Machine Learning Assignment 1

Kyriacos Xanthos
CID: 01389741

February 3, 2022

Question 1

In this question we are given a dataset of the form $\mathcal{D} = \{x_i, z_i, y_i\}_{i=1}^{100}$ and we consider a model of the form:

$$Y_i = aZ_i + (b_1 + b_2 Z_i)\cos(X_i) + (c_1 + c_2 Z_i) X_i + (d_1 + d_2 Z_i) X_i^2 + (e_1 + e_2 Z_i) X_i^3 + \epsilon_i \quad (1)$$

where ϵ_i are independent random variables with mean 0.

1.

In this part we consider ridge regression. Note that ridge regression is an extension to linear regression by introducing an l_2 penalty on the coefficients to a standard least squares problem. Mathematically we are trying to solve the optimization problem:

$$\underset{\boldsymbol{\beta}}{\operatorname{argmin}} \|\boldsymbol{y} - \boldsymbol{X\beta}\|^2 + \lambda\left(\boldsymbol{\beta}^T\boldsymbol{\beta}\right) \quad (2)$$

where in our example $\boldsymbol{\beta} = \begin{bmatrix} a & b_1 & b_2 & c_1 & c_2 & d_1 & d_2 & e_1 & e_2 \end{bmatrix}^T$,$\lambda$ is the regularization parameter, and $\boldsymbol{y}$ and $\boldsymbol{X}$ are defined as:

$$\boldsymbol{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \quad , \quad \boldsymbol{X}^T = \begin{bmatrix} Z_1 & Z_2 & \dots & Z_n \\ \cos(X_1) & \cos(X_2) & \dots & \cos(X_n) \\ Z_1\cos(X_1) & Z_2\cos(X_2) & \dots & Z_n\cos(X_n) \\ X_1 & X_2 & \dots & X_n \\ Z_1X_1 & Z_2X_2 & \dots & Z_nX_n \\ X_1^2 & X_2^2 & \dots & X_n^2 \\ Z_1X_1^2 & Z_2X_2^2 & \dots & Z_nX_n^2 \\ X_1^3 & X_2^3 & \dots & X_n^3 \\ Z_1X_1^3 & Z_2X_2^3 & \dots & Z_nX_n^3 \end{bmatrix}$$

where n is 100 for our dataset.

We will use the function `Ridge` from the package `sklearn` in python which solves equation 2 and estimates the coefficients in the following form:

$$\hat{\beta} = \left(\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I}\right)^{-1}\mathbf{X}^T\mathbf{y} \quad (3)$$

We set the `fit_intercept` parameter to `False` so that it does not include an intercept and use the `svd` solver which performs singular value Decomposition at a low computational cost. We will assess our model performance by using the root mean squared error, defined as:

$$\text{rmse} = \left(\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2\right)^{1/2} \quad (4)$$

where $\hat{\boldsymbol{y}} = \boldsymbol{X\beta}$. With `sklearn` this is calculated using the function `mean_squared_error` with the argument `squared=False`.

λ is actually a hyperparameter and we need to tune it so that we get the optimum value. To tune it we use cross validation. The idea is that we split the data $\mathcal{D}$ randomly, into K-folds. We then fit the model K times, we validate on k^{th} fold and train on the remaining $K-1$ folds. By averaging across multiple folds of the data we decrease the variability in the model fit and validation error caused by the randomness of the splits. Figure 1 shows how the model RMSE varies for different parameters λ and for $K = 5$ folds. It is clear that many of the folds have not reached a minimum and suggest that the minimum is less than 0. However, we cannot have $\lambda < 0$ because then the matrix $\left(\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I}\right)^{-1}$ would not be invertible.

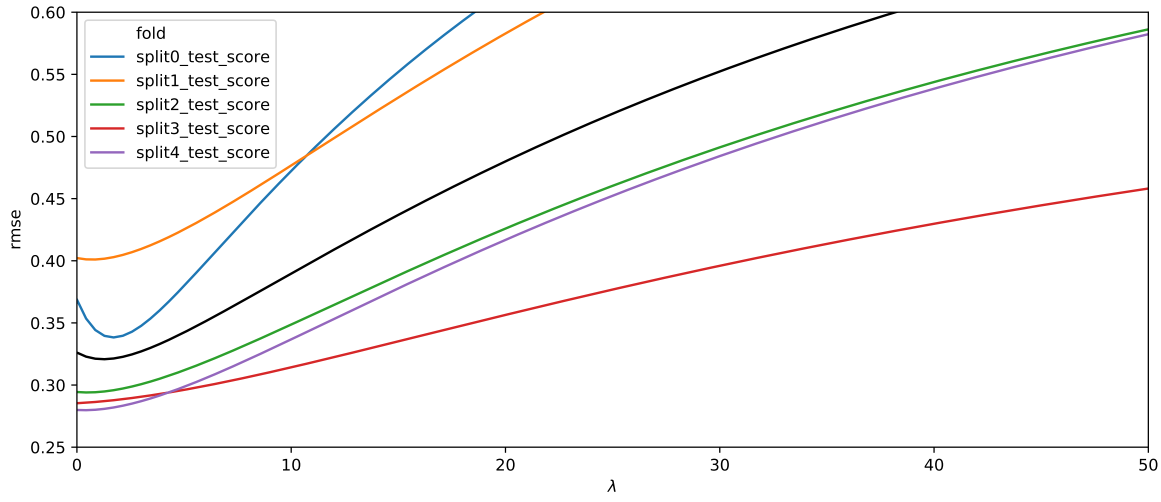

Figure 1: RMSE as defined in 4 against different values of hyperparameter λ for different folds.

We utilise cross validation using the `GridSearchCV` function from `sklearn` to search for the optimum value of λ that minimises the RMSE. This value is found to be $\lambda = 1.2156$ and looking at figure 2 we see that it indeed minimises the RMSE. We interprete this result with a grain of salt since the splits in figure 1 did not show a clear minimum before zero but this might be because our dataset only has 100 values and so the splits just consist of 20 points.

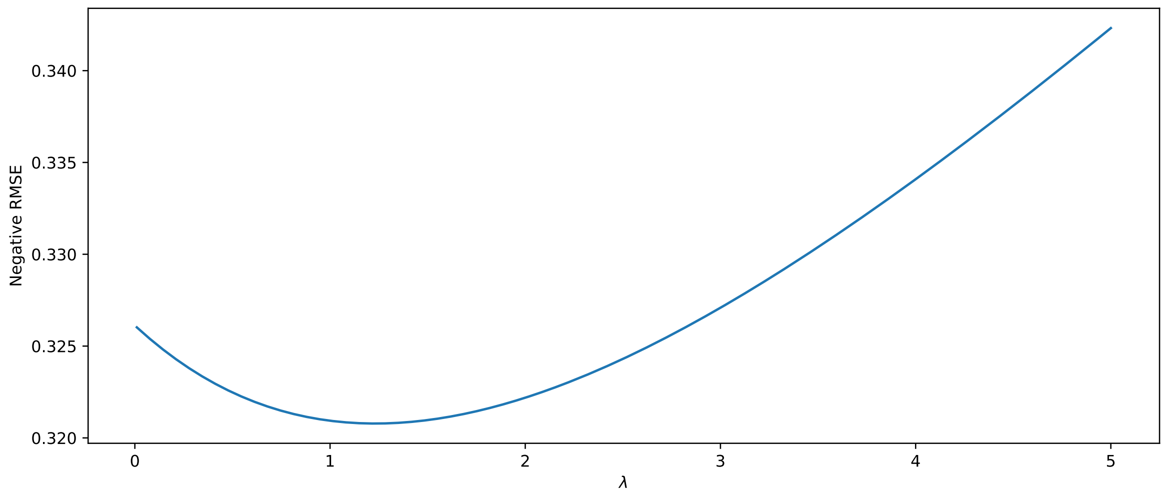

Figure 2: RMSE as defined in 4 against different values of hyperparameter λ.

Using this optimum value of λ we can re-run the Ridge regression and obtain the model summary in figure 3. The model seems to predict the true y with a high accuracy, with RMSE $= 0.286$ and the residual plot shows that there is not any particular bias. The coefficients are found to be:

$$\hat{\beta} = \begin{bmatrix} 0.0159 & 0.9993 & 0.8435 & 0.0242 & 0.01536 & -0.0006 & 0.1834 & 0.01453 & -0.0006 \end{bmatrix}^T$$

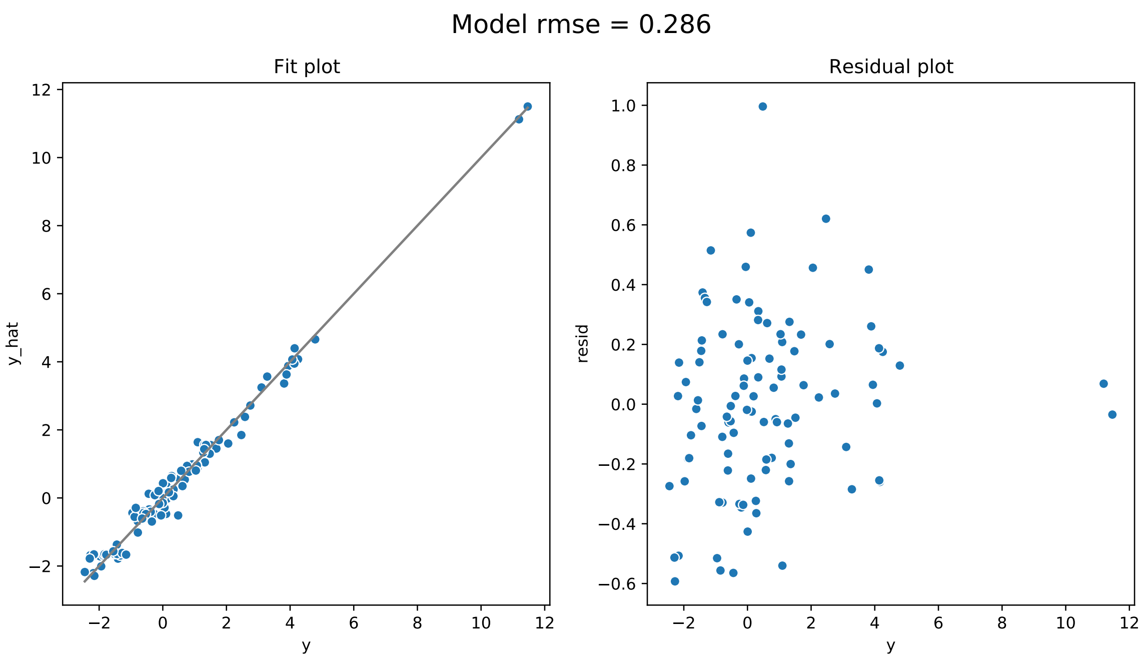

Figure 3: Summary of Ridge regression model with $\lambda = 1.2156$. Comparison of predicted y values with true y values (left) and residual plot (right)

Now we try to predict the value of Y^* for $X^* \in \{-7, -6.9, -6.8, \ldots 6.8, 6.9, 7\}$ and $Z^* \in \{0, 1\}$. The equation we use to predict this is $Y^* = X^*\hat{\beta}$ where $\hat{\beta}$ are the model parameter from our fit above.

We can see the fit of these two models in figure 4. It is clear that our model performs well for both z values. The fit seems to follow the trend of all points and even captures the few points which have higher values of y. This confirms a good model qualitatively on unseen data.

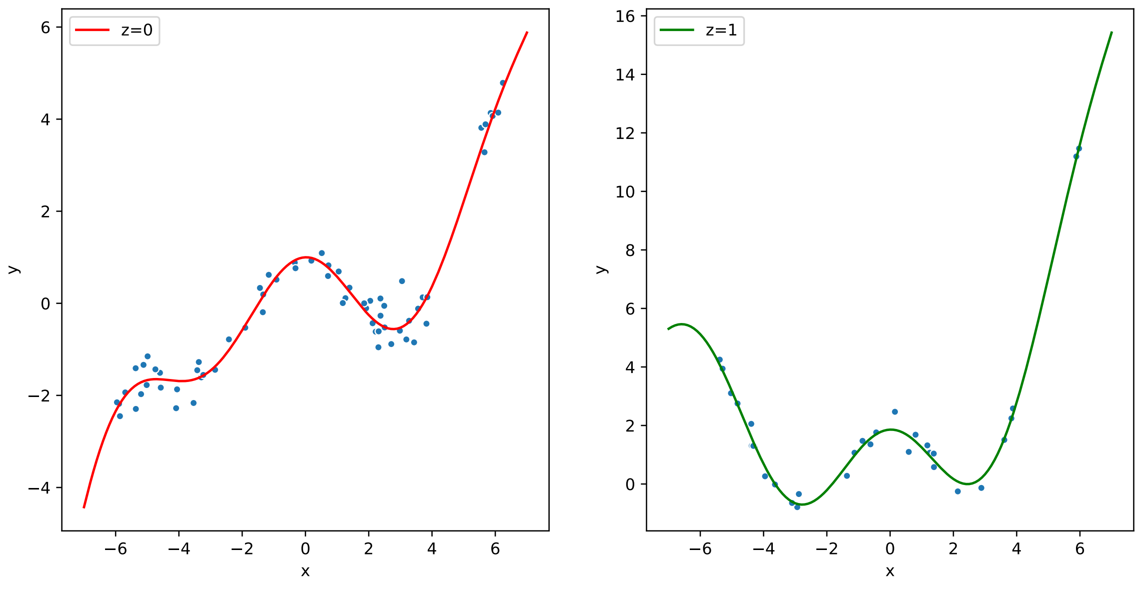

Figure 4: Fit of the model using Ridge regression on unseen data for $z = 0$ (left) and $z = 1$ (right).

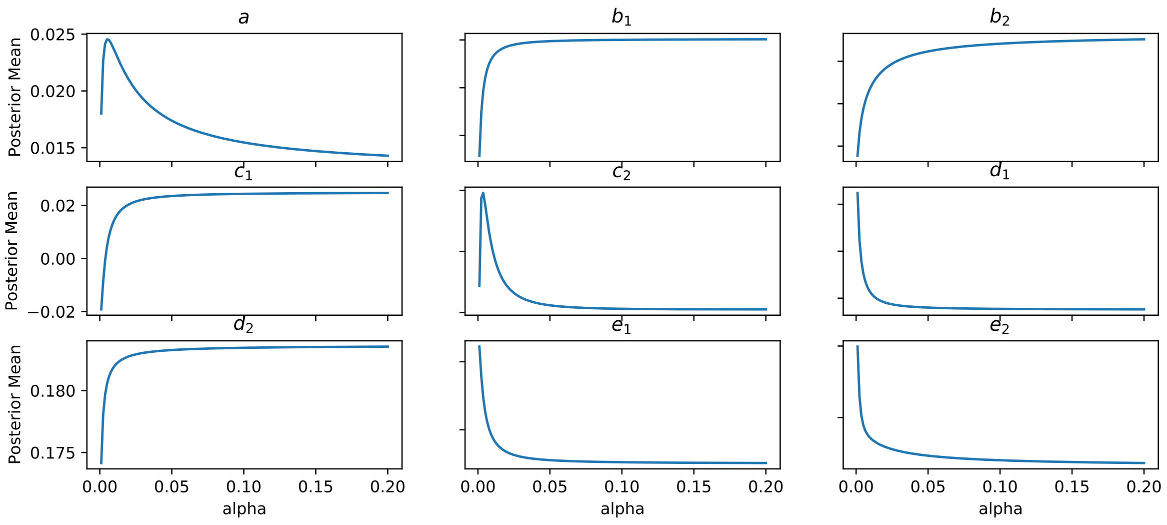

Figure 5: Posterior mean of 9 model parameters as a function of α

2.

We now perform Bayesian inference to determine the same parameters of this problem. The fundamental idea is that these parameters cannot be considered as single values, instead they acquire probability distributions. We are therefore interested in the posterior distribution over the parameters $\boldsymbol{\beta}$,

$$p(\boldsymbol{\beta} \mid \mathbf{X}, \mathbf{y}, \sigma) = \frac{p(\mathbf{y} \mid \boldsymbol{\beta}, \mathbf{X}, \sigma) p(\boldsymbol{\beta})}{p(\mathbf{y} \mid \mathbf{X}, \sigma)} \tag{5}$$

where $\sigma^2 = 0.01$ and the marginal likelihood is written as $p(\mathbf{y} \mid \mathbf{X}, \sigma) = \int p(\mathbf{y} \mid \boldsymbol{\beta}, \mathbf{X}, \sigma) p(\boldsymbol{\beta}) d\boldsymbol{\beta}$. We will consider a Gaussian prior of the form $p(\boldsymbol{\beta}) = \mathcal{N}_{\boldsymbol{\beta}}(\mathbf{0}, \Lambda)$, where $\Lambda = \alpha \boldsymbol{I}$, the multivariate normal centred at zero and with scaled identity covariance which encodes the belief that the parameters should be small which means that α is a parameter which we choose.

This is solved by [1]

$$\begin{aligned} p(\boldsymbol{\beta} \mid \mathbf{X}, \mathbf{y}, \sigma) &= \frac{\mathcal{N}_{\mathbf{y}}(\mathbf{x}\beta, \sigma \mathbf{I}) \mathcal{N}_{\beta}(\mathbf{0}, \boldsymbol{\Lambda})}{\int \mathcal{N}_{\mathbf{y}}(\mathbf{X}\beta, \sigma \mathbf{I}) \mathcal{N}_{\boldsymbol{\beta}}(\mathbf{0}, \boldsymbol{\Lambda}) d\boldsymbol{\beta}} \\ &= \mathcal{N}_{\boldsymbol{\beta}}(\boldsymbol{\mu}, \boldsymbol{\Sigma}) \end{aligned} \tag{6}$$

where

$$\boldsymbol{\mu} = \left(\mathbf{X}^T \mathbf{X} + \frac{\sigma^2}{\alpha} \mathbf{I} \right)^{-1} \mathbf{X}^T \mathbf{y} \quad , \quad \boldsymbol{\Sigma} = \sigma^2 \left(\mathbf{X}^T \mathbf{X} + \frac{\sigma^2}{\alpha} \mathbf{I} \right)^{-1} \tag{7}$$

Calculating 6 in `python` we can find the probability distribution of each parameter. Plotting the posterior mean of the 9 model parameters as a function of α we obtain figure 5.

It is clear that as the value of α increases, all of the 9 parameters converge either to a maximum or a minimum value. This means that for any $\alpha > 0.2$ we can be confident that the posterior mean of the parameter is converged.

To calculate the predictive posterior distribution we use

$$\begin{aligned} p\left(\mathbf{y}_{\text{new}} \mid \mathbf{x}_{\text{new}}, \mathbf{X}, \mathbf{y}, \sigma\right) &= \int p\left(\mathbf{y}_{\text{new}} \mid \mathbf{x}_{\text{new}}, \boldsymbol{\beta}, \sigma\right) p(\boldsymbol{\beta} \mid \mathbf{X}, \mathbf{y}, \sigma) d\boldsymbol{\beta} \\ &= \mathcal{N}_{\mathbf{y}_{\text{new}}}\left(\mathbf{x}_{\text{new}}^T \boldsymbol{\mu}, \mathbf{x}_{\text{new}}^T \boldsymbol{\Sigma} \mathbf{x}_{\text{new}} + \sigma^2\right) \end{aligned} \tag{8}$$

as stated in the notes [1]. We again use the same new values as above, namely $X^* \in \{-7, -6.9, -6.8, \ldots 6.8, 6.9, 7\}$ and $Z^* \in \{0, 1\}$ to obtain the new matrix $\boldsymbol{x}_{\text{new}}$. We then calculate the posterior mean and its 95% credible interval. Credible intervals are analogous to the confidence intervals in frequentist statistics but for the bayesian interpretation. They determine the distribution of possible values of the posterior mean [2]. We calculate the

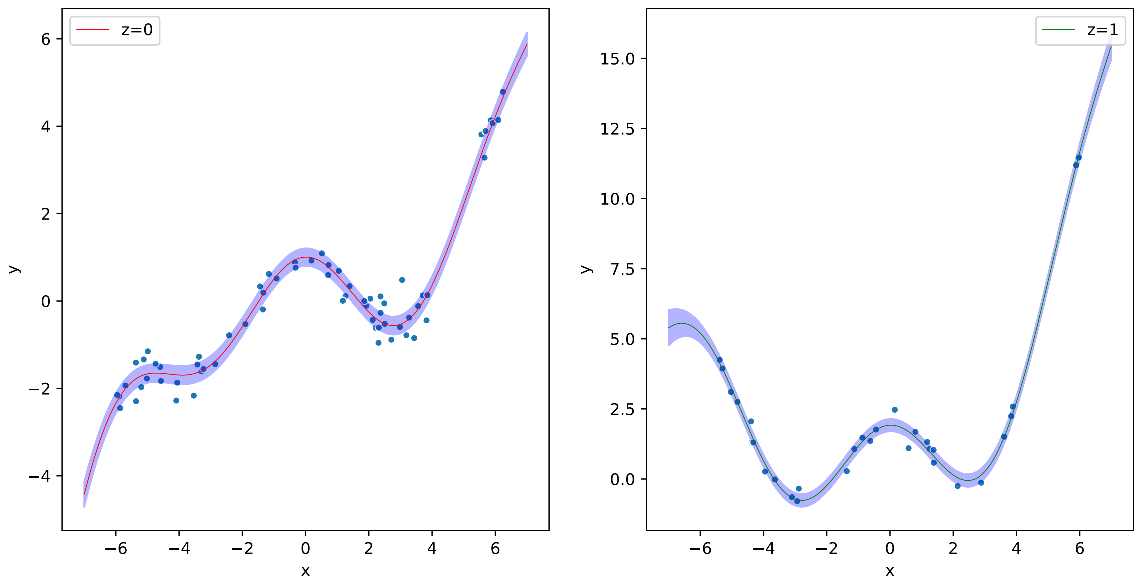

Figure 6: Mean of Posterior Predictive Distribution as a function of X with $z = 0$ (left) and $z = 1$ (right). The shaded blue shows the 95% credible interval of the computation.

credible interval using the 2.5% and 97.5% quantiles of the Gaussian distribution with the parameters stated in equation 6. The plots for $z = 0$ and $z = 1$ are shown in figure 6.

The fit of the model seems to be very good for both values of z. The 95% credible interval shows that it covers most of the points in our data except some of the points that exhibit local maxima and local minima. If the range of the credible interval was increased to 90% then the interval would cover all points.

3.

Now we will make predictions on $(X, Z) \in \{-5, 7\} \times \{0, 1\}$. We will use 3 different values of α for our predictions. The first one consists of understanding the the regularised ridge least squares problem is exactly the maximum a posteriori estimate of the parameters β with the regularisation parameter $\lambda = \sigma^2/\alpha$. So then using the optimum λ found in part 1, $\alpha = 0.1/1.2156 = 0.0823$. The second α will be a value much smaller which would mean that the parameters in figure 5 would not have converged (here we use $\alpha = 0.001$) and the third value would be a much larger value that ensures that the posterior mean of the parameters has converged ($\alpha = 1$).

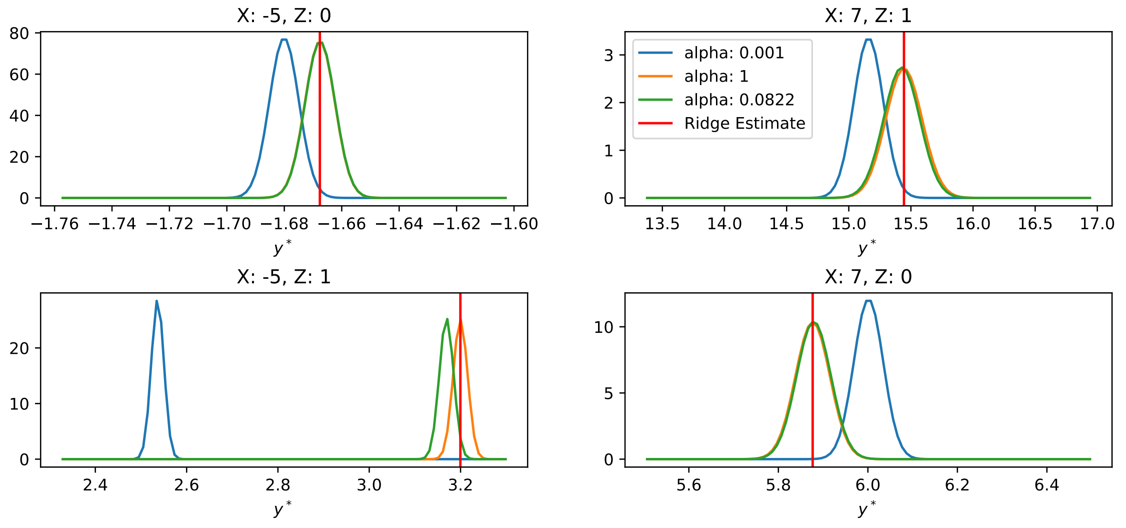

Figure 7: PDF of Predictive distributions for different observation points.

Looking at figure 7 we can observe that the posterior distributions for $\alpha = 0.0822, 1$ are very close for all 4 observations and this makes sense because as observed in figure 5 our parameters seemed to almost converge after 0.1. For $\alpha = 0.001$ as expected in all 4 observations the posterior distribution is far away from the other 2 values especially for $X = -5, Z = 1$ which shows that the prediction would not be accurate. Indeed, using the relationship with the λ of the ridge regression we see that a very small value of α means there is heavy regularisation in the problem which was not the case when we searched for the optimum λ using cross validation. The red vertical line shows that the predictions using Bayesian inference are almost equivalent to the Ridge regression problem, as expected.

Question 2

We now consider a slightly modified version of the Million Song Dataset [3]. For each song in the dataset there are 91 attributes with the first one being the year it was published. We will create models that predict if a song was produced before or after 1980. This means we will change the first attribute to a binary variable for each song, namely it will be a 0 if it was produced before 1980 and 1 if it was produced after. We can see the distribution of songs for each year in figure 8.

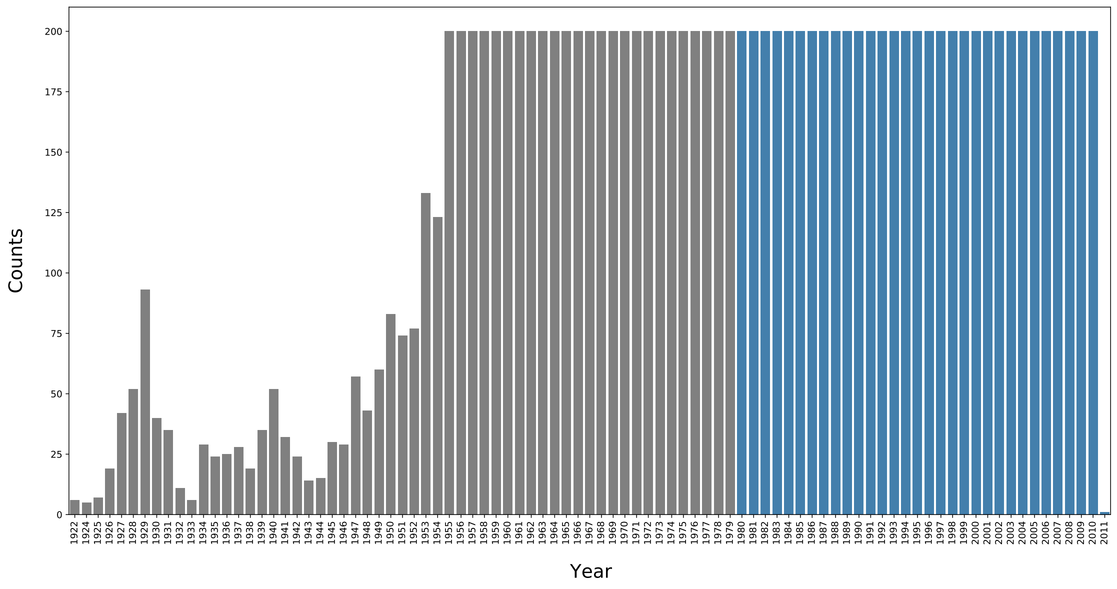

Figure 8: Distribution of songs produced each year. Grey are all the songs before 1980 and blue all the songs after.

We can observe that the number of songs after 1955 has been right censored to 200 counts. This was done in order to have roughly the same number of songs before and after 1980. Indeed when we count them, we have a dataset with 6201 songs after 1980 and 6322 songs before, a balanced dataset.

Since we have 90 attributes, we cannot visualise the attributes to explore some of their features. We note that we have no missing data, and therefore we can right away use classification. In order to be able to assess the efficiency of our classifiers, we split the data to training and testing using the function `train_test_split` from `sklearn` which shuffles the data and then randomly splits it in training and testing subsets. Since we have a large dataset we use a test size of 30% to reduce the computational cost in training. We then use the `StandardScaler` function of `sklern` to scale all of the attributes of the songs so that they have mean 0 and variance 1. We do that so that the effect of each attribute will be captured in our model and we make sure to do the scaling after the split of train and testing sets.

LDA

The Linear Discriminant Analysis (LDA) classifier partitions our data into regions with the same class predictions via separating hyperplanes. This is a generative approach that models the class conditional densities $p(x \mid y = k)$ and class priors $p(y = k)$. Then using Bayes Theorem we compute the posterior probabilities

$$p(y = k \mid x) = \frac{p(x \mid y = k)p(y = k)}{p(x)} \tag{9}$$

where $p(x) = \sum_{j=1}^{K} p(x \mid y = j)p(y = j)$ [1].
Then it assumes that the class-conditional densities are Gaussian,

$$x \mid y = k \sim \mathcal{N}(\mu_k, \Sigma), \quad \text{for } 1 \le k \le K \tag{10}$$

where Σ is the covariance matrix and μ_k are the means. Then to create the boundaries we use parameter estimation using Maximum Likelihood. These estimates are

$$\begin{aligned} \hat{\pi}_k &= \frac{N_k}{N} \\ \hat{\mu}_k &= \frac{1}{N_k} \sum_{i, y^{(i)}=k} x^{(i)} \\ \hat{\Sigma} &= \frac{1}{N} \sum_{k=1}^{K} \sum_{i, y^{(i)}=k} \left(x^{(i)} - \mu_k\right)\left(x^{(i)} - \mu_k\right)^T \end{aligned} \tag{11}$$

where N_k are the number of is such that $y^{(i)} = k$, N are the total number of points.

To calculate this we use the `LinearDiscriminantAnalysis` function of `sklearn`. It has a linear decision boundary which is generated by fitting Gaussian class conditional densities to the data assuming that all classes share the same covariance matrix [4]. We assume no shrinkage, and use the `svd` solver which performs singular value decomposition. We could also use the `lsqr` method which uses the dedicated least squares routine and it is the fastest solver but it produced less accurate results so we preferred `svd`.

We then fit this solver to our training data and make predictions on our testing data. A common metric for the performance of our classifier is a confusion matrix [5] which summarises the correct and false classifications.

	FALSE	TRUE
FALSE	1470	428
TRUE	475	1384

	FALSE	TRUE
FALSE	1478	459
TRUE	479	1341

	FALSE	TRUE
FALSE	994	910
TRUE	355	1498

Table 1: Confusion Matrix of Test Set (30%) using LDA (left), KNN (middle) and Naive Bayes (right)

As seen in table 1 our classifier was able to classify 76% of the songs correctly if they were before or after 1980 (Accuracy) with error rate 24%. Another useful metric for the performance of classifiers are the Receiver Operatic Characteristic Curves (ROC) [6]. They illustrate the diagnostic ability of a binary classifier as its discrimination threshold is varied. We plot the True Positive Rate

$$\text{TPR} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Negative}}$$

against the False Negative Rate ($\text{FNR} = 1 - \text{TPR}$)[1]. The perfect classifier would exhibit a vertical line to $\text{TPR} = 1$ and then a horizontal line to $\text{FPR} = 1$. We can see the ROC Curve of our classifier in figure 10 and it clear that it follows the pattern we expect with the curve peaking relatively close to to the (0,1) point.

A quantitative metric that can judge the performance of our classifier is the AUC (Area Under the Curve) of an ROC curve. This area under the curve is equal to the probability that the classifier will rank a randomly chosen positive instance higher than a randomly chosen negative one [7]. The LDA classifier achieves an 84.6 % AUC score for the Training set and 83.3% for the Test set. These are very good scores for a dataset in its high dimensional nature.

[1]True Positive is when a test result correctly indicates the presence of a condition. False Negative is a test result which wrongly indicates that a particular condition is absent.

KNN

K-nearest neighbours (KNN) is a discriminative approach which aims to model the conditional probability $p(y = k \mid x)$ directly. This method measures distances between each data point and the number of nearest neighbours. The pseudo algorithm for this method is found in Algorithm 1.

Algorithm 1 KNN algorithm [8]

```
Input: $(X_i, C_i)$ where $i = 1, \dots n$ be $n$ data points, $X_i$ denotes feture values and $C_i$ denotes
       labels for each $i$. Integer $\boldsymbol{k}$.
for $i$=1,2,... do
    Calculate $\boldsymbol{d}(x, x_i)$ where $\boldsymbol{d}$ is the Euclidean distance
end
Arrange $\boldsymbol{d}$ in increasing order
 Take first $\boldsymbol{k}$ distances from the sorted list
 Find those $\boldsymbol{k}$-points corresponding to these $\boldsymbol{k}$-distances
 Let $\boldsymbol{k}_i$ be the number of points in the $i^{\text{th}}$ class among $\boldsymbol{k}$ points

for $i$=1,2,... do
    for $j$=1,2,... do
        if $\boldsymbol{k}_i > \boldsymbol{k}_j$ for all $i \neq j$ then
            put x in class i
        end
    end
end
Return: Data points in different classes
```

To compute the KNN we use the `KNeighborsClassifier` function from `sklearn` which implements learning based on the k nearest neighbors of each query point. We choose the Euclidean distance as the metric since it provides the least error rate. The function uses uniform weights (all points i each neighbourhood are weighted equally). The algorithm used to compute the nearest neighbours was `ball_tree` with leaf size 30. This is a similar algorithm to 1, more about the algorithm can be found in [9]. The leaf size is 30 which is a good compromise between the computational time and amount of nodes created.

In order to tune the hyperparameter k, we try different k-values and for each value we calculate the mean accuracy rate (the times the predicted label matches the true label). To do this we again use cross validation with 5 folds as described in question 1. Using the function `GridSearchCV` we were able to plot the mean accuracy for each value of k for $k \in \{5, 10, 15, 20, 25, 30, 35, 40, 45, 50\}$ as seen in figure 9. We see that the accuracy stabilizes after $k = 25$ so we choose this value to avoid underfitting (decision boundary is closer to being linear as k increases). A smaller value would result in the decision boundary to be overly flexible and find patterns inn the data that do not correspond to the Bayes decision boundary (overfitting).

We can asses the success of this classifier again by looking at the confusion matrix, error rate, ROC curve and AUC score. The confusion matrix can be seen in table 1. The error rate is 25%, 1% larger than the LDA classifier. The ROC curve seems to be slightly better than the LDA, with an AUC of 81.9% for the test set. For a model with only 12 nearest neighbours, these are very good statistics for the classifier.

Naive Bayes Classifier

The Naive Bayes classifier is a generative approach and it assumes all measured variables and features are independent given the label. The clasifer makes the assumption that the features are conditionally independent for the class label:

$$p(x \mid y = k) = \prod_{i=1}^{d} p\left(x_i \mid y = k, \beta_{ik}\right) \tag{12}$$

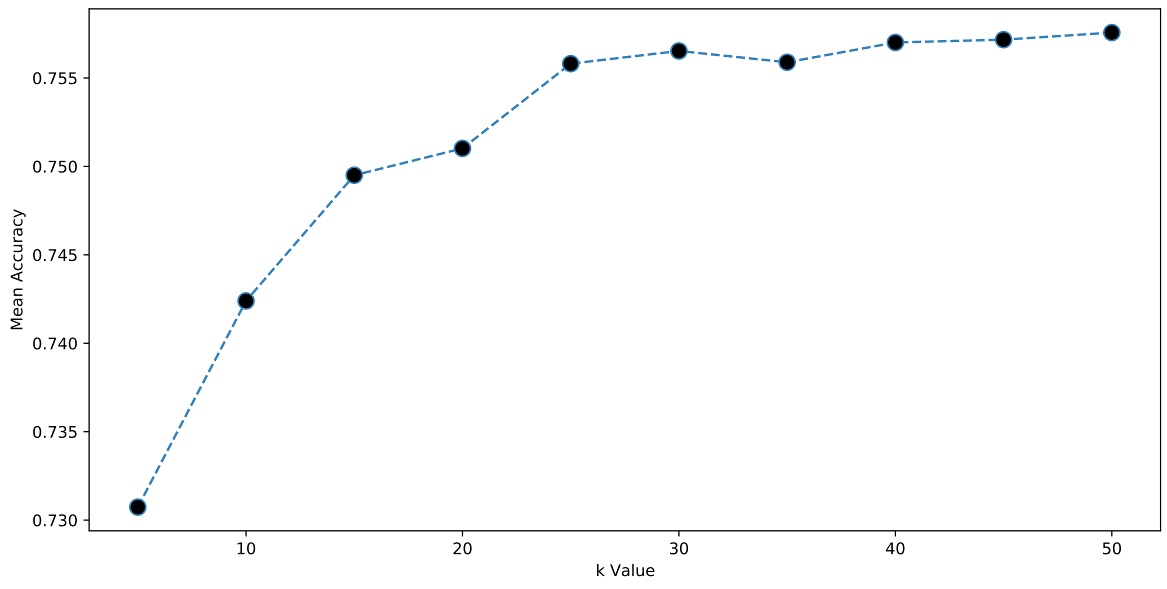

Figure 9: Mean Accurary against number of nearest neighbours for KNN model.

Since in this dataset we have real-valued features we decided to use the Gaussian distribution, ie:

$$p\left(x \mid y=k', \theta\right)=\prod_{j=1}^{d} \Phi\left(x_{j} \mid \mu_{jk}, \sigma_{jk}^{2}\right) \tag{13}$$

where Φ is the probability density function of the normal distribution.

We then use the Maximum Likelihood approach for parameter estimation, with the probability for a single data point x^i, y^i given by:

$$p\left(x^{(i)}, y^{(i)} \mid \theta, \pi\right)=p\left(y^{(i)} \mid \pi\right) \prod_{j=1}^{d} p\left(x_{j}^{(i)} \mid \theta_{j}\right)=\prod_{k=1}^{K} \pi_{k}^{1\left(y^{(i)}=k\right)} \prod_{j=1}^{d} \prod_{k=1}^{K} p\left(x_{j}^{(i)} \mid \theta_{jk}\right)^{1\left(y^{(i)}=k\right)} \tag{14}$$

where π_k is the prior probability that $y^{(i)}=k$.

To fit the Naive Bayes classifier we will use the `GaussianNB` function from `sklearn` which implements the Gaussian Naive Bayes algorithm mentioned above. Fitting on the training set and testing on the test set we observe that the confusion matrix is much worse than the other two classifiers, with an error rate of 33%. This is mostly due to the number of instances the true label was 0 (before 1980) and was misclassified by 1 (after 1980). Looking at the ROC curve in figure 10 we see a slight decrease in the peak with comparison of the other curves. The AUC score for the test set was just at 75.6%, not bad in general but worse than our other two classifiers.

Conclusion

All three classifiers are able to classify songs before and after 1980 fairly good. By the nature of the problem, there is no hard decision boundary at 1980 that differentiates music in a fundamental way. That is to say, we can expect that songs of 1982 for example do not defer a lot with 1978. This suggests that an extension to the study would be to create softer margins, and group maybe decades together to see clear difference between the classes. Nevertheless, since there is a clear difference of the music style for songs of the 40s, 50s, 60s and 70s with everything after the 90s, we achieved very good classifiers that can predict the age of a song. Right-censoring the data so that we have roughly equal number of songs before and after the decision boundary really helped with the classification problem and meant that we can use the underlying assumptions of LDA and Naive Bayes. When trying to decide between the three classifiers we can refer to figure 11 to test that LDA seems to have the best ROC Curve with the highest AUC followed by KNN and then by Naive Bayes.

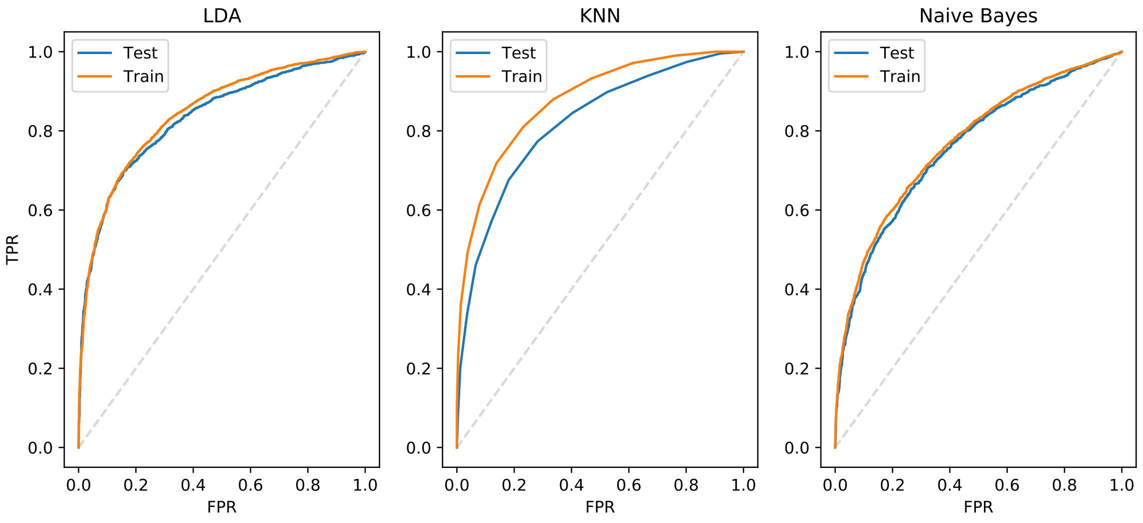

Figure 10: ROC curves of LDA (left), KNN (right) and Naive Bayes (right) for both Test and Training sets (30-70 % split)

Classifier	LDA	KNN	NB
Computation time	0.5s	5.7s	0.4s

Table 2: Computational time of the fitting of all three classifiers for the same training subset of the data. The measurements were made on the same computer, with minimal background processes at the fitting procedure.

KNN was the only classifier which we could tune hyperparameters, which suggest that the other two models cannot be improved to a large extent, but we also believe we have found a good hyperparameter k for the KNN that does not overfit our data. Comparing the computational times of the three classifiers in table 2 it is clear that the KNN is the slowest and LDA and Naive Bayes to be almost identical with Naive Bayes having the edge. This makes sense since Naive Bayes makes the (crude) assumption of independent features given a label. We did not challenge this assumption because of how many features we had, but computing the correlation matrix did not show any clear correlation with the labels.

The KNN iteratively goes through every point and assignns it to a classs which is computationally extensive. The ball tree algorithm does not work very well with high dimensions since the split of the tree at each node is based on all of the labels of our dataset. LDA works very well with high dimensional data.

In conclusion, the LDA seems to be the best classifier, satisfying both accuracy, computational cost, and clear interpretability of results.

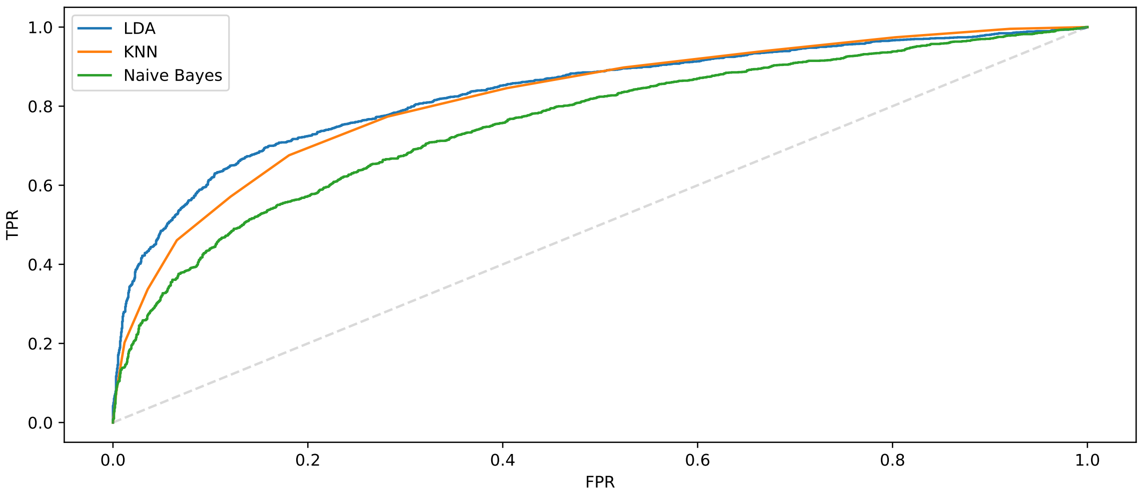

Figure 11: ROC curves of Test sets of all 3 classifiers (LDA, KNN, Naive Bayes)

References

[1] Dr Sarah Filippi. Machine learning lecture notes. *Imperial College London*, pages 30–40, October 2021.

[2] Richard D Morey, Rink Hoekstra, Jeffrey N Rouder, Michael D Lee, and Eric-Jan Wagenmakers. The fallacy of placing confidence in confidence intervals. *Psychonomic bulletin & review*, 23(1):103–123, 02 2016. doi: 10.3758/s13423-015-0947-8. URL https://pubmed.ncbi.nlm.nih.gov/26450628.

[3] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017. URL http://archive.ics.uci.edu/ml.

[4] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[5] Wikipedia. Confusion matrix — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Confusion%20matrix&oldid=1058352752, 2022. [Online; accessed 29-January-2022].

[6] Wikipedia. Receiver operating characteristic — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Receiver%20operating%20characteristic&oldid=1067917334, 2022. [Online; accessed 29-January-2022].

[7] Fawcett, Tom. An introduction to roc analysis. 2006. URL https://www.math.ucdavis.edu/~saito/data/roc/fawcett-roc.pdf.

[8] Rahul Saxena. Knn classifier, introduction to k-nearest neighbor algorithm. https://dataaspirant.com/k-nearest-neighbor-classifier-intro/, 2022. [Online; accessed 29-January-2022].

[9] Stephen M. Omohundro. Five balltree construction algorithms. Technical report, 1989.

A Code for Question 1

```python
# %%
import pandas as pd
import numpy as np
import os
import matplotlib.pyplot as plt
import seaborn as sns
# from sklearn.preprocessing import PolynomialFeatures
# from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import GridSearchCV, KFold
# from sklearn.preprocessing import LabelEncoder   # binary encoding
from sklearn.model_selection import train_test_split
# from sklearn.preprocessing import StandardScaler
# from sklearn.cluster import KMeans
import sklearn.pipeline
from sklearn.linear_model import Ridge
from sklearn.linear_model import RidgeCV
from scipy.stats import norm
seed = 222
np.random.seed(seed)

plt.rcParams['figure.figsize'] = (12,5)
plt.rcParams['figure.dpi'] = 80
```

```


# %%
d = pd.read_csv('data.csv')
d.describe()

# %%
sns.scatterplot(x='x', y='y', data = d, hue= 'z', s=20)
plt.savefig('Plots/my_data.pdf')
plt.show()

# %%
def create_design_matrix(x,z):

    X = np.vstack((z, np.cos(x), z * np.cos(x), x, z * x, x * x,
    z * x * x, x * x * x, z * x * x * x ))
    X = X.T
    return(X)

X =create_design_matrix(d.x, d.z)
y = d.y


# %%
def get_coefs(m):
    if (isinstance(m, sklearn.pipeline.Pipeline)):
        m = m.steps[-1][1]
    if m.intercept_ is None:
        return m.coef_
    return np.concatenate([[m.intercept_], m.coef_])


# %%
def model_fit(m, X, y, plot = False):

    y_hat = m.predict(X)
    rmse = mean_squared_error(y, y_hat, squared=False)
    res = pd.DataFrame(
            data = {'y': y, 'y_hat': y_hat, 'resid': y - y_hat}
    )
    if plot:
            plt.figure(figsize=(12, 6))
            plt.subplot(121)
            sns.lineplot(x='y', y='y_hat', color="grey", data = pd.DataFrame(dat
            sns.scatterplot(x='y', y='y_hat', data=res).set_title("Fit plot")
            plt.subplot(122)
            sns.scatterplot(x='y', y='resid', data=res).set_title("Residual plot
            plt.suptitle("Model rmse = " + str(round(rmse, 4)), fontsize=16)
            plt.savefig('Plots/model_summary.pdf')
            plt.show()
    return(rmse)


# %%
r = make_pipeline(
    Ridge(alpha=1)
).fit(X, y)

# %%
model_fit(r, X, y, plot=True)

# %%
alphas = np.linspace(0.01, 5, num=150)

```

```
gs = GridSearchCV(
    make_pipeline(
        Ridge()
        ),
        param_grid={'ridge__alpha': alphas},
        cv=KFold(5, shuffle=True, random_state=1234),
        scoring="neg_root_mean_squared_error"
).fit(X, y)

# %%
print(gs.best_params_)
model_fit(gs.best_estimator_, X, y, plot=True)
best_alpha = gs.best_params_['ridge__alpha']

# %%
cv_res = pd.DataFrame(
    data = gs.cv_results_
).filter(
    regex = '(split[0-9]+|mean)_test_score'
).assign(
# Add the alphas as a column
    alpha = alphas
)
cv_res.update(
# Convert negative rmses to positive
-1 * cv_res.filter(regex = '_test_score')
)
sns.lineplot(x='alpha', y='mean_test_score', data=cv_res)
plt.xlabel('$\lambda$')
plt.ylabel('Negative RMSE')
# plt.savefig('Plots/single_l.pdf')
plt.show()

# %%
d1 = cv_res.melt(
    id_vars=('alpha','mean_test_score'),
    var_name='fold',
    value_name='rmse'
)

sns.lineplot(x='alpha', y='rmse', color='black', ci = None, data = d1) # P l o t
sns.lineplot(x='alpha', y='rmse', hue='fold', data = d1) # Plot the curves f o r
plt.xlabel('$\lambda$')
# plt.savefig('Plots/splits.pdf')
plt.show()

# %%
m = make_pipeline(
    Ridge(alpha=best_alpha, fit_intercept = False, solver='svd' )
).fit(X, y)

# %%
model_fit(m, X, y, plot= True)

# %%
new_x = np.arange(-7, 7.1, step=0.1)

X_new1 = create_design_matrix(new_x, np.zeros(new_x.shape))
X_new2 = create_design_matrix(new_x, np.ones(new_x.shape))

# %%
y_hat1 = m.predict(X_new1)
y_hat2 = m.predict(X_new2)

```

```

# %%
plt.figure(figsize=(12, 6))
plt.subplot(121)
sns.scatterplot(x='x', y='y', data = d[d.z == 0], s=20)
plt.plot(new_x, y_hat1, color = 'red', label ='z=0')
plt.legend()
plt.subplot(122)
sns.scatterplot(x='x', y='y', data = d[d.z == 1], s=20)
plt.plot(new_x, y_hat2, color = 'green', label='z=1')
plt.legend()
# plt.savefig('Plots/two_models_ridge.pdf')
plt.show()

# %% [markdown]
# ## Part 2

# %%
sigma_sq = 0.1

# %%
# see equations page 31 of slides
def solve_posterior(
    X: np.ndarray, y: np.ndarray, sigma_sq: np.ndarray, alpha: np.ndarray
) -> np.ndarray:
    n = X.shape[0]
    p = X.shape[1]
    mu = np.linalg.solve(X.T @ X + sigma_sq / alpha * np.eye(p), X.T @ y)
    Sigma = sigma_sq * np.linalg.solve(
        X.T @ X + (sigma_sq / alpha) * np.eye(p), np.eye(p)
    )
    return mu, Sigma

# %%
alphas = np.linspace(0.001, 0.2, 150)
mus = np.empty((alphas.shape[0], 9))
for i, a in enumerate(alphas):
    mus[i] = solve_posterior(X, y, sigma_sq, a)[0]
mus.shape


# %%
my_lst = [(0,0), (0,1), (0,2), (1,0), (1, 1), (1,2), (2,0), (2,1), (2,2)]
all_par = ['$a$', '$b_1$', '$b_2$', '$c_1$', '$c_2$', '$d_1$', '$d_2$', '$e_1$',
fig, axs = plt.subplots(3, 3)
for i, (t, p) in enumerate(zip(my_lst, all_par)):
    axs[t].plot(alphas, mus[:,i])
    axs[t].set_title(p)

for ax in axs.flat:
    ax.set(xlabel='alpha', ylabel='Posterior Mean')

for ax in axs.flat:
    ax.label_outer()

# fig.savefig('Plots/all_par.pdf')

# %%
mu, sigma = solve_posterior(X, y, sigma_sq, 1)

# %%
pred_1 = X_new1 @ mu
pred_2 = X_new2 @ mu
pred_1_var = X_new1 @ sigma @ X_new1.T + sigma_sq
```

```
pred_1_var = X_new2 @ sigma @ X_new2.T + sigma_sq

# %%
plt.scatter(d.x,d.y)
plt.plot(new_x, pred_1, color = 'red')
plt.plot(new_x, pred_2, color = 'green')


# %%
cred_int_1 = np.empty((X_new1.shape[0],2))
for i in range(0, X_new1.shape[0]):
    big_sigma = X_new1[i] @ sigma @ X_new1[i].T + sigma_sq
    cred_int_1[i,0] = norm.ppf(0.025, pred_1[i], big_sigma)
    cred_int_1[i,1] = norm.ppf(0.975, pred_1[i], big_sigma)

cred_int_2 = np.empty((X_new2.shape[0],2))
for i in range(0, X_new2.shape[0]):
    big_sigma = X_new2[i] @ sigma @ X_new2[i].T + sigma_sq
    cred_int_2[i,0] = norm.ppf(0.025, pred_2[i], big_sigma)
    cred_int_2[i,1] = norm.ppf(0.975, pred_2[i], big_sigma)

# %%
plt.figure(figsize=(12, 6))
plt.subplot(121)
sns.scatterplot(x='x', y='y', data = d[d.z == 0], s=20)
plt.plot(new_x, pred_1, color = 'red', linewidth=0.5, label='z=0')
plt.fill_between(new_x, cred_int_1[:,0], cred_int_1[:,1], alpha=0.3, color='blue
plt.legend()
plt.subplot(122)
sns.scatterplot(x='x', y='y', data = d[d.z == 1], s=20)
plt.plot(new_x, pred_2, color = 'green', linewidth=0.5, label='z=1')
plt.fill_between(new_x, cred_int_2[:,0], cred_int_2[:,1], alpha=0.3, color='blue
plt.legend()
# plt.savefig('Plots/two_models_bayes.pdf')
plt.show()

# %%
X_trial = create_design_matrix(np.array([-5,7, -5, 7]), np.array([0,1, 1,0]))

# %%
pred_bayesian1 = X_trial @ mu

pred_ridge = m.predict(X_trial)

# %%
ridge_alpha = sigma_sq / best_alpha
ridge_alpha

# %%
def calc_all(i):
    new_mu = solve_posterior(X,y,sigma_sq,0.001)[0]
    new_sigma =  solve_posterior(X,y,sigma_sq,0.001)[1]
    mean_small = X_trial[i] @ new_mu
    sigma_small = X_trial[i] @ new_sigma @ X_trial[i].T
    new_mu = solve_posterior(X,y,sigma_sq,1)[0]
    new_sigma =  solve_posterior(X,y,sigma_sq,1)[1]
    mean_medium = X_trial[i] @ new_mu
    sigma_medium = X_trial[i] @ new_sigma @ X_trial[i].T
    new_mu = solve_posterior(X,y,sigma_sq,ridge_alpha)[0]
    new_sigma =  solve_posterior(X,y,sigma_sq,ridge_alpha)[1]
    mean_ridge = X_trial[i] @ new_mu
    sigma_ridge = X_trial[i] @ new_sigma @ X_trial[i].T
    return(mean_small, sigma_small, mean_medium, sigma_medium, mean_ridge, sigma

# %%
plt.subplots_adjust(hspace=0.5)

plt.subplot(221)
mean_small, sigma_small, mean_medium, sigma_medium, mean_ridge, sigma_ridge = ca
t1 = np.linspace(mean_small - 15*sigma_small, mean_small + 15*sigma_small, 100)
plt.plot(t1, norm.pdf(t1, mean_small, sigma_small),label='alpha: 0.001' )
plt.plot(t1, norm.pdf(t1, mean_medium, sigma_medium), label='alpha: 1')
plt.plot(t1, norm.pdf(t1, mean_ridge, sigma_ridge),label='alpha: 0.0822' )
plt.axvline(x=pred_bayesian1[0], color='red', label ='Ridge Estimate')
plt.xlabel('$y^*$')
plt.title('X: -5, Z: 0')
# plt.legend()

plt.subplot(222)
mean_small, sigma_small, mean_medium, sigma_medium, mean_ridge, sigma_ridge = ca
t1 = np.linspace(mean_small - 15*sigma_small, mean_small + 15*sigma_small, 100)
plt.plot(t1, norm.pdf(t1, mean_small, sigma_small),label='alpha: 0.001' )
plt.plot(t1, norm.pdf(t1, mean_medium, sigma_medium), label='alpha: 1')
plt.plot(t1, norm.pdf(t1, mean_ridge, sigma_ridge),label='alpha: 0.0822' )
plt.axvline(x=pred_bayesian1[1], color='red', label ='Ridge Estimate')
plt.title('X: 7, Z: 1')
plt.xlabel('$y^*$')

plt.legend()

plt.subplot(223)
mean_small, sigma_small, mean_medium, sigma_medium, mean_ridge, sigma_ridge = ca
t1 = np.linspace(mean_small - 15*sigma_small, mean_small + 55*sigma_small, 100)
plt.plot(t1, norm.pdf(t1, mean_small, sigma_small),label='alpha: 0.001' )
plt.plot(t1, norm.pdf(t1, mean_medium, sigma_medium), label='alpha: 1')
plt.plot(t1, norm.pdf(t1, mean_ridge, sigma_ridge),label='alpha: 0.0822' )
plt.axvline(x=pred_bayesian1[2], color='red', label ='Ridge Estimate')
plt.xlabel('$y^*$')

plt.title('X: -5, Z: 1')
# plt.legend()

plt.subplot(224)
mean_small, sigma_small, mean_medium, sigma_medium, mean_ridge, sigma_ridge = ca
t1 = np.linspace(mean_small - 15*sigma_small, mean_small + 15*sigma_small, 100)
plt.plot(t1, norm.pdf(t1, mean_small, sigma_small),label='alpha: 0.001' )
plt.plot(t1, norm.pdf(t1, mean_medium, sigma_medium), label='alpha: 1')
plt.plot(t1, norm.pdf(t1, mean_ridge, sigma_ridge),label='alpha: 0.0822' )
plt.axvline(x=pred_bayesian1[3], color='red', label ='Ridge Estimate')
plt.title('X: 7, Z: 0')
plt.xlabel('$y^*$')
# plt.legend()

# plt.savefig('Plots/post_all.pdf')


# %%
```

B Code for Question 2

```
# %%
import pandas as pd
import numpy as np
import os
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import mean_squared_error
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import KFold
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import confusion_matrix
from sklearn.metrics import roc_curve, precision_recall_curve
from sklearn.metrics import roc_auc_score
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import GridSearchCV, KFold

np.random.seed(222)

# %%
plt.rcParams['figure.figsize'] = (12,5)
plt.rcParams['figure.dpi'] = 80

# %%
d = pd.read_csv("dataQ2.csv")

# %%
df = d.groupby('V1').count()

clrs = ['grey' if (x < 1980) else '#3282bd' for x in df.index ]

plt.figure(figsize=(20,10))
plt.xticks(rotation=90)
# plot = sns.barplot(x=df.index, y = df.V3, color='#3282bd')
plot = sns.barplot(x=df.index, y = df.V3, palette=clrs)
plot.set_ylabel('Counts', fontsize=20, labelpad=20)
plot.set_xlabel('Year', fontsize=20, labelpad=20)
plt.savefig('Plots/song_year.pdf')
plt.show()

# %%
d['V1'] = np.where(d['V1'] >= 1980, 1,0)
d

# %% [markdown]
# ### LDA

# %%
y = d['V1']
X = d.loc[:,d.columns != 'V1']



# %%
TEST_SIZE = 0.3

# train/test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=TEST_SIZE)

# %%
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

# %%
lda = LDA(solver='svd', tol=1.e-4)
song_lda = lda.fit(X=X_train, y=y_train)

# %%
print(f"Prior probabilities of groups: \n {song_lda.priors_} \n")
```

```

# %%
lda_train = song_lda.predict(X_train)
lda_test = song_lda.predict(X_test)

# %%
cf_train = confusion_matrix(y_train, lda_train)
pd.DataFrame(cf_train, columns=["FALSE", "TRUE"], index=["FALSE", "TRUE"])

# %%
cf_test = confusion_matrix(y_test, lda_test)
pd.DataFrame(cf_test, columns=["FALSE", "TRUE"], index=["FALSE", "TRUE"])

# %%
from sklearn.metrics import roc_curve
fpr_1,tpr_1,thresh=roc_curve(y_test,lda.predict_proba(X_test)[:,-1])
auc_score=roc_auc_score(y_test,lda.predict_proba(X_test)[:,-1])

fpr_12,tpr_12,thresh2=roc_curve(y_train,lda.predict_proba(X_train)[:,-1])
auc_score2=roc_auc_score(y_train,lda.predict_proba(X_train)[:,-1])

print(f’AUC of Train: {auc_score2} \nAUC of Test: {auc_score}’)

plt.plot(fpr_1,tpr_1, label=’Test’)
plt.plot(fpr_12, tpr_12, label=’Train’)
plt.xlabel(’fpr1’)
plt.ylabel(’TPR’)
plt.legend()

# %% [markdown]
# ## KNN

# %%
d2 = d.copy()
y = d2[’V1’]
X = d2.loc[:,d.columns != ’V1’]

TEST_SIZE = 0.3

# train/test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=TEST_SIZE,

sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)


# X = sc.fit_transform(X)

# %%
KNeighborsClassifier().get_params().keys()

# %%
#trial:
ks = np.arange(5, 35)
ks = np.array([5,10,15,20,25,30,35,40,45,50])
gs = GridSearchCV(
    make_pipeline(
        StandardScaler(),
        KNeighborsClassifier()
        ),
        param_grid={’kneighborsclassifier__n_neighbors’: ks},
        cv=KFold(5, shuffle=True, random_state=1234)
).fit(X, y)
```

```


# %%
print(gs.best_params_)
test_score = gs.cv_results_['mean_test_score']

plt.figure(figsize=(12, 6))
plt.plot(ks, test_score, color='#3282bd', linestyle='dashed', marker='o',
         markerfacecolor='black', markersize=10)
plt.xlabel('k Value')
plt.ylabel('Mean Accuracy')
plt.savefig('Plots/errorknn.pdf')
plt.show()

# %%
classifier = KNeighborsClassifier(n_neighbors=25)
classifier.fit(X_train, y_train)

y_pred = classifier.predict(X_test)


# %%
cf_test = confusion_matrix(y_test, y_pred)
pd.DataFrame(cf_test, columns=["FALSE", "TRUE"], index=["FALSE", "TRUE"])

# %%
(479+459)/(1478+459+479+1341)

# %%
fpr_2,tpr_2,thresh=roc_curve(y_test,classifier.predict_proba(X_test)[:,-1])
auc_score=roc_auc_score(y_test,classifier.predict_proba(X_test)[:,-1])

fpr_22,tpr_22,thresh2=roc_curve(y_train,classifier.predict_proba(X_train)[:,-1]
auc_score2=roc_auc_score(y_train,classifier.predict_proba(X_train)[:,-1])

print(f'AUC of Train: {auc_score2} \nAUC of Test: {auc_score}')

plt.plot(fpr_2,tpr_2, label='Test')
plt.plot(fpr_22, tpr_22, label='Train')
plt.legend()

# %% [markdown]
# ## Naive Bayes

# %%
d3 = d.copy()

y = d3['V1']
X = d3.loc[:,d3.columns != 'V1']

TEST_SIZE = 0.3 # smaller training set to reduce computational cost

# train/test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=TEST_SIZE)

sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

# %%
from sklearn.naive_bayes import *

NB = GaussianNB()
# NB = MultinomialNB()
# NB = ComplementNB()

nb_fit = NB.fit(X_train, y_train)

y_pred = nb_fit.predict(X_test)

# %%
print("Number of mislabeled points out of a total %d points : %d"% (X_test.shape

# %%
cf_test = confusion_matrix(y_test, y_pred)
pd.DataFrame(cf_test, columns=["FALSE", "TRUE"], index=["FALSE", "TRUE"])

# %%
fpr_3,tpr_3,thresh=roc_curve(y_test,NB.predict_proba(X_test)[:,-1])
auc_score=roc_auc_score(y_test,NB.predict_proba(X_test)[:,-1])

fpr_32,tpr_32,thresh2=roc_curve(y_train,NB.predict_proba(X_train)[:,-1])
auc_score2=roc_auc_score(y_train,NB.predict_proba(X_train)[:,-1])

print(f'AUC of Train: {auc_score2} \nAUC of Test: {auc_score}')

plt.plot(fpr_3,tpr_3, label='Test')
plt.plot(fpr_32, tpr_32, label='Train')
plt.legend()

# %%
nb_fit.score(X_test, y_test)

# %%
t = np.linspace(0,1,50)
plt.plot(fpr_1,tpr_1, label='LDA')
# plt.plot(fpr_12, tpr_12, label='Train')
plt.plot(fpr_2,tpr_2, label='KNN')
# plt.plot(fpr_22, tpr_22, label='Train')
plt.plot(fpr_3,tpr_3, label='Naive Bayes')
# plt.plot(fpr_32, tpr_32, label='Train')
plt.plot(t, t, '--', alpha=0.3, color='grey')
plt.legend()
plt.xlabel('FPR')
plt.ylabel('TPR')
plt.savefig('Plots/together_plots.pdf')
plt.show()


# %%
t = np.linspace(0,1,50)
plt.subplot(131)
plt.plot(fpr_1,tpr_1, label='Test')
plt.plot(fpr_12, tpr_12, label='Train')
plt.plot(t, t, '--', alpha=0.3, color='grey')
plt.title('LDA')
plt.xlabel('FPR')
plt.ylabel('TPR')
plt.legend()
plt.subplot(132)
plt.plot(fpr_2,tpr_2, label='Test')
plt.plot(fpr_22, tpr_22, label='Train')
plt.plot(t, t, '--', alpha=0.3, color='grey')
plt.title('KNN')
plt.xlabel('FPR')
plt.legend()
plt.subplot(133)
plt.plot(fpr_3,tpr_3, label='Test')
plt.plot(fpr_32, tpr_32, label='Train')
plt.plot(t, t, '--', alpha=0.3, color='grey')
plt.title('Naive Bayes')
plt.xlabel('FPR')
plt.legend()
plt.savefig('Plots/allroc.pdf')
plt.show()

# %%
```