Machine Learning: Coursework I
CID: 02476922

Question 1

We have the data $\mathcal{D} = \{x_i, y_i\}_{i=1}^{200}$, with $n = 200$ observations, as shown in figure 1 below:

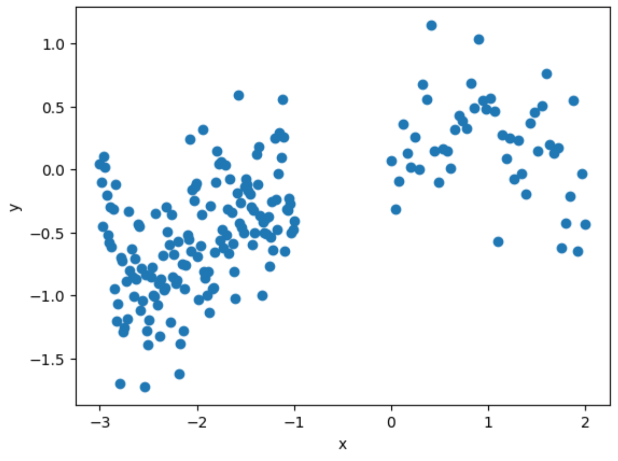


Figure 1: A plot showing the provided data, note the region between -1 and 0 where we lack any data points.

We shall consider the following model:

$$y_i = a\sin(\frac{2\pi}{0.3}x_i) + \sum_{k=1}^{p} b_k x_i^k + \epsilon_i$$

Where ϵ_i are independent random variables with $\mathbb{E}(\epsilon_i) = 0$.

We can rewrite the model as:

$$y_i = \sum_{k=0}^{p} \theta_k \phi_k(x_i) + \epsilon_i$$

Where $\phi(x_i) = (\sin(\frac{2\pi}{0.3}x_i), x_i, x_i^2, ..., x_i^p)$ is the $p+1$ vector of basis functions, and $\theta = (a, b_1, ..., b_p)^T$ is the vector of parameters that we want to infer. For ease of use later, we define $\Phi \in \mathbb{R}^{n\times(p+1)}$ to be the design matrix such that $\Phi_{ij} = \phi_j(x_i)$, that is, the j-th component of the basis vector for observation x_i.

We want to infer the parameter vector θ for the regression function:

$$f(x_i|\theta) = \sum_{k=0}^{p} \theta_k \phi_k(x_i)$$

that minimizes the corresponding loss function that we choose.

Part 1

Firstly, we perform ordinary least squares, for varying values of p, to show how this model can both overfit and underfit. We will use mean-squared-error (MSE) loss, that is, we seek the parameter vector θ that minimizes:

$$\mathcal{L}(\{(x_i, y_i)\}_{i=1}^n) = \frac{1}{n}\sum_{i=1}^n (y_i - f(x_i|\theta))^2 = \frac{1}{n}(\mathbf{y} - \Phi\theta)^T(\mathbf{y} - \Phi\theta)$$

Where we denote $\mathbf{y} = (y_1, ..., y_n)^T$.

Due to using MSE loss, θ has the analytic solution $\hat{\theta} = (\Phi^T\Phi)^{-1}\Phi^T\mathbf{Y}$, which we obtain by solving $\nabla_\theta\mathcal{L} = 0$.

To show both underfitting and overfitting, we split the dataset into a training set and a test set, using a 70% to 30% split. We then iterate over the maximum order, $p \in \{0, 1, ..., 30\}$, calculating $\hat{\theta}$ each time, along with the MSE on both the training set and the test set. The results are plotted below in figure 2

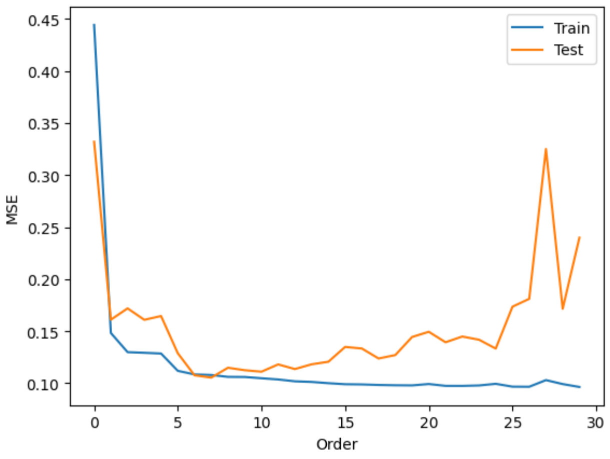


Figure 2: A plot showing how the training and test MSE change with the maximum order of the polynomial terms in the model.

We see that when for both the training and the test set, the MSE starts off relatively high at $p = 0$, before decreasing up until around $p = 10$ (noting that the test MSE is greater than the training MSE as expected). After $p = 10$, the training MSE continues to decrease as the maximum order increases, whilst the test MSE starts to increase. The initial reduction in MSE suggests the low-order models were underfitting the data, that is, they were unable to capture the relationship between x and y accurately enough. Then, since the training MSE continues to decrease past $p = 10$ while the test MSE increases, we observe that the model starts to overfit the data. It learns the training dataset too well, allowing it to achieve this lower MSE, but it then fails to generalise upon seeing the test set, resulting in a higher MSE.

Part 2

For the remainder of this question, we fix the maximum order $p = 10$.

Part 2a

We will compare ridge regression and OLS regression using a training set and a test set. Ridge regression includes a penalization term to stop the coefficients from becoming too large, resulting in the following loss function:

$$\mathcal{L}_{\text{ridge}}(\{(x_i, y_i)\}_{i=1}^n) = \frac{1}{n}\sum_{i=1}^n (y_i - f(x_i|\theta))^2 + \lambda\sum_{i=1}^p \theta_i^2 = \frac{1}{n}(\mathbf{y} - \Phi\theta)^T(\mathbf{y} - \Phi\theta) + \lambda\theta^T\theta$$

Like OLS, the solution is analytic, giving $\hat{\theta}_{\text{ridge}} = (\Phi^T\Phi + \lambda I_p)^{-1}\Phi^T\mathbf{Y}$. λ is the regularisation parameter that controls how much the model is penalized. As $\lambda \to 0$, we recover the OLS estimate $\hat{\theta}$, and as $\lambda \to \infty$, all of the parameters are shrunk to 0. To choose the optimal λ that minimizes the loss, we perform leave-one-out cross-validation (LOO-CV) on a grid of λ values, which we choose as $\lambda = 2^k, k \in \{-10, -9.9, .., 1\}$. The algorithm is then as follows:

Algorithm 1 LOO-CV

```
Given the data $\{(x_i, y_i)\}_{i=1}^n$, and a value of $\lambda$
for $i = 1, ..., n$ do
    1. Set $(x_{\text{test}}, y_{\text{test}}) = (x_i, y_i)$, and $(x_{\text{train}}, y_{\text{train}}) = \{(x_i, y_i)\}_{i=1}^n \backslash (x_i, y_i)$, meaning for the design matrix $\Phi$ we delete the $i$th row
    2. Fit a ridge regression model using the $n-1$ training observations
    3. Use this ridge model to predict the output for the one test point $x_{\text{test}}$
    4. Calculate the square error between this prediction and the actual value $y_{\text{test}}$
end for

Average the square error across all of the iterations. This is the MSE for LOO-CV.
```

We would then choose the value of λ that minimizes the MSE across the LOO-CV scheme. Denote this value λ_{ridge}. Note that we perform this on the whole dataset since we are interested in parameter estimation for this part of the question. We plot the LOO-CV test MSE below as a function of λ, using the grid specified earlier:

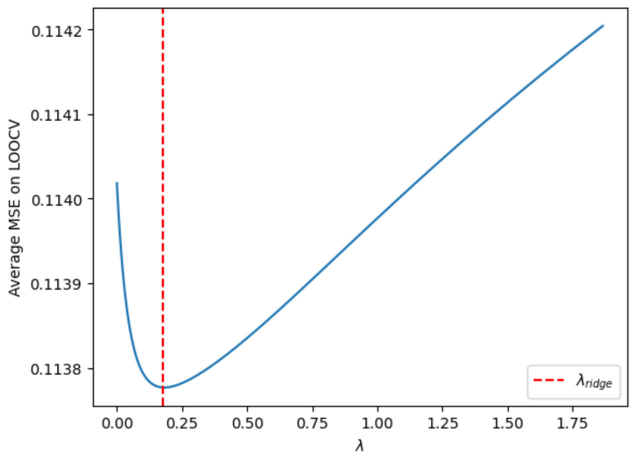


Figure 3: A plot showing the LOO-CV test MSE as the regularisation parameter λ varies. The red dashed line represents the value of λ that minimizes the MSE, being $\lambda_{\text{ridge}} = 0.1768$

Next, we split the data into a training set and a test set, we choose a 70% to 30% train-to-test split. We fit both OLS regression and ridge regression models on the training set and compare parameter estimates and MSE on both the training and test sets. Note for the ridge regression we choose $\lambda_{\text{ridge}} = 0.1768$ as found earlier. The results are plotted in table 1 below:

Parameter	a	b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8	b_9	b_{10}
$\hat{\theta}$	0.2192	1.0439	0.0418	-1.1382	-0.0218	0.5928	0.0257	-0.1481	-0.0191	0.0143	0.0031
$\hat{\theta}_{\text{ridge}}$	0.2127	0.6658	0.1839	-0.4467	-0.1362	0.1604	0.0287	-0.0348	-0.0062	0.0036	0.0009

Table 1: A comparison of OLS regression and ridge regression coefficients.

We see that the parameters are different, and note that for almost every parameter, ridge regression results in a lower absolute value, which makes sense, since it penalizes having higher absolute values. Moreover, in the case of this training and test set, OLS regression resulted in a training MSE of 0.0968 and a test MSE of 0.1219, whereas ridge regression resulted in a training MSE of 0.0975 and a test MSE of 0.1206. So OLS performed better on the training set, but ridge performed better on the test set.

Part 2b

Figure 4 below shows the data, split into the training and test set, along with the OLS regression function and the ridge regression function:

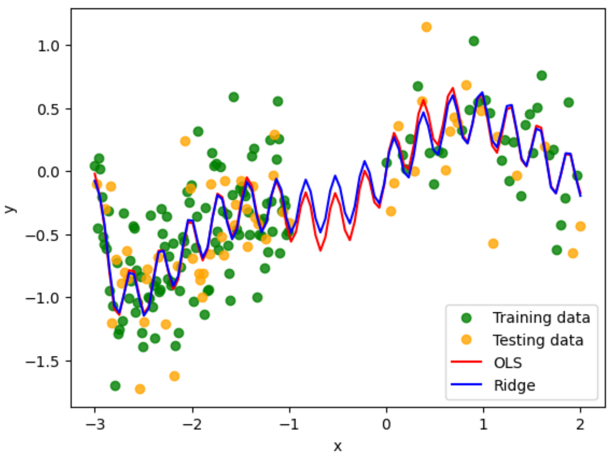


Figure 4: A plot showing the training and test data, along with the OLS regression line and ridge regression line

We see that ridge and OLS both give a similar fit, only really having any real difference towards the middle region, where there is no data. Realistically, both models are a decent fit, and either model would be feasible.

Part 2c

We now suppose that $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$, where $\sigma = 0.3$, and take a bayesian approach to the inference of θ, assigning a normal prior distribution with mean 0 and variance γ to each component which are independent of each other a-priori. Due to the normality and independence assumptions of the errors, we obtain the following likelihood function:

$$p(\mathbf{y}|\Phi, \theta, \sigma) = \mathcal{N}(\mathbf{y}; \Phi\theta, \sigma^2 I_n)$$

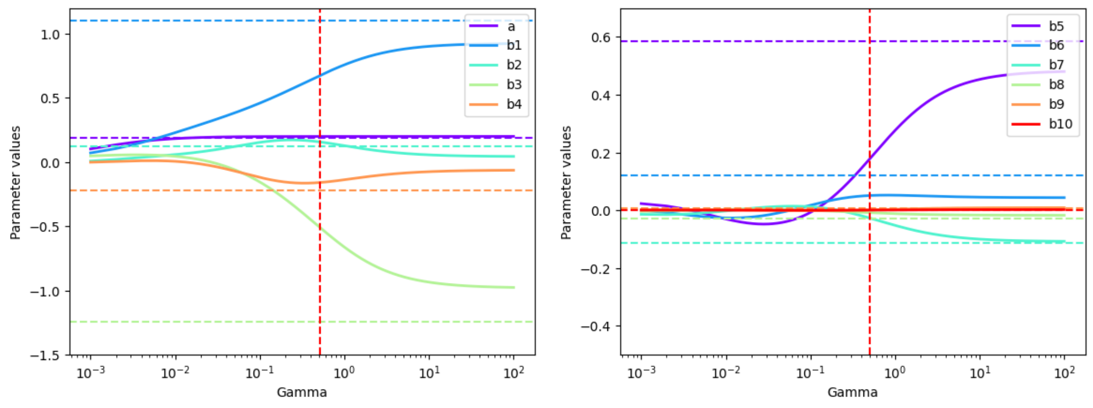
Where $\mathcal{N}(\mathbf{y}; a, bI_n)$ represents the normal density for $\mathbf{y}$ with mean a and variance bI_n. Then, using Bayes theorem, and the independence of the parameters a priori (meaning we can express the joint density as MVN), we obtain the following posterior:

$$p(\theta|\mathbf{y}, \Phi, \sigma) = \frac{p(\mathbf{y}|\Phi, \theta, \sigma)p(\theta)}{p(\mathbf{y}|\Phi, \sigma)} = \frac{\mathcal{N}(\mathbf{y}; \Phi\theta, \sigma^2 I_n)\mathcal{N}(\theta; \mathbf{0}, \gamma I_{11})}{\int \mathcal{N}(\mathbf{y}; \Phi\theta, \sigma^2 I_n)\mathcal{N}(\theta; \mathbf{0}, \gamma I_{11})d\theta} = \mathcal{N}(\theta, \mu, \mathbf{\Sigma})$$

Where the denominator is known as the marginal likelihood, and:

$$\mu = (\Phi^T\Phi + \frac{\sigma^2}{\gamma} I_{11})^{-1}\Phi^T\mathbf{y} \in \mathbb{R}^{11}, \qquad \mathbf{\Sigma} = \sigma^2(\Phi^T\Phi + \frac{\sigma^2}{\gamma} I_{11})^{-1} \in \mathbb{R}^{11\times 11}$$

We notice that μ is the same as the ridge regression parameter estimate $\hat{\theta}_{\text{ridge}}$, but just with $\lambda = \frac{\sigma^2}{\gamma}$. So since we know $\sigma = 0.3$, by varying γ, we essentially have the same effect as varying λ in ridge regression. We can see this in figure 5 below, where we vary $\gamma \in (0.001, 100]$:
We see that as $\gamma \to 0$, this corresponds to $\lambda \to \infty$, which forces all parameters to 0. This would correspond to an infinitely precise prior with all of its mass at 0 (Dirac delta), which explains why the parameters shrink to 0 in the Bayesian sense, since the finite amount of observations we have are unable to pull the parameters away from 0. As γ increases, we see that all of the parameters appear to converge to finite values. γ increasing means λ decreases, eventually to 0, meaning the parameters tend to the OLS parameters. In the Bayesian sense, this corresponds to a diffuse (flat) prior. The OLS parameters (which are obtained from the training set) are shown in figure 5 by the dashed horizontal lines. There are slight discrepancies between the limiting Bayesian values and the OLS values, purely because, the OLS parameters were found using a training set, whereas the limiting Bayesian values were found using



(a) First 5 parameters (b) Last 6 parameters

Figure 5: Plots showing how the parameter values change as we vary γ. The solid lines are the parameter values, the horizontal dashed lines correspond to the OLS parameter estimates, and the vertical dashed line corresponds to the value of γ that results in $\lambda = \lambda_{\text{ridge}}$, which was found earlier.

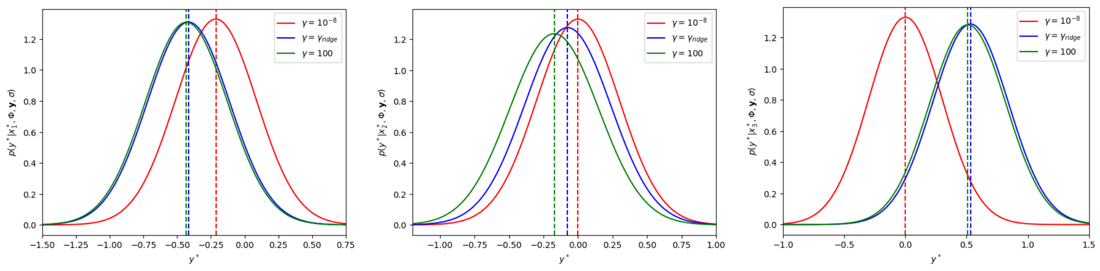
the full data set. These values would be exactly the same if the OLS parameters were found using the full dataset.

It is also interesting to see that some of the parameter values vary more than others, for example, b_1, b_3 and b_5 all vary a lot more with γ compared to the other parameters. The red dashed line shows the value of γ that results in $\lambda = \lambda_{\text{ridge}}$, which we found earlier, call this $\gamma_{\text{ridge}} = 0.5091$. We can read off the corresponding parameter values for this case, and note again that they wouldn't be exactly the same as $\hat{\theta}_{\text{ridge}}$ found earlier, due to the use of a training set in ridge regression, but the full dataset was used here.

Given this posterior distribution, we can find the posterior predictive distribution for y_j^* given new data $x_j^* \in \{-2, -0.5, 1\}$. The posterior predictive distribution is as follows (given the assumption that new data is still independent of all other data) :

$$p(y_j^*|\underline{x_j^*}, \Phi, \mathbf{y}, \sigma) = \int p(y_j^*, \theta|\underline{x_j^*}, \Phi, \mathbf{y}, \sigma)d\theta = \int p(y_j^*|\underline{x_j^*}, \theta, \sigma)p(\theta|\Phi, \mathbf{y}, \sigma)d\theta$$
$$= \mathcal{N}(y_j^*; \underline{x_j^*}^T \mu, \underline{x_j^*}^T \mathbf{\Sigma} \underline{x_j^*} + \sigma^2)$$

Where μ and $\mathbf{\Sigma}$ were defined earlier, and $\underline{x_j^*} = (\sin(\frac{2\pi}{0.3}x_j^*), x_j^*, ..., x_j^{*10})^T$. Note that the posterior predictive distribution depends on γ through μ and $\mathbf{\Sigma}$. We plot the distribution for each of the three points and vary $\gamma \in \{10^{-8}, \gamma_{\text{ridge}}, 100\}$:

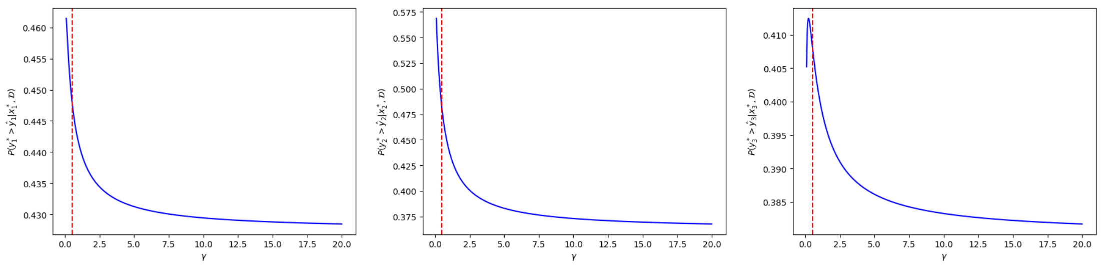


(a) Predictive distribution when $x^* = -2$ (b) Predictive distribution when $x^* = -0.5$ (c) Predictive distribution when $x^* = 1$

Figure 6: Plots showing the predictive distribution for each given x value, for 3 different values of γ. The dotted lines are the mean of each distribution.

Figure 6 shows the different distributions for each of the new data points $x_j^* \in \{-2, -0.5, 1\}$. We see in each case, that as $\gamma \to 0$ (red curve), the mean of the distribution is heading towards 0, which is what we expect since the parameter values are then shrunk to 0. Comparing the means of these distributions to the original data plot, we see that they are distributed where we would expect them to be. However, care should be taken for $x_2^* = -0.5$, since this is extrapolation. We do not have any data in this region, so any predictions about points within this region could be completely incorrect. Nevertheless, the distributions for γ_{ridge} and $\gamma = 100$ appear to be very similar when $x_1^* = -2$ or $x_3^* = 1$.

Finally, we are interested in the probability $P(y_j^* > \hat{y_j}|x_j^*, \mathcal{D})$, where $\hat{y_j}$ is the ridge regression model prediction. We use $\hat{y_j}$ calculated using the ridge coefficients found using the training set earlier, note that for $x_j^* \in \{-2, -0.5, 1\}$, $\hat{y_j} \in \{-0.3761, -0.0598, 0.6036\}$ respectively. We plot this below in figure 7 as a function of γ:

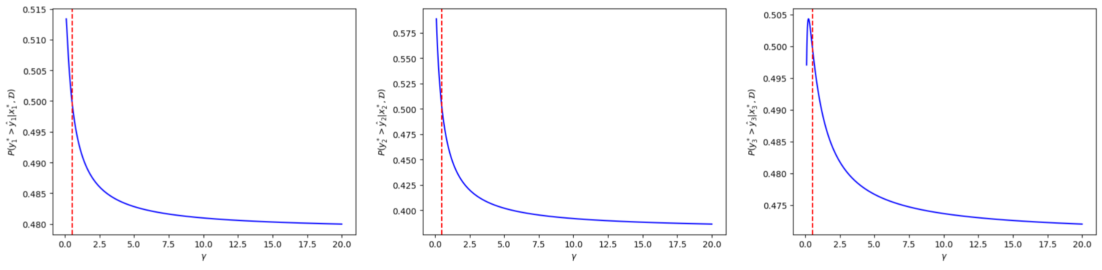


(a) Probability when $x_j^* = -2$ (b) Probability when $x_j^* = -0.5$ (c) Probability when $x_j^* = 1$

Figure 7: Plots showing the probability that the predicted value for y_j^* is greater than the corresponding value of the ridge prediction using $\theta_{\hat{\text{ridge}}}$ found on the training data. The red dotted line represents γ_{ridge}.

These plots show that in all three cases, the probability tends to a finite limit as γ increases. In the case of $x_j^* = -2$ and $x_j^* = -0.5$, the limit is from above. This makes sense, because when $\gamma \to 0$, $\lambda \to \infty$, meaning the coefficients are shrunk to 0. However, in both of these cases, the ridge estimate is less than 0, meaning as γ approaches 0, the probability will be 1. In the case that $x_j^* = 1$, the curve takes a more unusual shape. It makes sense that the probability tends to 0 as $\gamma \to 0$, since again, the coefficients are shrunk to 0, but this time the ridge estimate is greater than 0. Regardless, it is reassuring to see the probability approaches a limit as γ increases, which we expect since $\mu \to \hat{\theta}$, resulting in a fixed probability. Also, notice the value of the probabilities where the red dotted line representing γ_{ridge} intersects. They are all reasonably close to 0.5, but again we have this discrepancy since we used a training set to calculate $\hat{\theta}_{\text{ridge}}$.

As a sanity check, we repeat the plots but comparing to predictions from the ridge regression model that was built on the full dataset. In this case, we obtain $\hat{y_j} \in \{-0.4157, -0.0755, 0.5317\}$ for $x_j^* \in \{-2, -0.5, 1\}$ respectively. We plot this below in figure 8:



(a) Probability when $x_j^* = -2$ (b) Probability when $x_j^* = -0.5$ (c) Probability when $x_j^* = 1$

Figure 8: Plots showing the probability that the predicted value for y_j^* is greater than the corresponding value of the ridge prediction using $\hat{\theta}_{\text{ridge}}$ found on the full dataset. The red dotted line represents γ_{ridge}.

This time, notice that the red dotted line intersects the curve exactly at $P = 0.5$, which is what we expect, since at this point, the posterior predictive has mean equal to $x_j^{*T}\hat{\theta}_{\text{ridge}}$ where $\hat{\theta}_{\text{ridge}}$ is found on the whole dataset, and x_j^{*T} represents the design matrix form of the new data point. Thus the posterior probability that it is greater than this value is 0.5.

Question 2

Now, we study the Breast Cancer Wisconsin Data Set which contains 569 observations, each with 30 features, and a response of either "Malignant" or "Benign". We shall label this dataset as $\mathcal{D}$. The version we received already had all of the features normalized, which is necessary when it comes to classification via K-nearest neighbors, since if the features are on different scales, then it can impact the algorithm. We shall denote the design matrix as $X \in \mathbb{R}^{n\times p}$ and the response as $\mathbf{y} \in \mathbb{R}^n$ where $n = 569$ and $p = 30$. Each y_i is a binary variable, which originally took either "M" for malignant or "B" for benign. We modify this so $y_i = 1$ or $y_i = 0$ for malignant and benign respectively.

Of the 569 observations, 357 were benign and 212 were malignant. This means 37.3% of the data has $y_i = 1$. So there is a mild imbalance in the data. However, it shouldn't be enough to affect results. Below, in figure 9 we plot a histogram for each feature, split by group.

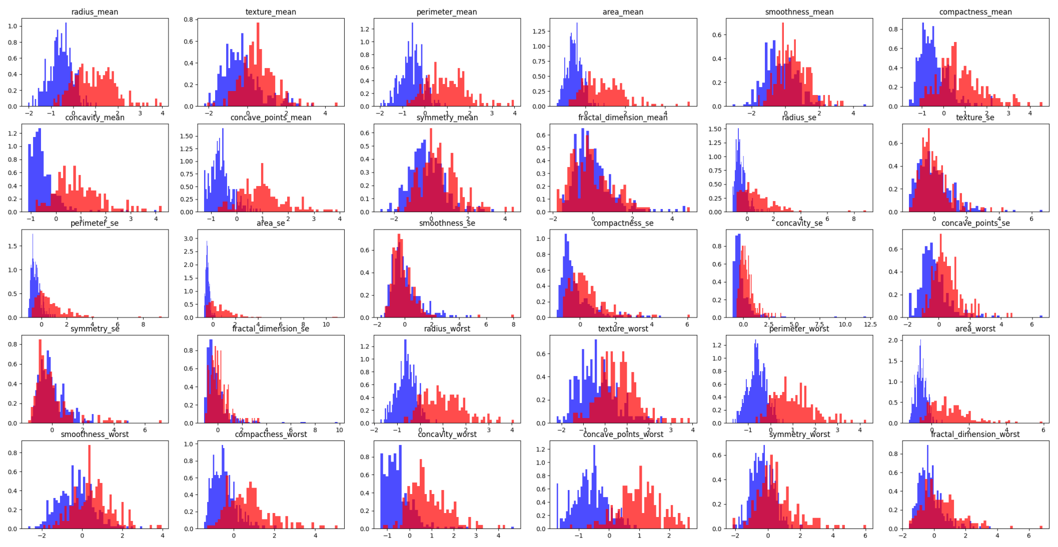
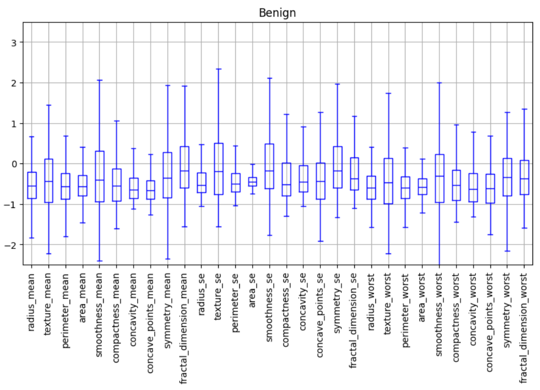
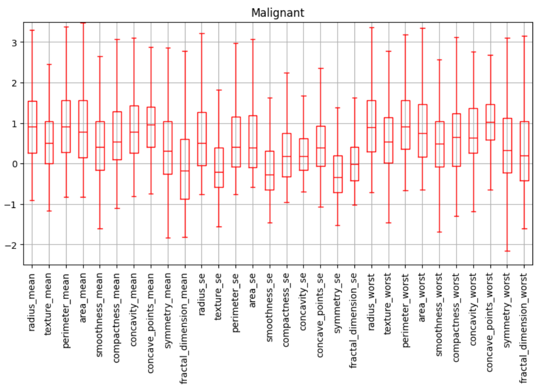


Figure 9: A plot showing histograms for each predictor variable, blue corresponds to benign, and orange is malignant.

In the figure, we can see that some features have more of a split in distribution between the two groups, for example, "radius_mean" and "concave_points_worst", whereas others have similar distributions between the two groups, for example, "symmetry_se" and "smoothness_se". This suggests that some variables will be more useful than others when it comes to building a predictive model. We also provide boxplots for each predictor, split by group once again, as seen below in figure 10:



(a) Boxplot for benign class



(b) Boxplot for malignant class

Figure 10: Boxplots for the two classes

This plot makes the differences between the two groups even more apparent; it appears that the malignant group tends to have higher means and quantiles than the benign group, suggesting classification is very much possible.

Part 2

Train/Test Split

We will be performing cross-validation later just like we did for finding λ in ridge regression. Before doing this, we shall split the data into a training set $\mathcal{D}_{\text{train}}$ and a test set $\mathcal{D}_{\text{test}}$, and compare the methods using cross-validation on $\mathcal{D}_{\text{train}}$, with a final comparison on $\mathcal{D}_{\text{test}}$. We split the data randomly, 70% into the training set, and 30% into the test set.

Logistic Regression

Logistic regression is a discriminative approach, meaning it models $p(Y = c|x)$ directly, without any assumptions for the data given the class or the probabilities of the classes. Since we are in the binary classification context, a reasonable assumption is that the class, Y, is Bernoulli, that is, $Y \sim \text{Ber}(\pi(\mathbf{x}))$, where $\mathbf{x}$ is a single observation. We then take a response function $h(.)$ such that $\mathbb{E}(Y|\mathbf{x}) = \mathbb{P}(Y = 1|\mathbf{x}) = \pi(\mathbf{x}) = h(\eta)$, where $\eta = \boldsymbol{\beta}^T\mathbf{x}$ is known as the linear predictor, and $\boldsymbol{\beta}$ is the parameter vector.

The response function used for logistic regression is the logistic function:

$$h(\eta) = \frac{e^{\eta}}{1 + e^{\eta}}$$

This is the standard choice, since it is bounded between 0 and 1, making it suitable to model probabilities. This means the probability is as follows:

$$\mathbb{P}(Y = 1|\mathbf{x}) = \frac{e^{\boldsymbol{\beta}^T\mathbf{x}}}{1 + e^{\boldsymbol{\beta}^T\mathbf{x}}}$$

The parameter $\boldsymbol{\beta}$ is found using some form of optimizer to find the maximum of the corresponding likelihood function. This could be IWLS, or gradient descent, to name a few. To perform this, we use the method **LogisticRegression** from the library **sklearn**. By default, this function uses L-2 penalization, with $\lambda = 1$. However, with a few trials to find optimal λ using the **LogisticRegressionCV** method, this made little impact. There is no other hyperparameter tuning required for this.

K-Nearest Neighbours

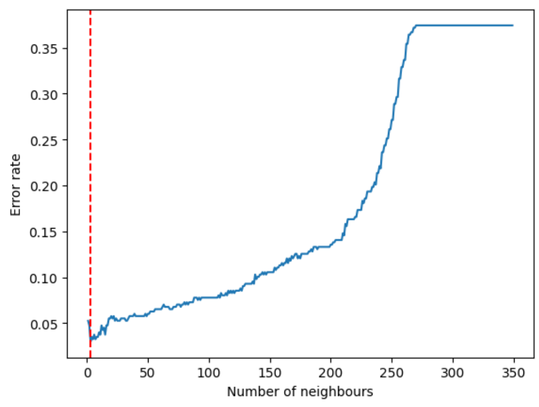
The K-nearest neighbour classifier does not assume any model for the data. It is non-parametric. It simply categorizes points based on what the surrounding data points are classified as. For this classifier, we need to specify two things: the number of neighbours K, and the distance metric $d(\cdot,\cdot) : \mathbb{R}^p \times \mathbb{R}^p \to \mathbb{R}$. Then, for a new data point x, we find its K nearest neighbours using the distance metric and can either report the most common class as the prediction or simply proportions of classes as probabilities (giving the most common class as the one with $p > 0.5$ in the binary setting.

The standard choice for the metric is the Minkowski metric:

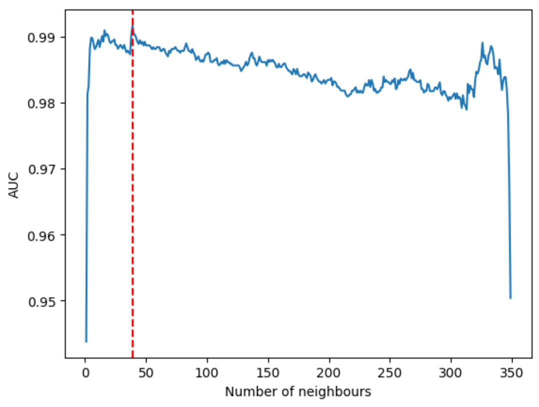
$$d(x^{(i)}, y^{(i)}) = \left(\sum_{j=1}^{p} ||x_j^{(i)} - y_j^{(i)}||^a \right)^{\frac{1}{a}}$$

$a = 2$ corresponds to the Euclidean norm, which is what we shall use. For unscaled data, different features having different scales can effect the distance metric, however, since the data has already been scaled, we do not need to worry about this. As for the number of neighbours K, we choose this by using 10-fold CV on $\mathcal{D}_{\text{train}}$, which makes this method the most computationally costly, since for multiple K we perform all of these distance calculations for each subset of points. We implement this in python using the method **KNeighborsClassifier** from the library **sklearn**. The below plots in figure 11 show a plot for the error rate and a plot for the area under the ROC curve (AUC) as the number of neighbours K varies:

Upon running 10-fold cross-validation and plotting the corresponding curves, we found that $k = 3$ was optimal for minimizing error rate, and $K = 39$ was optimal for maximizing AUC. We shall consider both of these values in the final comparisons between all of these models.



(a) Error rate for K-NN as K varies



(b) Error rate for K-NN as K varies

Figure 11: Error rate and AUC plots for K-NN, the red dotted line gives the best K.

Naive Bayes

The Naive Bayes classifier is a generative approach, that is, it directly defines the class priors and class conditional likelihoods. The Naive Bayes assumption is that all features are independent given the class label. For a given observation $x^{(i)} = (x_1^{(i)}, ..., x_p^{(i)})$, we have:

$$p(x^{(i)}|y = k) = \prod_{j=1}^{p} p(x_j^{(i)}|y = k, \theta_{jk})$$

Where θ_{jk} is a parameter for the class conditional likelihood, associated with the $j-th$ covariate for the $k-th$ class. Since all of the data is continuous and scaled, we assign Gaussian class conditional densities:

$$x_j^{(i)}|y = k \sim N(\mu_{jk}, \sigma_{jk}^2), j = 1, ..., p, k = 0, 1$$

Then for the dataset $\mathcal{D}$, we get, using the independence assumption, the following log-likelihood function:

$$\log p(\mathcal{D}|\mu, \sigma\pi) = \sum_{k=0}^{1} N_k \log \pi_k + \sum_{k=0}^{1} \sum_{j=1}^{p} \sum_{i:y^{(i)}=k} \log \phi(x_j^{(i)}; \mu_{jk}, \sigma_{jk}^2)$$

Where ϕ is the normal pdf, N_k is the number of observations in class k, and π_k are the prior probabilities for each class. μ, σ and π would be estimated using maximum likelihood estimation. Luckily, there is no hyper-parameter tuning required for this model.

Prediction is then done by computing the discriminant and comparing it to different thresholds:

$$\log \left(\frac{p(y = 1|x)}{p(y = 0|x)} \right) = \log \left(\frac{\pi_1 p(x|y = 1)}{\pi_0 p(x|y = 0)} \right) \tag{1}$$

We implement this in python using the method **GaussianNB** from the library **sklearn**.

Model Comparison

Now, we compare the three models, using 10-fold CV on the $\mathcal{D}_{\text{train}}$, and then finally on the test set $\mathcal{D}_{\text{test}}$. We show the results in table 2 below, for accuracy and AUC, with the best for each column in bold:

Model	CV-Acc-mean	CV-Acc-var	CV-AUC-mean	CV-AUC-var	Test-Acc	Test-AUC
Logistic	**0.9774**	**0.0175**	**0.9919**	0.0115	**0.9707**	**0.9959**
KNN (K = 3)	0.9699	0.0245	0.9825	0.0206	0.9649	0.9832
KNN (K = 39)	0.9397	0.0230	0.9917	**0.0108**	0.9649	0.9906
Naive Bayes	0.9398	0.0277	0.9874	0.0135	0.9357	0.9840

Table 2: A comparison of OLS regression and ridge regression coefficients.

The models all achieve similar results in each metric, with Naive Bayes perhaps falling a little behind. However, we can see that logistic regression is the best in each form of comparison. When it comes to stability, all models seem to have relatively high stability across the cross validation scheme. Logistic regression was the most stable for accuracy, and KNN with $K = 39$ was the most stable for AUC. As a final visualization, we plot the ROC curves for the test set, which is a plot of how the True positive rate and false positive rate change as the decision threshold (the probability where we separate the classes) is varied:

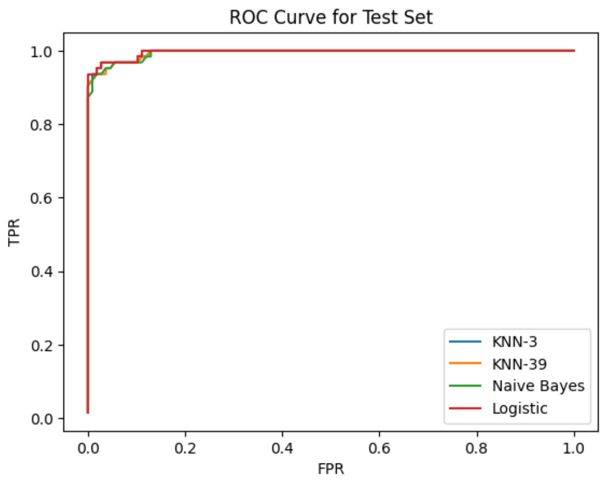


Figure 12: The ROC curve on the test set for the 4 models

Figure 12 shows that all of the models perform similarly, however, again, logistic regression performs the best as the decision threshold is varied. With this in mind, in combination with table 2, we conclude that logistic regression is the best in this situation. Naive Bayes performs a little worse than the other models. It was based on the assumption that the class conditional densities for each feature were normal. However, looking at some of the histograms in figure 9, some of the distributions do appear skewed, which explains why this classifier doesn't perform as well.

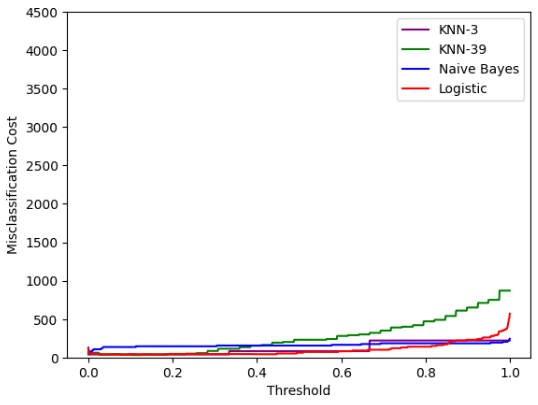
Part 3

Now, we consider the cost of wrongly misclassifying a mass as benign as 100, and the cost of wrongly misclassifying a mass as malignant as 5. We aim to analyse how the misclassification cost varies as we vary the decision threshold for each model.

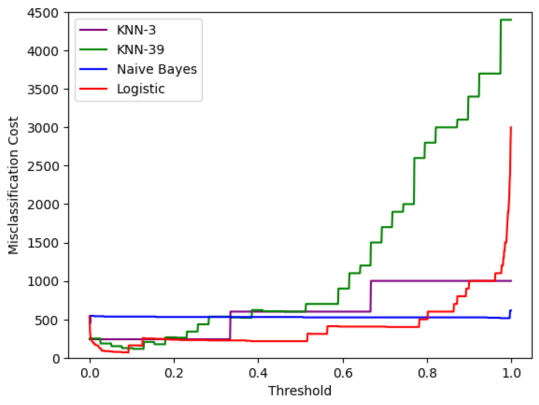
Part 3a

We shall consider all 4 models in the previous question, it will be interesting to see how the two KNN classifiers compare now there is a cost associated with misclassification. We shall use the same $\mathcal{D}_{\text{train}}$ and $\mathcal{D}_{\text{test}}$ as before, performing 10-fold cross-validation on the training set, and then a final comparison on the test set. As for what we are comparing, we shall calculate the total misclassification cost on each iteration as $C_{\text{total}} = 100N_{\text{malignant missclassified}} + 5N_{\text{benign missclassified}}$. For the CV version, the curve shall be averaged across the iterations. We plot the CV curves and the test set curves below, on the same scale, in figure 13:

This figure shows that for a low threshold value, all 4 classifiers have a low misclassification cost, which makes sense since we took malignant as the positive class (1), so if everything is classified as malignant, then we don't misclassify any malignant tumors as benign, meaning we avoid the higher cost. In both cases, KNN with $K = 39$ performs the worst as the decision threshold is varied, and logistic regression performs the best for most thresholds, only really beginning to increase past KNN-3 and Naive Bayes towards the upper end. However, realistically we would never set the threshold to be that high, since then we are only classifying a tumor as malignant if the models are absolutely certain, which is a bad thing, it is better to set the threshold lower and risk misclassifying benign tumors, taking a more cautious viewpoint.



(a) Misclassification cost curve for CV on $\mathcal{D}_{\text{train}}$



(b) Misclassification cost curve for $\mathcal{D}_{\text{test}}$

Figure 13: Error rate and AUC plots for KNN, the red dotted line gives the best K.

Finally, we plot figure 13a but on a different scale:

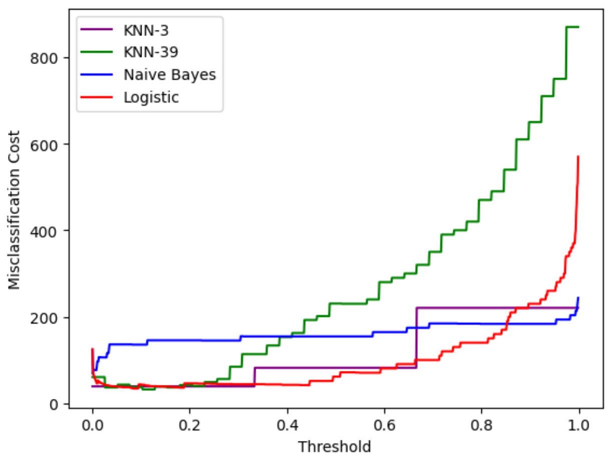


Figure 14: Misclassification cost curve for CV on $\mathcal{D}_{\text{train}}$

This figure shows very similar results to figure 13b, but on a smaller scale. KNN-39 performs the worst, and logistic regression the best (up towards the upper threshold region), with Naive Bayes and KNN-39 somewhere in the middle.

Part 3b

To ensure that the cost is minimized, it would depend on where the threshold is set within the context of the situation. Logistic regression would be the best overall, since it retains a lower cost for the majority of thresholds. Should the situation arise where the decision threshold needs to be set high, then either Naive Bayes or KNN with $K = 3$ should be favoured. Interestingly, this analysis has shown that despite KNN with $K = 39$ being selected as optimal when maximizing AUC, it falls behind $K = 3$ when it comes to classification costs.

A Code Appendix

A.1 Question 1

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.model_selection import train_test_split

#function to make design matrix
def make_design(x,poly_order):
    #first we make the sin term:
    X = np.sin(2*np.pi/0.3 * x)

    #now add polynomial terms
    for p in range(1, poly_order + 1):
        X = np.concatenate([X, x ** p], axis=1)
    return X

### 1) ###
q1_data = pd.read_csv("/University-local\/Imperial/Term 2/Machine Learning/ML/CW_1/data_2476922.csv"

x = q1_data["x"].to_numpy()[:,None]
y = q1_data[" y"].to_numpy()[:,None]

#do train split so we can show overfitting and underfitting
plt.plot(x,y,"o")
plt.xlabel("x")
plt.ylabel("y")
plt.show()

def get_mse(x, y, poly_order) -> float:
    X = make_design(x,poly_order=poly_order)
    #now split the same each time
    X_train,X_test,y_train,y_test = train_test_split(X,y,test_size=0.3,random_state=8)
    # solve for coefs such that X^T @ X @ coefs = X.T @ y
    coefs = np.linalg.solve(X_train.T @ X_train, X_train.T @ y_train)
    pred_y = X_train @ coefs
    mse = np.mean((pred_y - y_train) ** 2)
    #test
    pred_y_test = X_test @ coefs
    mse_test = np.mean((pred_y_test - y_test) ** 2)

    return (mse,mse_test,coefs)

#we now do plots
orders = [i for i in range(0, 30)]
MSE = [get_mse(x, y, p) for p in orders]
coefs = [sublist[2] for sublist in MSE]
plt.plot(
    orders,
    [sublist[0] for sublist in MSE],
    label="Train",
)
plt.plot(
    orders,
    [sublist[1] for sublist in MSE],
    label="Test",
)
plt.legend()
plt.ylabel("MSE")
plt.xlabel("Order")
plt.show()
print(MSE[0][0])

### 1) 2 ###

polyorder = 10

X = make_design(x,polyorder)

def LOOCV_lambda(X, y, lambda_):
    # X is the matrix of features (including polynomials)
    # y is a vector containing the targets
    # lambda_ is the parameter for ridge regression
    NumOfDataPairs = len(y)
    # Initialize CV variable for storing results
    CV = np.zeros(NumOfDataPairs)
    for n in range(NumOfDataPairs):
        # Create training design matrix and target data, leaving one out each time
        Train_X = np.delete(X, n, axis=0)
        Train_y = np.delete(y, n)
        # Create testing design matrix and target data
        Test_X = X[n, :]
        Test_y = y[n]
        # Learn the optimal parameters using MSE loss
        Paras_hat = np.linalg.solve(Train_X.T @ Train_X + lambda_ * np.eye(X.shape[1]),
        Train_X.T @ Train_y)
        Pred_y = Test_X @ Paras_hat
        # Calculate the MSE of prediction using training data
        CV[n] = (Pred_y - Test_y) ** 2
    return np.mean(CV)

lmbdas = np.concatenate(([0], 2 ** np.arange(-10, 1, 0.1)))
CVlambda = np.array([LOOCV_lambda(X, y, lambda_) for lambda_ in lmbdas])
plt.plot(lmbdas, CVlambda)
plt.ylabel("Average MSE on LOOCV")
plt.xlabel("$\lambda$")

# Find the lambda value that minimizes the average MSE
lambda_l = lmbdas[np.argmin(CVlambda)]

plt.axvline(x = lambda_l, color = "r", linestyle = "--", label = "$\lambda_{ridge}$")
plt.legend()
plt.show()
print(f"Average MSE on LOOCV minimized for lambda_hat = {lambda_l:.4f}")


#we now have our best lambda, so we fit both OLS and ridge on training set,
#and test it on a test set
x_train,x_test,y_train,y_test = train_test_split(x,y,test_size=0.3,random_state=50)
#design matrices
X_train= make_design(x_train,polyorder)
X_test= make_design(x_test,polyorder)

#printing coefs
```

```
OLS_coefs = np.linalg.solve(X_train.T @ X_train, X_train.T @ y_train)
#OLS_coefs = np.linalg.solve(X.T @ X, X.T @ y)
Ridge_coefs = np.linalg.solve(X_train.T @ X_train + lambda_l * np.eye(X_train.shape[1]),
X_train.T @ y_train)
Ridge_coefs =np.linalg.solve(X.T @ X + lambda_l * np.eye(X.shape[1]), X.T @ y)
print(OLS_coefs)
print(Ridge_coefs)

y_test_pred_OLS = X_test @ OLS_coefs
OLS_mse_test = np.mean((y_test_pred_OLS - y_test)**2)
y_test_pred_Ridge = X_test @ Ridge_coefs
Ridge_mse_test = np.mean((y_test_pred_Ridge - y_test)**2)

y_train_pred_OLS = X_train @ OLS_coefs
OLS_mse_train = np.mean((y_train_pred_OLS - y_train)**2)
y_train_pred_ridge = X_train @ Ridge_coefs
Ridge_mse_train = np.mean((y_train_pred_ridge - y_train)**2)

#plot curves

plt.plot(x_train,y_train,"o", color = "g", label = "Training data", alpha = 0.8)
plt.plot(x_test,y_test, "o", color = "orange",label = "Testing data", alpha = 0.8)

x_range = np.linspace(-3,2,100)[:,None]
X_range = np.sin(2*np.pi/0.3 * x_range)
polyorder = 10
X_range = make_design(x_range, polyorder)
#OLS
y_range_OLS = X_range @ OLS_coefs
# Ridge
y_range_Ridge = X_range @ Ridge_coefs

plt.plot(x_range, y_range_OLS, color = "r" ,label = "OLS")
plt.plot(x_range, y_range_Ridge, color = "b", label = "Ridge")

plt.legend()
plt.xlabel("x")
plt.ylabel("y")
plt.show()

### 1) 2c ###

sigma = 0.3

gammalist = np.arange(0.001, 100.001, 0.001)
#find value of gamma that gives the ridge value:
gamma_ridge = sigma**2 / lambda_l
print(gamma_ridge)
# Calculate mu for each alpha
mu = np.array([np.linalg.solve(X.T @ X + np.diag(np.repeat(sigma**2/gamma, X.shape[1])), X.T @ y)
for gamma in gammalist])
# Define color palette
color_palette = plt.cm.rainbow(np.linspace(0, 1, 6))


# Plot the first 5 rows of mu against gamma
param_names = ["a", "b1", "b2", "b3", "b4","b5", "b6", "b7", "b8","b9","b10"]
plt.figure()
```

```
for i in range(5):
    plt.plot(gammalist, mu[:, i], color=color_palette[i], linewidth=2,label = param_names[i])
    plt.axhline(y = OLS_coefs[i][0], color=color_palette[i], linestyle = "--")
plt.axvline(x = gamma_ridge,linestyle = "--",color = "r")
plt.ylim((-1.5,1.2))
plt.xlabel("Gamma")
plt.ylabel("Parameter values")
plt.xscale("log")
plt.legend(loc="upper right")

# Plot the last 5 rows of mu against gamma
plt.figure()
for i in range(5, 11):
    plt.plot(gammalist, mu[:, i], color=color_palette[i-5], linewidth=2,label = param_names[i])
    plt.axhline(y = OLS_coefs[i][0], color=color_palette[i-5], linestyle = "--")
plt.axvline(x = gamma_ridge,linestyle = "--",color = "r")
plt.ylim((-0.5,0.7))
plt.xlabel("Gamma")
plt.ylabel("Parameter values")
plt.xscale("log")
plt.legend(loc="upper right")

plt.show()

# posterior predictive
X = make_design(x,polyorder)
def posterior_predictive(x_point,gamma):
    mu = np.linalg.solve(X.T @ X + np.diag(np.repeat(sigma**2/gamma, X.shape[1])), X.T @ y)
    Sigma = np.linalg.inv(X.T @ X + np.diag(np.repeat(sigma**2/gamma, X.shape[1]))) * sigma**2
    #change x to be of desired form
    x_point = np.array([x_point])[:,None]
    x_point = make_design(x_point,polyorder)[0][:,None]
    mu_new = x_point.T @ mu
    Sigma_new = x_point.T@ Sigma @ x_point + sigma**2
    return (mu_new,Sigma_new)

#plot predictive distributions, for each input value
from scipy.stats import norm
xlist = [-2,-0.5,1]
gamma_range = [0.00000001,gamma_ridge,100]
x_axis = np.arange(-2,1,0.01)
#color
color_palette = ["red","blue","green"]
labels = ["$\gamma = 10^{-8}$", "$\gamma = \gamma_{ridge}$", "$\gamma = 100$"]
for i in range(3):
    gamma = gamma_range[i]
    plt.plot(x_axis,norm.pdf(x_axis,
                         posterior_predictive(xlist[0],gamma)[0][0][0],
                         np.sqrt(posterior_predictive(xlist[0],gamma)[1][0][0])),
                         color = color_palette[i], label = labels[i])
    plt.axvline(x =posterior_predictive(xlist[0],gamma)[0][0][0], color = color_palette[i] ,
    linestyle = "--")
plt.xlim((-1.5,0.75))
plt.xlabel("$y^*$")
plt.ylabel("$p(y^*|x_1^*,\Phi,\mathbf{y},\sigma)$")
plt.legend()
plt.show()
```

```
from scipy.stats import norm

xlist = [-2,-0.5,1]
gamma_range = [0.00000001,gamma_ridge,100]
x_axis = np.arange(-1.5,1.5,0.01)
#color
color_palette = ["red","blue","green"]
labels = ["$\gamma = 10^{-8}$", "$\gamma = \gamma_{ridge}$", "$\gamma = 100$"]
for i in range(3):
    gamma = gamma_range[i]
    plt.plot(x_axis,norm.pdf(x_axis,posterior_predictive(xlist[1],gamma)[0][0][0],
    np.sqrt(posterior_predictive(xlist[1],gamma)[1][0][0])),color = color_palette[i],
    label = labels[i])
    plt.axvline(x =posterior_predictive(xlist[1],gamma)[0][0][0], color = color_palette[i] ,
    linestyle = "--")
plt.xlim((-1.2,1))
plt.xlabel("$y^*$")
plt.ylabel("$p(y^*|x_2^*,\Phi,\mathbf{y},\sigma)$")
plt.legend()
plt.show()

from scipy.stats import norm

xlist = [-2,-0.5,1]
gamma_range = [0.00000001,gamma_ridge,100]
x_axis = np.arange(-1,2,0.01)
#color
color_palette = ["red","blue","green"]
labels = ["$\gamma = 10^{-8}$", "$\gamma = \gamma_{ridge}$", "$\gamma = 100$"]
for i in range(3):
    gamma = gamma_range[i]
    plt.plot(x_axis,norm.pdf(x_axis,posterior_predictive(xlist[2],gamma)[0][0][0],
    np.sqrt(posterior_predictive(xlist[2],gamma)[1][0][0])),color = color_palette[i],
    label = labels[i])
    plt.axvline(x =posterior_predictive(xlist[2],gamma)[0][0][0], color = color_palette[i] ,
    linestyle = "--")
plt.xlim((-1,1.5))
plt.xlabel("$y^*$")
plt.ylabel("$p(y^*|x_3^*,\Phi,\mathbf{y},\sigma)$")
plt.legend()
plt.show()

#Now for probabilities greater than ridge, using ridge found from training set

xlist = [-2,-0.5,1]

Ridge_coefs = np.linalg.solve(X_train.T @ X_train + lambda_l * np.eye(X_train.shape[1]),
X_train.T @ y_train)

#calc ridge prediction for each:
x_design_list = [np.array([i])[:,None] for i in xlist]
x_design_list = [make_design(i,polyorder)[0][:,None] for i in x_design_list]
ridge_predict = [X_new.T @ Ridge_coefs for X_new in x_design_list]
ridge_predict = [i[0][0] for i in ridge_predict]
print(ridge_predict)

#plot
```

```
from scipy.stats import norm
gammalist = np.arange(0.1, 20.01, 0.01)
#now find the probability
def find_prob(mu,var,ridge_val):
    return 1 - norm.cdf(ridge_val,loc = mu, scale = np.sqrt(var))

#take first x case
xpar = xlist[0]
p = [find_prob(posterior_predictive(xpar,g)[0][0][0],
posterior_predictive(xpar,g)[1][0][0],ridge_predict[0]) for g in gammalist]
plt.plot(gammalist,p, label = "$$", color = "b")
plt.axvline(x = gamma_ridge, color = "red", linestyle = "--")
plt.xlabel("$\gamma$")
plt.ylabel("$P(y_1^* > \hat{y}_{1} | x_1^*, \mathcal{D})$")
plt.show()
#take second x case
xpar = xlist[1]
p = [find_prob(posterior_predictive(xpar,g)[0][0][0],
posterior_predictive(xpar,g)[1][0][0],ridge_predict[1]) for g in gammalist]
plt.plot(gammalist,p, color = "b")
plt.axvline(x = gamma_ridge, color = "red", linestyle = "--")
plt.xlabel("$\gamma$")
plt.ylabel("$P(y_2^* > \hat{y}_{2} | x_2^*, \mathcal{D})$")
plt.show()
#take third x case
xpar = xlist[2]
p = [find_prob(posterior_predictive(xpar,g)[0][0][0],
posterior_predictive(xpar,g)[1][0][0],ridge_predict[2]) for g in gammalist]
plt.plot(gammalist,p, color = "b")
plt.axvline(x = gamma_ridge, color = "red", linestyle = "--")
plt.xlabel("$\gamma$")
plt.ylabel("$P(y_3^* > \hat{y}_{3} | x_3^*, \mathcal{D})$")
plt.show()

Ridge_coefs =np.linalg.solve(X.T @ X + lambda_l * np.eye(X.shape[1]), X.T @ y)

#now for ridge coeffs found on full dataset


xlist = [-2,-0.5,1]

#change ridge coefs to be full dataset
Ridge_coefs =np.linalg.solve(X.T @ X + lambda_l * np.eye(X.shape[1]), X.T @ y)

#calc ridge prediction for each:
x_design_list = [np.array([i])[:,None] for i in xlist]
x_design_list = [make_design(i,polyorder)[0][:,None] for i in x_design_list]
ridge_predict = [X_new.T @ Ridge_coefs for X_new in x_design_list]
ridge_predict = [i[0][0] for i in ridge_predict]


from scipy.stats import norm
gammalist = np.arange(0.1, 20.01, 0.01)
#now find the probability
def find_prob(mu,var,ridge_val):
    return 1 - norm.cdf(ridge_val,loc = mu, scale = np.sqrt(var))

#take first x case
```

```
xpar = xlist[0]
p = [find_prob(posterior_predictive(xpar,g)[0][0][0],
posterior_predictive(xpar,g)[1][0][0],ridge_predict[0]) for g in gammalist]
plt.plot(gammalist,p, label = "$$", color = "b")
plt.axvline(x = gamma_ridge, color = "red", linestyle = "--")
plt.xlabel("$\gamma$")
plt.ylabel("$P(y_1^* > \hat{y}_{1} | x_1^*, \mathcal{D})$")
plt.show()
#take second x case
xpar = xlist[1]
p = [find_prob(posterior_predictive(xpar,g)[0][0][0],
posterior_predictive(xpar,g)[1][0][0],ridge_predict[1]) for g in gammalist]
plt.plot(gammalist,p, color = "b")
plt.axvline(x = gamma_ridge, color = "red", linestyle = "--")
plt.xlabel("$\gamma$")
plt.ylabel("$P(y_2^* > \hat{y}_{2} | x_2^*, \mathcal{D})$")
plt.show()
#take third x case
xpar = xlist[2]
p = [find_prob(posterior_predictive(xpar,g)[0][0][0],
posterior_predictive(xpar,g)[1][0][0],ridge_predict[2]) for g in gammalist]
plt.plot(gammalist,p, color = "b")
plt.axvline(x = gamma_ridge, color = "red", linestyle = "--")
plt.xlabel("$\gamma$")
plt.ylabel("$P(y_3^* > \hat{y}_{3} | x_3^*, \mathcal{D})$")
plt.show()
```

A.2 Question 2

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.model_selection import train_test_split

data = pd.read_csv("D:/University-local/Imperial/Term 2/Machine Learning/ML/CW_1/breast_cancer_scale
#print(data.shape)
response = data["diagnosis"].to_numpy()
#convert to 1 for malignant and 0 for benign
response = np.array([1 if i == "M" else 0 for i in response ])[:,None]
#get features only
feature_data = data.drop("diagnosis",axis = 1)

#class imbalance?
print(data[response == 1].shape)
print(data[response == 0].shape)
print(212/569)

#produce boxplot for this:
data.boxplot(rot = 90)
plt.show()

#boxplot by group:
feature_data_d = feature_data[response == 0]
feature_data_m = feature_data[response == 1]

plt.figure().set_figwidth(10)
axs= feature_data_d.boxplot(rot = 90,showfliers = False,color = "blue")
axs.set_title("Benign")
axs.set_ylim([-2.5,3.5])

plt.show()
plt.figure().set_figwidth(10)
ax = feature_data_m.boxplot(rot = 90,showfliers = False,color = "red")
ax.set_ylim([-2.5,3.5])
ax.set_title("Malignant")
plt.show()

#plot feature histograms
def plot_histograms(data):
    col_names = data.columns
    fig,axes = plt.subplots(nrows = 5, ncols =6)
    fig.set_figheight(15)
    fig.set_figwidth(30)

    for i in range(5):
        for j in range(6):
            axes[i,j].hist(data.iloc[:,6*i + j],bins = 30,alpha = 0.7)
            axes[i,j].set_title(col_names[6*i + j])
    plt.show()

plot_histograms(feature_data)


#partitioning by group
def plot_histograms_group(data,response):
    col_names = data.columns
    data_b = data[response == 0]
    data_m = data[response == 1]
    fig,axes = plt.subplots(nrows = 5, ncols =6)
    fig.set_figheight(15)
    fig.set_figwidth(30)

    for i in range(5):
        for j in range(6):
            axes[i,j].hist(data_b.iloc[:,6*i + j],bins = 50,alpha = 0.7,density= True,
            color = "blue")
            axes[i,j].hist(data_m.iloc[:,6*i + j],bins = 50,alpha = 0.7,density= True,
            color = "red")
            axes[i,j].set_title(col_names[6*i + j])

    plt.show()

plot_histograms_group(feature_data,response)


#split the data into a training set and a test set
y = response
X = feature_data.to_numpy()
X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=0.3,random_state=1)

# Now actually begin the model comparison
# We begin with logistic regression
from sklearn.linear_model import LogisticRegression
logistic = LogisticRegression()
cancer_logit = logistic.fit(X_train,y_train.ravel())
cancer_logit_test = cancer_logit.predict(X_test)
```

```
print("Test error for logistic regression:", np.mean(cancer_logit_test != y_test.ravel()))
print("Accuracy for logistic regression:", np.mean(cancer_logit_test == y_test.ravel()))

#so we get a pretty decent error rate

from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import KFold

#This curve is from lecture code
def roc_curve(labels, scores):

    idx = np.argsort(scores)[::-1]
    labels = labels[idx]

    res = pd.DataFrame()

    res["TPR"] = np.cumsum(labels) / np.sum(labels)
    res["FPR"] = np.cumsum(1 - labels) / np.sum(1 - labels)

    return res

#k fold cross validation for KNN
def knn_CV_Kfold(X,y,k,fold):
    CV = np.zeros([fold,1])
    AUC = np.zeros([fold,1])
    for i in range(fold):
        split = KFold(n_splits=10)
        idx = list(split.split(X))
        X_train = X[idx[i][0],:]
        X_test = X[idx[i][1],:]
        y_train = y[idx[i][0]]
        y_test = y[idx[i][1]]
        #model
        knn = KNeighborsClassifier(n_neighbors=k)
        knn.fit(X_train, y_train.ravel())
        knn_test = knn.predict(X_test)
        #auc
        knn_test_probs = knn.predict_proba(X_test)
        knn_roc = roc_curve(y_test, knn_test_probs[:, -1])
        knn_auc = np.trapz(y=knn_roc["TPR"], x=knn_roc["FPR"])
        #acc
        CV[i] = np.mean(knn_test != y_test.ravel())
        AUC[i] = knn_auc
    return np.mean(CV), np.mean(AUC)

k_range = np.arange(1,350)
k_CVfold= [knn_CV_Kfold(X_train,y_train,k,10) for k in k_range]


#plotting AUC and ACC
k_CVfold_err = [i[0] for i in k_CVfold]
plt.plot(k_range, k_CVfold_err)
plt.xlabel("Number of neighbours")
plt.ylabel("Error rate")
min_k = k_range[np.argmin(k_CVfold_err)]
plt.axvline(x = min_k, color = "r", linestyle = "--")
plt.show()
```

```
print(min_k)

k_CVfold_AUC = [i[1] for i in k_CVfold]
plt.plot(k_range, k_CVfold_AUC)
plt.xlabel("Number of neighbours")
plt.ylabel("AUC")
max_k= k_range[np.argmax(k_CVfold_AUC)]
plt.axvline(x = max_k, color = "r", linestyle = "--")
plt.show()
print(max_k)


### GAUSSIAN NAIVE BAYES ###
#since features could be gaussian, not an unreasonable assumtion
from sklearn.naive_bayes import GaussianNB
gnb = GaussianNB()
gnb.fit(X_train,y_train)
gnb_pred = gnb.predict(X_test)
print(gnb_pred)
print(np.mean(gnb_pred == y_test.ravel()))

### FINAL COMPARISONS - KNN ###

from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import KFold
#KNN
#function to do k-fold CV

def CV_Kfold(X,y,fold,model):
    CV = np.zeros([fold,1])
    AUC = np.zeros([fold,1])
    for i in range(fold):
        split = KFold(n_splits=10)
        idx = list(split.split(X))
        X_train = X[idx[i][0],:]
        X_test = X[idx[i][1],:]
        y_train = y[idx[i][0]]
        y_test = y[idx[i][1]]
        #model
        mod = model
        mod.fit(X_train, y_train.ravel())
        mod_test = mod.predict(X_test)
        #auc
        mod_test_probs = mod.predict_proba(X_test)
        mod_roc = roc_curve(y_test, mod_test_probs[:, -1])
        mod_auc = np.trapz(y=mod_roc["TPR"], x=mod_roc["FPR"])
        #acc
        CV[i] = np.mean(mod_test == y_test.ravel())
        AUC[i] = mod_auc
    return np.mean(CV), np.std(CV), np.mean(AUC), np.std(AUC)

#3
knn_3 = CV_Kfold(X_train, y_train, 10, KNeighborsClassifier(n_neighbors=3))
print(knn_3)
#39
knn_39 = CV_Kfold(X_train, y_train, 10, KNeighborsClassifier(n_neighbors=39))
print(knn_39)
```

```
# Test set k = 3:

knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train, y_train.ravel())
knn_test = knn.predict(X_test)
print(np.mean(knn_test == y_test.ravel()))
knn_test_probs = knn.predict_proba(X_test)
knn_roc = roc_curve(y_test, knn_test_probs[:, -1])
knn_auc = np.trapz(y=knn_roc["TPR"], x=knn_roc["FPR"])
print(knn_auc)

#k = 39
knn_39 = KNeighborsClassifier(n_neighbors=39)
knn_39.fit(X_train, y_train.ravel())
knn_39_test = knn_39.predict(X_test)
print(np.mean(knn_39_test == y_test.ravel()))
knn_39_test_probs = knn_39.predict_proba(X_test)
knn_roc_39 = roc_curve(y_test, knn_39_test_probs[:, -1])
knn_39_auc = np.trapz(y=knn_roc_39["TPR"], x=knn_roc_39["FPR"])
print(knn_39_auc)



### LOGISTIC ###
logistic = CV_Kfold(X_train, y_train, 10, LogisticRegression())
print(logistic)
#test set
logistic = LogisticRegression()
logistic.fit(X_train, y_train.ravel())
logistic_test = logistic.predict(X_test)
print(np.mean(logistic_test == y_test.ravel()))
logistic_test_probs = logistic.predict_proba(X_test)
logistic_roc = roc_curve(y_test, logistic_test_probs[:, -1])
logistic_auc = np.trapz(y=logistic_roc["TPR"], x=logistic_roc["FPR"])
print(logistic_auc)


### NAIVE BAYES ###

nb = CV_Kfold(X_train, y_train, 10, GaussianNB())
print(nb)

#test set
nb = GaussianNB()
nb.fit(X_train, y_train.ravel())
nb_test = nb.predict(X_test)
print(np.mean(nb_test == y_test.ravel()))
nb_test_probs = nb.predict_proba(X_test)
nb_roc = roc_curve(y_test, nb_test_probs[:, -1])
nb_auc = np.trapz(y=nb_roc["TPR"], x=nb_roc["FPR"])
print(nb_auc)


### ROC CURVE ###

plt.plot(knn_roc["FPR"], logistic_roc["TPR"], label = "KNN-3",color = "purple",alpha = 0.7)
plt.plot(knn_roc_39["FPR"], logistic_roc["TPR"], label = "KNN-39",color = "green",alpha = 0.7)
plt.plot(nb_roc["FPR"], logistic_roc["TPR"], label = "Naive Bayes",color = "blue",alpha = 0.7)
plt.plot(logistic_roc["FPR"], logistic_roc["TPR"], label = "Logistic",color = "red",alpha = 0.7)

plt.xlabel("FPR")
plt.ylabel("TPR")
plt.legend()
plt.title("ROC Curve for Test Set")
plt.show()

### CLASSIFICATION COSTS ###

def classification_cost(labels,scores,threshold):

    scores_prediction = np.array([1 if i > threshold else 0 for i in scores])

    TP = np.sum((labels == 1) & (scores_prediction == 1))
    TN = np.sum((labels == 0) & (scores_prediction == 0))
    FP = np.sum((labels == 0) & (scores_prediction == 1))
    FN = np.sum((labels == 1) & (scores_prediction == 0))

    return 100 * FN + 5 * FP

threshold_range = np.arange(0,1,0.001)

logistic_cost_curve = [classification_cost(y_test.ravel(),logistic_test_probs[:, -1],i)
for i in threshold_range]
knn_cost_curve = [classification_cost(y_test.ravel(),knn_test_probs[:, -1],i)
for i in threshold_range]
knn_39_cost_curve = [classification_cost(y_test.ravel(),knn_39_test_probs[:, -1],i)
for i in threshold_range]
nb_cost_curve = [classification_cost(y_test.ravel(),nb_test_probs[:, -1],i)
for i in threshold_range]


plt.plot(threshold_range, knn_cost_curve, label = "KNN-3", color = "purple")
plt.plot(threshold_range, knn_39_cost_curve, label = "KNN-39",color = "green")
plt.plot(threshold_range, nb_cost_curve, label = "Naive Bayes", color = "blue")
plt.plot(threshold_range, logistic_cost_curve, label = "Logistic", color = "red")
plt.xlabel("Threshold")
plt.ylabel("Misclassification Cost")
plt.ylim((0,4500))
plt.legend()
plt.show()


### 10-fold cross validation version ###

def CV_Kfold_misclassification(X,y,fold,model):
    CV = np.zeros([fold,threshold_range.shape[0]])
    for i in range(fold):
        split = KFold(n_splits=10)
        idx = list(split.split(X))
        X_train = X[idx[i][0],:]
        X_test = X[idx[i][1],:]
        y_train = y[idx[i][0]]
        y_test = y[idx[i][1]]
        #model
        mod = model
        mod.fit(X_train, y_train.ravel())
        mod_test_probs = mod.predict_proba(X_test)
        CV[i,:] = [classification_cost(y_test.ravel(),mod_test_probs[:, -1],i)
        for i in threshold_range]
    #return the mean and std of the curve
    return np.mean(CV,axis = 0)

logistic_cost_curve_CV  = CV_Kfold_misclassification(X_train,y_train,10,LogisticRegression())
knn_cost_curve_CV = CV_Kfold_misclassification(X_train,y_train,10,
KNeighborsClassifier(n_neighbors=3))
knn_39_cost_curve_CV = CV_Kfold_misclassification(X_train,y_train,10,
KNeighborsClassifier(n_neighbors=39))
nb_cost_curve_CV = CV_Kfold_misclassification(X_train,y_train,10,GaussianNB())


plt.plot(threshold_range,knn_cost_curve_CV, label = "KNN-3",color = "purple")
plt.plot(threshold_range,knn_39_cost_curve_CV, label = "KNN-39",color = "green" )
plt.plot(threshold_range,nb_cost_curve_CV, label = "Naive Bayes", color = "blue"  )
plt.plot(threshold_range,logistic_cost_curve_CV , label = "Logistic", color = "red")

plt.xlabel("Threshold")
plt.ylabel("Misclassification Cost")
plt.legend()
plt.show()
```