MA5832 – ASSESSMENT 2

Jayden Dzierbicki

Jayden Dzierbicki
Jayden.dzierbicki@my.jcu.edu.au

Contents

Part I: An analytical Problem

Scatter Plot of training data

Data was loaded into R, and the following graph was produced with ggplot2() utilising the following characteristics

- Red colour when Y=1
- Blue colour when Y=-1
- X1 on vertical axis (usually referred to as y-axis)
- X2 on horizontal axis (usually referred to as x-axis)

We observe in figure 1 a clear linear separation between the points, suggesting a linear hyperplane could be appropriate.

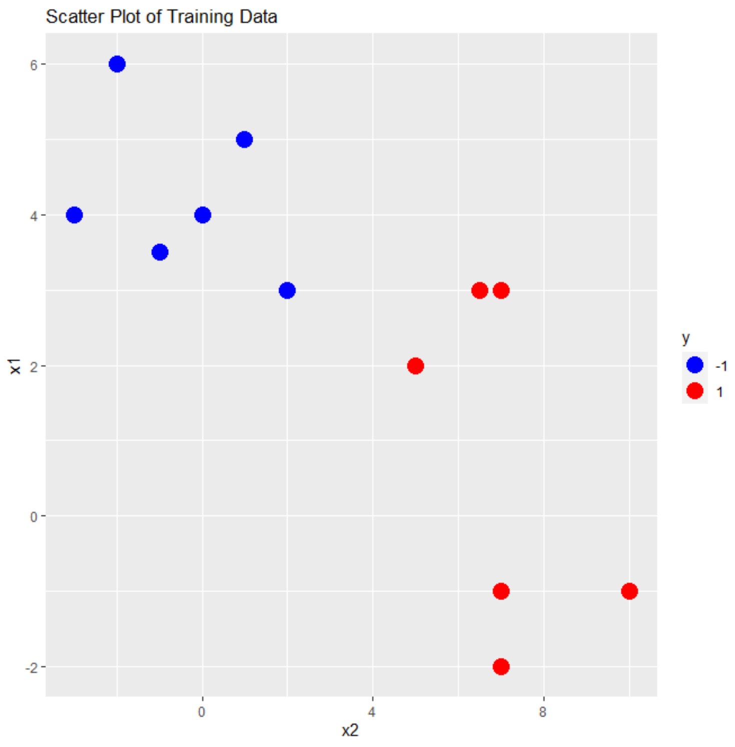


Figure 1: Scatter plot of training data

Find the optimal separating hyperplane of the classification problem using the function solve.QP() and then sketch the output

We provide the following code to solve the problem and plot It our utilising R.

Quadratic programming is an optimisation problem (minimisation or maximisation) that involves quadratic functions. The way R summarise the solve.QP is as:

$$\begin{aligned} \min_b \quad & -d^T b + \tfrac{1}{2} b^T D b \\ \text{s.t.} \quad & A^T b \geq b_0 \end{aligned}$$

Using R we can solve it with the following function:

Solve.QP(Dmat, dvec, Amat, bvec), with the following inputs being described as:

- Dmat: Matrix needing to be minimised
- dvec: Vector to be minimised
- Amat: Matrix defining constraints under which we want to minimise
- Bvec: Vector holding the values of b0

For solving our SVM problem using quadratic programming we can do the following:

dvec = 0

b' = [w,b]

Dmat = [Identity matrix, 0; 0, 0]

Amat = [yx1, yx2, …, -y]

b0 = Bvec = [1,..,1]

Table 1: Code for solve.QP to demonstrate workings

The following code was used to solve this question

```
# Note:
# We know that sole.qp() package offers optimisation for:
# min_b: -d^T*b + 0.5*b^T*D*b
# Such that. A^T*b > b0
#
#
# We can map our problam such that
# d = 0
# b' = [w, b]
# D = [I,0;0,0]
# A^T = [y*x_1, y*x_2, -y]
# b_0 = [1,1,1,...,1]
# Source: https://biodatascience.github.io/statcomp/ml/svm.html


# Spit data into features and target
x <- df[, c("x2", "x1")]
y <- df$y

# Obtain demsnions of x
n <- nrow(x)
p <- ncol(x)

# Create matrix x
x <- as.matrix(df[,2:1]) # Since X2 is on X axis and X1 on y axis we select these positions
y <- c(df$y)

# find number observations
n <- length(y)

# find number varibles
p <- ncol(x)
```

```
# Create Dmat matrix (nxn)
Dmat <- matrix(0, nrow=p+1, ncol=p+1)

# Make D matrix Identity
diag(Dmat) <- 1
Dmat[p+1, p+1] <- 1e-6

# Create D vector
dvec <- matrix(0, nrow=p+1)

# Create A^T matrix
Amat <- t(cbind(x * y, -y))

# Create b vectors (all ones)
bvec <- matrix(1, nrow=n)

# Solve the quadratic problem
qp_output <- solve.QP(Dmat, dvec, Amat, bvec)

# Fitted w and b
w <- qp_output$solution[1:p]
b <- qp_output$solution[p+1]

# Plot model
ggplot(df, aes(y = x1, x= x2,colour= factor(y))) +
  geom_point(size = 5) +
  geom_abline(intercept=(b+1)/w[2],slope=-w[1]/w[2],alpha=.2,linetype=2) +
  geom_abline(intercept=(b-1)/w[2],slope=-w[1]/w[2],alpha=.2,linetype=2) +
  geom_abline(intercept=b/w[2],slope=-w[1]/w[2],linetype=3) +
  scale_colour_manual(values = c("blue", "red"), name = "y",
                  labels = c("-1", "1"))

# Obtain margin value
w_norm <- sqrt(sum(w^2))
margin <- 2/ w_norm
```

Once we solved the QP function we can obtain the values of the coefficients w and b, which can be used to define the decision boundary of the SVC. The vector w is orthogonal to the decision boundary and determines the orientation of the hyperplane, while b is the intercept term which shifts the hyperplane along the direction of w. We obtain two sets of w from the output as one w value corresponds to the case when the closest data point from one class is on the margin, whilst the other set of w values corresponds to the case when the closest data point from the other class is on the margin.

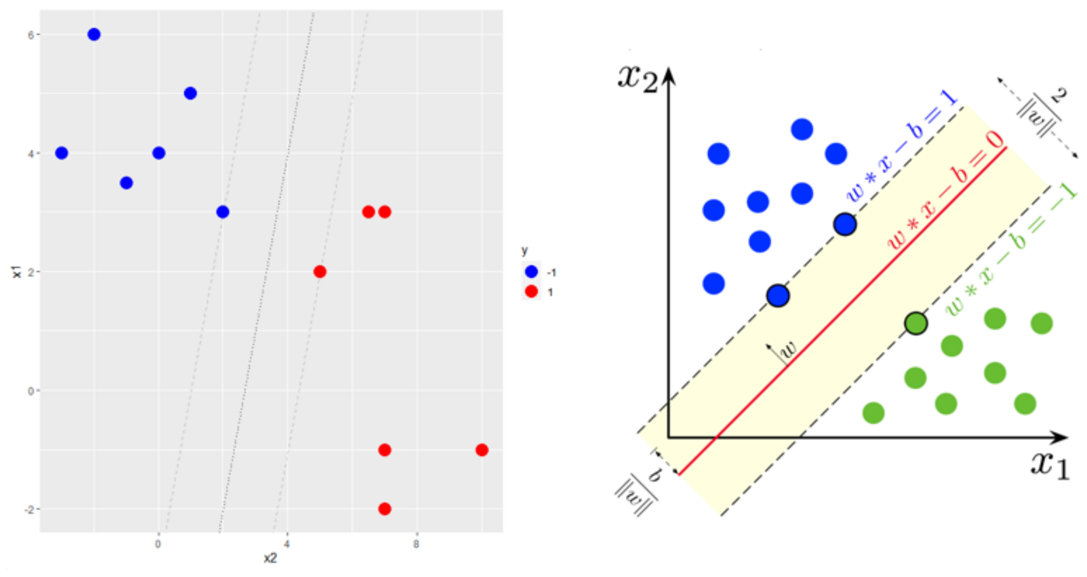


*Figure 2: Scatter plot of training data with hyperplane from solve.qp() (left) vs summary of equation of line w*x-b=0 (right)*

Describe the classification rule for the maximal margin classifier

The conceptual and mathematical framework behind SVM will now be explored to understand how it works and what it is trying to achieve. In the image above it is very clear that we have two features (**p = 2)** which has two distinct classes as indicated by colour, the goal of SVC is to search through the space of **p** dimensional features and construct a (**p-1**) dimensional hyperplane, in our example that would result in 1 hyperplane separating the data points.

As with any problem we can draw an infinite number of hyperplanes which separate the points, though with any machine learning model we have an objective to minimise some loss or cost function. The goal of the SVM is to produce the best hyperplane in a way such that the distance of the hyperplane to the nearest data points on each side is maximised, and we see this demonstrated in the image below (Gandhi, 2018). We refter to this as the maximal margin classifier:

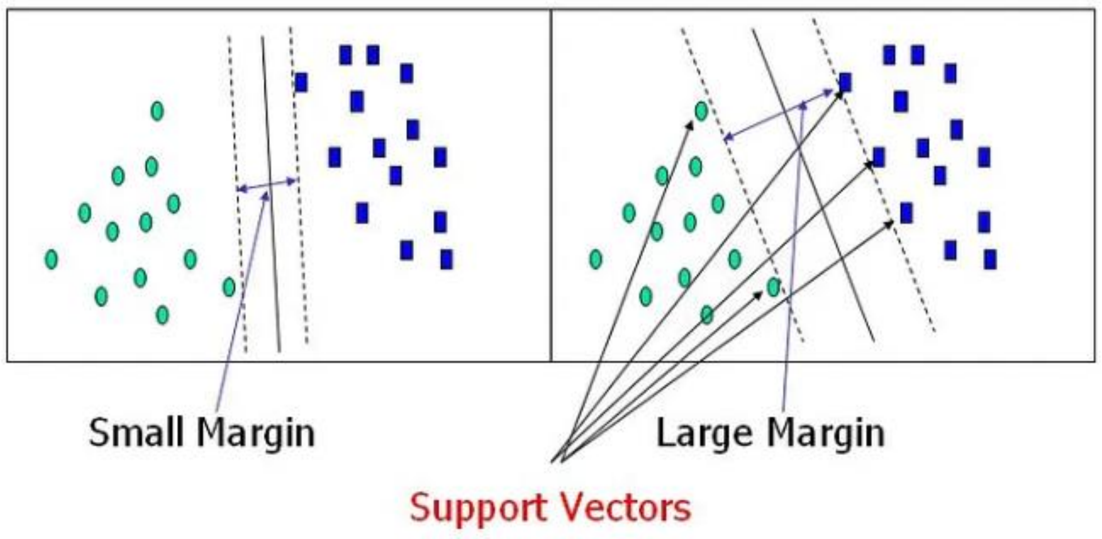


Figure 3: Summary of margin demosntraing small vs long

Before we delve into the maximal margin classifier, we will touch on the foundations of the hyperplane. For our problem we can define our hyperplane with the following equation for two-dimensions:

$$B_0 + B_1X_1 + B_2X_2 = 0 \ldots \boldsymbol{eq\ 1}$$

With $B_0 + B_1 + B_2$ being the parameters.

If $B_0 = 0$

Then the data points belong to the class {1} if:

$B_0 + B_1X_1 + B_2X_2 > 0 \; if \; y_i = 1$ … **eq 2**

Otherwise belong to {-1}

$B_0 + B_1X_1 + B_2X_2 < 0 \; if \; y_i = -1$ … **eq3**

We can represent equation 2 and equation 3 as

$y_i(B_o + B_1X_1 + B_2X_2) > 0$ … **eq 4**

Essentially the hyperplane will divide the data points in p-dimensional space into two halves, in our case two-dimensional space for simplicity. This means that if a separating hyperplane exists, then we can classify a test observation x^*based on the sign of $f(x^*) = B_0 + B_1x_1^* + B_2x_2^*$, with a positive sign being assigned class 1 and a negative sign being assigned class -1 as seen in eq 1 and 2. In addition to the sign, the magnitude of $f(x^*)$ can also be used to determine proximity to the hyperplane, with a larger absolute value indicating being further away relative to an absolute smaller value, this magnitude can then be interpreted as a form of 'confidence' for classification.

The maximal margin hyperplane (also known as the optimal separating hyperplane) is the farthest from the training observations points, meaning that we can calculate the perpendicular distance from each training observation to a given separating hyperplane; the smallest distance is the minimal distance from the observations to the hyperplane and is known as the margin. The maximal margin hyperplane is the separating hyperplane for which the margin is the largest. This can easily be explained by referencing the image below in figure 4, the maximal margin hyperplane is shown as the solid line, the margin is the distance from the solid line to either of the dashed lines. The two blue points and the purple point are the support vectors, and the distance from those points to the margin is indicated by the arrows. The purpose of the support vector is to 'support' the maximal margin hyperplane that if the training set was to change and these points move then the maximal margin hyperplane would also move, noting that the maximal margin hyperplane is un impacted on the other observations in X1 or X2.

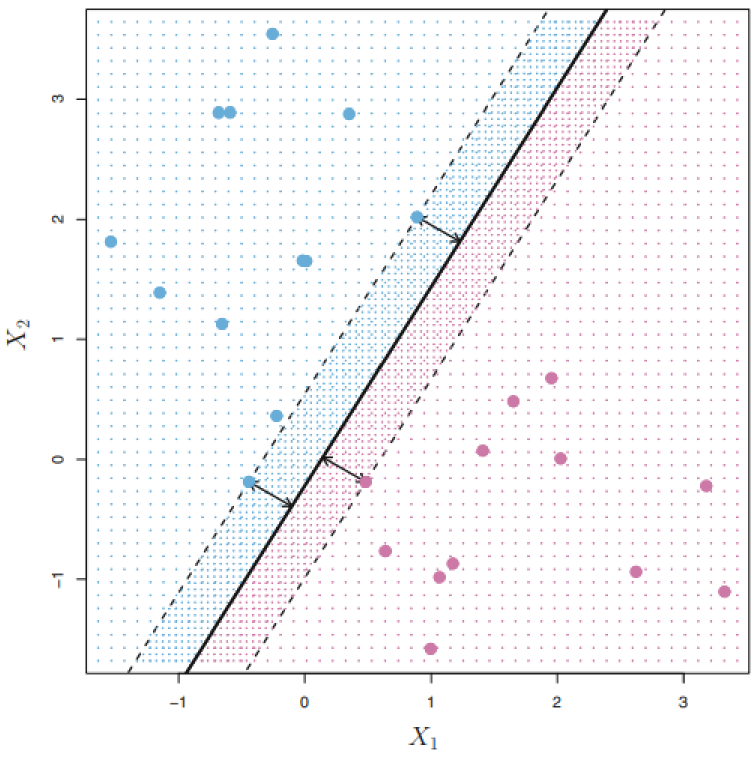


Figure 4: Demsontration of hyperplane, margins and support vectors

When we attempt to construct the maximal margin classifier, we are trying to do the following:

$Maximise_{B0,B_1,B_2} M$ … **eq 4**

Subject to: $B_1 + B_2 = 1$ (We have expended the summation to demonstrate how it relates to 2-dimensinal question on hand) … **eq 5**

$y_i(B_0 + B_1x_{i1} + B_2x_{i2})$ >= M ∀ i=1,…,n … **eq6**

What this tells us is that our constraints equation 5 and equation 6 ensure that each observation is on the correct side of the hyperplane and at least a distance M from the hyperplane, with M representing the margin of our hyperplane, and the optimisation problem choses a $B_1 \; and \; B_2$ to maximise M. We also highlight that we can expend our above equations to include p-dimension space by the more generalised equation not shown here, though to include B_p.

Calculate the margin of the classifier

We described the margin of the classifier in the previous section as the distance between the line and the closest data points on each side. A larger margin tells that the line is a better separator as it has a wider gap between the categories, resulting in possible higher accuracy due to a reduction in misclassifying data points. Whilst a smaller margin, indicates the line is not a good separator, as it is to close to the data points and could result in misclassification.

For our question we obtained a margin of 3.162278, this means that the maximal margin classifier is 3.16 units away from the closes data point on each side. We utilise the equation seen in figure 2 (right image) which is: $margin = \frac{2}{||w||}$, where $||w|| = sqrt(sum(w))$ obtained in R through the solve.QP() function.

Part II: An Application

2.2.1 Data

Prior to conducting our analysis, the following data checks & pre-processing steps occurred, a crucial step for any machine learning model:

- Removal of row 1, associated with descriptive headings
- Removal of column 1, associated with record ID
- Test for duplications, no duplicated records observed
- Testing for NA observations, no NA observations observed
- X3 re-encoded as duplication of 'unknown' (0,5 & 6 all encoded as unknown)
- X3 and X4 re-encoded to group other and unknown together, for the purpose of analysis unknown and other can be thought of the same thing, though we could argue they are different and will list this as a limitation of the analysis. Though whilst a possible limitation this grouping is consistent with Yeh & Lien's (2009) comparative study which appears to have grouped 'other' and 'unknown' together as 'others' for X3 and X4

We then looked at the data types, observing all data was imported as a character via the summary() function. We propose the following transformations based on the data description provided.

Table 2: Data processing step as when imported all variables were in character format

Variable	Conversion	Justification
X1	As.integer()	# NT $, whole number
X2	As.factor()	# Gender, as factor
X3	As.factor()	# Education, as factor
X4	As.factor()	# Maritial status, as factor
X5	As.integer()	# Age, as whole number
X6	As.factor()	# Hist payment, as factor
X7	As.factor()	# Hist payment, as factor
X8	As.factor()	# Hist payment, as factor
X9	As.factor()	# Hist payment, as factor
X10	As.factor()	# Hist payment, as factor
X11	As.factor()	# Hist payment, as factor
X12	As.integer()	# Amount of bill, as whole
X13	As.integer()	# Amount of bill, as whole
X14	As.integer()	# Amount of bill, as whole
X15	As.integer()	# Amount of bill, as whole
X16	As.integer()	# Amount of bill, as whole
X17	As.integer()	# Amount of bill, as whole
X18	As.integer()	# Amount of previous payment, as whole
X19	As.integer()	# Amount of previous payment, as whole

X20	As.integer()	# Amount of previous payment, as whole
X21	As.integer()	# Amount of previous payment, as whole
X22	As.integer()	# Amount of previous payment, as whole
X23	As.integer()	# Amount of previous payment, as whole
Y	As.factor()	# Default, as factor

For reproducibility we utilised the set.seed() function and conducted analysis utilising R Version 4.2.2. The data was then split utilising the createDataPartition() function at a level of 70%, and data was allocated to a training data set and a testing data set used for validation. Based on the data we imported above we observed that the goal was to correctly classify customers as either defaulting (1) or not defaulting (0) on payment. With this in mind, we proposed the development of three classification models, (1) random forest, (2) bagging and (3) classic classification tree. As we see below each model will utilise the following training data set which consists of 70% of the total data, with 21001 rows and 24 observations (23 explanatory and 1 response). We utilised all the data inputs and did not omit any features through feature selection, consistent with Yeh and Lien (2009) comparative study; though feature selection is common in machine learning problems (Sahithi et al., 2022; Jemima Jebaseeli et al., 2020).

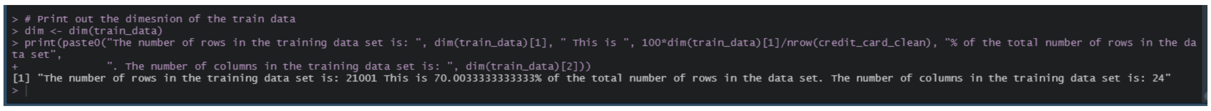

```
> # Print out the dimesnion of the train data
> dim <- dim(train_data)
> print(paste0("The number of rows in the training data set is: ", dim(train_data)[1], " This is ", 100*dim(train_data)[1]/nrow(credit_card_clean), "% of the total number of rows in the da
ta set",
+              ". The number of columns in the training data set is: ", dim(train_data)[2]))
[1] "The number of rows in the training data set is: 21001 This is 70.0033333333333% of the total number of rows in the data set. The number of columns in the training data set is: 24"
> 
```

Figure 5: Print out of dimensions

Finally, we undertook some basic data analysis to understand any issues which we might experience on the training data set and observed that our training data set is imbalanced..

In addition, we produced several plots to see if we can observe any obvious discrimination of our numeric predictors on Y, it appears variables such as X1 and X5 have some obvious visual elements which could indicate that these predictors are impacting the likelihood of falling into either Y=0 or Y=1.

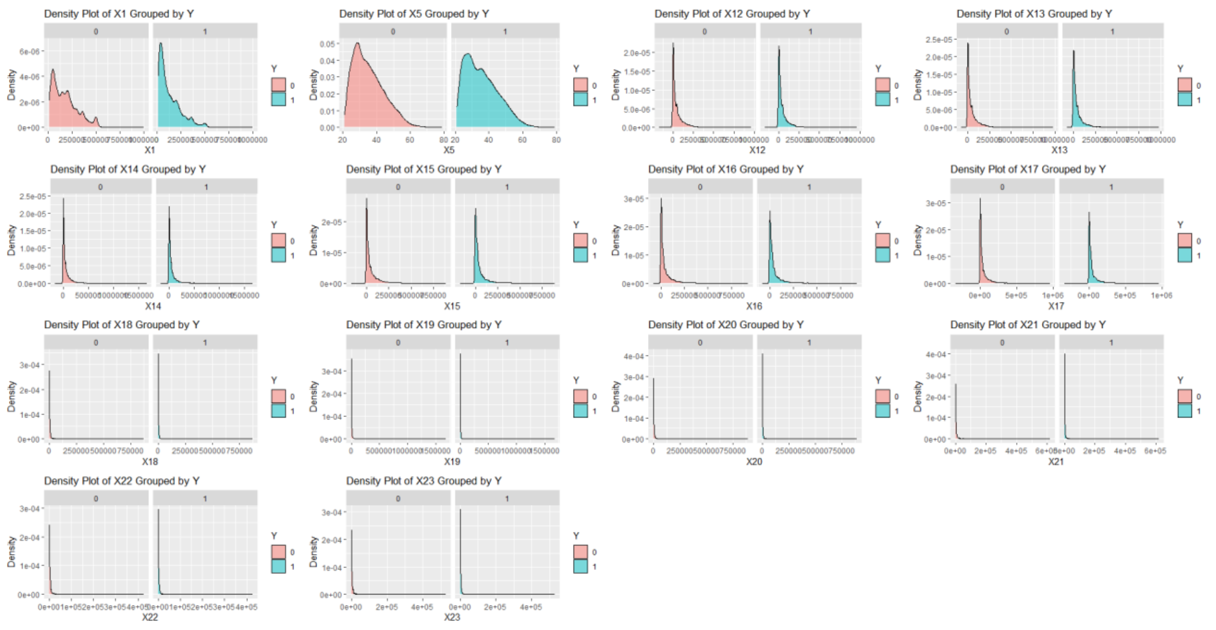


Figure 6: Density plot of continuous variables

Furthermore, we performed the same analysis for our categorical variables, and observed that several predictors such as X6, X7, X8, X10, and X11 appeared to have a relationship with the likelihood of falling into either Y=0 or Y=1, as evidenced by the bar plots below.

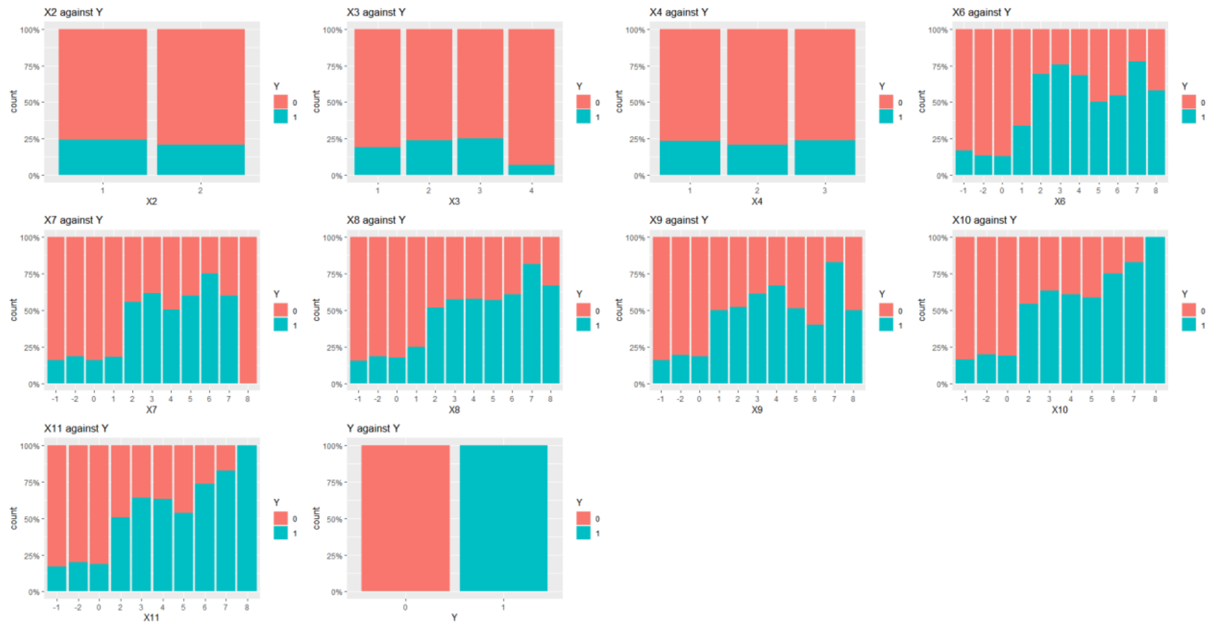


Figure 7: Distribution of response type for non-continuous variables

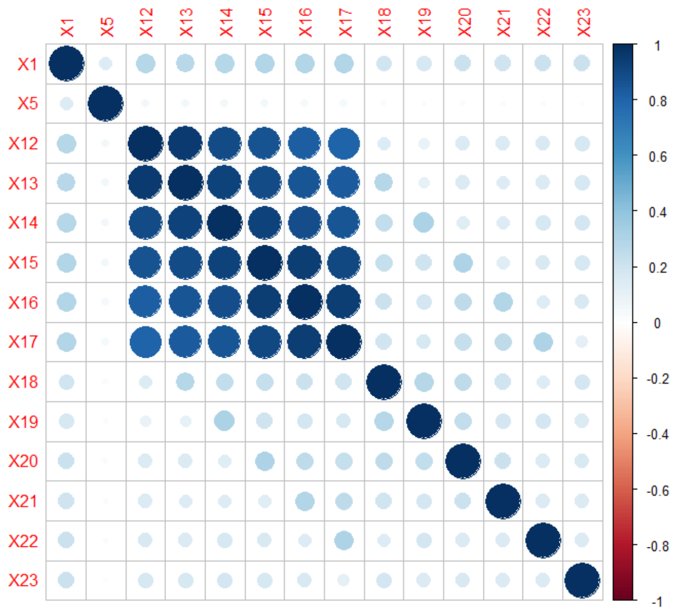
2.2.2 Tree Based Algorithms – Selection Random Forest (Note to marker: I have merged discussion of model selection based on performance of training data, please refer to multiple sections for understanding)

As discussed above in the data section we are limited to algorithms which are suitable for classification problems, this can include:

- Classic classification tree
- PRIM-Bump hunting
- Bagging
- Random Forests
- Boosted trees (AdaBoost.M1)

When selecting which model to use there is no simple answer, as each model comes with a list of merits and limitations, and a common approach is to even compare various models and methods (Sahithi et al., 2022; Jemima Jebaseeli et al., 2020). We decided to consult the literature for common tree-based methods and found various studies which looked at credit card data which employed random forest (Jemima Jebaseeli et al., 2020), and other credit card studies looking at bagging and boosting (Sahithi et al., 2022). Due to time constraints, we focused our analysis on random forest, bagging and classic classification tree as these are some of the most common and well-known methods employed; in addition this will allow us to talk about the evolution of the random forest which is built upon the preceding two models. At this stage it would be deemed immature to select one model, as a good data scientist would aim to evaluate many models as this seems to be best practise and a common method used as reported in the literature (Sahithi et al., 2022; Jemima Jebaseeli et al., 2020).

Due to multiple variables in our data set being correlated as seen in the heatmap below we elected to procced with the random forest model as it is better suited to handling this type of correlation amongst features as it utilises a bootstrap approach and considers a subset of predictors at each split instead of the whole feature space, reducing the impact of multicollinearity (James, 2017). In addition, overcomes the limitations of the classic classification method which we discuss below. In addition, random forest also produced the best metrics on the training data set compared to the other models we explored using the default settings in R prior to any hyper-parameter tunning.



Random Forest Model (Including: Assumptions, hyper-parameters used, model summary and variable discussion)

The random forest model employs additional algorithms to improve the predictability of classification models, therefore it is important to understand that the random forest is built upon the classification tree and bagging. For us to understand the model we will briefly introduce classification tree and bagging.

Classification tree

The classification and regression tree are a form of supervised machine learning which are very popular due to their flowchart like structure, making it easy for many people to interpret. The structure of a decision tree includes (1) root node, (2) internal/parent node and (3) terminal/child nodes, with the root node being the point where the data starts dividing (James, 2017).

In the image below we can observe the root node contains both green and purple circles, as we progress our way down the goal of the classification tree is to partition those green and purple circles correctly by segregating them into J distinct and non-overlapping regions (Ri). For every point in a unique region will have the same prediction & classification. We also observe that we have increase node purity as we propagate our way down the classification tree, an important topic we will highlight further below.

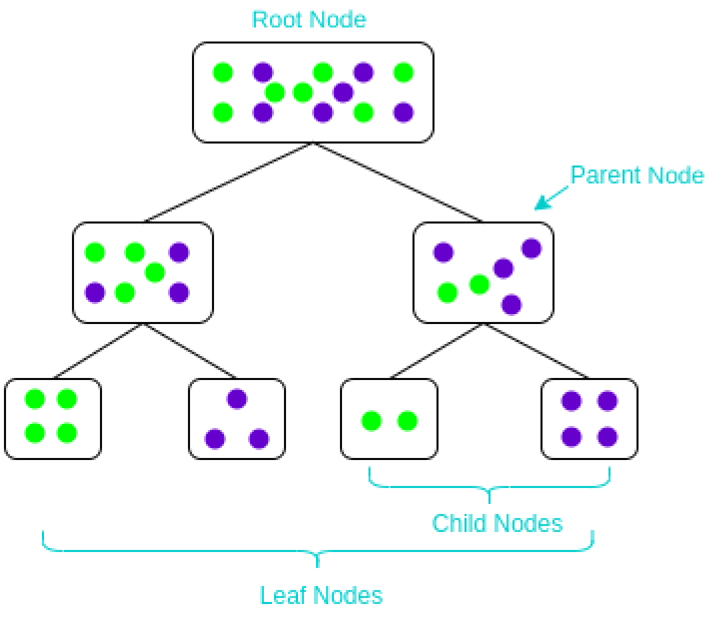


Figure 8: Simple classification tree highlighting structure and purity

The prediction of a region's classification is determined by the mode of all y in subspace Ri, using a majority rule approach, that is the most common observation within a region is used to make a prediction for any new observation that falls within that region in space. This is represented mathematically as:

$$k(m) = \underset{k}{\operatorname{argmax}}(\hat{p}_{mk})$$

$$\hat{p}_{mk} = \frac{1}{N_m} \sum_{x_i \in R_m} I(y_i = k)$$

The goal of the decision tree is to split the data into distinct regions which are as pure as possible, with the Gini index being a favourite measure for classification trees and the one we will focus on which is summarised as (James, 2017), the Gini index is applied at the terminal node and we will see it again when we talk about random forest:

The *Gini index* is defined by

$$G = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk}),$$

A smaller Gini index is associated with a purer node, that is a node contains predominantly observations from a single class. Gini index is just one cost function we can explore, with others including cross entropy and misclassification error; with both Gini index and cross-entropy being differentiable allowing for numerical optimisation and more sensitive to node probability which allows them to be used in growing trees (James, 2017).

Limitations of classification trees (James, 2017):

- Missing data needs to be addressed, various methods have limitations and strengths and should be addressed on a case-to-case basis
- Small changes in the data can result in different trees, thus different training data sets can result in different classifications trees/models

Bagging and Random Forest

We discussed that classification trees are known to be subject to instability due to dependency on the training data set. To overcome this limitation bagging and random forest techniques are used to produce a number (B) of trees using bootstrapped samples and combing the estimates of all the trees (B) produced. In the image below, the training data is split into smaller bootstrapped samples with replacement and a subsequent tree model is produced. Each bagged tree will use a percentage of the training data and the reaming data not used is referred to as out-of-bag observations/samples (James, 2017):

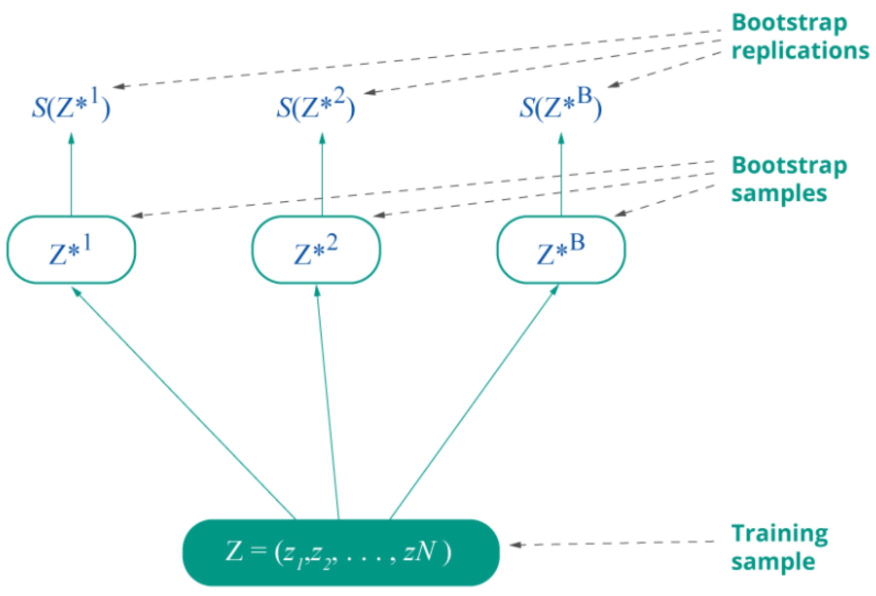


Figure 9: Bootstrapping sample method employed by bagging and random forest model

Limitation of bagging:

- Does not produce a single tree, though the algorithm does produce a summary of the Gini index we discussed before to highlight variable importance. This allows us to add up the total amount that the Gini index is decreased by splits over a given predictor, averaged across all B trees to highlight variable importance
- Considers the whole feature space which can lead to correlated trees
- A bagged tree is not a tree

Random Forests

The random forest model is like the bagging technique, but with one crucial difference in the number of features used in the model for each bootstrapped sample. In bagging, P features are taken, while in Random Forest, a smaller number of m features, where m<P, is chosen. The square root of P is typically considered as the value of m, but this hyperparameter can be fine-tuned (James, 2017). By doing so, the strong predictors are not given more weight in the model, allowing other predictors to also be considered. On average, (p-m)/p of the splits do not include the strong predictor (James, 2017).

The way bagging/random forest determines the final class can be summarised as follows (James, 2017):

1- For b = 1,…,B
   a. Draw a bootstrapping sample, Z* (fig 9) of size N from the training data (with replacement)
   b. Grow a random-forest tree Tb to the boostrappted data by recursively repeating the following steps for each terminal node of the tree, until the minimum node side is reached
      i. Select m variables at random from p variables (if m=p then bagging)
      ii. Pick the best variable/split point among the m
      iii. Split the node
   c. Output of ensemble of tree

2- Classification is done by aggregating the predictions of all the trees in the ensemble using majority vote which can be represented as

$$\hat{C}_{rf}^{B}(x) = \text{majority vote}\{\hat{C}_b(x)\}_1^B$$

Assumptions of Random Forest include (James, 2017; M, 2023):

- The number of trees (B) is not a critical parameter, large value of B tends not to lead to overfitting, though can result in a more robust model; though this increases computational cost
- At each step of building the tree, the best split of data is found to maximize the accuracy of the model.
- The model assumes that the distribution of the target variable classes is roughly equal across the training data set, to avoid the model becoming biased towards any one class
- Each decision tree in the random forest is assumed to be independently generated from the others

Limitation of random forest:

- The algorithm does not produce a single tree, but instead, it produces a summary of the Gini index, which is used to highlight the variable importance. This allows us to add up the total amount that the Gini index is decreased by splits over a given predictor, averaged across all B trees to highlight the variable importance.
- Can be computationally demanding, as number of predictors or observations grow
- The random forest algorithm requires tuning of certain hyperparameters such as the number of trees and the number of features considered in each tree

Hyper-parameter selection

To refine our model, we elected to undertake hyper-parameter tunning and selection. We highlight that we did not modify the default number of trees (500 in R) as it is generally not considered a crucial hyper-parameter as previously highlighted (James, 2017); though if time persisted, we could have included it in our hyper-parameter selection.

We selected the optimal number of hyper parameters via grid search approach utilising the caret package and the train function. We compared the grid of the following values: square root of p, p/2 and p/4. After conducting the grid search, we determined the optimal number to use was 6, round up from 5.75 as opposed to the default square root of p (James, 2017).

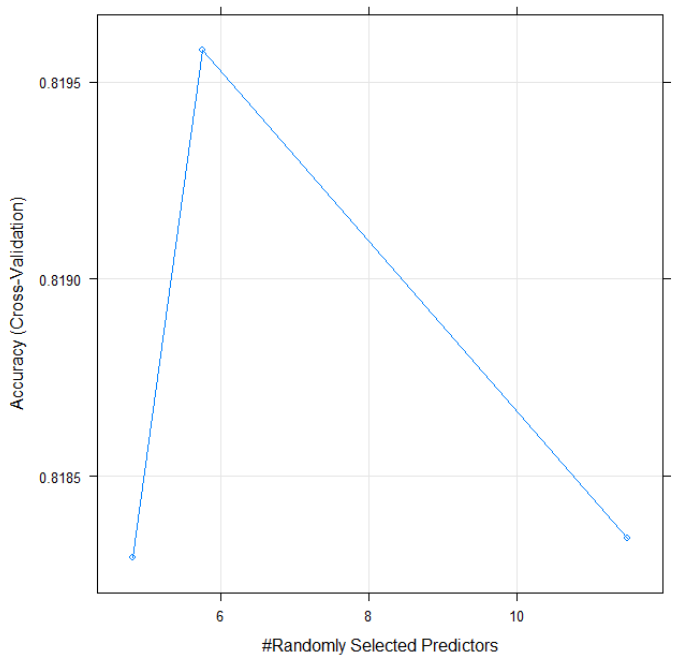


Figure 10: Demonstration of hyper-parameter selection, model suggest that 6 is the optimal number of parameters in our final random forest model

It is worth noting that this approach took a total of 58 minutes due to the inclusion of cross-fold validation. An alternative approach could have been to toggle the number of features in the randomForest() function and selecting the best performing models based on metrics, though this allowed us to automate the hyper-parameter tunning.

Variable Importance

Mean Decreased Accuracy: This metric provides an estimate of the reduction in prediction performance that occurs when a particular variable is excluded from the bootstrapped sample. We observe the following top 3 variables which have the greatest impact on the prediction accuracy.

- X6: Repayment statues in September 2005
- X15: Bill amount 4
- X14: Bill amount 3

Mean Decreased Gini: This metric is related to the cost function used in classification trees, specifically the Gini index. The mean decreased Gini coefficient measures the contribution of each feature to the homogeneity of each node. We observe the following top 3 variables which have the greatest impact on the gini index:

- X6: Repayment statues in September 2005
- X5: Age
- X12: Bill amount 1

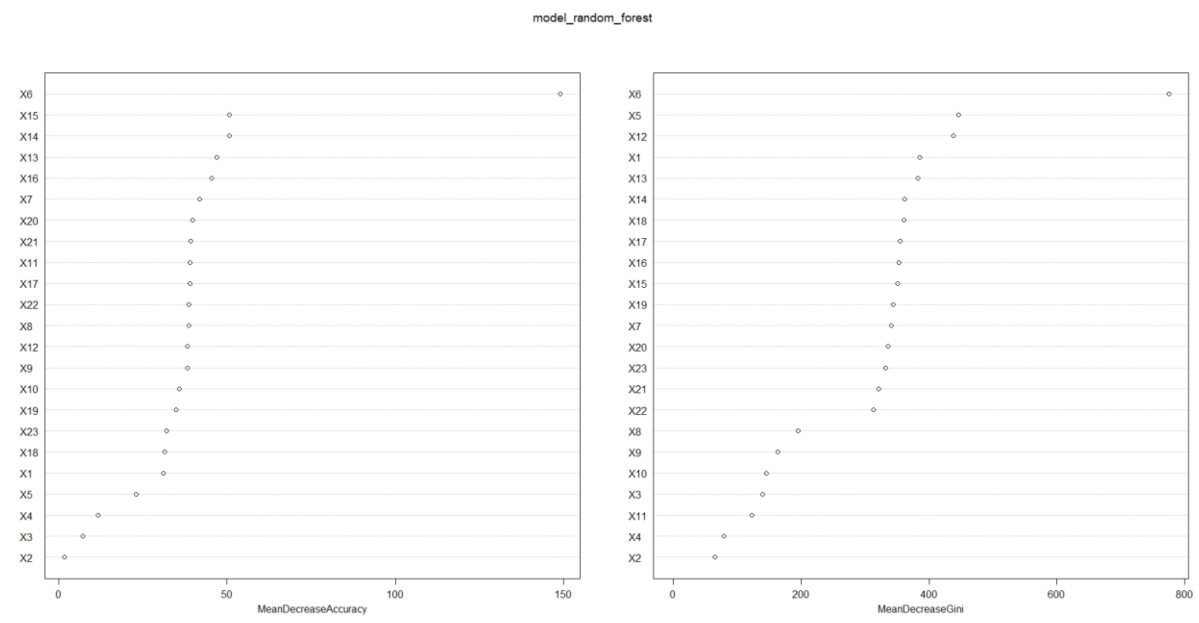
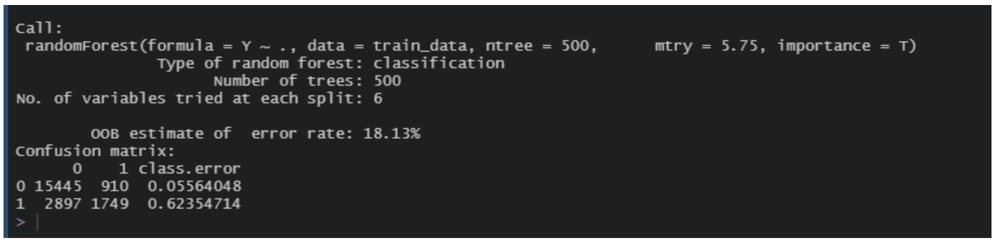


Figure 11: Plot of variable importance .

Both these metrics suggest that X6 is the most influential variable on the model, though after that there is discrepancy amongst variable importance relative to each metric. This suggest that X6 contributes the most to both the accuracy of the model and the homogeneity of nodes, through other variables have different influences on the model. An obvious example is X1 (limit balance), which has a relatively high impact on node purity, but has little relative impact on model accuracy as seen by being at opposite ends of each graph in figure 11.

Model Summary

Based on the model summary below for the random forest model we know that the algorithm used the default number of trees which was set to 500, considering 6 variables at each split. The out of bag (OOB) error rate was estimated to be 18.13% for our model using our out-of-bag sample. The confusion matrix shows the true positives (TP), true negatives (TN), false positives (FP) and false negatives (FN) for each class, in our model we observe error rate of 5.5% for class 0 and 62% for class 1. Suggesting that our model is relatively better at predicting people into class 0 then class 1.



```
Call:
 randomForest(formula = Y ~ ., data = train_data, ntree = 500,      mtry = 5.75, importance = T)
               Type of random forest: classification
                     Number of trees: 500
No. of variables tried at each split: 6

        OOB estimate of  error rate: 18.13%
Confusion matrix:
      0    1 class.error
0 15445  910  0.05564048
1  2897 1749  0.62354714
>
```

Figure 12: Model summary for random forest

Justification for Model Choice & Model Comparison & Evaluation on the training **(combination of 2.2.2.a & 2.2.2.c: As complimentary discussion)**

We selected to procced with the Random Forest model based on both the literature and performance on both the training data relative to the other models. As seen in the table below we have metrics for each model in relation to performance on training data set. When deciding which metric to select when determining the optimal model there is no one size fits all approach, as the metric selected will depend on the need being addressed by the model.

One thing which stands out across all the models, is that the Random Forest possibly overfits the model, given the high level of accuracy in the training data set relative to the other models. Though we observe the OOB error rate is consistent to the error rate of the other models; suggesting that whilst it performs quite well on seen data, it does not perform as well on unseen data which is calculated and aggregated using the out-of-bag sample; the OOB error rate is similar to that seen in the paper from which the data is sourced from (Yeh & Lien, 2009) which was reported to be around 0.18 for both training and validation for classification trees in the paper. Overall, the summary of performance of the model on the training data set can be summarised in the following way from table 3:

- Out-of-bag (OOB) estimate of error rate: 18.13% - This is an estimate of how well the model is likely to perform on new, unseen data
- Sensitivity/recall: 0.9845 - This measures the proportion of true positive cases that are correctly identified by the model
- Specificity: 0.9998 - This measures the proportion of true negative cases that are correctly identified by the model
- Positive predictive value/precision: 0.9991 - This measures the proportion of cases that are predicted as positive that are actually positive
- Negative predictive value: 0.9956 - This measures the proportion of cases that are predicted as negative that are actually negative
- Prevalence: 0.2212 - This is the proportion of positive cases in the dataset
- Detection rate: 0.2178 - This measures the proportion of actual positive cases that are correctly identified by the model
- Detection prevalence: 0.218 - This measures the proportion of predicted positive cases out of all cases predicted by the model

- Balanced Accuracy: 0.9921 - This takes into account the imbalance in the dataset between the two classes, and provides an overall measure of the model's performance
- Accuracy: 0.9964 - This measures the proportion of all cases that are correctly classified by
- F1 Score: 0.991746 – This is a measure which takes into account both precision and recall and is useful on unbalanced data sets

Table 3: Summary of model performance, highlighting that Random forest excel in recall, precision and F1-score

Model	**Simple Classification**	**Random Forest**	**Bagging**
Metric	Training	Training	Training
OOB estimate of error rate		18.13%	18.48%
Sensitivity/recall	0.32609	0.9845	0.38592
Specificity	0.95946	0.9998	0.93941
Pos Pred Value/precision	0.69559	0.9991	0.4404
Neg Pred Value	0.83366	0.9956	0.84339
Prevalance	0.22123	0.2212	0.22123
Detection Rate	0.07214	0.2178	0.08538
Detection Prevalence	0.0371	0.2180	0.13257
Balanced Accuracy	0.64277	0.9921	0.66267
Accuracy	0.8193	0.9964	0.817
F1-Score	0.44402346	0.991746	0.411364

When selecting which model is optimal it is crucial to consider the processing time of the model, as this can be an added cost both computationally and nominally for an organisation. What we see in the table below is the run time for each model produced, as expected the simple classification performs quite fast; whilst random forest and bagging take more than one and two minutes respectively. In the realm of big data this can pose a challenge in model training. Whilst in the scheme of things our training data set was relatively small, comprising of only 22,001 rows and 24 columns (528,024 objects) this can become a real problem when dealing with big data and is something to be wary of when proposing and training a model. The reason that random forest is faster than bagging would be due to a smaller p value associated with number of predictors.

Table: Computational run time for various model, for direct comparison of random forest and bagging we set ntrees equal to 500. With the concept of big data, the need to understand computational processing time it a critical factor in determining which model we might select.

Model	Run time
Classification	0.894486seconds
Random Forest	1.112431 mins
Bagging	2.118453 mins

What model should we use?

This ultimately comes down to the specified requirements and constraints we might be faced with, thus a possible trade-off between accuracy and computational costs, as well as potential for overfitting and sensitivity to changes in the training data set; in addition, it is also important to consider the underlying data and any strengths, limitations and assumptions of the model. The simple classification model whilst performs quite well across the various metrics in the above table may be susceptible to changes in the training data, and as the data set grows this could come an issue with different training sets used. The Random Forest model has a low error rate and overcomes the issue of classification tree such as being susceptible to change, though we have the possibility of the model overfitting the training data set given its performance on seen data relative to unseen as evident by the training metrics. The bagging model also performs quite well across various metrics, though has increased computational costs compared to any other model tested, almost double the processing time of random forest. This poses the issue of what if we have the inclusion of more rows or predictors as this could see an increase in computational cost when training. We also highlight that our error rate appears consistent with Yeh & Lien (2009) at around the 18% mark for training and validation, though our approach would have been different. Overall, the random forest is a popular model which is heavily reported in the literature, as well as exceling in almost all the metrics and having a better OOB error, as such we elected to procced with the random forest model for our comparison with SVM, in addition we also considered that bagging model took twice as long to run with a worse OOB accuracy relatively to Random forest & did not elect the classic method due to susceptibility to change. Another key highlight of the random forest model is its ability to handle features which are correlated together, which we demonstrated with the correlation map amongst features.

2.2.2 SVM – Selection SVM with kernel (Note to marker: I have merged discussion of model selection based on performance of training data, please refer to multiple sections for understanding)

Prior to conducting the final train on SVM we scaled the data for all continuous variables, SVM is based on distance, so it is an important step which should be included as if one feature has large values it has the potential to dominate the other features when calculating distance (Tokuç, 2022). This was not required for random forest. We have described the support vector classifier in detail at the start of the assignment and will not repeat what we have previously discussed. Though now we will talk about the support vector machine which is an extension of the support vector classifier that results from enlarging the feature space in a specific way using kernels which we will describe below. The goal of SVM is to accommodate for non-linear boundaries between classes (James, 2017), we hypothesis that our data set would not have clear linear boundaries, and we can demonstrate this by plotting some of the features (figure 13) highlighting the data points are indeed heavily integrated with each other with no clear linear separation in 2-D. Whilst not shown here we did the same on all the continuous variables and the same issue was observed in 2-D and we assume this would be true for n-dimension; thus we have assumed that the same problem would persist in higher dimension, though this is not always true and we did not explore it further. We ruled the out the use of a linear classification, highlighting the need to utilise a SVM with kernel.

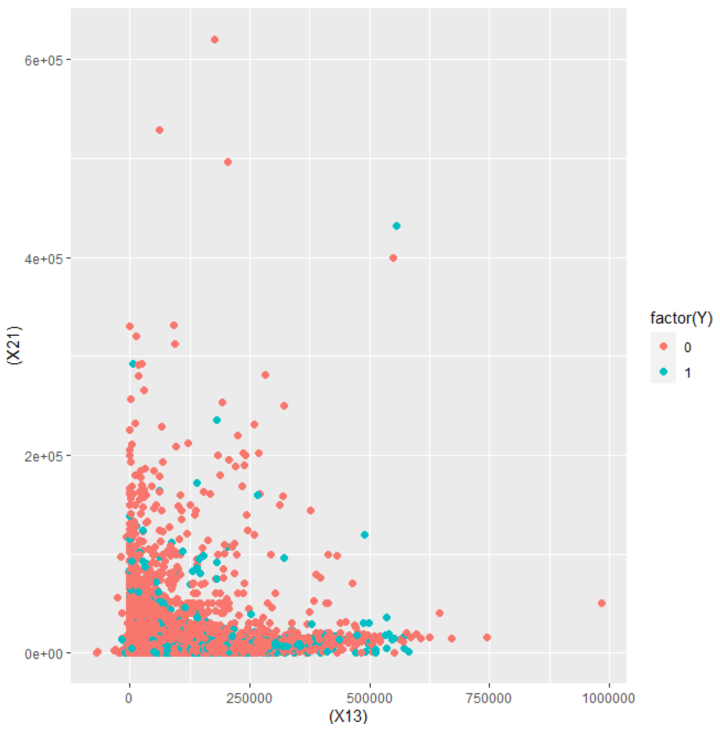
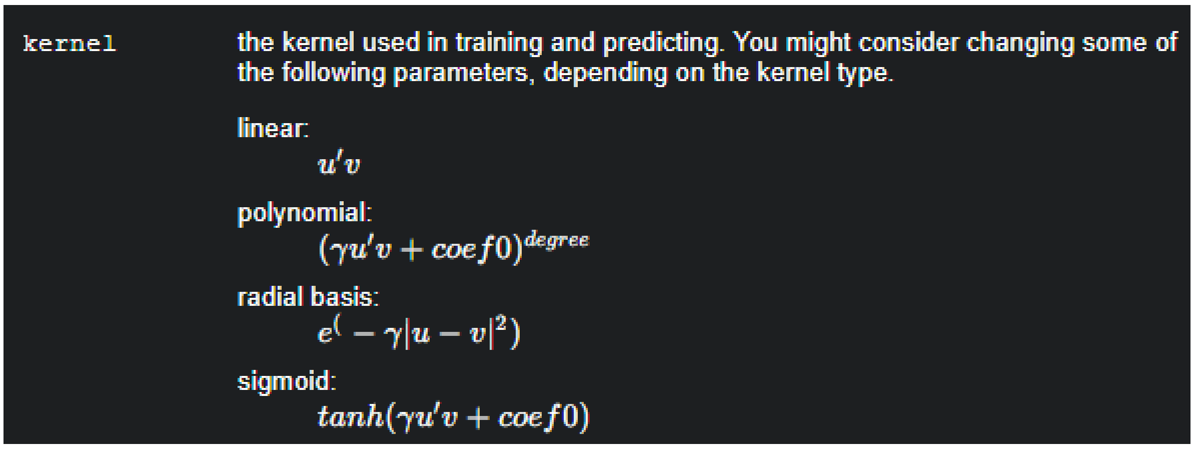


Figure 13: Simple demonstration the data points are heavily integrated in 2-D and that linear boundaries are not clear.

Looking at the SVM() function in R we have the following kernel hyper-parameters (figure 14) to select from when designing and building our model, and based on our previous discussion above we have elected to exclude the linear kernel.


kernel the kernel used in training and predicting. You might consider changing some of the following parameters, depending on the kernel type.

linear:
$u'v$

polynomial:
$(\gamma u'v + coef0)^{degree}$

radial basis:
$e^(- \gamma|u - v|^2)$

sigmoid:
$tanh(\gamma u'v + coef0)$

Figure 14: Summary of kernals availibe in svm function (R Documentation, Support Vector Machines*)*

With that in mind we elected to procced with a kernel/non-linear SVM as our underlying model. Though selecting the best kernel posed new challenges compared to Random Forest, particularly in terms of computational requirements in the form of undertaking a grid search on various costs, game and kernel hyper-parameters. Though, by observing the data we suggested that the radial basis kernel would be best suited for this process. The process of selecting and training the model was time-consuming, and hyperparameter selection was more complex, as there are various kernels we could explore. The three main hyperparameters considered were:

- Cost: In the SVM function cost is the cost of constraint violation, representing the 'C' constant of the regularization term in the lagrange formulation (R Documentation, Support Vector Machines). A low cost means that the SVM is more tolerant to misclassification and will allow some data points to be misclassified, whilst a high-cost value means the SVM model will try to classify all the training data points correctly. Essentially a trade of between achieving a low training error and a low testing error.
- Gama: Parameter needed for all kernels except linear. A low gamma means point furthest from the hyperplane are considered and influence the model, whilst a high gamma only considers points closes to the hyperplane.
- Kernal: The kernel allows data to be transformed into a higher dimensional space where it can be more easily separated or analysed; allowing us to compute complex non-linear decision boundaries that cannot be obtained via SVC, we describe the 4 kernels available in SVM function in figure 14

Tuning the hyperparameters using 70% of the data was computationally intensive, and even overnight processing was not successful with our machine to test all the required hyper-parameters, especially when undertaking a gird-search approach which can take hours, days or even weeks to fine tune, with more advance methods beyond a grid search available which we did not explore (Fayed & Atiya, 2019). To overcome this limitation, we decided to select the hyperparameters using a smaller dataset (10% or 1501 observations) before training the full dataset with the selected hyperparameters. This approach has both limitations and benefits, which were not explored in detail, but we discuss briefly below; Though it has been documented that other people have also reported utilising this approach given the computational resources required to undertake hyper-parameter tunning on the whole data set for SVM models given the sub-sample is sufficiently large (Marsupial, 2016).

As a thought experiment we decided to produce two SVM models: one using the default hyper-parameters of the SVM() function and another using the hyper-parameters selected through a grid search with the tune() function as discussed above to highlight if a grid search on a smaller subset of data lead to enhanced results. It was hypothesized that the accuracy on the training dataset would be slightly improved with the grid search approach, despite being tuned on a smaller subset of the data; we assumed that 10% was sufficiently large enough (Marsupial, 2016). This demonstrates that whilst hyper-parameter tuning is a crucial step, it can be a limitation of a ML model depending on available resources and highlights a trade-off between efficiency and accuracy, especially when a limitation of SVM is big data sets. In addition, another concern was that SVM can be heavily influenced by data points along the margin, and as such a smaller training set of 10% may not accurately reflect the distribution of data points along the margin, and as a result the chosen hyper-parameters may not be optimal for the full data set if our assumption was incorrect (Marsupial, 2016).

Limitations (Support Vector Machine in machine learning 2023):

- SVMs are not suitable for large data sets due to training and memory requirements associated with larger data sets
- SVMs may not perform if the number of features is larger than the number of samples, “curse of dimensionality”
- The choice of kernel can greatly affect the performance of an SVM, and it can be difficult to determine the best kernel for the given data set
- The gamma variable and cost tunning parameter are subjective, with smaller C values making the classifier prone to very high sensitivity in the training data which can lead to higher classification errors in test data

- An SVM model may not be an ideal classifier if the features of the data are dynamic in space or time

Hyper-parameter selection:

Unlike with Random Forest, we could not undertake an extensive hyper-parameter selection on the complete training data set, which was 70%. Instead, we elected to do hyper-parameter tunning on just 10% of the data set to demonstrate how we could do it if we had the computational resources needed.

We utilised the following code to conduct a grid search using the tune() function of the e1071 package, this took in total around 2 hours on just 10% of the data set. We attempted 70% of the data, though this was not a success and time was not a luxury we had.

Hyper-parameter tunning code on 70%

```
# Define a list of cost parameters to try
gammas <- c(0.01, 0.1, 1, 2) # Test gama
costs <- c(seq(0.001, 2, length = 10))

# Train the model using the tune() function and cross-validation
set.seed(123)
start_time <- Sys.time()
model <- tune(svm, Y ~ ., data = svm_pre_train, ranges = list(cost = costs, gamma = gammas), kernel = "radial")
model_poly <- tune(svm, Y ~ ., data = svm_pre_train, ranges = list(cost = costs, gamma = gammas), kernel = "polynomial", degree = 2)
model_sigmoid <- tune(svm, Y ~ ., data = svm_pre_train,ranges = list(cost = costs, gamma = gammas), kernel = "sigmoid")
```

Based on our grid search demonstrated in the code above it was determined that the optimal hyperparameters were the following:

- Cost: 1.555778
- Kernal: Radial (As suspected based on the plot of various data points)
- Gama: 0.1

We highlight that if we increase our sample size of 10% then we might have obtained a different kernel, though given the computational cost of undertaking this we elected not to procced with further analysis on the kernel selection, even if a better kernel would have been more appropriate (Support Vector Machine in machine learning, 2023).

As a thought experiment we then conducted the same analysis on just the radial kernel model, though on the full 70% training data set as seen with random forest. This yielded the exact same results for cost and gamma and took a total of 9.5 hours. We then plotted this visually by selecting the cost and Gama with the smallest error as indicated by blue colour intensity, though we obtained the exact figures just by calling the model produced via the tune() function to extract the exact figures as visually this can be a challenge as seen in figure 15.

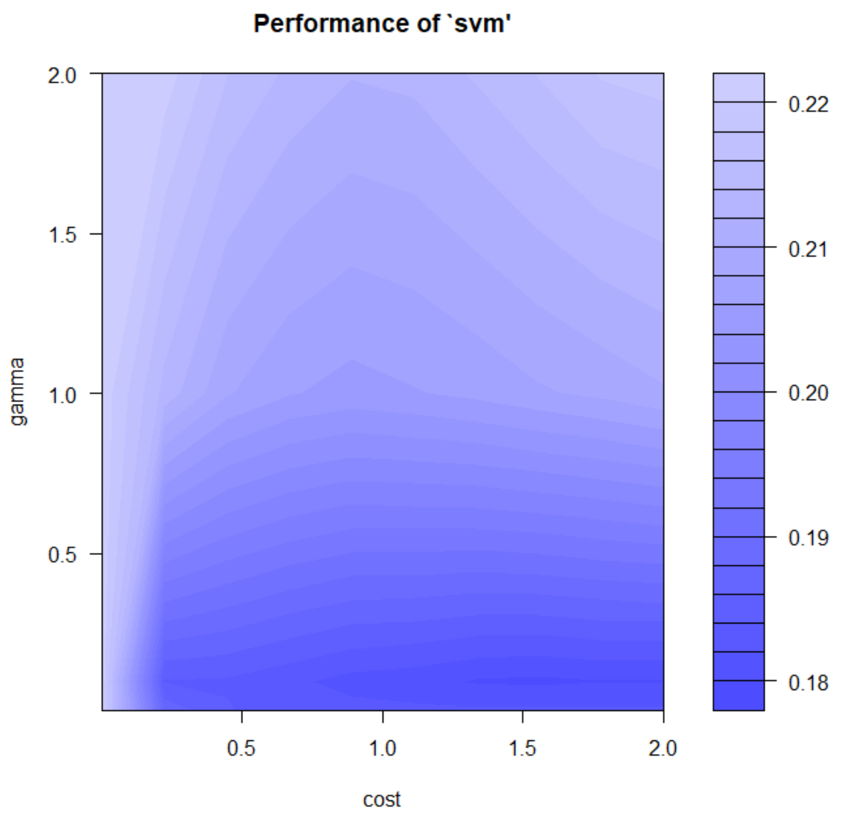
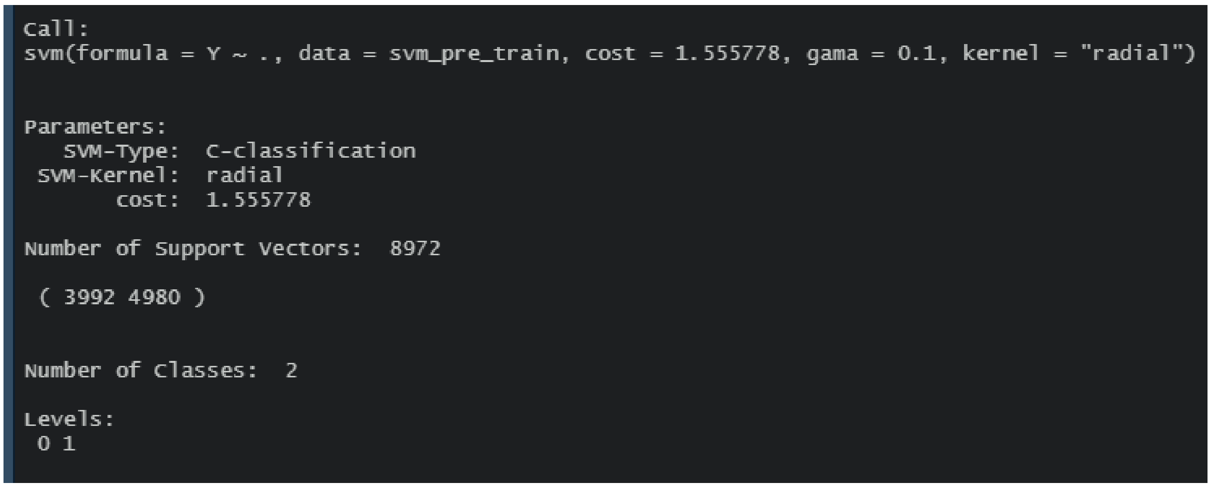


Figure 15: Plot of cost and gamma produced via the tune() function on 70% of the data to select the best hyper-parameters for the given radial kernel.

We highlight that this was a limitation in our study, we were unable to conduct a comprehensive hyper-parameter tune on all kernels; though we wanted to demonstrate one approach which could be used if we had the available resources utilising a grid search approach method; we highlight that this is a limitation of the SVM model as previously highlighted.

SVM (model summary and variable discussion)

Based on the model summary bellow we observe that we have a SVM model, the cost argument specifies the cost parameter, which determines the trade of between allowing training errors and ensuring the margins are wide, the default is set to 1 and our model is greater than 1 suggesting that we have a wider margin then the default.



```
Call:
svm(formula = Y ~ ., data = svm_pre_train, cost = 1.555778, gama = 0.1, kernel = "radial")


Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  radial
       cost:  1.555778

Number of Support Vectors:  8972

 ( 3992 4980 )


Number of Classes:  2

Levels:
 0 1
```

Figure 16

Unlike random forests which displays a nice summary of variable importance we have support vectors. Support vectors represent data points that are the closest to the decision boundary and have the most impact on the boundaries location, representing the importance of individual data points rather than individual variables. In our model summary we observe that we have a total number of 8972 support vectors (data points) that are closest to the decision boundary. We observe that we have 3992 support vectors for class 0 and 4980 support vectors for class 1.

Variable Importance

We extracted variable importance by using the train() and varimp() function of caret, though this was not the final model we used as we experienced poor documentation online in respect to svm() and tune() to extract variable importance, highlighting a need for review in documentation of these packages. The way varimp() function in R calculated importance scores of the variables is based on the decrease in accuracy after permuting their values, with a higher score indicating that a variable is more important in the model. As we can see the following three variables are the most influential. This suggests that the variable importance produced here would be most like mean decrease accuracy produced in the random forest model as it is a form of accuracy metric and not associated with node purity as this is not a function of SVM:

- X6: Repayment statues in September 2005
- X7: Repayment statues in period 2
- X8: Repayment statues in period 3

We observe that X6 is the most important variable out of all the models produced, as well as the most important across all the various metrics as well. Interestingly, X15 (bill amount) is ranked poorly in

SVM, whilst had a relatively high mean decrease accuracy in our random forest model. It is possible that the X15 variable has a stronger association with the target variable in the random forest model compared to the SVM, this could explain why it has a relatively high mean decrease accuracy in the random forest model. It is also possible that the SVM model is not able to capture the relationship between X15 and the target variable as effectively as the random forest model due to differences in their underlying assumptions and model structures overall, which we did not explore.

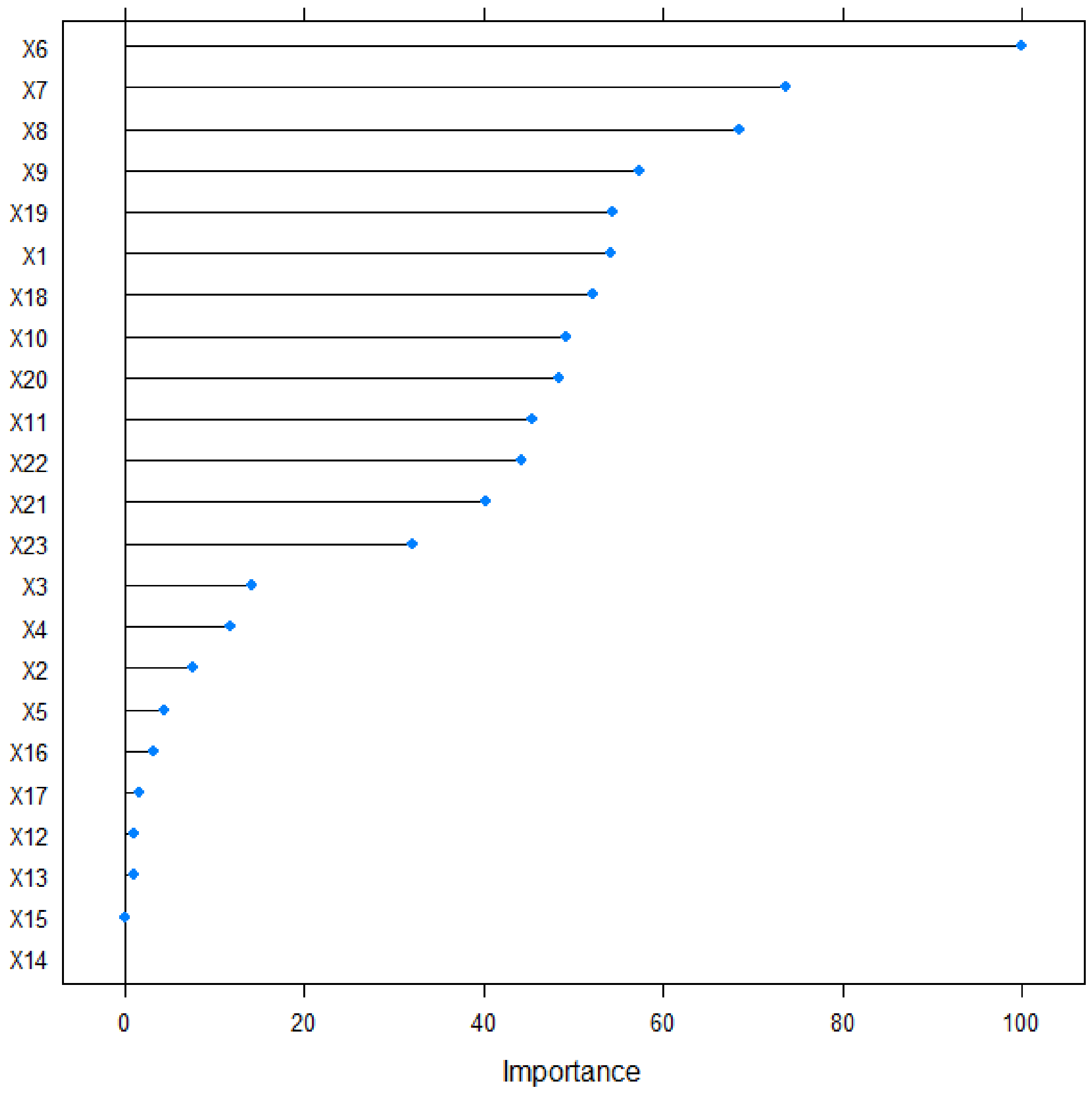


Figure 17: Importance of variables in SVM model produced by train() function of caret package on 10% of the data set for illustrative purposes of how to extract. If time persisted and computational resources allowed, we would have preferred to do 70%

Evaluate the performance of the algorithm on the training data and comment on the results:

As we hypothesised, we were able to improve our model through hyper-parameter tunning to obtain improved metrics across the board generally. Though the gain was minimal and demonstrates the trade of between accuracy and efficiency, as this process took many hours, in excel of 10; a much longer processing time then the classification tree. Based on the metrics, it appears that the tuned SVM model performed slightly better than the default SVM model (radial kernal) as we anticipated. For example, the sensitivity, positive predictive value, negative predictive value, detection rate, detection prevalence, balanced accuracy, accuracy, and error rate all showed an improvement. Though the metrics were not as high in comparison to the metrics of the Random Forest model, though interestingly the model performed like that of the bagging and classical classification tree in relation to the training data set; though we did suggest that the random forest model could have been overfitting. This gives us confidence that the model might perform better then we expect on the validation data set which we will explore in the next section as we previously highlighted our concern of SVM in relation to the underlaying data set.

Overall, the summary of performance of the model on the training data set can be summarised in the following way:

- Sensitivity/recall: 0.31597 - This measures the proportion of true positive cases that are correctly identified by the model
- Specificity: 0.96185 - This measures the proportion of true negative cases that are correctly identified by the model
- Positive predictive value/precision: 0.70172 - This measures the proportion of cases that are predicted as positive that are actually positive
- Negative predictive value: 0.83193 - This measures the proportion of cases that are predicted as negative that are actually negative
- Prevalence: 0.22123 - This is the proportion of positive cases in the dataset
- Detection rate: 0.06990 - This measures the proportion of actual positive cases that are correctly identified by the model
- Detection prevalence: 0.09961 - This measures the proportion of predicted positive cases out of all cases predicted by the model

- Balanced Accuracy: 0.63891 - This takes into account the imbalance in the dataset between the two classes, and provides an overall measure of the model's performance
- Accuracy: 0.819 - This measures the proportion of all cases that are correctly classified by the model
- F1 Score: 0.43574 – This is a measure which takes into account both precision and recall and is useful on unbalanced data sets (Toshniwal, 2020)

The biggest thing which stands out relative to the random forest model in relation to the training data is that we have a reduction in many metrics such as sensitivity/recall and positive prediction value/pression. This suggest that our model is incorrectly classifying people as non-defaulted when they should be classed as defaulted. This may be an area for further investigation to improve the model's accuracy and reliability. It is important to note that these results are based on the training dataset, and the performance of the model may differ when evaluated on a test dataset or when applied to real-world data as previously suggested that the random forest model might potentially be overfitting the model.

Model – 70% training data set	**SVM -default model**	**SVM – tuned model**
Metric	Training	Training
Sensitivity	0.31317	0.31597
Specificity	0.96185	0.96185
Pos Pred Value	0.69986	0.70172
Neg Pred Value	0.83136	0.83193
Prevalance	0.22123	0.22123
Detection Rate	0.06928	0.06990
Detection Prevalence	0.09900	0.09961
Balanced Accuracy	0.63751	0.63891
Accuracy	0.8183	0.819
F1-score	0.43271	0.43574

Again, we highlight the difference in run times between the two models, we observe that the SVM default model took a total of 1.29 minutes using the default parameters as previously mentioned, whilst the tuned SVM model took in total of excess of 10 hours including hyper parameter selection via a grid search approach which is a substantial processing time, highlighting the limitation of SVM as previously discussed on big data sets.

Model	Run time
SVM -default model (70% data scaled)	1.29 minuets
SVM – tuned model (70% data scaled)	1.42 minuets
Tunning time to select hyper parameters	10 hours

Which model should we use?

This ultimately comes down to the specified requirements and constraints we might be faced with, thus a possible trade-off between accuracy and computational costs. If computational costs are not an issue then undertaking hyper-parameter tunning yielded improved results, though it was marginally improved. In addition, one might not know the gain associated with tunning if they simply elect to ignore it and we believe it is an important aspect of machine learning especially as SVM is limited by incorrect hyper parameter tunning. In our example whilst the gain was minimal, in other real-world examples the gain could be substantial. For the purpose of comparing SVM to regression tree we elected to procced with the tuned model given the gain in pression, recall and f1-score which we will discuss in detail in the preceding section.

2.2.4 Prediction on validation data (Random Forest vs. SVM):

Hypothesis – Justification:

Based on our performance on the training data set we hypothesised that the Random Forest would excel compared to SVM model, though we did have concerns on performance of Random Forest due to the data set being unbalanced in relation to the predictor, which can be a limitation of the random forest model. In addition, we raised our concern of using SVM on our data set due to the inability of clear boundaries between classes when we plotted the data; though given we utilised a radial kernel it gave us confidence that it should perform relative well compared to if we used a linear SVC approach given the distribution of the data points.

Evaluations against test data

As highlighted previously, when determining which model is 'better' there is no one metric which is superior as it depends on the context of the problem being addressed.

- The SVM model has a higher specificity (0.96233), meaning it correctly identifies a larger proportion of non-defaulted payments compared to the tuned SVM model (0.94165).
- The random forest model has a higher sensitivity (0.38593), meaning it correctly identifies a larger proportion of defaulted payments compared to the SVM model (0.31509).
- The tuned SVM model has a higher positive predictive value/precision (0.70596), meaning a positive prediction is a more reliable indicator of a defaulted payment compared to the random forest model (0.65251).
- The random forest model has a higher negative predictive value (0.84377), meaning a negative prediction is a more reliable indicator of a non-defaulted payment compared to the tuned SVM model (0.83251).
- Both models have similar balanced accuracy scores (0.6400 and 0.66379, respectively), meaning they perform similarly in terms of correctly classifying both defaulted and non-defaulted payments.
- Both models have similar F1 scores, though Random forest excels, this is a good metric for data sets which are not balanced (Toshniwal, 2020)

Model – 70% training data set	Random Forest	SVM – tuned model
Metric	Training	Training

Sensitivity/recall	0.38593	0.31509
Specificity	0.94165	0.96233
Pos Pred Value/precision	0.65251	0.70596
Neg Pred Value	0.84377	0.83251
Prevalance	0.22114	0.22114
Detection Rate	0.08534	0.07034
Detection Prevalence	0.13079	0.09968
Balanced Accuracy	0.66379	0.64021
Accuracy	0.8188	0.8199
F1 Score	0.485002859	0.435710174

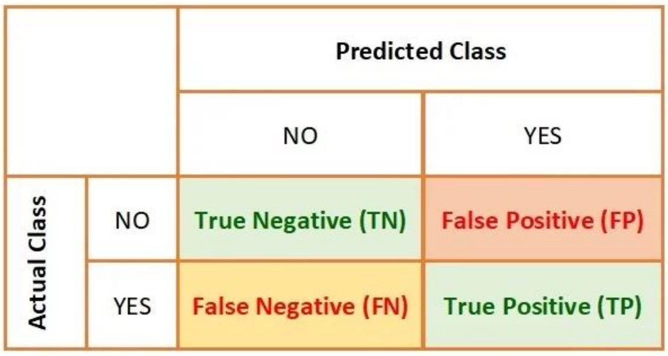
Which model do we prefer?

When comparing which model to use we should understand what the metrics mean, many of the metrics in the above table are derived from a confusion matrix which can be summarised in figure 18. In addition, three common metrics used to assess machine learning models include recall, precision and the F1-score which we summarise (Toshniwal, 2020):

- Sensitivity/recall = TP/(TP+FN)
    - Important when: Identifying the positives is crucial
    - Used when: Occurrence of false negatives is unacceptable
        - False positives are acceptable
        - False negatives are not-acceptable

- Pos Pred Value/Precision = TP/(TP+FP)
    - Important when: We want confidence of our predicated positive results
    - Used when: The occurrence of false positives is unacceptable
        - False positives are unacceptable
        - False negatives are acceptable

- F1 Score = 2*(Recall*Precision)/(Recall + Precision)
    - Important when: You have an uneven class distribution
    - Used when: The cost of false positives and false negatives are different

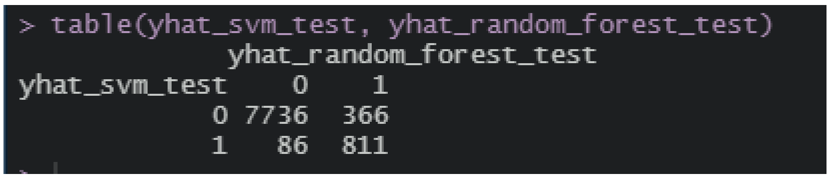


		Predicted Class	
		NO	YES
Actual Class	NO	True Negative (TN)	False Positive (FP)
	YES	False Negative (FN)	True Positive (TP)

Figure 18: Confusion matrix used to explain key concepts of common metrics used to evaluate machine learning models (Toshniwal, 2020)

When selecting which metric to use it is dependent on the question being addressed, as we described above each metric implies different outcomes, and we can see clearly by those definitions that each model has merits depending on which metric is elected.

For this question it could be suggested that sensitivity/recall to be a good measure, as the occurrence of false negatives is unacceptable as the goal of the bank could be to capture as many customers at risk of defaulting to ensure they are able to intervene earlier rather than later to reduce financial loss. We see this in figure 19 with the RF model classifying more people as defaulting then the SVM model. We have assumed that our goal is to capture as many default customers as possible based on the above discussion, as such we would elect to procced with the random forest model, which has a higher sensitivity/recall compared to SVM. In addition, we also observed during data exploration that our underlaying data set has an uneven class distribution for default payment, which is also an important feature of F1-score as well, highlighting two metrics supporting the need of the random forest model over SVM. In addition to the above points, the RF model was tuned and trained withing a period of 1 hours vs 10 hours making it our preferred model based on the above metrics and computational aspect.



```
> table(yhat_svm_test, yhat_random_forest_test)
             yhat_random_forest_test
yhat_svm_test    0    1
            0 7736  366
            1   86  811
```

Figure 19: Comparison confusion matrix of svm against random forest validation data set

Suggestions to improve both models – Justifications

Our main concern was computational aspect, especially that of the SVM model. As such, we propose two documented methods to increase computational aspect of the training and tuning such as PCA and feature selection. Whilst our random forest model took a total of 1 hour to tune and train this can increase substantially if we had a bigger data set which is a real word problem in the realm of big data; in addition 10 hours for SVM is a very long time, in the realm of big data this would need to be addressed:

- PCA: Principal component analysis is a form of dimension reduction which has been used to reduce the dimension of large data sets. A reported limitation of SVM as we seen is the computational processing required on large data sets, thus we would propose that SVM would benefit from a hybrid PCA-SVM model. Allowing us to reduce the overall dimension of the data set. It has been documented in the literature that the use of a hybrid PCA-SVM model has led to better accuracy in the overall model with a reduction in processing time (Mustaqeem & Saqib, 2021). Though a major pitfall of employing PCA is the ability to interpret the results of the PCA-SVM model, as the transformed varies are a combination of the original variables.

- Feature selection: One issues with building a model is computational processing time, whilst not shown here when we remove features which had little impact on node purity and accuracy we resulted in a random forest model which had a reduction in computational processing time by 50% with little reduction in overall accuracy; we did not apply this to SVM so unable to comment on the impact it might have had on SVM. Though, It has been documented in the literature that through feature selection models such as random forest (Prasetiyowati et al., 2022) and SVM (Haury et al., 2011) can have a decrease in computational processing time with minimal impact on accuracy. This might be particularly useful when attempting to train and develop an SVM model which in our example took over 10 hours of tuning and training time. We suggest a future study would benefit at looking at statistical methods to reduce the feature space down in order to improve computational aspect of SVM and see the impact that has on overall model accuracy. One major concern of feature selection is that It could unintentionally insert bias into our model.

Overall, what we have demonstrated is addressing limitations of one model can improve aspects of the model but also result in new limitations in the new model; this highlights that machine learning is both an art and a science in our opinion; a no one size fits all approach and dependent on the needs of the day and the particular data challenge. In addition, we could also explore cross validation much easier with SVM if the computational aspect is reduced to improve model accuracy.

References

Gandhi, R. (2018) Support Vector Machine - introduction to machine learning algorithms, Medium. Towards Data Science. Available at: https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47 (Accessed: February 6, 2023).

Haury, A.-C., Gestraud, P. and Vert, J.-P. (2011) “The influence of feature selection methods on accuracy, stability and interpretability of molecular signatures,” PLoS ONE, 6(12). Available at: https://doi.org/10.1371/journal.pone.0028210.

James, G. (2017) An introduction to statistical learning: With applications in R. New York: Springer.

Jemima Jebaseeli, T., Venkatesan, R. and Ramalakshmi, K. (2020) “Fraud detection for credit card transactions using Random Forest algorithm,” Intelligence in Big Data Technologies—Beyond the Hype, pp. 189–197. Available at: https://doi.org/10.1007/978-981-15-5285-4_18.

M, S. (2023) Introduction to random forest in R, Simplilearn.com. Simplilearn. Available at: https://www.simplilearn.com/tutorials/data-science-tutorial/random-forest-in-r#:~:text=newly%20launched%20phone.-,Assumptions%20for%20the%20Random%20Forest%20Algorithm,handled%20from%20training%20the%20model. (Accessed: February 12, 2023).

Marsupial, D. (1963) Is hyperparameter tuning on sample of dataset a bad idea?, Cross Validated. Available at: https://stats.stackexchange.com/questions/233548/is-hyperparameter-tuning-on-sample-of-dataset-a-bad-idea (Accessed: February 12, 2023).

Mustaqeem, M. and Saqib, M. (2021) “Principal component based Support Vector Machine (PC-SVM): A hybrid technique for software defect detection,” Cluster Computing, 24(3), pp. 2581–2595. Available at: https://doi.org/10.1007/s10586-021-03282-8.

Prasetiyowati, M.I., Maulidevi, N.U. and Surendro, K. (2022) “The accuracy of random forest performance can be improved by conducting a feature selection with a balancing strategy,” PeerJ Computer Science, 8. Available at: https://doi.org/10.7717/peerj-cs.1041.

Sahithi, G.L. et al. (2022) “Credit card fraud detection using ensemble methods in machine learning,” 2022 6th International Conference on Trends in Electronics and Informatics (ICOEI) [Preprint]. Available at: https://doi.org/10.1109/icoei53556.2022.9776955.

Support Vector Machine in machine learning (2023) GeeksforGeeks. GeeksforGeeks. Available at: https://www.geeksforgeeks.org/support-vector-machine-in-machine-learning/ (Accessed: February 12, 2023).

Support Vector Machines (no date) R. Available at: https://search.r-project.org/CRAN/refmans/e1071/html/svm.html (Accessed: February 12, 2023).

Tokuç, A.A. (2022) Why feature scaling in SVM?, Baeldung on Computer Science. Available at: https://www.baeldung.com/cs/svm-feature-scaling (Accessed: February 12, 2023).

Yeh, I.-C. and Lien, C.-hui (2009) “The comparisons of data mining techniques for the predictive accuracy of probability of default of credit card clients,” Expert Systems with Applications, 36(2), pp. 2473–2480. Available at: https://doi.org/10.1016/j.eswa.2007.12.020.

R-Code

```
#==================================================================================

# Script to answer assessment 2

# Created by: Jayden Dzierbicki

#

# The purpose of this code is to answer questions from assessment

#==================================================================================




# Clear global environment

rm(list=ls())



# SETUP============================================================================

getRversion() # Get r-version for reproducability


# Load in required packages

install.packages("e1071")

library(e1071)

library(ggplot2)

library(dplyr)

library(quadprog)

library(readxl)

library(lattice)

library(caret)

library(randomForest)

library(ggplot2)

library(ggparty)

library(rpart)
library(tree)

library(dplyr)

library(MASS)

library(pROC)

library(ipred)

library(gridExtra)

library(pROC)

library(doParallel)

library(parallel)

library(kernlab)

library(rminer)


# Start Q1=====================================================================


# Load in training data as specified in Q1

df <- data.frame(x1 = c(3,4,3.5,5,4,6,2,-1,3,3,-2,-1),

                 x2 = c(2,0,-1,1,-3,-2,5,7,6.5,7,7,10),

                 y = c(-1,-1,-1,-1,-1,-1, 1,1,1,1,1,1))






# Draw a scatter plot of training data, colour -1 and 1

plot_1 <- ggplot(data = df, mapping = aes(x = x2, y=x1, colour = factor(y))) +

  geom_point(size = 5) + # Increase size for ease of reading

  scale_colour_manual(values = c("blue", "red"), name = "y",

                      labels = c("-1", "1")) +

  ggtitle("Scatter Plot of Training Data")


# Print out the above plot produced

print(plot_1)
```

```
# Note:

# We know that sole.qp() package offers optimisation for:

# min_b: -d^T*b + 0.5*b^T*D*b

# Such that. A^T*b > b0

#

#

# We can map our problam such that

# d = 0

# b' = [w, b]

# D = [I,0;0,0]

# A^T = [y*x_1, y*x_2, -y]

# b_0 = [1,1,1,...,1]

# Source: https://biodatascience.github.io/statcomp/ml/svm.html



# Spit data into features and target

x <- df[, c("x2", "x1")] # obtain features

y <- df$y # Obtain target


# Obtain dimension of features

n <- nrow(x)

p <- ncol(x)


# Create matrix x, this will allow us to solve the function

x <- as.matrix(df[,2:1]) # Since X2 is on X axis and X1 on y axis we select these positions

y <- c(df$y)


# find number observations in y

n <- length(y)
```

```
# Count the number of features/columns

p <- ncol(x)


# Create Dmat matrix (nxn)

Dmat <- matrix(0, nrow=p+1, ncol=p+1)


# Make D matrix Identity I, set a small value to prevent error

diag(Dmat) <- 1

Dmat[p+1, p+1] <- 1e-6 # Prevent error


# Create D vector

dvec <- matrix(0, nrow=p+1)


# Create A^T matrix

Amat <- t(cbind(x * y, -y))


# Create b vectors (all ones)

bvec <- matrix(1, nrow=n)


# Solve the quadratic problem

qp_output <- solve.QP(Dmat, dvec, Amat, bvec)


# Extract W and B

w <- qp_output$solution[1:p]

b <- qp_output$solution[p+1]


# Plot model with hyperplane and margins as per question

ggplot(df, aes(y = x1, x= x2,colour= factor(y))) +

  geom_point(size = 5) +

  geom_abline(intercept=(b+1)/w[2],slope=-w[1]/w[2],alpha=.2,linetype=2) +
  geom_abline(intercept=(b-1)/w[2],slope=-w[1]/w[2],alpha=.2,linetype=2) +
  geom_abline(intercept=b/w[2],slope=-w[1]/w[2],linetype=3) +
  scale_colour_manual(values = c("blue", "red"), name = "y",
                     labels = c("-1", "1"))

# Obtain margin value, from the value w extracted from qd_output
w_norm <- sqrt(sum(w^2))
margin <- 2/ w_norm

# END Q1======================================================================

# Start Q2 ===================================================================

# Load in credit card data
credit_card <- read_excel("CreditCard_Data.xls") # Load in data
credit_card <- credit_card[-1,] # Delete  row 1, as this is not needed
credit_card <- credit_card[,-1] # Delete column 1 - ID, this does not need to be included in analysis



# We also observe whilst the data is generally clean, some groupings have repetition
# X3: encodes, 4 as other & 5/6 as unknown
# X4: encodes 0 as unknown, and 3 as other
# For this purpose we will assume other and unkown are the same thing from an
# analysis purpose and group these together
# This is confirmed thoruhg orignal research paper

# Re-categorise group X3 and X4 based on above issue
credit_card <- credit_card %>%
  mutate(X3 = case_when(X3 == "0" | X3 == "4" | X3 == "5" | X3 == "6" ~ "4",
                        TRUE ~ X3)) %>%
  mutate(X4 = case_when(X4 == "0" | X4 == "3" ~ "3",
                        TRUE ~ X4) )

barchart(credit_card$Y) # Confirm maniupultion worked
barchart(credit_card_t$X3) # Confirm manipulation worked

# Data exploration, wrangling and pre-prossesing
(na_count <- sapply(credit_card, function(y) sum(length(which(is.na(y))))) ) # No NA values observed,
data is compete

summary(credit_card) # Currently all data is of character type, will possibly need to convert to
numric, int or factor. ID could be turned into rowname

# Mutate data to correct data type as it imported as character
credit_card_clean <- credit_card %>%
  mutate(X1 = as.integer(X1),
         X2 = as.factor(X2),
         X3 = as.factor(X3),
         X4 = as.factor(X4),
         X5 = as.integer(X5),
         X6 = as.factor(X6),
         X7 = as.factor(X7),
         X8 = as.factor(X8),
         X9 = as.factor(X9),
         X10 = as.factor(X10),
         X11 = as.factor(X11),
         X12 = as.integer(X12),
         X13 = as.integer(X13),
         X14 = as.integer(X14),
         X15 = as.integer(X15),
         X16 = as.integer(X16),
         X17 = as.integer(X17),
         X18 = as.integer(X18),
         X19 = as.integer(X19),
         X20 = as.integer(X20),
         X21 = as.integer(X21),
         X22 = as.integer(X22),
         X23 = as.integer(X23),
         Y = as.factor(Y) )

# Demosntrate if data types are integrated heavily by feature (This is related to SVM which is disccused in paper why we did it)
ggplot(credit_card_clean, aes(y = (X21), x= (X13),colour= factor(Y))) +
  geom_point(size = 2)

# Extract integer/factor columns to do easy visuals in one plot
integer_cols <- names(credit_card_clean)[sapply(credit_card_clean, is.integer)]
facotr_cols <- names(credit_card_clean)[sapply(credit_card_clean, is.factor)]
```

```r
# Set up plot list, to easily loop through interger and factor data and save to list
plots <- list()
plots_facotr <- list()

# Loop through and plot numeric data and save to list
for (i in 1:length(integer_cols)) {
  plots[[i]] <- ggplot(credit_card_clean, aes_string(x = integer_cols[i], fill = "Y")) +
    geom_density(alpha = 0.5) +
    scale_x_continuous(limits = range(credit_card_clean[, integer_cols[i]])) +
    ggtitle(paste("Density Plot of", integer_cols[i], "Grouped by Y")) +
    xlab(integer_cols[i]) +
    ylab("Density") +
    facet_wrap(~Y, nrow = 1)
}

# Loop through and save plot to list for factored data
for (i in 1:length(facotr_cols)) {
  plots_facotr[[i]] <- ggplot(credit_card_clean, aes_string(x = facotr_cols[i], fill = "Y")) +
    geom_bar(stat = "count", position = "fill") +
    ggtitle(paste0(facotr_cols[i], " against Y")) +
    scale_y_continuous(labels = scales::percent)
}

# Produce correltion plot
library(corrplot)
corrplot((cor(credit_card_clean[integer_cols])) )# Some strong cor, thus keep in mind for assumption testing

# Print the plots on one output
grid.arrange(grobs = plots, ncol = 4)
grid.arrange(grobs = plots_facotr, ncol = 4)
```

```
# Select a random sample of 70% of the full dataset as the training data
set.seed(123)  # For reproducability

# Split data & generate a train and test data set
split <- createDataPartition(credit_card_clean$Y, p = 0.7, list = F)
train_data <- credit_card_clean[split,]
test_data <- credit_card_clean[-split,]


# Print out the dimesnion of the train data
dim <- dim(train_data)
print(paste0("The number of rows in the training data set is: ", dim(train_data)[1], " This is ",
100*dim(train_data)[1]/nrow(credit_card_clean), "% of the total number of rows in the data set",
        ". The number of columns in the training data set is: ", dim(train_data)[2]))


# We will now compare 3 models, and then select the best which was RF

# Random forrest model ===
set.seed(123)
start_time <- Sys.time() # Keep track of RF run time

# Define the control parameters for the random forest
# Hyper parameters - just demoing hyper-parameter selection
p <- ncol(credit_card_clean) - 1 # Number of predcitors, less y
sqrt_p <- sqrt(p) # Will be used in mtry
ctrl <- trainControl(method = "cv", number = 10) # Set up cross fold validation in train
```

```
# Define the grid of ntree and mtry values to search

mtry_grid <- c(sqrt(p), p / 2, p / 4) # Compare some p-values


# Perform the grid search using the train function for rf. This will allow us to select the best p-value for bagging (number of features at each spit)

rf_hyperparameter_selection <- train(Y~., data = train_data,

                               method = "rf",

                               trControl = ctrl,

                               tuneGrid = grid,

                               importance = T)


end_tune <- Sys.time() - start_time # 58 mins

best_mtry <- rf_hyperparameter_selection$bestTune$mtry # Extract best mtry 5.75

plot(rf_hyperparameter_selection) # Plot to visualise the best mtry value


set.seed(123)  # For reproducability

start_time <- Sys.time() # Start time, we will run RF using the randomforest model

model_random_forest <- randomForest(Y ~., data = train_data, ntree = 500, mtry = 5.75, importance = T) # Produce random forest model

model_rf_run_time <- Sys.time() - start_time # Takes over 1min on current data set.

summary(model_random_forest) # Produce summary of model produced

print(model_random_forest) # OOB error



# We will now look at var importnace of random forest model


# Varible importance

# Mean decrease accuracy:

#       Gives estimate of the loss in prediction performance when that

#       particular varible is ommted from the training set

#

# Mean decrease GINI:
```

```
#	GINI is a measure of node impurity, if we use this feature to split
#	the data, how pure will the nodes be?
#	higher purity means that each node
#	contains elements of only a single class
varImpPlot(model_random_forest) # Print variable importance



# Produce prediction on test data set, using the RF model on test and train
yhat_random_forest_test <- predict(model_random_forest, newdata = test_data, type = "class")
yhat_random_forest_train <- predict(model_random_forest, newdata = train_data, type = "class")


# Confusion matrix - produce confusion matrix summary
(confusion_matrix_random_forest_test <- confusionMatrix(as.factor(yhat_random_forest_test),
as.factor(test_data$Y),  positive ="1"))



# Confusion matrix - train
(confusion_matrix_random_forest_train <- confusionMatrix(yhat_random_forest_train,
train_data$Y,positive ="1"))







# Bagging method: Just to select model based on training =======================
# Just using defult settings
set.seed(123)  # For reproducability
start_time <- Sys.time() # Keep track of model run time
model_bagging <- randomForest(Y ~., data = train_data, ntree = 500, mtry = p, importance = T) #
Produce bagging model, using mtry = p for bagging
model_bagging_run_time <- Sys.time() - start_time # Takes over 1min on current data set.
summary(model_bagging) # Produce summary
```

```
print(model_bagging) # OOB error: 18.06%


# Predict and plot confusion matrix on train data only

yhat_bagging_train <- predict(model_bagging, new_data = train_data, type = "class")


(confusion_matrix_bagging_train <- confusionMatrix(yhat_bagging_train, train_data$Y, positive ="1"
))






# Simple classification tree: Just to select model based on training results====

# Just using default setting

set.seed(123)  # For reproducability

start_time <- Sys.time() # Keep track of model run time

M1 <- rpart(Y ~ ., data = train_data, method = "class") # Produce M1

M1_run_time <- Sys.time() - start_time

plotcp(M1) # No need to refine model

library(rattle) # Package to plot tree

rattle::fancyRpartPlot(M1) # Plot model


# Predict and plot confusion matrix

yhat_m1_train <- predict(M1, newdata = train_data, type = "class")


(confusion_matrix_m1_train <- confusionMatrix(yhat_m1_train, train_data$Y, positive ="1" ))
```

```
#SVM Hyper-paramter grid search demo==============================================

# Will do analysis on 10% of data due to constraints and then train full model==
# Split data & generate a train and test data set, 10% due to process time======

# SVM Should be scaled - we will scale the data using caret package via preProccess()
num_vars <- select_if(credit_card_clean, is.numeric) # Select all numeric vars to be scaled
preprocess_info <- preProcess(num_vars, method = c("center", "scale")) # Scale numeric varibles
scaled_num_vars <- predict(preprocess_info, num_vars)
scaled_df <- bind_cols(scaled_num_vars, select_if(credit_card_clean, is.factor)) #Bind scaled and
factored data, we do not need to scale factored data


# Partition data with set seed 123 & create data partition - as we need to partition scaled data
set.seed(123)  # For reproducability

# Set SVM split based on requirement (this is mentioned in the paper why we toggle)
svm_pre_split <- createDataPartition(scaled_df$Y, p = 0.1, list = F) # For tunning
svm_pre_train <- as.data.frame(scaled_df[svm_pre_split,]) # Define train scaled data for svm
svm_pre_test <- (scaled_df[-svm_pre_split,]) # Define validation data for svm

# Define a list of cost/gama parameters to try during our tune proccess on 10% data
gammas <- c(0.01, 0.1, 1, 2) # Test gama
costs <- c(seq(0.001, 2, length = 10))

# Train the model using the tune() function and cross-validation================
set.seed(123) # Set seed for reproducability
```

```
start_time <- Sys.time() # Start system time to track how long it takes to run valdiation

model <- tune(svm, Y ~ ., data = svm_pre_train, ranges = list(cost = costs, gamma = gammas), kernel = "radial") # Radial tune

model_poly <- tune(svm, Y ~ ., data = svm_pre_train, ranges = list(cost = costs, gamma = gammas), kernel = "polynomial", degree = 2) # Polynomial tune

model_sigmoid <- tune(svm, Y ~ ., data = svm_pre_train,ranges = list(cost = costs, gamma = gammas), kernel = "sigmoid") # Sigmoid tune

svm_end_time <- Sys.time() - start_time


# Compare above models and find the best model

model_performance <- model$best.performance # Best cost/gama for radia

model_poly_performance <- model_poly$best.performance  # Best cost/gama for polynomial degree 2

model_sigmoid_performance <- model_sigmoid$best.performance  # Best cost/gama for sigmoid

(best_model_index <- which.min(c(model_performance, model_linear_performance, model_poly_performance, model_sigmoid_performance))) # Easy way to find which model was the best. smallest error

plot(model) # Plot to demonstrate visually


# We will use caret package train() to look at varible importance. This

# looks at deacrease in accuraycby removing features from the model.

# SVM() and tune() appear not to do this feature

var_importance <- train(Y ~ ., data = svm_pre_train, method = "svmRadial")

varImp(var_importance$coefnames)




# For curiosity, we wanted to see how radial perfomred on 70%, plus we will use

# 70% in final model

set.seed(123)  # For reproducability


# Set SVM split based on requirement (this is mentioned in the paper why we toggle)

svm_pre_split <- createDataPartition(scaled_df$Y, p = 0.7, list = F) # For tunning

svm_pre_train <- as.data.frame(scaled_df[svm_pre_split,]) # Define train scaled data for svm
```

```
svm_pre_test <- (scaled_df[-svm_pre_split,]) # Define validation data for svm


# Start 70% hyper-paremeter tune

start_time <- Sys.time() # Start system time to track how long it takes to run valdiation

model <- tune(svm, Y ~ ., data = svm_pre_train, ranges = list(cost = costs, gamma = gammas), kernel
= "radial") # Radial tune

svm_end_time <- Sys.time() - start_time


# We found that regadless of 70% or 10% on radial model, we had the following

# hyper paretmers

# - Cost: 1.55576

# - Gama: 0.1







# Produce two final SVM models, one based on tuning, and one based on default===

# Use 70 data set as seen in random forest


# Tuned SVM Model - final 70% data

start_time <- Sys.time() # Keep track of time

svm_model_sig <- svm(Y ~ ., data = svm_pre_train, cost = 1.555778 , gama = 0.1, kernel = "radial") #
Produce tuned svm model

model_svm_run_time <- Sys.time() - start_time # End time

summary(svm_model_sig) # Summary of model

print(svm_model_sig) # summary of model


# Defult SVM model - for compairsion to see if tuned was better

start_time <- Sys.time() # Start time

svm_model_basic <- svm(Y ~ ., data = svm_pre_train) # Defult settings

model_svm_run_time <- Sys.time() - start_time # End time
```

```
# Produce summary on 70% trained data for confusion matrix. Tuned model

yhat_rad <- predict(svm_model_sig, newdata = svm_pre_train)

confusionMatrix(yhat_rad, svm_pre_train$Y, positive ="1" )


# Produce summary on 70% trained data for confusion matrix. Basic model

yhat_svm <- predict(svm_model_basic, newdata = svm_pre_train)

confusionMatrix(yhat_svm, svm_pre_train$Y, positive ="1" )



# Compare models of RF and SVM against test data================================


# Provide estimates on validation data set from tuned model

yhat_svm_test <- predict(svm_model_sig, newdata = svm_pre_test)


# Print confusion matrix for test data set on tuned model

confusionMatrix(yhat_svm_test, svm_pre_test$Y, positive ="1")

confusion_matrix_random_forest_test  # We have already saved RF confusion matrix and calling on
it


# Comapre svm vs random forest model in table format to see how predictions varied between the
two models

table(yhat_svm_test, yhat_random_forest_test)








# END Q2 and assesment ========================================================
```