<br>Is the data correct? Is the analysis of the data correct? Are the conclusions correct? | | | |
| | **Depth of understanding, quality of discussion**<br>Does the report show a good technical understanding? Have all the relevant conclusions been drawn? | | | |
| | Comments: | | | |
| PRESENTATION | **Attention to detail, typesetting and typographical errors**<br>Is the report free of typographical errors? Are the figures/tables/references presented professionally? | | | |
| | Comments: | | | |

Marker: Date:

# Prediction of Drug Solubility

**BGN:** 2265V

March 19, 2025

**Abstract**

This report explores the use of Kernel Ridge Regression for predicting drug solubility. We discuss the theoretical background, dataset characteristics, model training, evaluation, and results. We perform hyperparameter selection for a toy example and for a chemical solubility dataset. We then introduce noise in our training set and deliberate the effect this has on our model. We then use the understanding gained to model a real-world drug solubility dataset, seeing the effect of hyperparameter optimisation over a higher-dimensional dataset.

# Contents

# 1 Introduction

Suppose you have some property that you want to predict. You also have some data points taken together with the property you want to predict. We want to construct a model which allows us to predict a property given a set of data about the samples - this is known as 'function fitting', statistics, or more recently 'AI' (really, it's machine learning). In this project, we wish to develop the theoretical framework for kernel ridge regression, a machine learning method which allows us to take in some data, $\boldsymbol{X}$, with associated property we would like to predict, $\boldsymbol{y}$, and fit a model such that whenever we encounter a new set of data $\boldsymbol{X}^\star$, we may predict $\boldsymbol{y}^\star$. We will scaffold a theoretical understanding of this method and then use it for a drug solubility dataset. Drug solubility is a critical factor in pharmaceutical development, influencing bioavailability and efficacy. Hence, the ability to predict the solubility of a drug is of utmost importance to biotechnologists, chemists and those involved in drug discovery.

# 2 Building towards Kernel Ridge Regression

Suppose we have a dataset, $\boldsymbol{X}$. We will use a concrete example, where $\boldsymbol{X} \in \mathbb{R}^{100}, \boldsymbol{X} = [0, 0.101 \ldots 9.89, 10]$. We take $\boldsymbol{y} = f(x)$ for $x \in \boldsymbol{X}$, where $f(x) = sin(x)e^{\frac{x}{5}}$..

We want to find a model, that, given $\boldsymbol{X}$ and $\boldsymbol{y}$, can find (and reproduce) the function that was used to produce $\boldsymbol{y}$, explicitly *without* access to $f(x)$.

How can we do this? Arguably, this problem motivates nearly the entirety of the field of statistical machine learning. A full exhaustive list of methods for this problem is beyond the scope of this paper, however we will use kernel ridge regression, and we hope to motivate its usage in this section.

Let us continue with our problem. Figure 1 shows the data points plotted on the Cartesian system.

Again, someone familiar with exponential functions or sinusoidal functions may take one look at this and instantly recognise it. But this model only has access to the two vectors of $\boldsymbol{X}$ and $\boldsymbol{y}$.

## 2.1 Linear Regression

We wish to fit a model using these data points. Let us begin where all good statistical investigations should start: by plotting a linear regression.

Linear regression assumes a linear relationship between the input features and the output:

$$y = w_0 + w_1 x + \epsilon$$

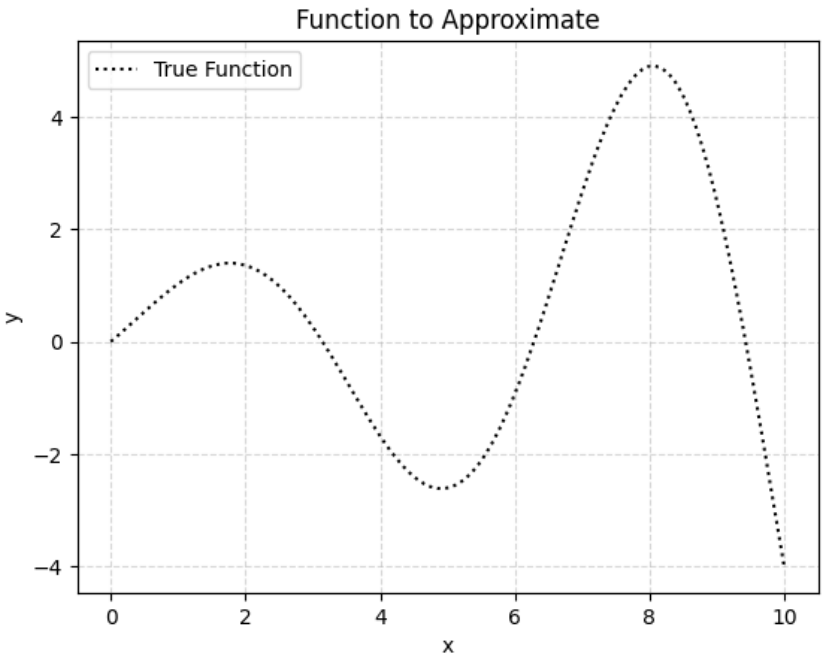


Figure 1: The ground truth of the function

In matrix form, for a dataset of $n$ observations, we express this as:

$$\boldsymbol{y} = \mathbf{X}\boldsymbol{w} + \boldsymbol{\epsilon}$$

where: $\mathbf{X} \in \mathbb{R}^{n \times d}$ is the design matrix (including a bias column if necessary), $\boldsymbol{w} \in \mathbb{R}^d$ is the weight vector and $\boldsymbol{y} \in \mathbb{R}^n$ is the target vector. To find the optimal weight vector $\boldsymbol{w}$, we minimise the sum of squared errors (or, equivalently, minimise the root mean sum of squared errors):

$$\min_{\boldsymbol{w}} ||\mathbf{X}\boldsymbol{w} - \boldsymbol{y}||^2$$

This solution is given by the normal equation:

$$\boldsymbol{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\boldsymbol{y}$$

We plot the model in Figure 2. This model is of the form:

$$\boldsymbol{y}^\star = \boldsymbol{w}^T\mathbf{X}^\star$$

Clearly, this isn't a very good fit! Before we even begin to inspect MSE and RMSE, we can clearly see that our linear model is failing to capture the true underlying function that has produced the data.

So, we may establish that a linear regression model is not suitable for this task. We would like to use kernel ridge regression, but to do that, let us introduce ridge regression.

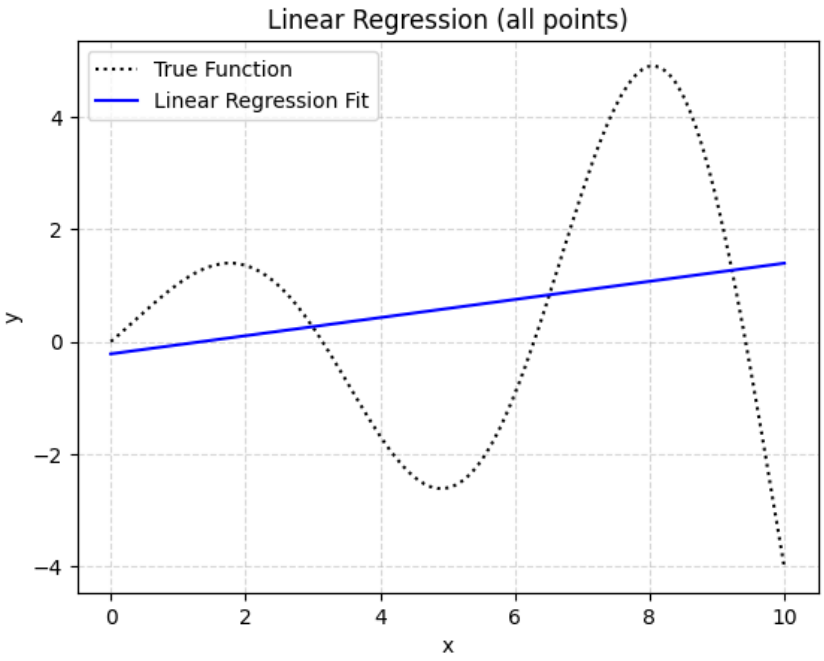


Figure 2: Linear Regression using all points

## 2.2 Ridge Regression

Computing linear regression involves finding the inverse of the covariance matrix $\mathbf{X}^T\mathbf{X}$. If $\mathbf{X}^T\mathbf{X}$ is close to being singular, then the solution may be unstable, leading to extreme values for $\mathbf{w}$. Further, when working in high-dimensional spaces, we may wish to control for overfitting. A natural reformulation of linear regression may therefore be to include some form of penalty for the size of the coefficient vector $\mathbf{w}$. Here, we introduce *ridge regression* [3].

Instead of minimising the objective function that solely includes the mean squared error, we introduce an $L_2$-norm penalty on the weight vector:

$$\min_{\boldsymbol{w}} ||\mathbf{X}\boldsymbol{w} - \boldsymbol{y}||^2 + \lambda||\boldsymbol{w}||^2$$

where $\lambda \geq 0$ is a regularisation parameter that controls the tradeoff between fit quality and coefficient shrinkage. One might be able to notice that if we set $\lambda = 0$. we recover standard linear regression, and if we set $\lambda$ to some large enough value, we would approach $\boldsymbol{y}^\star = \alpha, \alpha \in \mathbb{R}$ (where the coefficients have approached 0). We now solve for the coefficient vector $\boldsymbol{w}$ using:

$$\boldsymbol{w} = (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^T\boldsymbol{y}$$

With the new model still of the form:

$$\boldsymbol{y}^\star = \boldsymbol{w}^T\mathbf{X}^\star$$

However, for our toy example described previously, this still doesn't capture the inherent nonlinearity in our data. We are now in a position to develop the final amendment we will make to our model.

## 2.3 Kernel Ridge Regression

To model the nonlinearity of the relationship between our features and the target vector, we have multiple options. One approach to capturing nonlinearity is to explicitly define a set of nonlinear feature transformations and optimise them. However, this is computationally expensive and impractical, as the number of possible nonlinear transformations is infinite. Instead, we take a more nuanced approach: rather than explicitly defining feature transformations, we rely on kernels, which allow us to implicitly work in a high-dimensional space without ever computing the transformation directly [4]:

$$\boldsymbol{y} = \Phi \boldsymbol{w} + \boldsymbol{\epsilon},$$

where $\Phi$ is a nonlinear transformation:

$$\Phi = \begin{bmatrix} \phi(x_1)^T \\ \phi(x_2)^T \\ \vdots \\ \phi(x_n)^T \end{bmatrix} \in \mathbb{R}^{n \times p}.$$

Applying ridge regression in this transformed space gives:

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

The key insight is that the weight vector $\boldsymbol{w}$ can be rewritten as a linear combination of training points:

$$\boldsymbol{w} = \sum_i c_i \phi(x_i) = \Phi^T \boldsymbol{c}.$$

Substituting this into the ridge regression solution:

$$\Phi^T \boldsymbol{c} = (\Phi^T \Phi + \lambda \mathbf{I})^{-1} \Phi^T \boldsymbol{y}.$$

We now rewrite the problem in terms of the Gram matrix $K$, defined as $K = \Phi \Phi^T$, and solve for $c$:

$$\Phi \Phi^T \boldsymbol{c} = (\Phi \Phi^T + \lambda \mathbf{I})^{-1} \boldsymbol{y}.$$

We define the kernel matrix $\mathbf{K}$:

$$K_{ij} = k(x_i, x_j) = \phi(x_i)^T \phi(x_j),$$

which gives:

$$\boldsymbol{c} = (\mathbf{K} + \lambda \mathbf{I})^{-1} \boldsymbol{y}.$$

Predictions for a new point $x^\star$ are made as:

$$\hat{y}^\star = \sum_{i=1}^{n} c_i k(x_i, x^\star).$$

We define $\mathbf{K_s}$ as the test kernel matrix, where each entry represents the kernel function evaluation between training points and new test points $\mathbf{K_s} = \sum_{i=1}^{n} k(x_i, x^\star)$. This allows us to express predictions compactly as

$$\hat{\boldsymbol{y}}^\star = \mathbf{K_s} \boldsymbol{c}.$$

Thus, kernel ridge regression extends ridge regression to nonlinear functions while avoiding explicit feature transformation.

## 2.4 Use of Kernel Ridge Regression for our Toy Example

In our toy model problem, we require a kernel that can capture both the smooth and nonlinear nature of the function we aim to approximate. We choose the Gaussian (Radial Basis Function, RBF) kernel. It is defined as:

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

The Gaussian kernel has strong theoretical properties, as it is a universal approximator, meaning it can approximate any continuous function given sufficient data. Although many other kernels exist, and the choice of kernel is basically a 'hyper-hyperparameter', we leave a full exploration of different kernels outside of the scope of this report.

In Figure 3, we see the training points we use to fit our kernel ridge regression model.

Here, we have that the number of training points is $n = 10$. The number of training points is generated randomly in the range $[0, 10]$. We have $y \in \mathbb{R}^n$ : Training outputs, where $y_i = f(x_i)$. Our test data comprises of the $m = 100$ test points, with $X^\star \in \mathbb{R}^m$ test inputs, evenly spaced between $[0, 10]$ and $y^\star = f(X^\star)$.

Further, we have choice over 2 hyperparameters: $\sigma$ and $\lambda$. As previously discussed, $\lambda$ is our regulariation parameter that controls the tradeoff between fit quality and coefficient shrinkage. The length scale parameter  determines how rapidly similarity between data points decays in the feature space. The hyperparamter $\sigma$ is the length scale, which controls

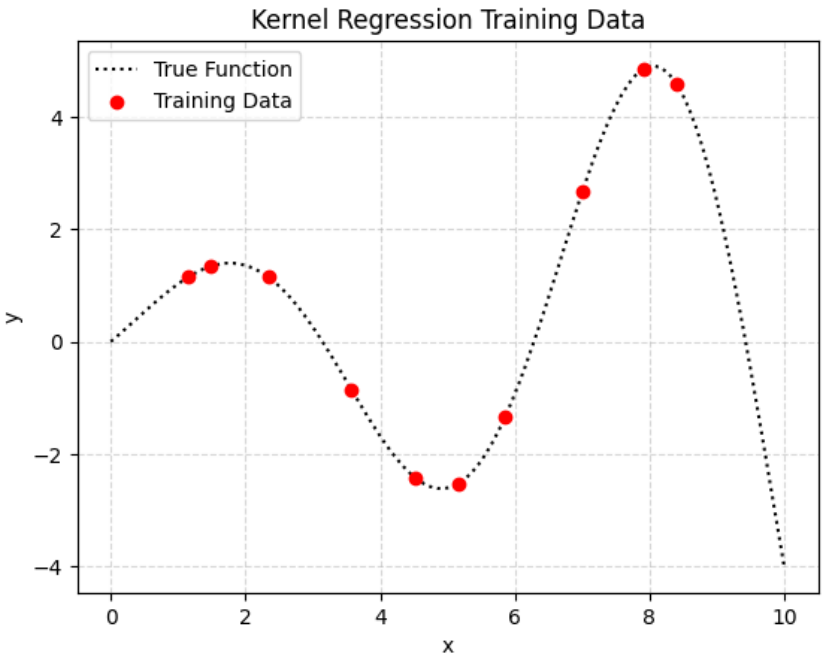


Figure 3: Training Points

how smooth the function is. A small $\sigma$ captures local patterns but risks overfitting, while a large $\sigma$ smooths the function but may underfit. Further exploration of the role of $\sigma$ is given in the next section. In our model fitting, given in Figure 4 and Figure 5, the hyperparameters are set to $\sigma = 1$ and $\lambda = 0.1$ for both.

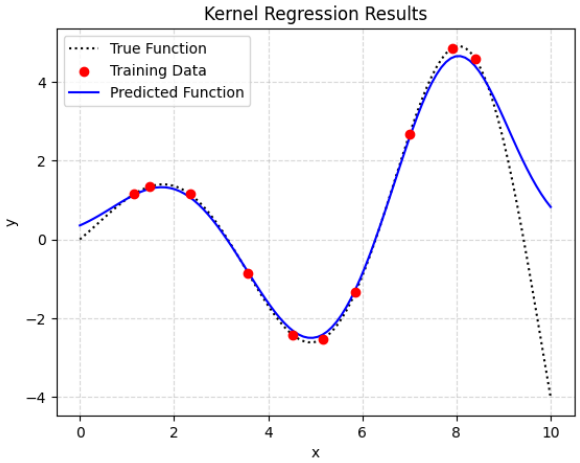


Figure 4: Kernel Ridge Regression - 1

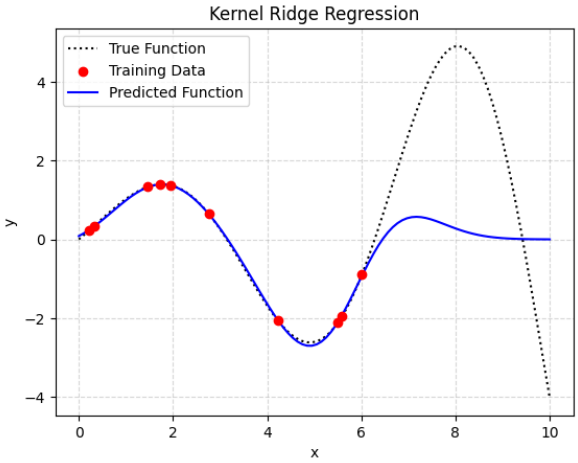


Figure 5: Kernel Ridge Regression - 2

## 2.5 Analysis of Kernel Ridge Regression Results

Examining Figure 4, we see that kernel ridge regression has successfully captured the underlying non-linear function, demonstrating its ability to model complex, nonlinear

relationships when provided with a well-representative training dataset.

However, Figure 5 clearly has worked less well! While the model captures the approximate shape of the function, its predictive capability is severely hindered by the fact that the training data is non-representative of the true distribution. This exemplifies the well-known principle: *garbage in, garbage out* - the quality of predictions is fundamentally limited by the quality and representativeness of the training data.

Importantly, this phenomenon is easy to see in our controlled, toy setting, where we have access to the ground truth function. However, in real-world applications, the true function is almost never available, making it difficult to assess whether poor predictions stem from model limitations or inadequate data. This underscores the importance of increasing $n$ (the number of training points) and exploring alternative data sampling strategies. Exploring the optimality of sampling strategies is beyond the scope of this report but a crucial consideration in practical machine learning.

## 2.6 Hyperparameter Tuning

We performed a grid search over key parameters:

- **Lengthscale** ($\sigma$): $\{0.1, 0.5, 1, 2, 5, 10\}$.
- **Regularization Strength** ($\lambda$): $\{0.001, 0.01, 0.1, 1, 2\}$.
- **Number of Training Points** ($n$): $\{1, 2, 5, 10, 15, 20\}$.

For each combination of these hyperparameters, we trained a kernel ridge regression model using a shared set of randomly sampled training points and evaluated its performance on the fixed test set, 100 linear spaced points spanning the domain $[0, 10]$ with respective function values as the predictor variable. For each model, we calculate the mean-squared error (MSE) and root mean-squared error (RMSE).

$$\text{MSE} = \frac{1}{m}\sum_{i=1}^{m}(y_i^\star - \hat{y}_i^\star)^2, \quad \text{RMSE} = \sqrt{\text{MSE}}.$$

Full results may be found in the Appendix.

We show results for $n = 10$ in Figure 6 and Figure 7.

Let us discuss these results. When inspecting the heatmap in Figure 6 we see that the hyperparameter selection $\lambda = 0.001$ and $\sigma = 2$ corresponds to the minimal RMSE. When we inspect the plotted function in Figure 7, we see almost an exact fit of the ground truth function with the kernel ridge regression.

What is arguably more interesting when inspecting this plot is that it shows a good description of the causes of ill-fitting from the other choices of hyperparameters.

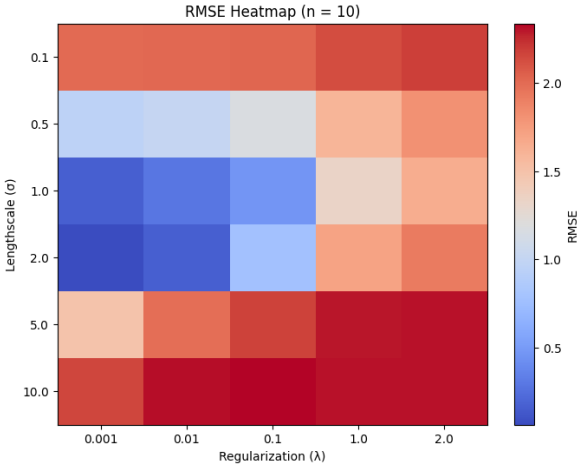


Figure 6: Heatmap for $n = 10$

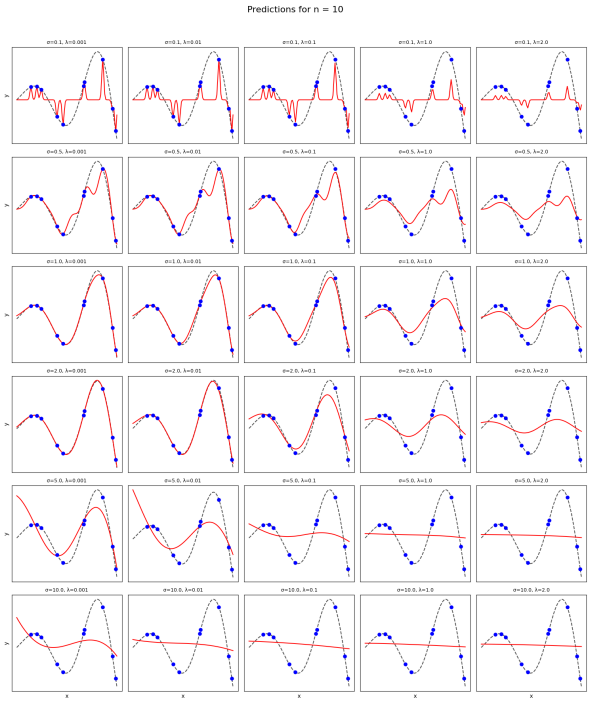


Figure 7: Kernel Ridge Regression Prediction for $n = 10$

As can be seen, for all plots with $\sigma = 10$, there is severe underfitting. This reinforces what was mentioned earlier: a larger $\sigma$ encourages a smoother function, which isn't ideal in this scenario as the nonlinearity in the dataset arises from the ground truth nonlinearity in the function producing these datapoints. Our investigation later discusses the interplay between a larger $\sigma$ and the ability to discern signal from noise. An inverse pattern is observed when $\sigma = 0.1$. The models produced by the kernel ridge regression all have a similar form. This form comprises of a seemingly constant $y$ value, around the average of the $y$ values for this given test set. We also see sharp spikes towards the training points. For constant $n$ and $\sigma$, we see how if we increase $\lambda$, the spikes become less pronounced around the training points. This is a good example of how increasing the regularisation parameter reduces the tendency of the model produced by kernel ridge regression to fit towards the variance in the training data.

Full heatmaps and plots may be found in the Appendix. Here, we are able to compare the effect of different $n$ values. As could be expected, for smaller $n$ values, ($n \leq 5$, hyperparameter selection can only go so far. The inherent complexity of the ground truth function requires a large enough set of representative training points to fit to the function.

For $n$ values greater than 10, we see that the kernel ridge regression model is able to fit exceedingly well with extremely small RMSE's. Of course, with bad choices of hyperparameters, we still see bad fitting models and large RMSE's. This highlights that

whilst a larger amount of $n$ is almost always advisable (given that we know that the $n$ training points are representative of the data), we still need careful choice of hyperparameters. Furthermore, there are dimininishing returns of larger $n$ as $n$ gets larger and larger. In this toy example, computational demand was negligible. In future experiments, with greater computational complexity, the trade-off between size of the training set and computational demand should be dealt with incredible care.

### 2.7 Training on Noisy Samples

We now introduce noise into the training data, perturbing the target values. We modify the training targets, $\boldsymbol{y}$, by adding independent Gaussian noise,

$$\boldsymbol{y}^{\text{noisy}} = \boldsymbol{y} + \boldsymbol{\epsilon}, \quad \boldsymbol{\epsilon} \sim \mathcal{N}(0, \sigma^2_{\text{noise}}).$$

This reflects real-world scenarios where observed measurements of $\boldsymbol{y}$ are subject to random fluctuations due to experimental noise, sensor inaccuracies, or inherent stochasticity. We then train the kernel ridge regression model on this noisy dataset, keeping the same test set, $\boldsymbol{X}^\star$, and evaluating its ability to generalise despite increased randomness in the training labels.

The results of this experiment are presented in Figure 8 and Figure 9. Comparing Figure 8 to Figure 6, we see that the introduction of noise broadens the regions of optimal hyperparameter selection, particularly for $\lambda$, as stronger regularisation is required to prevent overfitting to the noise. The predictions in Figure 19 further illustrate this effect. Previously the model closely followed the ground truth function, we now see small deviations, particularly near training points. Low $\lambda$ values still lead to overfitting, capturing noise rather than signal, while excessively large $\sigma$ smooths over meaningful structure in the data. This highlights the crucial interplay between regularisation strength and kernel smoothness in determining model robustness under noisy conditions. Further plots are given in the Appendix. We suggest using more training points when working with noisy data - it can be expected that the noise should 'cancel out' for large enough size of training set.

## 3 Drug Solubility Prediction: Kernel Ridge Regression in the Wild

We now shift from a toy example to a real-world problem: drug solubility prediction. Given a dataset of molecular descriptors and corresponding solubility values, our goal is to construct a model that, given molecular features $\boldsymbol{X}$, can accurately predict solubility values $\boldsymbol{y}$. This is an extension of the function-fitting framework previously introduced,

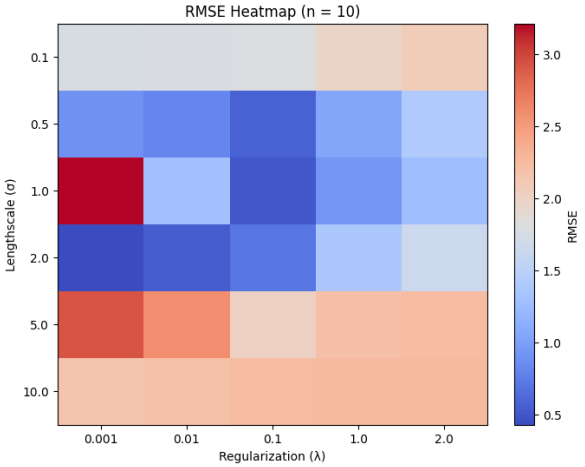


Figure 8: Heatmap for $n = 10$ with noisy data

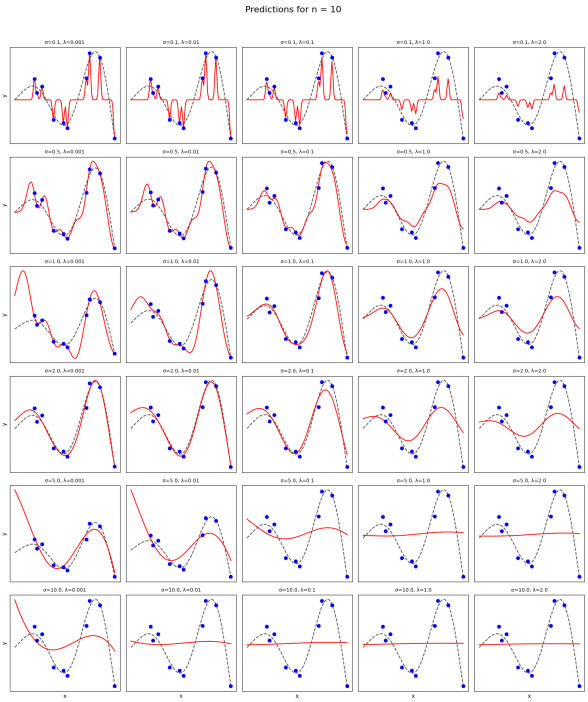


Figure 9: Kernel Ridge Regression Prediction for $n = 10$ with noisy data

but now applied to a dataset with significantly more complexity, higher dimensionality, and real-world relevance.

The dataset consists of 9982 molecules, each described by a set of physicochemical properties, including molecular weight, number of hydrogen bond donors/acceptors, ring counts, and other descriptors (a full description of molecular features is given in the Appendix). This design matrix, denoted as $\boldsymbol{X} \in \mathbb{R}^{9982\times 11}$, serves as input to our model. The solubility values, $\boldsymbol{y} \in \mathbb{R}^{9982}$, represent the target variable we seek to predict. To ensure comparability across molecules, extensive properties such as atomic counts are normalised by molecular weight, and the log of molecular weight is included as an additional feature.

We extend the kernel ridge regression methodology to the solubility dataset. Unlike the previous univariate case, our input data is now multidimensional, requiring a Gaussian kernel that operates on vector inputs. The kernel function takes the form:

$$K(\boldsymbol{x}, \boldsymbol{x}') = \exp\left(-\sum_i \frac{|x_i - x_i'|^2}{2\sigma_i^2}\right),$$

where $x_i$ refers to the $i$th feature of a given molecular descriptor vector $\boldsymbol{x}$, and $\sigma_i$ represents the characteristic lengthscale for each feature dimension. Hence, $\boldsymbol{\sigma} \in \mathbb{R}^{11}$.

Aside from the multidimensional kernel, we perform kernel ridge regression following the exact same framework as previously introduced.

## 3.1 $\lambda$ Optimisation

We begin our investigation by using a fixed set of hyperparameters for $n$ and $\boldsymbol{\sigma}$. We set

$$n = 7986 = \lfloor 0.8 \times 9982 \rfloor \quad \text{and} \quad \sigma_i = \sqrt{\frac{1}{N}\sum_{k=1}^{N}\left(X_{k,i} - \overline{X}_i\right)^2} \quad \text{for each feature } i,$$

where

$$\overline{X}_i = \frac{1}{N}\sum_{k=1}^{N} X_{k,i}$$

is the mean of the $i$-th feature, and $N$ is the total number of data points in the dataset $X$.

We loop over different $\lambda$ values, where $\lambda = [10^{-6}, 10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1, 10]$. In Figure 10, we plot the train RMSE and test RMSE for each of the $\lambda$ values.

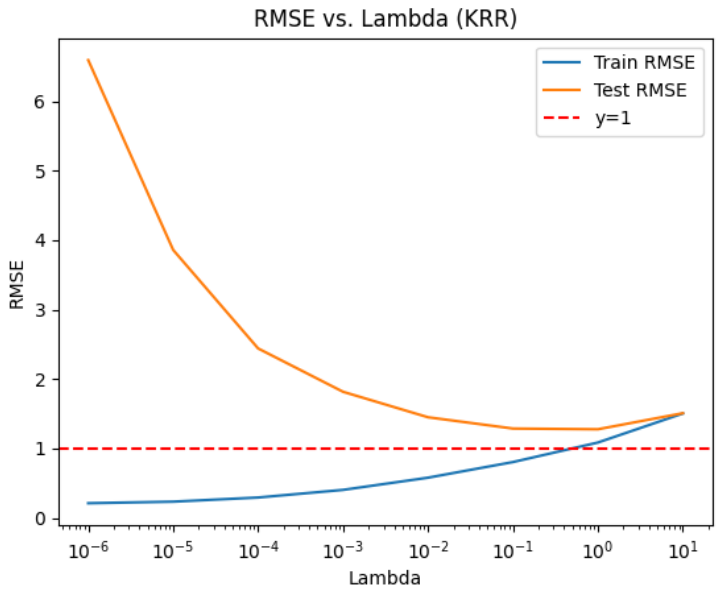


Figure 10: Lambda Hyperparameter Grid Search

In Figure 10, we see the canonical pattern of train and test errors: as we increase lambda (hence, encouraging underfitting), our test error decreases whilst our train error increases. We seemingly reach a minimum value at $\lambda = 0.1$, where we reach RMSE = 1.3, and after this the test error increases again. Unfortunately, we are not able to decrease our test RMSE below 1, the arbitrary value given as a useful model. This inspires the next section, where we perform optimisation over the full set of hyperparameters ($\lambda$ and $\boldsymbol{\sigma}$).

### 3.2 Full Hyperparameter Optimisation

We attempted to reduce the test RMSE below 1. We ran multiple experiments, performing hyperparameter optimisation over $\lambda$ and $\boldsymbol{\sigma}$. This, unfortunately, did not give the required accuracy gains: we still achieved an RMSE of $\approx 1.3$. We then trained two alternative models: a simple multi-layer perceptron (MLP) model, and a Gaussian Process (GP) model. Full code implementation can be found in the Appendix.

The minimum RMSE found across these alternative models was still $\approx 1.2$. What do these results suggest about the limitations of these methods?

Other papers, using methods similar to ours [1], have found RMSE at around 1.5. We posit that, given the features that were used in our model construction, we are approaching a minimum test RMSE. What does this mean? That is, we are approaching the true ground truth function that predicts solubility from these 11 predictors, and the predictors are noisy enough such that they provide $\simeq 1.2$ RMSE.

We expect that to achieve a lower RMSE, we must utilise this dataset more efficiently. We also have access to the SMILES strings, which allows us to explore molecular structure-based representations. The models we are currently using use these features:

```
'HeavyAtomCount', 'NumHAcceptors', 'NumHDonors', 'NumHeteroatoms',
'NumRotatableBonds', 'NumValenceElectrons', 'NumAromaticRings',
'NumSaturatedRings', 'NumAliphaticRings', 'RingCount'
```

These are incredibly valuable data points, but there is no measure of how these are connected. That is to say, for lower RMSE (and that is, approximating the ground truth function of how molecular structure impacts solubility), we need to better approximate the structure through our feature representation. We recommend using a graphical structure, which captures the atomic make-up of these molecules whilst also providing a natural structure for the representation of interatomic forces. For example, research has been conducted into using a Graph Convolutional Neural Network approach, treating each molecule with a graph representation [2]. This approach achieved an RMSE of 0.76.

## 4 Conclusion

In this study, we applied kernel ridge regression to the problem of drug solubility prediction, building upon a theoretical foundation that included linear and ridge regression. Through a toy example, we demonstrated the effectiveness of KRR in capturing nonlinear relationships, followed by a real-world application on a high-dimensional solubility dataset. Despite extensive hyperparameter optimization, we found that the predictive performance was limited by the quality of available molecular descriptors, with an RMSE plateauing at approximately 1.2–1.3. This suggests that further improvements require richer molecular representations, such as graph-based methods. Future work could

explore deep learning approaches, including Graph Neural Networks, or alternatively, graph-based probabilistic methods.

# References


[1] Paul G. Francoeur and David R. Koes. Soltrannet–a machine learning tool for fast aqueous solubility prediction. *Journal of Chemical Information and Modeling*, 61(6):2543–2547, May 2021.

[2] John Ho, Zhao-Heng Yin, Colin Zhang, Nicole Guo, and Yang Ha. Predicting drug solubility using different machine learning methods - linear regression model with extracted chemical features vs graph convolutional neural network. *arXiv*, 2308.12325v2, January 2024.

[3] Sifan Liu and Edgar Dobriban. Ridge regression: Structure, cross-validation, and sketching. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2020. Published as a conference paper at ICLR 2020.

[4] Max Welling. Kernel ridge regression. *Technical Report*. This is a note to explain kernel ridge regression.

# 5 Appendix

## 5.1 Generative AI Statement

I used Generative AI throughout the coding and writing up of this project. The logic used to derive all of the mathematical derivations and the code implementation of linear regression, ridge regression and kernel ridge regression was developed through familiarity with the subject and the lecture notes.

The scripts for grid search over the hyperparameters was created in ChatGPT (by copying in the code I have created for the kernel ridge regression function and the hyperparameters I wanted to loop over). Almost all of the matplotlib code was done by Claude. For the optimisation script, we ran multiple experiments.

ChatGPT and Claude were both used when formatting the LaTeX file, especially for plotting images and formatting in correct way.

## 5.2 Hyperparameter Tuning Results - Heatmaps

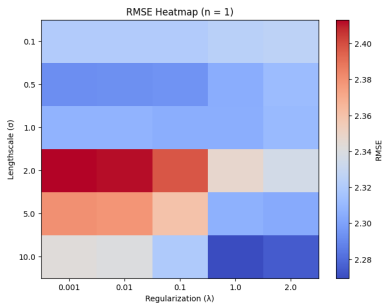


Figure 11: Heatmap for $n = 1$

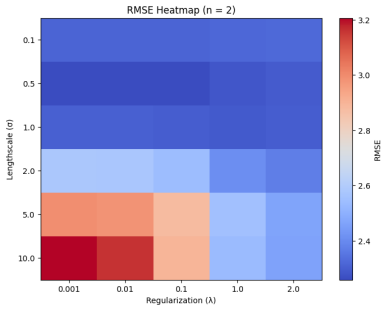


Figure 12: Heatmap for $n = 2$

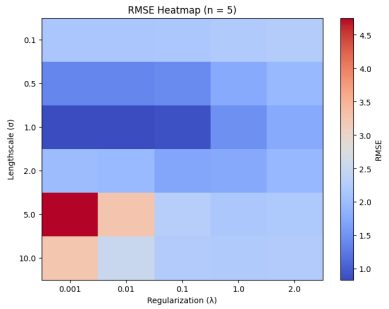


Figure 13: Heatmap for $n = 5$

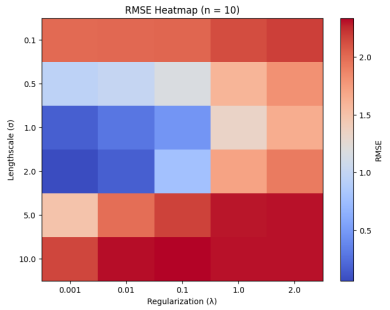


Figure 14: Heatmap for $n = 10$

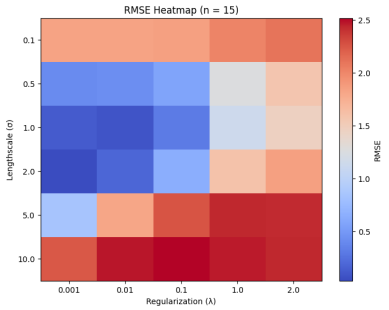


Figure 15: Heatmap for $n = 15$

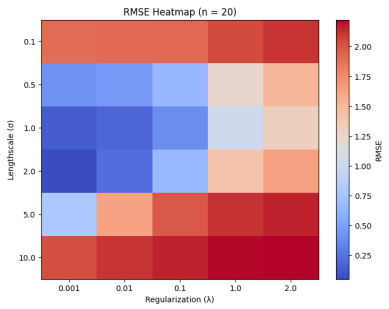


Figure 16: Heatmap for $n = 20$

Figure 17: Comparison of heatmaps across different training set sizes $n$.

## 5.3 Hyperparameter Tuning Results - Plots

## 5.4 Noisy Data Hyperparameter Tuning - Heatmaps

## 5.5 Noisy Data - Hyperparameter Tuning - Plots

## 5.6 Full Notebook File

```
# %% [markdown]
# Function fitting (Kernel Ridge Regression) : molecular solubility
# ====
#
# This computer project introduces Kernel Ridge Regression (also known
↪  as Gaussian process regression) for fitting
# functions in many dimensions. The concrete application is to make a
↪  model of the solubility of small organic
# molecules as a function of their molecular structure.
#
# The number of different organic molecules is enormous, even just
↪  considering the small ones. There is an [online
↪  database](https://gdb.unibe.ch/downloads)
# of all possible organic molecules made from various number of heavy
↪  (non-hydrogen) atoms.
# The number of possibilities is enormous, here is a fun fact: you can
↪  make nearly [1 billion different shapes from
# 6 LEGO
↪  bricks](https://brickset.com/article/30827/review-624210-lego-house-6-bricks).
↪  One of them is shown below
# (printed on a visitor card at the end of a tour, each one unique).
↪  Organic molecules are bit similar: you can branch the carbon
↪  framework and hang
# off three other elements (H,O,N) or more (if you include halides like
↪  Cl and F, or second row elements like P and S).
#
# <img src="lego_6_bricks_card.jpg" width=300>

# %% [markdown]
# Function fitting
# ---
#
```

```python
# We start by fitting a simple one variable function using a variety of
↪  ways.

# %%
import numpy as np
from matplotlib.pyplot import *
import matplotlib.pyplot as plt
# target function
def f(x):
    return np.sin(x)*np.exp(x/5)

x_values = np.linspace(0,10, 100)
y_values = f(x_values)

plot(x_values, y_values, ':', label="True Function", color='black')
# Plot training data

# Labels and title
plt.xlabel("x")
plt.ylabel("y")
plt.title("Function to Approximate")

# Legend
plt.legend()

# Grid for better readability
plt.grid(True, linestyle='--', alpha=0.5)

# Show the plot
plt.show()

print(x_values[1])
print(x_values[-2])
print(len(x_values))

# %%
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
```

```python
# Target function
def f(x):
    return np.sin(x) * np.exp(x / 5)

# Generate data
x_values = np.linspace(0, 10, 100)
y_values = f(x_values)

# Fit linear regression model
x_values_reshaped = x_values.reshape(-1, 1)  # Reshape for sklearn
model = LinearRegression()
model.fit(x_values_reshaped, y_values)
y_pred = model.predict(x_values_reshaped)

# Plot the true function
plt.plot(x_values, y_values, ':', label="True Function", color='black')

# Plot the linear regression line
plt.plot(x_values, y_pred, '-', label="Linear Regression Fit",
↪  color='blue')

# Labels and title
plt.xlabel("x")
plt.ylabel("y")
plt.title("Linear Regression (all points)")

# Legend
plt.legend()

# Grid for better readability
plt.grid(True, linestyle='--', alpha=0.5)

# Show the plot
plt.show()


# %%
# gather some data points which we will fit to
```

```
import random
import matplotlib.pyplot as plt

n = 10 # number of points

random.seed(42069)
x_train = np.random.random(n)*n
y_train = f(x_train)

# Plot ground truth function

plot(x_values, y_values, ':', label="True Function", color='black')
# Plot training data
plt.scatter(x_train, y_train, color='red', label="Training Data",
↪  zorder=3)

# Labels and title
plt.xlabel("x")
plt.ylabel("y")
plt.title("Kernel Regression Training Data")

# Legend
plt.legend()

# Grid for better readability
plt.grid(True, linestyle='--', alpha=0.5)

# Show the plot
plt.show()

# %%
x_train_reshaped = x_train.reshape(-1, 1)
model_train = LinearRegression()
model_train.fit(x_train_reshaped, y_train)
y_pred_train = model_train.predict(x_values.reshape(-1, 1))

# Plot the true function
plt.plot(x_values, y_values, ':', label="True Function", color='black')
```

```
# Plot training data
plt.scatter(x_train, y_train, color='red', label="Training Data",
↪  zorder=3)

# Plot linear regression fit using training points
plt.plot(x_values, y_pred_train, '-', label="Linear Regression Fit",
↪  color='blue')

# Labels and title
plt.xlabel("x")
plt.ylabel("y")
plt.title("Linear Regression (training points)")

# Legend
plt.legend()

# Grid for better readability
plt.grid(True, linestyle='--', alpha=0.5)

# Show the plot
plt.show()



# %%
# define a Gaussian kernel
sig = 1.0 # length scale
def kernel(x1, x2, sig):
    return np.exp(-(x1-x2)**2/(2*sig**2))

K = np.zeros((n,n))
# Now apply the formulas from the lecture to fill in the kernel matrix
↪  and
# compute fitting coefficients. Use the function numpy.linalg.lstsq()
↪  to solve the linear problem K@c=y
# (look up its documentation, and don't forget that it returns 4
↪  things, but you only need the solution vector)

lam = 0.1 # strength of regulariser
```

```
# now use your coefficients to predict the function on the xx array
# y = sum over n points of the c power of the kernel

# construct the K matrix
K = np.zeros((n,n))
for i in range(n):
    for j in range(n):
        K[i,j] = kernel(x_train[i], x_train[j], sig)


# Construct the K_s matrix
K_s = np.zeros((len(x_values), len(x_train)))
for i in range(len(x_values)):
    for j in range(len(x_train)):
        K_s[i,j] = kernel(x_values[i], x_train[j], sig)


# Find the (regularised) c vector
c_reg, _, _, _ = np.linalg.lstsq(K + lam * np.eye(n), y_train,
↪  rcond=None)  # Regularized K


y_pred = K_s @ c_reg




# %%
import matplotlib.pyplot as plt

# Plot ground truth function
plt.plot(x_values, y_values, ':', label="True Function", color='black')

# Plot training data
plt.scatter(x_train, y_train, color='red', label="Training Data",
↪  zorder=3)

# Plot predicted function
plt.plot(x_values, y_pred, label="Predicted Function", color='blue')
```

```
# Labels and title
plt.xlabel("x")
plt.ylabel("y")
plt.title("Kernel Regression Results")

# Legend
plt.legend()

# Grid for better readability
plt.grid(True, linestyle='--', alpha=0.5)

# Show the plot
plt.show()


# %% [markdown]
# Task 1
# --
#
# Using now explore the predictions by varying
# - the lengthscale sig between 0.1 and 10,
# - the regularisation strength lam between 0.001 and 2,
# - the number of data points between 1 and 20
#
# Give a summary of your findings, including the interplay between the
↪  above parameters!

# %% [markdown]
# Hence, we have clarified that our function returns the _exact_ same
↪  function as the script for task 1. We may proceed.

# %% [markdown]
# To answer the question above, we should
# - run a hyperparameter sweep over the above hyperparameters
# - plot lengthscale v regularisation strength
# - plot the interplay for different data points

# %% [markdown]
```

```
# So we have a set number of
# - x_values: np.linspace(0,10, 100)
# - y_values: f(xx)
# - x_train: inducing points
# So we can think of xx and yy as our 'ground truth' function.
#
# We have to go through our approximation and test three different
↪  parameters:
# - number of points we are using for interpolation
# - lengthscale (£\sigma£)
# - regularisation (£\lambda£)
#
#
# Lets set up a function using our code above to easily run *kernel
↪  ridge regression*.

# %%
# We will make our function as general as possible to try and scaffold
↪  as much understanding as possible
# when, later, we are worried about computational efficiency, we will
↪  perform less in this loop, factoring out as much as possible

def kernel_ridge_regression(x_values, y_values, n_inducing_points, sig,
↪  lam, func, x_train, use_x_train):

    x_values = x_values # this is our full dataset

    n_points = len(x_values) # number of samples in our dataset

    y_values = func(x_values) # this is our dependent variable - what we
    ↪  are trying to predict

    if use_x_train == False:

        x_train = np.random.choice(x_values, n_inducing_points,
        ↪  replace=False) # getting our x_train

    else:
```

```
        n_inducing_points = len(x_train)

    y_train = func(x_train) # setting our y_train values


    # Training Kernel
    K = np.zeros((n_inducing_points,n_inducing_points))

    for i in range(n_inducing_points):
        for j in range(n_inducing_points):
            K[i,j] = kernel(x_train[i], x_train[j], sig) # kernel func
             ↪  between training points

    # Cross-kernel
    K_s= np.zeros((n_points, n_inducing_points))

    for i in range(n_points):
        for j in range(n_inducing_points):
            K_s[i,j] = kernel(x_values[i], x_train[j], sig) # kernel
             ↪  between data and training points

    c, _, _, _ = np.linalg.lstsq(K + lam * np.eye(n_inducing_points),
     ↪  y_train, rcond=None)  # Regularized K

    y_pred = K_s @ c

    return y_pred, x_train

# %%
# Testing our function to see if it works the same as previously

# Plot ground truth function
plt.plot(x_values, y_values, ':', label="True Function", color='black')

x_train = np.random.random(n)*n

y_pred, x_train = kernel_ridge_regression(x_values, y_values, 10, 1,
 ↪  0.01, f, x_train, use_x_train=True)
```

```
# Plot training data
plt.scatter(x_train, f(x_train), color='red', label="Training Data",
↪  zorder=3)


# Plot predicted function
plt.plot(x_values, y_pred, label="Predicted Function", color='blue')

# Labels and title
plt.xlabel("x")
plt.ylabel("y")
plt.title("Kernel Ridge Regression")

# Legend
plt.legend()

# Grid for better readability
plt.grid(True, linestyle='--', alpha=0.5)

# Show the plot
plt.show()

# %%


# %%
import itertools as tools
import pandas as pd

# Define new parameter ranges
sig_values = [0.1, 0.5, 1, 2, 5, 10]  # Lengthscale
lam_values = [0.001, 0.01, 0.1, 1, 2]  # Regularization strength
n_values = [1, 2, 5, 10, 15, 20]  # Number of training points

# Full dataset
x_values = np.linspace(0, 10, 100)
y_values = f(x_values)

# Initialize list to store results before creating DataFrame
```

```
results_list = []

# Loop through hyperparameter combinations
for n in n_values:

    x_train = np.random.random(n) * 10  # Setting same points for each
    ↪ hyperparam loop

    for sig in sig_values:
        for lam in lam_values:

            y_pred, x_train = kernel_ridge_regression(x_values, y_values,
            ↪ n, sig, lam, f, x_train, True)

            mse = np.mean((y_pred - y_values) ** 2)  # MSE
            rmse = np.sqrt(mse)  # RMSE

            # Append results to list
            results_list.append({
                'Lengthscale ()': sig,
                'Regularization ()': lam,
                'Num Points (n)': n,
                'Predictions (y_pred)': y_pred,  # Storing the entire
                ↪ prediction vector
                'MSE': mse,
                'RMSE': rmse,
                'x_train': x_train
            })

# Convert list to DataFrame
results = pd.DataFrame(results_list)


# %%
from IPython.display import display

# Convert list to DataFrame
results = pd.DataFrame(results_list)
```

```python
# Display the results DataFrame
display(results)  # Works in Jupyter Notebook

# If using a script, you can also print a preview
print(results.head())  # Shows first 5 rows

# %%
import matplotlib.pyplot as plt
import numpy as np

# Get unique values of n
n_unique = results["Num Points (n)"].unique()

# Create heatmaps for each n
for n in n_unique:
    subset = results[results["Num Points (n)"] == n]

    # Pivot the data into a 2D array
    sig_values = np.sort(subset["Lengthscale ()"].unique())
    lam_values = np.sort(subset["Regularization ()"].unique())

    mse_matrix = np.zeros((len(sig_values), len(lam_values)))

    for i, sig in enumerate(sig_values):
        for j, lam in enumerate(lam_values):
            mse_matrix[i, j] = subset[(subset["Lengthscale ()"] == sig) &
                                      (subset["Regularization ()"] ==
                                      ↪  lam)]["RMSE"].values[0]

    # Create heatmap using Matplotlib
    fig, ax = plt.subplots(figsize=(8, 6))
    cax = ax.imshow(mse_matrix, cmap="coolwarm", aspect="auto")

    # Labels and ticks
    ax.set_xticks(np.arange(len(lam_values)))
    ax.set_yticks(np.arange(len(sig_values)))
    ax.set_xticklabels(lam_values)
    ax.set_yticklabels(sig_values)
    ax.set_xlabel("Regularization ()")
```

```
    ax.set_ylabel("Lengthscale ()")
    ax.set_title(f"RMSE Heatmap (n = {n})")

    # Colorbar
    fig.colorbar(cax, label="RMSE")

    # Show the heatmap
    plt.show()


# %%
import matplotlib.pyplot as plt

# Unique values of n, sig, and lam
n_unique = results["Num Points (n)"].unique()
sig_values = np.sort(results["Lengthscale ()"].unique())
lam_values = np.sort(results["Regularization ()"].unique())

# Full dataset (ground truth)
x_values = np.linspace(0, 10, 100)
y_values = f(x_values)  # True function

# Loop through each training data size (n)
for n in n_unique:
    fig, axes = plt.subplots(nrows=len(sig_values),
    ↪  ncols=len(lam_values), figsize=(15, 18))
    fig.suptitle(f"Predictions for n = {n}", fontsize=16)

    for i, sig in enumerate(sig_values):
        for j, lam in enumerate(lam_values):
            # Select the corresponding result
            subset = results[
                (results["Num Points (n)"] == n) &
                (results["Lengthscale ()"] == sig) &
                (results["Regularization ()"] == lam)
            ]

            if not subset.empty:
```

```
                y_pred = subset["Predictions (y_pred)"].values[0]  #
                ↪ Extract predictions
                x_train = subset["x_train"].values[0]  # Extract
                ↪ training x-values
                y_train = f(x_train)  # Compute training y-values using
                ↪ the true function

                ax = axes[i, j]
                ax.plot(x_values, y_values, label="True Function",
                ↪ color="black", linestyle="dashed", alpha=0.7)
                ax.plot(x_values, y_pred, label="Predicted", color="red")
                ax.scatter(x_train, y_train, color="blue", marker="o",
                ↪ label="Training Points", zorder=3)

                # Titles and labels
                ax.set_title(f"={sig}, ={lam}", fontsize=9)
                ax.set_xticks([])
                ax.set_yticks([])

                # Reduce clutter by only showing some axis labels
                if i == len(sig_values) - 1:
                    ax.set_xlabel("x")
                if j == 0:
                    ax.set_ylabel("y")

    # Adjust layout and show
    plt.tight_layout(rect=[0, 0, 1, 0.96])
    plt.show()


# %% [markdown]
# Optional Extension - Noisy Data
# ----
#
# Investigate what happens when you add i.i.d random noise of a certain
↪ size to the _data values_ that you
# fit to (But still of course plot the original function without noise
↪ to compare with, as well as the data points
# with noise added).
```

```python
# %%
def kernel_ridge_regression_with_noise(
    x_values, n_inducing_points, sig, lam, func, x_train, y_train_noisy
):
    """
    We remove 'noise_std' and 'use_x_train' as arguments,
    and assume y_train_noisy is already computed outside.
    """
    # x_values, y_values, etc. might not even be needed if you only
    # use them for shape or for K_s. Adjust to your needs.

    n_points = len(x_values)
    # This part is the same
    K = np.zeros((n_inducing_points, n_inducing_points))
    for i in range(n_inducing_points):
        for j in range(n_inducing_points):
            K[i, j] = kernel(x_train[i], x_train[j], sig)

    # Cross-kernel
    K_s = np.zeros((n_points, n_inducing_points))
    for i in range(n_points):
        for j in range(n_inducing_points):
            K_s[i, j] = kernel(x_values[i], x_train[j], sig)

    # Solve for c using the noisy y_train
    c, _, _, _ = np.linalg.lstsq(
        K + lam * np.eye(n_inducing_points), y_train_noisy, rcond=None
    )
    y_pred = K_s @ c

    return y_pred


# %%
sig_values = [0.1, 0.5, 1, 2, 5, 10]    # Lengthscale
lam_values = [0.001, 0.01, 0.1, 1, 2]   # Regularization strength
n_values = [1, 2, 5, 10, 15, 20]        # Number of training points
```

```
np.random.seed(42049)  # Fix the random seed for reproducibility

# Full "ground truth" dataset
x_values = np.linspace(0, 10, 100)
y_values = f(x_values)   # f(...) is your true function

# Initialize a list to store results
results_list = []

# Loop over different training set sizes
for n in n_values:
    # Generate one random set of training points for this n
    x_train = np.random.random(n) * 10
    y_train = f(x_train)

    # Generate *one* noise vector for this n
    noise = np.random.normal(0, 0.5, size=y_train.shape)
    y_train_noisy = y_train + noise

    # Now loop over all sigma-lambda combinations
    for sig in sig_values:
        for lam in lam_values:

            # Use the function with the new signature
            y_pred = kernel_ridge_regression_with_noise(
                x_values=x_values,
                n_inducing_points=n,
                sig=sig,
                lam=lam,
                func=f,
                x_train=x_train,
                y_train_noisy=y_train_noisy
            )

            # Compute MSE/RMSE relative to the true function
            mse = np.mean((y_pred - y_values) ** 2)
            rmse = np.sqrt(mse)

            # Append results to list
```

```
            results_list.append({
                'Lengthscale ()': sig,
                'Regularization ()': lam,
                'Num Points (n)': n,
                'Predictions (y_pred)': y_pred,
                'MSE': mse,
                'RMSE': rmse,
                'x_train': x_train,            # Just so you have it
                'y_train_noisy': y_train_noisy # Just so you have it
            })

# Convert list to DataFrame
results = pd.DataFrame(results_list)

# %%
import matplotlib.pyplot as plt
import numpy as np

# Get unique values of n
n_unique = results["Num Points (n)"].unique()

# Create heatmaps for each n
for n in n_unique:
    subset = results[results["Num Points (n)"] == n]

    # Pivot the data into a 2D array
    sig_values = np.sort(subset["Lengthscale ()"].unique())
    lam_values = np.sort(subset["Regularization ()"].unique())

    mse_matrix = np.zeros((len(sig_values), len(lam_values)))

    for i, sig in enumerate(sig_values):
        for j, lam in enumerate(lam_values):
            mse_matrix[i, j] = subset[(subset["Lengthscale ()"] == sig) &
                                      (subset["Regularization ()"] ==
                                      ↪  lam)]["RMSE"].values[0]

    # Create heatmap using Matplotlib
    fig, ax = plt.subplots(figsize=(8, 6))
```

```
    cax = ax.imshow(mse_matrix, cmap="coolwarm", aspect="auto")

    # Labels and ticks
    ax.set_xticks(np.arange(len(lam_values)))
    ax.set_yticks(np.arange(len(sig_values)))
    ax.set_xticklabels(lam_values)
    ax.set_yticklabels(sig_values)
    ax.set_xlabel("Regularization ()")
    ax.set_ylabel("Lengthscale ()")
    ax.set_title(f"RMSE Heatmap (n = {n})")

    # Colorbar
    fig.colorbar(cax, label="RMSE")

    # Show the heatmap
    plt.show()


# %%
import matplotlib.pyplot as plt

# Unique values of n, sig, and lam
n_unique = results["Num Points (n)"].unique()
sig_values = np.sort(results["Lengthscale ()"].unique())
lam_values = np.sort(results["Regularization ()"].unique())

# Full dataset (ground truth)
x_values = np.linspace(0, 10, 100)
y_values = f(x_values)  # True function

# Loop through each training data size (n)
for n in n_unique:
    fig, axes = plt.subplots(nrows=len(sig_values),
    ↪  ncols=len(lam_values), figsize=(15, 18))
    fig.suptitle(f"Predictions for n = {n}", fontsize=16)

    for i, sig in enumerate(sig_values):
        for j, lam in enumerate(lam_values):
            # Select the corresponding result
```

```
            subset = results[
                (results["Num Points (n)"] == n) &
                (results["Lengthscale ()"] == sig) &
                (results["Regularization ()"] == lam)
            ]

            if not subset.empty:
                y_pred = subset["Predictions (y_pred)"].values[0]  #
                ↪ Extract predictions
                x_train = subset["x_train"].values[0]  # Extract
                ↪ training x-values
                y_train_noisy = subset["y_train_noisy"].values[0]  #
                ↪ noisy y values

                ax = axes[i, j]
                ax.plot(x_values, y_values, label="True Function",
                ↪ color="black", linestyle="dashed", alpha=0.7)
                ax.plot(x_values, y_pred, label="Predicted", color="red")
                ax.scatter(x_train, y_train_noisy, color="blue",
                ↪ marker="o", label="Training Points", zorder=3)

                # Titles and labels
                ax.set_title(f"={sig}, ={lam}", fontsize=9)
                ax.set_xticks([])
                ax.set_yticks([])

                # Reduce clutter by only showing some axis labels
                if i == len(sig_values) - 1:
                    ax.set_xlabel("x")
                if j == 0:
                    ax.set_ylabel("y")

    # Adjust layout and show
    plt.tight_layout(rect=[0, 0, 1, 0.96])
    plt.show()


# %% [markdown]
# Molecular solubility dataset
```

```
# ----
#
# The dataset below contains the solubility of nearly 10K molecules.
↪  The original data came from here:
#
#
↪  https://dataverse.harvard.edu/dataset.xhtml?persistentId=doi:10.7910/DVN/OVHAW8
# https://www.nature.com/articles/s41597-019-0151-1

# %%
import pandas

# %%
sol = pandas.read_csv("curated-solubility-dataset.csv")

# %%
sol

# %%
sol.columns

# %%
sol['Solubility'][0] # here is how to access a given property, here the
↪  solubility of the first molecule

# %%
hist(sol['Solubility'], bins=50)

# %%
# we can plot the solubility against any given property. Clearly there
↪  are relationships, but the solubility is not determined well
# by any single property. Note how it's better to work with the log of
↪  the molecular weight (MolWt)
scatter(np.log(sol['MolWt']), sol['Solubility'], s=2)

# %%
import matplotlib.pyplot as plt
```

```
plt.scatter((sol['NumHAcceptors']+sol['NumHDonors'])/sol['MolWt'],
↪  sol['Solubility'], s=2)

# %% [markdown]
# Linear fit
# ---

# %%
Y = np.array(sol['Solubility']);

proplist = ['HeavyAtomCount',
       'NumHAcceptors', 'NumHDonors', 'NumHeteroatoms',
       ↪  'NumRotatableBonds',
       'NumValenceElectrons', 'NumAromaticRings', 'NumSaturatedRings',
       'NumAliphaticRings', 'RingCount']

X = np.array([list(sol[prop]/sol['MolWt']) for prop in proplist]) # Many
↪  properties are extensive, so we divide by the molecular weight
X = np.insert(X, 0, list(np.log(sol['MolWt'])), axis=0) # add the log
↪  MolWt as well

# %%
X.shape # we have 11 properties

# %%
c,_,_,_ = np.linalg.lstsq(X.T, Y) # here we fit a LINEAR MODEL , solving
↪  the equation  X.T @ c = Y

# %%
c # these are the resulting fitting coefficients

# %%
# now we make a scatter plot of target solubility versus predicted
↪  solubility
plt.scatter(Y, X.T @ c)
plt.plot([-50,50], [-50,50], 'k--')
plt.xlim(-12,6)
plt.ylim(-12,6)
plt.ylabel('predicted')
```

```
plt.xlabel('target')
plt.gca().set_aspect('equal')

# %%
# We can calculate the mean squared error of our fit. The solubility values are such that a method with prediction error > 1 is not that useful.
np.sqrt(sum((Y- X.T @ c)**2)/len(Y))

# %% [markdown]
# Task 2
# ---
#
# Now use the Kernel Ridge Regression method to make a kernel fit of the solubility. Your kernel should still be the Gaussian function, but
# it operates on vector arguments. You can start with a single sigma, identical for each property, but then choose different lengthscales (sig) in each dimension (a sensible starting value is the standard deviation of the data in each dimension)
#
# $$
# K(x, x') = e^{-\sum_i|x_i-x'_i|^2/(2\sigma_i^2)}
# $$
#
# Here $x_i$ refers not to the $i$th data point, but to the $i$th element of the multidimensional data point vector $x$. Optimise the regularisation
# strength to obtain the best fit. You should split your data into a training and test set, and report the prediction error separately.

# %%
import numpy as np

def vector_kernel(x1, x2, sig):
    """
    Multidimensional Gaussian Kernel.
    x1, x2: vectors of the same dimension
    sig: vector of lengthscales (one per dimension)
```

```
    """
    diff = x1 - x2  # Element-wise difference
    return np.exp(-np.sum((diff ** 2) / (2 * sig ** 2)))  # Apply
    ↪ per-dimension lengthscale

def train_krr(x_train, y_train, sig, lam):
    """Computes c = (K + lam I)^-1 y for kernel ridge regression."""
    n_train = x_train.shape[0]
    K = np.zeros((n_train, n_train))
    for i in range(n_train):
        for j in range(n_train):
            K[i, j] = vector_kernel(x_train[i], x_train[j], sig)

    c = np.linalg.solve(K + lam * np.eye(n_train), y_train)
    return c

def predict_krr(x_test, x_train, c, sig):
    """Uses the cross-kernel K_s = k(x_test, x_train) to predict."""
    n_test = x_test.shape[0]
    n_train = x_train.shape[0]
    K_s = np.zeros((n_test, n_train))
    for i in range(n_test):
        for j in range(n_train):
            K_s[i, j] = vector_kernel(x_test[i], x_train[j], sig)
    return K_s @ c



# %%
from sklearn.model_selection import train_test_split

Y = np.array(sol['Solubility']).flatten()  # Ensure Y is 1D

# Feature matrix (each row = sample, each column = feature)
proplist = ['HeavyAtomCount', 'NumHAcceptors', 'NumHDonors',
↪ 'NumHeteroatoms',
            'NumRotatableBonds', 'NumValenceElectrons',
            ↪ 'NumAromaticRings',
            'NumSaturatedRings', 'NumAliphaticRings', 'RingCount']
```

```
X = np.array([list(sol[prop] / sol['MolWt']) for prop in proplist])  #
↪  Normalize extensive properties
X = np.insert(X, 0, list(np.log(sol['MolWt'])), axis=0)  # Add log MolWt
X = X.T  # Transpose so rows = samples, columns = features

# Compute lengthscales (std per feature)
sig = np.std(X, axis=0)

# Split into training and test sets
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2,
↪  random_state=42)

print(sig)

# %%
# Define lambda (regularization)
lam = 0.75  # Can be optimized

sig = np.std(X, axis=0)

# 1) Subselect or not from X_train, Y_train
n_inducing_points = 1000
idx = np.random.choice(len(X_train), n_inducing_points, replace=False)
x_sub = X_train[idx]
y_sub = Y_train[idx]

# 2) Train
c = train_krr(x_sub, y_sub, sig, lam)

# 3) Predict on train
y_pred_train = predict_krr(x_sub, x_sub, c, sig)

# 4) Predict on test
y_pred_test = predict_krr(X_test, x_sub, c, sig)

# 5) Compute errors
train_mse = np.mean((y_pred_train - y_sub) ** 2)
test_mse = np.mean((y_pred_test - Y_test) ** 2)
```

```
print(f"Training RMSE: {train_mse**(1/2):.4f}")
print(f"Test RMSE: {test_mse**(1/2):.4f}")

# %%
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Assume train_krr, predict_krr, X_train, Y_train, X_test, Y_test, and
↪  sig are defined.

# 1) Subselect 500 points from X_train, Y_train
n_inducing_points = 1000
idx = np.random.choice(len(X_train), n_inducing_points, replace=False)
x_sub = X_train[idx]
y_sub = Y_train[idx]

# Define the list of lambda values you want to sweep over
lambda_values = [1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1.0, 10.0]

results = []

# Loop over each lambda and evaluate
for lam in lambda_values:
    # 2) Train
    c = train_krr(x_sub, y_sub, sig, lam)

    # 3) Predict on the training subset
    y_pred_train = predict_krr(x_sub, x_sub, c, sig)

    # 4) Predict on the test set
    y_pred_test = predict_krr(X_test, x_sub, c, sig)

    # 5) Compute errors
    train_mse = np.mean((y_pred_train - y_sub) ** 2)
    test_mse = np.mean((y_pred_test - Y_test) ** 2)
    train_rmse = np.sqrt(train_mse)
    test_rmse = np.sqrt(test_mse)
```

```
    # Print to keep track of progress and results
    print(f"Lambda: {lam}")
    print(f"Training RMSE: {train_rmse:.4f}")
    print(f"Test RMSE: {test_rmse:.4f}")
    print("-----------")

    # Save results for plotting later
    results.append({
        'lambda': lam,
        'train_rmse': train_rmse,
        'test_rmse': test_rmse
    })

# Create a pandas DataFrame for easier analysis & visualization later
df_results = pd.DataFrame(results)
print(df_results)


# %%
# Example: you can then plot RMSE vs lambda (optional)
plt.plot(df_results['lambda'], df_results['train_rmse'], label='Train
↪  RMSE')
plt.plot(df_results['lambda'], df_results['test_rmse'], label='Test
↪  RMSE')

# Add a red dashed horizontal line at y = 1
plt.axhline(y=1, color='red', linestyle='--', label='y = 1')

plt.title('RMSE for 1000 training points')
plt.xscale('log')  # often helpful with lambda sweeps
plt.xlabel('Lambda')
plt.ylabel('RMSE')
plt.legend()
plt.show()

# %% [markdown]
# Optional extension
# ---
```

```
#
# You can optimise the £\sigma£ array formally, by minimising an
↪  objective function, e.g. the error on a _test set_, or even better,
# average error on multiple test sets (given multiple random test/train
↪  splits). You can make the regulariser £\lambda£ part of the
# optimisation too.

# %% [markdown]
#

# %%
import numpy as np
import pandas as pd
from scipy.optimize import minimize
from sklearn.model_selection import train_test_split

# Store all results for later analysis
results_list = []

# Define the objective function: RMSE on test data
def krr_objective(params, X, Y, test_size=0.2, random_state=42):
    """
    Objective function to optimize sigma (lengthscales) and lambda.
    The params array consists of:
        - First d values: sigma (lengthscale per feature)
        - Last value: lambda (regularization term)
    Returns: Average RMSE over multiple train-test splits.
    """
    d = X.shape[1]  # Number of features
    sig = np.abs(params[:d])  # Ensure sigma values are positive
    lam = np.abs(params[d])   # Ensure lambda is positive

    # Perform multiple train-test splits and compute the average RMSE
    num_splits = 5  # Number of random splits
    total_rmse = 0

    for split in range(num_splits):
        X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
        ↪  test_size=test_size, random_state=random_state+split)
```

```
        # Train KRR model **only on training data**
        c = train_krr(X_train, Y_train, sig, lam)

        # Predict on test set
        Y_pred_test = predict_krr(X_test, X_train, c, sig)

        # Compute RMSE
        rmse = np.sqrt(np.mean((Y_pred_test - Y_test) ** 2))
        total_rmse += rmse

        # Print intermediate results
        print(f"Split {split+1}/{num_splits}: RMSE = {rmse:.4f}")

    avg_rmse = total_rmse / num_splits  # Compute average RMSE
    print(f"\nLambda: {lam:.6f}, Average RMSE: {avg_rmse:.4f}")

    # Store results in the list
    results_list.append({
        'sigma': sig.copy(),
        'lambda': lam,
        'avg_rmse': avg_rmse
    })

    return avg_rmse  # Return RMSE for optimization

# Initialize sigmas to feature standard deviations and lambda to a
↪ reasonable value
initial_sigma = np.std(X, axis=0)  # Initial sigma values
initial_lambda = 0.1  # Initial lambda value (from previous
↪ investigations)
initial_params = np.hstack([initial_sigma, initial_lambda])

# Optimize sigma and lambda
result = minimize(
    krr_objective, initial_params, args=(X, Y),
    method='L-BFGS-B', bounds=[(1e-6, None)] * (X.shape[1] + 1),  #
    ↪ Ensure positivity
    options={'disp': True}
```

```
)

# Extract optimized sigmas and lambda
optimal_sigma = result.x[:-1]  # Optimized sigma values
optimal_lambda = result.x[-1]   # Optimized lambda

print("\n--- Optimization Complete ---")
print("Optimized Sigma:", optimal_sigma)
print("Optimized Lambda:", optimal_lambda)

# Convert results list to DataFrame for later visualization
df_results = pd.DataFrame(results_list)
print("\nAll results:\n", df_results)


# %% [markdown]
# <a
↪  style='text-decoration:none;line-height:16px;display:flex;color:#5B5B62;padding
↪  href='https://deepnote.com?utm_source=created-in-deepnote-cell&projectId=9fbaf2
↪  target="_blank">
# <img alt='Created in deepnote.com'
↪  style='display:inline;max-height:16px;margin:0px;margin-right:7.5px;'
↪  src='
↪  > </img>
# Created in <span
↪  style='font-weight:600;margin-left:4px;'>Deepnote</span></a>

# %%
def vector_kernel_matrix(X1, X2, sig):
    """
    Compute the full Gaussian kernel matrix without loops.
    """
    X1_sq = np.sum(X1**2, axis=1).reshape(-1, 1)
    X2_sq = np.sum(X2**2, axis=1).reshape(1, -1)
    cross_term = -2 * np.dot(X1, X2.T)

    return np.exp(-(X1_sq + X2_sq + cross_term) / (2 * sig**2))

def train_krr(X_train, Y_train, sig, lam):
```

```
    """Vectorized kernel ridge regression."""
    K = vector_kernel_matrix(X_train, X_train, sig)
    return np.linalg.solve(K + lam * np.eye(len(X_train)), Y_train)

def predict_krr(X_test, X_train, c, sig):
    """Vectorized prediction."""
    K_s = vector_kernel_matrix(X_test, X_train, sig)
    return np.dot(K_s, c)


# %% [markdown]
# # Lets try some Neural Nets

# %% [markdown]
# simple MLP model - with lr 0.01 and not much else

# %%
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import numpy as np

# Convert data to the correct shape (N_samples, 11)
X = np.array([list(sol[prop]/sol['MolWt']) for prop in proplist])  #
↪  Normalize extensive properties
X = np.insert(X, 0, list(np.log(sol['MolWt'])), axis=0)  # Add log
↪  molecular weight
X = X.T  # Transpose to shape (N, 11)
Y = np.array(sol['Solubility']).reshape(-1, 1)  # Shape (N, 1)

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.2,
↪  random_state=42)

# Normalize inputs (zero mean, unit variance)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
```

```
X_test = scaler.transform(X_test)

# Convert to PyTorch tensors
X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train, dtype=torch.float32)
X_test_tensor = torch.tensor(X_test, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test, dtype=torch.float32)

# Define the neural network model
class SolubilityNN(nn.Module):
    def __init__(self):
        super(SolubilityNN, self).__init__()
        self.fc1 = nn.Linear(11, 64)  # First hidden layer
        self.fc2 = nn.Linear(64, 32)  # Second hidden layer
        self.fc3 = nn.Linear(32, 1)   # Output layer

        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.fc3(x)  # No activation in the final layer (regression
        ↪  task)
        return x

# Initialize model, loss function, and optimizer
model = SolubilityNN()
criterion = nn.MSELoss()  # Mean Squared Error for regression
optimizer = optim.Adam(model.parameters(), lr=0.01)

# Training loop
num_epochs = 200
batch_size = 32
dataset = torch.utils.data.TensorDataset(X_train_tensor, y_train_tensor)
dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size,
↪  shuffle=True)

for epoch in range(num_epochs):
    for batch_X, batch_y in dataloader:
```

```
            optimizer.zero_grad()
            predictions = model(batch_X)
            loss = criterion(predictions, batch_y)
            loss.backward()
            optimizer.step()

        if epoch % 20 == 0:
            print(f"Epoch {epoch}, Loss: {loss.item():.4f}")

# Evaluate model
with torch.no_grad():
    y_pred = model(X_test_tensor)
    test_rmse = torch.sqrt(criterion(y_pred, y_test_tensor)).item()

print(f"Test RMSE: {test_rmse:.4f}")


# %%
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.model_selection import KFold
from sklearn.preprocessing import StandardScaler
import numpy as np

# Convert data to the correct shape (N_samples, 11)
X = np.array([list(sol[prop] / sol['MolWt']) for prop in proplist])  #
↪ Normalize extensive properties
X = np.insert(X, 0, list(np.log(sol['MolWt'])), axis=0)  # Add log
↪ molecular weight
X = X.T  # Transpose to shape (N, 11)
Y = np.array(sol['Solubility']).reshape(-1, 1)  # Shape (N, 1)

# Normalize inputs (zero mean, unit variance)
scaler = StandardScaler()
X = scaler.fit_transform(X)  # Scale all data

# Convert to PyTorch tensors
X_tensor = torch.tensor(X, dtype=torch.float32)
```

```
Y_tensor = torch.tensor(Y, dtype=torch.float32)

# Define the neural network model
class SolubilityNN(nn.Module):
    def __init__(self):
        super(SolubilityNN, self).__init__()
        self.fc1 = nn.Linear(11, 64)  # First hidden layer
        self.fc2 = nn.Linear(64, 32)  # Second hidden layer
        self.fc3 = nn.Linear(32, 1)   # Output layer

        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(0.2)  # Dropout to prevent overfitting

    def forward(self, x):
        x = self.relu(self.fc1(x))
        x = self.dropout(x)  # Apply dropout after activation
        x = self.relu(self.fc2(x))
        x = self.fc3(x)  # No activation in the final layer (regression
        ↪  task)
        return x

# Hyperparameters
num_epochs = 200
batch_size = 128  # Increased batch size for stability
learning_rate = 0.001  # Lowered LR for more stable training
k_folds = 5  # 5-Fold Cross-Validation

kf = KFold(n_splits=k_folds, shuffle=True, random_state=42)
fold_rmse_scores = []

# Cross-validation loop
for fold, (train_idx, val_idx) in enumerate(kf.split(X)):
    print(f"Training Fold {fold+1}/{k_folds}...")

    # Split data into training and validation sets
    X_train, X_val = X_tensor[train_idx], X_tensor[val_idx]
    y_train, y_val = Y_tensor[train_idx], Y_tensor[val_idx]

    # Create DataLoader for batch training
```

```
    train_dataset = torch.utils.data.TensorDataset(X_train, y_train)
    train_loader = torch.utils.data.DataLoader(train_dataset,
    ↪  batch_size=batch_size, shuffle=True)

    # Initialize model, loss function, and optimizer
    model = SolubilityNN()
    criterion = nn.MSELoss()
    optimizer = optim.Adam(model.parameters(), lr=learning_rate)
    scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=50,
    ↪  gamma=0.5)  # Reduce LR every 50 epochs

    # Training loop
    for epoch in range(num_epochs):
        model.train()
        for batch_X, batch_y in train_loader:
            optimizer.zero_grad()
            predictions = model(batch_X)
            loss = criterion(predictions, batch_y)
            loss.backward()
            optimizer.step()

        scheduler.step()  # Adjust learning rate

        # Print loss every 20 epochs
        if epoch % 20 == 0:
            print(f"Fold {fold+1}, Epoch {epoch}, Loss:
            ↪  {loss.item():.4f}")

    # Evaluate model on validation set
    model.eval()
    with torch.no_grad():
        y_pred = model(X_val)
        val_rmse = torch.sqrt(criterion(y_pred, y_val)).item()

    fold_rmse_scores.append(val_rmse)
    print(f"Fold {fold+1} Validation RMSE: {val_rmse:.4f}")

# Compute average RMSE across folds
avg_rmse = np.mean(fold_rmse_scores)
```

```
print(f"Average Cross-Validation RMSE: {avg_rmse:.4f}")


# %%
import torch
import gpytorch
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Convert data to the correct shape (N_samples, 11)
X = np.array([list(sol[prop] / sol['MolWt']) for prop in proplist])  #
↪  Normalize extensive properties
X = np.insert(X, 0, list(np.log(sol['MolWt'])), axis=0)  # Add log
↪  molecular weight
X = X.T  # Transpose to shape (N, 11)
Y = np.array(sol['Solubility']).reshape(-1, 1)  # Shape (N, 1)

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.2,
↪  random_state=42)

# Normalize inputs
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Convert to PyTorch tensors
X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train.flatten(), dtype=torch.float32)
X_test_tensor = torch.tensor(X_test, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test.flatten(), dtype=torch.float32)

# Define Gaussian Process Model
class GPModel(gpytorch.models.ExactGP):
    def __init__(self, train_x, train_y, likelihood):
        super(GPModel, self).__init__(train_x, train_y, likelihood)
        self.mean_module = gpytorch.means.ConstantMean()
        self.covar_module = gpytorch.kernels.ScaleKernel(
```

```
            gpytorch.kernels.RBFKernel(ard_num_dims=11)  # ARD: learns a
            ↪ different lengthscale per feature
        )

    def forward(self, x):
        mean_x = self.mean_module(x)
        covar_x = self.covar_module(x)
        return gpytorch.distributions.MultivariateNormal(mean_x, covar_x)

# Use Gaussian likelihood for regression
likelihood = gpytorch.likelihoods.GaussianLikelihood()
model = GPModel(X_train_tensor, y_train_tensor, likelihood)

# Switch to training mode
model.train()
likelihood.train()

# Define optimizer & marginal likelihood loss
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
mll = gpytorch.mlls.ExactMarginalLogLikelihood(likelihood, model)

# Training loop
num_epochs = 200
for epoch in range(num_epochs):
    optimizer.zero_grad()
    output = model(X_train_tensor)
    loss = -mll(output, y_train_tensor)  # Negative log likelihood
    loss.backward()
    optimizer.step()

    if epoch % 20 == 0:
        print(f"Epoch {epoch}, Loss: {loss.item():.4f}")

# Switch to evaluation mode
model.eval()
likelihood.eval()

# Make predictions
with torch.no_grad():
```

```
    pred = likelihood(model(X_test_tensor))
    y_pred = pred.mean
    test_rmse = torch.sqrt(torch.mean((y_pred - y_test_tensor) **
    ↪  2)).item()

print(f"Test RMSE: {test_rmse:.4f}")


# %%
print(np.std(Y))  # If std(Y) < 1, then RMSE < 1 may be realistic.


# %%


# %% [markdown]
# So we have achieved better than this result:
↪  https://pubs.acs.org/doi/10.1021/acs.jcim.1c00331

# %%
```

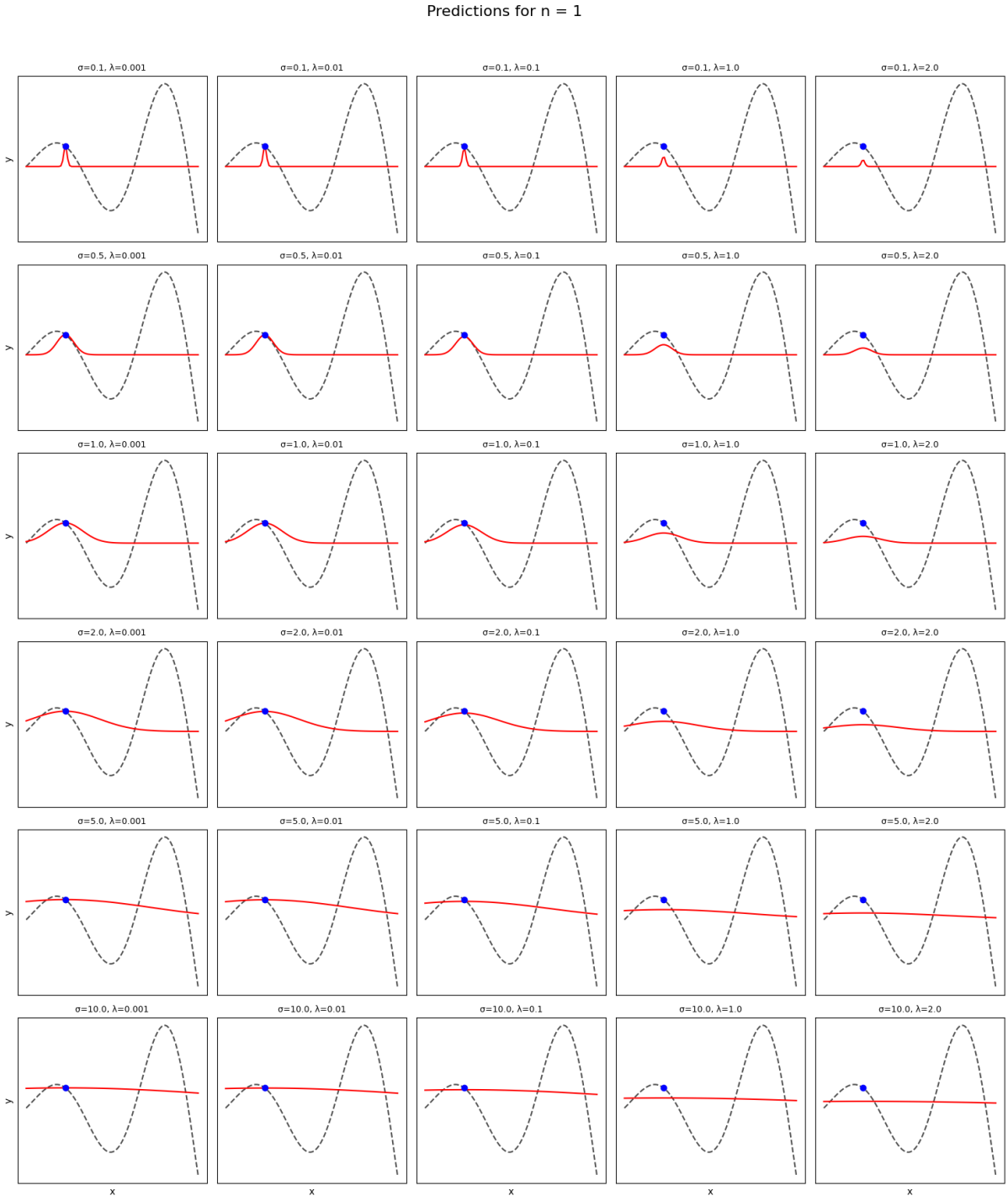


Figure 18: Kernel Ridge Regression Prediction for $n = 1$

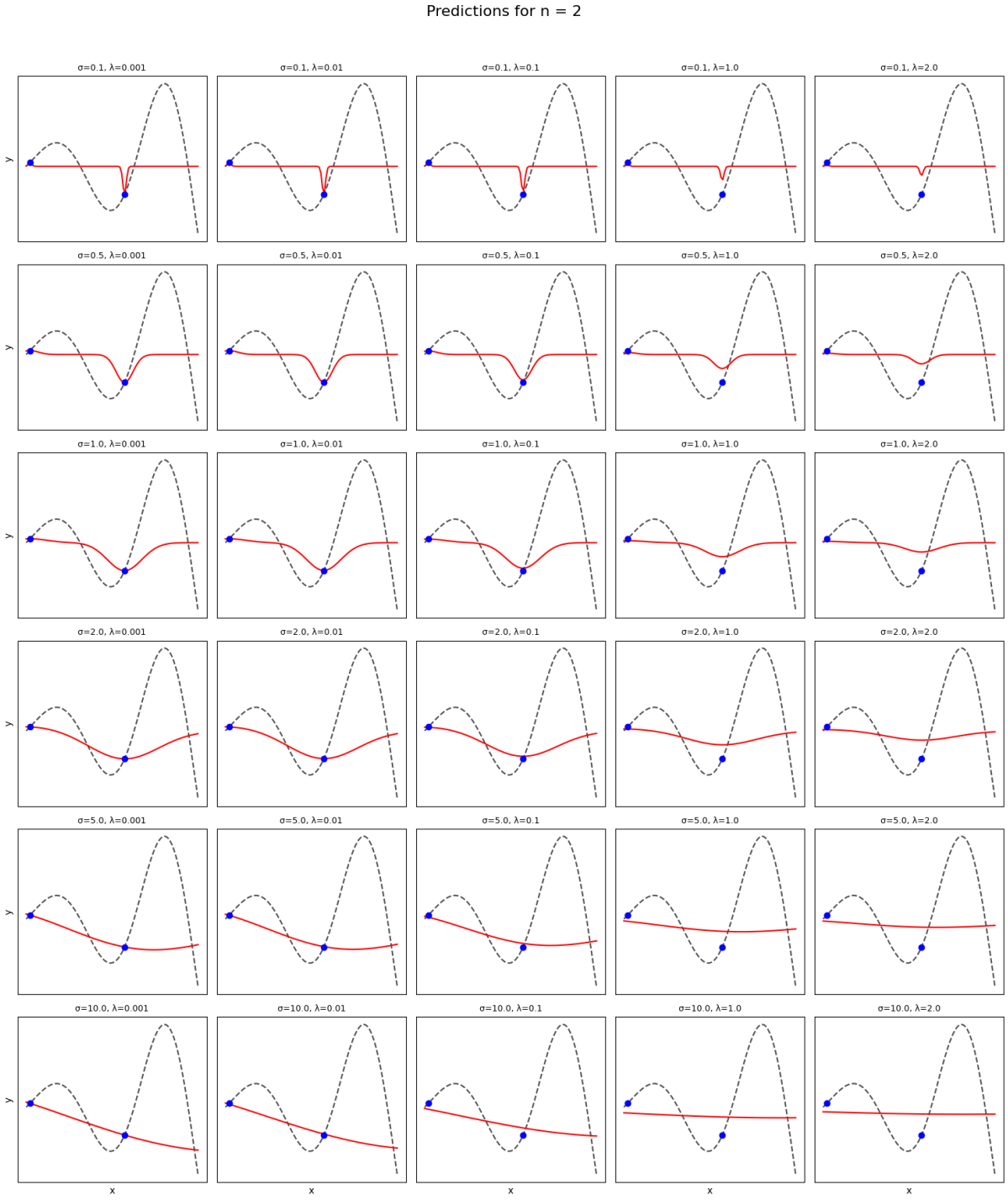


Figure 19: Kernel Ridge Regression Prediction for $n = 2$

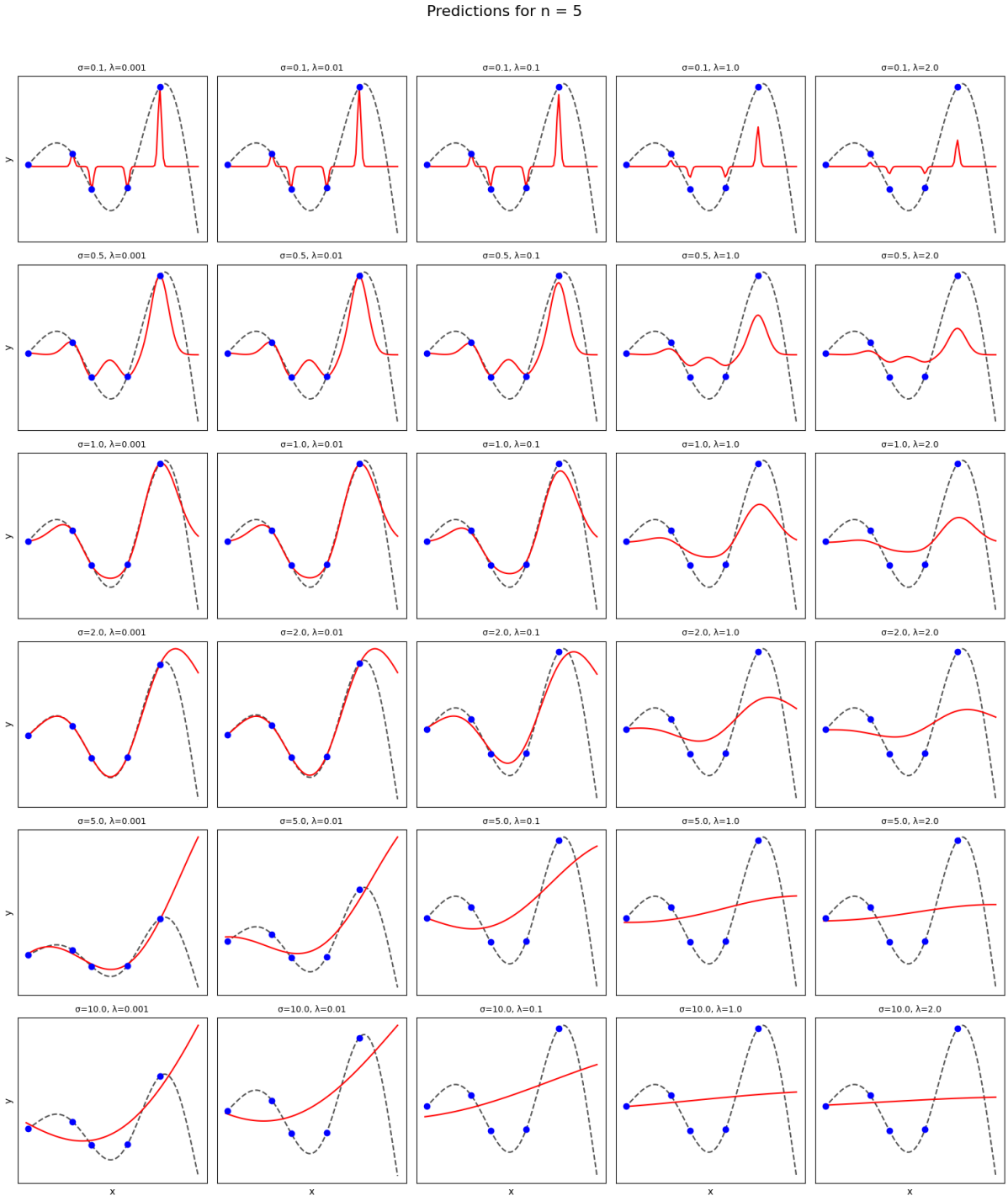


Figure 20: Kernel Ridge Regression Prediction for $n = 5$

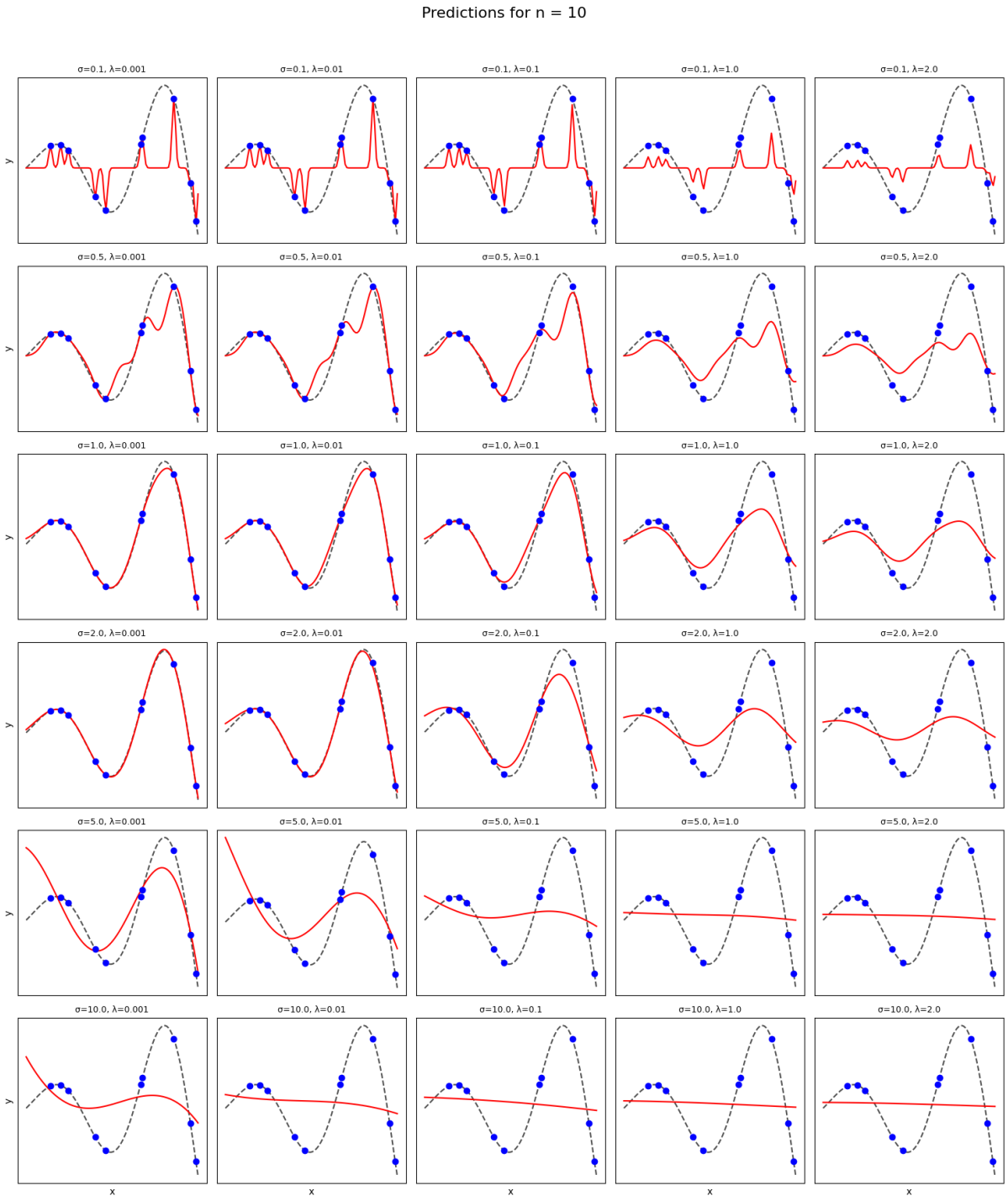


Figure 21: Kernel Ridge Regression Prediction for $n = 10$

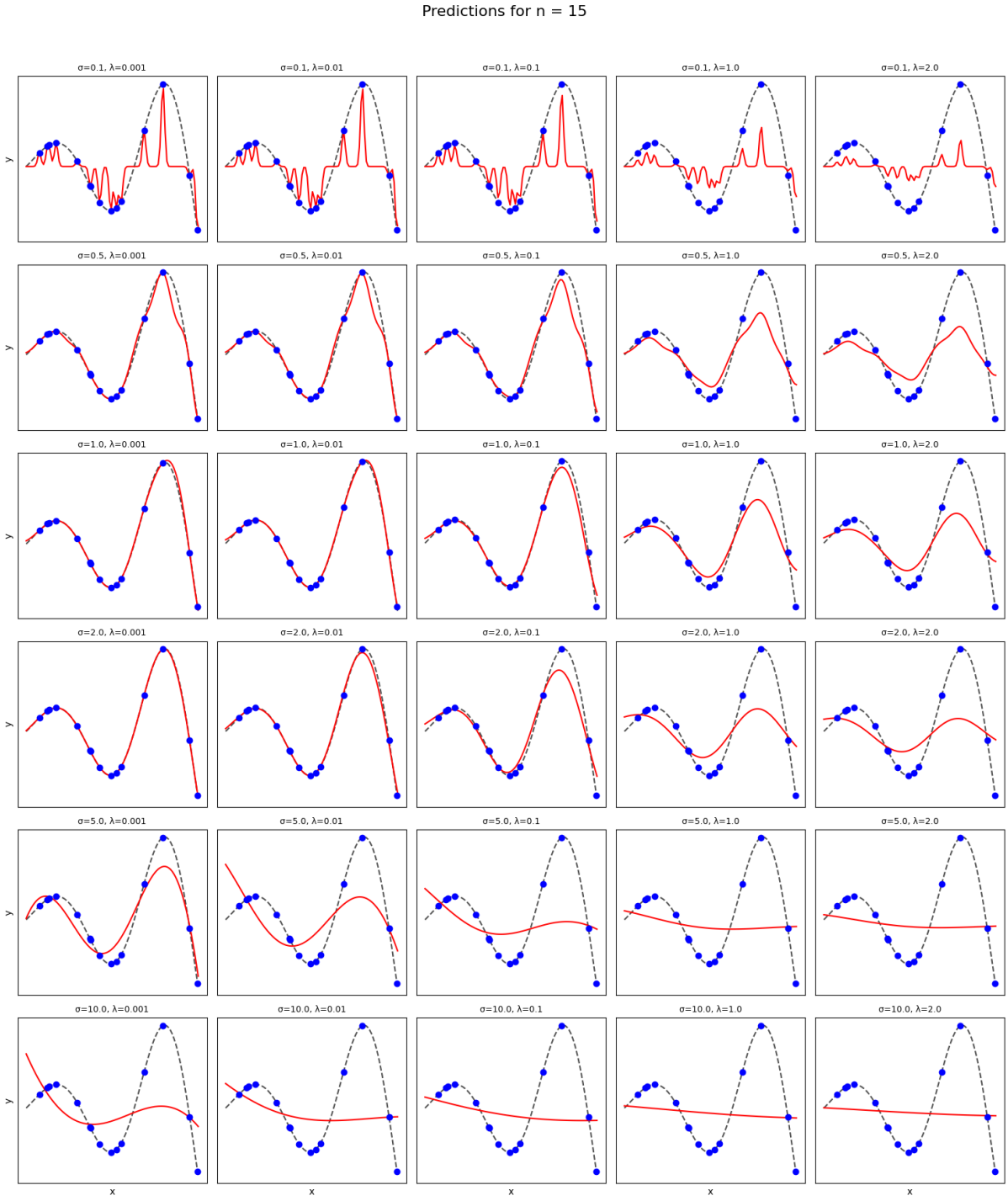


Figure 22: Kernel Ridge Regression Prediction for $n = 15$

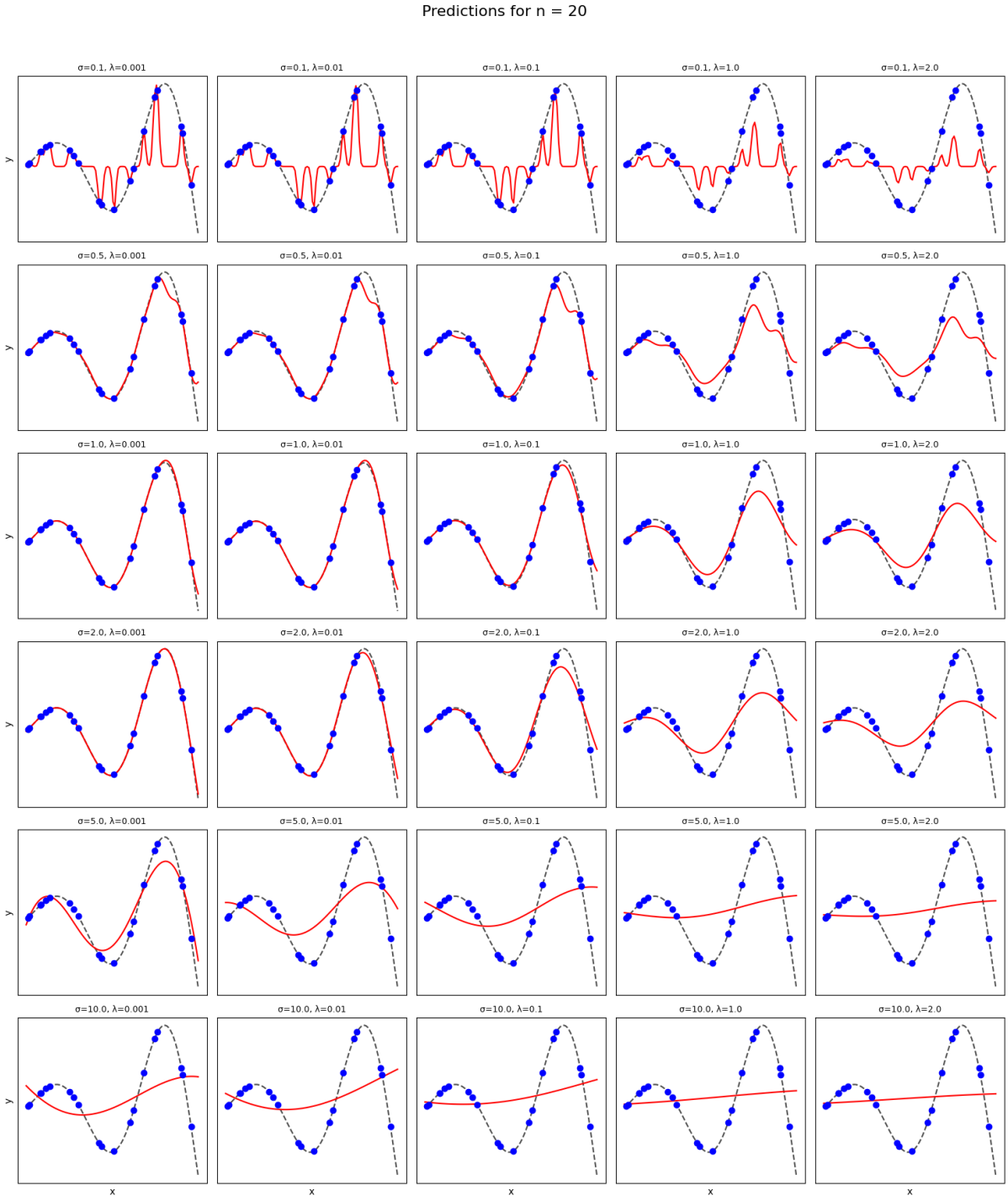


Figure 23: Kernel Ridge Regression Prediction for $n = 20$

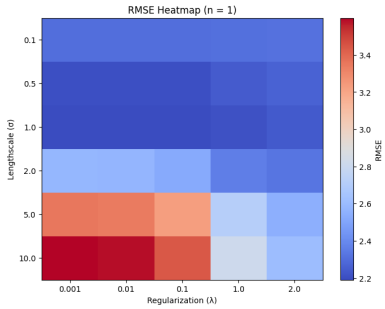


Figure 24: Heatmap for $n = 1$ with noisy data

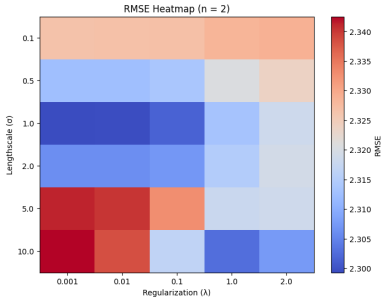


Figure 25: Heatmap for $n = 2$ with noisy data

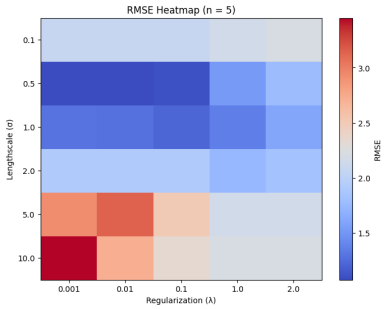


Figure 26: Heatmap for $n = 5$ with noisy data

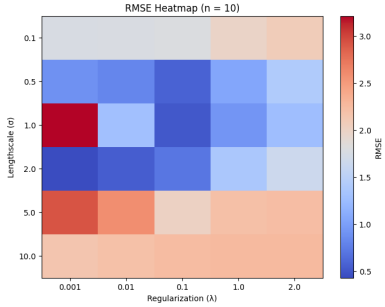


Figure 27: Heatmap for $n = 10$ with noisy data

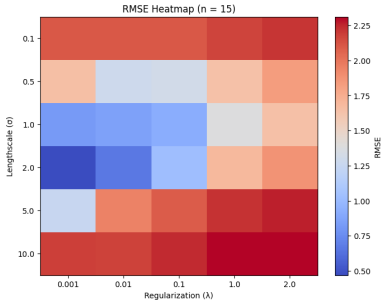


Figure 28: Heatmap for $n = 15$ with noisy data

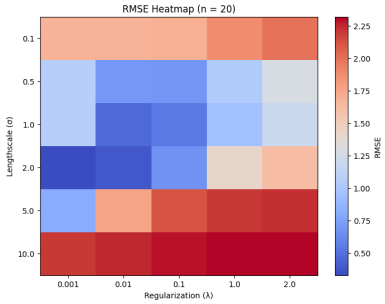


Figure 29: Heatmap for $n = 20$ with noisy data

Figure 30: Comparison of heatmaps across different training set sizes $n$ with noisy data.

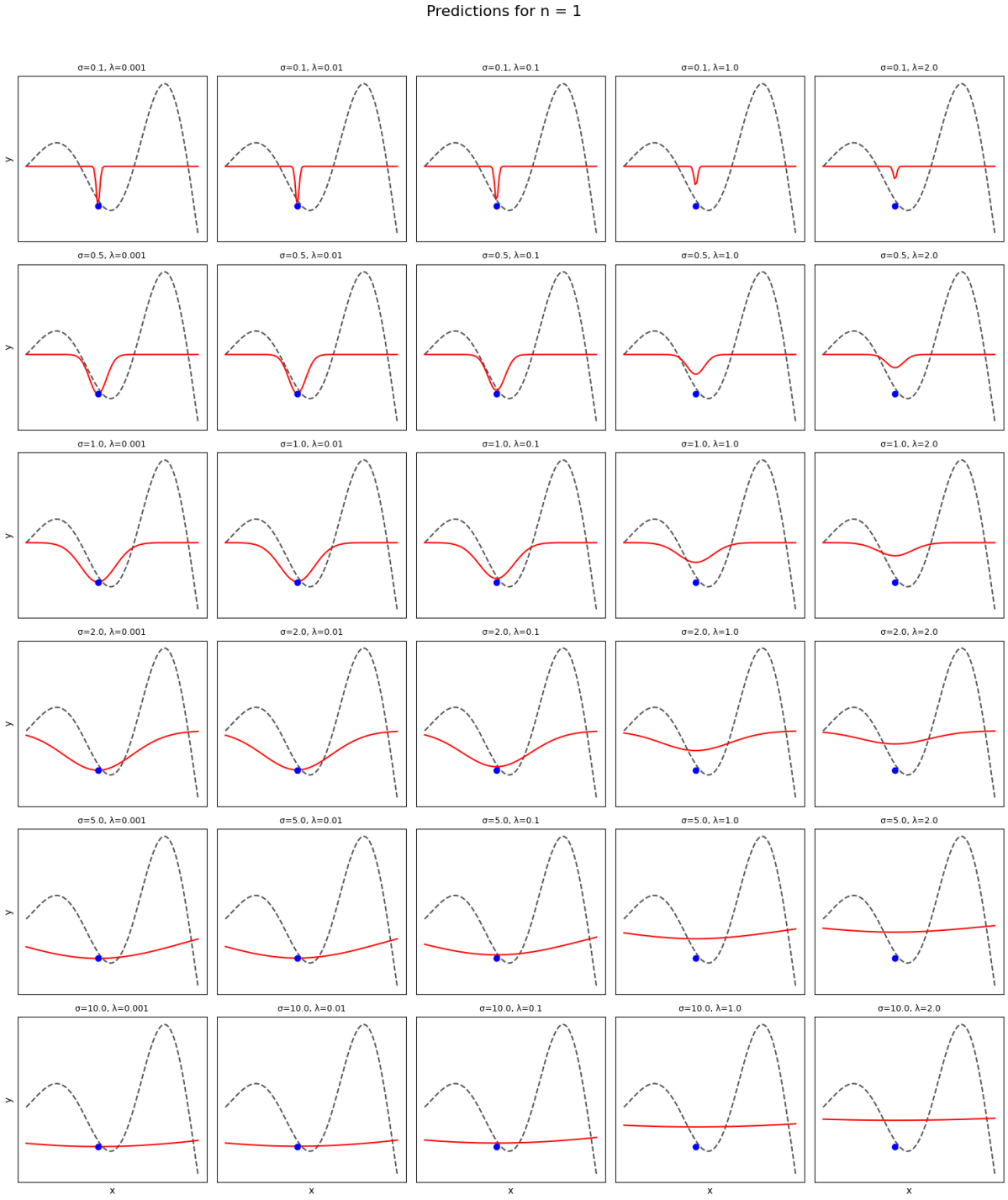


Figure 31: Kernel ridge prediction with noisy data for $n = 1$

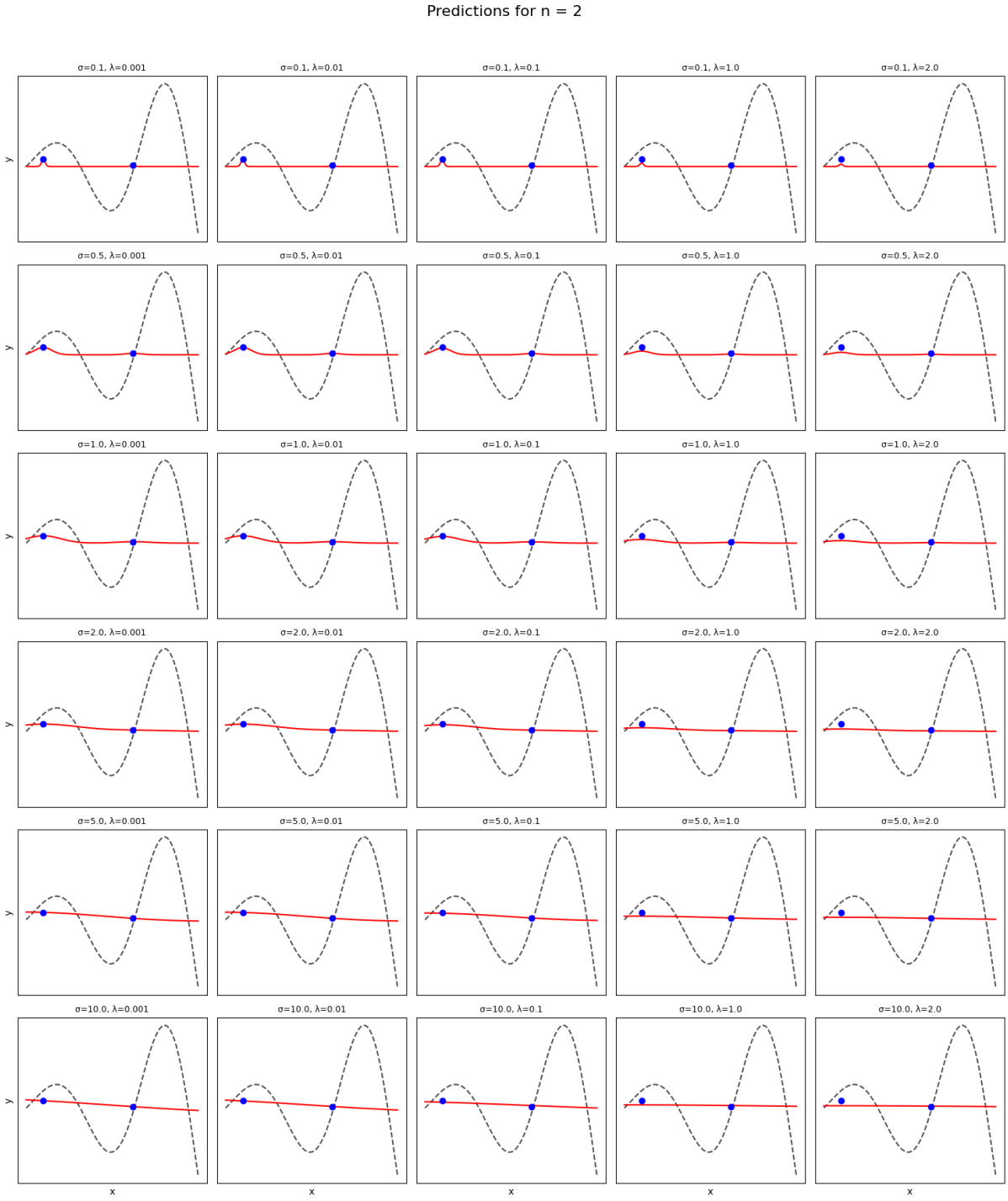


Figure 32: Kernel ridge prediction with noisy data for $n = 2$

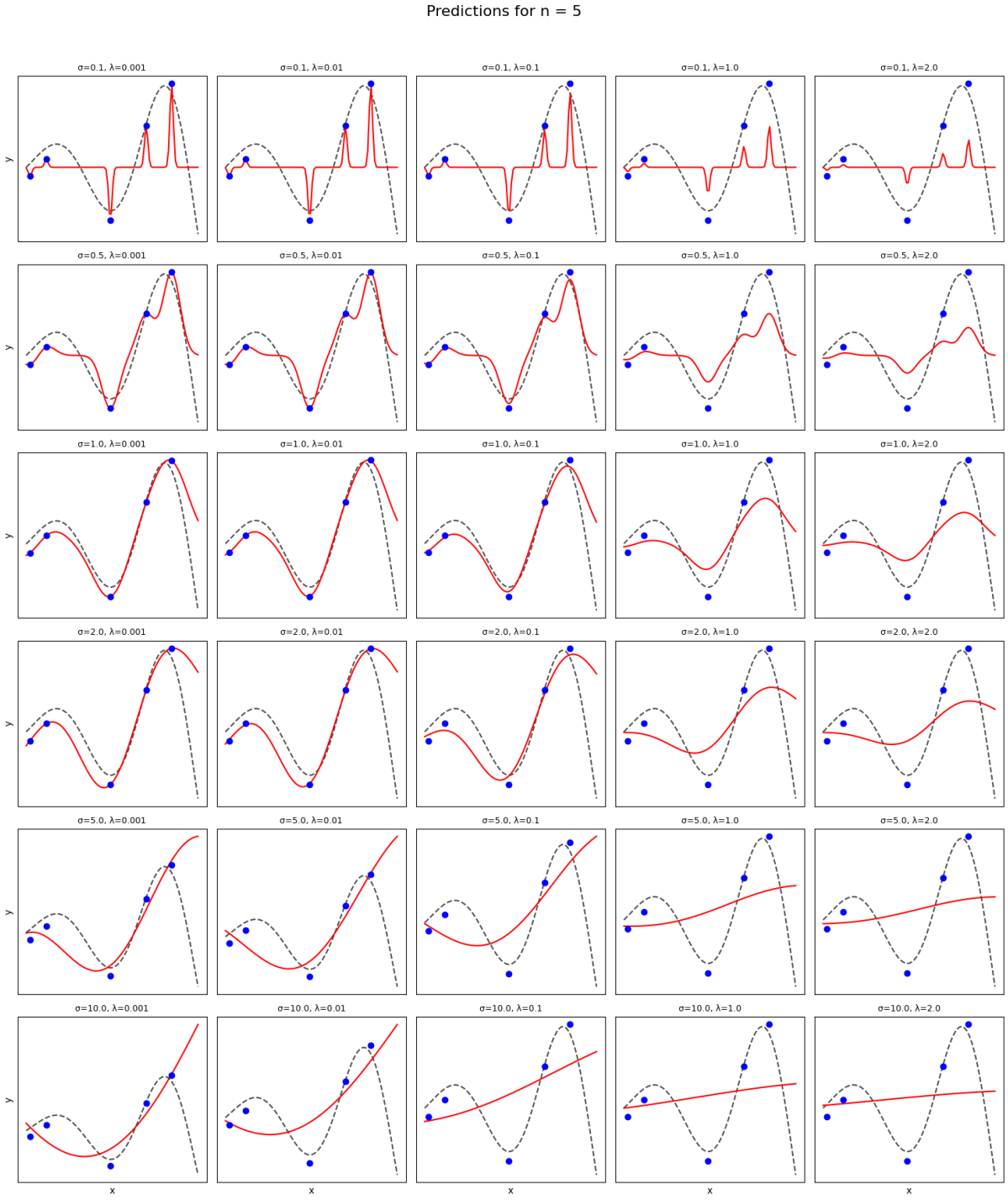


Figure 33: Kernel ridge prediction with noisy data for $n = 5$

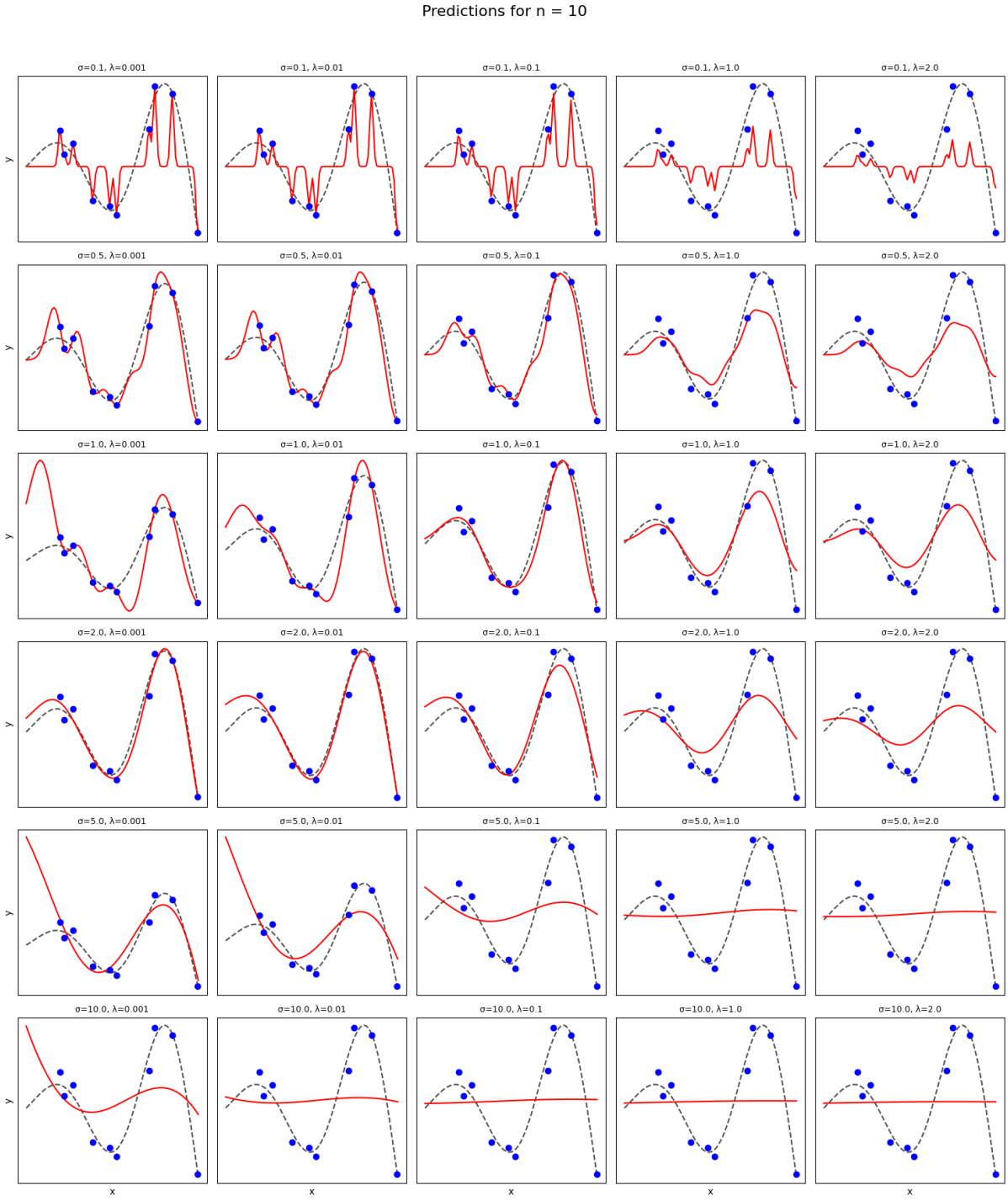


Figure 34: Kernel ridge prediction with noisy data for $n = 10$

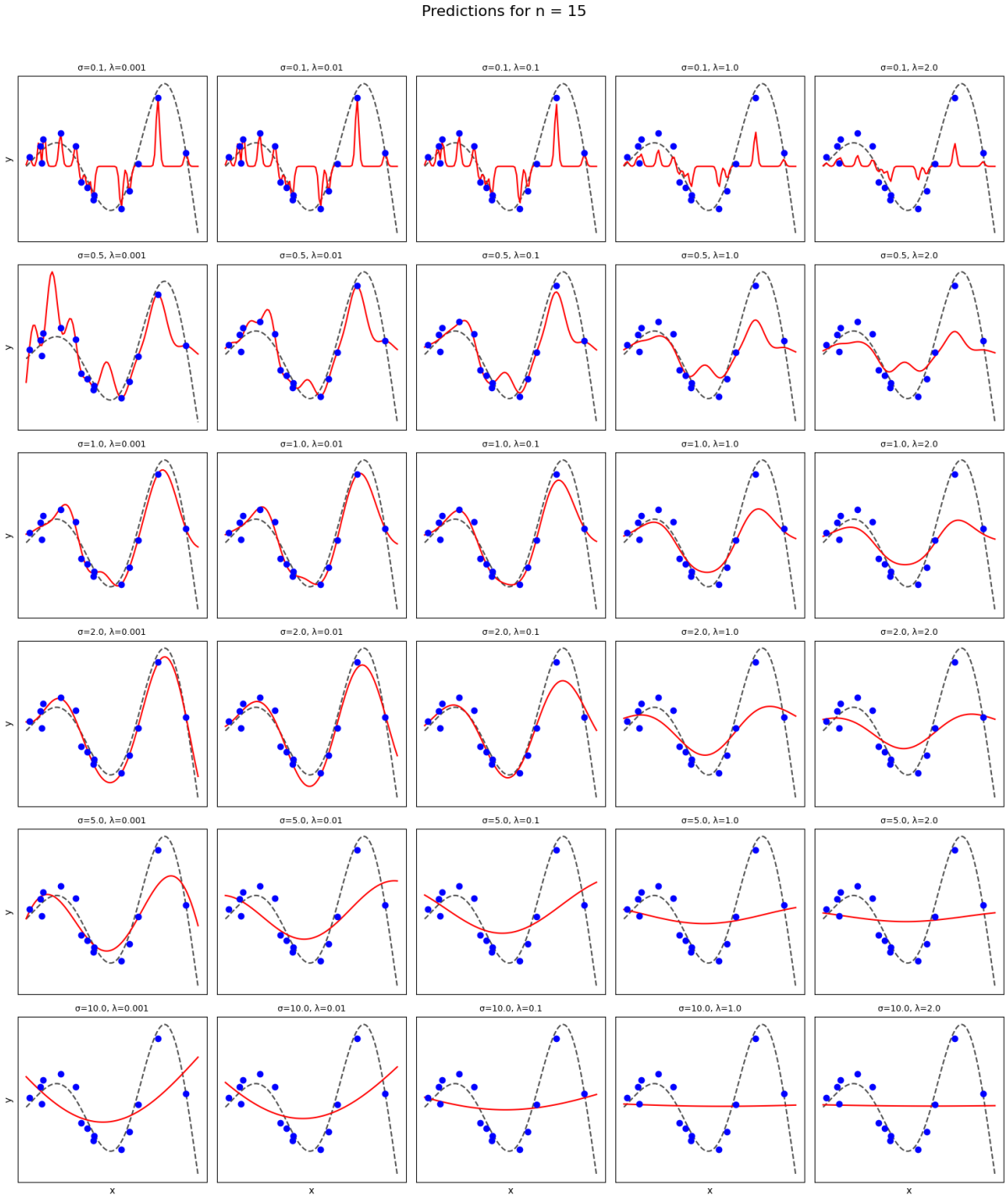


Figure 35: Kernel ridge prediction with noisy data for $n = 15$

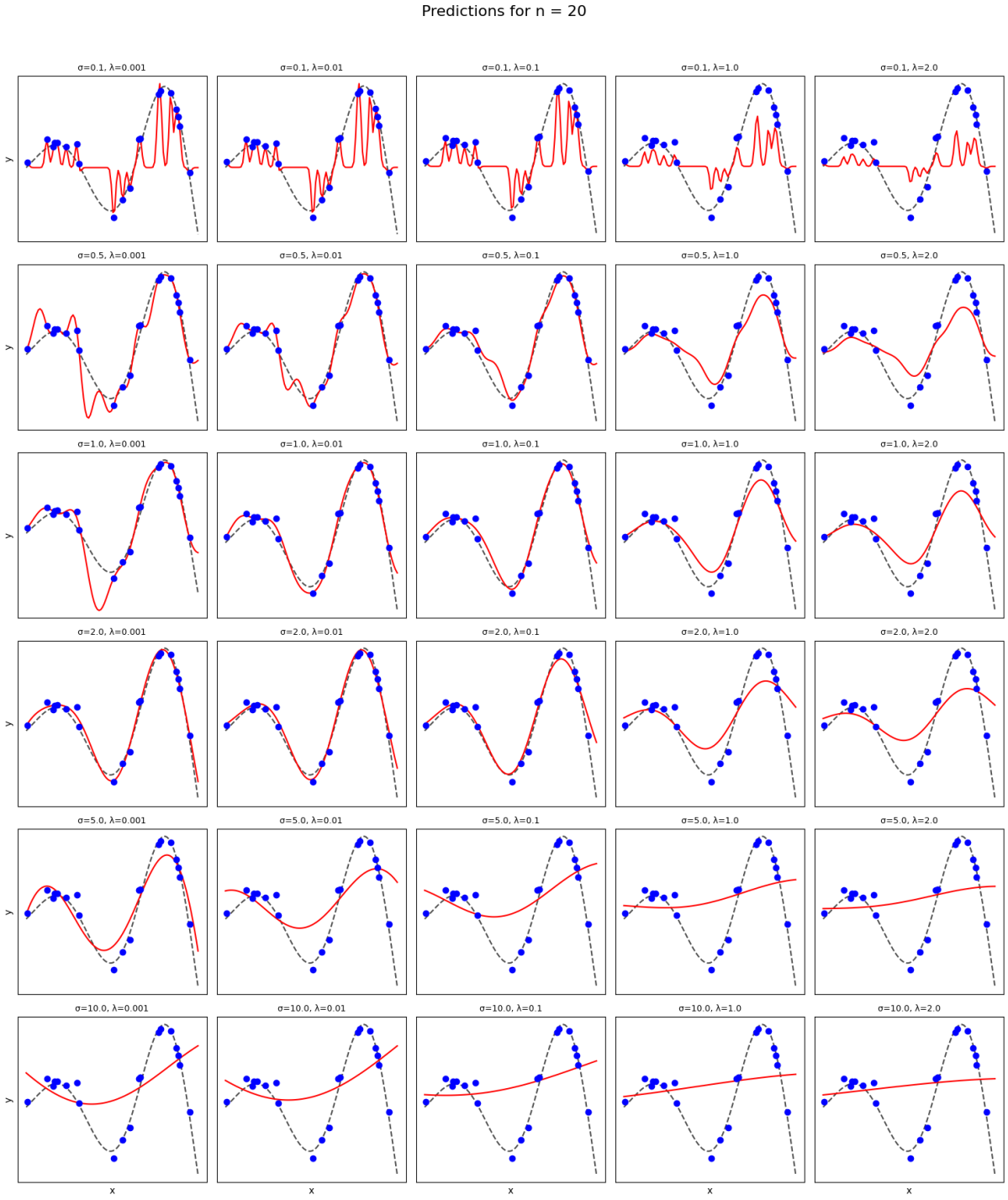


Figure 36: Kernel ridge prediction with noisy data for $n = 20$