SLT Exam Questions

Jonathan Rossouw

20858345

Contents

Assignment 1

1.1 Question 1

In this question I analyse the hourly electricity usage of a household. The data are sourced from Kaggle and are available at https://www.kaggle.com/uciml/electric-power-consumption-data-set. The data are organised by date and time and consist of 7 features. For this analysis only the "global active power usage" feature was used making this a untivariate time series regression problem. Three separate classes of models are applied to the data. The first is the Support Vector Regression (SVR), implemented using liquidSVM, the second is the Gradient Boosting Machine (GBM), implemented using lightGBM, and thirdly the Long-Short Term Memory Recurrent Neural Network (LSTM), implemented using Keras and Tensorflow. The specific task is to forecast the following 12 hours of electricity usage from the previous 48 hours of electricity usage. The task is a multi-output regression problem.

The data consist of measurements from 16-12-2006 to 26-11-2010 which is over 47 months and is 34584 hours in total. The data are wrangled into the correct format for modelling. Firstly the data consisted of minute by minute usage readings. However, due to computational constraints, the data were averaged per hour to reduce the number of observations. Missing values are set to the rolling mean of the previous 100 and next 100 observations. The data are reformatted into a 60 hour long observations, 48 hours for the features and 12 hours for the targets. The observations are created such that the 12 hour long target sequences form a continuous sequence the length of the data set minus the first 48 hours. For the SVR and GBM models, the implementations do not allow for multi-output regressions, thus 12 separate models are trained for each class of model. This corresponds to one model per output. For the SVR the data are formatted into 12 dataframes each with the 48 features and a single target corresponding to which hour the model intends to forecast. For the GBM model a similar approach is taken, however using the lgbDataset format which is a sparse matrix format. The LSTM model has the ability to output a sequence thus multi-output regression is possible. The data are formatted into separate feature and target arrays. Each column corresponds to a specific feature or target and each row a 60 hour observation. Since this is a univariate problem, the arrays are single-channel arrays. The data for the LSTM are reformatted using min-max scaling given by

$$x_{scaled} = \frac{x - \min(x)}{\max(x) - \min(x)}$$

The data are split into training, validation and test sets. The dates from and to, minimum and maximum

Data Set	Date From	Date To	No. Months	No. Hours	No. Obs	Min	Max
Training	2006-12-16	2008-12-31	24	17904	2218	0.124	6.56
Validation	2009-01-01	2009-12-31	11	8688	726	0.180	6.52
Test	2010-01-01	2010-11-26	10	7896	655	0.195	5.63

Table 1.1: Descriptive Statistics for Data

values, the number of months and hours and number of observations for each data set are contained in table 1.1. Figure 1.1 displays a plot of the data over its entire length. Clearly the data do not follow a distinct pattern with large up and down movements. This justifies the use of machine learning techniques over traditional time series statistical modelling.

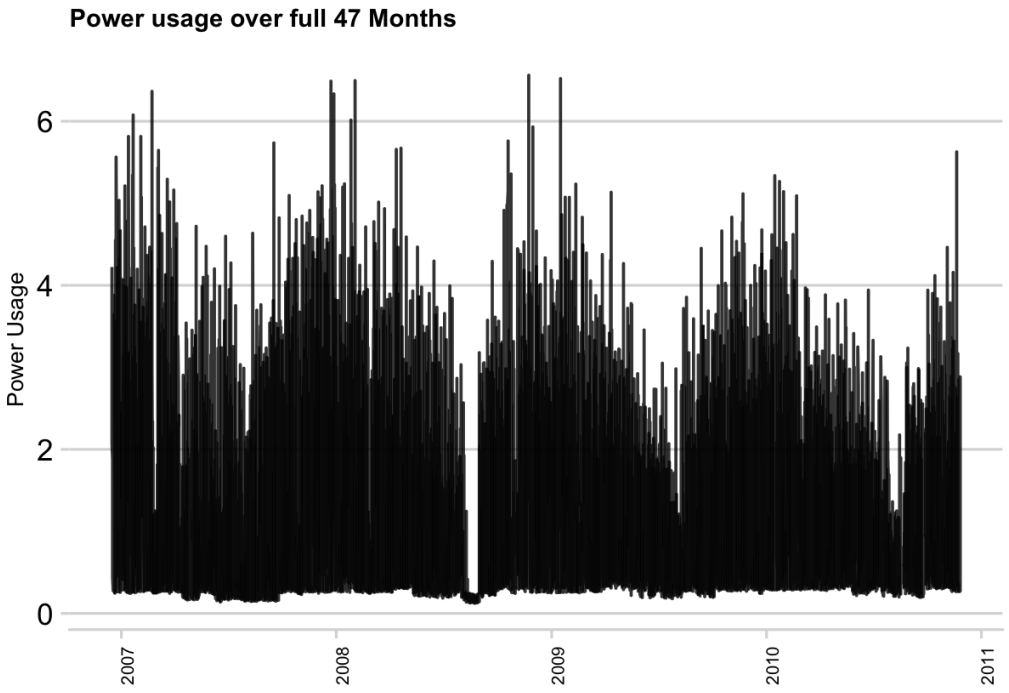

Figure 1.1: Plot of power usage over full time period

The first class of models applied to the above problem is the SVR. The SVR is an extension to the Support Vector Machine for regression problems. The SVR attempts to fit a "hyper-tube" in high dimensional space around true function which is analogous to the SVM hyperplane. Observations within the "hyper-tube" are no penalized while observations on the "hyper-tube" or outside the "hyper-tube" are penalised and become the support vectors for the "hyper-tube". Making use of a kernel for a Reproducing Kernel Hilbert space (RKHS) the "hyper-tube" can be expressed in infinitely high dimensional function space. This allows for extremely non-linear "hyper-tubes" to be formed.

The SVR can be expressed as a Lagrangian. The primal is given as

$$\begin{aligned} \underset{\beta_0,\boldsymbol{\tau},\boldsymbol{\xi},\boldsymbol{\xi^*}}{\text{maximize}} \quad & \tfrac{1}{2}\boldsymbol{\tau'K\tau} + C\sum_{i=1}^{N}(\xi_i + \xi_i^*) \\ \text{subject to} \quad & \beta_0 + \sum_{j=1}^{N}\tau_j K(\mathbf{x}_i,\mathbf{x}_j) - y_i \leq \varepsilon + \xi_i \text{ for all } i = 1,2,...,N \\ & y_i - \beta_0 - \sum_{j=1}^{N}\tau_j K(\mathbf{x}_i,\mathbf{x}_j) \leq \varepsilon + \xi_i^* \text{ for all } i = 1,2,...,N \\ & \xi_i \geq 0 \text{ for all } i = 1,2,...,N. \end{aligned} \tag{1.1}$$

The dual optimisation problem is given by

$$\underset{\boldsymbol{\alpha\alpha^*}}{\text{maximize}} \quad -\varepsilon\mathbf{1}'(\boldsymbol{\alpha}+\boldsymbol{\alpha^*}) + \mathbf{y}'(\boldsymbol{\alpha^*}-\boldsymbol{\alpha}) - \tfrac{1}{2}(\boldsymbol{\alpha^*}-\boldsymbol{\alpha})'\mathbf{G}(\boldsymbol{\alpha^*}-\boldsymbol{\alpha})$$

$$\text{subject to} \quad 0 \leq \alpha_i, \alpha_i^* \leq C; \text{ for all } i = 1,2,...,N$$

$$\sum_{i=1}^{N}(\alpha_i^* - \alpha_i) = 0$$

where $\mathbf{G}_{ij} = K(\mathbf{x}_i,\mathbf{x}_j)$. The optimisation of the SVR can be achieved through quadratic programming. In the implementation of liquidSVM, only a Gaussian radial basis function kernel is possible. The hyperparameters for the SVR are $\lambda = \frac{1}{C}$ from equation 1.1. Additionally, the hyperparameter γ is given as the bandwidth of the kernel.

The second class of models applied to the problem is the gradient boosted decision tree implemented using lightGBM. The model works by fitting shallow decision trees to the training data, then determining the residuals from the predictions of the model and the target before using these residuals as the new targets from another shallow decision tree. This process is iteratively continued forming an ensemble of models. Regardless of the type of problem or loss function stipulated, the GBM uses Mean Squared Error (MSE) loss to guide the growing of the shallow trees. Predictions are made by combining the predictions the ensemble of trees. The simple gradient tree boosting algorithm is given by algorithm 1.

Algorithm 1: Gradient Tree Boosting

1. Initialize $\widehat{\mathrm{T}}_{add}^{(0)}(\mathbf{x};\mathcal{T}) = \underset{c}{\text{argmin}}\{\sum_{i=1}^{N}\mathcal{L}(y_i,c)\}$.
2. For $m = 1,2,...,M$:
   (a) Compute $r_i^{(m)} = -(\nabla_f\mathcal{L})\Big(y_i, \widehat{\mathrm{T}}_{add}^{(m-1)}(\mathbf{x}_i;\mathcal{T})\Big)$ for $i = 1,2,...,N$
   (b) Find the tree topology $\mathfrak{T}_{\mathcal{T}}^{(m)}$ by fitting a regression tree to $r_i^{(m)}$ for $i = 1,2,...,N$
   (c) Calculate $\widehat{c}^{(m)}(\mathfrak{R}) = \underset{c}{\text{argmin}}\Big\{\sum_{\mathbf{x}_i\in\mathfrak{R}}\mathcal{L}\big(y_i, \widehat{\mathrm{T}}_{add}^{(m-1)}(\mathbf{x}_i;\mathcal{T}) + c\big)\Big\}$ for all $\mathfrak{R} \in \mathrm{Ter}\big(\mathfrak{T}_{\mathcal{T}}^{(m)}\big)$
   (d) Update $\widehat{\mathrm{T}}_{add}^{(m)}(\mathbf{x}_i;\mathcal{T}) = \widehat{\mathrm{T}}_{add}^{(m-1)}(\mathbf{x}_i;\mathcal{T}) + \mathrm{T}(\mathbf{x}_i;\widehat{\boldsymbol{\Theta}}_{\mathcal{T}}^{(m)})$, where $\widehat{\boldsymbol{\Theta}}_{\mathcal{T}}^{(m)}) = \{\widehat{\mathfrak{T}}_{\mathcal{T}}^{(m)}, \widehat{c}(.;m)\}$.
3. Output $\widehat{\mathrm{T}}_{add}^{(M)}(\mathbf{x}_i;\mathcal{T})$

The lightGBM implementation of the GBM class of models has some specific novelties. LightGBM is

implemented using gradient one-side sampling which puts greater weights on observations with larger residuals. Thus during subsampling, the least well fitted observations are more likely to be selected. This results in the model attempting to focus on fitting the observations that it is unable to accurately predict. Another novelty is called exclusive feature bundling in which categorical variables or sparse features are combined into dense features without a loss of information. These two novelties along with histogram node splitting along for much greater speed of implementation. This allows for greater hyperparameter tuning capabilities. The following are a few of the hyperparameters: the boosting type, the learning rate, feature fraction, maximum number of leaves, maximum depth of tree, number of iterations, subsample size, L_1 and L_2 penalisation. Specifically, the boosting type can be any of normal gradient boosting decision tree as described in algorithm 1, goss, dart which implements regularisation through dropping out some early trees, and random forest.

The third class of models are Neural Networks specifically the LSTM model. The LSTM architecture specifically refers to a type of recurrent cell designed specifically for sequential data. The LSTM can be viewed as a type of encoder that encodes the features of a sequence that can then be fed to a dense multilayer perceptron which then produces predictions. The LSTM is special in that along with taking in the current observation and the encoded previous observation it also keeps track of the cell state of the previous cell. Together the three inputs are fed through gates to determine the importance of each input in the encoding of the current observation. The output is an encoded observation and the current cell state. The steps of a single LSTM cell are give by

Step a: compute forget and input gates

$$\text{forget gate: } \mathbf{g}^{(f)} = \boldsymbol{\sigma}_{sig}(\mathbf{b}^{(f)} + \mathbf{V}^{(f)}\mathbf{h}^{-} + \mathbf{W}^{(f)}\mathbf{x})$$
$$\text{input gate: } \mathbf{g}^{(i)} = \boldsymbol{\sigma}_{sig}(\mathbf{b}^{(i)} + \mathbf{V}^{(i)}\mathbf{h}^{-} + \mathbf{W}^{(i)}\mathbf{x})$$

Step b: compute the Elman update

$$\mathbf{h}^{+} = \boldsymbol{\sigma}_{tanh}(\mathbf{b} + \mathbf{V}\mathbf{h}^{-}\mathbf{W}\mathbf{x})$$

Step c: compute output gate

$$\text{output gate: } \mathbf{g}^{(o)} = \boldsymbol{\sigma}_{sig}(\mathbf{b}^{(o)} + \mathbf{V}^{(o)}\mathbf{h}^{-} + \mathbf{W}^{(o)}\mathbf{x})$$

Step d: update cell state

$$\mathbf{c}^{+} = \mathbf{g}^{(f)} \odot \mathbf{c}- + \mathbf{g}^{(i)} \odot \mathbf{h}^{+}$$

Step d: update memory

$$\mathbf{h}^{+} \leftarrow \mathbf{g}^{(o)} \odot \boldsymbol{\sigma}_{tanh}(\mathbf{c}^{+})$$

where $\mathbf{V}, \mathbf{W}$ are weight matrices, $\mathbf{b}$ are bias vectors, $\boldsymbol{\sigma}$ are activation functions, $\mathbf{h}$ is the hidden or encoded state, $\mathbf{c}$ is the cell state and $\mathbf{x}$ is the current observation.

LSTM models are highly flexible as the LSTM only refers to a single cell that can be used in any configuration of neural network architecture. For this problem a single layer and two layer LSTM with different numbers of LSTM units were trained. The LSTM layers fed into either a dropout layers or just into a final dense layer. The number of epochs and whether or not the first LSTM layer returned a sequence were tuned for. The LSTM

models were implemented using the Keras API for Tensorflow. Figure 1.2 shows a diagram of the LSTM model architecture.

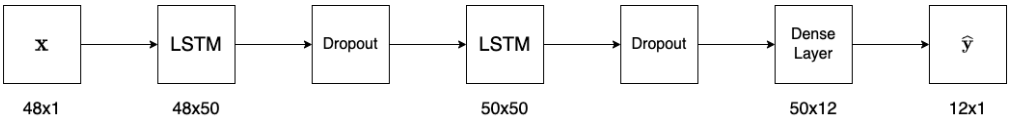

Figure 1.2: LSTM model architecture

Each model is tuned for the best combination of hyperparameters as discussed above. Using a grid search, each class of models was fit to a training data set before performance was evaluated on the validation set. The hyperparameters over which the models are tuned is given in table 1.2. The best selection of hyperparameters is given in table 1.3. For the SVR, each combination performed equally well on the validation set thus the smaller values for gamma and lambda were selected.

Class of Models	Hyperparameters
SVR	gamma: 0.082, 0.368, 1.649, 7.389, 33.115
	lambda: 0.082, 0.368, 1.649, 7.389, 33.115
GBM	boosting: dart, goss
	learning rate: 0.01, 0.05, 0.1
	feature fraction: 0.6, 0.8
	num leaves: 2, 4, 8
	max depth: 2, 4, 8
	nrounds: 50, 100, 150, 200, 250, 300, 350, 500
	subsample: 0.6, 1
	lambda l1: 0.1, 1, 10
	lambda l2: 0.1, 1, 10
LSTM	epochs: 20, 50, 100, 150
	lstm layers: 1, 2
	lstm units: 20, 50, 100
	return sequences: TRUE, FALSE
	dropout rate: 0, 0.1

Table 1.2: Hyperparameter tuning grid

The performances are given in table 1.4. The GBM resulted in the lowest training MSE of 0.369 while the LSTM had the highest training MSE of 0.525. However, the LSTM had the lowest validation MSE of 0.721. This indicates that while the LSTM does not fit the training data as closely as the other classes of models, the LSTM may generally perform better and avoid overfitting the data. From the figures 1.3-1.8, it can be seen how the forecasts on the training data fits the training observations far more closely than the validation set forecasts to the validation observations. Specifically, the validation set forecasts are closer to the mean of the observations when compared to the training set forecasts. Of the models, only the LSTM forecast negative

Class of Models	Hyperparameters
SVR	gamma: 0.082
	lambda: 0.082
GBM	boosting: goss
	learning rate: 0.05
	feature fraction: 0.6
	num leaves: 8
	max depth: 4
	nrounds: 150
	subsample: 0.6
	lambda l1: 10
	lambda l2: 10
LSTM	epochs: 100
	lstm layers: 2
	lstm units: 50
	return sequences: TRUE
	dropout rate: 0.1

Table 1.3: Best performing hyperparameters on validation set

values which is clearly not possible in this scenario. These forecasts would have to be changed heuristically in post processing of the predictions.

Model	Training MSE	Validation MSE
SVR	0.397	0.775
GBM	0.369	0.759
LSTM	0.525	0.721

Table 1.4: Best hyperparameters training and validation performance

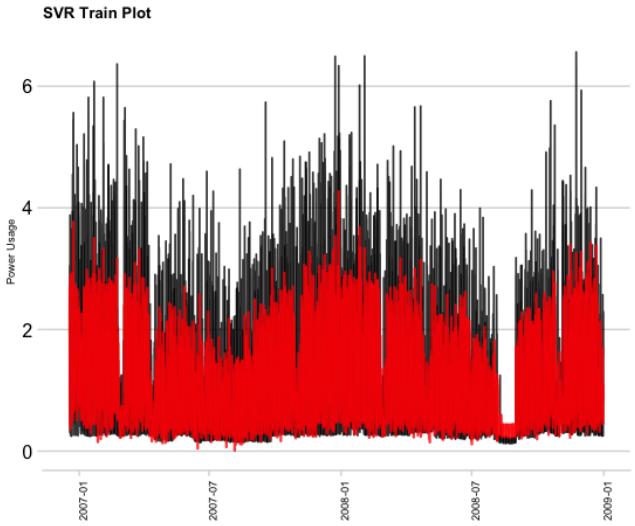

Figure 1.3: SVR training plot

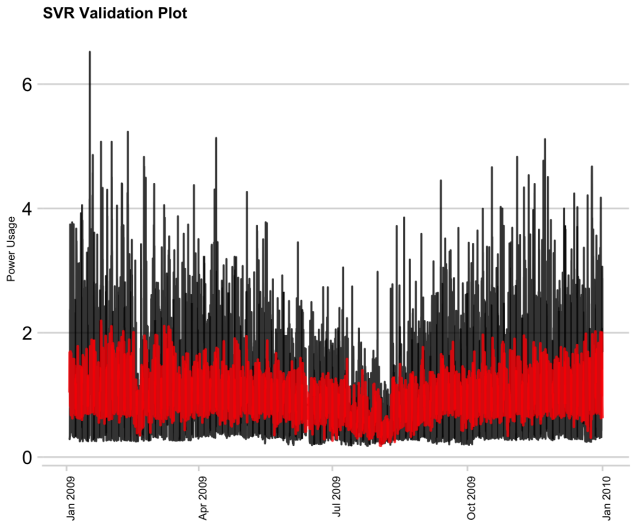

Figure 1.4: SVR validation plot

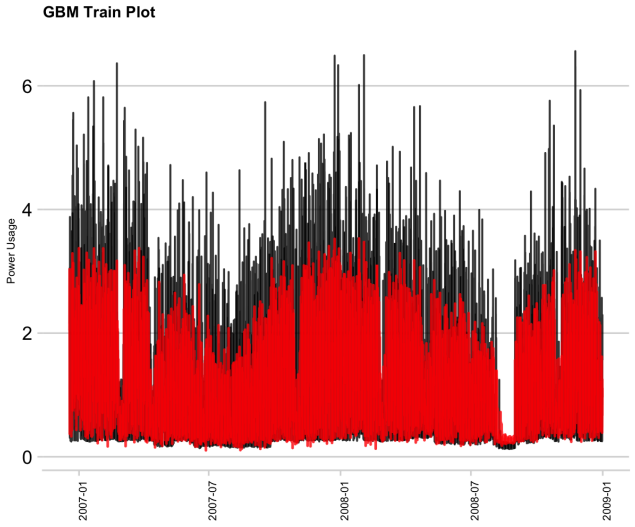

Figure 1.5: GBM training plot

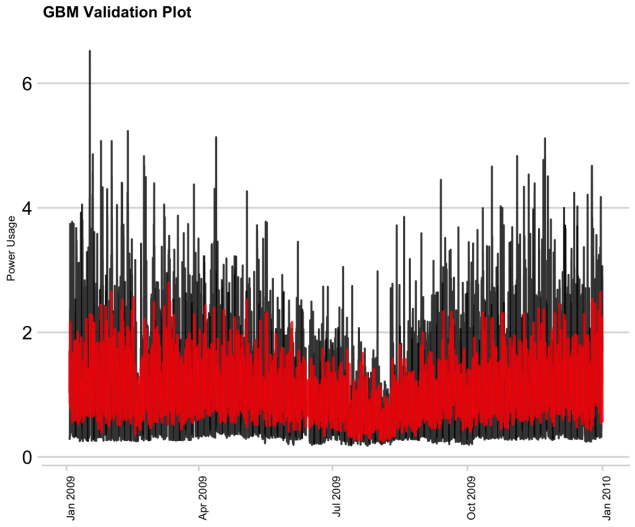

Figure 1.6: GBM validation plot

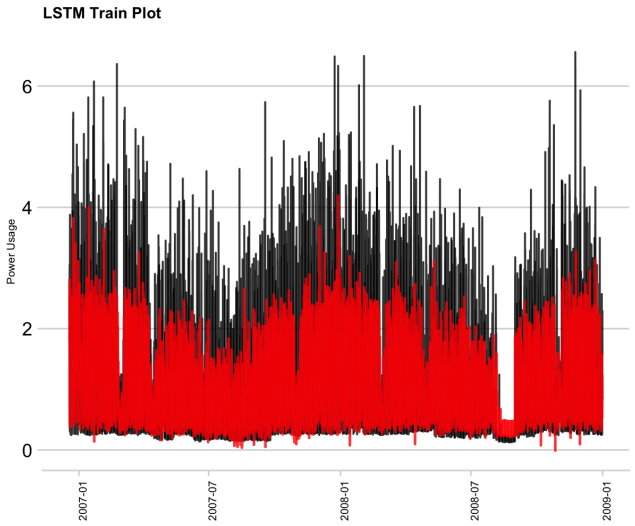

Figure 1.7: LSTM training plot

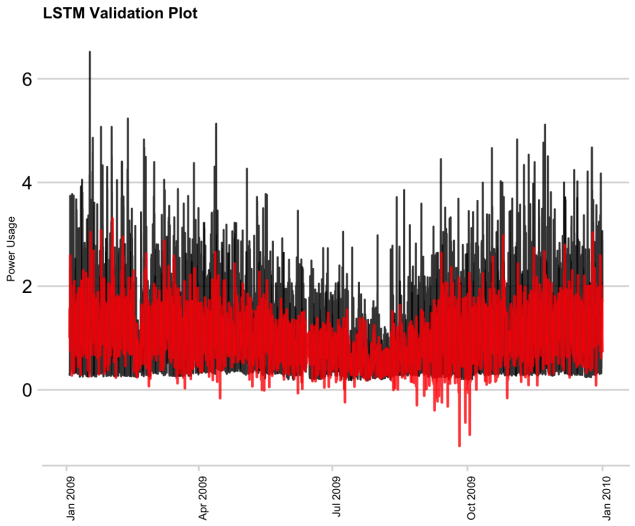

Figure 1.8: LSTM validation plot

The test performances are given in table 1.5. Once again the GBM had the lowest training MSE and the LSTM the highest training MSE. Again, the LSTM performed best on the test data with an MSE 0.642. The SVR again performed worse on the test set with a MSE of 0.661. All of the test MSEs were lower than the validation MSEs. This could be attributed to the full training set consisting of both the original training and validation sets. Thus the models were trained on larger data sets and could more closely generalise to a new data set. The LSTM most likely performed best since the architecture allows for the cell states to be kept which allows the parameters to change depending on the inputs. The LSTM model is a deep learning model and has over 5000 parameters. The GBM model performed second best and is the second most flexible behind the LSTM. The flexibility of the model clearly allows for improved predictive performance as the SVR is the least flexible even though the Gaussian Radial Basis Kernel is used. Once again the LSTM model forecast negative values which is not feasible. Figure 1.15 shows the hourly forecast for the final 7 days. From this it is clear that the data fluctuates greatly yet the LSTM is able to predict the general pattern of electricity usage.

The use of a grid search for hyperparameter tuning and the use of a validation set rather than cross validation may have limited the performance of the models. While the SVR and GBM where trained fairly exhaustively across many combinations of hyperparameters, the LSTM was not tuned thoroughly. Many different architectures could have been tried with the LSTM to find the model that has the best possible forecasts. However, in spite of the relatively limited tuning the LSTM model performed best.

Model	Training MSE	Test MSE
SVR	0.388	0.661
GBM	0.367	0.658
LSTM	0.491	0.642

Table 1.5: Training and testing performance

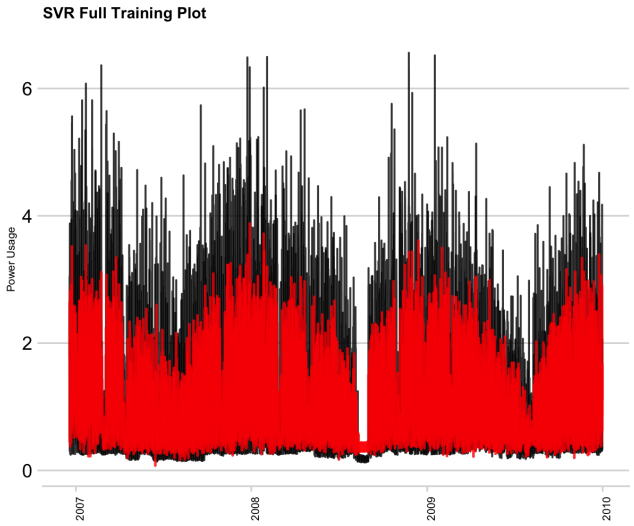

Figure 1.9: SVR combined training and validation plot

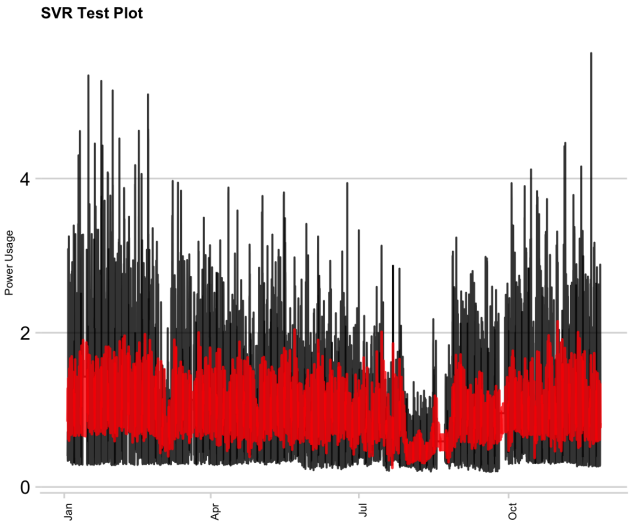

Figure 1.10: SVR test plot

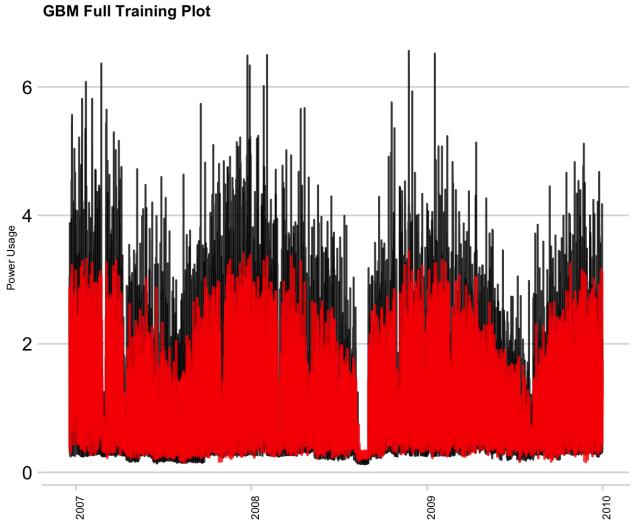

Figure 1.11: GBM combined training and validation plot

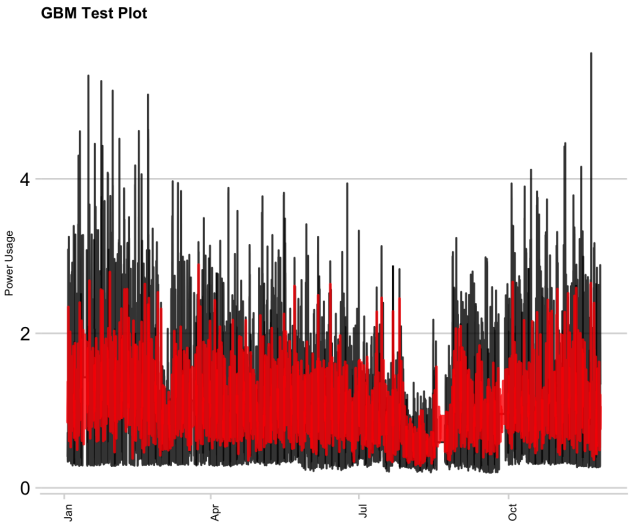

Figure 1.12: GBM test plot

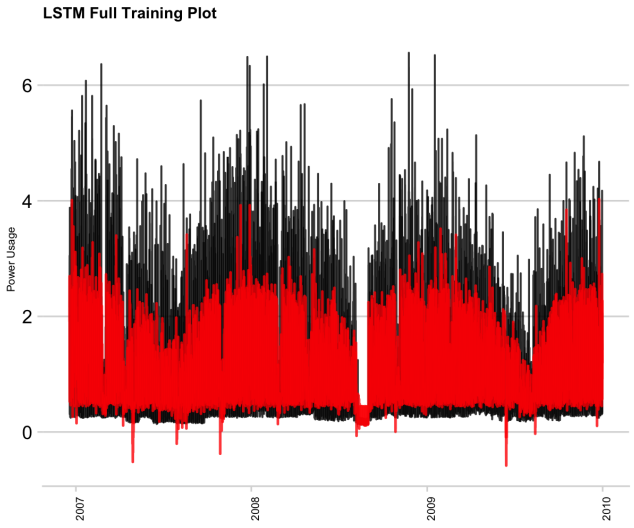

Figure 1.13: LSTM combined training and validation plot

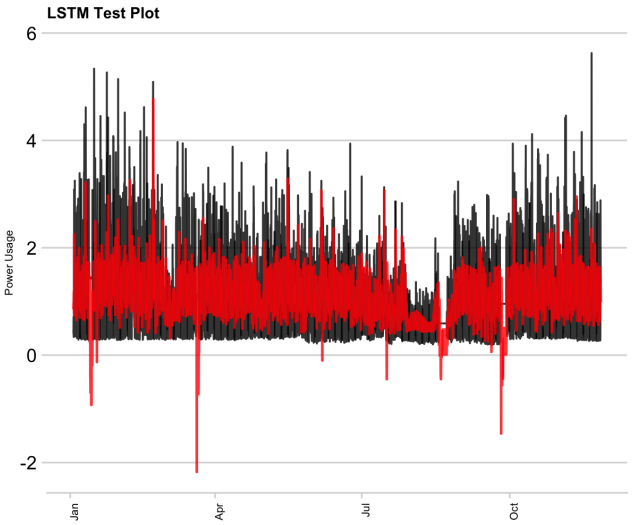

Figure 1.14: LSTM test plot

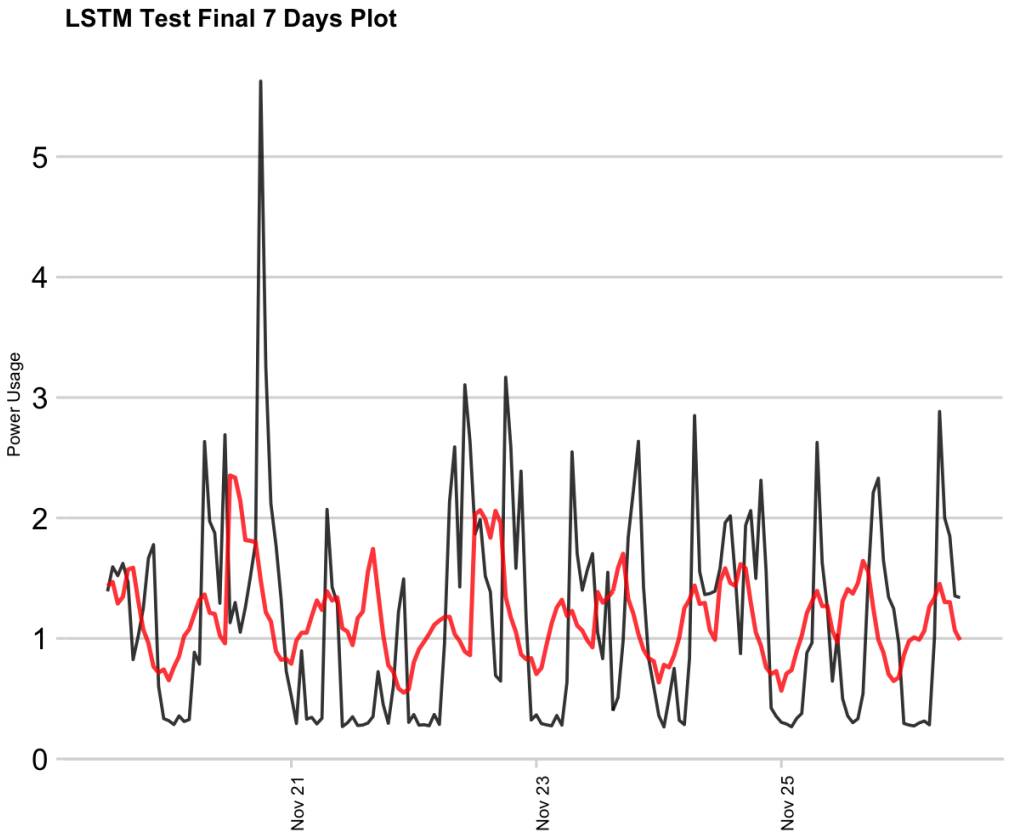

Figure 1.15: LSTM forecast for final 7 days

1.2 Question 2

Supervised learning is a branch of statistical learning theory that attempts to predict $\mathbf{y} \in \mathcal{Y}$ from the corresponding $\mathbf{x} \in \mathcal{X}$. The data is in the form of $\mathcal{T} = \{(\mathbf{x}_i, \mathbf{y}_i) : i = 1, 2, ..., N\}$ where $\mathbf{x}_i \in \mathcal{X} \subseteq \Re^p$ and $\mathbf{y}_i \in \mathcal{Y} \subseteq \Re^q$. Using the probabilistic approach, the assumption is made that each $(\mathbf{x}_i, \mathbf{y}_i) : i = 1, 2, ..., N\}$ are independent identically distributed realizations from the random vectors $(\mathbf{X}, \mathbf{Y})$ with the density function $\mathbb{P}_{\mathbf{X},\mathbf{Y}} : \mathcal{X} \times \mathcal{Y} \to [0, \infty)$. Depending on the form of the set $\mathcal{Y}$, there are different types of learning problems. When the set $\mathcal{Y}$ is a continuous subset of $\Re^q$, the learning problem is a regression problem. These problems can take on the form of a simple regression problems when $q = 1$ and multiple regression problems when $q > 1$. When $\mathcal{Y}$ is in the form of a binary category $\mathcal{G} \in \{G_1, G_2\}$, the problem is binary classification. Here the categories are usually encoded using either

$$Y = \begin{cases} 0 & \text{if } \mathcal{G} = G_1, \\ 1 & \text{if } \mathcal{G} = G_2 \end{cases}$$

or

$$Y = \begin{cases} 1 & \text{if } \mathcal{G} = G_1, \\ -1 & \text{if } \mathcal{G} = G_2. \end{cases}$$

Where there are more than two categories such that $\mathcal{G} \in \{G_1, G_2, ..., G_k\}$ with $K > 2$, the problem is multiclass classification. These can be encoded either by setting $Y = K$ if $\mathcal{G} = G_k$ or by using the random vector $\mathbf{Y} \in \Re^K$ where

$$Y_k = \begin{cases} 1 & \text{if } \mathcal{G} = G_k, \\ 0; & \text{otherwise} \end{cases}$$

The role of the loss function is to determine how well $\mathbf{Y}$ is estimated by the function $\mathbf{f}(\mathbf{X}), \mathbf{F} : \mathcal{X} \to \mathcal{A}$ where $\mathcal{A}$ is the appropriate subset of $\Re^q$. Loss functions are of the form $\mathcal{L} : \mathcal{Y} \times \mathcal{A} \to [0, \infty)$ and quantify the difference between the estimated $\mathcal{Y}$ and the actual observed Y. Using the loss function, the risk of the estimated function can be defined as the average loss over all $\mathcal{X} \times \mathcal{Y}$:

$$\begin{aligned} \mathcal{R}^*[\mathbf{f}] &= \mathbb{E}_{\mathbf{X},\mathbf{Y}}\Big[\mathcal{L}(\mathbf{Y}, \mathbf{f}(\mathbf{X}))\Big] \\ &= \int_{\mathcal{X}} \int_{\mathcal{Y}} \mathcal{L}(\mathbf{y}, \mathbf{f}(\mathbf{x})) \mathbb{P}_{\mathbf{X},\mathbf{Y}}(\mathbf{x}, \mathbf{y}) d\mathbf{y} d\mathbf{x}. \end{aligned} \tag{1.2}$$

Thus, the objective is to estimate the function that minimizes the risk, the population minimizer, over all functions from $\mathcal{X} \to \mathcal{A}$ given by

$$\mathbf{f}^* = \underset{\mathbf{f}}{\operatorname{argmin}}\{\mathcal{R}^*[\mathbf{f}]\}$$

Alternatively, following conditional expectation rules, the population minimizer can be given by

$$\mathbf{f}^* = \underset{\mathbf{c}}{\operatorname{argmin}}\Big\{\mathbb{E}_{\mathbf{Y}}\Big[\mathcal{L}(\mathbf{Y}, \mathbf{c})|\mathbf{X} = \mathbf{x}\Big]\Big\}$$

for all vectors $\mathbf{c} \in \mathcal{A}$.

There are many different loss functions depending on the learning problem. For regression, squared-error loss give by

$$\mathcal{L}(Y, f(\mathbf{X})) = (Y - f(\mathbf{X}))^2 \quad \text{where } f^{**}(\mathbf{x}) = \mathbb{E}_Y[Y|\mathbf{X} = \mathbf{x}]$$

and absolute loss given by

$$\mathcal{L}(Y, f(\mathbf{X})) = |Y - f(\mathbf{X})| \quad \text{where} \quad f^{**}(\mathbf{x}) = \text{median}[Y|\mathbf{X} = \mathbf{x}]$$

are common choices. 0-1 loss given by

$$\mathcal{L}(Y, g(\mathbf{X})) = I(Y \neq g(\mathbf{X})) \quad \text{and} \quad g^{**}(\mathbf{x}) = I(p^{**}(\mathbf{x}) > 0.5)$$
$$\text{where} \quad p^{**}(\mathbf{x}) = \mathbb{P}_Y(Y = 1|\mathbf{X} = \mathbf{x})$$

is used for binary classification with g^{**} known as the Bayes error. Since $p^{**}(\mathbf{x})$ is the conditional probability, for models that do not directly estimate the classification rule, a loss function is specified to guide the estimation of p^{**}. Cross entropy is such as loss function where $p(\mathbf{x}) = \frac{e^f(\mathbf{x})}{1+e^f(\mathbf{x})}$ for $f : \mathcal{X} \to \Re$ and is given by

$$\mathcal{L}(Y, f(\mathbf{X})) = -Y \log(p(\mathbf{x})) - (1 - Y) \log(1 - p(\mathbf{x})).$$

For multiclass classification learning problems, the loss function given by

$$\mathcal{L}(\mathbf{Y}, \mathbf{f}(\mathbf{X})) = -\sum_{k=1}^{K} Y_k \log(p_k(\mathbf{X})) \quad \text{where} \quad p_k(\mathbf{X}) = \frac{e^{f_k(\mathbf{x})}}{\sum_{l=1}^{K} e^{f_l(\mathbf{x})}}.$$

Since, supervised learning attempts to estimate a function that minimizes the risk, there is a model space $\mathcal{F}$ which is a set of all functions and contains the function that best represents the population minimizer. This function is the optimal model given by

$$\mathbf{f}^* = \underset{\mathbf{f} \in \mathcal{F}}{\text{argmin}}\{\mathcal{R}^*[\mathbf{f}]\}.$$

Learning algorithms are functions that map data to the model space as $\hat{\mathbf{f}} : \mathcal{D} \to \mathcal{F}$ where $\mathcal{D}$ is all the possible sets of data that can be observed. When applying a learning algorithm to a specific data set give by $\mathcal{T} = \{(\mathbf{x}_i, \mathbf{y}_i) : i = 1, 2, ..., N\}$, the fitted model $\hat{\mathbf{f}}(\mathbf{x}; \mathcal{T})$ is obtained.

Since the optimal model minimizes $\mathcal{R}^*[\mathbf{f}]$, from equation 1.2, in order to obtain the optimal model the joint distribution $\mathbb{P}_{\mathbf{Y},\mathbf{X}}$ or conditional distribution $\mathbb{P}_{\mathbf{Y}}(.|\mathbf{X})$ are required. This can be estimated using the empirical distribution resulting in the empirical risk

$$\frac{1}{N}\mathcal{R}[\mathbf{f}] = \frac{1}{N}\sum_{i=1}^{N} \mathcal{L}(\mathbf{y}_i, \mathbf{f}(\mathbf{x}_i))$$

and the fitted model resulting from minimizing the empirical risk for the data set $\mathcal{T} = \{(\mathbf{x}_i, \mathbf{y}_i) : i = 1, 2, ..., N\}$ is

$$\hat{\mathbf{f}}(.; \mathcal{T}) = \underset{\mathbf{f} \in \mathcal{F}}{\text{argmin}}\{\mathcal{R}[\mathbf{f}]\}$$

The performance of a fitted model $\hat{\mathbf{f}}(.; \mathcal{T})$ can be determined from the training data set $\mathcal{T} = \{(\mathbf{x}_i, \mathbf{y}_i) : i = 1, 2, ..., N\}$ as

$$\overline{err}(\hat{\mathbf{f}}(.; \mathcal{T})) = \frac{1}{N}\sum_{i=1}^{N} \mathcal{L}(\mathbf{y}_i, \hat{f}(\mathbf{x}_i; \mathcal{T})) = \frac{1}{N}\mathcal{R}[\hat{\mathbf{f}}(.; \mathcal{T})]. \tag{1.3}$$

However, this is not a good measure of performance and only shows how closely the model can fit the given data and not how well the model can predict observation from a different data set. The generalization error is

a measure of performance of a fitted model on data that was not used in fitting the model. The generalization error is given by

$$\text{Err}(\hat{\mathbf{f}}(.;\mathcal{T})) = \mathbb{E}_{\mathbf{X},\mathbf{Y}}\big[\mathcal{L}(\mathbf{Y},\hat{\mathbf{f}}(\mathbf{X};\mathcal{T}))\big] \tag{1.4}$$

where $(\mathbf{X},\mathbf{Y})$ is independent of $\mathcal{T}$. In order to measure the performance of a learning algorithm more generally, the expected generalization error

$$\text{Err}(\hat{\mathbf{f}}) = \mathbb{E}_{\mathcal{T}}\big[\text{Err}(\hat{\mathbf{f}}(.;\mathcal{T}))\big]$$

can be determined.

The complexity of a model, in a statistical learning sense, means how closely the learning algorithm can fit and thus predict observations from the data that was used in its construction. In certain cases, by minimizing the empirical risk, complex models do not perform well on other data and thus have a high generalisation error. A method to overcome this is to use regularization to fit less complex models by not fully minimizing the empirical risk. Regularization can be seen as penalising a model for being too complex or as restricting the model space to models that are not too complex. Regularization requires a specified functional $\mathcal{J}:\mathcal{F}\to[0,\infty)$ that results in a higher value for more complex functions. When restricting the model space for a specific functional the model space becomes

$$\mathcal{F}_t = \{\mathbf{f}\in\mathcal{F}:\mathcal{J}[\mathbf{f}\leq t\}$$

and the fitted model is determined by optimizing

$$\underset{\mathbf{f}\in\mathcal{F}_t}{\text{argmin}}\mathcal{R}[\mathbf{f}]$$

for specified t. This results in a collection of learning algorithm for each t. When penalising a model, the fitted model can be determined by optimising

$$\hat{\mathbf{f}}_\lambda(.;\mathcal{T}) = \underset{\mathbf{f}\in\mathcal{F}_t}{\text{argmin}}\{\mathcal{R}[\mathbf{f}]+\lambda\mathcal{J}[\mathbf{f}]\} = \underset{\mathbf{f}\in\mathcal{F}_t}{\text{argmin}}\mathcal{R}[\mathbf{f}] \tag{1.5}$$

where for a specific t, there usually exist $\lambda\geq 0$. The fitted model in equation 1.5 is determined by minimizing the penalized empirical risk $\mathcal{R}[\mathbf{f};\lambda]=\mathcal{R}[\mathbf{f}]+\lambda\mathcal{J}[\mathbf{f}]$ resulting in a collection of learning algorithms $\{\mathbf{f}_\lambda:\lambda\geq 0\}$ that can be fit to the data.

There are many different functionals $\mathcal{J}[f]$ that penalize the empirical risk in different ways. For ridge regression and lasso, the model space is defined as

$$\mathcal{F} = \{f : f(\mathbf{x}) = \beta_0 + \boldsymbol{\beta}'\mathbf{x}, \beta_0\in\Re, \boldsymbol{\beta}\in\Re^p\}$$

where the functional $\mathcal{J}[f] = \left\|\mathbb{E}_{\mathbf{X}}\left[\frac{\partial f}{\partial\mathbf{X}}\right]\right\|_2^2$ for ridge regression and $\mathcal{J}[f] = \left\|\mathbb{E}_{\mathbf{X}}\left[\frac{\partial f}{\partial\mathbf{X}}\right]\right\|_1$ for lasso. For smoothing splines with a single feature, the functional $\mathcal{J}[f]=\int_{\mathcal{X}}[f''(t)]^2$ is used.

Along with interpretability, selecting the model that performs best on the data you are trying to predict is an important part of statistical learning. However, using the training error given in equation 1.3 is not a good measure of how well the model can predict new observations. More complex models have lower training errors but that does not necessarily mean that the model will generalize to new observations. Thus it is important to measure the generalization error given in equation 1.4. The generalization error requires the expected value of

the fitted models loss for observations independent of the training data $\mathcal{T}$. In most cases, determining this is not feasible so the test error is used instead. The test error is determined by first splitting all available data into a training data set $\mathcal{T}$ and a test data set $\mathcal{T}_e = \{(\tilde{\mathbf{x}}_i, \tilde{\mathbf{y}}_i) : i = 1, 2, ..., N_e\}$, fitting a model to the training data set and then calculating the test error given by

$$\widehat{\text{Err}}\big(\hat{\mathbf{f}}(.;\mathcal{T})\big) = \frac{1}{N_e}\sum_{i=1}^{N_e}\mathcal{L}\big(\tilde{\mathbf{y}}_i, \hat{\mathbf{f}}(\tilde{\mathbf{x}}_i;\mathcal{T})\big).$$

It is often the case that either the same learning algorithm with different parameters or a number of entirely different learning algorithms are fit to the training data denoted $\{\hat{f}_\gamma; \gamma \in \Gamma\}$ and then a single best performing fitted model is selected. In order to determine which fitted model $\hat{f}_{\hat{\gamma}}$ has the best generalization error, the training data $\mathcal{T}$ must be further split into a training set $\mathcal{T}_r$ and a validation set $\mathcal{T}_v = \ddot{\mathbf{x}}_i, \ddot{\mathbf{y}}_i : i = 1, 2, ..., N_v)\}$. In a data rich environment it is possible to determine the best model by first fitting $\widehat{f_\gamma}(.;\mathcal{T}_r)$ for all $\gamma \in \Gamma$, secondly determining the validation error

$$\mathcal{V}(\gamma) = \frac{1}{N_v}\sum_{i=1}^{N_v}\mathcal{L}\big(\ddot{\mathbf{y}}_i, \widehat{\mathbf{f}_\gamma}(\ddot{\mathbf{x}}_i;\mathcal{T}_r)\big).$$

for the fitted models, thirdly minimizing $\mathcal{V}_\gamma$ over all $\gamma \in \Gamma$ resulting in a $\widehat{\mathbf{f}_{\hat{\gamma}}}(.;\mathcal{T}_r)$, fourthly re-estimating $\widehat{f_{\hat{\gamma}}}(.;\mathcal{T})$, and finally determining the test error $\widehat{\text{Err}}\big(\widehat{\mathbf{f}_{\hat{\gamma}}}(.;\mathcal{T})\big)$. Here the test error is a good estimate for the generalization error. Figure 1.16 shows how the training error and validation error differ for different levels of model complexity. The most complex model results in the lowest training error but does not generalize well to new data. Figure 1.17 shows the distribution of the test root mean squared error (RMSE) for different levels of model complexity. The red box shows the model that was select due to the best performance on the validation set. Clearly the model selection process selected a level of model complexity close to the ideal level.

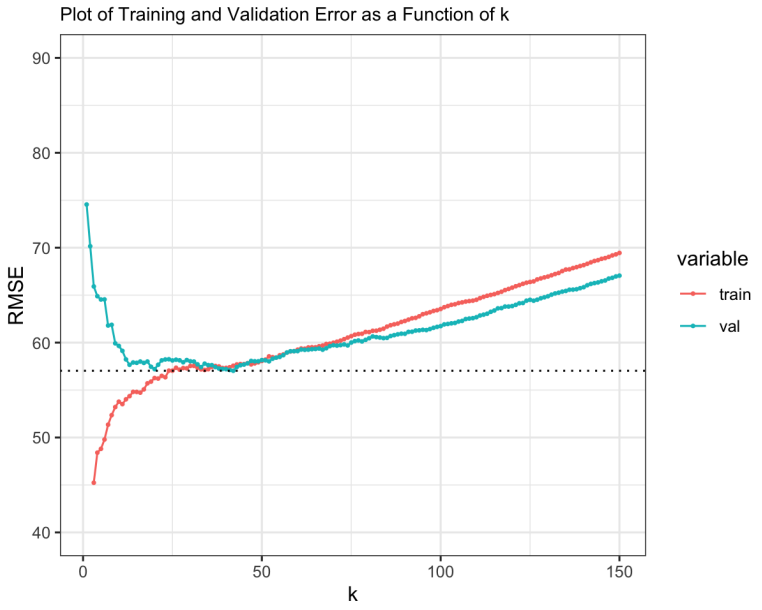

Figure 1.16: KNN comparison of training and validation error

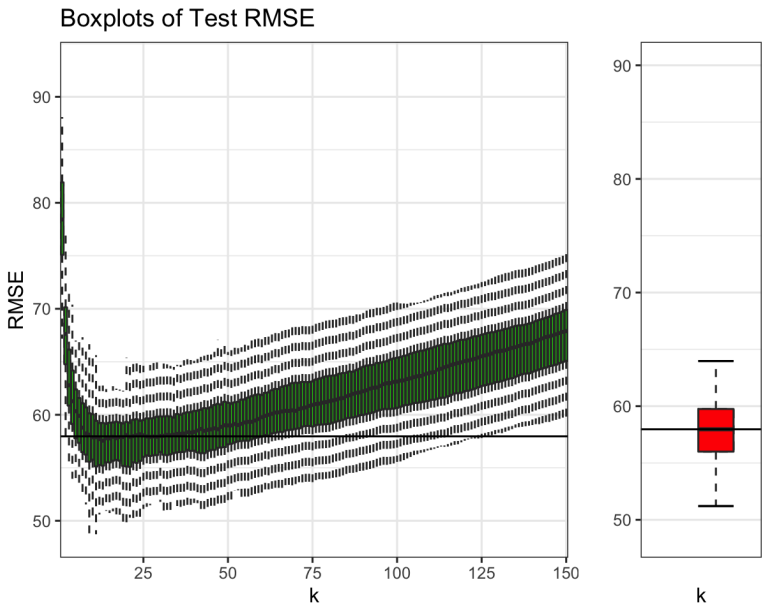

Figure 1.17: KNN comparison of test RMSE

In statistical learning the main objective is to find a model that best predicts $\mathbf{y} \in \mathcal{Y}$ from the corresponding $\mathbf{x} \in \mathcal{X}$. This requires a model to be fitted to data that has the best generalization error. The term overfitting refers to when a decision is taken than reduces the training error but increases the generalization error. Conversely, the term underfitting refers to when a decision that reduces the training error and the generalization error is not taken. When choosing different models, the greater the model complexity the less there is a chance of underfitting but with the greater complexity comes the risk of overfitting. The mean squared error (MSE) can be used to understand the trade-off between the risk of over- and underfitting. For the additive model $Y = f^{**}(\mathbf{X}) + \epsilon$ where $\epsilon \sim (0, \sigma^2)$ and $\epsilon, \mathbf{X}$ are independent, under squared error loss the generalization error can be written as

$$\begin{aligned} \text{Err}[\hat{f}] &= \sigma^2 + MSE[\hat{f}] \\ &= \sigma^2 + \mathbb{E}_{\mathbf{x}_0}\left[\left(f^{**}(\mathbf{x}_0) - \mathbb{E}_{\mathcal{T}}[\hat{f}(\mathbf{x}_0)]\right)^2\right] + \mathbb{E}_{\mathbf{x}_0}\left[\mathbb{V}_{\mathcal{T}}[\hat{f}(\mathbf{x}_0)]\right]. \end{aligned} \tag{1.6}$$

In equation 1.6, the squared bias is given by $\text{bias}^2[\hat{f}] = \mathbb{E}_{\mathbf{x}_0}\left[\left(f^{**}(\mathbf{x}_0) - \mathbb{E}_{\mathcal{T}}[\hat{f}(\mathbf{x}_0)]\right)^2\right]$ and the variance is given by $\text{var}[\hat{f}] = \mathbb{E}_{\mathbf{x}_0}\left[\mathbb{V}_{\mathcal{T}}[\hat{f}(\mathbf{x}_0)]\right]$. The bias-variance trade-off is displayed in figure 1.18 for the ridge regression model.

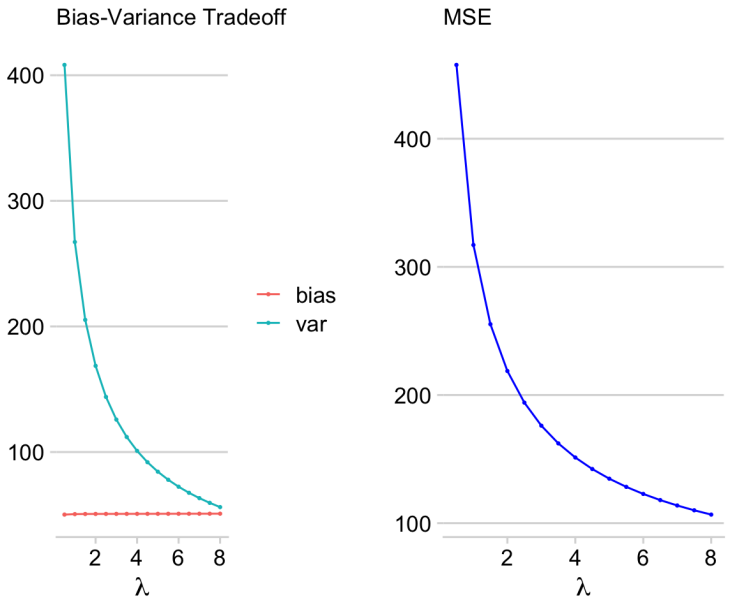

Figure 1.18: Ridge Regression Bias-Variance Trade-off

The curse of dimensionality, refers to the exponential increase in the volume of an object in Euclidean space when dimensions are added. In a statistical learning setting, this refers to the increase in the sample size required to estimate the effects that features have on a response as the number of features p of $\mathbf{X}$, or dimensions of $\mathbf{X}$, increase. The increase in the dimensions results in an increase in the distance between observations thus more observations are required to accurately predict new observations. This is illustrated in figures 1.19 and 1.20 where the change in the MSE is illustrated for two different functions as the dimensions increase. The figures also show how the bias-variance trade-off changes for different functions. Figure 1.19 shows the function $Y = f^{**}(\mathbf{X}) = \exp(-8\|\mathbf{X}\|_2^2)$ and figure 1.20 the function $Y = f^{**}(\mathbf{X}) = \frac{1}{2}(X_1 + 1)^3$ where $X_i \sim$ iid unif$(-1, 1)$ and $\mathbf{X} = (X_1, X_2, ..., X_p)'$. Both figures are created by first simulating $\mathcal{T}^{(b)}$ for $b = 1, 2, ..., 10000$, then calculating $\hat{f}_k^{(b)}(\mathbf{0})$ for $b = 1, 2, ..., 10000$ and from the fitted models determining $E_{\mathcal{T}}\left[\hat{f}_k^{(b)}(\mathbf{0})\right]$ and $\text{var}_{\mathcal{T}}\left[\hat{f}_k^{(b)}(\mathbf{0})\right]$. In figure 1.19 the bias dominates the variance as the MSE increases, and in figure 1.20 the variance dominates the bias as the MSE increases. This is due to the nature of the functions.

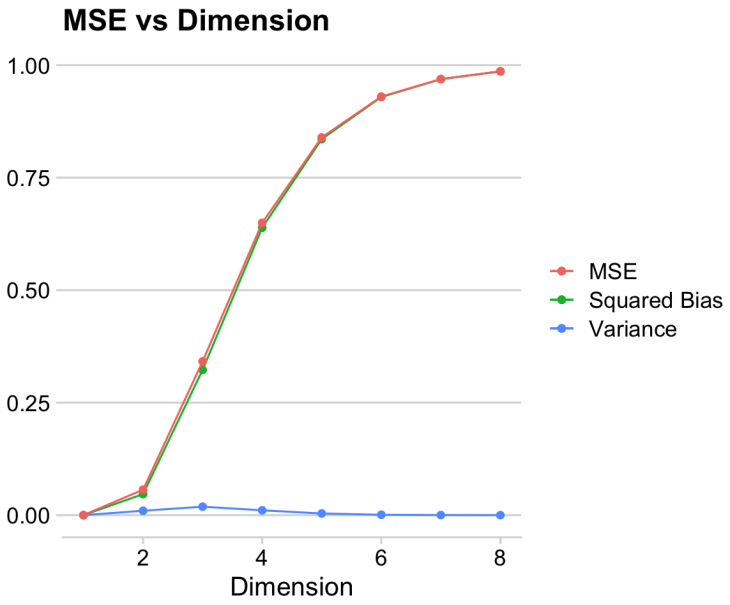

Figure 1.19: KNN Bias Variance Trade-off

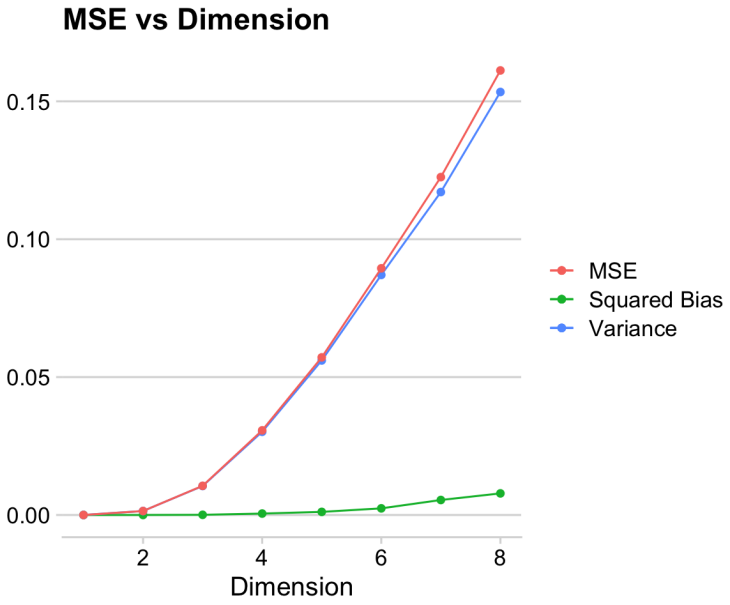

Figure 1.20: KNN Bias Variance Trade-off

Assignment 2

2.1 Question 1

Backpropagation is a method of calculating derivatives of complicated functions by iteratively applying the chain rule. For complicated composite functions which may contain many nested functions it is very difficult to determine the derivative through symbolic differentiation. Backpropagation overcomes these issues by breaking down the function into individual steps that contain a limited number of inputs and usually only a single operation. The first step takes in inputs in the form of data or parameters, performs an operation on the inputs and results in an output. This output forms the input of the next step and so on. A step may include the output of a single step, multiple steps, the original data and parameters or additional data or parameters. The final step's output is the output of the function as a whole. The process of breaking down the function into these consecutive steps is referred to as the forwards pass. The forwards pass is often illustrated in the form of a computational graph as seen in figure 2.1. For the simple function $f = x \times y$ the forward pass would be $f = x \times y$ since there the is only a single operation.

The process of determining the derivatives of the function is referred to as the backwards pass. The backwards pass is initialised by creating accumulators for each input and the outputs of each step. The accumulator for the final output is set equal to one while the remainder are set equal to zero. The backwards pass consists of iteratively updating each accumulator. The update process for the above simple function takes the form of

$$\delta_x = \delta_x + \frac{\partial f}{\partial x}\delta_f = y$$
$$\delta_y = \delta_y + \frac{\partial f}{\partial y}\delta_f = x$$

where $\delta_x = 0$ and $\delta_y = 0$ are the accumulators for the inputs x and y respectively, and $\delta_f = 1$ is the accumulator for the final output. Thus the update step consists of adding the product of the partial derivative of the output with respect to the input and the accumulator of the output to the accumulator of the input. The accumulators are updated by iteratively working through the forward pass steps in reverse, from the final output to the initial inputs. Thus once the accumulators of the initial inputs are updated a final time they are the partial derivative of the final output with regards to the initial inputs.

Backpropagation is underpinned by the chain rule. Through multiplying the partial derivative of the out with regards to the input with the accumulator of the output, the chain rule is effectively being applied step by step on each nested function. With many statistical learning techniques requiring an optimization step, the usefulness

of backpropagation comes in its ability to determine the derivatives of functions that consist of many composite functions. For example, multilayer neural networks many linear combinations of inputs and weights which are then transformed with a non-linear function. The weights in the hidden layers are updated through gradient descent which requires the calculation of the derivatives. To calculate the derivatives symbolically becomes prohibitively complicated and expensive as hidden layers are added. By breaking down the layers into single operations, backpropagation allows for the derivatives to be calculated relatively inexpensively. Furthermore, by reducing the complex structures into simple operations, libraries of operations can be stored and accessed to quickly determine updates for steps in the backwards pass.

When applying backpropagation to a function, especially functions used in a statistical learning setting, there often arises the need to differentiate scalar or vector functions by scalars, vectors or matrices. In these settings, normal differentiation techniques are not applicable. Thus vector calculus is required to ensure that the derivatives maintain the correct structure. The denominator convention is followed here.

The derivatives of functions that map from a vector space to a scalar, $f : \Re^p \to \Re$, in terms of vectors are given by

$$\frac{\partial f}{\partial \mathbf{x}} = (\nabla_{\mathbf{x}} f)(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial \mathsf{x}_1} \\ \frac{\partial f}{\partial \mathsf{x}_2} \\ \cdots \\ \frac{\partial f}{\partial \mathsf{x}_p}. \end{bmatrix}$$

Similarly, the derivatives of functions that map scalars to a vectors space, $f : \Re \to \Re^q$, in terms of a scalar are given by

$$\frac{\partial \mathbf{f}}{\partial \mathsf{x}} = (\nabla_{\mathsf{x}} \mathbf{f})(\mathsf{x}) = \begin{bmatrix} \frac{\partial f_1}{\partial \mathsf{x}} & \frac{\partial f_2}{\partial \mathsf{x}} & \cdots & \frac{\partial f_q}{\partial \mathsf{x}} \end{bmatrix}.$$

The derivatives of functions that map from a vector space to a vector space, $\mathbf{f} : \Re^p \to \Re^q$, in terms of vectors are given by

$$\frac{\partial \mathbf{f}}{\partial \mathbf{x}} \in \Re^{p \times q} = (\nabla_{\mathbf{x}} \mathbf{f})(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_1}{\partial \mathbf{x}} & \frac{\partial f_2}{\partial \mathbf{x}} & \cdots & \frac{\partial f_q}{\partial \mathbf{x}} \end{bmatrix}$$

where $\mathbf{f}(\mathbf{x}) = (f_1(\mathbf{x}), f_2(\mathbf{x}), \ldots, f_q(\mathbf{x}))'$.

The derivatives of functions that map from a matrix space to scalars, $f : \Re^{n \times p} \to \Re$, are given by

$$\frac{\partial f}{\partial \mathbf{x}} = (\nabla_{\mathbf{X}} f)(\mathbf{X}) = \begin{bmatrix} \frac{\partial f}{\partial \mathsf{x}_{11}} & \frac{\partial f}{\partial \mathsf{x}_{12}} & \cdots & \frac{\partial f}{\partial \mathsf{x}_{1p}} \\ \frac{\partial f}{\partial \mathsf{x}_{21}} & \frac{\partial f}{\partial \mathsf{x}_{22}} & \cdots & \frac{\partial f}{\partial \mathsf{x}_{2p}} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial f}{\partial \mathsf{x}_{n1}} & \frac{\partial f}{\partial \mathsf{x}_{n2}} & \cdots & \frac{\partial f}{\partial \mathsf{x}_{np}} \end{bmatrix}$$

However, the derivatives of functions that map from a matrix space to a vector space, $\mathbf{f} : \Re^{n \times p} \to \Re^q$, in terms of vectors requires the use of tensors, three-dimensional arrays, in order to correctly structure the derivatives. This requires the use of tensor algebra which may be inaccessible for many. A workaround is to vectorize the argument of the derivative. Thus the function becomes $\mathbf{f} : \Re^{np} \to \Re^q$ which is more manageable but requires special care in order to maintain the correct structure.

Specific instances that require vector calculus when using backpropagation to determine derivatives in statistical learning include functions of the form $\mathbf{f} = \mathbf{W}'\mathbf{x}_i$ where $\boldsymbol{W}$ is a matrix of weights and $\mathbf{x}_i$ is a vector of

inputs. Since $\frac{\partial \mathbf{f}}{\partial \mathbf{W}}$ results in a tensor, it is more convenient to apply vector calculus to the individual rows of $\mathbf{W}$ as $\frac{\partial \mathbf{f}}{\partial \mathbf{w}_m}$ for $m = 1, 2, ..., M$ and then recreate

$$\frac{\partial \mathbf{f}}{\partial \mathbf{W}} = \begin{bmatrix} \frac{\partial \mathbf{f}}{\partial \mathbf{w}_1}' \\ \frac{\partial \mathbf{f}}{\partial \mathbf{w}_2}' \\ \vdots \\ \frac{\partial \mathbf{f}}{\partial \mathbf{w}_M}' \end{bmatrix}.$$

To illustrate backpropagation, the partial derivatives in terms of x and y for the function

$$f(xy) = \frac{(\sin(x)+1)^{xy} - e^{tanh(e^x y)}}{\sqrt[3]{\frac{y^2}{\sin(\sqrt{xy})}}}$$

are to be determined. First, using symbolic differentiation yields

$$\frac{\partial f}{\partial x} = \frac{y\cot(\sqrt{xy})\left((\sin(x)+1)^{xy} - e^{\tanh(e^x y)}\right)}{6\sqrt{xy}\sqrt[3]{y^2\csc(\sqrt{xy})}} + \frac{(\sin(x)+1)^{xy}\left(y\log(\sin(x)+1) + \frac{xy\cos(x)}{\sin(x)+1}\right) - ye^{\tanh(e^x y)+x}\operatorname{sech}^2(e^x y)}{\sqrt[3]{y^2\csc(\sqrt{xy})}}$$

$$\frac{\partial f}{\partial y} = \frac{x(\sin(x)+1)^{xy}\log(\sin(x)+1) - e^{\tanh(e^x y)+x}\operatorname{sech}^2(e^x y)}{\sqrt[3]{y^2\csc(\sqrt{xy})}} - \frac{\sin(\sqrt{xy})\left((\sin(x)+1)^{xy} - e^{\tanh(e^x y)}\right)\left(2y\csc(\sqrt{xy}) - \frac{xy^2\cot(\sqrt{xy})\csc(\sqrt{xy})}{2\sqrt{xy}}\right)}{3y^2\sqrt[3]{y^2\csc(\sqrt{xy})}}$$

which is a complicated mess that cannot be quickly evaluated. When evaluated at the point $(1, 1)$ the partial derivatives yield

$$f_x(1,1) = 1.3659399717387952 \text{ and } f_y(1,1) = 1.3929338938392877$$

In order to perform backpropagation, the computational graph of the function, given in figure 2.1, is required. Then the function is broken up into functions with single operations during the forward pass

1. $a = \sin(x)$
2. $b = xy$
3. $c = a + 1$
4. $d = c^b$
5. $e = e^x$
6. $f = e \times y$
7. $g = \tanh f$
8. $h = e^g$

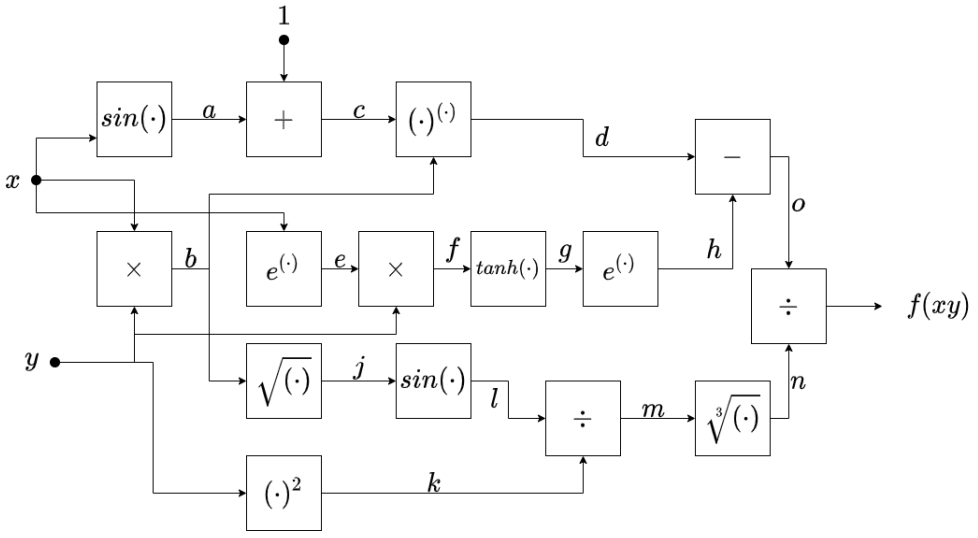

Figure 2.1: Computational Graph of Complicated Function

9. $j = \sqrt{b}$
10. $k = y^2$
11. $l = \sin(j)$
12. $m = \frac{k}{l}$
13. $n = \sqrt[3]{m}$
14. $o = d - h$
15. $f(xy) = \frac{o}{n}$

Then the backwards pass begins by setting the accumulators $\partial_x, \partial_y, \partial_a, \partial_b, \partial_c, \partial_d, \partial_e, \partial_f, \partial_g, \partial_h, \partial_i, \partial_j, \partial_k, \partial_l, \partial_m, \partial_n, \partial_o$ all equal to 0 and $\partial_{f(xy)} = 1$. The backwards pass is then

1. $\delta_o = \delta_o + \frac{\partial f(xy)}{\partial o}\delta_{f(xy)} = \frac{1}{n}$
   $\delta_n = \delta_n + \frac{\partial f(xy)}{\partial n}\delta_{f(xy)} = -\frac{o}{n^2}$
2. $\delta_d = \delta_d + \frac{\partial o}{\partial d}\delta_o = \frac{1}{n}$
   $\delta_h = \delta_h + \frac{\partial o}{\partial h}\delta_o = -\frac{1}{n}$
3. $\delta_m = \delta_m + \frac{\partial n}{\partial m}\delta_n = \frac{1}{3m^{\frac{2}{3}}} \cdot \frac{-o}{n^2}$
4. $\delta_k = \delta_k + \frac{\partial m}{\partial k}\delta_m = \frac{1}{l} \cdot \frac{-o}{3n^2m^{\frac{2}{3}}}$
   $\delta_l = \delta_l + \frac{\partial m}{\partial l}\delta_m = (-\frac{k}{l^2}) \cdot \frac{-ok}{3n^2m^{\frac{2}{3}}}$
5. $\delta_j = \delta_j + \frac{\partial l}{\partial j}\delta_l = \cos(j) \cdot \frac{ok}{3l^2n^2m^{\frac{2}{3}}}$
6. $\delta_y = \delta_y + \frac{\partial k}{\partial y}\delta_k = 2y \cdot \frac{-o}{3ln^2m^{\frac{2}{3}}}$
7. $\delta_b = \delta_b + \frac{\partial j}{\partial b}\delta_j = \frac{1}{2\sqrt{b}} \cdot \frac{\cos(j)ok}{3l^2n^2m^{\frac{2}{3}}}$

8. $\delta_g = \delta_g + \frac{\partial h}{\partial g}\delta_h = e^g \cdot \frac{-1}{n}$

9. $\delta_f = \delta_f + \frac{\partial g}{\partial f}\delta_g = \operatorname{sech}^2(f) \cdot \frac{-e^g}{n}$

10. $\delta_e = \delta_e + \frac{\partial f}{\partial e}\delta_f = y \cdot \frac{-\operatorname{sech}^2(f)e^g}{n}$
    $\delta_y = \delta_y + \frac{\partial f}{\partial y}\delta_f = \frac{-2oy}{3ln^2m^{\frac{2}{3}}} + e \cdot \frac{-y\operatorname{sech}^2(f)e^g}{n}$

11. $\delta_x = \delta_x + \frac{\partial e}{\partial x}\delta_e = e^x \cdot \frac{-y\operatorname{sech}^2(f)e^g}{n}$

12. $\delta_b = \delta_b + \frac{\partial d}{\partial b}\delta_d = \frac{\cos(j)ok}{3l^2n^2m^{\frac{2}{3}}} + c^b \ln c \cdot \frac{1}{n}$
    $\delta_c = \delta_c + \frac{\partial d}{\partial c}\delta_d = bc^{b-1} \cdot \frac{1}{n}$

13. $\delta_a = \delta_a + \frac{\partial c}{\partial a}\delta_c = \frac{bc^{b-1}}{n}$

14. $\delta_x = \delta_x + \frac{\partial b}{\partial x}\delta_b = -\frac{e^{g+x}y\operatorname{sech}^2(f)}{n} + \frac{y\cos(j)ok}{6\sqrt{b}l^2n^2m^{\frac{2}{3}}} + y \cdot \frac{bc^{b-1}}{n}$
    $\delta_y = \delta_y + \frac{\partial b}{\partial y}\delta_b = \frac{-2oy}{3ln^2m^{\frac{2}{3}}} - \frac{e\operatorname{sech}^2(f)e^g}{n}\frac{x\cos(j)ok}{6\sqrt{b}l^2n^2m^{\frac{2}{3}}} + x \cdot \frac{c^b \ln c}{n}$

15. $\delta_x = \delta_x + \frac{\partial a}{\partial x}\delta_a = \frac{e^{g+x}y\operatorname{sech}^2(f)}{n} + \frac{y\cos(j)ok}{6\sqrt{b}l^2n^2m^{\frac{2}{3}}} + \frac{bc^{b-1}y}{n} + \cos(x) \cdot \frac{bc^{b-1}}{n}$

To determine the partial derivatives of $f(x, y)$ with regards to x and y at $(1, 1)$, δ_x and δ_y need to be evaluated at $(1, 1)$ resulting in

$$\delta_x = 1.3659399717387952 \text{ and } \delta_y = 1.3929338938392877.$$

These values agree with the symbolic differentiation above.

The ridge regression is a penalised linear model where a complexity parameter λ is applied to the sum of the squares of the regression coefficients. The objective of minimizing the risk when using squared error loss for a given λ is given by

$$R(\beta_0, \boldsymbol{\beta}, \lambda) = \sum_{i=1}^{N}(y_i - \beta_0 - \boldsymbol{\beta}'\mathbf{x}_i)^2 + \lambda\sum_{j=1}^{p}\beta_j^2$$

or written in matrix notation as

$$\mathbf{R} = (\mathbf{y} - \beta_0\mathbf{1} - \mathbf{X}\boldsymbol{\beta})'(\mathbf{y} - \beta_0\mathbf{1} - \mathbf{X}\boldsymbol{\beta}) + \lambda\boldsymbol{\beta}'\boldsymbol{\beta}.$$

This problem can be solved using least squares. This requires the gradients of $\mathbf{R}$ with respect to β_0 and $\boldsymbol{\beta}$ to be calculated. The gradients can calculated using symbolic differentiation yielding

$$\frac{\partial \mathbf{R}}{\partial \beta_0} = -2 \cdot \mathbf{1}'(\mathbf{y} - \beta_0\mathbf{1} - \mathbf{X}\boldsymbol{\beta})$$
$$\frac{\partial \mathbf{R}}{\partial \boldsymbol{\beta}} = -2\mathbf{X}'(\mathbf{y} - \beta_0\mathbf{1} - \mathbf{X}\boldsymbol{\beta}) + 2\lambda\boldsymbol{\beta}.$$

The gradients can also be calculated using backpropagation. Using the computation graph in Figure 2.2 the forward pass results in

1. $a = \boldsymbol{\beta}'\mathbf{x}_i$

2. $b = a + \beta_0$

3. $c = y_i - b$

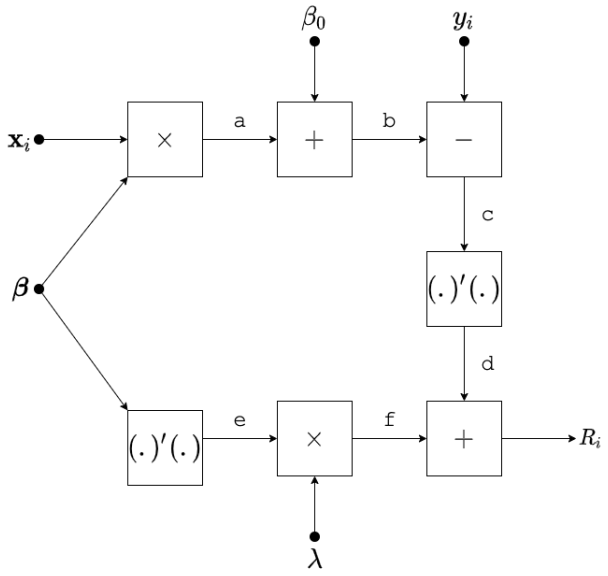

Figure 2.2: Computational Graph of Ridge Regression

4. $d = c^2$
5. $e = \boldsymbol{\beta}'\boldsymbol{\beta}$
6. $f = \lambda \times e$
7. $R_i = d + f$

Setting $\delta_a, \delta_b, \delta_c, \delta_d, \delta_e, \delta_f$ all equal to 0 and $\delta_{R_i} = 1$, the backwards pass results in

1. $\delta_f = \delta_f + \frac{\partial R_i}{\partial f}\delta_{R_i} = 1$
   $\delta_d = \delta_d + \frac{\partial R_i}{\partial d}\delta_{R_i} = 1$
2. $\delta_e = \delta_e + \frac{\partial f}{\partial e}\delta_f = \lambda$
3. $\delta_{\boldsymbol{\beta}} = \delta_{\boldsymbol{\beta}} + \frac{\partial e}{\partial \boldsymbol{\beta}}\delta_e = 2\boldsymbol{\beta} \cdot \lambda$
4. $\delta_c = \delta_c + \frac{\partial d}{\partial c}\delta_d = 2(y_i - \beta_0 - \boldsymbol{\beta}'\mathbf{x}_i)$
5. $\delta_b = \delta_b + \frac{\partial c}{\partial b}\delta_c = -2(y_i - \beta_0 - \boldsymbol{\beta}'\mathbf{x}_i)$
   $\delta_{\mathbf{y}} = \delta_{\mathbf{y}} + \frac{\partial c}{\partial \mathbf{y}}\delta_d = 2(y_i - \beta_0 - \boldsymbol{\beta}'\mathbf{x}_i)$
6. $\delta_a = \delta_a + \frac{\partial b}{\partial a}\delta_b = -2(y_i - \beta_0 - \boldsymbol{\beta}'\mathbf{x}_i)$
   $\delta_{\beta_0} = \delta_{\beta_0} + \frac{\partial b}{\partial \beta_0}\delta_b = -2(y_i - \beta_0 - \boldsymbol{\beta}'\mathbf{x}_i)$
7. $\delta_{\boldsymbol{\beta}} = \delta_{\boldsymbol{\beta}} + \frac{\partial a}{\partial \boldsymbol{\beta}}\delta_a = 2\lambda\boldsymbol{\beta}'\mathbf{1} - 2\mathbf{x}_i'(y_i - \beta_0 - \boldsymbol{\beta}'\mathbf{x}_i)$
   $\delta_{\mathbf{x}_i} = \delta_{\mathbf{x}_i} + \frac{\partial a}{\partial \mathbf{x}_i}\delta_a = -2\boldsymbol{\beta}'(y_i - \beta_0 - \boldsymbol{\beta}'\mathbf{x}_i)$

Thus, using chain rule, the gradients of R_i with regards to β_0 and $\boldsymbol{\beta}$ can be written in terms of the partial derivatives as

$$\frac{\partial R_i}{\partial \beta_0} = \frac{\partial R_i}{\partial d}\frac{\partial d}{\partial c}\frac{\partial c}{\partial b}\frac{\partial b}{\partial \beta_0} = -2(y_i - \beta_0 - \boldsymbol{\beta}'\mathbf{x}_i)$$

$$\frac{\partial R_i}{\partial \boldsymbol{\beta}} = \frac{\partial R_i}{\partial d}\frac{\partial d}{\partial c}\frac{\partial c}{\partial b}\frac{\partial b}{\partial a}\frac{\partial a}{\partial \boldsymbol{\beta}} = 2\lambda\boldsymbol{\beta}'\mathbf{1} - 2\mathbf{x}_i'(y_i - \beta_0 - \boldsymbol{\beta}'\mathbf{x}_i)$$

since $\mathbf{R} = \sum_{i=1}^{N} R_i$, the gradients $\frac{\partial R_i}{\partial \beta_0}$ and $\frac{\partial R_i}{\partial \boldsymbol{\beta}}$ can be written in terms of matrices as

$$\frac{\partial \mathbf{R}}{\partial \beta_0} = -2 \cdot \mathbf{1}'(\mathbf{y} - \beta_0\mathbf{1} - \mathbf{X}\boldsymbol{\beta})$$

$$\frac{\partial \mathbf{R}}{\partial \boldsymbol{\beta}} = 2\lambda\boldsymbol{\beta} - 2 \cdot \mathbf{X}'(\mathbf{y} - \beta_0\mathbf{1} - \mathbf{X}\boldsymbol{\beta})$$

Therefore the gradients derived through backpropagation results in the same gradients as the symbolic differentiation.

2.2 Question 2

Optimization is a central aspect to statistical learning. In many applications, there is a convex objective function whose parameters need to be optimised for a given input of data. For some cases this is simply achieved by determining the derivative through vector calculus, setting the derivative equal to zero and then solving the equation algebraically for the parameters. However, in many statistical learning cases it is no possible to set the derivative equal to zero and solve the equations algebrically. Thus, other methods of optimisation are required to determine the optimal parameters.

Of the many optimisation techniques, Newton's Method is one. The pure Newton's Method can be applied to solve the unconstrained, smooth optimisation problem

$$\min_x f(x)$$

where $f(x)$ is convex, twice differentiable, and the domain of $f(x) = \Re^n$ where x is the parameter that is to be optimised. Newton's Method starts with initial estimates for the parameters, and then repeatedly updates the parameters in the direction of the minimum of $f(x)$. This is done by applying the step

$$x^{(k)} = x^{(k-1)} - (\nabla^2 f(x^{(k-1)}))^{-1} \nabla f(x^{(k-1)}), k = 1, 2, 3, ...$$

where $x^{(k-1)}$ are the current parameter values, $f(x^{(k)})$ are the updated parameter values after step k, $\nabla f(x^{(k-1)})$ is the gradient of the $f(x)$ for the current parameter values, and $\nabla^2 f(x^{(k-1)})$ is the Hessian matrix of $f(x)$ for the current parameter values. The rational behind Newton's Method is that each update step minimizes the quadratic approximation

$$f(y) \approx f(x) + \nabla f(x)^T (y - x) + \frac{1}{2}(y - x)^T \nabla^2 f(x)(y - x)$$

over y. Solving the equation results in the $x^+ = x - (\nabla^2 f(x))^{-1} \nabla f(x)$, the update step.

Newton's Method dates back to the late 17th century as a method for finding the roots of polynomials. This was then extended to finding the roots of the gradients of non-linear functions. Thus the update step in Newton's Method can be seen as finding the direction v so that $\nabla f(x + v) = 0$. When letting $F(x) = \nabla f(x)$, linearizing $F(x)$ around v, via first-order approximation,

$$0 = F(x + v) \approx F(x) + DF(x)v$$

and solving for v results in $v = -(DF(x))^{-1} F(x) = -(\nabla^2 f(x))^{-1} \nabla f(x)$, the update step for Newton's Method.

The pure form of Newton's Method may not always converge. To ensure convergence, the backtracking line search is applied to Newton's Method where

$$x^+ = x - t(\nabla^2 f(x))^{-1} \nabla f(x)$$

is the update step. The step size $t = 1$ in the pure method. Step size t is determined by backtracking line search where $0 < \alpha \leq \frac{1}{2}, 0 < \beta \leq 1$. At each step, set $t = 1$ and while

$$f(x + tv) > f(x) + \alpha t \nabla f(x)^T v$$

the step size is shrunk by $t = \beta t$, else the Newton update step is performed with the current value of t. Here $v = -(\nabla^2 f(x))^{-1}\nabla f(x)$.

Newton's Method can be applied to linear regression with squared-error loss function in the following way. In order to minimize

$$\min_{\boldsymbol{\beta}} f(\mathbf{X}) = (\mathbf{X}\boldsymbol{\beta} - \mathbf{Y})^T(\mathbf{X}\boldsymbol{\beta} - \mathbf{Y})$$

both $\nabla f(\boldsymbol{\beta})$ and the Hessian matrix $\nabla^2 f(\boldsymbol{\beta})$ need to be calculated. For some functions, calculating the Hessian can be computationally intensive which is a major drawback of the Newton Method. However for the above function, the $\nabla f(\boldsymbol{\beta}) = 2\mathbf{X}^T(\mathbf{X}\boldsymbol{\beta} - \mathbf{Y})$ and $\nabla^2 f(\boldsymbol{\beta}) = 2\mathbf{X}^T\mathbf{X}$ can be easily calculated if $\mathbf{X}$ is not too large. Thus the update step becomes,

$$\boldsymbol{\beta}^+ = \boldsymbol{\beta} - (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T(\mathbf{X}\boldsymbol{\beta} - \mathbf{Y}).$$

When backtracking line search is applied to linear regression the update step becomes

$$\boldsymbol{\beta}^+ = \boldsymbol{\beta} - t(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T(\mathbf{X}\boldsymbol{\beta} - \mathbf{Y}).$$

Step size t is determined by backtracking line search where $0 < \alpha \leq \frac{1}{2}, 0 < \beta \leq 1$. At each step, set $t = 1$ and while

$$(\mathbf{X}\boldsymbol{\beta} + tv - \mathbf{Y})^T(\mathbf{X}\boldsymbol{\beta} + tv - \mathbf{Y}) > (\mathbf{X}\boldsymbol{\beta} - \mathbf{Y})^T(\mathbf{X}\boldsymbol{\beta} - \mathbf{Y}) + 2\alpha t(\mathbf{X}\boldsymbol{\beta} - \mathbf{Y})^T\mathbf{X}v$$

the step size is shrunk by $t = \beta t$, else the Newton update step is performed with the current value of t. Here $v = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T(\mathbf{X}\boldsymbol{\beta} - \mathbf{Y})$.

Newton's Method can also be applied to penalised linear regression that are twice differentiable such as the ridge regression. The objective for the ridge regression is

$$\min_{\boldsymbol{\beta}} f(\mathbf{X}) = (\mathbf{X}\boldsymbol{\beta} - \mathbf{Y})^T(\mathbf{X}\boldsymbol{\beta} - \mathbf{Y}) + \lambda\boldsymbol{\beta}^T\boldsymbol{\beta} \tag{2.1}$$

with gradient $\nabla f(\boldsymbol{\beta}) = 2\mathbf{X}^T(\mathbf{X}\boldsymbol{\beta} - \mathbf{Y}) + 2\lambda\boldsymbol{\beta}$ and Hessian Matrix $\nabla^2 f(\boldsymbol{\beta}) = 2\mathbf{X}^T\mathbf{X} + 2\lambda$. Thus, the update step with backtracking line search is given by

$$\boldsymbol{\beta}^+ = \boldsymbol{\beta} - t(\mathbf{X}^T\mathbf{X} + \lambda)^{-1}(\mathbf{X}^T(\mathbf{X}\boldsymbol{\beta} - \mathbf{Y}) + \lambda\boldsymbol{\beta}).$$

where step size t is determined by backtracking line search where $0 < \alpha \leq \frac{1}{2}, 0 < \beta \leq 1$, at each step, set $t = 1$ and while

$$(\mathbf{X}\boldsymbol{\beta} + tv - \mathbf{Y})^T(\mathbf{X}\boldsymbol{\beta} + tv - \mathbf{Y}) > (\mathbf{X}\boldsymbol{\beta} - \mathbf{Y})^T(\mathbf{X}\boldsymbol{\beta} - \mathbf{Y}) + 2\alpha t(\mathbf{X}^T(\mathbf{X}\boldsymbol{\beta} - \mathbf{Y}) + \lambda\boldsymbol{\beta})^T v$$

the step size is shrunk by $t = \beta t$, else the Newton update step is performed with the current value of t. Here $v = (\mathbf{X}^T\mathbf{X} + \lambda)^{-1}(\mathbf{X}^T(\mathbf{X}\boldsymbol{\beta} - \mathbf{Y}) + \lambda\boldsymbol{\beta})$.

Coordinate descent is another optimisation technique. Coordinate descent is considered a less complicated optimisation technique compared to Newton's Method, but can be used to optimise some non-differentiable functions. More general optimisation problems of the form

$$\min_x f(x) = g(x) + \sum_{i=1}^{n} h_i(x_i)$$

where g is convex and differentiable and each h_i is convex can be solved by coordinate descent. The method consists of setting $x^{(0)} \in \Re^n$ and repeating

$$x_i^{(k)} = \underset{x_i}{\operatorname{argmin}} f(x_1^{(k)}, ..., x_{i-1}^{(k)}, x_i, x_{i+1}^{(k-1)}, ..., x_n^{(k-1)}), \qquad i = 1, ..., n$$

for $k = 1, 2, 3, ...$ Thus the method consists of iteratively optimising the function for each parameter individually and repeating cycles through the parameters until convergence.

When applied to linear regression with squared-error loss function, in order to minimize

$$\min_{\boldsymbol{\beta}} f(\mathbf{X}) = (\mathbf{X}\boldsymbol{\beta} - \mathbf{Y})^T(\mathbf{X}\boldsymbol{\beta} - \mathbf{Y})$$

the objective function is optimised for each β_i from the vector of parameters $\boldsymbol{\beta}$ while all $\beta_j, j \neq i$ are kept fixed. This results in

$$0 = \nabla_i f(\boldsymbol{\beta}) = 2\mathbf{X}_i^T(\mathbf{X}\boldsymbol{\beta} - \mathbf{Y}) = 2\mathbf{X}_i^T(\mathbf{X}_i\beta_i + \mathbf{X}_{-i}\boldsymbol{\beta}_{-i} - \mathbf{Y})$$

thus each update step results in

$$\beta_i = \frac{\mathbf{X}_i^T(\mathbf{Y} - \mathbf{X}_{-i}\boldsymbol{\beta}_{-i})}{\mathbf{X}_i^T\mathbf{X}_i}$$

The update is repeated for $i = 1, 2, ..., p, 1, 2, ...$

Coordinate descent can also be applied to the ridge regression as above. In order to optimize equation 2.1, the equation is minimise over β_i as

$$0 = \nabla_i f(\boldsymbol{\beta}) = 2\mathbf{X}_i^T(\mathbf{X}_i\beta_i + \mathbf{X}_{-i}\boldsymbol{\beta}_{-i} - \mathbf{Y}) + 2\lambda\beta_i$$

which results in the update step

$$\beta_i = \frac{\mathbf{X}_i^T(\mathbf{Y} - \mathbf{X}_{-i}\boldsymbol{\beta}_{-i})}{\mathbf{X}_i^T\mathbf{X}_i + \lambda}$$

Furthermore, since coordinate descent only requires that the objective function be in the form $f(x) = g(x) + \sum_{i=1}^n h_i(x_i)$ where $g(x)$ is convex and smooth and $h_i(x_i)$ is convex, it can be applied to the lasso. The lasso objective function is

$$\min_{\boldsymbol{\beta}} f(\mathbf{X}) = (\mathbf{X}\boldsymbol{\beta} - \mathbf{Y})^T(\mathbf{X}\boldsymbol{\beta} - \mathbf{Y}) + \lambda\|\boldsymbol{\beta}\|_1$$

where $\|\boldsymbol{\beta}\|_1 = \sum_{i=1}^n |\beta_i|$ is the L_1-norm. Minimising over β_i results in

$$0 = \nabla_i f(\boldsymbol{\beta}) = 2\mathbf{X}_i^T(\mathbf{X}_i\beta_i + \mathbf{X}_{-i}\boldsymbol{\beta}_{-i} - \mathbf{Y}) + 2\lambda s_i$$

where $s_i \in \partial|\beta_i|$ is a sub-gradient of β_i. Thus the update step is given by soft-thresholding

$$\beta_i = S_{\lambda/(\mathbf{X}_i^T\mathbf{X})}\left(\frac{\mathbf{X}_i^T(\mathbf{Y} - \mathbf{X}_{-i}\boldsymbol{\beta}_{-i})}{\mathbf{X}_i^T\mathbf{X}_i}\right).$$

Below the performance of Newton's Method and Coordinate descent are compared along with simple gradient descent and gradient descent with backtracking line search. The optimisation techniques are compared in terms of the number of iterations until the mean squared error (MSE) converges and by the time it takes for the optimisation in seconds. The optimisation techniques are compared for simple linear regression and ridge regression, while coordinate descent is also compared for lasso regression. The data is simulated with $\mathbf{X} \sim N(\mathbf{0}, \mathbf{V})$ with $V = [v_{ij}] \in \Re^{20\times 20}$ and $v_{ii} = 1$ for $i = 1, 2, ..., 20$ and $v_{ij} = 0.7$ otherwise, and with

$\mathbf{Y} = \mathbf{X}\boldsymbol{\beta} + \epsilon$ with $\epsilon \sim N(0, 1)$ where $\beta_i \sim N(0, 1)$. The simulated data consists of 100 observations where the seed was set equal to 2021.

For the linear regression and ridge regression, the parameters for each optimisation technique are kept the same. The step size for the standard gradient descent is kept at 0.0001 to ensure convergence, the backtracking line search settings for the gradient descent with backtracking line search and Newton's Method are set to $\alpha = 0.2$ and $\beta = 0.7$ while $t = 0.1$ for gradient descent with backtracking line search and $t = 1$ for Newton's Method.

Comparing the different optimisation techniques when applied to linear regression, Newton's Method only took 1 iteration which shows the benefits of the closer quadratic approximation from using the Hessian matrix. Newton's method was also the fastest algorithm taking 0.004 seconds to converge. For the ridge regression with $\lambda = 10$, the number of iterations was also 1 iteration and Newton's method was the fastest method too. However, in this simulation, the size of the data was small and thus calculating the Hessian matrix is not too computationally intensive. Newton's method may not be as fast for larger data sets.

Coordinate descent took 12 iterations to converge, the second most after simple gradient descent. It was also the second slowest taking 0.016 seconds. Applied to the ridge regression, coordinate descent remains the second slowest method taking 0.01 seconds while taking the second most iterations with 4 iterations. When applied to the lasso also with $\lambda = 10$, coordinate descent takes 3 iterations and 0.026 seconds. These results are very similar between the ridge and lasso. All results can be seen in table 2.1.

Method	Iterations	Time
Linear Regression		
Gradient descent	288	0.057
Backtracking line search	4	0.005
Newton's method	1	0.004
Coordinate descent	12	0.016
Ridge Regression		
Gradient descent	103	0.029
Backtracking line search	3	0.005
Newton's method	1	0.001
Coordinate descent	4	0.01
Lasso Regression		
Coordinate descent	3	0.026

Table 2.1: Results of different optimisation techniques

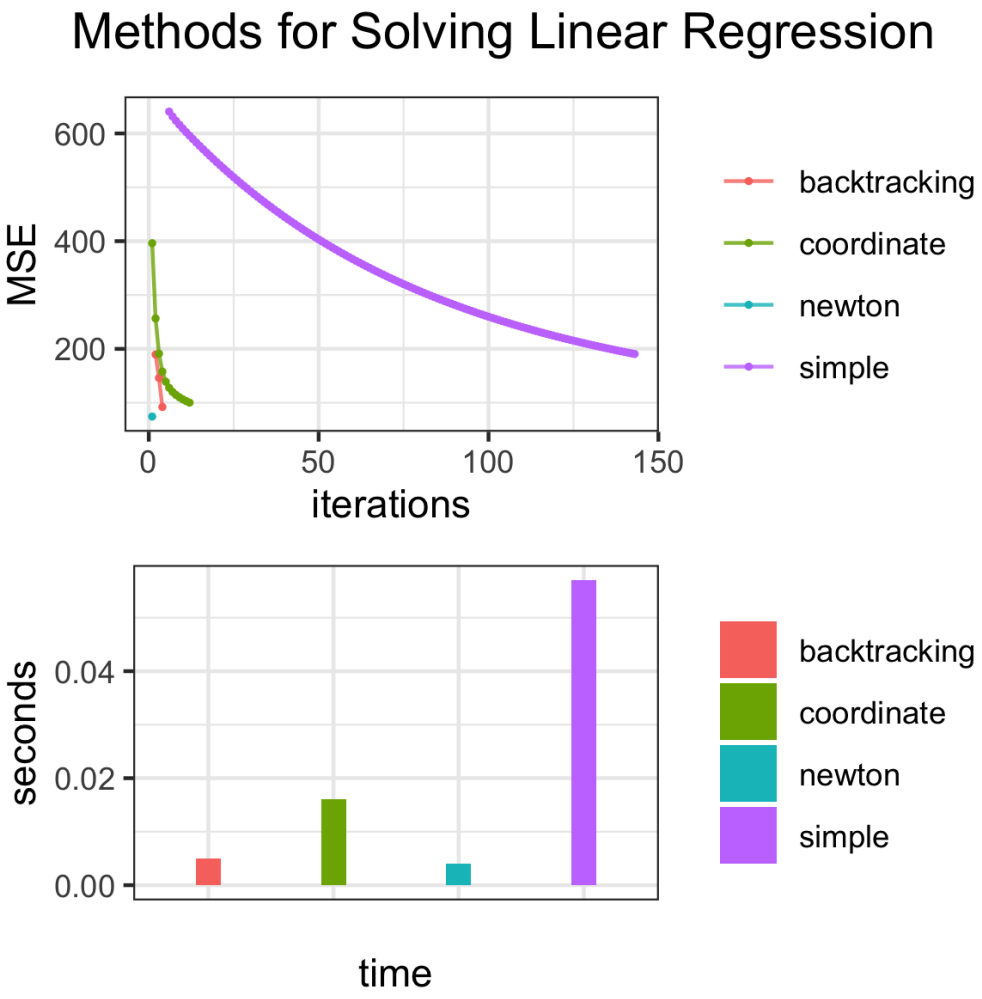

Figure 2.3: Methods for solving linear regression

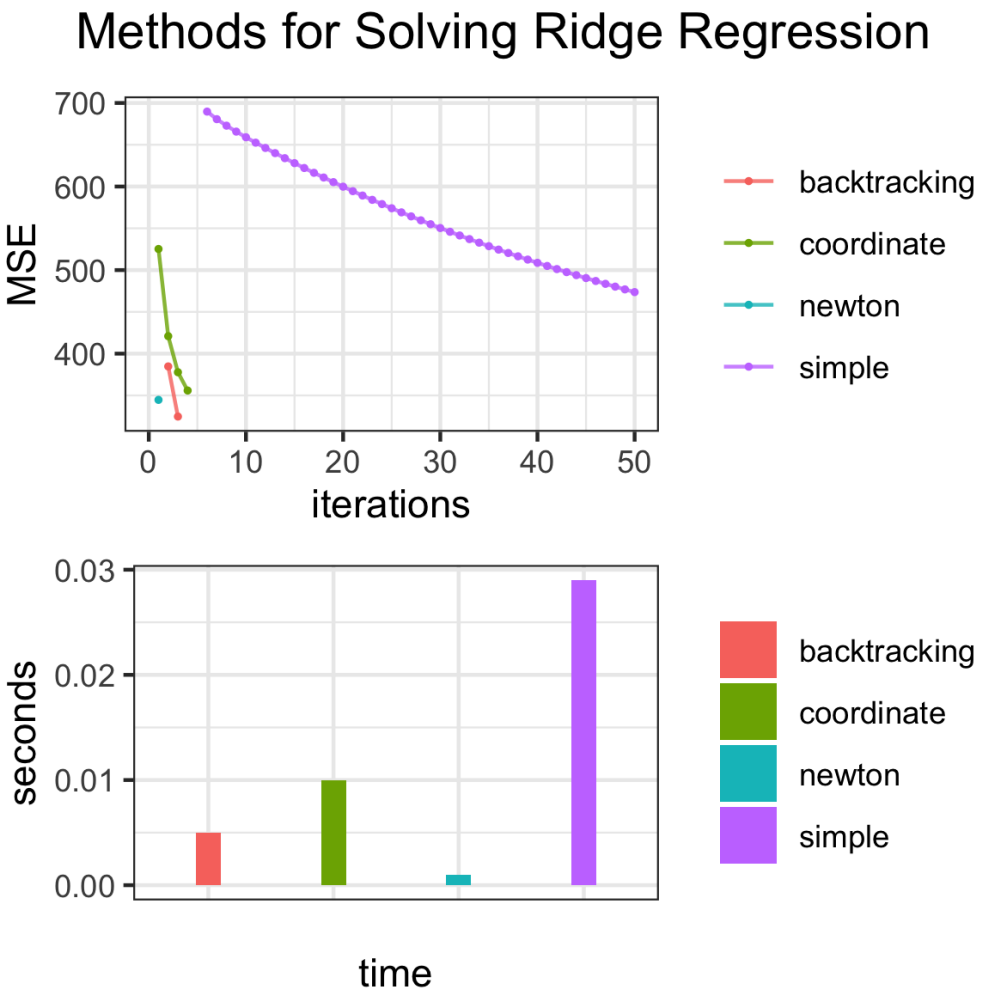

Figure 2.4: Methods for solving ridge regression

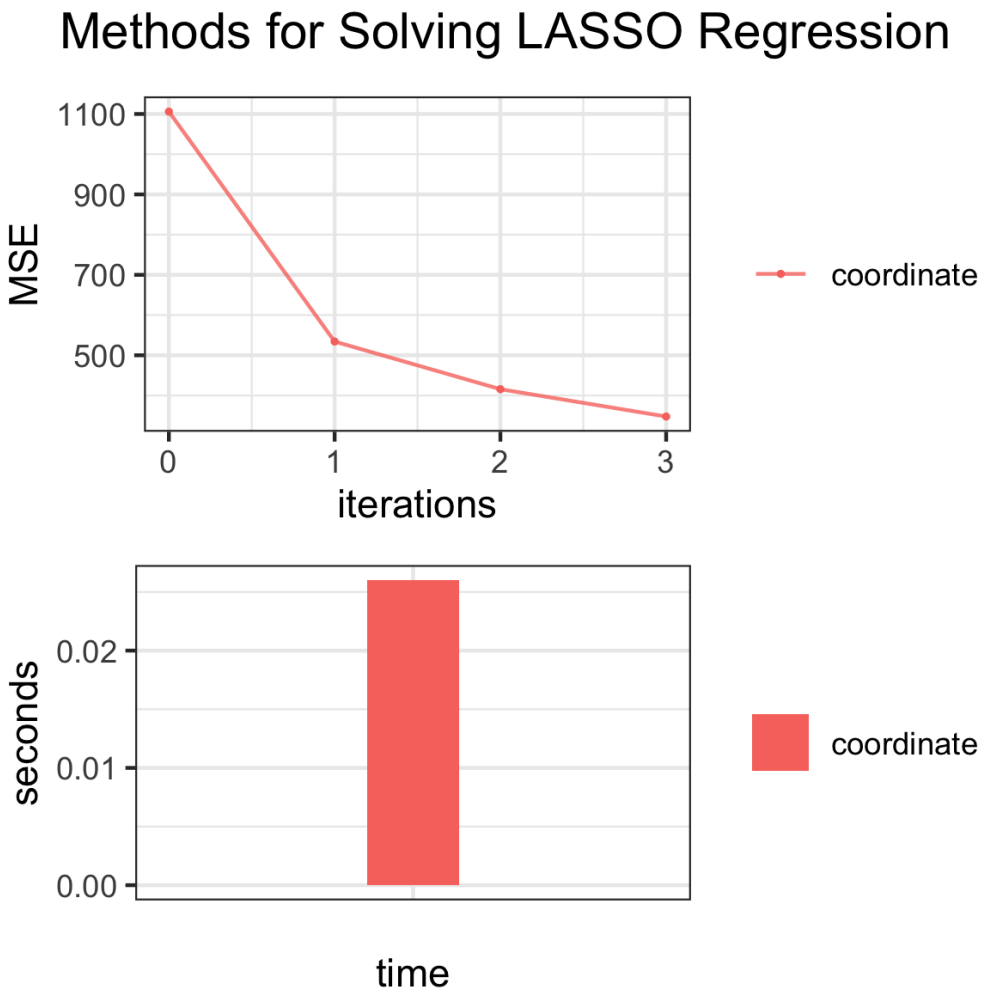

Figure 2.5: Coordinate descent for lasso regression

Assignment 3

3.1 Question 1

This report investigates the use of different model assessment and selection techniques using parametric and non-parametric regression techniques on small and large sample size datasets. The data used in this assignment are the Bike-Sharing data sourced from the UCI machine learning repository and comes from research by Fanaee-T & Gama (2013). The data consist of historic data from Capital Bikeshare system in Washington D.C., USA between 2011 and 2012. There are two datasets, a daily dataset and an hourly dataset. There are 11 variables consisting of 7 factor variables season, year, month, whether or not it is a public holiday, day of week, whether or not it is a working day, and weather conditions, and 4 numeric variables temperature, feeling temperature, humidity, and windspeed. The factor variables were expanded into dummy variables using one-hot-encoding one factor level removed per variable. The hourly data includes a factor variable for the hour and was also one-hot-encoded.

The data were used in the regression setting to predict the number of riders per day or per hour using the above mentioned variables. The daily dataset consists of 731 observations which can be considered a small sample while the hourly dataset consists of 17379 observations which can be considered a medium to large sample. For both datasets, the data were split into a training and test set broken up 75% to 25% respectively.

The two model assessment techniques used were the C_p statistic and K-Fold Cross-Validation (CV). These were chosen as to investigate their ability to best assess a technique in a small and large sample setting. The C_p statistic is considered an in-sample model assessment technique suited for small samples while CV is more suitable for large samples.

Model assessment is used to determine either which parameters for a certain model class result in the best generalization error or which model from a number of models result in the best generalization error. In most scenarios the generalization error given by

$$\mathrm{Err}(\hat{\mathbf{f}}(.;\mathcal{T})) = \mathbb{E}_{\mathbf{X},\mathbf{Y}}\left[\mathcal{L}(\mathbf{Y},\hat{\mathbf{f}}(\mathbf{X};\mathcal{T}))\right]$$

cannot be determined due to the underlying distribution not being available. While the training error given by

$$\overline{\mathrm{err}}(\hat{\mathbf{f}}(.;\mathcal{T})) = \frac{1}{N}\sum_{i=1}^{N}\mathcal{L}(\mathbf{y}_i,\hat{f}(\mathbf{x}_i;\mathcal{T}))$$

is a measure the predictive ability of a model. It does not indicate how the model will do on unseen data. Additionally, models with tuning parameters cannot use the training error to select the tuning parameters that result in the best generalization error. The generalization error measures the performance of the model on observations from out of the training data sample, known as extra-sample error. A measure for performance of the model on responses generated from the training inputs is called in-sample error. In-sample error is defined as

$$\begin{aligned}\mathrm{Err}_{\mathrm{in}} &= E_{\mathbf{y}^*}\Big[\frac{1}{N}\sum_{i=1}^{N}\mathcal{L}(y_i^*, \hat{f}(\mathbf{x}_i, \mathcal{T}))\Big|\mathcal{T}\Big] \\ &= \frac{1}{N}\sum_{i=1}^{N} E_{\mathbf{y}^*}[\mathcal{L}(y_i^*, \hat{f}(\mathbf{x}_i, \mathcal{T}))|\mathcal{T}]\end{aligned}$$

where $\mathbf{y}^*$ are responses generated from $P(.|\mathbf{X})$ independently of the $\mathbf{y}$ on which the model was trained.

The optimism of a model is defined as $\mathrm{op} \equiv \mathrm{Err}_{\mathrm{in}} - \overline{\mathrm{err}}$ and the average optimism is defined as $\omega \equiv E_{\mathbf{y}}[\mathrm{op}]$. The average optimism represents the average over all the training responses generated from $P(.|\mathbf{X})$. For regression with square-error loss, which was used in this case, the average optimism can be shown to be

$$\omega = \frac{2}{N}\sum_{i=1}^{N}\mathrm{Cov}(\hat{y}_i, y_i)$$

where $\hat{y}_i = E[\widehat{Y|\mathbf{X}_i = \mathbf{x}_i}]$. Thus in-sample error can be rewritten as

$$\begin{aligned}\mathrm{Err}_{\mathrm{in}} &= \overline{\mathrm{err}} + \omega \\ &= \overline{\mathrm{err}} + \frac{2}{N}\sum_{i=1}^{N}\mathrm{Cov}(\hat{y}_i, y_i).\end{aligned}$$

Under the assumption that the data follows an additive model defined as

$$y_i = f(\mathbf{x}_i) + \epsilon_i \text{ where } \epsilon_i \sim iid(0, \sigma^2)$$

when using a model from the class of linear smoothers, the predictions from our model can be written as $\hat{\mathbf{y}} = \mathbf{S}\mathbf{y}$ where $\mathbf{S}$ is independent of $\mathbf{y}$. For a linear smoother, the degrees of freedom of the model can be defined as $\mathrm{df}(\mathbf{S}) = \frac{1}{\sigma^2}\sum_{i=1}^{N}\mathrm{Cov}(\hat{y}_i, y_i) = \mathrm{trace}(\mathbf{S})$. Thus the in-sample error can be reformulated as $\mathrm{Err}_{\mathrm{in}} = \overline{\mathrm{err}} + 2\frac{\mathrm{df}(\mathbf{S})}{N}\sigma^2$ where σ^2 is the variance of a low bias model generally from the same class of models. This formulation of the in-sample error is known as the C_p statistic. The lower the C_p statistic, the better the model generalises from the training set.

K-fold Cross-Validation (CV) is a form of resampling model assessment technique where the training data $\mathcal{T}$ is split into K non-overlapping folds that are all approximately equal in size. Each fold is then used as a validation set for a model fit on the remaining folds as the training set. The folds are denoted $\mathcal{V}_k : k = 1, 2, ..., K$ such that $\bigcup_{k=1}^{K}\mathcal{V}_k = \mathcal{T}$ and $\mathcal{V}_k \bigcap \mathcal{V}_l = \emptyset$. While $\mathcal{T}_k = \mathcal{T}\backslash\mathcal{V}_k$ represents the training data without the k^{th} fold. Using the indexing function $\kappa(i)$ such that $\kappa(i) = k$ if the i^{th} observation belongs to the k^{th} fold, the model from the class of learning algorithms with tuning parameter indexed by γ fitted to $\mathcal{T}_k$ is denoted by $\hat{f}_\gamma^{-k}(.)$. Thus the CV error for the model with tuning parameter γ is defined as

$$\mathrm{CV} = \frac{1}{N}\sum_{i=1}^{N}\mathcal{L}(y_i, \hat{f}_\gamma^{-\kappa(i)}(\mathbf{x}_i)).$$

Here the fitted model is evaluated on unseen observations and is similar to the test error. Thus the CV error should be a better approximation for the generalization error than the training error. CV is used for model assessment and selection by determining

$$\hat{\gamma} = \underset{\gamma \in \Gamma}{\operatorname{argmin}}\{\mathrm{CV}(\gamma)\}$$

then refitting the model for the given value of γ using the entire training set $\mathcal{T}$. The final fitted model is then $\hat{f}_{\hat{\gamma}}(.;\mathcal{T})$.

The learning methods used were the lasso and K-Nearest Neighbours (KNN). The lasso is a shrinkage method defined as

$$\hat{\boldsymbol{\beta}}^{lasso} = \underset{\boldsymbol{\beta}}{\operatorname{argmin}} \sum_{i=1}^{N} \left(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j \right)^2$$
$$\text{subject to } \sum_{j=1}^{p} |\beta_j| \leq t$$

and can be written in the Lagrangian form as

$$\hat{\boldsymbol{\beta}}^{lasso} = \underset{\boldsymbol{\beta}}{\operatorname{argmin}} \left\{ \frac{1}{2} \sum_{i=1}^{N} \left(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j \right)^2 + \lambda \sum_{j=1}^{p} |\beta_j| \right\}.$$

The lasso uses the L_1 penalty on the size of the β parameters to act as form of subset selection since for certain values of λ the β parameters become zero. Since the solution is nonlinear in y_i, a number of algorithms have been developed to compute the lasso solution including the Least Angle Regression (LARS) algorithm which was used here. The LARS algorithm including the lasso modification is a follows

---

Algorithm 2: Least Angle Regression: Lasso Modification

---

1. Standardize the predictors to have mean zero and unit norm. Start with the residual $\mathbf{r} = \mathbf{y} - \bar{\mathbf{y}}, \beta_1, \beta_2, ..., \beta_p = 0$

2. Find the predictor $\mathbf{x}_j$ most correlated with $\mathbf{r}$.

3. Move β_j from 0 towards its least-squares coefficient $\langle \mathbf{x}_j, \mathbf{r} \rangle$, until some other competitor $\mathbf{x}_k$ has as much correlation with the current residual as does $\mathbf{x}_j$.

4. (a) Move β_j and β_k in the direction defined by their joint least squares coefficient of the current residual on $(\mathbf{x}_j, \mathbf{x}_k)$, until some other competitor $\mathbf{x}_l$ has as much correlation with the current residual.

4. (b) If a non-zero coefficient hits zero, drop its variable from the active set of variables and recompute the current joint least squares direction.

5. Continue in this way until all p predictors have been entered. After $\min(N-1, p)$ steps, we arrive at the full least-squares solution.

(Hastie et al. 2017)

Algorithm 2 computes the entire solution path which provides insights to the importance of each variables. The degrees of freedom for the lasso is determined by the number of non-zero predictors in the model. Thus the C_p statistic can be determined for any value of λ.

KNN is a non-parametric learning method that relies on predicting a new observation based on the average the response of the K nearest observations to the new observation. The nearest observations is determined by Euclidean distance given by

$$D(\mathbf{x}_i, \mathbf{x}_k) = \sqrt{\sum_{j=1}^{p}(x_{ij} - x_{kj})^2}.$$

KNN generally requires large sample sizes for good performance. This is since increasing the number of features results in the curse of dimensionality causing the distance between observations to increase. The degrees of freedom for KNN is given by $\frac{N}{K}$ which is used to determine the C_p statistic. The $\hat{\sigma}^2$ used in the C_p statistic for KNN was determined by the MSE of a KNN model with $k = 5$ here.

Both the lasso and KNN were fit to the daily data training dataset and hourly data training data set. The C_p statistic and CV were then separately used to determine the best value for λ for the lasso and the best value for k for the KNN models. This resulted in 8 models. Each of the models were fit to the entire training dataset from the daily and hourly data with the best tuning parameters. The fitted models were used to predict the responses from the test dataset for the daily and hourly data sets respectively. The MSE were then compared to determine which model assessment technique and which model best predicted new observations.

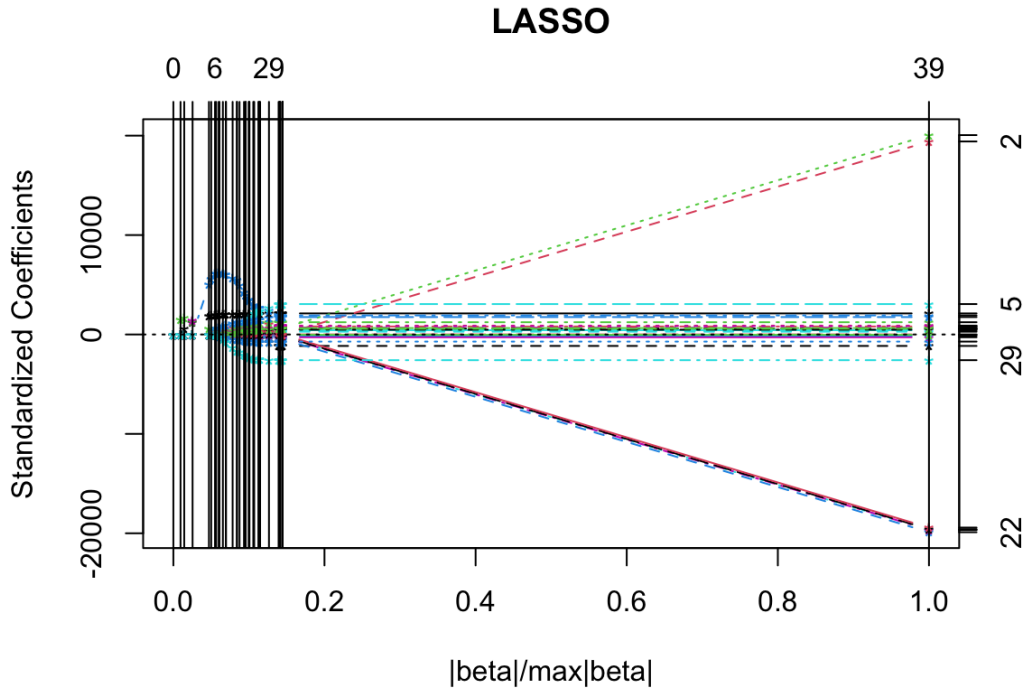

Figure 3.1: Lasso solution path using LARS algorithm for daily dataset

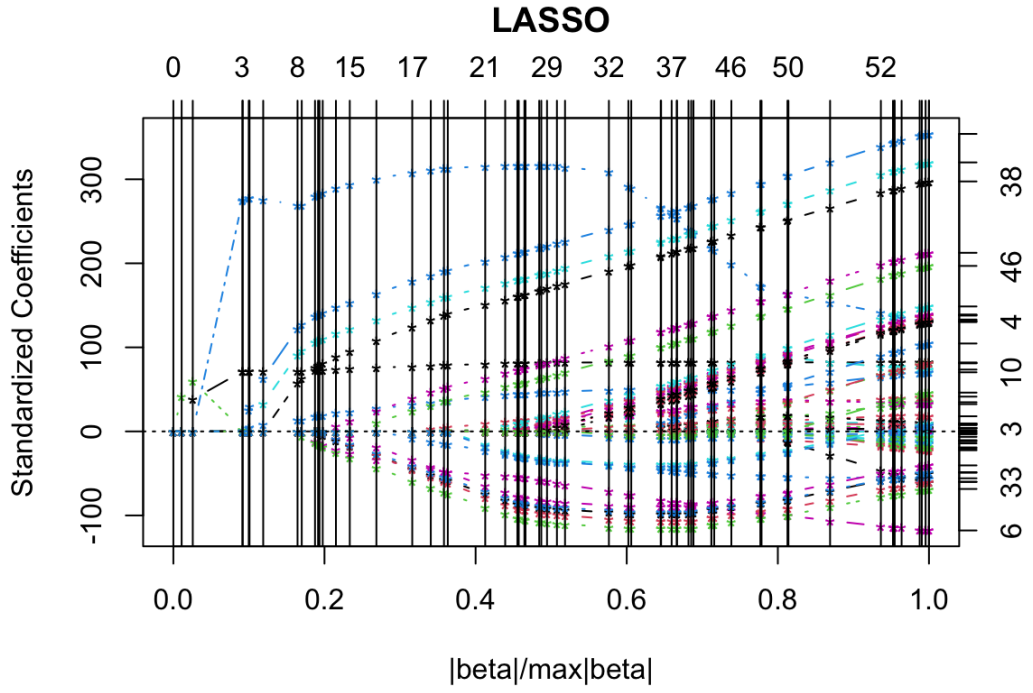

Figure 3.2: Lasso solution path using LARS algorithm for hourly dataset

The solution paths for the lasso models determined using the C_p statistic are given in figures 3.1 and 3.2. For the daily data the large values of λ does not result in many zero coefficients while for the hourly data across the range of λ values coefficients are forced to zero. Figures 3.3 to 3.6 show the C_p statistic and CV error for the different values of K. From table 3.1 for the daily dataset KNN selected $k = 14$ and for the hourly dataset the C_p statistic selected $k = 4$ while CV selected $k = 3$.

For the daily data, the lasso model that used the C_p statistic resulted in the lowest MSE from the test set. This could be due to the daily dataset being a small sample size which could have suited the lasso compared to KNN as described above. The C_p statistic could have offered the best selection of λ since it is an in-sample technique that penalizes higher degrees of freedom. Furthermore, CV results in high variance in small samples due to the number of observations in the validation folds being too small. For the hourly dataset, the KNN model using CV as a model selection technique for K resulted in the lowest test MSE. This could be due to the larger sample suiting the flexibility of KNN. Additionally, CV outperformed the C_p statistic for both the lasso and KNN possibly due to the large sample size resulting in validation folds large enough to estimate an accurate generalization error. From this it can be determined that different model assessment techniques are suited to different settings dependent on the types of models used, the sample size and the nature of the data.

	Method	Data	Selection	Best Tuning Parameter	MSE
1	LASSO	day	Cp	0.06	846012.81
2	LASSO	day	CV	1.28	847953.91
3	LASSO	hour	Cp	0.06	10615.47
4	LASSO	hour	CV	0.07	10562.87
5	KNN	day	Cp	14.00	1237271.00
6	KNN	day	CV	14.00	1237271.00
7	KNN	hour	Cp	4.00	9013.38
8	KNN	hour	CV	3.00	8812.45

Table 3.1: Performance of Best Models on Test Dataset

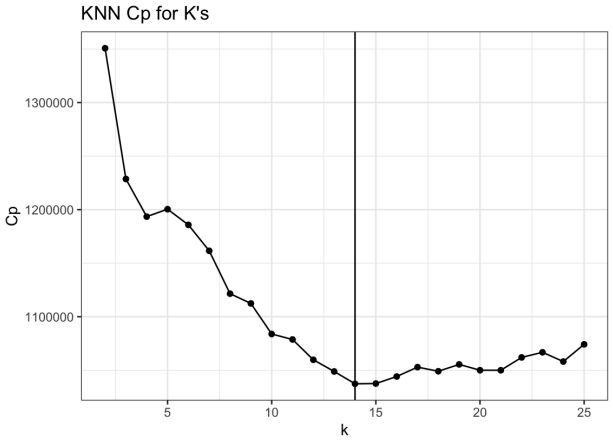

Figure 3.3: C_p values fordifferent K's for daily data

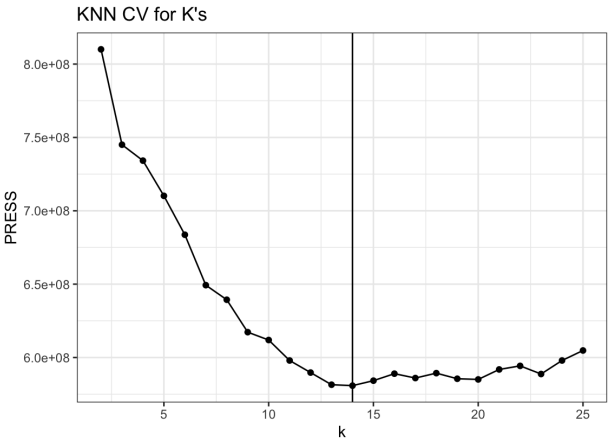

Figure 3.4: CV error for different K's for daily data

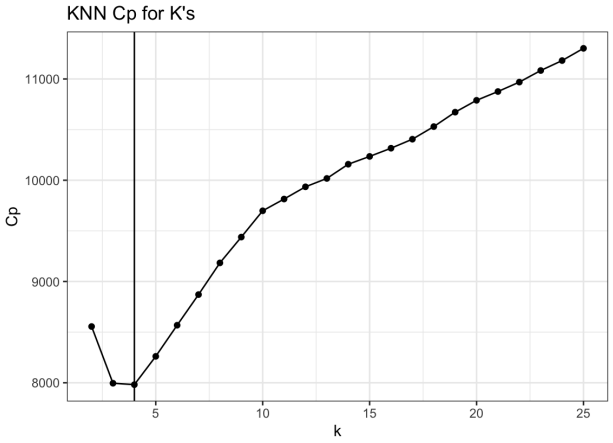

Figure 3.5: C_p values fordifferent K's for hourly data

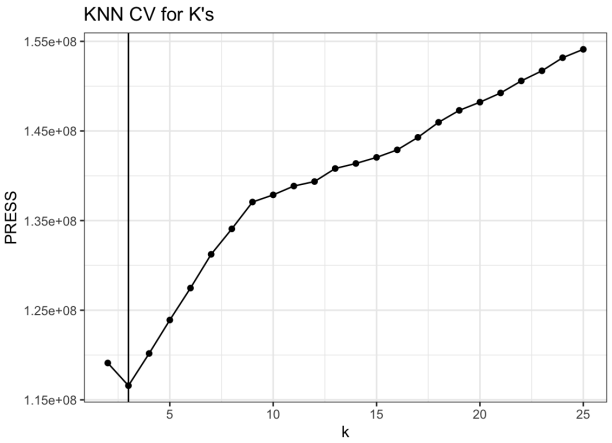

Figure 3.6: CV error for different K's for hourly data

Assignment 4

4.1 Question 1

Decision tree models consist of a sequence of yes/no questions that form a tree topology. The tree is traversed by answering the questions until a terminal node is reached containing the information needed for the decision. The nature of the tree topology's simple yes/no questions results in a non-linear model that is highly interpretable. Decision trees can be easily displayed in a graph of the tree topology. Additionally, they result in high predictive power for a class of models that do not require extensive hyperparameter tuning. Thus they are amongst the best "off-the-shelf" models.

Decision trees are a particular case of a class of models of the form

$$f(\mathbf{x}, \mathcal{W}, c(.)) = \sum_{\mathfrak{R} \in \mathcal{W}} c(\mathfrak{R}) I(\mathbf{x} \in \mathfrak{R})$$

where $\mathcal{W}$ contains mutually exclusive and exhaustive subsets of the feature space and $c(.) : \mathcal{W} \to \mathfrak{R}$. While estimating $c(.)$ given $\mathcal{W}$ is straightforward, the number of partitions to consider makes finding $\mathcal{W}$ difficult. Decision trees are a special case where $\mathcal{W}$ consists of rectangular partitions of the feature space. The regions in $\mathcal{W}$ used for decision making, denoted by $\mathfrak{R}$, are given by the terminal nodes of a tree topology, with a tree topology denoted by $\mathfrak{T}$. The internal and terminal nodes of a tree topology are denoted by $\text{Int}(\mathfrak{T})$ and $\text{Ter}(\mathfrak{T})$ respectively. The response in a decision tree is modeled by a constant that in the terminal region of the tree topology following the function $c(.) : \text{Ter}(\mathfrak{T}) \to \Re$, which is to be estimated. A fitted decision tree used for making decision can be modelled as

$$\text{T}(\mathbf{x}; \mathfrak{T}, c(.)) = \sum_{\mathfrak{R} \in \text{Ter}(\mathfrak{T})} c(\mathfrak{R}) I(\mathbf{x} \in \mathfrak{R}).$$

In a scalar response setting, the estimation of the topology and function $c(.)$ of a decision tree is guided by a loss function $\mathcal{L}(.,.)$ with the empirical risk formulated by

$$R(\mathbf{\Theta}) = \sum_{i=1}^{N} \mathcal{L}\Big(y_i, \text{T}(\mathbf{x}_i; \mathbf{\Theta})\Big) = \sum_{\mathfrak{R} \in \text{Ter}} \sum_{\mathbf{x}_i \in \mathfrak{R}} \mathcal{L}(y_i, c(\mathfrak{R})) \tag{4.1}$$

where $\mathbf{\Theta} = (\mathfrak{T}, c(.))$ and $\mathcal{T} = \{\mathbf{x}_i, y_i\}_{i=1}^{N}$. The objective of a decision tree is to minimize the empirical risk given in equation 4.1. The primary challenge of a decision tree is to determine the regions, $\mathfrak{T}$, since the constants in

equation 4.1 can be minimised out by computing

$$\hat{c}(\mathfrak{R}) = \underset{c}{\operatorname{argmin}} \left\{ \sum_{\mathbf{x}_i \in \mathfrak{R}} \mathcal{L}(y_i, c) \right\}$$

for a given $\mathfrak{T}$.

Determining the regions for a decision tree is infeasible as there are too many partitions to enumerate over. Thus the Classification and Regression Tree (CART) algorithm approximates the regions through the process of recursive binary partitioning (RBP). RBP works by splitting the terminal node into two child nodes such that the reduction in the empirical risk is maximised. CART specifies the split of a region, $\mathfrak{R}$, into two child nodes through a splitting variable and a splitting point. A candidate split of a region into two child nodes is given by

$$\mathcal{S}^{-}_{j,s}(\mathfrak{R}) = \{\mathbf{x} \in \mathfrak{R} : x_j \leq s\}$$
$$\mathcal{S}^{+}_{j,s}(\mathfrak{R}) = \{\mathbf{x} \in \mathfrak{R} : x_j > s\}$$

respectively. From this, the decrease in the empirical risk from splitting the terminal node $\text{Ter}(\mathfrak{T})$ with region $\mathfrak{R}$ into the two child nodes following splitting variable j and splitting point s is as follows

$$\begin{aligned}
\Delta \text{R}(j, s; \mathfrak{R}) = & \sum_{\mathbf{x}_i \in \mathfrak{R}} \mathcal{L}\Big(y_i, \hat{c}(\mathfrak{R})\Big) - \sum_{\mathbf{x}_i \in \mathcal{S}^{-}_{j,s}(\mathfrak{R})} \mathcal{L}\Big(y_i, \hat{c}\big(\mathcal{S}^{-}_{j,s}(\mathfrak{R})\big)\Big) \\
& - \sum_{\mathbf{x}_i \in \mathcal{S}^{+}_{j,s}(\mathfrak{R})} \mathcal{L}\Big(y_i, \hat{c}\big(\mathcal{S}^{+}_{j,s}(\mathfrak{R})\big)\Big) \\
& = n(\mathfrak{R}) \cdot Q(\mathfrak{R}) - n(\mathcal{S}^{-}_{j,s}) \cdot Q(\mathcal{S}^{-}_{j,s}) \\
& - n(\mathcal{S}^{+}_{j,s}) \cdot Q(\mathcal{S}^{+}_{j,s})
\end{aligned}$$

where $n(\mathfrak{R}) = \sum_{i=1}^{N} \text{I}(\mathbf{x}_i \in \mathfrak{R})$ and $Q(\mathfrak{R}) = \frac{1}{n(\mathfrak{R})} \sum_{\mathbf{x}_i \in \mathfrak{R}} \mathcal{L}((y_i, \hat{c}(\mathfrak{R}))$. The quantity given by $Q(\mathfrak{R})$ is the node impurity for the region $\mathfrak{R}$. The RBP seeks to find the splitting variable and splitting point

$$(j(\mathfrak{R}), s(\mathfrak{R})) = \underset{(j,s)}{\operatorname{argmax}} \Big\{ d(j, s, \mathfrak{R}) \Big\}.$$

Rather than considering all values over the range for a given variable j, only the $s \in \{x_{ij} : i \in \mathcal{I}(\mathfrak{R})\}$ are considered with $\mathcal{I}(\mathfrak{R}) = \{i : \mathbf{x}_i \in \mathfrak{R}\}$. The RBP algorithm for a particular node is given in algorithm 3

Algorithm 3: Recursive Binary Partitioning

Consider a terminal node with region $\mathfrak{R}$. Set $d^{(\max)} = 0$

1. for $j = 1, 2, ..., p$

   for $i \in \mathcal{I}(\mathfrak{R})$

   if $\Delta R(j, x_{ij}; \mathfrak{R}) > d^{(\max)}$ set $j(\mathfrak{R}) = j$, $s(\mathfrak{R}) = x_{ij}$ and

   $d^{(\max)} = \Delta R(j, x_{ij}; \mathfrak{R})$

2. return $j(\mathfrak{R}), s(\mathfrak{R})$

---

and the CART algorithm for balanced splitting is given in algorithm 4

Algorithm 4: CART with balanced splitting

Initialize $\mathfrak{T}$ to contain one node with a region containing all the data (a stump containing $\{\mathbf{x} : \mathbf{x} \in \Re^p\}$)

1. Set $\mathcal{A} = \{\mathfrak{R} \in \text{Ter}(\mathfrak{T}) : \text{stopping condition is true}\}$
2. If $\mathcal{A} = \text{Ter}(\mathfrak{T})$ break
3. for $\mathfrak{R} \in \text{Ter}(\mathfrak{T})$ $\mathcal{A}$

   add two child nodes to the node in $\mathfrak{T}$ represented by $\mathfrak{R}$ with $\mathfrak{R}^-(\mathfrak{R})$ and $\mathfrak{R}^+(\mathfrak{R})$.
4. Repeat from step 1.

Algorithm 5: CART with unbalanced splitting

Initialize $\mathfrak{T}$ to contain one node with a region containing all the data (a stump containing $\{\mathbf{x} : \mathbf{x} \in \Re^p\}$)

1. Set $\mathcal{A} = \{\mathfrak{R} \in \text{Ter}(\mathfrak{T}) : \text{stopping condition is true}\}$
2. if $\mathcal{A} = \text{Ter}(\mathfrak{T})$ break, else set $d^{(\max)} = 0$
3. for $\mathfrak{R} \in \text{Ter}(\mathfrak{T})$ $\mathcal{A}$ if $d(\mathfrak{R}) > d^{(\max)}$ set $d^{(\max)} = d(\mathfrak{R})$ and $\mathfrak{R}_{add} = \mathfrak{R}$
4. add two child nodes to the node in $\mathfrak{T}$ represented by $\mathfrak{R}_{add}$ with $\mathfrak{R}^-(\mathfrak{R})$ and $\mathfrak{R}^+(\mathfrak{R})$.
5. Repeat from step 1.

and the CART algorithm for unbalanced splitting is given in algorithm 5

where $\mathfrak{R}^-(\mathfrak{R}) = \mathcal{S}^-_{j(\mathfrak{R}),s(\mathfrak{R})}(\mathfrak{R})$ and $\mathfrak{R}^+(\mathfrak{R}) = \mathcal{S}^+_{j(\mathfrak{R}),s(\mathfrak{R})}(\mathfrak{R})$. The stopping conditions mentioned above can be one of the following

1. A terminal node must have a minimum number of observations to be considered for a split.
2. The number of observations in a terminal node resulting from a split may not be less than a minimum number.
3. A split must reduce the empirical risk by a certain minimum amount.

Decision trees can be trained on data using RBP following either the balanced or unbalanced CART algorithm. The resulting fitted tree is denoted by $\text{T}(\mathbf{x}; \hat{\mathbf{\Theta}}(\mathcal{T})$.

In order to estimate a decision tree for a regression or classification problem specific loss functions $\mathcal{L}(.,.)$ and constant functions $c(.)$ must be specified. For regression problems, squared-error loss

$$\mathcal{L}(Y, f(\mathbf{X})) = (Y - f(\mathbf{X})^2$$

with constant function

$$\hat{c}(\mathfrak{R}) = \bar{y}(\mathfrak{R}) = \frac{1}{n(\mathfrak{R})} \sum_{\mathbf{x}_i \in \mathfrak{R}} y_i.$$

The absolute loss function given by

$$\mathcal{L}(Y, f(\mathbf{X})) = |Y - f(\mathbf{X}|$$

with constant function

$$\hat{c}(\mathfrak{R}) = \text{med}(y_i : \mathbf{x}_i \in \mathfrak{R})$$

can also be used.

For classification problems, a number of loss functions can be used. The following are common

1. 0-1 loss: $\mathcal{L}(Y, f(\mathbf{X})) = \text{I}(Y = f(\mathbf{X})$
2. cross-entropy loss: $\mathcal{L}(Y, f(\mathbf{X})) = -\sum_{k=1}^{K} \log(p_k(\mathbf{X}))\text{I}(Y = k)$ with $p_k(\mathbf{X}) = \frac{e^{f_k(\mathbf{X})}}{\sum_{l=1}^{K} e^{f_l(\mathbf{X})}} : k = 1, 2, ..., K$

When using $\hat{p}_k(\mathbf{X}) = \frac{1}{n(\mathfrak{R})} \sum_{\mathbf{x}_i \in \mathfrak{R}} \text{I}(y_i = k)$, the node impurity for the above loss functions are given by

1. 0-1 loss: $Q(\mathfrak{R}) = 1 - \max_k\{\hat{p}_k(\mathfrak{R})\}$
2. cross-entropy loss: $Q(\mathfrak{R}) = -\sum_{k=1}^{K} \hat{p}_k(\mathfrak{R}) \log(\hat{p}_k(\mathfrak{R}))$.

An alternative to classifying an observations by the majority class of the region, is to provide a probability of the observation belonging to a class, $\hat{p}_k(\mathfrak{R})$. From this, the Gini index is derived as the expected impurity given by

$$Q(\mathfrak{R}) = \sum_{k \neq k'} \hat{p}_k(\mathfrak{R})\hat{p}_{k'}(\mathfrak{R}) = \sum_{k=1}^{K} \hat{p}_k(\mathfrak{R})[1 - \hat{p}_k(\mathfrak{R})].$$

The use of the Gini index or cross entropy results in purer splits than simply using misclassification error. In order to avoid overfitting of the training data, a form of regularization is possible by using cost-complexity pruning. This works similarly to the C_p value and prefers a more parsimonious model to a very deep complex tree.

While decision trees are easy to implement for a non-linear model and still offer high predictive powers while remaining interpretable, they are inherent problems with the model. Decision trees make greedy splits that may seem like the best split at the terminal node but may cause better splits deeper in the tree to be missed. A major problem with deep decision trees are their inherent high variance and instability. Slight changes to data may cause very different initial splits which then cause the remaining tree topology to be drastically different between data. This problem reduces predictive performance on new unseen data through the bias-variance trade-off. Additionally, the nature of the RBP results in non-smooth prediction surfaces which are an undesirable property.

A method developed to overcome the shortcomings of decision trees and improve their predictive power is through the use of bootstrap aggregating or bagging. Bagging works on the principle that combining the predictions from a number of uncorrelated high-variance decision trees improves the predictive performance of the model. Bagging attempts to approximate the aggregate predictor

$$f_{ag}(\mathbf{x}) = \mathbb{E}_{\mathcal{T}}\left[\hat{f}(\mathbf{x}; \mathcal{T})\right]$$

where the training data $\mathcal{T} = \{(\mathbf{x}_i, y_i) : i = 1, 2, ..., N\}$ are iid realisations from the density function $\mathbb{P}_{Y,\mathbf{X}} = (.,.)$ and the base learning algorithm, potentially a decision tree, is given by $\hat{f}(.; \mathcal{T})$. While the true density is unavailable, it can be approximated through the empirical density given by

$$\hat{\mathbb{P}}_{Y,\mathbf{X}}(y_i, \mathbf{x}_i) = \frac{1}{N} \text{ for } i = 1, 2, ..., N.$$

The aggregate predictor can be approximate by the ideal bootstrap predictor given by

$$\hat{f}_{bag}(\mathbf{x};\mathcal{T}) = \mathbb{E}_{\mathcal{T}^*\sim\hat{\mathbb{P}}_{Y,\mathbf{X}}}\Big[\hat{f}(\mathbf{x};\mathcal{T}^*)\Big]$$

where the data $\mathcal{T}^*$ is obtained from N independent draws from $\hat{\mathbb{P}}_{Y,\mathbf{X}}(.,.)$. Since computing $\hat{f}_{bag}(\mathbf{x};\mathcal{T})$ is infeasible, the Monte Carlo approximation is used given by

$$\hat{f}^{(B)}_{bag}(\mathbf{x};\mathcal{T}) = \frac{1}{B}\sum_{b=1}^{B}\hat{f}(\mathbf{x};\mathcal{T}^*_b)$$

where $\mathcal{T}^*_b : b = 1, 2, ..., B$ are B independent draws of size N with replacement from $\mathcal{T}$.

Bagging decision trees improve the predictive performance of decision trees through their reduction of the variance from a single decision tree. This reduction follows from the bias-variance decomposition of the point-wise mean-squared error for a decision tree trained by RBP, $\mathrm{T}(\mathbf{x};\hat{\mathbf{\Theta}}(\mathcal{T}))$, given by

$$\begin{aligned}\mathrm{MSE}\Big[\mathrm{T}(\mathbf{x};\hat{\mathbf{\Theta}}(.))\Big] &= \mathbb{E}_{\mathcal{T},Y}\Big[\big(Y - \mathrm{T}(\mathbf{x};\hat{\mathbf{\Theta}}(\mathcal{T}))\big)^2|\mathbf{X}=\mathbf{x}\Big]\\ &= \Big(\mathrm{bias}_{tree}(\mathbf{x})\Big)^2 + \sigma^2_{tree}(\mathbf{x})\end{aligned}$$

where $\mathrm{bias}_{tree}(\mathbf{x}) = f^{**}(\mathbf{x}) - \mathbb{E}_{\mathcal{T}}[\mathrm{T}(\mathbf{x};\hat{\mathbf{\Theta}}(\mathcal{T}))]$ and $\sigma^2_{tree}(\mathbf{x}) = \mathbb{V}_{\mathcal{T}}[\mathrm{T}(\mathbf{x};\hat{\mathbf{\Theta}}(\mathcal{T})]$. The bagged predictor with a decision tree base learner given by

$$\hat{f}^{(B)}_{bag}(\mathbf{x};\mathcal{T}) = \frac{1}{B}\sum_{b=1}^{B}\mathrm{T}\big(\mathbf{x};\hat{\mathbf{\Theta}}(\mathcal{T}^*_b)\big).$$

Applying the bias-variance decomposition to the point-wise mean-squared error of the bagged decision tree, results in the same bias of a single decision tree but different variance. For iid random variable $X_b : b = 1, 2, ..., B$ with variance σ^2 and pairwise correlation $\mathrm{cor}(X_b, X_c) = \rho$ for $b \neq c$, the following stands

$$\mathbb{V}\Big(\frac{1}{B}\sum_{b=1}^{B}X_b\Big) = \rho\sigma^2 + \frac{1-\rho}{B}\sigma^2.$$

From the above, and using the following definitions for a bagged decision tree

1. $\mathrm{bias}_{tree}(\mathbf{x}) = f^{**}(\mathbf{x}) - \mathbb{E}_{\mathcal{T}^*}[\mathrm{T}(\mathbf{x};\hat{\mathbf{\Theta}}(\mathcal{T}^*))]$
2. $\sigma^2_{tree}(\mathbf{x}) = \mathbb{V}_{\mathcal{T}^*}[\mathrm{T}(\mathbf{x};\hat{\mathbf{\Theta}}(\mathcal{T}^*)]$
3. $\rho_{tree}(\mathbf{x}) = \mathrm{cor}_{\mathcal{T}^*_1,\mathcal{T}^*_2}\Big[\mathrm{T}(\mathbf{x};\hat{\mathbf{\Theta}}(\mathcal{T}^*_1),\mathrm{T}(\mathbf{x};\hat{\mathbf{\Theta}}(\mathcal{T}^*_2)\Big]$

the it follows

$$\mathbb{V}_{\mathcal{T}}\big[\hat{\mathrm{T}}^{(B)}_{bag}(\mathbf{x};\mathcal{T})\big] = \rho_{btree}(\mathbf{x})\sigma^2_{btree}(\mathbf{x}) + \frac{1-\rho_{btree}(\mathbf{x})}{B}\sigma^2_{btree}(\mathbf{x}).$$

Generally, $\sigma^2_{btree}(\mathbf{x})$ is not overly larger than $\sigma_{tree}(\mathbf{x})$, while a typical value for $\rho_{btree}(\mathbf{x})$ is 0.05 or less. Therefore, for a large B, $\mathbb{V}_{\mathcal{T}}[\hat{\mathrm{T}}^{(B)}_{bag}(\mathbf{x};\mathcal{T})] \approx 0.05\sigma^2_{tree}(\mathbf{x})$. It is clear that the variance of the bagged estimator is less than the decision trees variance. Therefore, the predictive power of the bagged decision tree is greater than the a single decision tree.

Bagging improves on the predictive power of a single decision tree by reducing the variance. The uncorrelatedness of the different bootstrap trees reduces the variance of the model. Random forests further reduce

the variance of the bagging decision trees through a change to the RBP algorithm. For each split only $m \leq p$ variables are considered for candidate splits. The variables are randomly and independently selected for each split which means less information being shared between the trees compared to bagging and results in even less correlation between the trees.

A random forest applied to a regression problem results in the model given by

$$\hat{\mathrm{T}}_{rf}^{(B)}(\mathbf{x}; \mathcal{T}, m) = \frac{1}{B} \sum_{b=1}^{B} \mathrm{T}\big(\mathbf{x}; \tilde{\boldsymbol{\Theta}}_m(\mathcal{T}_b^*)\big).$$

with the follow properties

1. $\mathrm{bias}_{tree}(\mathbf{x}; m) = f^{**}(\mathbf{x}) - \mathbb{E}_{\mathcal{T}^*}[\mathrm{T}(\mathbf{x}; \tilde{\boldsymbol{\Theta}}_m(\mathcal{T}^*))]$
2. $\sigma^2_{tree}(\mathbf{x}; m) = \mathbb{V}_{\mathcal{T}^*}[\mathrm{T}(\mathbf{x}; \tilde{\boldsymbol{\Theta}}_m(\mathcal{T}^*)]$
3. $\rho_{tree}(\mathbf{x}; m) = \mathrm{cor}_{\mathcal{T}_1^*, \mathcal{T}_2^*}\Big[\mathrm{T}(\mathbf{x}; \tilde{\boldsymbol{\Theta}}_m(\mathcal{T}_1^*), \mathrm{T}(\mathbf{x}; \tilde{\boldsymbol{\Theta}}_m(\mathcal{T}_2^*)\Big]$

the variance of a random forest prediction $\mathbf{x}$ is given by

$$\mathbb{V}_{\mathcal{T}}\big[\hat{\mathrm{T}}_{rf}^{(B)}(\mathbf{x}; \mathcal{T}, m)\big] = \rho_{btree}(\mathbf{x}; m)\sigma^2_{btree}(\mathbf{x}; m) + \frac{1 - \rho_{btree}(\mathbf{x}; m))}{B} \cdot \sigma^2_{btree}(\mathbf{x}; m)$$

For $m < p$, the follow properties are expected

1. $\sigma^2_{btree}(\mathbf{x}; m) \geq \sigma^2_{btree}(\mathbf{x})$ as the modification to RBP in the random forest induce more randomness
2. $(\mathrm{bias}_{btree}(\mathbf{x}; m))^2 \geq (\mathrm{bias}_{btree}(\mathbf{x})$ as the random subset selected for candidate split variables results in a smaller reduction in the empirical risk compared to uses all variables for candidate splits
3. $\rho_{btree}(\mathbf{x}; m) < \rho_{btree}(\mathbf{x})$ as the randomisation is applied independently between trees

Thus, it follows that a random forest model will increase the predictive performance compared to a bagging regression tree model if the increase in variance and squared bias from the random forest modification is justified by the decrease in the correlation between bootstrapped trees in the random forest.

In a classification setting, bagging and random forests follow similar principals. There are two methods: by consensus and by probability maximising. Using the consensus method, each classification tree outputs a prediction of the class that the observations belongs to then the class that has the majority assignments from the trees is the predicted classification for the observation. The probability maximising method relies on each tree providing the probability the observation belongs to each class. The probabilities are then averaged across bootstrapped trees and the class with the highest average probability is the class assigned to the observation. Bagging and random forests possess a property that allows for a more convenient form of cross validation. Out-of-bag (OOB) sampling is used to tune hyperparameters. The OOB sample is a validation set containing all the observations not included in the bootstrap sample used to train the tree. The OOB samples are used to validate the performance of the bagged tree or random forest model.

Bagging and random forests greatly improve upon the performance of decision tree models. While the intrepretability of the decision tree is lost, the models allow for informative variable importance metrics. Variable

importance determines the variables on which splits resulted in the greatest decrease in the empirical risk. Additionally, while random forests and bagging require many hundreds more trees to be estimated resulting in far more computations, this is no longer an issue with modern parallel computing. The benefit from the improvement in predictive performance of these models outweighs the drawbacks from long computational times.

4.2 Question 2

Boosting refers to the use of improving the predictive power of a number of weak learners by creating a single ensemble of learners. Unlike bagging or random forests, the weak learners are no trained in parallel but are iteratively trained from the predictions of the existing ensemble in the previous step. The AdaBoost.M1 algorithm is one of the earliest examples of a boosting algorithm. The algorithm is suited for binary classification problems with the encoding

$$Y = \begin{cases} -1 & \text{if } G = \mathcal{G}_1 \\ 1 & \text{if } G = \mathcal{G}_2 \end{cases} \tag{4.2}$$

AdaBoost.M1 differs in a number of ways to consensus bagging where the bagged classifier is given by

$$\hat{g}_{bag}^{(B)}(\mathbf{x}; \mathcal{T}) = \text{sign}\Big(\sum_{b=1}^{B} g(\mathbf{x}; \mathcal{T}_b^*) \Big)$$

for the binary classification problem encoded in equation 4.2. The first difference is in that each classifier in the ensemble is given a weight, α_m, in order to give more importance to better classifiers. The second difference is that boosting results in classifiers being sequentially added to the ensemble depending on the classifiers already in the ensemble. This differs from the bagging and random forests where deep trees are trained in parallel in order to reduce the variance associated with each of the deep trees. Boosting improves predictive power of high bias learners, for example a shallow decision tree. The algorithm is as follows

Algorithm 6: AdaBoost.M1

1. Initialize observational weights $w_i = \frac{1}{N} : i = 1, 2, ..., N$.
2. For $m = 1, 2, ..., M$:

   set: $\hat{g}^{(m)}(.) = \hat{g}(.; \mathcal{T}, \mathbf{w})$ and compute:

   $\text{err}_m = \frac{\sum_{i=1}^{N} w_i I(y_i \neq \hat{g}^{(m)}(\mathbf{x}_i))}{\sum_{i=1}^{N} w_i}$

   set: $\alpha_m = \log(\frac{1-\text{err}_m}{\text{err}_m})$

   update: $w_i \leftarrow w_i \cdot \exp\big(\alpha_m I(y_i \neq \hat{g}^{(m)}(\mathbf{x}_i))\big)$, for $i = 1, 2, ..., N$
3. Output $\hat{g}_{boost}^{(M)}(\mathbf{x}; \mathcal{T}) = \text{sign}\Big(\sum_{m=1}^{M} \alpha_m \hat{g}^{(m)}(\mathbf{x}) \Big)$

The algorithm requires a weak learner. Initially, observations are equally weighted. The first classifier is trained normally, then for successive classifiers, the observational weights are updated in order to put greater importance on observations that are well classified by the previous classifiers in the ensemble. The weights are

increased according to a factor $\exp(\alpha_m)$ for the classifier in question. The weight α_m is given to classifier $\hat{g}^{(m)}(.)$ according to the misclassification error err_m. This results in classifiers that reduce the misclassification error most to have greater importance in the ensemble of classifiers.

For a given class of base learners, $\mathcal{B} = \{b(.;\gamma) : \gamma \in \Gamma\}$ where $b(.;\gamma) : \Re^p \to \Re$ are functions indexed by γ, the class of additive models associated with $\mathcal{B}$ is defined as

$$\mathcal{F}(\mathcal{B}) = \Big\{ f(.) = \sum_{m=1}^{M} \beta_m b(.;\gamma_m) : \beta_m \in \Re, \gamma_m \in \Gamma, m < \infty \Big\}$$

For a given M, the parameters $\{\gamma_m, \beta_m\}_{m=1}^M$ are estimated by minimizing the empirical risk

$$R(\{\gamma_m, \beta_m\}_{m=1}^M) = \sum_{i=1}^{N} \mathcal{L}\Big(y_i, \sum_{m=1}^{M} \beta_m b(\mathbf{x}_i;\gamma_m)\Big). \tag{4.3}$$

An approximation for equation 4.3 can be obtained by using forward stagewise additive modelling (FSAM). FSAM attempts to add the base learners individually rather than to jointly optimise the parameters. The FSAM algorithm is given by

Algorithm 7: Forward Stagewise Additive Modeling

1. Initialize $f^{(0)}(\mathbf{x};\mathcal{T}) = 0$.
2. For $m = 1, 2, ..., M$:
   (a) Compute:
   $\{\hat{\gamma}_m, \hat{\beta}_m\}_{m=1}^M = \underset{\beta,\gamma}{\text{argmin}} \Big\{ \sum_{i=1}^{N} \mathcal{L}\big(y_i, f^{(m-1)}(\mathbf{x}_i;\mathcal{T}) + \beta b(\mathbf{x}_i;\gamma)\big) \Big\}$
   (b) Set $\hat{f}^{(m)}(\mathbf{x};\mathcal{T}) = \hat{f}^{(m-1)}(\mathbf{x};\mathcal{T}) + \hat{\beta}_m \cdot b(\mathbf{x}_i;\hat{\gamma}_m)\big)$

An important point for FSAM is that for certain base learners, $\beta \cdot b(\mathbf{x}_i;\gamma) = b(\mathbf{x}_i;\tilde{\gamma})$ for $\tilde{\gamma} \in \Gamma$. Using this property, only $\tilde{\gamma}$ needs to be optimised.

The AdaBoost.M1 algorithm given in algorithm 6 is a special implementation of the FSAM algorithm given in algorithm 7 with the exponential loss function given by

$$\mathcal{L}\big(y, f(\mathbf{X})\big) = \exp(-Y \cdot f(\mathbf{X})).$$

While the exponential loss function can be easily expressed in the FSAM algorithm, not all loss functions can be expressed as such. When selecting a loss function, the following properties are considered

1. What is the population minimizer for the loss function?
2. How robust is the loss function to outliers?
3. Is the loss function computationally easy so work with?

The exponential loss function has the population minimize given by

$$f^{**}(\mathbf{x}) = \frac{1}{2} \log \Big(\frac{\mathcal{P}(Y = 1|\mathbf{X} = \mathbf{x})}{\mathcal{P}(Y = -1|\mathbf{X} = \mathbf{x})} \Big). \tag{4.4}$$

The fitted function $\hat{f}(.;\mathcal{T})$ obtained by minimizing the exponential loss function results in the following probabilities

$$\widehat{p}_{y_i}(\mathbf{x}_i;\mathcal{T}) = \frac{e^{\hat{f}(\mathbf{x}_i;\mathcal{T})}}{1+e^{2\hat{f}(\mathbf{x}_i;\mathcal{T})}} \cdot e^{y_i \hat{f}(\mathbf{x}_i;\mathcal{T})}$$

Here, the probability is an increasing function of $y_i(\mathbf{x}_i;\mathcal{T})$, which is called the margin, and determines the confidence of the prediction. A positive margin indicates a correct classification and a negative margin indicates a misclassification. The larger the absolute value of the margin, the greater certainty of the prediction. Thus, a more robust loss function more heavily penalises a large negative margin. The following is a list of loss functions for binary classification problems with the encoding in equation 4.2

1. Binomial Log-Likelihood: $\ell(y, f(\mathbf{x}) = \log \mathcal{P}(Y = y|\mathbf{X} = \mathbf{x})$

   $= \frac{y+1}{2} \cdot \log(p_1(\mathbf{x})) + \frac{1-y}{2} \cdot \log(1 - p_1(\mathbf{x}))$

2. Binomial Deviance Loss Function: $\mathcal{L}(y, f(\mathbf{x})) = -\ell(y, f(\mathbf{x})) =$

   $\log(1 + e^{-2 \cdot f(\mathbf{x})})$

3. Misclassification Loss Function: $\mathcal{L}(y, f(\mathbf{x})) = I(y \cdot f(\mathbf{x}) < 0)$

4. Squared-Error Loss: $\mathcal{L}(y, f(\mathbf{x})) = (y - f(\mathbf{x}))^2 = (1 - y \cdot f(\mathbf{x}))^2$

5. Hinge Loss: $\mathcal{L}(y, f(\mathbf{x})) = [1 - y \cdot f(\mathbf{x})]_+ = \max(0, 1 - y \cdot f(\mathbf{x}))$

where $p_1(\mathbf{x}) = \frac{e^{2 \cdot f(\mathbf{x})}}{1+e^{2 \cdot f(\mathbf{x})}}$. The population minimizer for the binomial deviance is also given by equation 4.4. The population minimizer for the squared-error loss and Hinge loss functions is given by

$$f^{**}(\mathbf{x}) = 2 \cdot \mathcal{P}(Y = 1|\mathbf{X} = \mathbf{x}) - 1$$

The exponential loss, binomial deviance and Hinge loss are all monotone decreasing functions of the margin, and thus preferred to the misclassification and squared-error loss functions. Furthermore, the exponential loss function increases exponentially in the negative margin and is thus less robost to mislabelled observations and outliers compared to the binomial deviance loss function that increases linearly in the negative margin. Overall, the binomial deviance is the most preferred loss function for binary classification.

For regression problems, the two most common loss functions and their population minimizers are

1. Squared-error loss: $\mathcal{L}(y, f(\mathbf{x})) = (y - f(\mathbf{x}))^2$

   with $f^{**}(\mathbf{x}) = \mathbb{E}[Y|\mathbf{X} = \mathbf{x}]$

2. Absolute loss: $\mathcal{L}(y, f(\mathbf{x})) = |y - f(\mathbf{x})|$

   with $f^{**}(\mathbf{x}) = \text{median}[Y|\mathbf{X} = \mathbf{x}]$

Additionally, the Huber loss function which forms a compromise between the squared-error loss and the absolute loss is given by

$$\mathcal{L}(y, f(\mathbf{x})) = \begin{cases} (y - f(\mathbf{x}))^2 : & \text{for } |y - f(\mathbf{x})| < \delta \\ 2\delta|y - f(\mathbf{x})| - \delta^2 : & \text{otherwise} \end{cases}$$

For regression problems, absolute loss and Huber loss offer more robust loss functions since the increase linearly for large residuals. However, they are not as computationally friendly.

When implementing FSAM with different loss functions, the solutions may not be as elegant as the AdaBoost.M1 algorithm or similar algorithms using squared-error loss. Additionally, the learners used in the boosting algorithms may similarly not be easily computed for arbitrary loss functions. While the binomial deviance and absolute loss functions may be preferred, the exponential loss for classification and squared-error loss for regression have easily implemented tree building algorithms. These algorithms are also more efficient.

Gradient boosting was developed to circumvent the issues that arise from attempting boosting with complicated loss functions. Gradient boosting solves the same problem as FSAM given in equation 4.3. However, the empirical risk is reformulated as follows

$$R(\widehat{\mathbf{f}}) = \sum_{i=1}^{N} \mathcal{L}(y, \hat{f}_i)$$

where the predictions are directly compared to the response using the loss function rather than through a model. From this formulation, the gradient of the empirical risk with regard to the prediction is given as

$$(\nabla_{f_i} R)(\widehat{\mathbf{f}}) = (\nabla_f \mathcal{L})(y_i, \hat{f}_i) : i = 1, 2, ..., N.$$

The prediction given by $\hat{f}_{add}^{(m-1)}(.;\mathcal{T})$, a model fitted to an arbitrary loss function using FSAM, is given by $\widehat{\mathbf{f}}^{(m-1)} = \big(\hat{f}^{(m-1)}(\mathbf{x}_1;\mathcal{T}), ..., \hat{f}^{(m-1)}(\mathbf{x}_N;\mathcal{T})\big)^T$. Following this formulation, equation 4.3 can be rewritten as

$$\begin{aligned} R^{(m)}(\gamma, \beta) &= \sum_{i=1}^{N} \mathcal{L}\Big(y_i, \hat{f}^{(m-1)}(\mathbf{x}_i;\mathcal{T}) + \beta \cdot b(\mathbf{x}_i;\gamma)\Big) \\ &= \sum_{i=1}^{N} \mathcal{L}\Big(y_i, \hat{f}_i^{(m-1)} + \beta \cdot b(\mathbf{x}_i;\gamma)\Big) \\ &= \mathrm{R}\Big(\widehat{\mathbf{f}}^{(m-1} + \beta \cdot \mathbf{b}(\gamma)\Big). \end{aligned}$$

where $\mathbf{b}(\gamma) = \big(b(\mathbf{x}_1;\gamma), ..., b(\mathbf{x}_N;\gamma)\big)^T$. A quadratic approximation of R(.) are the point $\widehat{\mathbf{f}}^{(m-1}$ is given by

$$\begin{aligned} R^{(m)}(\gamma, \beta) &= \mathrm{R}\Big(\widehat{\mathbf{f}}^{(m-1} + \beta \cdot \mathbf{b}(\gamma)\Big) \\ &\approx \mathrm{R}(\widehat{\mathbf{f}}^{(m-1)}) + \beta \cdot \mathbf{b}^T(\gamma) \cdot (\nabla_{\mathbf{f}} \mathrm{R})(\widehat{\mathbf{f}}^{(m-1)}) + \frac{\beta^2}{2t} \cdot ||\mathbf{b}(\gamma)||_2^2 \\ &= \frac{1}{2} t \Big(||\frac{\beta}{t} \cdot \mathbf{b}(\gamma) - \mathbf{r}^{(m)}||_2^2 - ||\mathbf{r}^{(m-1)}||_2^2\Big) - \mathrm{R}(\widehat{\mathbf{f}}^{(m-1)}), \end{aligned}$$

where $\mathbf{r}^{(m)} = -(\nabla_{\mathbf{f}} \mathrm{R})(\widehat{\mathbf{f}}^{(m-1)})$. This approximation is the objective in the gradient boosting minimization problem. First

$$\hat{\gamma}_m = \underset{\gamma}{\operatorname{argmin}} \Big\{ ||\mathbf{b}(\gamma) - \mathbf{r}^{(m)}||_2^2 \Big\}$$

is estimated before adding $t \cdot b(.;\hat{\gamma})$ to the $m-1$ base learners since the base learner is scale invariant as mentioned above. Since $||\mathbf{b}(\gamma) - \mathbf{r}^{(m)}||_2^2 = \sum_{i=1}^{N} \big(r_i^{(m)} - b(\mathbf{x}_i;\gamma)\big)^2$, $\hat{\gamma}_m$ is estimated by fitting the a base learner using squared-error loss to the negative gradient in the prediction space. The negative gradients are analogous to the residual in regression. The step size for the update can be obtained by exact line search using the original loss function given by

$$\hat{\beta}_m = \underset{t}{\operatorname{argmin}} \Big\{ \sum_{i=1}^{N} \mathcal{L}(y_i, \hat{f}^{(m-1)}(\mathbf{x}_i;\mathcal{T}) + t \cdot b(\mathbf{x}_i;\hat{\gamma}_m) \Big\}.$$

The update is then given by

$$\hat{f}_{add}^{(m)}(\mathbf{x};\mathcal{T}) = \hat{f}_{add}^{(m-1)}(\mathbf{x};\mathcal{T}) + \hat{\beta}_m \cdot b(\mathbf{x};\hat{\gamma}_m).$$

The gradient boosting algorithm is as follows

Algorithm 8: Gradient Boosting

---

1. Initialize $\hat{f}_{add}^{(0)}(\mathbf{x};\mathcal{T}) = \underset{c}{\text{argmin}}\{\sum_{i=1}^{N} \mathcal{L}(y_i, c)\}$.
2. For $m = 1, 2, ..., M$:
   (a) Compute $r_i^{(m)} = -(\nabla_f \mathcal{L})(y_i, \hat{f}_i^{(m-1)})$
   for $i = 1, 2, ..., N$
   (b) Find $\hat{\gamma}_m = \underset{\gamma}{\text{argmin}}\Big\{ \sum_{i=1}^{N} \big(r_i^{(m)} - b(\mathbf{x}_i;\gamma)\big)^2 \Big\}$ and
   $\hat{\beta}_m = \underset{t}{\text{argmin}}\Big\{ \sum_{i=1}^{N} \mathcal{L}(y_i, \hat{f}^{(m-1)}(\mathbf{x}_i;\mathcal{T}) + t \cdot b(\mathbf{x}_i;\hat{\gamma}_m) \Big\}$
   (c) Update: $\hat{f}_{add}^{(m)}(\mathbf{x};\mathcal{T}) = \hat{f}_{add}^{(m-1)}(\mathbf{x};\mathcal{T}) + \hat{\beta}_m \cdot b(\mathbf{x};\hat{\gamma}_m)$
3. Output: $\hat{f}_{add}^{(M)}(\mathbf{x};\mathcal{T})$

From the above, it is clear that gradient boosting allows for an efficient framework to fit complicated loss functions using FSAM. By using squared-error loss to fit the negative gradients of the prediction space, the complications that arise from other loss functions are avoided.

Gradient tree boosting makes use of shallow decision trees as the base learner. This is due to the high bias of shallow decision trees. Gradient tree boosting follows similarly to algorithm 8 with minor differences. Gradient tree boosting makes use of squared-error loss as a surrogate loss function to fit regression trees to the negative gradients. The tree topology determined by the regression tree is taken but the constants in the terminal nodes are estimated using the original loss function. The step size is thus unnecessary. The gradient tree boosting algorithm is given in algorithm 9.

For gradient tree boosting fit using the absolute loss function, Step 2a in algorithm 9 becomes

$$r_i^{(m)} = \text{sign}(y_i - \hat{t}_i^{(m-1)})$$

and the constants in the terminal nodes in Step 2c become

$$\hat{c}^{(m)}(\mathfrak{R}) = \text{median}(\{y_i - \hat{t}_i^{(m-1)} : \mathbf{x}_i \in \mathfrak{R}\}$$

for all $\mathfrak{R} \in \text{Ter}\big(\mathfrak{T}_{\mathcal{T}}^{(m)}\big)$, where $\hat{t}_i^{(m-1)} = \widehat{\text{T}}_{add}^{(m-1)}(\mathbf{x}_i;\mathcal{T})$. For binary classification problems, gradient tree boosting fit using binomial deviance loss function, Step 2a in algorithm 9 becomes

$$r_i^{(m)} = \frac{2y_i}{1 + e^{2 \cdot y_i \cdot \hat{t}_i^{(m-1)}}}$$

and Step 2c becomes

$$\hat{c}^{(m)}(\mathfrak{R}) = \underset{c}{\text{argmin}}\Big\{ \sum_{\mathbf{x}_i \in \mathfrak{R}} \log\Big(1 + e^{-2y_i \cdot c} \cdot e^{-2 \cdot y_i \cdot \hat{t}_i^{(m-1)}}\Big) \Big\}$$

for all $\mathfrak{R} \in \text{Ter}\big(\mathfrak{T}_{\mathcal{T}}^{(m)}\big)$.

Algorithm 9: Gradient Tree Boosting

1. Initialize $\widehat{\mathrm{T}}_{add}^{(0)}(\mathbf{x};\mathcal{T}) = \underset{c}{\operatorname{argmin}}\{\sum_{i=1}^{N}\mathcal{L}(y_i,c)\}$.
2. For $m = 1,2,...,M$:
   (a) Compute $r_i^{(m)} = -(\nabla_f\mathcal{L})\Big(y_i, \widehat{\mathrm{T}}_{add}^{(m-1)}(\mathbf{x}_i;\mathcal{T})\Big)$ for $i = 1,2,...,N$
   (b) Find the tree topology $\mathfrak{T}_{\mathcal{T}}^{(m)}$ by fitting a regression tree to $r_i^{(m)}$ for $i = 1,2,...,N$
   (c) Calculate $\widehat{c}^{(m)}(\mathfrak{R}) = \underset{c}{\operatorname{argmin}}\Big\{\sum_{\mathbf{x}_i\in\mathfrak{R}}\mathcal{L}\big(y_i, \widehat{\mathrm{T}}_{add}^{(m-1)}(\mathbf{x}_i;\mathcal{T}) + c\big)\Big\}$ for all $\mathfrak{R} \in \mathrm{Ter}\big(\mathfrak{T}_{\mathcal{T}}^{(m)}\big)$
   (d) Update $\widehat{\mathrm{T}}_{add}^{(m)}(\mathbf{x}_i;\mathcal{T}) = \widehat{\mathrm{T}}_{add}^{(m-1)}(\mathbf{x}_i;\mathcal{T}) + \mathrm{T}(\mathbf{x}_i;\widehat{\boldsymbol{\Theta}}_{\mathcal{T}}^{(m)})$, where $\widehat{\boldsymbol{\Theta}}_{\mathcal{T}}^{(m)}) = \{\widehat{\mathfrak{T}}_{\mathcal{T}}^{(m)}, \widehat{c}(.;m)\}$.
3. Output $\widehat{\mathrm{T}}_{add}^{(M)}(\mathbf{x}_i;\mathcal{T})$

4.3 Question 3

Modern gradient boosting machines (GBM), including eXtreme Gradient Boosting (XGBoost), light Gradient Boosting Machine (lightGBM) and Categorical Boosting (CatBoost), make a number of changes to historical gradient tree boosting algorithms to increase efficiencies that allow for models to be scaled up to larger datasets. One of these innovations is histogram node splitting. Histogram node splitting focuses on decreasing the computational intensity of determining split points during tree building. Rather than considering all possible split points when fitting the negative gradient in the terminal nodes as is done in historical gradient tree boosting, histogram node splitting allows for only a subset of the split points to be considered. This allows for a decrease in computational costs and acts as a form of regularization.

The splits considered are $\{s_{jb} : b = 1,2,...,B\}$. The proposal splits can be determined globally before the tree building process, or locally before each split. Observations are binned based on their values for feature j. For a region $\mathfrak{R}$, the bins for feature j are given by

$$\mathrm{bin}_j[b] = \{\mathbf{x}_i \in \mathfrak{R} : s_{j,b-1} < x_{ij} \leq s_{j,b}\}$$

where $s_{j,0} = -\infty$. The height of the histogram is given by aggreagting the negative gradients in the bin

$$\mathrm{r}_j[b] = \sum_{\mathbf{x}_i\in\mathrm{bin}_j[b]} r_i^{(m)}.$$

There are a number of ways of selecting the proposal splitting points for the bins. The simplest approach is to select the proposal splitting points such that the number of observations per bin is approximately equal. This can be achieved by setting the proposal splitting points to

$$s_{jb} = \mathrm{quan}_j\Big(\frac{b}{B};\mathfrak{R}\Big)$$

for $b = 1,2,...,B$ with $\mathrm{quan}_j(p;\mathfrak{R})$ the quantile at level p of feature j in region $\mathfrak{R}$. Modern GBMs make use of observational weights that indicate the importance of observations in the regions to influence the selection of

the proposal splits. In the XGBoost framework, a rank function is defined for feature j given by

$$\text{rank}_j(z;\mathfrak{R}) = \frac{\sum_{\mathbf{x}_i \in \mathfrak{R}} w_i \cdot I(x_{ij} < z)}{\sum_{\mathbf{x}_i \in \mathfrak{R}} w_i}$$

where w_i is the weight of the observation i. The objective is to finding split points $s_{jb} : b = 1, 2, ..., B$ for which

$$\left|\text{rank}_j(s_{jb};\mathfrak{R}) - \text{rank}_j(s_{j,b+1};\mathfrak{R})\right| < \varepsilon$$

for $b = 1, 2, ..., B$ where $s_1 = \min_i\{x_{ij} : \mathbf{x}_i \in \mathfrak{R}\}$ and $s_B = \max_i\{x_{ij} : \mathbf{x}_i \in \mathfrak{R}\}$. The ε above is treated as a hyperparameter that must be specified. Thus the sum of the weights in the bins does not exceed ε, and $B \approx \frac{1}{\varepsilon}$. The bins are balanced by weights and not observations. In XGBoost, the following algorithm is used to approximate the solution to finding the proposal splitting points

Algorithm 10: Histogram Node Splitting

Consider a terminal node with region $\mathfrak{R}$ and proposal splitting points $\{s_{jb} : \forall j, b\}$

1. Initialize: $r_{tot} = \sum_{\mathbf{x}_i \in \mathfrak{R}} r_i^{(m)}$, $n = \sum_{\mathbf{x}_i \in \mathfrak{R}} \text{I}(\mathbf{x}_i \in \mathfrak{R})$, $score_{max} = 0$
2. For $j = 1, 2, ..., p$:
   (a) build histogram for feature j
   (b) Set $j^- = 0, n^- = 0$
   (c) for $b = 1, 2, ..., B$:
      i. Update: $r^- = r^- + \text{r}_j[b], n^- = n^- + \text{n}_j[b]$.
      ii. Compute: $score = \frac{(r^-)^2}{n^-} + \frac{(r_{tot} - r^-)^2}{(n - n^-)}$
      iii. if $score > score_{max}$ then:
      $j(\mathfrak{R}) = j, s(\mathfrak{R}) = s_{jb}; score_{max} = score$
3. Output: $j(\mathfrak{R}), s(\mathfrak{R})$

Droupout meets additive regression trees (DART) is an implementation of the droupout principle, often seen in neural networks, to gradient tree boosting. DART is implemented to combat over-specialisation in decision trees. Over-specialization occurs when decision trees, added later to the ensemble, influence the predictions of only a few observations. This can result in two problems

1. Few initially added trees have an exaggerated influence on the predictions of the model.
2. The later trees that only focus on a few observations can be seen as over-fitting and negatively impact the generalization error.

Shrinkage methods are generally used to combat over-specialization. While shrinkage methods overcome the first problem, they do no address the second problem.

In DART, the negative gradients used to estimate new decision trees are calculated using only a subset of the decision trees in the ensemble. An additive tree model, estimated through gradient tree boosting, is given

by

$$\widehat{\mathrm{T}}_{add}^{(m)}(.;\mathcal{T}) = \sum_{l=1}^{m-1} \mathrm{T}(.;\widehat{\boldsymbol{\Theta}}^{(l)}).$$

If $S \subset \{1, 2, ..., m-1\}$ contains the indices of the selected trees, the regression tree is fitted to the following gradient

$$r_i^{(m)} = (\nabla_f \mathcal{L})\big(y_i, \sum_{l \in S} \mathrm{T}(\mathbf{x}_i;\widehat{\boldsymbol{\Theta}}^{(l)})\big) \text{ for } i = 1, 2, ..., N$$

using squared-error loss. This tree is denoted $\mathrm{T}(\mathbf{x}_i;\widehat{\boldsymbol{\Theta}}^{(m)})$. It follows that both $\mathrm{T}(\mathbf{x}_i;\widehat{\boldsymbol{\Theta}}^{(m)})$ and $\sum_{l \notin S} \mathrm{T}(\mathbf{x}_i;\widehat{\boldsymbol{\Theta}}^{(l)})$ are estimated for $r_i^{(m)}$. Thus to avoid overestimating $r_i^{(m)}$ with $\mathrm{T}(\mathbf{x}_i;\widehat{\boldsymbol{\Theta}}^{(m)}) + \sum_{l \notin S} \mathrm{T}(\mathbf{x}_i;\widehat{\boldsymbol{\Theta}}^{(l)})$, the prediction is rescaled the following manner

$$\frac{1}{1+|S^c|}\mathrm{T}(\mathbf{x}_i;\widehat{\boldsymbol{\Theta}}^{(m)}) + \frac{|S^c|}{1+|S^c|}\sum_{l \notin S} \mathrm{T}(\mathbf{x}_i;\widehat{\boldsymbol{\Theta}}^{(l)})$$

A larger weighting in the rescaling is given to $\sum_{l \notin S} \mathrm{T}(\mathbf{x}_i;\widehat{\boldsymbol{\Theta}}^{(l)})$ as it is seen as a more reliable prediction for $r_i^{(m)}$ as it contains more trees. The rescaling of the individual trees is given by

$$\mathrm{T}(.;\widehat{\boldsymbol{\Theta}}^{(m)}) \leftarrow \frac{1}{1+|S^c|}\mathrm{T}(.;\widehat{\boldsymbol{\Theta}}^{(m)})$$
$$\mathrm{T}(.;\widehat{\boldsymbol{\Theta}}^{(l)}) \leftarrow \frac{|S^c|}{1+|S^c|}\mathrm{T}(.;\widehat{\boldsymbol{\Theta}}^{(l)}) : l \notin S$$

with the update given by

$$\widehat{\mathrm{T}}_{add}^{(m)}(.;\mathcal{T}) = \sum_{l \in S} \mathrm{T}(.;\widehat{\boldsymbol{\Theta}}^{(l)}) + \sum_{l \notin S} \mathrm{T}(.;\widehat{\boldsymbol{\Theta}}^{(l)}) + \mathrm{T}(.;\widehat{\boldsymbol{\Theta}}^{(m)})$$

As seen in figure 4.1, the trees added later to the ensemble retain their high predictive performance. Thus an aspect of the over-specification associated with gradient tree boosting is overcome.

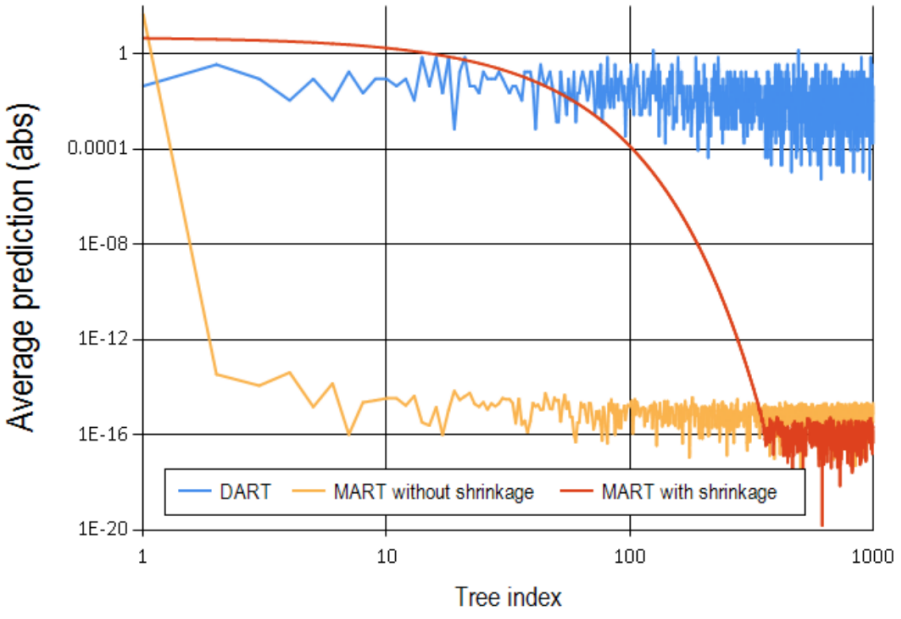

Figure 4.1: Effect of dropout on the predictions made by decision trees in the ensemble (Rashmi & Gilad-Bachrach 2015)

LightGBM is the second in a wave of modern gradient tree boosting algorithms. The model follows on from the XGBoost model and primarily focuses on increasing the speed of computation while retaining the predictive performance of other gradient boosting machines. The improved speed of computation allows for scaled up models trained on larger datasets with many more features. The two primary novelties of LightGBM are called Gradient One-Side Sampling (GOSS) and Exclusive Feature Bundling (EFB). LightGBM makes use of RBP to build new trees that minimize

$$\sum_{i=1}^{N}(r_i^{(m)} - \mathrm{T}\big(\mathbf{x}_i;\boldsymbol{\Theta}\big)\big)^2. \tag{4.5}$$

The additive model is then updated by

$$\widehat{\mathrm{T}}_{add}^{(m)}(\mathbf{x};\mathcal{T}) = \widehat{\mathrm{T}}_{add}^{(m-1)}(\mathbf{x};\mathcal{T}) + \eta\cdot\mathrm{T}\big(\mathbf{x};\widehat{\boldsymbol{\Theta}}^{(m)}\big)$$

with $\widehat{\boldsymbol{\Theta}}^{(m)}$ estimated via RBP in equation 4.5. This differs from histroical gradient tree boosting as the constants in the terminal regions are not estimated via a loss function once the tree topology is estimated. The RBP in 4.5, attempts to maximize

$$\frac{\big(r_{js}^{-}(\mathfrak{R})\big)^2}{n_{js}^{-}(\mathfrak{R})} + \frac{\big(r_{js}^{+}(\mathfrak{R})\big)^2}{n_{js}^{+}(\mathfrak{R})} \tag{4.6}$$

where

$$r_{js}^{-}(\mathfrak{R}) = \sum_{\mathbf{x}_i\in\mathfrak{R}} r_i^{(m)}\mathrm{I}(x_{ij}\le s), n_{js}^{-}(\mathfrak{R}) = \sum_{\mathbf{x}_i\in\mathfrak{R}}\mathrm{I}(x_{ij}\le s)$$

$r_{js}^{+}(\mathfrak{R}), n_{js}^{+}(\mathfrak{R})$ follow similarly.

The first novelty of LightGBM is the implementation of GOSS. GOSS is an attempt to provide weights for observations during row subsampling. While the AdaBoost.M1 algorithm has native weights that increase the probability of an observation being selected, gradient tree boosting lacks natural weights. GOSS is based on the principle that observations with large negative gradients, $\big|r_i^{(m)}\big|$, are not well predicted by the current additive model and thus should have a higher probability of being selected during row subsampling. GOSS begins by ordering the observations from largest $\big|r_i^{(m)}\big|$ to smallest and then assigns the top $N\times a\times 100\%$ to the set $\mathcal{A}$. Then a subset $\mathcal{B}$ of size $N\times b\times 100\%$ is randomly selected from the remaining observations. The total subsample is then $\mathcal{A}\cup\mathcal{B}$. Using the subsample, equation 4.5 can be reformulated using weights as

$$\sum_{i=1}^{N} w_i(r_i^{(m)} - \mathrm{T}\big(\mathbf{x}_i;\boldsymbol{\Theta}\big)\big)^2$$

where

$$w_i = \begin{cases} 1 & : \mathbf{x}_i\in\mathcal{A} \\ \frac{1-a}{b} & : \mathbf{x}_i\in\mathcal{B} \\ 0 & : \mathbf{x}_i\notin\mathcal{A},\mathcal{B} \end{cases}$$

From which, the splitting criterion given in equation 4.6 can be reformulated as maximising

$$\frac{\big(r_{js}^{-}(\mathcal{A}\cap\mathfrak{R}) + \frac{1-a}{b}r_{js}^{-}(\mathcal{B}\cap\mathfrak{R})\big)^2}{n_{js}^{-}(\mathfrak{R})} + \frac{\big(r_{js}^{+}(\mathcal{A}\cap\mathfrak{R}) + \frac{1-a}{b}r_{js}^{+}(\mathcal{B}\cap\mathfrak{R})\big)^2}{n_{js}^{+}(\mathfrak{R})}$$

The second novelty of LightGBM is the implementation of EFB. EFB is an attempt to reduce the sparsity in features containing many zeros. Large datasets often contain many features that are sparse. However, sparse

features are not as informative as dense continuous features for proposal splits during tree building. EFB overcomes this by bundling together mutually exclusive sparse features into a few more dense features without the loss of information. EFB's objective is to condense the features into as few bundles as possible to maximise training speed. Ke et al. (2017) explains that the problem of determining the minimum number of bundles is equivalent to a graph colouring problem. Conflicts between two sparse features occur when both features as non-zero. Since this problem is NP-hard, LightGBM implements a greedy algorithm to approximate the process.

The algorithm sorts the sparse features by their degree defined as the number of conflicts with other features. It then loops through the sparse features in order, adding the feature to a bundle if the number of conflicts is at most $\gamma \cdot N$, otherwise it creates a new bundle. This algorithm becomes costly as the number of sparse features becomes large. LightGBM proposes an alternative where the features are ranked according to the number of observations with non-zero values for the feature. In order to condense a bundle into a new feature, the method of merging two sparse features $\mathbf{x}_s$ and $\mathbf{x}_t$ into a new feature $\mathbf{z}$ is given by

$$\mathbf{z} = \begin{cases} 0 & : x_{is} = x_{it} = 0 \\ x_{is} & : x_{is} \neq 0 \\ x_{it} + a_s & x_{it} \neq 0 \end{cases}$$

where $a_s = \max_i\{x_{is}\}$. The novelties of LightGBM are best used when training in parallel using GPUs. LightGBM makes use of a modification to tradition data parallelization by first segmenting the data, then distributing it between machines where the machines determine split points from the histogram binning process. The machines then cast votes for the best split points with the points that receive the most votes being the designated split points.

Assignment 5

5.1 Question 1

To understand reproducing kernel Hilbert spaces (RKHS) and their usage in statistical learning, an understanding of vector, Banach and Hilbert spaces is required. A vector space $\mathcal{V}$ generally is a set of mathematical objects that posses the following additive properties for elements $\mathbf{u}, \mathbf{v}, \mathbf{w} \in \mathcal{V}$:

- $\mathbf{u} + (\mathbf{v} + \mathbf{w}) = (\mathbf{u} + \mathbf{v}) + \mathbf{w}$ (associative law)
- $\mathbf{u} + \mathbf{v} = \mathbf{v} + \mathbf{u}$ (commutative law)
- $\mathbf{u} + (-\mathbf{u}) = (-\mathbf{u}) + \mathbf{u} = \mathbf{0}$ (inverse law)
- $\mathbf{u} + \mathbf{0} = \mathbf{0} + \mathbf{u} = \mathbf{u}$ (identity law)

and the following multiplicative properties for elements $\mathbf{u}, \mathbf{v} \in \mathcal{V}$ and $k, l \in \mathfrak{R}$:

- $k \times (l \times \mathbf{u}) = (kl) \times \mathbf{u}$ (associative law)
- $k \times (\mathbf{u} + \mathbf{v}) = (k \times \mathbf{u}) + (k \times \mathbf{v})$ (distributive law over vector sum)
- $(k + l) \times \mathbf{u} = (k \times \mathbf{u}) + (l \times \mathbf{u})$ (distributive law over scalar sum)
- $1 \times \mathbf{u} = \mathbf{u}$ (identity law)

A norm $||.|| : \mathcal{V} \to \mathfrak{R}$ satisfies the following properties:

- $||\mathbf{v}|| \geq 0$
- $||\mathbf{v}|| = 0$ if $\mathbf{v} = \mathbf{0}$
- $||c\mathbf{v}|| = |c|||\mathbf{v}||$ for $c \in \mathfrak{R}$
- $||\mathbf{v} + \mathbf{v}|| \leq ||\mathbf{u}|| + ||\mathbf{v}||$.

A sequence $\{\mathbf{v}_\mathrm{n}\} \subset \mathcal{V}$ is called a Cauchy sequence if for $\varepsilon > 0$ there is an integer N such that $||\mathbf{v}_\mathrm{n} - \mathbf{v}_\mathrm{m}|| < \varepsilon$ where n,m $> N$. Following this, a vector space $\mathcal{V}$ is called a Banach space, if a norm is defined on this space, and all Cauchy sequences in $\mathcal{V}$ converge to elements of $\mathcal{V}$.

An inner product $\langle ., . \rangle : \mathcal{V} \times \mathcal{V} \to \mathfrak{R}$ of a vector space $\mathcal{V}$ satisfies the following:

1. $\langle c\mathbf{u}, \mathbf{v}\rangle = c\langle \mathbf{u}, \mathbf{v}\rangle$
2. $\langle \mathbf{u}, \mathbf{v}\rangle = \langle \mathbf{v}, \mathbf{u}\rangle$
3. $\langle \mathbf{u}, \mathbf{v} + \mathbf{w}\rangle = \langle \mathbf{u}, \mathbf{v}\rangle + \langle \mathbf{u}, \mathbf{w}\rangle$
4. $\langle \mathbf{u}, \mathbf{u} \geq 0\rangle$ with equality holding if and only if $\mathbf{u} = \mathbf{0}$

For a vector space $\mathcal{V}$, the completion of $\mathcal{V}$ is denoted by $\overline{\mathcal{V}}$, and obtained by adding the limits of all Cauchy sequences in $\mathcal{V}$ to $\mathcal{V}$. Furthermore, let $\mathcal{W} \subseteq \mathcal{V}$, for each $\mathbf{u} \in \text{span}(\mathcal{W})$ there exists an integer $1 \leq M < \infty$ and corresponding vectors $\mathbf{w}_1, \mathbf{w}_2, ..., \mathbf{w}_M \in \mathcal{W}$ such that $\mathbf{u} = \sum_{i=1}^{M} \alpha_i \mathbf{w}_i$. A Banach space where the norm is defined by an inner-product (for example: $||\mathbf{u}|| = \sqrt{\langle \mathbf{u}, \mathbf{u}\rangle}$) is a called a Hilbert space.

Consider the space of continuous functions on the compact set $[0, 1]$, denoted $\mathcal{V} = \mathcal{C}([0, 1])$. For any arbitrary functions $f(x)$ and $g(x)$ the inner product is defined as:

$$\langle f, g\rangle = \int_0^1 f(x)g(x)dx.$$

For this to be a valid inner product it must satisfy the above conditions:

1. $$\langle cf, g\rangle = \int_0^1 cf(x)g(x)dx = c\int_0^1 cf(x)g(x)dx = c\langle f, g\rangle \text{ where } c \in \Re$$
2. $$\langle f, g\rangle = \int_0^1 f(x)g(x)dx = \int_0^1 g(x)f(x)dx = \langle g, f\rangle$$
3. $$\begin{aligned}\langle f, (g + h)\rangle &= \int_0^1 f(x)(g(x) + h(x))dx = \int_0^1 (f(x)g(x)) + (f(x)h(x))dx \\ &= \int_0^1 f(x)g(x)dx + \int_0^1 f(x)h(x)dx = \langle f, g\rangle + \langle f, h\rangle\end{aligned}$$
4. $$\langle f, f\rangle = \int_0^1 f(x)f(x)dx = \int_0^1 (f(x))^2 dx \geq 0 \text{ (since } f : \Re \to \Re)$$

A positive definite kernel function is a function $K : \mathcal{X} \times \mathcal{X} \to \Re$ over $\mathcal{X} \times \mathcal{X}$ for $\mathcal{X}$ which is a compact subset of $\Re^p$ if it satisfies:

- $K(\mathbf{x}, \mathbf{y}) = K(\mathbf{y}, \mathbf{x})$ for all $\mathbf{x}, \mathbf{y} \in \mathcal{X}$
- $\int_{\mathcal{X}} \int_{\mathcal{X}} f(\mathbf{x})K(\mathbf{x}, \mathbf{y})f(\mathbf{y})d\mathbf{x}d\mathbf{y} \geq 0$ for all $f \in L_2(\mathcal{X})$

Positive definite kernels display the following property: for every integer $M \geq 1$ and any $\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_M \in \mathcal{X}$

$$\sum_{i=1}^{M}\sum_{j=1}^{M} \alpha_i K(\mathbf{x}_i, \mathbf{x}_j)\alpha_j \geq 0$$

which for continuous kernels, $K(\cdot,\cdot)$, is equivalent to the above property. For a compact set $\mathcal{X} \subseteq \Re^p$, positive definite kernels have an eigen-expansion

$$K(\mathbf{x},\mathbf{y}) = \sum_{j=1}^{\infty} \gamma_j \phi_j(\mathbf{x})\phi_j(\mathbf{y}) \tag{5.1}$$

where the function $\phi(\mathbf{x}) : j = 1, 2, ...$ are orthonormal in $L_2(\mathcal{X})$, $\gamma_1 \geq \gamma_2 \geq ... > 0$, and $\sum_{i=1}^{\infty} \gamma_i^2 < \infty$. A number of important kernels include:

- Linear Kernel

$$K(\mathbf{x},\mathbf{y}) = \mathbf{x}\mathbf{y}'$$

- Polynomial Kernel

$$K(\mathbf{x},\mathbf{y}) = (\langle \mathbf{x},\mathbf{y}\rangle + 1)^d$$

- Gaussian Kernel

$$K(\mathbf{x},\mathbf{y}) = \exp\left(-\nu||\mathbf{x}-\mathbf{y}||_2^2\right)$$

- Hyperbolic Tangent Kernel

$$K(\mathbf{x},\mathbf{y}) = \tanh(\kappa_1\langle \mathbf{x},\mathbf{y}\rangle + \kappa_2)$$

A reproducing Kernel Hilbert space $\mathcal{H}$ is a Hilbert space of functions defined on a compact set $\mathcal{X} \subseteq \Re^p$ such that the evaluation functionals are linear, bounded functionals. Let $\mathcal{X} \subseteq \Re^p$, for a Hilbert space $\mathcal{H}$ of functions $f : \mathcal{X} \to \Re$, the evaluation functional $\delta_{\mathbf{x}} : \mathcal{H} \to \Re$ is defined as $\delta_{\mathbf{x}}(f) = f(\mathbf{x})$. Evaluation functionals are linear if $\delta_{\mathbf{x}}(\alpha f + \beta g) = \alpha\delta_{\mathbf{x}}(f) + \beta\delta_{\mathbf{x}}(g)$ for all $\alpha, \beta \in \Re, \mathbf{x} \in \mathcal{X}$ and $f, g \in \mathcal{H}$. Evaluation functionals are bounded if, for all $\mathbf{x} \in \mathcal{X}$ there is a constant $U_{\mathbf{x}} \geq 0$ such that, for all $f \in \mathcal{H}$, $|\delta_{\mathbf{x}}(f)| \leq U_{\mathbf{x}}||f||_{\mathcal{H}}$. Furthermore, the boundedness of evaluation functionals results in for all $f, g \in \mathcal{H}, |f(\mathbf{x}) - g(\mathbf{x})| \leq U_{\mathbf{x}}||f - g||_{\mathcal{H}}$. This acts as a continuity constraint on the functions in the RKHS.

For every RKHS, there is a positive definite kernel with the following properties:

1. Reproducing property

$$f(\mathbf{x}) = \langle f(.), K(.,\mathbf{x})\rangle_{\mathcal{H}} \text{ for all } f \in \mathcal{H}$$

2. Let $\mathcal{H}_0 = \text{span}(\{K(.,\mathbf{x}) : \mathbf{x} \in \mathcal{X}\})$, then $\mathcal{H}$ is the completion of $\mathcal{H}_0$.

Property (2) implies that $K(.,\mathbf{x}) \in \mathcal{H}$ for all $\mathbf{x} \in \mathcal{X}$. Property (1) implies that, for all $\mathbf{x}, \mathbf{y} \in \mathcal{X}$, $K(\mathbf{x},\mathbf{y}) = \langle K(.,\mathbf{x}), K(.,\mathbf{y})\rangle_{\mathcal{H}}$. For $f, g \in \mathcal{H}_0$ such that $f(\mathbf{x}) = \sum_{i=1}^{M_1} \alpha_i K(\mathbf{x},\mathbf{x}_i)$ and $g(\mathbf{x}) = \sum_{j=1}^{M_2} \beta_j K(\mathbf{x},\mathbf{x}_j)$, the following properties exist

1.

$$\begin{aligned}
\langle f(.), g(.)\rangle &= \Big\langle \sum_{i=1}^{M_1} \alpha_i K(.,\mathbf{x}_i), \sum_{j=1}^{M_2} \beta_j K(.,\mathbf{y}_j)\Big\rangle \\
&= \sum_{i=1}^{M_1}\sum_{j=1}^{M_2} \alpha_i\beta_j \langle K(.,\mathbf{x}_i), K(.,\mathbf{y}_j)\rangle \\
&= \sum_{i=1}^{M_1}\sum_{j=1}^{M_2} \alpha_i\beta_j K(\mathbf{x_i},\mathbf{y_j})
\end{aligned}$$

2.

$$\begin{aligned} f(\mathbf{x}) &= \sum_{i=1}^{M_1} \alpha_i K(., \mathbf{x}_i) \\ &= \sum_{i=1}^{M_1} \sum_{j=1}^{\infty} \alpha_i \gamma_j \phi_j(\mathbf{x}) \phi_j(\mathbf{x}_i) \\ &= \sum_{j=1}^{\infty} \sum_{i=1}^{M_1} \alpha_i \gamma_j \phi_j(\mathbf{x}) \phi_j(\mathbf{x}_i) \\ &= \sum_{j=1}^{\infty} c_j \phi_j(\mathbf{x}) \text{ where } c_j = \gamma_j \sum_{i=1}^{M_1} \alpha_i \phi_j(\mathbf{x}_i) \end{aligned}$$

3.

$$\begin{aligned} \langle f, g \rangle_{\mathcal{H}} &= \Big\langle \sum_{i=1}^{M_1} \alpha_i K(\mathbf{x}, \mathbf{x}_i), \sum_{j=1}^{M_2} \beta_j K(\mathbf{x}, \mathbf{y}_j) \Big\rangle \\ &= \Big\langle \sum_{k=1}^{\infty} c_k \phi_k(\mathbf{x}), \sum_{l=1}^{\infty} d_l \phi_l(\mathbf{x}) \Big\rangle \\ &= \sum_{k=1}^{\infty} \sum_{l=1}^{\infty} c_k d_l \langle \phi_k(\mathbf{x}), \phi_l(\mathbf{x}) \rangle \\ &= \sum_{k=1}^{\infty} \frac{c_k d_k}{\gamma_k} \langle \phi_k(\mathbf{x}), \phi_k(\mathbf{x}) \rangle + \sum_{k \neq l} c_k d_l \langle \phi_k(\mathbf{x}), \phi_l(\mathbf{x}) \rangle \\ &= \sum_{k=1}^{\infty} \frac{c_k d_k}{\gamma_k} \text{ (since } \phi_k(\mathbf{x}) \text{ orthonormal)} \end{aligned}$$

From the above results the following follow:

- $\mathcal{H} = \Big\{ f(.) = \sum_{k=1}^{\infty} c_k \phi_k(\mathbf{x}) : \sum_{k=1}^{\infty} \frac{c_k^2}{\gamma_k} < \infty \Big\}$
- For $f(\mathbf{x}) = \sum_{k=1}^{\infty} c_k \phi_k(\mathbf{x})$ and $g(\mathbf{x}) = \sum_{k=1}^{\infty} d_k \phi_k(\mathbf{x})$, then:

$$\langle f, g \rangle_{\mathcal{H}} = \sum_{k=1}^{\infty} \frac{c_k d_k}{\gamma_k}$$

The regularization problems used in supervised learning can be written as

$$\underset{f \in \mathcal{H}}{\operatorname{argmin}} \Big\{ \sum_{i=1}^{N} \mathcal{L}(y_i, f(\mathbf{x}_i)) + \lambda J[f] \Big\}. \tag{5.2}$$

Equation 5.2 can be rewritten where $\mathcal{H}$ is a RKHS and $J[f] = ||f||_{\mathcal{H}}^2$ resulting the minimization problem

$$\underset{f \in \mathcal{H}}{\operatorname{argmin}} \Big\{ \sum_{i=1}^{N} \mathcal{L}(y_i, f(\mathbf{x}_i)) + \lambda ||f||_{\mathcal{H}}^2 \Big\}. \tag{5.3}$$

To determine the solution to the above minimization problem, consider $\mathcal{U} = \text{span}(\{K(., \mathbf{x}_i) : 1, 2, ..., N\}$ and f is any function in $\mathcal{H}$. The function f can be written as $f = g + h$ where $g \in \mathcal{U}$ and h is orthogonal to $\mathcal{U}$. From this it follows

$$\begin{aligned} f(\mathbf{x}_i) &= \langle f(.), K(., \mathbf{x}_i) \rangle_{\mathcal{H}} \\ &= \langle g(.) + h(.), K(., \mathbf{x}_i) \rangle_{\mathcal{H}} \\ &= \langle g(.), K(., \mathbf{x}_i) \rangle_{\mathcal{H}} + \langle h(.), K(., \mathbf{x}_i) \rangle_{\mathcal{H}} \\ &= g(\mathbf{x}_i). \end{aligned}$$

Furthermore,

$$\begin{aligned}||f||^2_{\mathcal{H}} &= \langle f(.), f(.)\rangle_{\mathcal{H}} \\ &= \langle g(.) + h(.), g(.) + h(.)\rangle_{\mathcal{H}} \\ &= \langle g(.), g(.)\rangle_{\mathcal{H}} + 2\langle g(.), h(.)\rangle_{\mathcal{H}} + \langle h(.), h(.)\rangle_{\mathcal{H}} \\ &= ||g||^2_{\mathcal{H}} + ||h||^2_{\mathcal{H}} \geq ||g||^2_{\mathcal{H}}.\end{aligned}$$

Thus,

$$\sum_{i=1}^{N} \mathcal{L}(y_i, f(\mathbf{x}_i)) + \lambda||f||^2_{\mathcal{H}} \geq \sum_{i=1}^{N} \mathcal{L}(y_i, g(\mathbf{x}_i)) + \lambda||g||^2_{\mathcal{H}} \tag{5.4}$$

Therefore, an objective value, that is at least as good as that generate by f, can be found by projecting f onto $\mathcal{U}$. Thus the search for the minimum of equation 5.3 can be found by searching over $\mathcal{U}$. The resulting minimizer is of the form $f(\mathbf{x}) = \sum_{i=1}^{N} \alpha_i K(\mathbf{x}, \mathbf{x}_i)$. Since,

$$\left|\left| \sum_{i=1}^{N} \alpha_i K(., \mathbf{x}_i) \right|\right|^2_{\mathcal{H}} = \sum_{i=1}^{N}\sum_{j=1}^{N} \alpha_i K(\mathbf{x}_i, \mathbf{x}_j)\alpha_j = \boldsymbol{\alpha}'\mathbf{K}\boldsymbol{\alpha}$$

equation 5.4 can be written as

$$\underset{\boldsymbol{\alpha}\in\Re^N}{\operatorname{argmin}} \left\{ \sum_{i=1}^{N} \mathcal{L}\Big(y_i, \sum_{i=j}^{N} \alpha_j K(\mathbf{x}_i, \mathbf{x}_j)\Big) + \lambda\boldsymbol{\alpha}'\mathbf{K}\boldsymbol{\alpha} \right\} \tag{5.5}$$

Many different statistical learning techniques can be derived from equation 5.5 by selecting different loss functions and kernels. The use of different RKHS's with specific kernels to create basis expansions can reduce risk and improve performance by introducing non-linearities. Using square-error loss results in the regularization network with objective

$$\underset{\boldsymbol{\alpha}\in\Re^N}{\operatorname{argmin}} \Big\{ ||\mathbf{y} - \mathbf{K}\boldsymbol{\alpha}||^2_2 + \lambda\boldsymbol{\alpha}'\mathbf{K}\boldsymbol{\alpha} \Big\}$$

with solution

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

The penalized polynomial regression can be derived by selecting the polynomial kernel $K(\mathbf{x}, \mathbf{y}) = (\langle \mathbf{x}, \mathbf{y}\rangle + 1)^d$.

Non-linear logistic regression can be derived by using the log loss function

$$\mathcal{L}(Y, f(\mathbf{X})) = -Y \log(f(\mathbf{X})) - (1 - Y)\log(1 - f(\mathbf{X}))$$

and the Gaussian radial basis kernel $K(\mathbf{x}, \mathbf{y}) = \exp(-\nu||\mathbf{x} - \mathbf{y}||^2_2)$.

The support vector machines for classification can be expressed as a penalized loss problem using the hinge loss function

$$\mathcal{L}(Y, f(\mathbf{X})) = [1 - Y f(\mathbf{X})]_+.$$

Similarly, the support vector machine for regression can be expressed as a penalized loss problem using the $\varepsilon-$sensitive loss function

$$\mathcal{L}(Y, f(\mathbf{X})) = [Y - f(\mathbf{X})]_\varepsilon.$$

5.2 Question 2

When discussing support vector machines (SVMs) it is best to begin in the context of binary classification. Binary classification problems require the prediction of a categorical variable $G \in \{\mathcal{G}_1, \mathcal{G}_2\}$ given a vector of inputs $\mathbf{X} : p \times 1$. A useful encoding of the categorical variables in the SVM literature is

$$Y = \begin{cases} -1 & \text{if } G = \mathcal{G}_1 \\ 1 & \text{if } G = \mathcal{G}_2 \end{cases}$$

For the data set $\mathcal{T} = \{(\mathbf{x}_i, y_i) : i = 1, 2, ..., N\}$, the classifier of the form

$$G(\mathbf{x}; \beta_0, \boldsymbol{\beta}) = \text{sign}(\beta_0 + \boldsymbol{\beta}'\mathbf{x})$$

can be constructed. This classifier constructs a hyperplane in $\Re^p$ defined as the set $\{\mathbf{x} : \beta_0 + \boldsymbol{\beta}'\mathbf{x} = 0\}$. A hyperplane that perfectly separates the observations in $\mathcal{T}$ is a separating hyperplane. The separating hyperplane defined by $\beta_0, \boldsymbol{\beta}$ has the property that

$$G(\mathbf{x}_i; \beta_0, \boldsymbol{\beta}) = y_i \quad \forall i = 1, 2, ..., N$$

or

$$y_i(\beta_0 + \boldsymbol{\beta}'\mathbf{x}_i) > 0 \quad \forall i = 1, 2, ..., N.$$

If such a separating hyperplane exists, the data $\mathcal{T}$ is linearly separable.

Rosenblatt's perceptron algorithm is a method of determining a separating hyperplane for linearly separable data. The algorithm attempts to minimize the following objective

$$D(\beta_0, \boldsymbol{\beta}) = -\sum_{i \in \mathcal{M}} y_i(\beta_0 + \boldsymbol{\beta}'\mathbf{x}_i) \tag{5.6}$$

where $\mathcal{M} = \{i : y_i(\beta_0 + \boldsymbol{\beta}'\mathbf{x}_i) \leq 0\}$ The objective follows since $D(\beta_0, \boldsymbol{\beta})$ is always non-negative and when $D(\beta_0, \boldsymbol{\beta}) = 0$, $\beta_0, \boldsymbol{\beta}$ defined a separating hyperplane. Stochastic gradient descent is used to minimize 5.6 using the following algorithm

---

Algorithm 11: Rosenblatt's Perceptron Algorithm

---

1. Randomly select i from $\mathcal{M}$ and compute:

$$-\frac{\partial y_i(\beta_0 + \boldsymbol{\beta}'\mathbf{x}_i)}{\partial \boldsymbol{\beta}} = -y_i\mathbf{x}_i$$
$$-\frac{\partial y_i(\beta_0 + \boldsymbol{\beta}'\mathbf{x}_i)}{\partial \beta_0} = -y_i$$

2. Update parameters as

$$\begin{bmatrix} \boldsymbol{\beta} \\ \beta_0 \end{bmatrix} \leftarrow \begin{bmatrix} \boldsymbol{\beta} \\ \beta_0 \end{bmatrix} + \rho \begin{bmatrix} y_i\mathbf{x}_i \\ y_i \end{bmatrix}$$

3. Recompute the set $\mathcal{M}$.

4. Repeat until $\mathcal{M}$ is empty.

Algorithm 11 has the following 3 fundamental problems:

1. The starting parameters determine the hyperplane selected. This is since linearly separable data contain infinitely many separating hyperplanes.

2. Convergence may require many steps.

3. Linear separability is required for convergence.

The optimal separating hyperplane (OSH) overcomes the first of the problems associated with the perceptron algorithm. The algorithm selects one from an infinite number of separating hyperplanes for linearly separable data. Which hyperplane is determined by the starting values for the algorithm and may not be the best suited hyperplane for unseen test data. The OSH overcomes this by determining the hyperplane that best separates the data. This is achieved by determining the separating hyperplane that maximises the distance between the hyperplane and nearest training observation input vector. The minimum distance from the training inputs to the separating hyperplane defined by $\beta_0, \boldsymbol{\beta}$ is given by the margin of the hyperplane defined as

$$\min_i \left\{ \frac{|\beta_0 + \boldsymbol{\beta}'\mathbf{x}_i|}{||\boldsymbol{\beta}||} \right\}.$$

The OSH optimization problem can be formulated as

$$\begin{aligned} \underset{\beta_0,\boldsymbol{\beta}}{\text{maximize}} \quad & \min_i \left\{ \frac{|\beta_0+\boldsymbol{\beta}'\mathbf{x}_i|}{||\boldsymbol{\beta}||} \right\} \\ \text{subject to} \quad & y_i(\beta_0 + \boldsymbol{\beta}'\mathbf{x}_i) > 0 \text{ for all } i = 1, 2, ..., N. \end{aligned} \tag{5.7}$$

In order to make the above problem convex, set $M = \min_i \left\{ \frac{|\beta_0+\boldsymbol{\beta}'\mathbf{x}_i|}{||\boldsymbol{\beta}||} \right\}$, then 5.7 becomes

$$\begin{aligned} \underset{\beta_0,\boldsymbol{\beta},M>0}{\text{maximize}} \quad & M \\ \text{subject to} \quad & y_i(\beta_0 + \boldsymbol{\beta}'\mathbf{x}_i) \geq M||\boldsymbol{\beta}|| \text{ for all } i = 1, 2, ..., N. \end{aligned} \tag{5.8}$$

The set $\{\mathbf{x} : |\beta_0 + \boldsymbol{\beta}'\mathbf{x}| \leq M||\boldsymbol{\beta}||\}$ defines the margin of thickness M around the hyperplane defined by $\beta_0, \boldsymbol{\beta}$. Here, each $\mathbf{x}$ is at most M distance away from the separating hyperplane thus no training observation is within the margin. The OSH seems to maximise the thickness of this margin. Assuming there is an OSH defined by $\tilde{\beta}_0, \tilde{\boldsymbol{\beta}}$, $\widehat{M}$ can be rewritten as

$$\begin{aligned} \widehat{M} &= \min_i \left\{ \frac{|\tilde{\beta}_0 + \tilde{\boldsymbol{\beta}}'\mathbf{x}_i|}{||\tilde{\boldsymbol{\beta}}||} \right\} \\ &= \frac{y_j(\tilde{\beta}_0 + \tilde{\boldsymbol{\beta}}'\mathbf{x}_j)}{||\tilde{\boldsymbol{\beta}}||_2} \\ &= \frac{c}{||\tilde{\boldsymbol{\beta}}||_2} \\ &= \frac{1}{||\hat{\boldsymbol{\beta}}||_2} \quad \left(\text{where } \hat{\boldsymbol{\beta}} = \frac{\tilde{\boldsymbol{\beta}}}{c} \right) \end{aligned}$$

and

$$\begin{aligned} y_i(\tilde{\beta}_0 + \tilde{\boldsymbol{\beta}}'\mathbf{x}_i) &\geq c \quad \forall_i \\ y_i(\frac{\tilde{\beta}_0}{c} + \hat{\boldsymbol{\beta}}'\mathbf{x}_i) &\geq 1 \quad \forall_i \\ y_i(\hat{\beta}_0 + \hat{\boldsymbol{\beta}}'\mathbf{x}_i) &\geq 1 \quad \forall_i. \end{aligned}$$

Following the above, the optimisation problem give in 5.8 can be reformulated as

$$\begin{aligned} \underset{\beta_0,\boldsymbol{\beta}}{\text{maximize}} \quad & \tfrac{1}{||\boldsymbol{\beta}||_2} \\ \text{subject to} \quad & y_i(\beta_0 + \boldsymbol{\beta}'\mathbf{x}_i) \geq 1 \text{ for all } i = 1, 2, ..., N. \end{aligned}$$

or equivalently, the quadratic programming (QP) problem

$$\begin{aligned} \underset{\beta_0,\boldsymbol{\beta}}{\text{minimize}} \quad & \tfrac{1}{2}||\boldsymbol{\beta}||_2^2 \\ \text{subject to} \quad & y_i(\beta_0 + \boldsymbol{\beta}'\mathbf{x}_i) \geq 1 \text{ for all } i = 1, 2, ..., N. \end{aligned}$$

Using the KKT conditions, the primal problem can be formulated as

$$\begin{aligned} \underset{\beta_0,\boldsymbol{\beta}}{\text{minimize}} \quad & \tfrac{1}{2}\boldsymbol{\beta}'\boldsymbol{\beta} \\ \text{subject to} \quad & 1 - y_i(\beta_0 + \boldsymbol{\beta}'\mathbf{x}_i) \leq 0 \text{ for all } i = 1, 2, ..., N. \end{aligned}$$

which can be solved by setting the derivatives of the following Lagrangian with regards to $\beta_0, \boldsymbol{\beta}$ equal to zero as follows

$$\mathcal{L}(\beta_0, \boldsymbol{\beta}, \boldsymbol{\alpha}) = \frac{1}{2}\boldsymbol{\beta}'\boldsymbol{\beta} + \sum_{i=1}^{N} \alpha_i(1 - y_i(\beta_0 + \boldsymbol{\beta}'\mathbf{x}_i)) \tag{5.9}$$

$$(\nabla_{\beta_0}\mathcal{L})(\beta_0, \boldsymbol{\beta}, \boldsymbol{\alpha}) = -\sum_{i=1}^{N} \alpha_i y_i = 0 \tag{5.10}$$

$$(\nabla_{\boldsymbol{\beta}}\mathcal{L})(\beta_0, \boldsymbol{\beta}, \boldsymbol{\alpha}) = \boldsymbol{\beta} - \sum_{i=1}^{N} \alpha_i y_i \mathbf{x}_i = 0 \tag{5.11}$$

The dual optimisation problem can be formulated by substituting 5.10 and 5.11 into 5.9 resulting in

$$\begin{aligned} \underset{\alpha_1,\alpha_2,...,\alpha_N}{\text{maximize}} \quad & \textstyle\sum_{i=1}^{N} \alpha_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} \alpha_i\alpha_j y_i y_j \mathbf{x}_i'\mathbf{x}_j \\ \text{subject to} \quad & \alpha_i \geq 0; \text{ for all } i = 1, 2, ..., N \\ & \textstyle\sum_{i=1}^{N} \alpha_i y_i = 0 \end{aligned}$$

which can be rewritten in matrix notation as a QP problem as

$$\begin{aligned} \underset{\boldsymbol{\alpha}}{\text{minimize}} \quad & \tfrac{1}{2}\boldsymbol{\alpha}'\mathbf{G}\boldsymbol{\alpha} - \mathbf{1}'\boldsymbol{\alpha} \\ \text{subject to} \quad & \alpha_i \geq 0; \text{ for all } i = 1, 2, ..., N \\ & \textstyle\sum_{i=1}^{N} \alpha_i y_i = 0 \end{aligned}$$

where $\mathbf{G}_{ij} = y_i y_j \mathbf{x}_i \mathbf{x}_j$. From the KKT conditions:

1. Primal Feasibility: $1 - y_i(\beta_0 + \boldsymbol{\beta}'\mathbf{x}_i) \leq 0$ for all $i = 1, 2, ..., N$
   Dual Feasibility: $\alpha_i \geq 0$ for all $i = 1, 2, ..., N$
2. Stationary: $\hat{\boldsymbol{\beta}} = \sum_{i=1}^{N} \hat{\alpha}_i y_i \mathbf{x}_i$
3. Complementary Slackness: $\hat{\alpha}_i[y_i(\beta_0 + \boldsymbol{\beta}'\mathbf{x}_i - 1] = 0$ for all $i = 1, 2, ..., N$.

Thus, only the support vectors contained in the set $\{\mathbf{x}_i : \alpha_i > 0\}$ contribute to $\boldsymbol{\beta}$. Since, for the support vectors $\alpha_i > 0$, therefore $y_i(\beta_0 + \boldsymbol{\beta}'\mathbf{x}_i) = 1$ by complementary slackness. This means that the support vectors are on the boundary of the margin. Additionally, this means that if $\mathbf{x}_i$ lies outside of the margin and $y_i(\beta_0 + \boldsymbol{\beta}'\mathbf{x}_i) > 1$ then $\alpha_i = 0$. In order to calculate $\hat{\beta}_0$, the equation $y_i(\hat{\beta}_0 + \boldsymbol{\beta}'\mathbf{x}_i) = 1$ is solved.

The OSH, however, has the following 2 fundamental problems:

1. It requires the data to be linearly separable.
2. It is very sensitive to outliers.

The support vector classifier (SVC) overcomes these issues by relaxing the constraints of the OSH. This allows for data that are not linearly separable and thus is less sensitive to outliers. Following the objective given in 5.8, the OSH attempts to maximise the thickness of the margin that does not contain any observations and there are not observations on the incorrect side of the hyperplane. This is infeasible when the data are not linearly separable. The SVC relaxes this constraint through the addition of slack variables that may be within the margin or be misclassified and lie on the incorrect side of the hyperplane. This is done by introducing a slack variable $\xi_i \geq 0$ to each observation $(\mathbf{x}_i, y_i)$ resulting in the constraint in objective 5.8 becoming

$$y_i(\beta_0 + \boldsymbol{\beta}'\mathbf{x}_i) \geq M(1 - \xi_i)||\boldsymbol{\beta}|| \text{ for all } i = 1, 2, ..., N.$$

The slack variables, ξ_i can take on 3 forms

1. $\xi_i = 0$: The observation $\mathbf{x}_i$ is correctly classified and lies either outside or on the boundary of the margin.
2. $0 < \xi_i < 1$: The observation $\mathbf{x}_i$ is correctly classified but may lie within the margin.
3. $\xi_i \geq 1$: The observation $\mathbf{x}_i$ may be misclassified.

Incorporating slack variables into the objective 5.8, results in

$$\begin{aligned} \underset{\beta_0, \boldsymbol{\beta}, M>0}{\text{maximize}} \quad & M \\ \text{subject to} \quad & y_i(\beta_0 + \boldsymbol{\beta}'\mathbf{x}_i) \geq M(1 - \xi_i)||\boldsymbol{\beta}|| \text{ for all } i = 1, 2, ..., N. \end{aligned}$$

However, specifying each ξ_i for all $i = 1, 2, ..., N$ is inconvenient. This can be avoided by including the decision of the slack variables in the optimisation problem as follows

$$\begin{aligned} \underset{\beta_0, \boldsymbol{\beta}, M>0, \boldsymbol{\xi}}{\text{maximize}} \quad & M \\ \text{subject to} \quad & y_i(\beta_0 + \boldsymbol{\beta}'\mathbf{x}_i) \geq M(1 - \xi_i)||\boldsymbol{\beta}|| \text{ for all } i = 1, 2, ..., N \\ & \xi_i \geq 0 \text{ for all } i = 1, 2, ..., N. \end{aligned} \tag{5.12}$$

The above equation, however, can be made arbitrarily large by including sufficient slack observations. It is preferred to have as few slack variables are possible while overcoming the problems relating to the OSH. This is achieved by altering the objective 5.12 to

$$\begin{aligned} \underset{\beta_0, \boldsymbol{\beta}, M>0, \boldsymbol{\xi}}{\text{maximize}} \quad & M \\ \text{subject to} \quad & y_i(\beta_0 + \boldsymbol{\beta}'\mathbf{x}_i) \geq M(1 - \xi_i)||\boldsymbol{\beta}|| \text{ for all } i = 1, 2, ..., N \\ & \xi_i \geq 0 \text{ for all } i = 1, 2, ..., N. \\ & \textstyle\sum_{i=1}^{N} \xi_i \leq K \end{aligned} \tag{5.13}$$

.

Here, the constraint $\sum_{i=1}^{N} \xi_i \leq K$ acts as a bound to the number of misclassified training observations. Following similar reasoning, the optimisation problem in 5.13 can be rewritten as a convex minimisation problem

$$\begin{aligned} \underset{\beta_0,\boldsymbol{\beta},\boldsymbol{\xi}}{\text{maximize}} \quad & \tfrac{1}{2}||\boldsymbol{\beta}||^2 \\ \text{subject to} \quad & y_i(\beta_0 + \boldsymbol{\beta}'\mathbf{x}_i) \geq 1 - \xi_i \text{ for all } i = 1, 2, ..., N \\ & \xi_i \geq 0 \text{ for all } i = 1, 2, ..., N. \\ & \textstyle\sum_{i=1}^{N} \xi_i \leq K \end{aligned} \quad .$$

Through the use of a multiplier, the constraint $\sum_{i=1}^{N} \xi_i \leq K$ can be incorporated into the objective as follows

$$\begin{aligned} \underset{\beta_0,\boldsymbol{\beta},\boldsymbol{\xi}}{\text{maximize}} \quad & \tfrac{1}{2}||\boldsymbol{\beta}||^2 + C\textstyle\sum_{i=1}^{N} \xi_i \\ \text{subject to} \quad & y_i(\beta_0 + \boldsymbol{\beta}'\mathbf{x}_i) \geq 1 - \xi_i \text{ for all } i = 1, 2, ..., N \\ & \xi_i \geq 0 \text{ for all } i = 1, 2, ..., N \end{aligned} \tag{5.14}$$

where C acts as a cost parameter for the number of training observations on the incorrect side of the margin. This is a hyperparameter. The problem in 5.14 is the primal SVC problem and is a QP problem. The dual problem follows similarly from the OSH and is given by

$$\begin{aligned} \underset{\boldsymbol{\alpha}}{\text{maximize}} \quad & \mathbf{1}'\boldsymbol{\alpha} - \tfrac{1}{2}\boldsymbol{\alpha}'\mathbf{G}\boldsymbol{\alpha} \\ \text{subject to} \quad & 0 \leq \alpha_i \leq C; \text{ for all } i = 1, 2, ..., N \\ & \textstyle\sum_{i=1}^{N} \alpha_i y_i = 0 \end{aligned} \tag{5.15}$$

where $\mathbf{G}_{ij} = y_i y_j \mathbf{x}_i \mathbf{x}_j$. The KKT conditions are as follows:

1. Stationarity
   (a) $\hat{\boldsymbol{\beta}} = \sum_{i=1}^{N} \hat{\alpha}_i y_i \mathbf{x}_i$
   (b) $\sum_{i=1}^{N} \hat{\alpha}_i y_i = 0$
   (c) $\hat{\alpha}_i = C - \hat{u}_i$
2. Primal Feasibility
   (a) $y_i(\hat{\beta}_0 + \hat{\boldsymbol{\beta}}'\mathbf{x}_i) \geq 1 - \hat{\xi}_i$ for all $i = 1, 2, ..., N$
   (b) $\hat{\xi}_i \geq 0$ for all $i = 1, 2, ..., N$
3. Dual Feasibility
   (a) $\hat{\alpha}_i \geq 0$ for all $i = 1, 2, ..., N$
   (b) $\hat{u}_i \geq 0$ for all $i = 1, 2, ..., N$
4. Complementary Slackness
   (a) $\hat{\alpha}_i[y_i(\hat{\beta}_0 + \hat{\boldsymbol{\beta}}'\mathbf{x}_i) - (1 - \hat{\xi}_i)] = 0$ for all $i = 1, 2, ..., N$
   (b) $\hat{u}_i\hat{\xi}_i = 0$ for all $i = 1, 2, ..., N$

An alternative perspective for formulating the SVC problem is through a regularization or penalty + loss framework. The constraint in the primal given in 5.14 can be rewritten as $\xi_i = \max(0, 1 - y_i(\beta_0 + \boldsymbol{\beta}'\mathbf{x}_i))$ which allows the primal to be rewritten as minimize$_{\beta_0,\boldsymbol{\beta}}$ $\frac{1}{2}||\boldsymbol{\beta}||^2 + C\sum_{i=1}^{N}\max(0, 1 - y_i(\beta_0 + \boldsymbol{\beta}'\mathbf{x}_i))$. By setting $\lambda = \frac{1}{C}$ and using Hinge loss, $\mathcal{L}(y, f(\mathbf{x})) = [1 - yf(\mathbf{x})]_+$, the primal can be reformulated as

$$\min_{\beta_0,\boldsymbol{\beta}}\left\{\sum_{i=1}^{N}\mathcal{L}(y_i, f(\mathbf{x_i})) + \frac{\lambda}{2}||\boldsymbol{\beta}||^2\right\} \tag{5.16}$$

where $f(\mathbf{x}_i) = \beta_0 + \boldsymbol{\beta}'\mathbf{x}_i$

While SVCs overcome some of the drawbacks from OSHs, they rely on linear decison boundaries. To overcome this drawback, support vector machines (SVMs) make use of basis expansions involving kernels. SVMs can be motivated through the analysis of reproducing kernel Hilbert spaces (RKHSs). RKHSs are discussed in detail in Question 5.1. From Question 5.1, equation 5.1 described the eigen expansion of a RKHS. This equation can be rewritten by setting $h_m(\mathbf{x}) = \sqrt{\gamma_m}\phi_m(\mathbf{x})$ resulting in

$$K(\mathbf{x}, \mathbf{y}) = \sum_{m=1}^{\infty} h_m(\mathbf{x})h_m(\mathbf{y}).$$

The goal of the SVC is fit a classifier of the form

$$G(\mathbf{x}) = \text{sign}(f(\mathbf{x})) \tag{5.17}$$

where $f(\mathbf{x}) = \beta_0 + \boldsymbol{\beta}'\mathbf{x}$. It is possible to create a non-linear decision boundary by using a basis expansion. The more general form of the $f(\mathbf{x})$ can be achieved by specifying function $h_m(.)$ for $m = 1, 2, ..., M$ resulting in

$$f(\mathbf{x}) = \beta_0 + \sum_{m=1}^{M}\beta_m h_m(\mathbf{x}).$$

The SVC classifier can create non-linear decision boundaries by estimating $\beta_0, \boldsymbol{\beta}$ using the derived features. In order to include basis expansions to the SVC, the dual problem is considered. The $\mathbf{G}$ matrix in 5.15 can be written in terms of an inner product where $\mathbf{G}_{ij} = y_i y_j \langle \mathbf{x}_i, \mathbf{x}_j \rangle$. From this, the fitted function in equation 5.17 becomes

$$\hat{f}(\mathbf{x}) = \hat{\beta}_0 + \hat{\boldsymbol{\beta}}'\mathbf{x} = \hat{\beta}_0 + \sum_{i=1}^{N} y_i\hat{\alpha}_i\langle \mathbf{x}, \mathbf{x}_i \rangle$$

which follows from the stationarity KKT condition. A basis expansion can be achieved by including the basis functions $h_m(.)$ for $m = 1, 2, ..., M$ in the inner product of the $\mathbf{G}$ matrix in 5.15 resulting in

$$\mathbf{G}_{ij} = y_i y_j \langle \mathbf{h}(\mathbf{x}_i), \mathbf{h}(\mathbf{x}_j) \rangle$$

where $\langle \mathbf{h}(\mathbf{x}), \mathbf{h}(\mathbf{y}) \rangle = \sum_{m=1}^{M} h_m(\mathbf{x})h_m(\mathbf{y})$. The incorporation of the inner product of the basis expansions can be achieved through the use of a kernel function. The resulting fitted function is given by

$$\hat{f}(\mathbf{x}) = \hat{\beta}_0 + \sum_{i=1}^{N} y_i\hat{\alpha}_i\langle \mathbf{h}(\mathbf{x}), \mathbf{h}(\mathbf{x}_i) \rangle = \hat{\beta}_0 + \sum_{i=1}^{N} y_i\hat{\alpha}_i K(\mathbf{x}, \mathbf{x}_i).$$

This results in the dual problem for the SVM.

In order to formulate the primal problem for the SVM, consider the minimisation problem in equation 5.16. By including the eigen expansion of a kernel, the objective of the minimisation problem becomes

$$\sum_{i=1}^{N} \mathcal{L}\Big(y_i, \beta_0 + \sum_{m=1}^{\infty} c_m \phi_m(\mathbf{x}_i)\Big) + \frac{\lambda}{2} \sum_{m=1}^{\infty} \frac{c_m^2}{\gamma_m}$$

which is minimised by a function of the form

$$f(\mathbf{x}) = \beta_0 + \sum_{i=1}^{N} \tau_i K(\mathbf{x}, \mathbf{x}_i).$$

Thus the resulting SVM minimisation problem becomes

$$\min_{\beta_0, \boldsymbol{\tau}} \left\{ \sum_{i=1}^{N} \Big[1 - y_i\big(\beta_0 + \sum_{j=1}^{N} \tau_j K(\mathbf{x}_i, \mathbf{x}_j)\big)\Big]_+ + \frac{\lambda}{2} \boldsymbol{\tau}' \boldsymbol{K} \boldsymbol{\tau} \right\}.$$

The above equation can be reformulate as

$$\begin{aligned} \underset{\beta_0, \boldsymbol{\tau}, \boldsymbol{\xi}}{\text{maximize}} \quad & \tfrac{1}{2} \boldsymbol{\tau}' \boldsymbol{K} \boldsymbol{\tau} + C \textstyle\sum_{i=1}^{N} \xi_i \\ \text{subject to} \quad & y_i\big(\beta_0 + \textstyle\sum_{j=1}^{N} \tau_j K(\mathbf{x}_i, \mathbf{x}_j)\big) \geq 1 - \xi_i \text{ for all } i = 1, 2, ..., N \\ & \xi_i \geq 0 \text{ for all } i = 1, 2, ..., N \end{aligned}$$

SVMs can be extended to regression problems by changing the loss function in the primal problem. The SVM for classification makes use of the Hinge loss function. When this is replaced by the ε-sensitive loss function $\mathcal{L}(y, f(\mathbf{x})) = V_\varepsilon(y - f(\mathbf{x}))$ where

$$V_\epsilon(r) = \begin{cases} 0 & \text{if } |r| < \varepsilon \\ |r| - \varepsilon & \text{otherwise} \end{cases}$$

The primal is formulated by introducing an additional slack variable into the objective resulting in

$$\begin{aligned} \underset{\beta_0, \boldsymbol{\tau}, \boldsymbol{\xi}, \boldsymbol{\xi^*}}{\text{maximize}} \quad & \tfrac{1}{2} \boldsymbol{\tau}' \boldsymbol{K} \boldsymbol{\tau} + C \textstyle\sum_{i=1}^{N} (\xi_i + \xi_i^*) \\ \text{subject to} \quad & \beta_0 + \textstyle\sum_{j=1}^{N} \tau_j K(\mathbf{x}_i, \mathbf{x}_j) - y_i \leq \varepsilon + \xi_i \text{ for all } i = 1, 2, ..., N \\ & y_i - \beta_0 - \textstyle\sum_{j=1}^{N} \tau_j K(\mathbf{x}_i, \mathbf{x}_j) \leq \varepsilon + \xi_i^* \text{ for all } i = 1, 2, ..., N \\ & \xi_i, \xi_i^* \geq 0 \text{ for all } i = 1, 2, ..., N. \end{aligned}$$

The dual optimisation problem is given by

$$\underset{\boldsymbol{\alpha}\boldsymbol{\alpha^*}}{\text{maximize}} \quad -\varepsilon \mathbf{1}'(\boldsymbol{\alpha} + \boldsymbol{\alpha}^*) + \mathbf{y}'(\boldsymbol{\alpha}^* - \boldsymbol{\alpha}) - \tfrac{1}{2}(\boldsymbol{\alpha}^* - \boldsymbol{\alpha})' \mathbf{G} (\boldsymbol{\alpha}^* - \boldsymbol{\alpha})$$

$$\text{subject to} \quad 0 \leq \alpha_i, \alpha_i^* \leq C; \text{ for all } i = 1, 2, ..., N$$

$$\textstyle\sum_{i=1}^{N} (\alpha_i^* - \alpha_i) = 0$$

where $\mathbf{G}_{ij} = K(\mathbf{x}_i, \mathbf{x}_j)$.

SVM can be extended to multiclass classification problems. This is done through one of two processes. The first being the one-versus-one (OVO) approach. In the OVO approach, all $\binom{K}{2}$ pairwise binary classification models are computed. Thus for each test observation, the predicted class is the class which wins the most pairwise contests. The second approach is the one-versus-all (OVA) approach. The OVA approach requires for each class the observations to be assigned to that class or not that class then for a classification SVM to be estimated. Thus only K classifiers are required. Test observations are assigned according to $\hat{G}(\mathbf{x}) = \text{argmax}_k\{\hat{f}_k(\mathbf{x})\}$.

Assignment 6

6.1 Question 1

Neural networks have fast become an extremely powerful class of learning algorithms. Although the initial ideas for the models have been around for decades, modern computing and massive amounts of data have allowed for the full potential of this class of models to be reached. The backbone of this class of models is the fully connected deep neural network also known as the multilayer perceptron (MLP). MLPs are based on a number of sequential linear and non-linear transformations of the data. These transformations are used to create predictions that are then compared to the observed value from which the empirical risk is determined. The empirical risk is minimized by altering the transformations through gradient descent.

A single input for a MLP, given by $\mathbf{x} \sim \Re^p$, enters the first linear node with the linear function, denoted $\boldsymbol{\Phi}(\cdot)$,

$$\mathbf{z} = \boldsymbol{\Phi}(\mathbf{x}) = \mathbf{b} + \mathbf{W}\mathbf{x} \tag{6.1}$$

where $\mathbf{b} \sim \Re^{q_1}$ is the bias and $\mathbf{W} \sim \Re^{p \times q_1}$ are the weights. The output of the linear transformation node is denoted by $\mathbf{z} \sim \Re^{q_1}$. The output then enters a non-linear activation function denoted $\boldsymbol{\sigma}(\mathbf{h})$ which outputs $\mathbf{z} \sim \Re^{q_1}$. A number of activation functions exist including the following

1. sigmoid function: $\sigma(z) = \frac{1}{1+e^{-z}}$
2. ReLu function: $\sigma(z) = \max(0, z)$
3. tanh function: $\sigma(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$

where z is a scalar value. With the input $\mathbf{z}$, the activation function outputs $\boldsymbol{\sigma}(\mathbf{z}) = \mathbf{h}$ where $\sigma(\mathbf{z}_{ij}) = \mathbf{h}_{ij}$. The combination of a linear and non-linear transformation is known as a hidden layer. In an MLP, the output from the first hidden layer, $\mathbf{h}_1$, is fed into a second hidden layer with bias $\mathbf{b}_2 \sim \Re^{q_2}$, weights $\mathbf{W}_2 \sim \Re^{q_2 \times q_1}$ and activation function $\boldsymbol{\sigma}(\cdot)$. This process of sequentially feeding the output from one hidden layer into another hidden layer is repeated for however many hidden layers that are specified in the architecture of the MLP. The final hidden layer has the following structure

$$f = \boldsymbol{\Phi}(\mathbf{x}) = \beta_0 + \boldsymbol{\beta}^T \mathbf{h}_L$$

where $\beta_0 \sim \Re$, $\boldsymbol{\beta} \sim \Re^{q_1}$ and $f \sim \Re$ for regression and $f \sim \Re^k$ for classification, where k is the number of classes. The output f is fed into an activation function denoted $s(\cdot)$ which is given as $\hat{y} = s(f) = f$ for regression problems and $\hat{p}_k = s_k(f) = \frac{e^{f_k}}{\sum_{j=1}^{K} e^{f_j}}$ for classification problems. The output from the final activation function is fed into the loss function in order to determine the contribution of the prediction to the empirical risk. In MLPs, for regression squared error loss is generally used where $\mathcal{L}(y, \hat{y}) = (y - \hat{y})^2$ and for classification negative log-loss is generally used where $\mathcal{L}(y, \hat{p}_k) = -\log(\hat{p}_k)$.

Figure 6.1 is a diagram of the architecture of a standard MLP for a single observation. Here it is clear how the data input, $\mathbf{x}$, feeds into the first linear and non-linear transformation nodes which make up the first hidden layer. The output from the first hidden layer feeds into the second hidden layer and so on until the first hidden layer that outputs a prediction $\hat{y}$ or $\hat{p}_k$. This process is known as a forwards pass. The parameters of an MLP consist of all the bias vectors and weights matrices $\mathbf{b}_l, \mathbf{W}_l : l = 1, 2, ..., L$ where L is the number of hidden layers. These parameters are optimised to minimise the empirical risk from the training data.

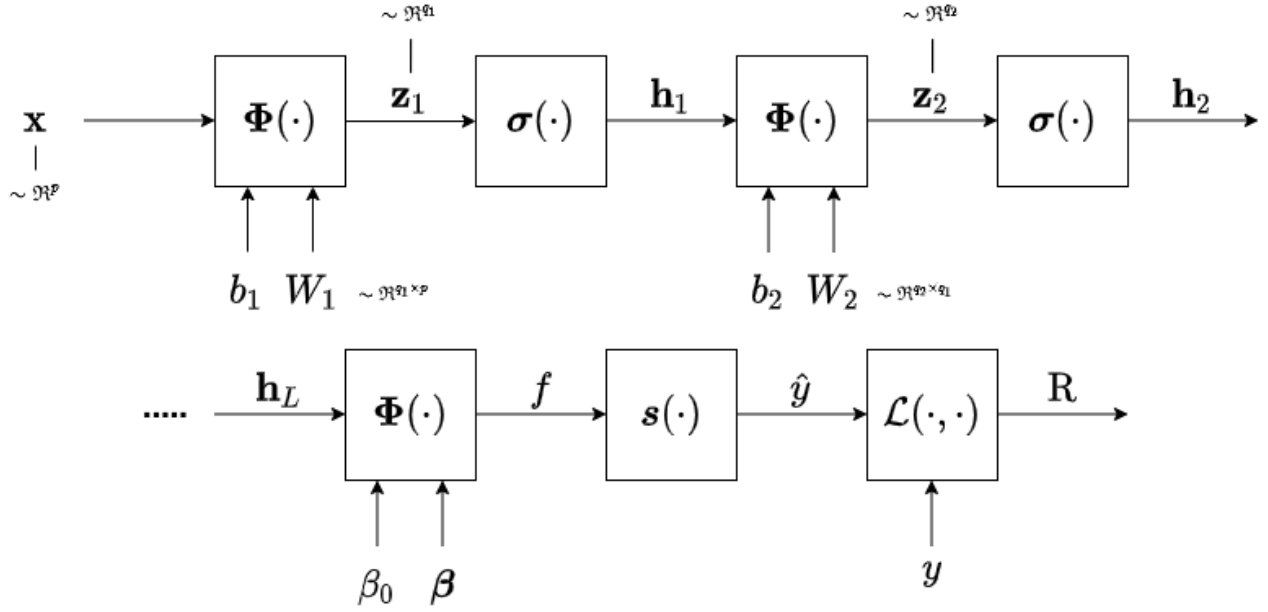

Figure 6.1: Multilayer Perceptron Architecture

The above explanation is for a single observation during the forwards pass in an MLP. Generally, batches of training observations are passed through the MLP during a forwards pass before the parameters are optimised. A batch of data is denoted $\mathbf{X}_b \sim \Re^{B \times p}$ where B is the batch size. Equation 6.1 can thus be reformulated as

$$\mathbf{H}_1 = \boldsymbol{\Phi}(\mathbf{X}) = \mathbf{X}\mathbf{W}_1 + \mathbf{j}\mathbf{b}_1^T$$

where $\mathbf{H}_1 \sim \Re^{N \times q_1}, \mathbf{W}_1$ and $\mathbf{b}_1$ remain as above and $\mathbf{j} = (1, 1, ..., 1)^T \sim \Re^N$. The activation functions are given by $\boldsymbol{\sigma}(\mathbf{H})$ where the function is computed element-wise. The final vector of predictions $\mathbf{f}$ for regression or matrix of predictions for classification $\widehat{\mathbf{P}}$ is fed row-wise into the loss function from which the empirical risk is determined.

In order to minimize the empirical risk during training the parameters been to be optimised. The parameters

in MLPs are optimised using gradient descent algorithms. These algorithms require the gradient of the risk in terms of the different parameters. Automatic differentiation through backpropagation is used determine the gradients. Backpropagation works by iteratively applying the chain rule. During the forwards pass of an MLP, inputs are fed through nodes which return outputs that become the input for the next node. Backpropagation makes use of this structure to determine the gradients of output with regards to the parameters in each node. This process begins by initializing the accumulators for each gradient. The accumulator for the risk is initialized as $\delta_{R_i} = 1$ while the remaining accumulators are initialized to equal 0. The backwards pass for a single observation begins with

$$\delta_{\hat{y}} = \delta_{\hat{y}} + \frac{\partial R_i}{\partial \hat{y}} \cdot \delta_{R_i}$$

for regression problems and

$$\delta_{\hat{p}_k} = \delta_{\hat{p}_k} + \frac{\partial R_i}{\partial \hat{p}_k} \cdot \delta_{R_i}$$

for classification problems. The updated accumulator for $\hat{y}$ (or $\hat{p}_k$) is then used to update the next accumulator, which is from the final hidden layer, given by

$$\boldsymbol{\delta}_{\mathbf{f}} = \boldsymbol{\delta}_{\mathbf{f}} + \frac{\partial \hat{y}}{\partial \mathbf{f}} \cdot \delta_{\hat{y}}$$

The accumulators for two sets of parameters, β_0 and $\boldsymbol{\beta}$, and the input to hidden layer $\mathbf{h}_L$ are updated in the following manner

$$\delta_{\beta_0} = \delta_{\beta_0} + \frac{\partial \mathbf{f}}{\partial \beta_0} \cdot \boldsymbol{\delta}_{\mathbf{f}}$$
$$\boldsymbol{\delta}_{\boldsymbol{\beta}} = \boldsymbol{\delta}_{\boldsymbol{\beta}} + \frac{\partial \mathbf{f}}{\partial \boldsymbol{\beta}} \cdot \boldsymbol{\delta}_{\mathbf{f}}$$
$$\boldsymbol{\delta}_{\mathbf{h}_L} = \boldsymbol{\delta}_{\mathbf{h}_L} + \frac{\partial \mathbf{f}}{\partial \mathbf{h}_L} \cdot \boldsymbol{\delta}_{\mathbf{f}}$$

The hidden layers are then backpropagated through as follows:

For $l = L, L-1, ..., 1$:

First through the non-linear activation function

$$\boldsymbol{\delta}_{\mathbf{z}_l} = \boldsymbol{\delta}_{\mathbf{z}_l} + \frac{\partial \mathbf{h}_l}{\partial \mathbf{z}_l} \cdot \boldsymbol{\delta}_{\mathbf{h}_l}$$

Second through the linear transformation

$$\boldsymbol{\delta}_{\mathbf{b}_l} = \boldsymbol{\delta}_{\mathbf{b}_l} + \frac{\partial \mathbf{z}_l}{\partial \mathbf{b}_l} \cdot \boldsymbol{\delta}_{\mathbf{z}_l}$$
$$\boldsymbol{\delta}_{\mathbf{W}_l} = \boldsymbol{\delta}_{\mathbf{W}_l} + \frac{\partial \mathbf{z}_l}{\partial \mathbf{W}_l} \cdot \boldsymbol{\delta}_{\mathbf{z}_l}$$
$$\boldsymbol{\delta}_{\mathbf{h}_{l-1}} = \boldsymbol{\delta}_{\mathbf{h}_{l-1}} + \frac{\partial \mathbf{z}_l}{\partial \mathbf{h}_{l-1}} \cdot \boldsymbol{\delta}_{\mathbf{z}_L}$$

The derivatives associated with individually backpropagation steps in the backwards pass are given below. For loss functions mentioned above for regression and classification problems the derivatives are given by

$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = -2(y - \hat{y})$$
$$\frac{\partial \mathcal{L}}{\partial \hat{p}_k} = \begin{cases} -\frac{1}{\hat{p}_k} & : k = y \\ 0 & : \text{otherwise.} \end{cases}$$

For the different activation functions mentioned above the derivatives are given by

$$\text{Sigmoid} : \frac{\partial h}{\partial z} = -(1+e^{-z})^{-2} - e^{-z} = \sigma(z)(1-\sigma(z))$$

$$\text{ReLu} : \frac{\partial h}{\partial z} = \begin{cases} 0 & : z \leq 0 \\ 1 & : z > 0 \end{cases}$$

$$\tanh : \frac{\partial h}{\partial z} = 1 - \frac{(e^{z} - e^{-z})^2}{(e^{z} + e^{-z})^2} = 1 - (\sigma(z))^2$$

The derivatives for the linear transformation as given by

$$\frac{\partial \mathbf{z}_l}{\partial \mathbf{b}_l} = \mathbf{I}_{q_l}$$

$$\frac{\partial \mathbf{z}_l}{\partial \mathbf{W}_{lij}} = z_j \mathbf{e}_i$$

$$\frac{\partial \mathbf{z}_l}{\partial \mathbf{h}_{l-1}} = \mathbf{W}^T$$

where $\mathbf{e}_i = (0, ..., 1, ..., 0)$. As mentioned above in the discussion of the forwards pass, generally MLPs are trained using batches of data. This does not greatly change the backwards pass. The accumulators are calculated individually for each observation. The final gradient of the parameter with regards to the risk is then taken as the sum over all the observations of the gradients calculated. These gradients are then used to optimise the parameters.

Gradient descent algorithms make use of the gradients from the backwards pass to optimise the parameters in MLPs to minimize the risk. Traditional gradient descent is given by

$$\theta^t = \theta^{t-1} - \gamma \nabla_\theta R(\theta)$$

where γ is a learning parameter that reduces the step size of the update. MLP parameter optimisation is associated with many problems, including many local minima, that result in traditional gradient descent methods performing poorly. Of the many proposed gradient descent algorithms, the Adam optimiser generally performs best. The Adam optimiser makes use of stochastic gradient descent with momentum to improve optimisation. Stochastic gradient descent differs in the fact that rather than sum the gradients of the risk in terms of the parameters for each observation, the update is applied using only the gradient from individual observations or batches of observations. This is done to better generalise over the stochastic and noisy risk function in terms of the data and the parameters.

The Adam optimiser makes use the gradient and the gradient squared. Adam makes use of exponential moving-averages of the gradient and gradient squared, which are estimates for the 1^{st} moment (mean) and 2^{nd} moment (the uncentered variance), to better approximate the gradient over stochastic samples. However, the moving-averages create bias in the gradients, Adam takes great care to reduce the bias in the update and ensure that the step-size of each update is ideal. This is achieved through the use of momentum parameters $\beta_1, \beta_2 \in [0, 1)$. The Adam optimiser is given in algorithm 12

Due to the powerful performance abilities of neural networks, they have a tendency to very closely fit the training data at the expense of reducing the generalization error. This overfitting is combated through

Algorithm 12: Adam Optimiser

Inputs: learning rate α, 1^{st} momentum parameter β_1 and 2^{nd} momentum parameter β_2

1. Initialize:

$$\theta^{(0)} \rightarrow \text{initial solution}$$
$$\mathbf{m}^{(0)} = 0 \rightarrow 1^{st} \text{ momentum vector}$$
$$\mathbf{v}^{(0)} = 0 \rightarrow 2^{nd} \text{ momentum vector}$$

2. Repeat until convergence:

$$\mathbf{g}^{(t)} = (\nabla_\theta f^{(t)})(\boldsymbol{\theta}^{(t-1)})$$
$$\mathbf{m}^{(t)} = \beta_1 \mathbf{m}^{(t-1)} + (1-\beta_1)\mathbf{g}^{(t)}$$
$$\mathbf{v}^{(t)} = \beta_2 \mathbf{v}^{(t-1)} + (1-\beta_2)\mathbf{g}^{(t)} \odot \mathbf{g}^{(t)}$$
$$\widehat{\mathbf{m}}^{(t)} = \frac{1}{1-\beta_1^t}\mathbf{m}^{(t)}$$
$$\widehat{\mathbf{v}}^{(t)} = \frac{1}{1-\beta_2^t}\mathbf{v}^{(t)}$$
$$\boldsymbol{\theta}^{(t)} \leftarrow \boldsymbol{\theta}^{(t-1)} - \alpha \cdot \frac{\widehat{\mathbf{m}}^{(t)}}{(\sqrt{\widehat{\mathbf{v}}^{(t)}} + \epsilon)}$$

regularization. There are many different regularization techniques used in MLPs but the most common include: weight decay, early-stopping, dropout and batch normalization.

Weight decay follows the traditional form of regularisation seen in ridge regression and other learning algorithms. Weight decay is implemented by including the L_2-norm of the weights in the calculation of the risk. For a MLP where a batch is given by $\mathcal{T} = \{(\mathbf{x}_i, \mathbf{y}_i) : i = 1, 2, ..., n\}$ the risk is formulated as

$$\mathrm{R}(\boldsymbol{\Theta}; \mathcal{T}, \lambda) = \sum_{i=1}^{n} \mathcal{L}\big(y_i, \mathbf{f}(\mathbf{x}_i; \boldsymbol{\Theta})\big) + \frac{\lambda}{2} \sum_{l=1}^{L} \sum_{j=1}^{d_l} ||\mathbf{w}_j^{(l)}||_2^2$$

where $\mathbf{w}_j^{(l)}$ is the j^{th} column in the weight matrix from the l^{th} hidden layer. Weight decay influences the the gradient of the weight matrices in the backwards pass as follows

$$\Delta_{\mathbf{w}^{(l)}} = \frac{\partial}{\partial \mathbf{w}} \frac{\lambda}{2} ||\mathbf{w}_j^{(l)}||_2^2 = \lambda \mathbf{w}$$

Early stopping focuses on halting the learning training process before the MLP is able to overfit the training data. In a data rich environment, the data is able to be split into training data $\mathcal{T}_r$ which is split further into batches $\mathcal{T}^{(1)}, \mathcal{T}^{(2)}, ..., \mathcal{T}^{(B)}$, a validation set $\mathcal{T}_v$ and a test set $\mathcal{T}_e$. Early stopping works by monitoring the validation error given by

$$\overline{\mathrm{err}}_{val}^{(t)} = \frac{1}{|\mathcal{T}_v|} \mathrm{R}(\hat{\boldsymbol{\Theta}}^{(t)}; \mathcal{T}_v)$$

where $\hat{\boldsymbol{\Theta}}^{(t)}$ are the estimated parameters after the t^{th} training epoch. An epoch is one run through all the training batches. With the patience parameter η and minimum improvement required δ, early stopping checks

the validation error for $t = 1 + k \cdot \eta$ for $k = 1, 2, ...$ and stops if the following criterion is met

$$\overline{\text{err}}_{val}^{(1+k\cdot\eta)} + \delta > \overline{\text{err}}_{val}^{(1+(k-1)\cdot\eta)}$$

Dropout is based on the principle of bagging in which an ensemble of models is used to improve the generalization error of the model. While traditional bagging makes use of a number of separate models, droupout makes use of an ensemble of subnetworks within the larger neural network. This is achieved by setting the output from some of the hidden nodes to zero. For a typical large MLP architecture given by $\boldsymbol{\Theta}$, the sub-architectures are denoted $\boldsymbol{\Theta}_m : m = 1, 2, ..., M$ where $M = 2^{d_1 \times d_2 \times ... \times d_L}$. The ensemble results in the model

$$\mathbf{f}^{(M)}(.; \{\Theta_m\}_{m=1}^{M}) = \sum_{m=1}^{M} p_m \mathbf{f}(.; \Theta_m)$$

where $p_m \geq 0$, $\sum_{m=1}^{M} p_m = 1$ and $\mathbf{f}(.; \Theta_m)$ is an MLP. The ensemble model is a regularized model since $\mathbf{f}^{(M)}(.; \{\Theta_m\}_{m=1}^{M})$ is less complex and thus less prone to overfitting than $\mathbf{f}(.; \Theta)$. However, it is infeasible to estimate all $\widehat{\boldsymbol{\Theta}}_m : m = 1, 2, ..., M$. Dropout approximates this model by including an addition step in the hidden layers in an MLP. The output from the activation function as follows

$$\begin{aligned} \boldsymbol{\sigma}(\mathbf{z}_l) &= \mathbf{h}_l \\ \mathbf{h}_l &= \frac{1}{1-p} \mathbf{h}_l \odot \mathbf{I}^{(l)}. \end{aligned} \tag{6.2}$$

The output form a single hidden layer is denoted $\mathbf{h}_l = f(\mathbf{x}; \Theta, \mathcal{I})$ where $\mathcal{I}$ is iid Bernoulli : $\{\mathbf{I}^{(1)}, \mathbf{I}^{(2)}, ..., \mathbf{I}^{(L)}\}$ with probability $1 - p$. Thus a dropout MLP attempts to model

$$\begin{aligned} \mathbf{f}^{(\text{dropout})}(\mathbf{x}; \Theta) &= \mathbb{E}_{\mathcal{I}}\Big[f(\mathbf{x}; \Theta, \mathcal{I})\Big] \\ &= \frac{1}{M} \sum_{m=1}^{M} f(\mathbf{x}; \Theta, \mathcal{I}^{(m)}). \end{aligned} \tag{6.3}$$

However, it is intractable to estimate new test points using all the M architectures. Thus a further approximation is made when implementing dropout. The approximation is made to equation 6.3 by taking

$$\approx f(\mathbf{x}_i; \mathbb{E}(\mathcal{I}), \widehat{\Theta})$$

which results in equation 6.2.

Batch normalization is implemented to overcome a problem associated with training deep MLPs. During backpropagation, the parameters are updated simultaneously, however, the update is done assuming that the other parameters remain constant. This can result in unforeseen undesirable outcomes. Batch normalization attempts to overcome these issues by including additional operations in a hidden layer. The batched output from the l^{th} hidden layer, $\mathbf{h}_l$, is fed into an additional normalization layer where, for $j = 1, 2, ..., d_l$, the individual observations are used to compute $\overline{m}_j^{(l)} = \frac{1}{n} \sum_{i=1}^{n} h_{ij}^{(l)}$
$\mathrm{s}_j^{(l)} = \sqrt{\frac{1}{n-1} \sum_{i=1}^{n} (h_{ij}^{(l)} - \overline{m}_j)^2}$ then the individual observations are updated by $\mathrm{h}_{ij}^{(l)} \leftarrow \frac{h_{ij}^{(l)} - \overline{m}_j^{(l)}}{s_j^{(l)}}$. During testing times $\overline{m}_j^{(l)}$ and $s_j^{(l)}$ are replaced by running averages from the testing observations.

6.2 Question 2

Convolutional neural networks (CNNs) are a special kind of neural network architecture that is designed to process data that have a grid-like structure. Examples of grid-like structured data include time series data which is a 1D grid structure, grey-scale images which have a 2D structure, colour images that have a 3D structure due to the Red-Green-Blue layers of pixels, then there are other forms of data that have higher dimensional structures including video, video with audio and other multimodal data forms.

CNNs get their name from the use of convolution operators in place of general matrix multiplication between layers. The convolutional operator can be seen as a sort of weighted average function over a sequence. For a continuous time convolution, for a given time-dependent function $x(t)$, which could be the position of an object over time, and a weight function $w(a)$, the smoothed function calculated using a convolution is given by

$$s(t) = \int x(a)w(t-a)da$$

which is typically denoted by

$$s(t) = (x * w)(t).$$

In the CNN terminology, $x(\cdot)$ is called the input, $w(\cdot)$ is the kernel and $s(\cdot)$ is the feature map. In discrete time a convolution operation is given by the following

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a).$$

This is usually applied only over a finite set of elements in the input array. Convolutions can be applied to multiple dimensions. For a 2D image input I, a 2D kernel K, the feature map is determined through

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n).$$

which is also known as the cross correlation and is used in practice rather than the regular convolution. Figure 6.1 gives an example of a 2D input and a 2D kernel with a 2D output. Here the matrix multiplication between the input and the kernel result in a smaller image made up of linear combinations of the sections of the input grid and the kernel. Since the kernel is smaller than the input, known as sparse weights, the dimension of the output is reduced. Furthermore, by using the same kernel over the entire image, CNNs have far fewer parameters than fully connected dense MLPs. This is known as parameter sharing and allows for sparser connections between inputs and outputs as seen in figure 6.3. Here the top image is a CNN where each input is only connected to two or three outputs. Additionally, the strength of the connection is the same of each connection represented by a direction (vertical lines represent the same weight, the diagonally right the same, and the diagonally left the same). Compared to the below MLP where each line represents a different weight and each output is connected to each input. The CNN thus requires far fewer computations to train allowing for more complex data to me used as inputs.

Furthermore, CNNs have the property of equivariance to translation which means that any shifting of inputs will result in the same shifting in the output. This means than for example an image that is shifted to the right compared to another image will not interfere with the output. The use of convolutions are used to extract

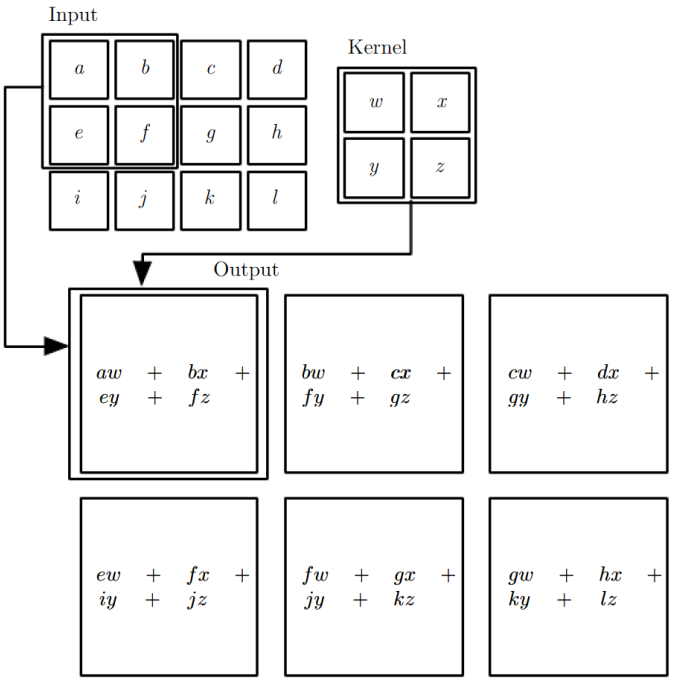

Figure 6.2: Example of 2D convolution (Source: Goodfellow et al. (2016))

meaningful feature from the input. For example, a CNN that is trained on 2D facial images might train a kernel to detect eyebrows and another to detect a nose. Thus the kernels extract from the input image whether or not these features (eyebrows, noses etc.) are present. Through the use of a number of convolutions in different layers, a feature map is created that can then be flattened into a vector which is input into a dense fully connected MLP.

The architecture of a CNN generally consists of three steps, the convolution layer, a non-linear activation functions layer known as the detection layer and a pooling layer. The convolution layer is described above. The detection layer is similar to that of an MLP's non-linear layer where ReLu, tanh or sigmoid activation functions are used. This layer is known as the detection layer since is determines the strength of the existence of specific features that the kernel is trying to uncover. Finally a pooling layer is included. A pooling layer replaces the input from the detection layer with a summary statistic of a region. Examples include max pooling which return the maximum value of the net for over a rectangular neighbourhood, average pooling which returns the average of the net over a rectangular neighbourhood or the L^2 norm of a rectangular neighbourhood. Pooling helps make the CNN invariant to rotations. This is important when the presence of a feature is more important than determining the position of the feature. Another important aspect of CNNs is padding. Padding refers to appending zero entries to the boundaries of the input to increase the dimension of the data thus when a lower dimensional kernel is applied in a convolution to the input the output is of the same dimension as the original input. This is important when a number of convolution layers are to be applied. Without padding each CNN layer with a smaller kernel reduces the dimensions of the data.

The convolution layer for a CNN layer for a grey-scale image is as follows. For an input image given by

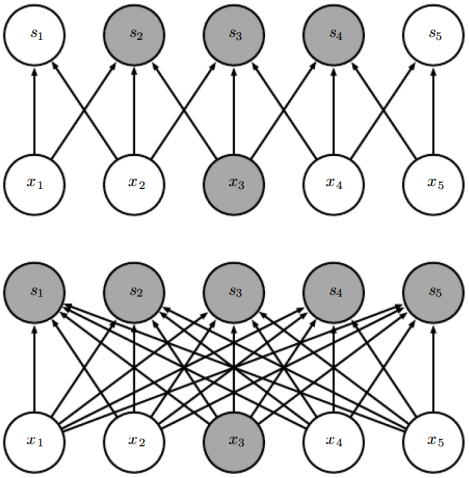

Figure 6.3: Sparse connections and parameter sharing in CNN (Source: Goodfellow et al. (2016))

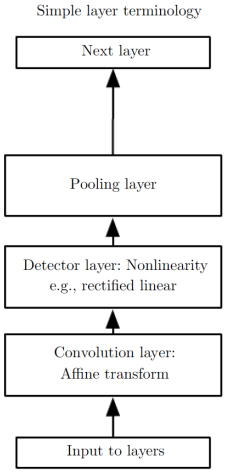

Figure 6.4: Typical CNN layer architecture (Source: Goodfellow et al. (2016))

$\mathbf{V} \in \Re^{L \times M}$, a kernel given by $\mathbf{K} \in \Re^{S \times T}$ where $S \leq L$ and $T \leq M$, and output given by $Z \in \Re^{\tilde{L} \times \tilde{M}}$, a single output array element is given by

$$\mathbf{Z}_{lj} = \sum_{s=1}^{S} \sum_{t=1}^{T} \mathbf{K}_{st} \mathbf{V}_{l+s-1, j+t-1}.$$

The output from the convolution layer is fed into a non-linear activation which operates element wise as follows

$$\mathbf{H}_{lj} = \sigma(\mathbf{Z}_{lj})$$

where σ can be the ReLu function, $\tanh(\cdot)$, or the sigmoid function. The output from this is fed into a pooling layer. For illustration, a max pooling layer operates as follows

$$\mathbf{P}_{ij} = \max(\mathbf{H}_{l,j}, \mathbf{H}_{l+1,j}, \mathbf{H}_{l,j+1}, \mathbf{H}_{l+1,j+1})$$

where $\mathbf{P} \in \Re^{\tilde{L}-1 \times \tilde{M}-1}$.

Similarly, the convolutional layer for a CNN layer for a colour image is as follows. For given RBG input image denoted by $\mathbf{V} \in \Re^{C_{in} \times L \times M}$ where C_{in} is the input channel and $\mathbf{V}_{c\cdot\cdot}$ is the c^{th} input channel, and an

output given by $\mathbf{Z} \in \Re^{C_{out} \times \tilde{L} \times \tilde{M}}$ where C_{out} is the output channel and $\mathbf{Z}_{j\cdot\cdot}$ is the j^{th} output channel, the convolutional operation is given by

$$\mathbf{Z} = \mathbf{\Phi}(\mathbf{K}, \mathbf{V}) : \mathbf{Z}_{j\cdot\cdot} = \mathbf{b} + \sum_{c=1}^{C_{in}} \mathbf{K}_{jc\cdot\cdot} \circledast \mathbf{V}_{c\cdot\cdot} : j = 1, 2, ..., C_{out}$$

where $\mathbf{K} \in \Re^{C_{out} \times C_{in} \times S \times T}$ where $S \leq L$ and $T \leq M$. The non-linear activation layer and pooling layers follow as above.

Self driving cars are a novel technology that relies fundamentally on deep learning. In this application deep learning is used for multiple different uses with CNNs specifically for image recognition. In a recent interview (Fridman & Musk 2021), Telsa CEO Elon Musk described the use of cameras and microphones to create a multimodal stream of input for their deeplearning architectures. The details of the types of neural networks were not described in great detail but it is speculated that CNNs are used to extract features about the cars surroundings before being fed into MLPs or deep reinforcement learning networks to make decisions. The types of features that are extracted are other cars, pedestrians, road signs, cyclists and other more uncommon objects found on roads. This problem is a combination of a classification problem to determine the type of object as well as regression problem in determining the direction and velocity of the object.

Recurrent neural networks (RNNs) are a special kind of neural network architecture that is designed to process sequential data. The data processed is generally in the form $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, ..., \mathbf{x}^{(T)}$. RNNs benefit from the ability to share parameters across inputs. This allows for sequences of different lengths to be fed into the network with the same features to be extracted along the sequence. This compares to an MLP where each point of the sequence has specific parameters that are trained to extract certain features. Thus the MLP will not be able to adequately extract features from sequences of different lengths. While the CNN architecture allows for time series sequences to be trained as 1D arrays, these models are shallow due to the size of the kernels. CNNs thus create outputs from a neighbourhood of inputs. RNNs differ in that previous outputs are used in conjunction with the current input to create a new output. RNNs can be seen as a form of encoder model that take in input sequences and output "cleaned" output sequences that are then fed into MLPs for prediction.

The Elman RNN, given in figure 6.5, is an example of a simple RNN architecture. The objective is to encode the input $\mathbf{x}$ into the output $\mathbf{h}$. An Elman RNN takes in an input $\mathbf{x} \in \Re^p$ and the previous output $\mathbf{h}^- \in \Re^q$ and creates a new output $\mathbf{h}^+ \in \Re^q$ form the following operation

$$\mathbf{h}^+ = \boldsymbol{\sigma}(\mathbf{b} + \mathbf{V}\mathbf{h}^- + \mathbf{W}\mathbf{x})$$

where $\mathbf{b} \in \Re^p$ is the bias, $\mathbf{V} \in \Re^{q \times q}$ are the recurrent weights, and $\mathbf{W} \in \Re^{q \times p}$ are the input weights. The activation function $\boldsymbol{\sigma}$ is also given and can be a ReLu or tanh function generally.

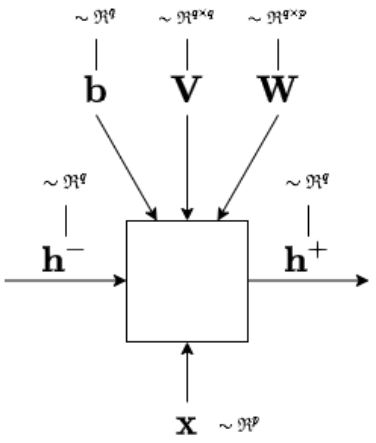

Figure 6.5: Single Elman RNN Unit

Elman RNN units can be connected such that each input in the sequence is fed into a separate RNN unit. This is a multilayer Elman RNN and is given in figure 6.6. The forwards pass of a multilayer Elman RNN with the input $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, ..., \mathbf{x}^{(T)}$ is as follows:

for $t = 1, 2, ..., T$:

$$\begin{aligned} \mathbf{z}^{(t)} &= \mathbf{b} + \mathbf{V}\mathbf{h}^{(t-1)} + \mathbf{W}\mathbf{x}^{(t)} \\ \mathbf{h}^{(t)} &= \boldsymbol{\sigma}(\mathbf{z}^{(t)}) \end{aligned} \tag{6.4}$$

resulting in output $\mathbf{h}^{(1)}, \mathbf{h}^{(2)}, ..., \mathbf{h}^{(T)}$ which can be fed into an MLP.

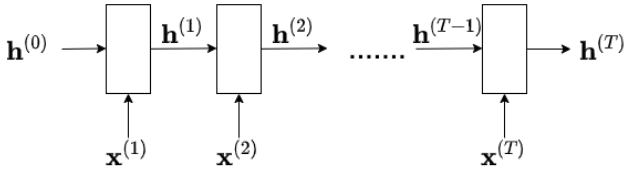

Figure 6.6: Multilayer Elman RNN

Another RNN architecture that builds of the multilayer Elman RNN is the Bidirectional RNN. These are used for speech recognition and handwriting recognition where the inputs following the current input are important for modelling the current input. The architecture is given in figure 6.7. The forwards pass begins with equation 6.4 and includes the following:

$$\begin{aligned} \overleftarrow{\mathbf{z}}^{(T)} &= \overleftarrow{\mathbf{b}} + \overleftarrow{\mathbf{V}}\overleftarrow{\mathbf{h}}^{(T)} + \overleftarrow{\mathbf{W}}\mathbf{x}^{(T)} \\ \overleftarrow{\mathbf{h}}^{(T-1)} &= \boldsymbol{\sigma}(\overleftarrow{\mathbf{z}}^{(T)} \end{aligned}$$

for $t = T-1, ..., 1$

$$\begin{aligned} \overleftarrow{\mathbf{z}}^{(t)} &= \overleftarrow{\mathbf{b}} + \overleftarrow{\mathbf{V}}\overleftarrow{\mathbf{h}}^{(t+1)} + \overleftarrow{\mathbf{W}}\mathbf{x}^{(t)}) \\ \overleftarrow{\mathbf{h}}^{(t)} &= \boldsymbol{\sigma}(\overleftarrow{\mathbf{z}}^{(t)}) \end{aligned}$$

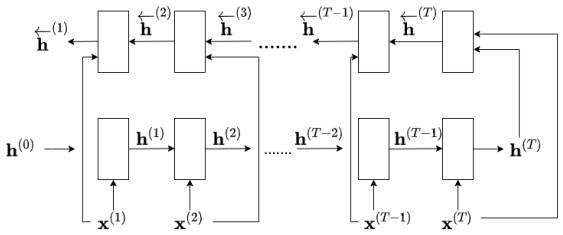

Figure 6.7: Bidireactional RNN

A problem with RNNs is that of long-term dependence. The problem is that basic gradients propagated over many time-steps either vanish or explode. This stems from the exponentially smaller weights associated with observations that took place much earlier in the sequence. Thus long-term interactions are not easily computed as short-term interactions. A number of methods have been proposed to overcome this issue including leaky units, echo state networks and liquid state machines but most prominently RNNs with Gated cells including Gated Recurrent Units (GRUs) and Long-Short Term Memory (LSTM) RNNs. Gated RNNs work by accumulating information over the length of the sequence and then deciding through the training process whether or not to incorporate the information in the current cell. This is useful is the sequence is made up of many subsequences.

A simple gated cell is as follows

$$\mathbf{h}^{+} = \boldsymbol{\sigma}(\mathbf{b} + \mathbf{V}\mathbf{h}^{-} + \mathbf{W}\mathbf{x})$$
$$\mathbf{h}^{+} \leftarrow (1-s) \times \mathbf{h}^{+} + s \times \mathbf{h}^{-} : s \in [0,1].$$

GRU cells include a reset gate and update gate given by

$$\text{reset gate: } \mathbf{g}^{(r)} = \boldsymbol{\sigma}_{sig}(\mathbf{b}^{(r)} + \mathbf{V}^{(r)}\mathbf{h}^{-} + \mathbf{W}^{(r)}\mathbf{x})$$
$$\text{update gate: } \mathbf{g}^{(u)} = \boldsymbol{\sigma}_{sig}(\mathbf{b}^{(u)} + \mathbf{V}^{(u)}\mathbf{h}^{-} + \mathbf{W}^{(u)}\mathbf{x}).$$

The GRU uses these gates in the following steps:

$$\text{reset memory: } \tilde{\mathbf{h}} = \mathbf{g}^{(r)} \odot (\tilde{\mathbf{b}} + \tilde{\mathbf{V}}\mathbf{h}^{-})$$
$$\text{compute candidate output: } \mathbf{h}^{+} = \boldsymbol{\sigma}_{tanh}(\mathbf{b}^{+} + \mathbf{W}^{+}\mathbf{x} + \tilde{\mathbf{h}})$$
$$\text{damping: } \mathbf{h}^{+} \leftarrow (1 - \mathbf{g}^{(u)}) \odot \mathbf{h}^{+} + \mathbf{g}^{(u)} \odot \mathbf{h}^{-}.$$

The LSTM RNN make use of gated internal self-loops that keep track of the cell state. This allows for different subsequences to be modelled and used to create the output. Following figure 6.8, a single LSTM cell as the following steps:

Step a: compute forget and input gates

$$\text{forget gate: } \mathbf{g}^{(f)} = \boldsymbol{\sigma}_{sig}(\mathbf{b}^{(f)} + \mathbf{V}^{(f)}\mathbf{h}^{-} + \mathbf{W}^{(f)}\mathbf{x})$$
$$\text{input gate: } \mathbf{g}^{(i)} = \boldsymbol{\sigma}_{sig}(\mathbf{b}^{(i)} + \mathbf{V}^{(i)}\mathbf{h}^{-} + \mathbf{W}^{(i)}\mathbf{x})$$

Step b: compute the Elman update

$$\mathbf{h}^{+} = \boldsymbol{\sigma}_{tanh}(\mathbf{b} + \mathbf{V}\mathbf{h}^{-}\mathbf{W}\mathbf{x})$$

Step c: compute output gate

$$\text{output gate: } \mathbf{g}^{(o)} = \boldsymbol{\sigma}_{sig}(\mathbf{b}^{(o)} + \mathbf{V}^{(o)}\mathbf{h}^{-} + \mathbf{W}^{(o)}\mathbf{x})$$

Step d: update cell state

$$\mathbf{c}^{+} = \mathbf{g}^{(f)} \odot \mathbf{c}- + \mathbf{g}^{(i)} \odot \mathbf{h}^{+}$$

Step d: update memory

$$\mathbf{h}^{+} \leftarrow \mathbf{g}^{(o)} \odot \boldsymbol{\sigma}_{tanh}(\mathbf{c}^{+})$$

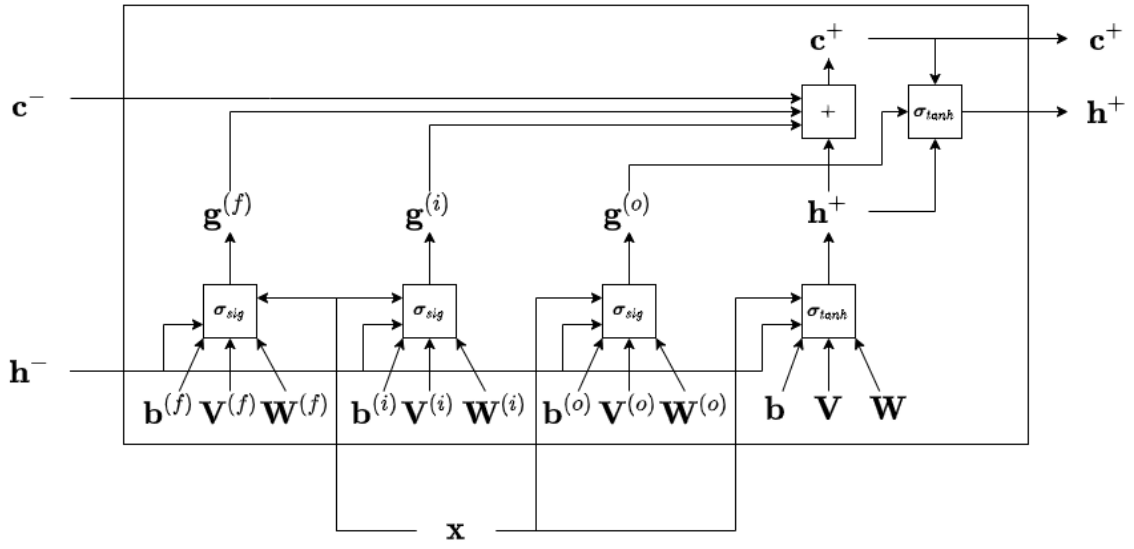

Figure 6.8: Long-Short Term Memory RNN Cell

LSTM models are created in a similar manner to multilayer Elman RNNs where a number of cells are connected in a sequence. The outputs from the LSTM cells can then be fed into a MLP. Additionally, multiple LSTM models can be stacked on top of each other such that the output from the first LSTM model become the inputs for the next LSTM model. This process of stacking applies to all RNNs.

RNNs are designed with sequential data in mind. Of the many fields that generate sequential data, natural languages processing (NLP) is a particularly complex but meaningful field. NLP is concerned with generating sentences, classifying texts, determining the sentiment of texts or translating texts. NLP data consists of sequences of words that are tokenized into numerical inputs. The sequences can be fed into a RNN which acts as an encoder. This is then fed into an MLP to classify the text, to determine the sentiment or to predict the next word. For translations, the encoded output from the first RNN model is fed into another RNN model that acts as a decoder and the output is a translated text. NLP tasks have greatly benefited from the introduction of the LSTM which is able to determine the context of the text and improves the performance. Recently however, the state of the art NLP models are built using the transformer neural network architecture. Transformer models use an architecture called self-attention where the current input can refer back to previous inputs and states. These models have been trained on gigantic data sets and produced astounding results.

Bibliography

Fanaee-T, H. & Gama, J. (2013), 'Event labeling combining ensemble detectors and background knowledge', *Progress in Artificial Intelligence* pp. 1–15.
URL: *http://dx.doi.org/10.1007/s13748-013-0040-3*

Fridman, L. & Musk, E. (2021), *SpaceX and Tesla Autopilot, Robotics, and AI*, Lex Fridman Podcasts.
URL: *https://www.youtube.com/watch?v=DxREm3s1scA*

Goodfellow, I., Bengio, Y. & Courville, A. (2016), *Deep Learnign*, MIT Press.
URL: *http://www.deeplearningbook.org*

Hastie, T., Tibshirani, R. & Friedman, J. (2017), *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, 2 edn, Springer New York.
URL: *http://doi.wiley.com/10.1111/j.1751-5823.2009.00095_18.x*

Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q. & Liu, T. Y. (2017), 'LightGBM: A highly efficient gradient boosting decision tree', *Advances in Neural Information Processing Systems* **2017-Decem**(Nips), 3147–3155.

Rashmi, K. V. & Gilad-Bachrach, R. (2015), 'DART: Dropouts meet multiple additive regression trees', *Journal of Machine Learning Research* **38**, 489–497.

Appendices

A.1 Assignment 1.1 code

```
# Load Packages

library(tidyverse)
pacman::p_load(cowplot, ggplot2, kableExtra, rsample, glue, lubridate)

# Read in Data


data <- read.csv("data/household_power_consumption.csv", header = TRUE, sep = ';')

# Wrangle Data

data_power <- data %>% as_tibble() %>%
  mutate(date = as.character(as.Date(Date, tryFormats =c("%d/%m/%Y")))) %>%
  mutate(date = paste(date, Time)) %>%
  mutate(date_time = as_datetime(date)) %>%
  rename(power = Global_active_power) %>%
  select(date_time, power) %>%
  mutate(power = as.numeric(power))

data_power <- data_power %>% mutate(date_time = floor_date(date_time, unit="hour")) %>%
  group_by(date_time) %>%
  summarise(power = mean(power, na.rm=TRUE))

data_power <-
  data_power %>% mutate(year = year(date_time), month = month(date_time)) %>%
  group_by(year, month) %>%
  mutate(avg = mean(power, na.rm=T)) %>%
  ungroup() %>%
  mutate(power = coalesce(power, avg)) %>% select(-c(avg, year, month))

data_power %>%
  mutate(Date = as.Date(date_time)) %>%
  ggplot() +
```

```
  geom_line(aes(x = Date, y = power), alpha = 0.8, size = 0.6) +
  labs(title = "Power usage over full 47 Months", x = "", y = "Power Usage") +
  theme_minimal_hgrid() +
  theme(axis.text.x = element_text(angle = 90, hjust = 1, size = 7),
        plot.title = ggplot2::element_text(size = 10),
        axis.title.y = element_text(size = 9))

# Create train, validation and test sets
data_train <- data_power %>% rename(Date = date_time) %>% filter(Date < as.Date("2009-01-01"))
    %>% tail(-48) %>% head(-7)

data_val <- data_power %>% rename(Date = date_time) %>% filter(Date >= as.Date("2009-01-01") &
    Date < as.Date("2010-01-01")) %>% tail(-48)

data_test <- data_power %>% rename(Date = date_time) %>% filter(Date >= as.Date("2010-01-01"))
    %>% tail(-58)


# Load python virtual environment to run Keras and Tensorflow

pacman::p_load(reticulate)

reticulate::use_virtualenv("/Users/jonathanrossouw/Desktop/SLT/Assignment 1_ Fundamentals of
    Statistical Learning Theory./Assignment 1.1/.venv", require = TRUE)
reticulate::py_config()

# Load Tensorflow and Keras
library(tensorflow)
library(keras)

#### Data Wrangling Function

data_wrangling_func <- function(data, type, final_date){
    # Filter dates and replace NAs with 0
    data <- data %>% select(Date, power) %>% mutate(power = coalesce(power, 0)) %>% filter(
        Date < as.Date(final_date))
    if(type == "lstm"){
        # Rescale Min Max
        min_max <- c(min(data$power), max(data$power))
        data_lstm <- data %>%
            mutate(power = (power - min_max[1])/(min_max[2] -
                                                 min_max[1]))
        return(list(data = data_lstm, min_max = min_max))
    }
    if(type=="svr"){
        return(data = data)
    }
    if(type=="lgb"){
        return(data = data)
```

```
    }
    else break
}

### Wrangle and Plot LSTM Data
# Wrangle data

data_lstm <- data_wrangling_func(data = data_power %>% rename(Date = date_time),
                                 type = "lstm",
                                 final_date = "2009-01-01")
data_lstm_val <- data_wrangling_func(data = data_power %>% rename(Date = date_time) %>% filter
    (Date >= as.Date("2009-01-01")),
                                     type = "lstm",
                                     final_date = "2010-01-01")
# Set min and max for later transformations
min_max <- data_lstm$min_max
data_lstm_train <- data_lstm$data

val_min_max <- data_lstm_val$min_max
data_lstm_val <- data_lstm_val$data
# Plot Rescaled LSTM data
data_lstm_train %>%
    ggplot() +
    geom_line(aes(x = Date, y = power), alpha = 0.8, size = 0.6) +
    labs(title = "Power Usage Training after Min-Max Rescaling", x = "", y = "Power Usage") +
    theme_minimal_hgrid() +
    theme(axis.text.x = element_text(angle = 90, hjust = 1, size = 7),
          plot.title = ggplot2::element_text(size = 10),
          axis.title.y = element_text(size = 7))

###### Simple LSTM Array Function

data_array_func <- function(data, val, initial, assess, skip){
    # Select Appropriate Data
    data_train <- data %>%
        dplyr::select(Date, power) #%>% head((nrow(data) - 14))
    # Creating Rolling Data Splits
    data_rol <- rolling_origin(data_train,
                               initial = initial,
                               assess = assess,
                               skip = skip,
                               cumulative = FALSE)
    # Create Training Predictor Array
    x_train_array <- lapply(data_rol$splits, FUN = function(X){analysis(X)$power})
    x_train_array <- do.call("rbind", x_train_array) %>%
        array(., dim = c(length(x_train_array), initial, 1))
    # Create Training Target Array
    y_train_array <- lapply(data_rol$splits, FUN = function(X){testing(X)$power})
    y_train_array <- do.call("rbind", y_train_array) %>%
```

```
127         array(., dim = c(length(y_train_array), assess, 1))
128
129
130     # Select Appropriate Validation Data
131     data_val <- val %>%
132         dplyr::select(Date, power) #%>% head((nrow(data) - 14))
133     # Creating Rolling Data Splits
134     data_rol_val <- rolling_origin(data_val,
135                                    initial = initial,
136                                    assess = assess,
137                                    skip = skip,
138                                    cumulative = FALSE)
139     # Create Training Predictor Array
140     x_val_array <- lapply(data_rol_val$splits, FUN = function(X){analysis(X)$power})
141     x_val_array <- do.call("rbind", x_val_array) %>%
142         array(., dim = c(length(x_val_array), initial, 1))
143     # Create Training Target Array
144     y_val_array <- lapply(data_rol_val$splits, FUN = function(X){testing(X)$power})
145     y_val_array <- do.call("rbind", y_val_array) %>%
146         array(., dim = c(length(y_val_array), assess, 1))
147
148     return(list(x_train_array = x_train_array,
149                 y_train_array = y_train_array,
150                 x_val_array = x_val_array,
151                 y_val_array = y_val_array))
152 }
153
154 ### Create LSTM Data Arrays
155 data_lstm_lists <- data_array_func(data = data_lstm_train,
156                                    val = data_lstm_val,
157                                    initial = 2*24, assess = 12, skip = 11)
158
159 ##### Set hyperparameter grid
160 lstm_hyp <- expand.grid(loss = c("mse"),
161                         optimizer = c("adam"),
162                         epochs = c(20, 50, 100, 150),
163                         lstm_layers = c(1,2),
164                         lstm_units = c(20, 50),
165                         return_sequences = c(TRUE, FALSE),
166                         dropout_rate = c(0, 0.1)) %>%
167     filter(!(return_sequences == TRUE & lstm_layers == 1)) %>%
168     split(., seq(nrow(.)))
169
170
171 # Hyperparameter tuning wrapper
172
173 hyp_tune_func <- function(data_lists = data_lists,
174                           data_actual = data_val,
175                           hyperparams = hyp,
```

```
                            fit_func,
                            type,
                            min_max = val_min_max){
    # Run iteratively through hyperparameter grid
    if(type == "lstm")
    {
        hyp_fit <- map(hyperparams, fit_func, data_lists, min_max)
    }
    else{hyp_fit <- map(hyperparams, fit_func, data_lists)}

    hyp_res <- map(hyp_fit, ~data_actual %>% mutate(pred = .x) %>%
                        summarise(mean((pred - power)^2, na.rm = TRUE)) %>% .[[1]])
    # Store Results
    res <- map2_df(hyperparams, hyp_res, ~ .x %>%
                        as_tibble() %>%
                        mutate(mse = .y))
    res
}


##### LSTM Fit Function

lstm_fit_func <- function(hyperparams,
                            data_lists,
                            min_max){
    # Set seed for reproducibility
    set_random_seed(seed = 43675)
    # Create Keras Sequential Model
    lstm_model <- keras_model_sequential()
    # Add Layers
    lstm_model %>%
        layer_lstm(units = hyperparams$lstm_units, # size of the layer
                   input_shape = c(48, 1),
                   # batch size, timesteps, features
                   return_sequences = hyperparams$return_sequences,
                   stateful = FALSE) %>%
        layer_dropout(rate = hyperparams$dropout_rate)
    if(hyperparams$lstm_layers == 2 & hyperparams$return_sequences == TRUE)
    {
        lstm_model %>%
            layer_lstm(units = hyperparams$lstm_units, # size of the layer
                       input_shape = c(48, 1),
                       # batch size, timesteps, features
                       return_sequences = FALSE,
                       stateful = FALSE) %>%
            layer_dropout(rate = hyperparams$dropout_rate)
    }
    lstm_model %>%
        layer_dense(units = 12)
```

```
    # Set model parameters
    lstm_model %>%
        compile(loss = hyperparams$loss, optimizer = hyperparams$optimizer, metrics =
            hyperparams$loss)
    # Fit Model
    lstm_model %>%
        fit(
            x = data_lists$x_train_array,
            y = data_lists$y_train_array,
            epochs = hyperparams$epochs,
            verbose = 1,
            shuffle = FALSE)
    # Predict training data
    lstm_train_forecast <- lstm_model %>%
        predict(data_lists$x_train_array) %>% t(.) %>% c(.)
    lstm_train_forecast <- (lstm_train_forecast * (min_max$train[2] - min_max$train[1]) + min_
        max$train[1])
    # Predict power
    lstm_val_forecast <- lstm_model %>%
        predict(data_lists$x_val_array) %>% t(.) %>% c(.)
    lstm_val_forecast <- (lstm_val_forecast * (min_max$val[2] - min_max$val[1]) + min_max$val
        [1])
    # Store Results
    return(list(train = lstm_train_forecast, val = lstm_val_forecast))
}

# Perform hyperparameter tuning for LSTM

lstm_fit <- hyp_tune_func(data_lists = data_lstm_lists,
                          data_actual = data_val,
                          hyperparams = lstm_hyp,
                          fit_func = lstm_fit_func,
                          min_max = val_min_max,
                          type = "lstm")

# Best hyperparameter

best_lstm_tune <- lstm_fit %>%
    filter(mse == min(mse))

# Performance and Plotting Function

perf_plot <- function(data_lists = data_lists,
                      data_train = data_train,
                      data_val = data_val,
                      hyperparams = hyp,
                      fit_func,
                      type,
                      min_max = val_min_max,
```

```
                        train_title,
                        val_title
){
    if(type == "lstm")
    {
        hyp_fit <- fit_func(hyperparams, data_lists, min_max)
    }
    else{hyp_fit <- fit_func(hyperparams, data_lists)}

    train_err <- data_train %>% mutate(pred = hyp_fit$train) %>%
        summarise(mean((pred - power)^2, na.rm = TRUE)) %>% .[[1]]

    hyp_res <- data_val %>% mutate(pred = hyp_fit$val) %>%
        summarise(mean((pred - power)^2, na.rm = TRUE)) %>% .[[1]]
    # Store Results

    train_res <- hyperparams %>%
        as_tibble() %>%
        mutate(mse = train_err)

    res <- hyperparams %>%
        as_tibble() %>%
        mutate(mse = hyp_res)

    train_forecast <- data_train %>% mutate(pred = hyp_fit$train)

    train_plot <- train_forecast %>% ggplot() +
        geom_line(aes(x = Date, y = power), alpha = 0.8, size = 0.6) +

        geom_line(aes(x = Date, y = pred), color = "red", size = 0.8, alpha = 0.8) +
        labs(title = train_title, x = "", y = "Power Usage") +
        theme_minimal_hgrid() +
        theme(axis.text.x = element_text(angle = 90, hjust = 1, size = 7),
              plot.title = ggplot2::element_text(size = 10),
              axis.title.y = element_text(size = 7))

    val_forecast <- data_val %>% mutate(pred = hyp_fit$val)

    val_plot <- val_forecast %>% ggplot() +
        geom_line(aes(x = Date, y = power), alpha = 0.8, size = 0.6) +

        geom_line(aes(x = Date, y = pred), color = "red", size = 0.8, alpha = 0.8) +
        labs(title = val_title, x = "", y = "Power Usage") +
        theme_minimal_hgrid() +
        theme(axis.text.x = element_text(angle = 90, hjust = 1, size = 7),
              plot.title = ggplot2::element_text(size = 10),
              axis.title.y = element_text(size = 7))

```

```
319     return(list(train_plot = train_plot,val_plot = val_plot, performance = res, training_error
            = train_res))
320 }
321
322 # Determine performance for best hyperparameters
323
324 lstm_perf <- perf_plot(data_lists = data_lstm_lists,
325                        data_val = data_val,
326                        data_train = data_train,
327                        hyperparams = best_lstm_tune, #best_lstm_tune,
328                        fit_func = lstm_fit_func,
329                        min_max = list(train = min_max,
330                                       val = val_min_max),
331                        type = "lstm",
332                        train_title = "LSTM Train Plot",
333                        val_title = "LSTM Validation Plot")
334
335 # Plot forecasts
336
337 lstm_perf$train_plot
338 lstm_perf$val_plot
339 lstm_perf$performance
340 lstm_perf$training_error
341
342 # SVR wrangle data
343
344 library(liquidSVM)
345
346 data_svr_train <- data_wrangling_func(data = data_power %>% rename(Date = date_time),
347                                       type = "svr",
348                                       final_date = "2009-01-01")
349
350 data_svr_val <- data_wrangling_func(data = data_power %>% rename(Date = date_time) %>% filter(
        Date >= as.Date("2009-01-01")),
351                                     type = "svr",
352                                     final_date = "2010-01-01")
353
354 data_svr_array <- data_array_func(data = data_svr_train,
355                                   val = data_svr_val,
356                                   initial = 2*24,
357                                   assess = 12,
358                                   skip = 11)
359
360 #### Function for SVR and GBM data wrangling
361
362
363 data_svr_lgb_wrangle <- function(data_array = data_array, type){
364
365     if(type == "svr")
```

```
366     {
367         # Create training list for each forcast ahead
368         x_train <- data_array$x_train_array[,,1] %>% as_tibble()
369         y_train <- data_array$y_train_array[,,1] %>% split(., rep(1:ncol(.), each = nrow(.)))
370         train <- map(y_train, ~cbind(Y=.x, x_train))
371 
372         # Create validation list for each forecast ahead
373 
374         x_val <- data_array$x_val_array[,,1] %>% as_tibble()
375         y_val <- data_array$y_val_array[,,1] %>% split(., rep(1:ncol(.), each = nrow(.)))
376         validation <- map(y_val, ~cbind(Y=.x, x_val))
377 
378         return(list(train_list = train, val_list = validation))
379     }
380     if(type == "lightgbm"){
381         # Create training list for each forcast ahead
382         x_train <- data_array$x_train_array[,,1]%>% as_tibble()
383         y_train <- data_array$y_train_array[,,1] %>% split(., rep(1:ncol(.), each = nrow(.)))
384         train <- map(y_train, ~cbind(Y=.x, x_train))
385 
386         train_pred <- map(train, ~.x[,-1] %>% as.matrix())
387         train_lgb <- map(train, ~.x[,-1] %>% as.matrix() %>% Matrix::Matrix(., sparse = TRUE)
                %>% lgb.Dataset(data = ., label = .x[,1]))
388 
389         # Create validation list for each forecast ahead
390 
391         x_val <- data_array$x_val_array[,,1]
392         y_val <- data_array$y_val_array[,,1] %>% split(., rep(1:ncol(.), each = nrow(.)))
393         validation <- map(y_val, ~cbind(Y=.x, x_val))
394         validation <- map(validation, ~.x[,-1] %>% as.matrix())
395 
396         return(list(train_list = train_lgb, train_err_list = train_pred, val_list = validation
                ))
397     }
398 
399     else break
400 }
401 
402 # Create list of training and validation sets
403 
404 data_svr_lists <- data_svr_lgb_wrangle(data_svr_array, type = "svr")
405 
406 # Tune SVR regressions
407 svr_hyp <- expand.grid(gamma = c(exp(seq(-2.5, by = 1.5, 3.5))),
408                        lambda = c(exp(seq(-2.5, by = 1.5, 3.5)))) %>%
409     split(., seq(nrow(.)))
410 
411 svr_fit <- hyp_tune_func(data_lists = data_svr_lists,
412                          data_actual = data_val,
```

```
                          hyperparams = svr_hyp,
                          fit_func = svr_fit_func, type = "svr")

# Support Vector Regression fit function

svr_fit_func <- function(hyperparams, data_lists){
    set.seed(12439078)
    SVR_fit_list <- list()
    SVR_pred_train <- list()
    SVR_pred_val <- list()
    for(i in 1:12){
        SVR_fit_list[[i]] <- svmRegression(x = data_lists$train_list[[i]][,-1],
                                           y = data_lists$train_list[[i]][,"Y"],
                                           gamma = hyperparams$gamma,
                                           lambda = hyperparams$lambda)

        SVR_pred_train[[i]] <- predict(SVR_fit_list[[i]], data_lists$train_list[[i]][,-1])
        SVR_pred_val[[i]] <- predict(SVR_fit_list[[i]], data_lists$val_list[[i]][,-1])
        print(paste(i,"DONE!!"))
    }

    SVR_train_df <- SVR_pred_train %>%  do.call("cbind", .) %>% t(.) %>% c(.)
    SVR_val_df <- SVR_pred_val %>%  do.call("cbind", .) %>% t(.) %>% c(.)
    print("DONE!!")
    list(train = SVR_train_df, val = SVR_val_df)
}

# Best hyperparameters

best_tune_svr <- svr_fit %>% filter(mse == min(mse))


# Plot performance
svr_perf <- perf_plot(data_lists = data_svr_lists,
                      data_val = data_val,
                      data_train = data_train,
                      hyperparams = best_tune_svr[1,],
                      fit_func = svr_fit_func,
                      type = "svr",
                      train_title = "SVR Train Plot",
                      val_title = "SVR Validation Plot")
svr_perf$train_plot
svr_perf$val_plot
svr_perf$training_error
svr_perf$performance

# GBM wrangle data

library(lightgbm)
```

```
462
463 data_lgb_train <- data_wrangling_func(data = data_power %>% rename(Date = date_time),
464                                        type = "lgb",
465                                        final_date = "2009-01-01")
466
467 data_lgb_val <- data_wrangling_func(data = data_power %>% rename(Date = date_time) %>% filter(
    Date >= as.Date("2009-01-01")),
468                                     type = "lgb",
469                                     final_date = "2010-01-01")
470
471 data_lgb_array <- data_array_func(data = data_lgb_train,
472                                   val = data_lgb_val,
473                                   initial = 2*24,
474                                   assess = 12,
475                                   skip = 11)
476
477 data_lgb_lists <- data_svr_lgb_wrangle(data_lgb_array, type = "lightgbm")
478
479 # GBM hyperparameter tuning
480
481
482 lgb_hyp <- expand.grid(boosting = c("dart", "goss"),
483                        learning_rate = c(0.01, 0.05, 0.1),
484                        feature_fraction = c(0.6, 0.8),
485                        num_leaves = c(2, 4, 8),
486                        max_depth = c(2),
487                        nrounds = c(50, 100, 150, 200, 250, 300, 350, 500),
488                        subsample = c(0.6),
489                        lambda_l1 = c(10),
490                        lambda_l2 = c(10)) %>%
491     split(., seq(nrow(.)))
492
493 lgb_fit <- hyp_tune_func(data_lists = data_lgb_lists,
494                          data_actual = data_val,
495                          hyperparams = lgb_hyp, fit_func = lgb_fit_func, type = "lightgbm")
496
497
498 # Function to fit lightGBM
499
500 lgb_fit_func <- function(hyperparams,
501                          data_lists){
502     set.seed(12439078)
503     GBM_fit_list <- list()
504     GBM_pred_train <- list()
505     GBM_pred_val <- list()
506     for(i in 1:length(data_lists$train_list)){
507         GBM_fit_list[[i]] <- lightgbm(
508             data = data_lists$train_list[[i]],
509             obj="regression",
```

```
            num_threads = 20,
            metric = "mse",
            nrounds = hyperparams$nrounds,
            boosting = hyperparams$boosting,
            params = list(learning_rate = hyperparams$learning_rate,
                          feature_fraction = hyperparams$feature_fraction,
                          num_leaves = hyperparams$num_leaves,
                          max_depth = hyperparams$max_depth,
                          subsample = hyperparams$subsample,
                          lambda_l1 = hyperparams$lambda_l1,
                          lambda_l2 = hyperparams$lambda_l2))

        lgb.get.eval.result(GBM_fit_list[[i]],
                            data_name = "train", eval_name = "l2")

        GBM_pred_train[[i]] <- predict(GBM_fit_list[[i]], data_lists$train_err_list[[i]])
        GBM_pred_val[[i]] <- predict(GBM_fit_list[[i]], data_lists$val_list[[i]])
    }
    GBM_pred_train_df <- GBM_pred_train %>%  do.call("cbind", .) %>% t(.) %>% c(.)
    GBM_pred_val_df <- GBM_pred_val %>%  do.call("cbind", .) %>% t(.) %>% c(.)
    return(list(train = GBM_pred_train_df, val = GBM_pred_val_df))
}


# Best hyperparameters

best_lgb_tune <- lgb_fit %>% filter(mse == min(mse))

# Plot performance


lgb_perf <- perf_plot(data_lists = data_lgb_lists,
                      data_train = data_train,
                      data_val = data_val,
                      hyperparams = best_tune,
                      fit_func = lgb_fit_func,
                      type = "lightgbm",
                      val_title = "GBM Validation Plot",
                      train_title = "GBM Train Plot")

lgb_perf$train_plot
lgb_perf$val_plot
lgb_perf$performance %>% mutate(across(everything(), ~as.character(.x))) %>%
    pivot_longer(cols = everything())
lgb_perf$training_error %>% mutate(across(everything(), ~as.character(.x))) %>%
    pivot_longer(cols = everything())


# Testing time
```

```
559
560 # Full Training and Test Data
561
562 data_full_train <- data_power %>% rename(Date = date_time) %>% filter(Date < as.Date("
        2010-01-01")) %>% tail(-48) %>% head(-7)
563
564 data_test <- data_power %>% rename(Date = date_time) %>% filter(Date >= as.Date("2010-01-01"))
        %>% tail(-48) %>% head(-10)
565
566
567 # LSTM Test Data
568 data_lstm_train <- data_wrangling_func(data = data_power %>% rename(Date = date_time),
569                                        type = "lstm",
570                                        final_date = "2010-01-01")
571 data_lstm_test <- data_wrangling_func(data = data_power %>% rename(Date = date_time) %>%
        filter(Date >= as.Date("2010-01-01")),
572                                       type = "lstm",
573                                       final_date = "2010-11-27")
574 # Set min and max for later transformations
575 min_max_train <- data_lstm_train$min_max
576 data_lstm_train <- data_lstm_train$data
577
578 test_min_max <- data_lstm_test$min_max
579 data_lstm_test <- data_lstm_test$data
580
581 ### Create LSTM Data Arrays
582 data_lstm_test_array <- data_array_func(data = data_lstm_train,
583                                         val = data_lstm_test,
584                                         initial = 2*24, assess = 12, skip = 11)
585
586 # SVR Test Data
587
588 data_svr_train <- data_wrangling_func(data = data_power %>% rename(Date = date_time),
589                                       type = "svr",
590                                       final_date = "2010-01-01")
591
592 data_svr_test <- data_wrangling_func(data = data_power %>% rename(Date = date_time) %>% filter
        (Date >= as.Date("2010-01-01")),
593                                      type = "svr",
594                                      final_date = "2010-11-27")
595
596 data_svr_test_array <- data_array_func(data = data_svr_train,
597                                        val = data_svr_test,
598                                        initial = 2*24,
599                                        assess = 12,
600                                        skip = 11)
601
602 data_svr_test_lists <- data_svr_lgb_wrangle(data_svr_test_array, type = "svr")
603
```

```
# LightGBM

data_lgb_train <- data_wrangling_func(data = data_power %>% rename(Date = date_time),
                                      type = "lgb",
                                      final_date = "2010-01-01")

data_lgb_test <- data_wrangling_func(data = data_power %>% rename(Date = date_time) %>% filter
    (Date >= as.Date("2010-01-01")),
                                     type = "lgb",
                                     final_date = "2010-11-27")

data_lgb_test_array <- data_array_func(data = data_lgb_train,
                                       val = data_lgb_test,
                                       initial = 2*24,
                                       assess = 12,
                                       skip = 11)

data_lgb_test_lists <- data_svr_lgb_wrangle(data_lgb_test_array, type = "lightgbm")


# Determine LSTM performance

lstm_test_perf <- perf_plot(data_lists = data_lstm_test_lists,
                            data_val = data_test,
                            data_train = data_full_train,
                            hyperparams = best_lstm_tune,
                            fit_func = lstm_fit_func,
                            min_max = list(train = min_max_train,
                                           val = test_min_max),
                            type = "lstm",
                            train_title = "LSTM Full Training Plot",
                            val_title = "LSTM Test Plot")

lstm_test_perf$train_plot
lstm_test_perf$val_plot
lstm_test_perf$performance
lstm_test_perf$training_error

# Determine SVR performance

svr_test_perf <- perf_plot(data_lists = data_svr_test_lists,
                           data_val = data_test,
                           data_train = data_full_train,
                           hyperparams = best_tune_svr[1,],
                           fit_func = svr_fit_func,
                           type = "svr",
                           train_title = "SVR Full Training Plot",
                           val_title = "SVR Test Plot")

```

```
svr_test_perf$train_plot
svr_test_perf$val_plot
svr_test_perf$training_error
svr_test_perf$performance

# Determine lgithGBM performance

lgb_test_perf <- perf_plot(data_lists = data_lgb_test_lists,
                           data_train = data_full_train,
                           data_val = data_test,
                           hyperparams = best_tune,
                           fit_func = lgb_fit_func,
                           type = "lightgbm",
                           val_title = "GBM Test Plot",
                           train_title = "GBM Full Training Plot")

lgb_test_perf$train_plot
lgb_test_perf$val_plot
lgb_test_perf$performance %>% mutate(across(everything(), ~as.character(.x))) %>%
    pivot_longer(cols = everything())
lgb_test_perf$training_error %>% mutate(across(everything(), ~as.character(.x))) %>%
    pivot_longer(cols = everything())
```

Code/Assignment 1.1 code.R

A.2 Assignment 1.2 code

```
# Assignment 1 ====

library(parallel)
library(glmnet)
#library(DMwR)
library(MASS)
library(caret)
library(rsample)
library(cowplot)
library(ggplot2)
library(FNN)
library(tidyverse)

# Exercises ----

# Fig 1 ----

#> Data and Functions ----

set.seed(09874235)
betas <- rnorm(n = 50, mean = 0, sd = 1)
```

```
22
23 V <- matrix(0.7, nrow = 50, ncol = 50)
24 diag(V) <- rep(1, 50)
25
26 data_fun <- function(my_arg = NULL, beta = betas, V = V){
27   if(is.null(my_arg)){}
28
29   X <- mvrnorm(n = 50, mu = rep(0, 50), Sigma = V)
30   eps <- rnorm(50, 0, 1)
31   Y <- t(betas) %*% X + eps
32
33   cbind(Y = t(Y), X)
34
35 }
36
37 train_dat  <- vector(mode = "list", length = 100)
38
39 train_dat <- lapply(train_dat, data_fun, beta = betas, V = V)
40
41 test_dat <- list(NULL)
42
43 test_dat <- lapply(test_dat, data_fun, beta = betas, V = V)
44
45 # Functions ----
46
47 # Estimate Betas
48 beta_est <- function(X, lam, test){
49   est <- solve(t(X[,-1]) %*% X[,-1] + (lam * diag(nrow = nrow(X)))) %*% t(X[,-1]) %*% X[,1]
50   est
51 }
52
53 # Estimate Y hat
54 fit_func <- function(est, test){
55   y_hat <- t(est) %*% test[,-1]
56   t(y_hat)
57 }
58
59 # Calculate MSE
60 mse_func <- function(y_hat, test){
61   mse <-  mean((y_hat-test[,1])^2)
62   mse
63 }
64
65 # Bias Variance Function
66 bias_var_func <- function(fits, test){
67   mat_fit <- do.call("rbind", fits)
68   bias <- apply(mat_fit, 2, mean)
69   var_fit <- apply(mat_fit, 2, var)
70
```

```r
  bias_square <- (bias - test[,1])^2

  list(var = mean(var_fit), bias_square = mean(bias_square))
}

# Function to wrap up all other functions

wrap_func <- function(lam, X, test){
  beta_ests <- mclapply(X, beta_est, lam = lam, test = test[[1]], mc.cores = 6)
  fits <- mclapply(beta_ests, fit_func,  test = test[[1]], mc.cores = 6)
  mse <- mclapply(fits, mse_func, test = test[[1]], mc.cores = 6)
  mse <- mean(unlist(mse))
  bias_var_fit <- bias_var_func(fits, test = test[[1]])
  data.frame(lambda = lam, bias = bias_var_fit$bias_square, var = bias_var_fit$var, mse = mse)
}

# Fit ----

# Lambda sequence

lambda <- as.list(seq(0,8, 0.5))

# Run Functions
bias_var_tradeoff <- map(lambda, wrap_func, X = train_dat, test = test_dat)

# Plot Bias Variance Trade Off
fig_1_left <- bias_var_tradeoff %>% do.call("rbind", .) %>% pivot_longer(cols = c("bias", "var", "mse")) %>%tail(-3) %>% filter(!name %in% "mse") %>%
  ggplot() +
  geom_line(aes(lambda, value, colour = name)) +
  geom_point(aes(lambda, value, colour = name), size = 0.4)+
  theme_minimal_hgrid() +
  labs(subtitle = "Bias-Variance Tradeoff", x = expression(lambda), y = "") +
  guides(colour = guide_legend(title = ""))

fig_1_right <- bias_var_tradeoff %>% do.call("rbind", .) %>% pivot_longer(cols = c("bias", "var", "mse")) %>%tail(-3) %>% filter(name %in% "mse") %>%
  ggplot() +
  geom_line(aes(lambda, value), colour = "blue") +
  geom_point(aes(lambda, value), colour = "blue", size = 0.4)+
  theme_minimal_hgrid() +
  labs(subtitle = "MSE", x = expression(lambda), y = "") +
  guides(colour = guide_legend(title = ""))

plot_grid(fig_1_left, fig_1_right)

# Fig 2 and 3 ====
#> Data ----

```

```r
source("data/diabetes.txt")

x_dat <- diabetes[[1]]
y_dat <- diabetes[[2]]

set.seed(2021)

split_func <- function(mat, y, x){

  if(is.null(mat)){}

  dat <- cbind(y, x)

  split <- initial_split(as.data.frame(dat), prop = 0.5)

  train <- training(split)
  test_val <- testing(split)

  split_test_val <- initial_split(test_val, prop = 0.5)

  test <- training(split_test_val)
  val <- testing(split_test_val)

  out <- list(train = train, validation = val, test = test)
  out
}

dat_list <- vector(mode = "list", 100)

dat_list <- lapply(dat_list, split_func, y = y_dat, x = x_dat)

#> KNN ------

# KNN function

knn_func <- function(ks, data){

  mod <- knnreg(x = data$train[,-1], y = data$train[,1], k = ks)
  pred_train <- predict(mod, data$train[,-1])
  rmse_train <- sqrt(mean((pred_train - data$train[,1])^2))

  pred_val <- predict(mod, data$val[,-1])
  rmse_val <- sqrt(mean((pred_val - data$val[,1])^2))

  train_full <- rbind(data$train, data$val)
  mod_test <- knnreg(x = train_full[,-1], y = train_full[,1], k = ks)
  pred_test <- predict(mod, data$test[,-1])
  rmse_test <- sqrt(mean((pred_test - data$test[,1])^2))

```

```
  out <- data.frame(rmse_train = rmse_train, rmse_val = rmse_val, rmse_test = rmse_test)
  out

}

ks_list <- as.list(seq(1, 150))

# Fit KNN models to all data sets

knn_res <- mclapply(ks_list, function(X){do.call("rbind", lapply(dat_list, knn_func, ks = X))
    }, mc.cores = 6)

# KNN training and validation RMSE for data set 2

knn_top <- data.frame(k = seq(1, 150), do.call(rbind, lapply(knn_res, function(X){X[2,]})))
colnames(knn_top) <- c("k", "train", "val", "test")

mdf <- reshape2::melt(knn_top[,-4], id.var = "k")
mdf %>%
  ggplot() +
  theme_bw()+
  geom_line(aes(x = k, y = value, color = variable)) +
  theme_bw() +
  geom_point(aes(x = k, y = value, color = variable), size = 0.5) +
  ylim(c(40, 90)) +
  labs(title = "Plot of Training and Validation Error as a Function of k", x = "k", y = "RMSE"
      ) +
  theme(plot.title = element_text(size=10), legend.position = "right") +
  geom_hline(yintercept = min(knn_top[,3]), linetype = "dotted")

# Average of all k's

knn_all <- data.frame(k = seq(1, 150), do.call(rbind, lapply(knn_res, function(X){apply(X , 2,
     mean)})))

# Create green box plots in Figure 3

knn_box <- data.frame(k = as.character(rep(1:150, each = 100)), do.call(rbind, knn_res))

colnames(knn_box) <- c("k", "train", "val", "test")

mdf_box <- reshape2::melt(knn_box[,-c(2,3)], id.var = "k")

mdf_box$k <- factor(mdf_box$k, levels = as.character(seq(1,150)))

# Best k for each data set with results

knn_dat_list <- as.list(seq(1:100))

```

```
knn_best <- as.data.frame(do.call(rbind, lapply(knn_dat_list, function(X, data){

  temp <- do.call(rbind, lapply(data, function(D){D[X,]}))

  temp_res <- temp[which.min(temp[, 2]), ]

  temp_res <- cbind(k = which.min(temp[, 2]), temp_res)

  temp_res

}, data = knn_res)))

# Which K is most common best k on validation set

names(which.max(table(knn_best$k)))

# Create red boxplot in figure 3

knn_best_plot <- cbind(k = rep(151, 100), knn_best[,-1])

mdf_best <- reshape2::melt(knn_best_plot[,-c(2,3)], id.var = "k")

fig_3_box <- mdf_box %>%
  ggplot() +
  theme_bw() +
  geom_boxplot(aes(x = k, y = value), linetype = "dashed", outlier.shape = NA, fill = "green")
      +
  stat_boxplot(aes(x = k, y = value, ymin = ..lower.., ymax = ..upper..), outlier.shape = NA,
      fill = "green") +
  scale_x_discrete(breaks = c(0, seq(25, 150, 25))) +
  labs(title = "Boxplots of Test RMSE", y = "RMSE") +
  geom_hline(yintercept = median(knn_best_plot$rmse_test))

fig_3_red <- mdf_best %>%
  ggplot() +
  theme_bw() +
  geom_boxplot(aes(x = k, y = value), linetype = "dashed", outlier.shape = NA) +
  stat_boxplot(aes(x = k, y = value, ymin = ..lower.., ymax = ..upper..), outlier.shape = NA,
      fill = "red") +
  scale_x_discrete(breaks = c(0, seq(25, 150, 25))) +
  labs(title = " ", y = " ") +
  scale_y_continuous(breaks = c(seq(50, 90, 10))) +
  stat_boxplot(geom = "errorbar", aes(x = k, y = value, ymin = ..ymax..)) +
  stat_boxplot(geom = "errorbar", aes(x = k, y = value, ymax = ..ymin..)) +
  geom_hline(yintercept = median(knn_best_plot$rmse_test)) +
  geom_point(data = data.frame(k = c(150, 150), value = c(min(mdf_box$value), 90)), mapping =
      aes(x = k, y = value), color = NA)

plot_grid(fig_3_box, fig_3_red, rel_widths = c(3,1), align = "hv")
```

```r

#> Ridge ----

# Fit ridge regressions over range of lambdas

ridge_lambdas <- seq(8.01, 0.01, by = -0.25)
ridge_lambds_rep <- unlist(lapply(as.list(ridge_lambdas), rep, 100))

dat_list_train <- lapply(dat_list, function(X){as.matrix(X$train)})
dat_list_val <- lapply(dat_list, function(X){as.matrix(X$val)})
dat_list_full_train <- lapply(dat_list, function(X){as.matrix(rbind(X$train, X$val))})
dat_list_test <- lapply(dat_list, function(X){as.matrix(X$test)})

ridge_mods_train <- lapply(dat_list_train, ridge_beta_func, lambda = ridge_lambdas)
ridge_mods_full <- lapply(dat_list_full_train, ridge_beta_func, lambda = ridge_lambdas)

# Calculate train, validation and test RMSE

mse_train <- do.call(rbind, lapply(dat_list_train, mse_func, model = ridge_mods_train))
mse_val <- do.call(rbind, lapply(dat_list_val, mse_func, model = ridge_mods_train))

mse_test <- do.call(rbind, lapply(dat_list_test, mse_func, model = ridge_mods_full))

# Fit Figure 2 with Ridge Regression

ridge_fig_2 <- data.frame(lambda = ridge_lambdas,
                          train = as.vector(apply(mse_train, 2, mean)),
                          validation = as.vector(apply(mse_val, 2, mean)),
                          test = as.vector(apply(mse_test, 2, mean)))

mdf_ridge <- pivot_longer(ridge_fig_2, cols = c("train", "validation", "test"))
mdf_ridge <- reshape2::melt(ridge_fig_2[,-4], id.var = "lambda")

ggplot(mdf_ridge, aes(x = lambda, y = value, color = variable)) +
  geom_line() +
  geom_point(size = 0.5) +
  labs(title = expression(paste("Plot of Training and Validation Error as a Function of ",
       lambda)), x = "lambda", y = "RMSE") +
  theme(plot.title = element_text(size=10), legend.position = "right")

# Fit Figure 3

ridge_fig_3 <- data.frame(lambda = ridge_lambds_rep,
                          train = as.vector(apply(mse_train, 2, mean)),
                          validation = as.vector(apply(mse_val, 2, mean)),
                          test = as.vector(apply(mse_test, 2, mean)))

mdf_ridge_box <- reshape2::melt(ridge_fig_3[,-c(2,3)], id.var = "lambda")

```

```
fig_3_box <- ggplot(data = mdf_ridge_box, aes(group = lambda, y = value)) +
  geom_boxplot(linetype = "dashed", outlier.shape = NA, fill = "green") +
  stat_boxplot(aes(ymin = ..lower.., ymax = ..upper..), outlier.shape = NA, fill = "green") +
  scale_x_discrete(breaks = ridge_lambds_rep) +
  labs(title = "Boxplots of Test RMSE", y = "RMSE")


  geom_hline(yintercept = median(knn_best_plot$rmse_test))


# Fig 4 ====

# Create Data

sim_func <- function(p, n){

  lapply(n, function(n){

  X <- matrix(NA, ncol = p, nrow = 1000)

  X <- apply(X, 2, function(x){x <- runif(n = 1000, min = -1, max = 1)})

  Y <- apply(X, 1, function(x){exp(-8 * sqrt(sum(x^2)))})

  out <- data.frame(Y = Y, X = X)

  n <- out}
  )

}

dats <- as.list(seq(1, 10000))
dims <- as.list(seq(1,8))

data_ex3 <- mclapply(dims, sim_func, n = dats, mc.cores = 6)

sim_func_2 <- function(p, n){

  lapply(n, function(n){

    X <- matrix(NA, ncol = p, nrow = 1000)

    X <- apply(X, 2, function(x){x <- runif(n = 1000, min = -1, max = 1)})

    Y <- apply(X, 1, function(x){0.5 * (x[1] + 1)^3})

    out <- data.frame(Y = Y, X = X)

    n <- out}
```

```
  )

}

data_ex3_2 <- mclapply(dims, sim_func_2, n = dats, mc.cores = 6)

# Fit KNN

knn_1_func <- function(data)
{
  temp <- data
  preds <- lapply(temp, function(X){
              fit <- knnreg(x = as.matrix(X[,-1]), y = as.matrix(X[,1]), k = 1)
              pred <- predict(fit, newdata = matrix(rep(0, ncol(X) - 1), nrow = 1))
              pred

  })
  preds

}

knn_1_res <- mclapply(data_ex3, knn_1_func, mc.cores = 6)
knn_1_res_2 <- mclapply(data_ex3_2, knn_1_func, mc.cores = 6)

# Calculate Bias, Variance, MSE

calc_func <- function(data){

  temp <- do.call(rbind, data)

  sqr_bias <- (1 - mean(temp))^2
  var_temp <- var(temp)
  mse <- mean((1 - temp)^2)

  out <- data.frame("Squared Bias" = sqr_bias, Variance = var_temp, MSE = mse)
  out

}

calc_func_2 <- function(data){

  temp <- do.call(rbind, data)

  sqr_bias <- (0.5 - mean(temp))^2
  var_temp <- var(temp)
  mse <- mean((0.5 - temp)^2)

  out <- data.frame("Squared Bias" = sqr_bias, Variance = var_temp, MSE = mse)
  out
```

```r

}

cal_res <- cbind(Dimension = seq(1,8), do.call(rbind, lapply(knn_1_res, calc_func)))

fig_ex_4 <- cal_res %>%
ggplot() +
  theme_minimal_hgrid() +
  geom_line(aes(x = Dimension, y = Squared.Bias, color = "Squared Bias")) +
  geom_point(aes(x = Dimension, y = Squared.Bias, color = "Squared Bias")) +
  geom_line(aes(x = Dimension, y = Variance, color = "Variance")) +
  geom_point(aes(x = Dimension, y = Variance, color = "Variance")) +
  geom_line(aes(x = Dimension, y = MSE, color = "MSE")) +
  geom_point(aes(x = Dimension, y = MSE, color = "MSE")) +
  labs(title = "MSE vs Dimension", y = "")+
  theme(legend.title = element_blank())

cal_res_2 <- cbind(Dimension = seq(1,8), do.call(rbind, lapply(knn_1_res_2, calc_func_2)))

fig_ex_4_2 <- cal_res_2 %>%
  ggplot() +
  theme_minimal_hgrid() +
  geom_line(aes(x = Dimension, y = Squared.Bias, color = "Squared Bias")) +
  geom_point(aes(x = Dimension, y = Squared.Bias, color = "Squared Bias")) +
  geom_line(aes(x = Dimension, y = Variance, color = "Variance")) +
  geom_point(aes(x = Dimension, y = Variance, color = "Variance")) +
  geom_line(aes(x = Dimension, y = MSE, color = "MSE")) +
  geom_point(aes(x = Dimension, y = MSE, color = "MSE")) +
  labs(title = "MSE vs Dimension", y = "")+
  theme(legend.title = element_blank())
```

Code/Assignment 1.2.R

A.3 Assignment 2.2 code

```r
# Assingment 2.2 code
pacman::p_load(MASS, tidyverse, tictoc, cowplot)

# Data

data_fun <- function(beta, V, p, n){

  X <- mvrnorm(n = n, mu = rep(0, p), Sigma = V)
  eps <- rnorm(n)
  Y <- X %*% beta + eps

  list(Y, X)
}
```

```r

# Gradient Descent Simple Linear Regression

simp_func <- function(Y, X, beta, t_b, limit, grad_func, mse_func){
  # Set timer and parameters
  tic.clearlog()
  tic()
  i = 0
  mse <- mse_func(X = X, Y = Y, beta = beta)
  # Create data.frame for results
  res <- as_tibble(data.frame("iterations" = i, "MSE" = mse))
  # Repeat update steps till MSE is close to true MSE
  while(mse > limit)
  {
    # Update iterations
    i <- i+1
    # Determine gradient
    grad <- grad_func(X, Y, beta)
    # Update parameter
    beta <- beta - t_b * grad
    # Determine mean squared error
    mse <- mse_func(X, Y, beta)
    # Record iteration
    res <- rbind(res, c(i, mse))
  }
  # Determine time taken
  toc(log = TRUE, quiet = TRUE)
  time <- tic.log(format = TRUE) %>%
    str_extract(., "[[:digit:]]+[[:punct:]][[:digit:]]+") %>%
    as.numeric()
  # Create output of results
  list("Final_Values" = beta,
       "No_Iterations" = i,
       "Results" = res,
       time = time)
}


#### Gradient Functions for Linear Regression

# Gradient
grad_lin_func <- function(X, Y, beta){
  out <- 2 * t(X) %*% (X %*% beta - Y)
  out
}
# Hessain matrix
hes_lin_func <- function(X){
  out <-  t(X) %*% X
  out
```

```
}
# Mean squared error
mse_lin_func <- function(X, Y, beta){
  out <- t(X %*% beta - Y) %*% (X %*% beta - Y)
  out
}
# Coordinate descent
coord_lin_func <- function(X_i, X_j, X_di, Y, beta_j){
  out <- t(X_i) %*% (Y - X_j %*% beta_j)/X_di
  out
}

#### Gradient Function for Ridge Regression

# Gradient
grad_ridge_func <- function(X, Y, beta, lambda = 10){
  out <- 2 * t(X) %*% (X %*% beta - Y) + 2 * lambda * beta
  out
}
# Hessian matrix
hes_ridge_func <- function(X, lambda = 10){

  out <-  2 * t(X) %*% X + 2 * lambda * diag(1, nrow = ncol(X))
  out
}
# Mean squared error
mse_ridge_func <- function(X, Y, beta, lambda = 10){
  out <- t(X %*% beta - Y) %*% (X %*% beta - Y) + lambda * t(beta) %*% beta
  out
}
# Coordinate descent
coord_ridge_func <- function(X_i, X_j, X_di, Y, beta_j, lambda = 10){
  out <- t(X_i) %*% (Y - X_j %*% beta_j)/(X_di + lambda)
  out
}

#### LASSO

# Mean squared error
mse_lasso_func <- function(X, Y, beta, lambda = 10){
  out <- t(X %*% beta - Y) %*% (X %*% beta - Y) + sum(lambda * abs(beta))
  out
}
# Coordinate descent
coord_lasso_func <- function(X_i, X_j, X_di, Y, beta_j, lambda = 10){
  fit <- t(X_i) %*% ((Y - X_j %*% beta_j)/X_di)
  thres <- lambda/X_di
  fit_sign <- ifelse(fit > 0, 1, -1)
  out <- fit_sign * ifelse(abs(fit) - thres < 0, 0, abs(fit) - thres)
```

```
  out
}

# Gradient Descent Simple Linear Regression

newt_func <-  function(X, Y, a, b, t_b, beta, limit, grad_func, mse_func, hes_func){
  # Set timer and parameters
  tic.clearlog()
  tic()
  i = 0
  mse <- mse_func(X, Y, beta)
  # Create data.frame for results
  res <- as_tibble(data.frame("iterations" = i, "MSE" = mse))
  # Repeat update steps till MSE is close to true MSE
  while(mse > limit)
  {
    # Set parameters
    f_t <- 2
    f_a <- 1
    t_b <- 1
    # Perform backtracking line search
    while(f_t > f_a)
    {
      # Calculate gradient
      grad <- grad_func(X, Y, beta)
      # Calculate v
      v <- -solve(hes_func(X)) %*% grad
      # Calculate values
      f_t <- mse_func(X, Y, (beta + t_b * v))
      f_a <- mse_func(X, Y, beta)  + a * t_b * t(grad) %*% v
      # Update step size
      t_b <- t_b*b
    }
    # Perform update
    beta <- beta + t_b * v
    # Update step number
    i <- i + 1
    # Determine performance
    mse <- mse_func(X, Y, beta)
    # Record iteration
    res <- rbind(res, c(i, mse))
  }
  # Determine time taken
  toc(log = TRUE, quiet = TRUE)
  time <- tic.log(format = TRUE) %>%
    str_extract(., "[[:digit:]]+[[:punct:]][[:digit:]]+") %>%
    as.numeric()
  # Create output of results
  list("Final_Values" = beta,
```

```
        "No_Iterations" = i,
        "Results" = res,
        time = time)
}

## Coordinate Descent Algorithm

coord_func <- function(X, Y, beta, limit, mse_func, up_func){
  # Set timer and parameters
  tic.clearlog()
  tic()
  run = 0
  p <- ncol(X)
  mse <- mse_func(X, Y, beta)
  # Create data.frame for results
  res <- as_tibble(data.frame("iterations" = run, "MSE" = mse))
  # Calculate diagonal of l2-norm of data
  X_d <- diag(t(X)%*%X)
  # Repeat update steps till MSE is close to true MSE
  while(mse >= limit){
    # Update iterations
    run <- run + 1
    # Cycle through parameters individually
    for(i in 1:p)
    {
      # Perform update step
      beta[i, 1] <- up_func(X_i = X[, i],
                            X_j = X[, -i],
                            X_di = X_d[i],
                            Y = Y,
                            beta_j = beta[-i, 1])
      # Update step number
      i = i + 1
    }
    # Determine MSE after full cycle
    mse <- mse_func(X, Y, beta)
    # Record iterations
    res <- rbind(res, c(run, mse))
  }
  # Determine time taken
  toc(log = TRUE, quiet = TRUE)
  time <- tic.log(format = TRUE) %>%
    str_extract(., "[[:digit:]]+[[:punct:]][[:digit:]]+") %>%
    as.numeric()
  # Create output of results
  list("Final_Values" = beta,
       "No_Iterations" = run,
       "Results" = res,
       time = time)
```

```
}


# Back Tracking Line Search

back_func <-  function(X, Y, a, b, t_b, beta, limit, grad_func, mse_func){
  # Set timer and parameters
  tic.clearlog()
  tic()
  i = 0
  mse <- mse_func(X, Y, beta)
  # Create data.frame for results
  res <- as_tibble(data.frame("iterations" = i, "MSE" = mse))
  # Repeat update steps till MSE is close to true MSE
  while(mse > limit)
  {
    # Set parameters
    f_t <- 2
    f_a <- 1
    t_b <- 0.1
    # Perform backtracking line search
    while(f_t > f_a)
    {
      # Calculate gradient
      grad <- grad_func(X, Y, beta)
      # Calculate values
      f_t <- mse_func(X, Y, (beta - t_b * grad))
      f_a <- mse_func(X, Y, beta) - a * t_b * t(grad) %*% grad
      # Update step size
      t_b <- t_b*b
    }
    # Perform update
    beta <- beta - t_b * grad
    # Update step number
    i <- i + 1
    # Determine performance
    mse <- mse_func(X, Y, beta)
    # Record iteration
    res <- rbind(res, c(i, mse))
  }
  # Determine time taken
  toc(log = TRUE, quiet = TRUE)
  time <- tic.log(format = TRUE) %>%
    str_extract(., "[[:digit:]]+[[:punct:]][[:digit:]]+") %>%
    as.numeric()
  # Create output of results
  list("Final_Values" = beta,
       "No_Iterations" = i,
       "Results" = res,
```

```
259         time = time)
260 }
261
262 # Function to plot results and time
263
264 plot_func <- function(results, time, title, full = TRUE){
265   # Get different methods
266   methods <- names(results)
267   # Create list of methods
268   methods_list <- methods %>% as.list() %>% modify(.x, .f = ~c("iterations", .))
269   # Combine iterations data
270   final_res <- results %>%
271     Map(setNames, ., methods_list) %>%
272     reduce(., full_join, by = "iterations") %>%
273     pivot_longer(., cols = methods) %>%
274     rename(method = name, MSE = value)
275   # Set plot limits
276   if(!full){
277     upper = sort(final_res$MSE, decreasing = TRUE)[10]
278     lower = sort(final_res$iterations)[length(final_res$iterations)/2]
279     final_res <- final_res %>%
280       mutate( MSE = ifelse(MSE < upper, MSE, NA)) %>%
281       filter(iterations < lower)
282   }
283   # Plot iterations
284   final_plot <- final_res %>%
285     group_by(method) %>%
286     ggplot() +
287     theme_bw() +
288     geom_line(aes(x = iterations, y = MSE, color = method), alpha = 0.8) +
289     labs(x = "iterations") +
290     geom_point(aes(iterations, y = MSE, color = method), size =0.4) +
291     theme(legend.title = element_blank())
292   # COmbine times taken data
293   time_res <- cbind(time) %>%
294     as_tibble() %>%
295     cbind(., as_tibble(methods)) %>%
296     rename(method = value) %>%
297     rename(seconds = time)
298   # Plot times taken
299   time_plot <- time_res %>%
300     group_by(method) %>%
301     ggplot() +
302     theme_bw() +
303     geom_bar(aes(x = method, y = seconds, fill = method), stat = "identity", width = 0.2) +
304     guides(fill = NULL) +
305     labs(x = "time") +
306     theme(legend.title = element_blank(),
307           axis.text.x=element_blank(),
```

```
          axis.ticks.x=element_blank())
  # Create single plot
  plots <- plot_grid(final_plot, time_plot, nrow = 2, ncol = 1, align = "h")
  title_plot <- ggdraw() + draw_label(title)
  plot_grid(title_plot, plots, ncol=1, rel_heights=c(0.1, 1))
}

#### Creating simulated data
# Setting seed
set.seed(2021)
# Setting number of parameters
p <- 20
# Creating parameters
betas <- matrix(rnorm(n = p, mean = 0, sd = 1), p, 1)
# Creating data
V <- matrix(0.7, nrow = p, ncol = p)
diag(V) <- rep(1, p)
train_dat <- data_fun(beta = betas, V = V, p = p, n = 100)
# Setting targets and predictors
Y <- train_dat[[1]]
X <- train_dat[[2]]


#### Linear Regression
# Setting MSE limit
limit_lin <- mse_lin_func(X, Y, betas) * 1.01
# Setting Initial Estimates for parameters
beta_first <- matrix(seq(from = 0.03, to = 1.01, by = 1/p), p, 1)
# Setting step size for gradient descent
t_b <- 0.0001
# Performing gradient descent
simp_res <- simp_func(Y = Y, X = X,  t_b = t_b,
                      beta = beta_first,
                      limit = limit_lin,
                      grad_func = grad_lin_func,
                      mse_func = mse_lin_func)
# Set backtracking line search parameters and initial step size
a <- 0.2
b <- 0.7
t_b <- 0.1
# Performing gradient descent with backtracking line search
back_res <- back_func(X = X, Y = Y, a = a, b = b, t_b,
                      beta = beta_first,
                      limit = limit_lin,
                      grad_func = grad_lin_func,
                      mse_func = mse_lin_func)
# Performing coordinate descent
coord_res <- coord_func(X = X, Y = Y,
                        beta = beta_first,
```

```
                          limit = limit_lin,
                          mse_func = mse_lin_func,
                          up_func = coord_lin_func)
# Performing Newton's Method
newt_res <- newt_func(X = X, Y = Y, a = a, b = b, t_b = 1,
                      beta = beta_first,
                      limit = limit_lin,
                      grad_func = grad_lin_func,
                      mse_func = mse_lin_func,
                      hes_func = hes_lin_func)
# Collecting number of iterations
results_lin <- list(simple = simp_res$Results,
                    backtracking = back_res$Results,
                    coordinate = coord_res$Results,
                    newton = newt_res$Results)
# Collecting time taken
time_lin <- list(simple = simp_res$time,
                 backtracking = back_res$time,
                 coordinate = coord_res$time,
                 newton = newt_res$time)


#### Ridge Regression
# Setting MSE limit
limit_ridge <- mse_ridge_func(X, Y, beta = betas, lambda = 10) * 1.01
# Setting Initial Estimates for parameters
beta_first <- matrix(seq(from = 0.03, to = 1.01, by = 1/p), p, 1)
# Setting step size for gradient descent
t_b <- 0.0001
# Performing gradient descent
simp_ridge_res <- simp_func(Y = Y, X = X, t_b = t_b,
                            beta = beta_first,
                            limit = limit_ridge,
                            grad_func = grad_ridge_func,
                            mse_func = mse_ridge_func)
# Set backtracking line search parameters and initial step size
a <- 0.2
b <- 0.7
t_b <- 0.1
# Performing gradient descent with backtracking line search
back_ridge_res <- back_func(X = X, Y = Y, a = a, b = b, t_b,
                            beta = beta_first,
                            limit = limit_ridge,
                            grad_func = grad_ridge_func,
                            mse_func = mse_ridge_func)
# Performing coordinate descent
coord_ridge_res <- coord_func(X = X, Y = Y,
                              beta = beta_first,
                              limit = limit_ridge,
```

```
                                mse_func = mse_ridge_func,
                                up_func = coord_ridge_func)
# Performing Newton's Method
newt_ridge_res <- newt_func(X = X, Y = Y, a = a, b = b, t_b = 1,
                            beta = beta_first,
                            limit = limit_ridge,
                            grad_func = grad_ridge_func,
                            mse_func = mse_ridge_func,
                            hes_func = hes_ridge_func)
# Collecting number of iterations
results_ridge <- list(simple = simp_ridge_res$Results,
                      backtracking = back_ridge_res$Results,
                      coordinate = coord_ridge_res$Results,
                      newton = newt_ridge_res$Results)
# Collecting time taken
time_ridge <- list(simple = simp_ridge_res$time,
                   backtracking = back_ridge_res$time,
                   coordinate = coord_ridge_res$time,
                   newton = newt_ridge_res$time)

#### Lasso regression
# Performing coordinate descent
coord_lasso_res <- coord_func(X = X, Y = Y,
                              beta = beta_first,
                              limit = limit_ridge,
                              mse_func = mse_lasso_func,
                              up_func = coord_lasso_func)
# Collecting iterations
results_lasso <- list(coordinate = coord_lasso_res$Results)
# Collecting time taken
time_lasso <- list(coordinate = coord_lasso_res$time)

#### Plotting linear regression performances
plot_func(results = results_lin,
          time = time_lin,
          title = "Methods for Solving Linear Regression",
          full = FALSE)

#### Plotting ridge regression performances
plot_func(results = results_ridge,
          time = time_ridge,
          title = "Methods for Solving Ridge Regression",
          full = FALSE)

# Plotting iterations and time taken
plot_func(results = results_lasso,
          time = time_lasso,
          title = "Methods for Solving LASSO Regression")
```

Code/Assignment 2.2 code.R

A.4 Assignment 3 code

```
####### Assignment 3

pacman::p_load(tidyverse, fastDummies, glmnet, lars, rsample, FNN, parallel)

# Read in Data

dat_day <- read.csv("Bike-Sharing-Dataset/day.csv")
dat_hour <- read.csv("Bike-Sharing-Dataset/hour.csv")

# Functions

# Data Wrangle Function

data_func <- \(dat, hour = FALSE){
  # Data wrangle
  dat_mod <- dat %>%
    select(-c(instant, dteday, casual, registered)) %>%
    dummy_cols(c("season", "mnth", "weekday", "weathersit"),
               remove_first_dummy = TRUE,
               remove_selected_columns = TRUE) %>%
    as_tibble()

  if(hour){
    dat_mod <- dat_mod %>%
      dummy_cols(., c("hr"), remove_first_dummy = TRUE,
                 remove_selected_columns = TRUE) %>%
      as_tibble()
  }

  dat_split <- initial_split(data = dat_mod,
                             prop = 0.75)
  # Traina and Test Split
  train_dat <- training(dat_split)
  test_dat <- testing(dat_split)
  list(Train = train_dat, Test = test_dat)
}

# KNN Cp Function

knn_cp <- \(train_data, test_data){

  # Fit low Bias Model for Sigma
```

```
  knn_fit_5 <- knn.reg(train = train_data %>% select(-cnt),
                       test = train_data %>%  select(-cnt),
                       y = train_data %>% pull(cnt),
                       k = 5,
                       algorithm = "brute")

  # Determine Sigma
  sigma_knn <- mean((knn_fit_5$pred - train_data %>% pull(cnt))^2)
  # Create function to test different K values
  knn_func <- \(k, sigma_knn){

    knn_fit <- knn.reg(train = train_data %>% select(-cnt),
                       test = train_data %>%  select(-cnt),
                       y = train_data %>% pull(cnt),
                       k = k,
                       algorithm = "brute")

    Cp <- mean((knn_fit$pred - train_data %>% pull(cnt))^2) + 2*sigma_knn/k
    Cp
  }

  # Set K values
  k_val <- seq(2, 25, 1)
  # Determine Cp values for K's
  knn_cp <- mcmapply(FUN = knn_func, k_val, MoreArgs = list(sigma_knn = sigma_knn), mc.cores =
      4)
  # Determine best K
  k_best <- k_val[which.min(knn_cp)]
  # Plot Performance over different Ks
  knn_plot <- data.frame(x = k_val, y = knn_cp) %>%
    ggplot() +
    theme_bw() +
    geom_line(aes(x,y)) +
    geom_point(aes(x,y)) +
    labs(title = "KNN Cp for K's",
         x = "k",
         y = "Cp") +
    geom_vline(xintercept = k_best)
  # Fit Model to test data
  knn_final <- knn.reg(train = train_data %>% select(-cnt),
                       test = test_data %>%  select(-cnt),
                       y = train_data %>% pull(cnt),
                       k = k_best)
  # Determine performance
  knn_mse <- mean((knn_final$pred - test_data %>% pull(cnt))^2)

  list("Best K" = k_best, Plot = knn_plot, "Min MSE" = knn_mse)
}

```

```r
# KNN CV Function

knn_cv <- \(train_data, test_data){
  # CV func
  knn_cv_func <- \(k){
    knn_fit <- knn.reg(train = train_data %>% select(-cnt),
                       y = train_data %>% pull(cnt),
                       k = k,
                       algorithm = "brute")
    knn_fit$PRESS
  }
  # Set K values
  k_val <- seq(2, 25, 1)
  # Fit K using CV for different K values
  knn_cv <- mcmapply(knn_cv_func, k_val, mc.cores = 4)
  # Determine best K
  k_cv_best <- k_val[which.min(knn_cv)]
  # Plot Perforamnce
  knn_cv_plot <- data.frame(x = k_val, y = knn_cv) %>%
    ggplot() +
    theme_bw() +
    geom_line(aes(x,y)) +
    geom_point(aes(x,y)) +
    labs(title = "KNN CV for K's",
         x = "k",
         y = "PRESS") +
    geom_vline(xintercept = k_cv_best)
  # Fit Model to test data
  knn_cv_final <- knn.reg(train = train_data %>% select(-cnt),
                          test = test_data %>%  select(-cnt),
                          y = train_data %>% pull(cnt),
                          k = k_cv_best)
  # Determine performance
  knn_cv_mse <- mean((knn_cv_final$pred - test_data %>% pull(cnt))^2)

  list("Best K" = k_cv_best, Plot = knn_cv_plot, "Min MSE" = knn_cv_mse)
}

# LASSO Cp Function

lasso_cp <- \(train_data, test_data){
  # Fit Model
  lars_fit <- lars(x = train_data %>%  select(-cnt) %>%  as.matrix(),
                   y = train_data %>%  select(cnt) %>%  pull(),
                   type = "lasso", normalize = FALSE, intercept = FALSE)
  # Use Cp to determine best model
  best_lamdba <- lars_fit$lambda[which.min(lars_fit$Cp) - 1]
  # Fit Best Model from Cp
  lars_pred <- predict(object = lars_fit,
```

```
                        newx = test_data %>% select(-cnt) %>%  as.matrix(),
                        type = "fit",
                        mode = "lambda")
  # Determine Performance
  lars_mse <- mean((lars_pred$fit[, which.min(lars_fit$Cp) - 1] - test_data$cnt)^2)

  list("Best Lambda" = best_lambda, "Plot" = lars_fit, "Min MSE" = lars_mse)
}

# LASSO CV Function

lasso_cv <- \(train_data, test_data){

  # Fit LASSO using 8 fold CV
  cv_fit <- cv.glmnet(x = train_data %>% select(-cnt) %>%  as.matrix(),
                      y = train_data %>% pull(cnt),
                      nfolds = 8,
                      type.measure = "mse")
  # Minimum Lmabda
  best_lambda <- cv_fit$lambda.min
  cv_plot <- plot(cv_fit)
  # Fit Best Lambda
  cv_best <- glmnet(x = train_data %>% select(-cnt) %>%  as.matrix(),
                    y = train_data %>% pull(cnt),
                    lambda = best_lambda,
                    type.measure = "mse")
  # Determine Performance
  best_cv <- mean((predict(cv_best, test_data %>% select(-cnt) %>% as.matrix())
                   - test_data %>% pull(cnt))^2)

  list("Best Lambda" = best_lambda, "Plot" = cv_fit, "Min MSE" = best_cv)
}


# Data Wrangling

set.seed(2343578)

dat_day_wrangle <- data_func(dat_day)
dat_hour_wrangle <- data_func(dat_hour, hour = TRUE)

train_dat_day <- dat_day_wrangle$Train
test_dat_day <- dat_day_wrangle$Test

train_dat_hour <- dat_hour_wrangle$Train
test_dat_hour <- dat_hour_wrangle$Test

#> Fit lasso model ----
# Day
```

```
# Cp LASSO
lasso_cp_day <- lasso_cp(train_dat_day, test_dat_day)

plot(lasso_cp_day$Plot)

# CV LASSO
lasso_cv_day <- lasso_cv(train_dat_day, test_dat_day)

plot(lasso_cv_day$Plot)

# Hour

# Cp LASSO
lasso_cp_hour <- lasso_cp(train_dat_hour, test_dat_hour)

plot(lasso_cp_hour$Plot)

# CV LASSO
lasso_cv_hour <- lasso_cv(train_dat_hour, test_dat_hour)

plot(lasso_cv_hour$Plot)

#> KNN model ----

# Day

# Cp func
knn_cp_day <- knn_cp(train_dat_day, test_dat_day)

# CV func
knn_cv_day <- knn_cv(train_dat_day, test_dat_day)

# Hour

# Cp func
knn_cp_hour <- knn_cp(train_dat_hour, test_dat_hour)

# CV func
knn_cv_hour <- knn_cv(train_dat_hour, test_dat_hour)


perf_mat <- data.frame(Method = c(rep("LASSO", 4), rep("KNN", 4)),
                       Data = rep(c("day", "day", "hour", "hour"),2),
                       Selection = rep(c("Cp", "CV"), 4),
                       "Best Tuning Parameter" = c(lasso_cp_day$`Best Lambda`,
                                                   lasso_cv_day$`Best Lambda`,
                                                   lasso_cp_hour$`Best Lambda`,
                                                   lasso_cv_hour$`Best Lambda`,
                                                   knn_cp_day$`Best K`,
```

```
                                                knn_cv_day$`Best K`,
                                                knn_cp_hour$`Best K`,
                                                knn_cv_hour$`Best K`),
                                MSE = c(lasso_cp_day$`Min MSE`,
                                        lasso_cv_day$`Min MSE`,
                                        lasso_cp_hour$`Min MSE`,
                                        lasso_cv_hour$`Min MSE`,
                                        knn_cp_day$`Min MSE`,
                                        knn_cv_day$`Min MSE`,
                                        knn_cp_hour$`Min MSE`,
                                        knn_cv_hour$`Min MSE`))

xtable::xtable(perf_mat, caption = "Performance of Best Models", digits = 2, label = "ass_3_
    tab")
```

Code/Assignment 3 code.R