COMP0078 Supervised Learning

Coursework 1 Report

Team: Yes
Student ID: 18064052 & 17126493

Department of Computer Science

December 21, 2022

Contents

1 Introduction **1**

2 Part 1 **1**

2.1 Linear Regression 1

2.1.1 Question 1: Fitting a polynomial basis 1

2.1.2 Question 2: Demonstrating over-fitting 2

2.1.3 Question 3: Fitting a sinusoidal basis 6

2.2 Filtered Boston Housing Linear and Kernel Regression 9

2.2.1 Question 4: Baseline versus full linear regression 9

2.2.2 Question 5: Kernelised Ridge Regression 10

2.2.3 Averaging our results over 20 random data splits 11

3 Part 2 **12**

3.1 k-Nearest Neighbours 12

3.1.1 Question 6: Visualisation of one sampled hypothesis 12

3.1.2 Question 7: Estimated generalisation error of k-NN as a function of k 13

3.1.3 Question 8: Determining the optimal k as a function of the size of the training set (m) 14

4 Part 3 **15**

4.1 Question 9: Dot-Product Kernel with a Constant 15

4.1.1 Part A: Proving K_c is PSD 15

4.1.2 Part B: The effect of c when using K_c in linear regression 16

4.2 Question 10: Simulating 1-NN Through Gaussian Kernel Regression . . . 17

4.3 Question 11: Whack-A-Mole 19

1 Introduction

A brief introduction, this report details our results and analysis for each experiment within the coursework 1 question sheet. We are delighted with our outcomes for each experiment, and we enjoyed the challenging nature of the questions. It was satisfying building the algorithms from scratch and not using libraries for a change!

2 Part 1

2.1 Linear Regression

2.1.1 Question 1: Fitting a polynomial basis

In question 1 we are performing non-linear regression through a non-linear transformation on a standard dataset. The non-linear transformation of choice is an nth-order polynomial basis function such that $\phi(x) = x^{k-1}$.

For a given dataset $x = \{(1,3),(2,2),(3,0),(4,5)\}$ we fit 4 versions of our non-linear regressions model, each for a polynomial order of k increasing from $1 \rightarrow 4$. Figure 1 shows a superimposed plot of each regression fit for each polynomial order transformation on our original dataset.

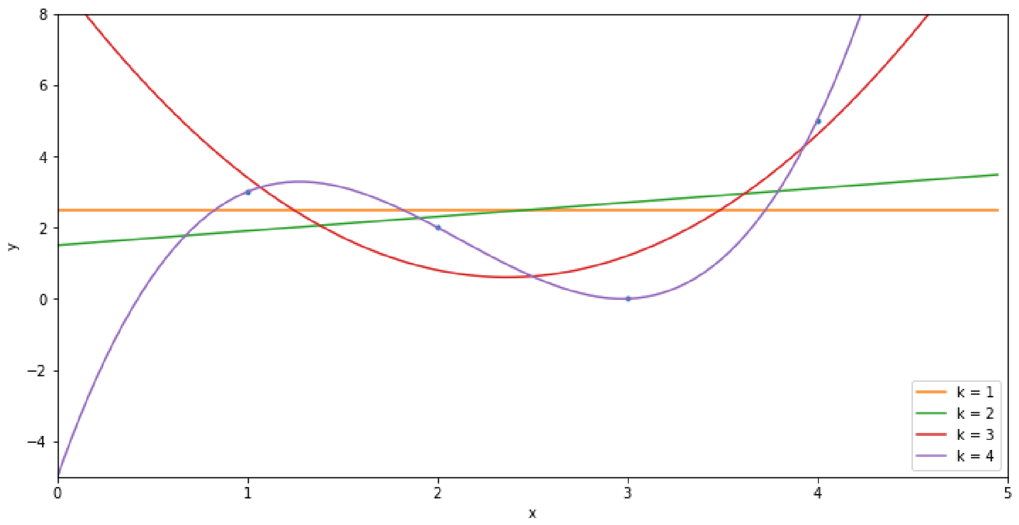

Figure 1: Superimposed plot of each degree polynomial fit on dataset.

For each curve we may also extract the underlying polynomial equation that defines its fit on our data. In polynomial regression our weight matrix, $w : w \in \mathbb{R}^k$, contains entries each corresponding to the weight associated with each of the orders of the polynomial fit. We can therefore extract these weights to obtain the polynomial coefficients as seen in Table 1.

Further to this we want to be able to quantify the accuracy of each polynomial fit. We can use mean-square-error (MSE) to achieve this as MSE is a way of measuring the absolute differences between the predictions and true values, given as:

$$MSE = \frac{1}{m} \sum_{t=1}^{m} (y_t - \vec{w} \cdot \vec{x_t}) \tag{1}$$

The MSE is calculated for each degree polynomial and can also be found in Table 1.

Table 1: Each order polynomial equation and its mean-squared-error on the dataset.

k	Polynomial Equation	MSE
k = 1	2.5	3.25
k = 2	$1.5 + 0.4x$	3.05
k = 3	$9 + -7.1x + 1.5x^2$	0.8
k = 4	$-5 + 15.17x + -8.5x^2 + 1.33x^3$	$3.5 * 10^{-23}$

The results demonstrate that as the order of the polynomial (k) increases, we obtain a better fit to our data, as indicated by the decreasing MSE for each polynomial fit. Of course this may not be the case at higher orders. The minimum error achieved occurs at $k = 4$ and is due to the polynomial order equating to the number of data points. This would typically lead to overfitting for a larger dataset but due to the simple nature of our points and small dataset, it is able to be modelled well by this degree fit.

2.1.2 Question 2: Demonstrating over-fitting

In question 2 we want to observe the effect of overfitting by introducing noise into data and measuring how higher order polynomials fit this noisy data.

We first define our data points x_i as 30 points sampled uniformly at random from the interval $[0, 1]$ thereby giving us $(x_1, ..., x_{30})$. Each data point's corresponding y value is assigned as a sinusoidal transformation on the datum with some additional normally distributed noise with mean 0 and variance σ^2 as given by equation 2.

$$y \leftarrow g_\sigma(x) : g_\sigma(x) = sin^2(2\pi x) + \epsilon \tag{2}$$

For now, we set $\sigma = 0.07$ and apply this function to all of our sampled $x_{i_{1...30}}$ values to obtain the corresponding $y_{i_{1...30}}$ values to construct our dataset.

To give a visualisation we plot the standard $sin^2(2\pi x)$ function and superimpose our 'noisy' data points on top (see Figure 2.)

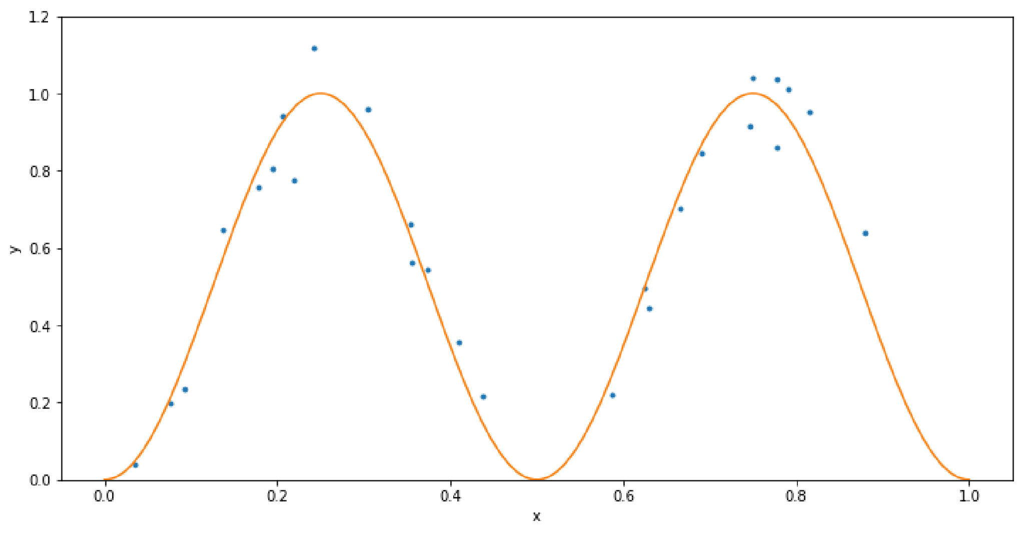

Figure 2: Superimposed plot standard sinusoid function wrt our noisy data points.

We then fit the dataset, S, with a polynomial basis of dimensions $k = \{2, 5, 10, 14, 18\}$ using the same non-linear regression method as in Question 1. A plot showing each of the 5 polynomial fits superimposed on our dataset can be seen in Figure 3.

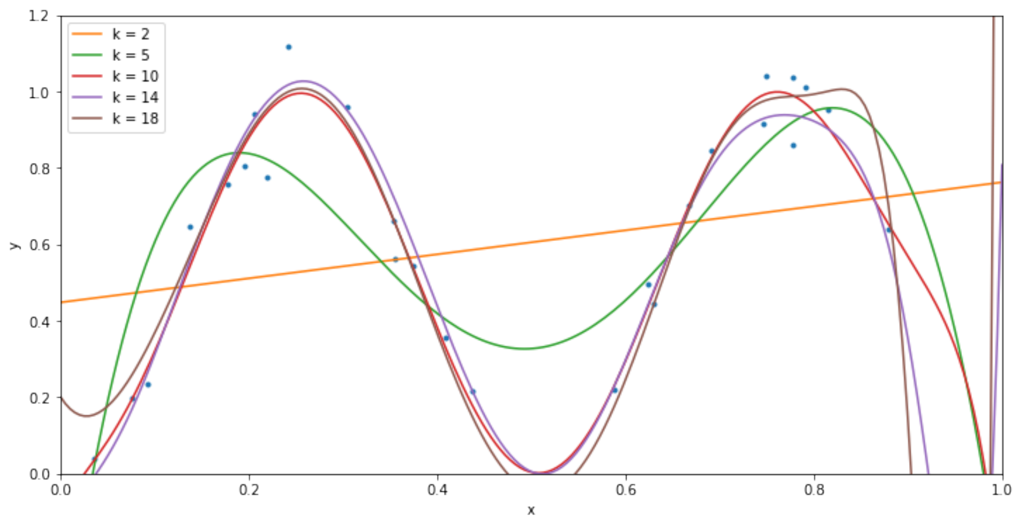

Figure 3: Each polynomial degree fit on our dataset, S.

The plot is somewhat cluttered, however we can see that the lower order polynomial terms, 2 and 5, have a much poorer fit to our data than the higher order terms. Once again, to quantify the accuracy of our fits we can compute the MSE on the training set, denoted $te_k(S)$, but this time for each polynomial degree $k = \{1, ..., 18\}$ to get a better understanding of how great our fits are for an increasing k.

However, the MSE on the training set does not tell how consistent our model is on new data points. We wish to obtain a more holistic view of our model performance with regard to k.

Let us generate a test set, T, of 1000 points, sampled uniformly at random with corresponding outputs the same as Equation 2 (sampled the same way as the training set), and computing the test-set MSE, $tse_k(S, T)$, on these points. We take the natural log (Ln) of each error to get a clearer comparison between small and large errors, and plot it against the polynomial degrees. Figure 4 shows the training error plot and Figure 5 shows the test error plot.

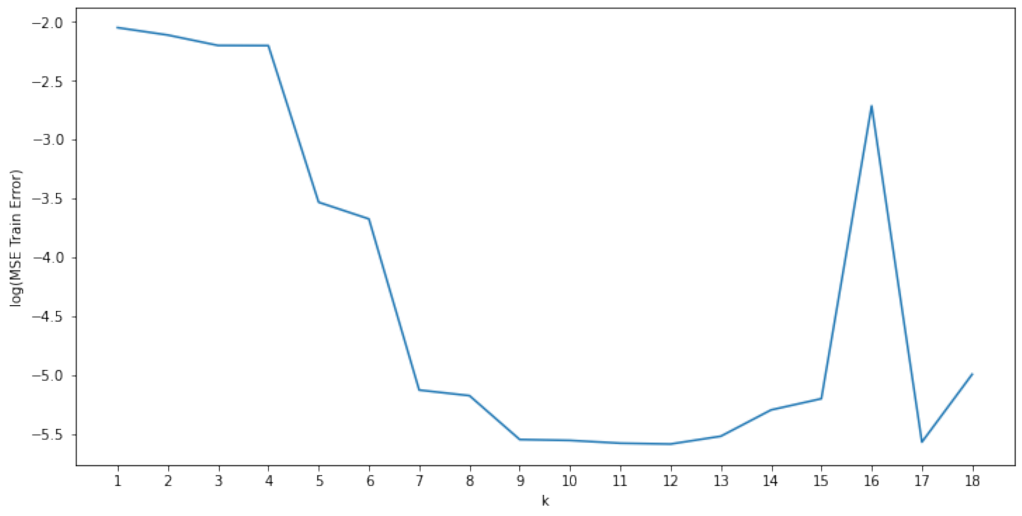

Figure 4: Ln(Train MSE) vs. Polynomial degree, k.

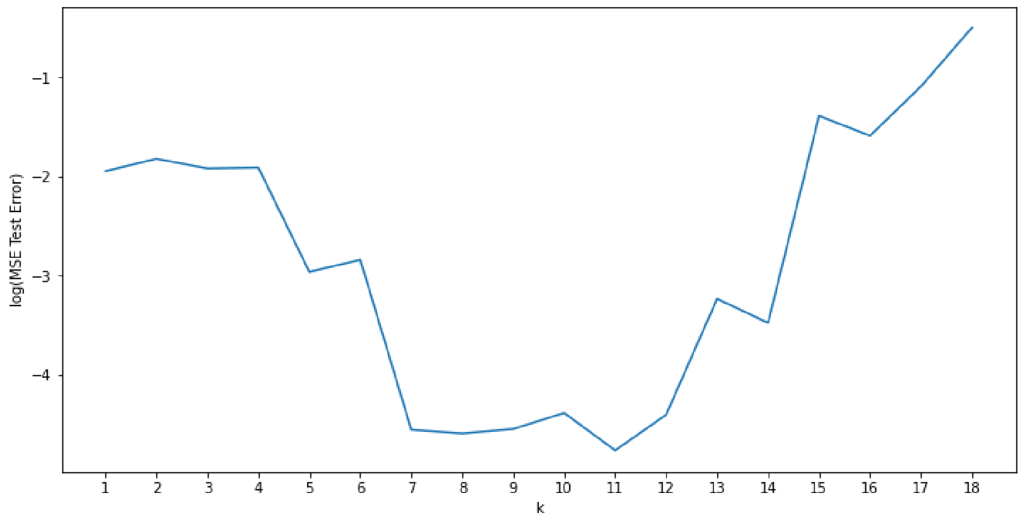

Figure 5: Ln(Test MSE) vs. Polynomial degree, k.

The training error is generally a decreasing function with increasing k. This is because higher order polynomials have more degrees of freedom available to fit the variations introduced by noise along the base sinusoidal. Since the training error is calculated to the training set, it is intuitive that a higher order fit will introduce a high variation on our prediction as our model tries to fit the training points closely. This is why the train error is decreasing.

The model underfits smaller k as there are simply not enough degrees of freedom to model our data. Also we see a large bias by our model. Mid-range k brings us to the optimal fits. For our model, the ideal polynomial degree constantly happens between 8 to 11, and higher degree models tend to overfit the training points thus resulting in worst performance.

With such a high variance introduced by the model overfitting to the training set, we see that our testing error is an increasing function with increasing k after the optimal range. With the newly generated test points being different from the training set and the model essentially fitting the noise instead of the underlying sinusoidal, we expect this higher test error at higher polynomial orders.

To consolidate our findings, we repeat the experiment over 100 run cycles and average the training and testing errors. The plots for the log (Ln) of the average train and test errors can be found in Figures 6 and 7, respectively. We see once again the same overall trend, confirming our results.

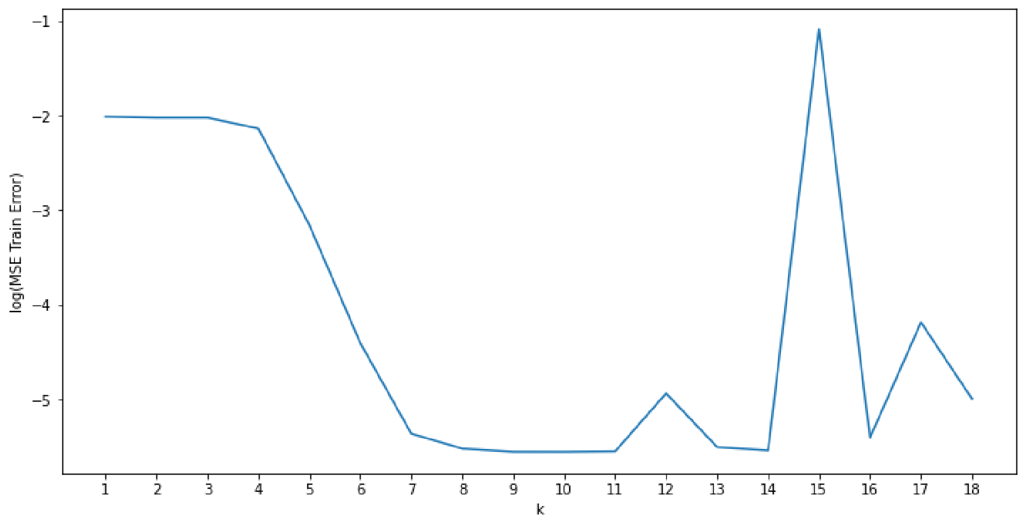

Figure 6: Ln(Avg Train MSE over 100 runs) vs. Polynomial degree, k.

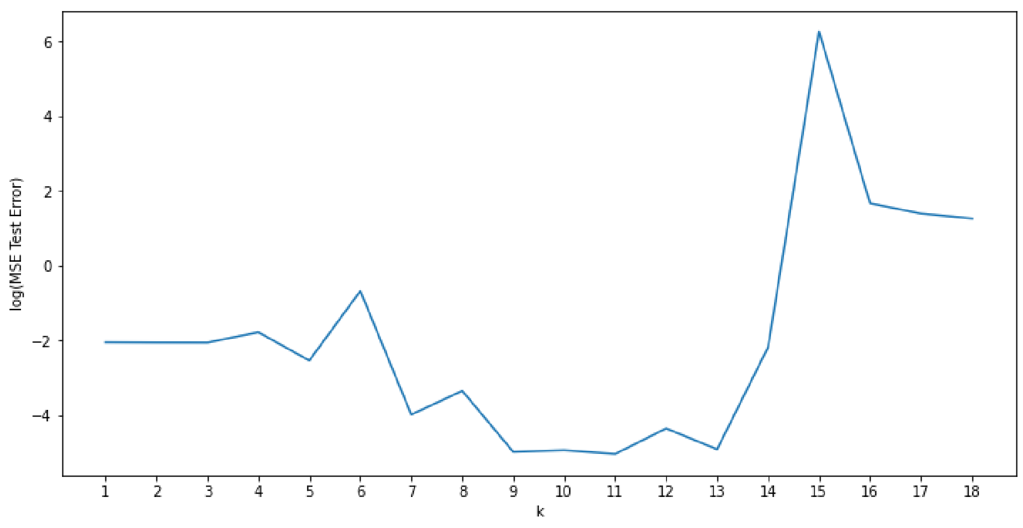

Figure 7: Ln(Avg Test MSE over 100 runs) vs. Polynomial degree, k.

2.1.3 Question 3: Fitting a sinusoidal basis

Replacing our polynomial basis function with a sinusoidal basis function such that we transform our data using $\phi(x) = sin(k\pi x)$. We repeat the experiments in Question 2 and evaluate how it affects our model fitting to our data.

Once again, we generate a training set of 30 samples and a testing set of 1000 samples using the same method outlined in Question 2. For consistency, we plot the sinusoidal basis regression fits for the set $k = \{2, 5, 10, 14, 18\}$ on the training set as Figure 8.

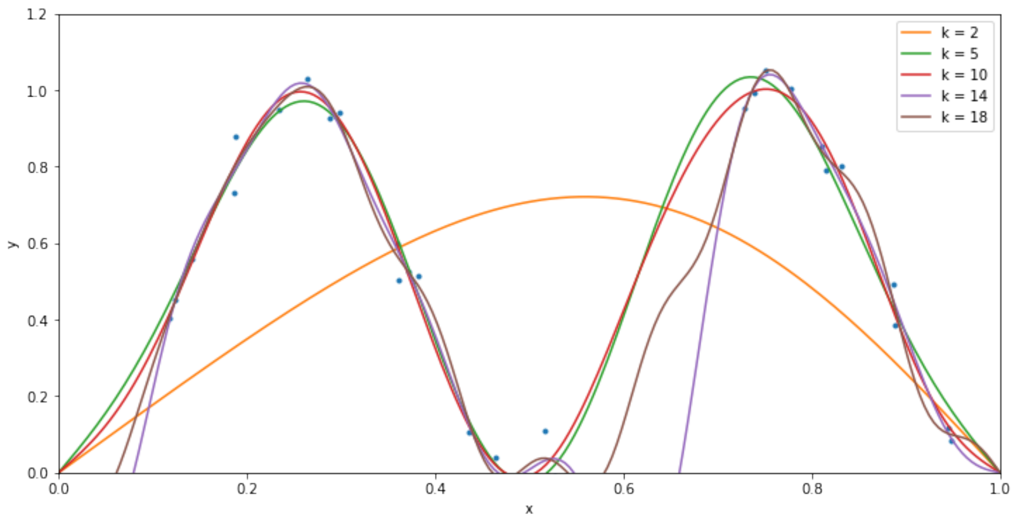

Figure 8: Each sinusoidal degree fit on our dataset, S.

Essentially, what we are achieving here for larger order k is a more and more complete Fourier series expansion on our training points, albeit not exact. From Fourier theory we know that any signal can be modelled by a sum of sinusoids with increasing accuracy as the number of sinusoids approaches ∞. A larger k is going to have more sinusoids mixed and these are weighted in such a way that the model will try to fit almost all the training points, which as we learned, will give us a large test error for increasing k. The MSE for the training and testing sets can be seen in Figures 9 and 10, respectively.

A smaller k leads to the model underfitting in the same way with the polynomial basis functions. There are not enough degrees of freedom to model the data, leading to high bias by the model as it ignores the training data. We observe a good fit to the data for degrees 5 to 13 as the test error is minimised around this range. But once again, for higher order k there is overfitting of the model due to the aforementioned reasons which give this higher test error.

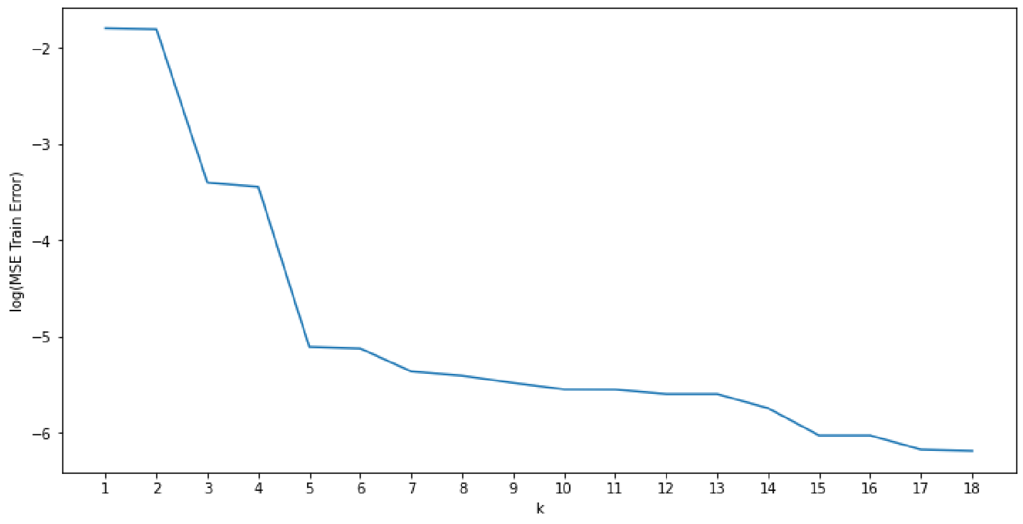

Figure 9: Ln(Train MSE) vs. Sinusoidal degree, k.

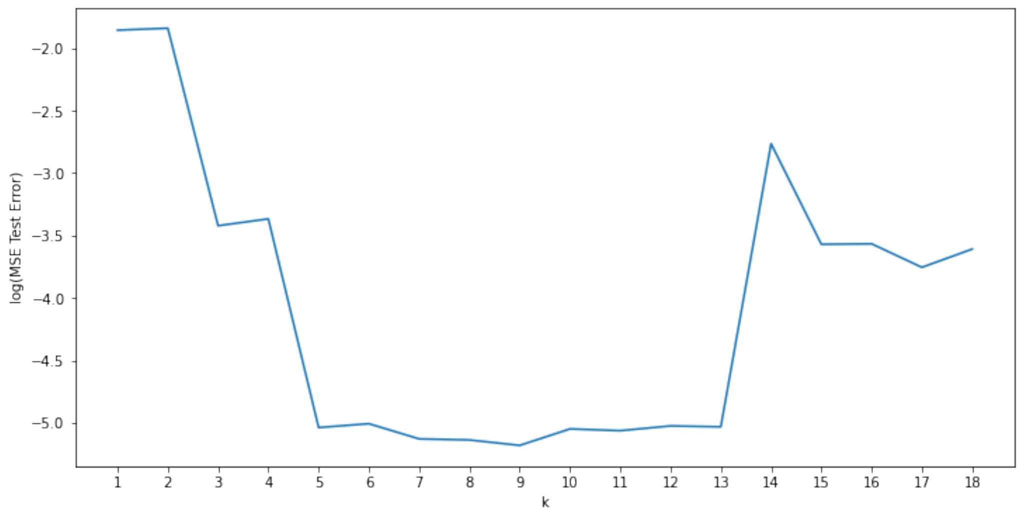

Figure 10: Ln(Test MSE) vs. Sinusoidal degree, k.

To confirm our results are consistent, we perform the experiments over 100 runs of randomly generated training and test points and average our errors. Figures 11 and 12 match our single run train and test errors, confirming our consistent results.

A notable result is that the average of over 100 runs is much more consistent using the sinusoid basis functions than the polynomial ones, even for newly randomly generated train and test points. This is probably due to the robust nature of the Fourier theory. Higher order sums of sinusoids approach this perfect representation of the data no matter how noisy the training data is. This could also be because the underlying train and test set are generated from a sinusoid themselves, so fitting a sinusoid basis is more correlative than a polynomial basis. The minimum test error is also smaller for the sinusoid basis in comparison to the polynomial basis, so for this application, the sinusoidal basis is better to model the data. However, the overarching result seems to be that for any basis function we experience a bias-variance trade-off concerning the order of dimensionality of the basis functions, k.

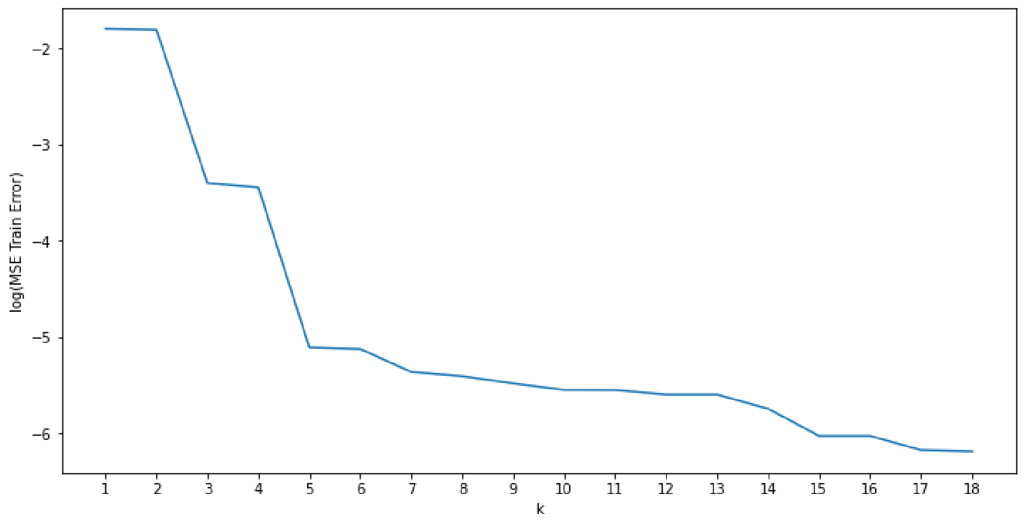

Figure 11: Ln(Avg Train MSE over 100 runs) vs. Sinusoidal degree, k.

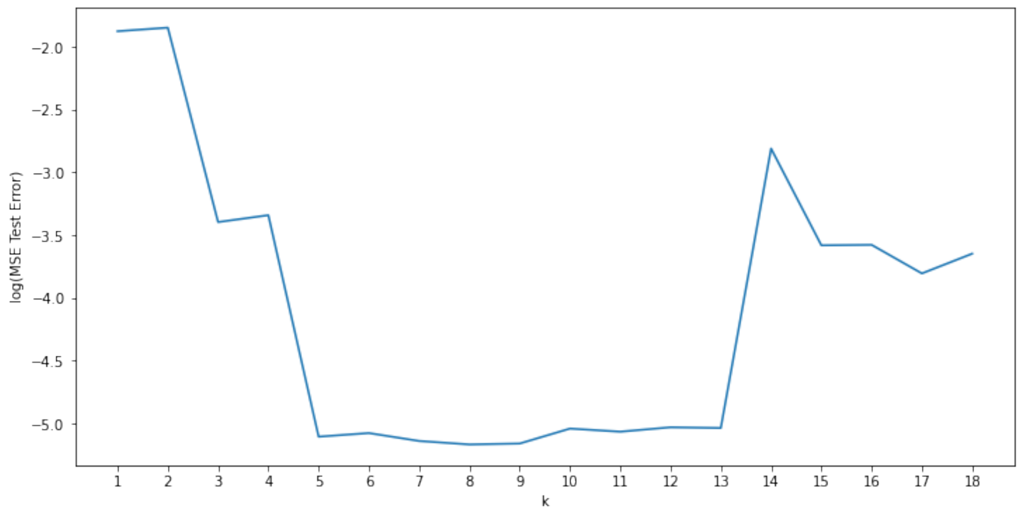

Figure 12: Ln(Avg Test MSE over 100 runs) vs. Sinusoidal degree, k.

2.2 Filtered Boston Housing Linear and Kernel Regression

2.2.1 Question 4: Baseline versus full linear regression

In this experiment, we compare the performance of a regression model on varying dimensions of the training sample. We use the Boston Housing dataset to predict the house price for a house given our inputs based on different attributes.

Part A and B: Naive regression
In part A we perform naive regression. That is we predict the mean y-value of our training set every time. This is equivalent to fitting the data with a constant function, as we only learn one weight equivalent to the mean y-value.

This works because we fit a vector of ones of the same size as the train set. Mathematically,

$$w = \left[X^T X\right]^{-1} X^T Y$$

and since we are using a vector of ones with length equal to the size of our training set then $\left[X^T X\right]^{-1}$ is simply the reciprocal of the training set length. $X^T Y$ will give the sum of all y's and so our weight will simply be the mean y-value.

Performing linear regression on the train and test sets and then calculating the MSE for both, we obtain results that can be seen in Table 2 for Naive Regression. The MSE is very high which is understandable as we consistently predict the mean y-value for all inputs into the model. The model is therefore very basic and 'naive' with no 'learning' on the training features at all. It is a constant predictor based on the median house pricing and not the factors that contribute to it. A side note is that the train set MSE calculates the variance of the y-values for the set.

Part C: Single attribute regression
We now move on to performing linear regression with one attribute at a time, synonymous with taking one dimension slice from each of our input data points and fitting this to the outputs. We append 1's to each data point so that we learn a bias term within the model, such that the second weight in the weight matrix corresponds to this bias (y-intercept).

We expect our MSE for both the train and test sets to be lower than naive regression as we are using attributes from the dataset to predict the output (median housing price) and not just blindly ignoring the inputs and predicting a constant. Some attributes may play a more important role in predicting a more accurate output and so we also expect a varying MSE across the attributes. Once again, Table 2 shows the MSE for each attribute as we perform regression.

Initially, some attributes seem to have much lower errors, implying that they are the most relevant attributes. However, further inspection of the results overthrows this assumption. We perform the regression several times and both the training errors and the testing errors fluctuate a lot which suggests that attributes may have different levels of importance but can not be inspected by doing single attribute regression.

Part D: Full attribute regression
In this final part, we perform regression with all the attributes at the same time. This will produce a more accurate model as a single attribute might not tell the whole story. Also, some attributes could be correlated to each other implicitly which leads to a lower error when putting them together in the training dataset. From Table 2 we can see that not only did the training error and testing error drop dramatically, the standard deviation of both errors also perform better than all the other previous methods.

2.2.2 Question 5: Kernelised Ridge Regression

Part A, B, and C: 5-fold CV to identify the best γ, σ parameters
In this question, we perform full ridge regression on our Boston Housing dataset through kernel regression. In this experiment we also utilise the dual-form of our non-linear regression expressions. Via Representer's Theorem, we may incorporate our kernels into our solution, allowing us to perform regression in higher-order dimensions without explicitly performing a non-linear transformation on our inputs, thereby reducing time complexity.

The kernel of choice is a Gaussian kernel. This kernel applies a radial basis function to our data points and is given in Equation 3. This kernel is equivalent to computing infinite length transformation vectors on the inputs where each entry of the transformation vector corresponds to evaluating a radial basis function for all points within the space of $\vec{x}$.

$$k\left[\vec{x}_i, \vec{x}_j\right] = exp\left[-\frac{\left(\vec{x}_i - \vec{x}_j\right)^T\left(\vec{x}_i - \vec{x}_j\right)}{2\sigma^2}\right] \tag{3}$$

We split the Boston Housing dataset again into a '2/3 Train, 1/3 Test' split. However we need to identify which σ, the deviation of our Gaussians, and which γ, the strength of regularisation, yield us with the best model.

We create a set of possible σ and γ and use 5-fold CV in order to compute which σ, γ pair yields the lowest loss where $\gamma = [2^{-40}, 2^{-39}, ..., 2^{-26}]$ and $\sigma = [2^7, 2^{7.5}, ..., 2^{13}]$. In a 5-fold CV we split out the training set into 5 parts of equal length and iteratively hold out one fold for validation while training a model on the remaining 4 folds using the passed σ, γ pair. The validation error is defined as the MSE between the 4 folds of training data and the validation fold. The MSE is averaged across each iteration and this gives the cross-validation error for that σ, γ pair. We perform this for every possible combination of σ and γ to obtain the optimal pair with the lowest validation error. This pair is then used to train our full ridge regression model.

Performing this 5-fold CV for one run, we obtain an optimal σ, γ pair of

$$\gamma = 9.3e - 10$$

$$\sigma = 2048$$

Using these optimal parameters to perform full ridge regression on the Boston Housing dataset we obtain an MSE Train error of 0.47 and an MSE Test error of 8.23! This is very good in comparison to even the full-attribute linear regression in Section 2.2.1 which only achieved an average MSE Train and Test error of 22.52 and 23.33, respectively.

To understand how the 5-fold CV identified the best σ, γ pair, we can plot how the cross-validation error changed with the different pairings (see Figure 13). The optimal parameters identified for this run are highlighted by the red 'X'. We plot the log of the pairing values against the log of the CV error as they vary by many orders of magnitude.

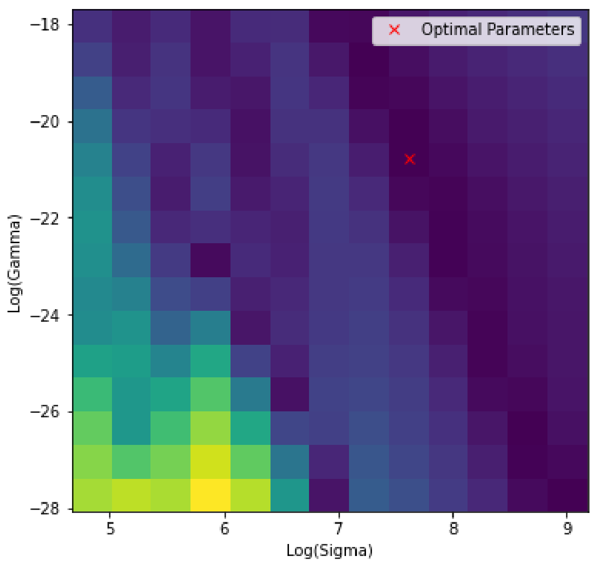

Figure 13: Cross-validation error for each γ, σ pair.

The plot in Figure 13 highlights low areas of CV error as darker and hence we find most minimas along a sort of ridge. Pairings along the bottom left of the plot seem to give very high CV error meaning the parameters provided either too little regularisation or the width of our radial Gaussians were too great to accurately model our data.

2.2.3 Averaging our results over 20 random data splits

To consolidate our findings and give a comparative overview on how the different methods perform on the Boston Housing dataset we run each experiment from Section 2.2 20 times with 20 random 2/3 1/3 splits of the Housing dataset to obtain random train and test sets each cycle. For each cycle and regression method, we calculate the MSE on both the train and test sets to quantify the performance on the data. After that, we calculate the standard deviation of the means to understand the consistency of the result.

This also meant for Section 2.2.2 that we obtained 20 best (γ, σ) pairs for each run. The average best γ, or strength of regularisation, was found to be $2.95e - 9$ and the average best σ was found 1911.67.

We can see that the train and test set errors generally decrease as each method becomes more complex. Single attribute regression beats naive regression, and full-attribute regression outperforms single attribute regression. Fundamentally, the more input features of our data are used to predict the output, the more informed our prediction can be. Kernel ridge regression performs the best and scores a very low train error of 1.04 and this is consistent across all cycles due to a low standard deviation of only 0.77. The test error is the lowest for all methods too, but we note that the standard deviation is much larger than the test error itself, which was unexpected*. A deeper dive into our test errors for the 20 best pairings, we noticed outlier cycles that yielded very high test errors of up to 98. The majority of test errors were considerably lower at less than 20, so we consider these anomalies.

Statistically speaking, this is very unlikely. And to prove we should ignore these anomalies, we perform normal analysis. We have a high standard and hope to keep the most test errors possible, so 99.7% confidence-interval is used. Taking 3 standard deviations away from the mean of 17.52 gives us 86.61, and any error beyond that can be considered 'very unlikely' to happen. Eliminating errors beyond this range and recalculating our MSE test error gives us

$$13.26 \pm 13.97$$

meaning that the outlier is truly on its own. Therefore we can state that ridge regression is indeed the best method for modelling our Boston Housing dataset.

Table 2: MSE for train and test sets for all methods averaged over 20 cycles with random sampling.

Method	MSE train	MSE test
Naive Regression	83.23±3.75	87.03±7.6
Linear Regression (attribute 1)	61.37±11.44	76.78±10.77
Linear Regression (attribute 2)	65.14±9.4	68.75±11.18
Linear Regression (attribute 3)	64.1±5.27	70.77±11.76
Linear Regression (attribute 4)	72.69±9.79	54.1±30.98
Linear Regression (attribute 5)	58.08±11.78	83.17±16.69
Linear Regression (attribute 6)	65.95±21.7	67.43±26.32
Linear Regression (attribute 7)	67.23±7.78	64.39±15.01
Linear Regression (attribute 8)	66.62±13.41	65.95±18.47
Linear Regression (attribute 9)	63.25±10.29	73.29±10.17
Linear Regression (attribute 10)	66.77±5.25	65.54±10.13
Linear Regression (attribute 11)	63.56±3.73	71.76±11.77
Linear Regression (attribute 12)	68.16±29.92	63.04±24.43
Linear Regression (all attributes)	22.52±1.36	23.33±3.11
Kernel Ridge Regression	1.04±0.77	17.52±23.03 (see * above)

3 Part 2

3.1 k-Nearest Neighbours

3.1.1 Question 6: Visualisation of one sampled hypothesis

In question 6 the goal is to generate a voted-centre hypothesis, $h_{S,v}$, using a random process while $|S| = 100$ and $v = 3$. First, we create 100 centres, x_i, uniformly at random from $[0,1]^2$. Next, we generate 100 labels uniformly at random from $\{0,1\}$ and assign them to the 100 centres created. For every $h_{S,v}(x)$, the corresponding y value will be determined by a majority of the 3 closest x_i's y values. To plot this hypothesis into a figure, we created an evenly spaced mesh grid with 999×999 crossing points between $[0,1]^2$ and calculated all the y values of the crossing points. Finally, we plot the mesh grid and the 100 centres onto the same figure and colour labelled them as red $= 0$ and blue $= 1$.

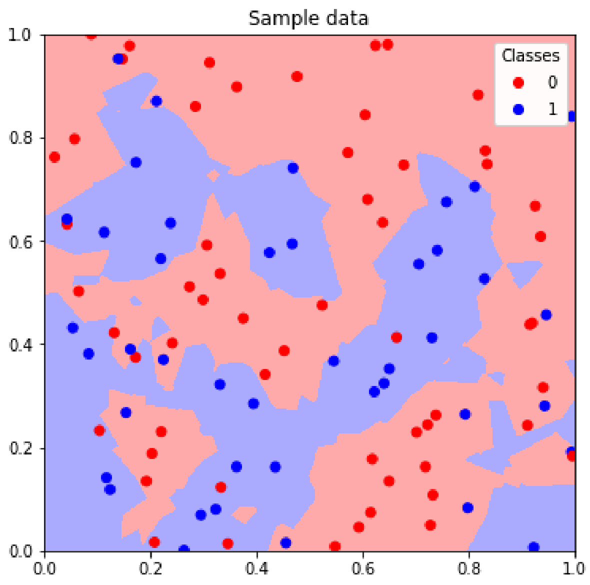

Figure 14: A hypothesis $h_{S,v}$ with $|S| = 100$ and $v = 3$

3.1.2 Question 7: Estimated generalisation error of k-NN as a function of k

In question 6 we obtain a method to reproduce voted-centre hypothesises. Now we can generate training points and testing points with reasonable distributions based on the hypothesises. In this part, we will like to investigate the performance of k -NN with different k values. Using the same parameters, $|S| = 100$ and $v = 3$ for $h_{S,v}(x)$, we sample 4000 training points and 1000 testing points uniformly at random from $[0,1]^2$ and let $y = h_{S,v}(x)$. However, we would like to add noise to our dataset. For a 20% chance, we re-sample the y values uniformly at random from $\{0,1\}$. Therefore, roughly 10% of our data will be miss-labelled as noise while the other 10% will remain correct.

For each k in the set of $\{1,\ldots,49\}$, we will do the following for 100 runs. Build a k-NN model according to different k using the 4000 training points, then use that k-NN model to predict the y of the 1000 testing points. Comparing the prediction and the actual y of the 1000 testing points will give us the error rate. Take the mean error rate across the 100 runs and we can get the estimated generalization error. Visualize the estimated generalization error of all the k values to clearly observe the effect of k in k-NN.

Before the visualization is generated, we can make some assumptions. Since we know that 1-NN error rate will be maximum twice the Bayes error rate, and we analysed that there are roughly 10% noise in our dataset, the estimated generalization errors should cap at approximately 20% at 1-NN. Also, larger k might take too many neighbour points that are actually irrelevant into account, so the error rate should start to increase. Overall, the estimated generalization errors should start high, decline to an optimal value, and gradually increase.

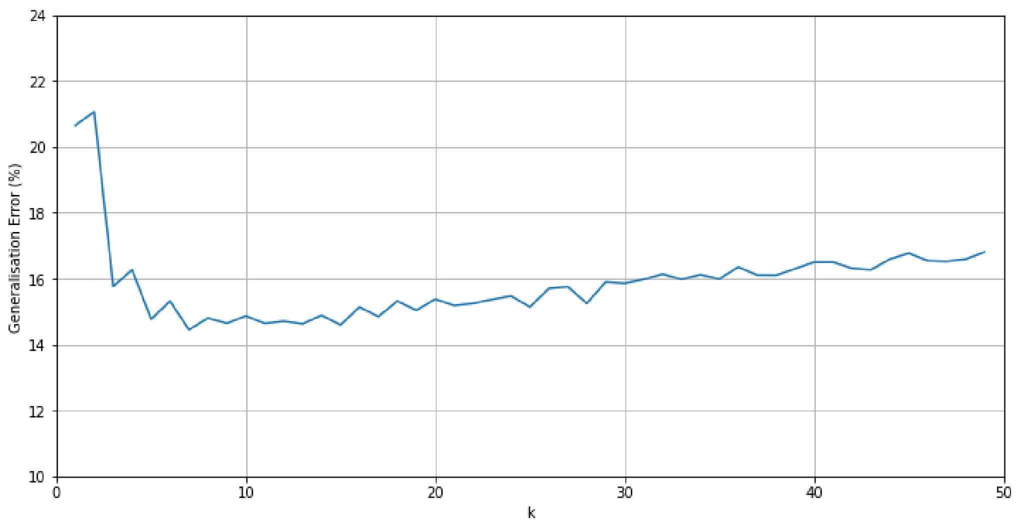

Figure 15: estimated generalization errors vs. k-NN degree, k.

As we expected, at 1-NN the algorithm is susceptible to over-fitting due to the noise of the data as we have a low bias (low training error due to 1-NN) but high variance. It is noteworthy that the peak error always happens at k=2, and this is logical as the nature of 2-NN is generating 2 different sets of 1-NN and randomly choosing the prediction. We run this experiment several times and the optimal k mostly happen between 10 - 20 with minimum error rates of roughly 14%. This is because as k increases, the model gains complexity and start to produce more reasonable 0-1 boundaries. However, even though larger k attempts to eliminate the noise in the training dataset, there are still 10% noise in the testing dataset. Therefore, the ideal minimum error rate will be 10% in theory, and 14% error rate in practice seems acceptable. Larger k above this range as predicted, start to consider points with little correlation to the test points, which increases the error rate.

3.1.3 Question 8: Determining the optimal k as a function of the size of the training set (m)

In question 8 we will examine whether the optimal k is related to the size of the training dataset. For each number of training points, m, in the set of $\{100, 500, 1000 \ldots 3500, 4000\}$, we will do the following for 100 runs. Similar to question 7, we have a set of $k \in \{1, \ldots, 49\}$, and we train the k-NN model with 4000 data points and test the model with 1000 testing points. The model and dataset generating processes are identical to question 7. Yet, this time we will be saving the optimal k which leads to the minimum error rate. Averaging out all the optimal k of the 100 runs will give us the estimated optimal k for this number of training points m.

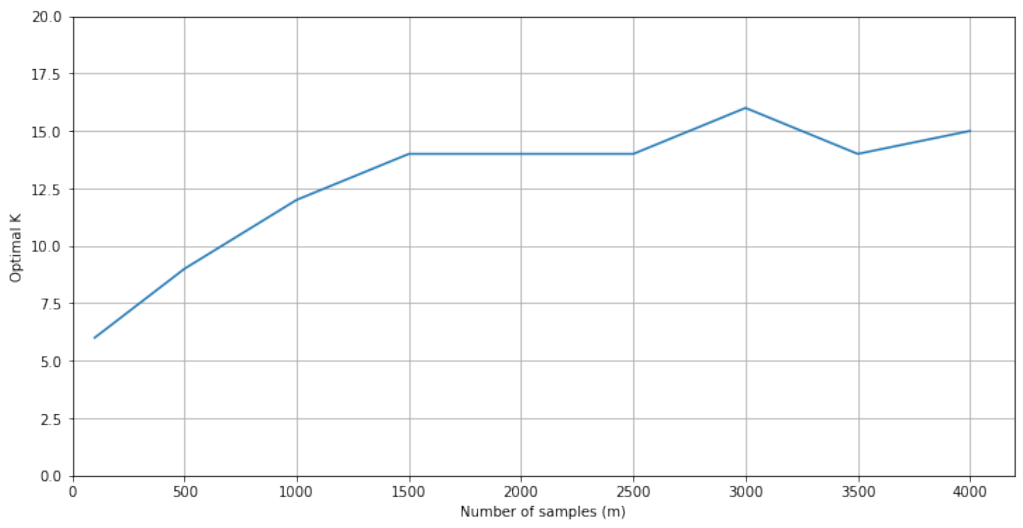

Figure 16: Optimal k vs. number of samples, m.

Keeping in mind the relative bias-variance trade-off we observed in the K vs error graph previously, we see that same relationship affecting the model here with respect to the training sample size. At the top end of the graph we see an optimal k of around 15 for 4000 training samples, this is what we observed in the Figure 16. As the training size decreases, the optimal k must also decrease as the range of where the minima of the generalisation error is found decreases too. The larger k error observed in the previous fig occurs much earlier due to this relative scaling and so the optimal k will decrease with decreasing training size. The point of low variance occurs much earlier for smaller training datasets as intuitively the same optimal k for the 4000 sampled training set would be too great for say the training set of size 100 since we would be taking into account neighbours with low correlation once again.

4 Part 3

4.1 Question 9: Dot-Product Kernel with a Constant

4.1.1 Part A: Proving K_c is PSD

To prove that $K_c(\vec{x}, \vec{z})$ is PSD we can take a closer look at the formulation of the kernel function. We note that the function is simply computing the inner-product and adding a constant c to the output, this is shown in Equation 4 and is equivocal to taking a dot-product kernel, denoted $K^{'}$, and adding a constant.

$$K_c(\vec{x}, \vec{z}) = c + \sum_{i=1}^{n} x_i z_i = c + \vec{x}^T \vec{z} = c + \langle \vec{x}, \vec{z} \rangle$$

$$= c + K^{'}(\vec{x}, \vec{z}) \tag{4}$$

For K_c to be PSD then $K^{'}$ itself must be PSD. Taking the Gram matrix $\mathbf{K}$ over $K^{'}$ such that

$$\mathbf{K} = \begin{bmatrix} & \vdots & \\ \cdots & K^{'}(\vec{x}_i, \vec{x}_j) & \cdots \\ & \vdots & \end{bmatrix}$$

then $\forall c \neq 0$, $\mathbf{c^T K c}$ must be positive. In other words this is equal to

$$= \sum_{n}^{i,j} c_i c_j K^{'}(\vec{x}_i, \vec{x}_j) = \sum_{n}^{i,j} c_i c_j \langle \vec{x}_i, \vec{x}_j \rangle$$

$$= \left\langle \sum_{i}^{n} c_i \vec{x}_i, \sum_{j}^{n} c_j \vec{x}_j \right\rangle$$

where $\langle$ ...$\rangle$ is in the bilinear form and using our norm identity, this must be greater than or equal to 0 for PSDness and this holds due to the square term implying it is always positive.

$$= \left\| \sum_{i}^{n} c_i \vec{x}_i \right\|^2 \geq 0$$

Going back and adding our constant, for K_c to be PSD then

$$\sum_{n}^{i,j} c_i c_j \left(K^{'}(\vec{x}_i, \vec{x}_j) + c \right) = \left\| \sum_{i}^{n} c_i \vec{x}_i \right\|^2 + c \left(\sum_{i}^{n} c_i \right)^2 \geq 0$$

which is only satisfied when $\mathbf{c} \geq \mathbf{0}$ since the first term may well be equal to 0. Of course we may just bound it to be when $\mathbf{c} > \mathbf{0}$, since if $c = 0$ it defeats the purpose of even adding c to the dot-product kernel.

4.1.2 Part B: The effect of c when using K_c in linear regression

The impact of adding a constant c to the kernel function with linear regression is equivalent to appending a $\sqrt{c}$ in front of the vectors $\mathbf{x}$ and $\mathbf{z}$. Take a simplified feature map from the lecture slides as an example,

$$\begin{aligned} \langle \phi(\mathbf{x}), \phi(\mathbf{z}) \rangle &= (x_1, x_2, ..., x_n)(z_1, z_2, ..., z_n)^T \\ &= (x_1 z_1, x_2 z_2, ..., x_n z_n) \\ &= \mathbf{x}^T \mathbf{z} \end{aligned}$$

We can see that adding c to the front of vectors $\mathbf{x}$ and $\mathbf{z}$ shows that,

$$\begin{aligned} \langle \phi(\mathbf{x}'), \phi(\mathbf{z}') \rangle &= \left(\sqrt{c}, x_1, x_2, ..., x_n\right)\left(\sqrt{c}, z_1, z_2, ..., z_n\right)^T \\ &= (c, x_1 z_1, x_2 z_2, ..., x_n z_n) \\ &= c + \mathbf{x}^T \mathbf{z} \end{aligned}$$

Since we know that we are simply appending a constant in front of the vectors, we can now analyse how the constant affects the result. Adding a constant mainly affects higher orders of the polynomial expansion of the kernel, as the constant might not appear too often in higher order multiplications. Thus, adding a constant to the kernel will gain more weights to the lower order terms. However, we are solving a linear regression. The order of our kernel is very low, which makes the constant useless. In other words, the constant adds weights to all terms, which is equivalent to doing nothing.

4.2 Question 10: Simulating 1-NN Through Gaussian Kernel Regression

In this section we evaluate the possibility of approximating a 1-NN algorithm through Gaussian kernel regression. Kernel linear regression can be thought of as a local linear regression through a Gaussian neighbourhood sampling. We have a dataset, S, of values labelled with either -1 or 1 such that $(\vec{x_m}, y_m) \in \mathbb{R}^n \times \{-1, 1\}$. Through dual-form kernel regression, as seen before, our output is given by $sign(f(\vec{t})) = sign\left(\sum_{i=1}^{m} \alpha_i K(\vec{x_i}, \vec{t})\right)$.

When we begin to think of our regression model in the comparative sense to a locally weighted regression (LWR) we draw on standard results as to how we can go about achieving 1-NN simulation. Firstly, understanding how the Gaussian kernel is acting in methods like LWR is that the predicted value y^* is a weighted average across all labels in our dataset, divided by the total of the weights:

$$y^* = \frac{\sum_{i=1}^{N} c_{it} y_i}{\sum_{i=1}^{N} c_{it}} = \frac{\sum_{i=1}^{N} K_\beta(\vec{x_i}, \vec{t}) y_i}{\sum_{i=1}^{N} K_\beta(\vec{x_i}, \vec{t})}$$

To compute the local weighted average, we would intuitively want higher weights on points closer to our test point, and smaller weights elsewise. How can we achieve this? As seen above, through a kernel function! Thinking of the Gaussian kernel (Equation 3), it produces a larger output when the two input vectors are of close euclidean distance. This means we could use such a kernel function in our LWR to control how neighbouring points influence our final predicted value, already sounding similar to k-NN.

We are given a Gaussian kernel of the form

$$K_\beta(\vec{x}, \vec{t}) = exp(-\beta \left\|\vec{x} - \vec{t}\right\|^2)$$

and so wish to identify which β can give us this 1-NN approximation of our ridge regression. From [1] and [2] we see that the β parameter is essentially controlling the window or 'bandwidth' of the locally evaluated neighbourhood around our points. Most notably from [2], we can use a kernel function with an adaptive β that is calculated based upon the test point to control this window and it should take the form of $\beta = \hat{\beta}(\vec{x_1}, ..., \vec{x_m}, \vec{t})$, such that we assign our neighbourhood window, the points we take the weighted average over, to include only the nearest neighbouring point, in effect simulating a 1-NN.

From [2] we see the window of the Gaussian kernel function of form $exp\left\{-\left(\frac{\vec{x}-\vec{t}}{h}\right)^2\right\}$, where h is the window size. Comparing this to Equation 3 we see that h is just σ which we know controls the width of our Gaussian functions. Now, setting our window size, h, bounded to the distance of our nearest neighbour we obtain:

$$h(t) = \underset{i \to m}{\mathbf{min}} \{|x_i - t|\}$$

That is, the minimum distance when comparing all training points to our test point is equal to the nearest neighbour distance. To ensure we ignore all points beyond our nearest neighbour, the Gaussian constructed on each test point should, strictly, only span up to our nearest neighbouring point. We know the width of a Gaussian is bounded around 3σ either side. This means that our $h(t) = 3\sigma$, to ensure this strict 1-NN local computation (we could go further and strictly take 4σ but 99.7% of values in the Gaussian lie within the 3σ range and so suffices). This means

$$\underset{i \to m}{\mathbf{min}} \{|x_i - t|\} = 3\sigma$$

$$\therefore \frac{1}{3} \cdot \underset{i \to m}{\mathbf{min}} \{|x_i - t|\} = \sigma$$

Plugging this back into our Gaussian kernel equation we get

$$K_\beta = exp\left[-\frac{(x_i - t)^2}{\frac{1}{9}\left(\underset{i \to m}{\mathbf{min}} \{|x_i - t|\}\right)^2}\right]$$

giving us,

$$\beta = \frac{1}{\frac{1}{9}\left(\underset{i \to m}{\mathbf{min}} \{|x_i - t|\}\right)^2}$$

Such that β is a function of the test point and all training points. Notice how the window of the Gaussian is adaptive dependent on the test point and whichever training point is closest, for example, a test point far away from its nearest neighbour forces the Gaussian bandwidth to increase until it includes that nearest neighbour in the weighted output. For a visualisation we perform this linear regression using the above β kernel as seen in Figure 17. Here we sampled randomly at uniform points between $[0, 1]$ and assigned uniform labels at random $-1, 1$ (in a similar way to the k-NN task previous) to test our predictor on. The code for this is found in Notebook 'coursework_Q10.ipynb'.

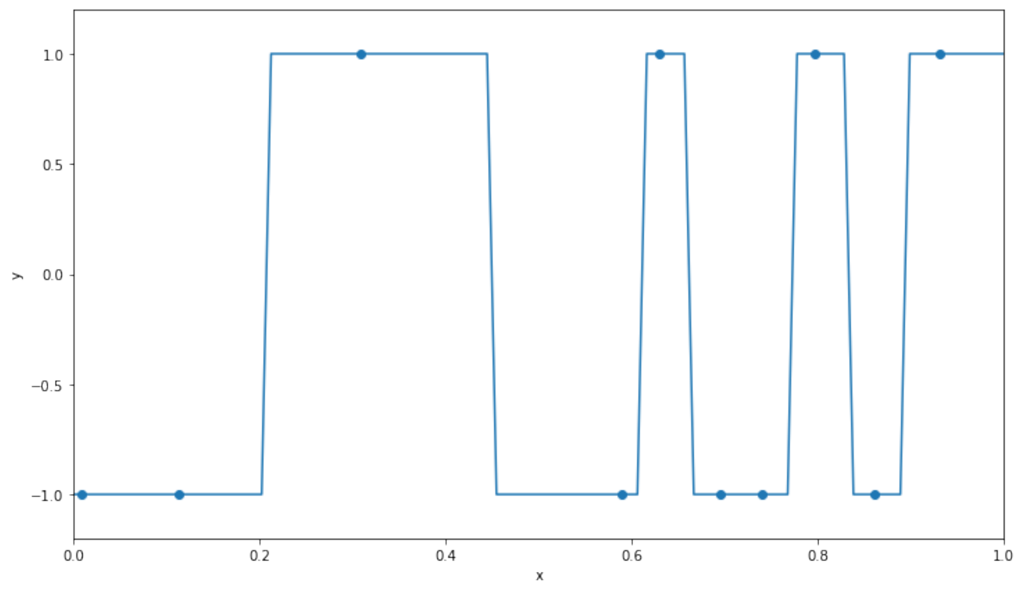

Figure 17: Input x vs prediction Sign(f(t)).

The plot shows that our model is approximating a 1-NN regression on $x \in \mathbb{R}$, single dimensional x points. (Please feel free to run the notebook multiple times to confirm results using this β). We also tested the kernel with windowing 4σ's and above but 3 works just fine to capture the nearest neighbour.

4.3 Question 11: Whack-A-Mole

We understand that the implementation of the algorithm is not needed, but we still write out the code to make sure that the algorithm works. The source code for the algorithm developed can be found as 'corsework_q11.ipynb'.

For a Whack-A-Mole problem, it is reasonable to only hit a point once per game, as hitting the same point over than one time simply hides the moles and pops up the exact same moles again. The following proof will be based on a board with $n \times n$ holes.

First, we create a $n^2 \times n^2$ action matrix A while the columns represent all the n^2 holes, and the rows represent the affected holes if we hit the corresponding column hole. Take the board in the question as example, if we hit position 10, all the holes surrounding position 10 will be affected, so position $\{(a_{6,10}), (a_{9,10}), (a_{10,10}), (a_{11,10}), (a_{14,10})\}$ will be marked as "1" and the rest of column 10 will be 0.

Next, we create a $n^2 \times 1$ holes_with_mole matrix M to store the positions of the moles. Now A has corresponding mole positions, and we will perform Gaussian elimination to get the hitting sequence of holes. Initially, the action matrix is a binary matrix, and adding different rows together will give us the result of hitting different holes. The addition results might be 0, 1, or 2, but recap that 1 means "mole change status, either rise or hide", and 2 simply means the mole changes its status twice which equals not changing at all, so we can rewrite all the results of 2 to 0.

High degree polynomial can be written as matrices and Gaussian elimination is a technique to solve such high degree polynomial. During Gaussian elimination, we aim to set only the diagonal of A to 1 and the rest to 0. This way the final output of hitting points will be in the proper sequence. Since we replace all the 2 after every step of Gaussian elimination, we will still get binary matrices for both A and M. The matrix M now stores the hitting points in the correct sequence. For example, if $\{m_2, m_4, m_{10}\}$ equals 1, then the hitting sequence $(2, 4, 10)$ will be the correct answer. We tested the algorithm with different combinations of board sizes and mole positions, and the results were satisfying.

References

[1] Agnieszka Gajewicz-Skretna, Supratik Kar, Magdalena Piotrowska, and Jerzy Leszczynski. The kernel-weighted local polynomial regression (kwlpr) approach: an efficient, novel tool for development of qsar/qsaar toxicity extrapolation models. *Journal of Cheminformatics*, 13, 02 2021. doi: 10.1186/s13321-021-00484-5.

[2] Douw Gert Brand Boshoff Joseph Awoamim Yacim. A comparison of bandwidth and kernel function selection in geographically weighted regression for house valuation. *International Journal of Technology*, 10(1):58–68, Jan 2019. ISSN 2087-2100. doi: https://doi.org/10.14716/ijtech.v10i1.975.