Supervised Learning Assignment 2

Tiffany Woodley

April 2019

Contents

List of Figures

List of Tables

1 Data Exploration

The dataset consists of 4925 rows and 12 columns, and since we are working with a classification problem we are looking for the predictor variables which have the biggest variance between classes. This is explored using box and density plots. One of the columns is an ID column which will not be of use, since it doesn't contain any information about the class and will just confuse the models as they will see the column as an ordinal variable.

1.1 Class Variation

The box plots in Figure 1 shows the range of the predictor variables for each class. From this we can see the variables with the most variation are: length, eccen, p_black and p_and. It seems that class 3 should be the easiest to predict as blackpptx, blackand and wbtrans have higher variation in class 3 in comparison with the other classes. Class height and area have some serious outliers in the 4th and 5th classes which could lead to the models not being as sensitive to the variation in the other classes. For instance if a weight in a neural net is set to a larger value to accommodate the smaller values it will create huge fluctuations in the system for the outliers. For example, the general range of of area is 0 to 2500, whilst the max is close too 150 000. Since the underlying distribution of the function we are trying to find is unknown, the variables were left as is since that is how the unseen information will come in.

The different columns all have different ranges and would need to be scaled in order for the models to work.

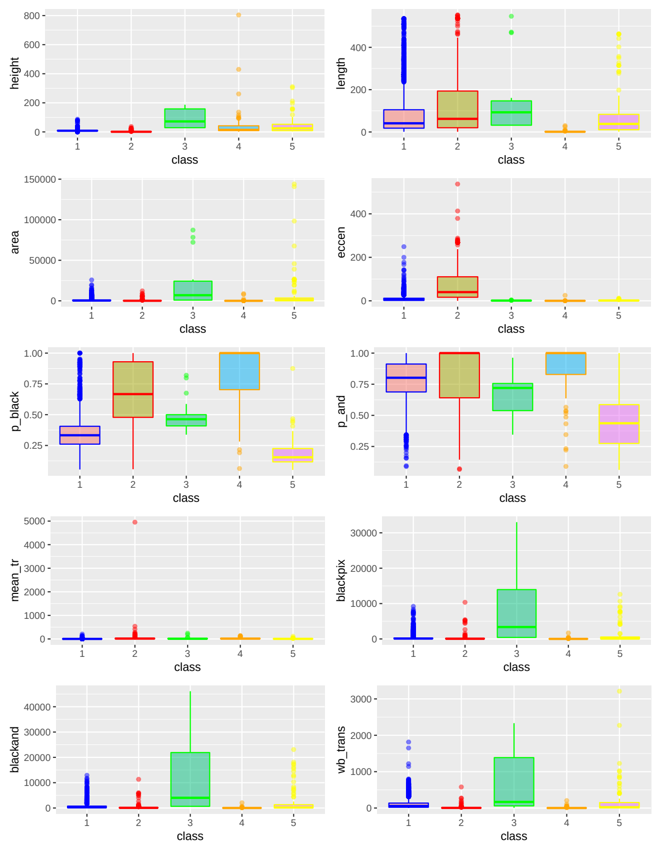


Figure 1: Predictor Variable Boxplots

1.2 Class Densities

The density plots as seen in Figure 2 highlight the point mentioned in the previous section about the outliers. All the variational information can be seen in the small range at the beginning of the plots. P_black and P_And show how the distributions differ. It looks like their is some information missing in class 4, as if the values of it have been capped.

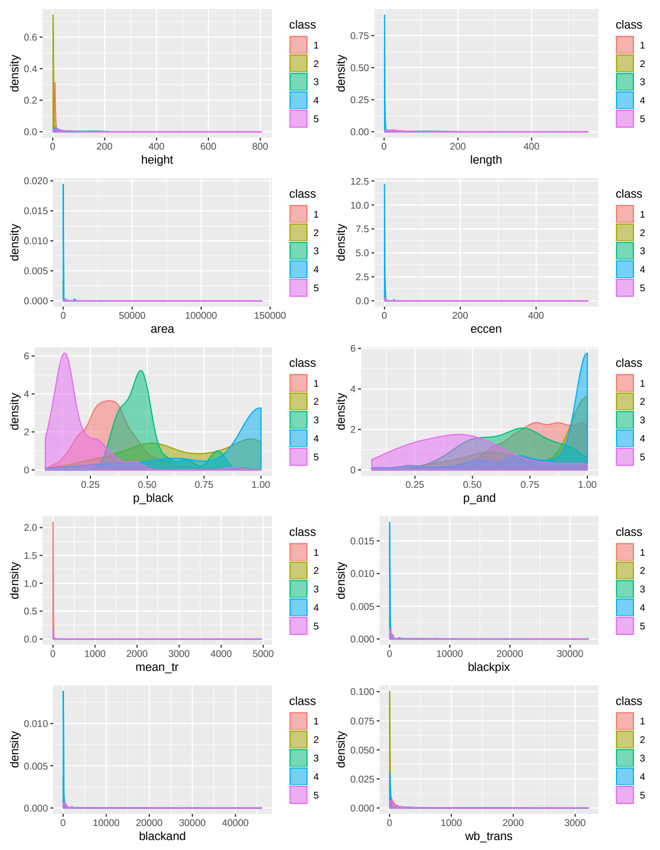


Figure 2: Predictor Variable Density Plots

1.3 Unbalanced Data Set

The dataset is very unbalanced, having most of the rows being in class 1 as seen in Figure 3. It is desirable to beat the classification rate of 89.75% ,which we would get if we just classified that every class was in the first class, because of the imbalance. The training/test set was split using a ratio on each class(stratified) so that dataset doesn't end up not including any class 3 in the training set and all of them in the test set, as this would lead to the model predicting on a class it has not had any training on. Since their are so few observations in class 3 we might have to use a bigger learning rate, as the few observations in the class that are used to train the model need to have an impact.

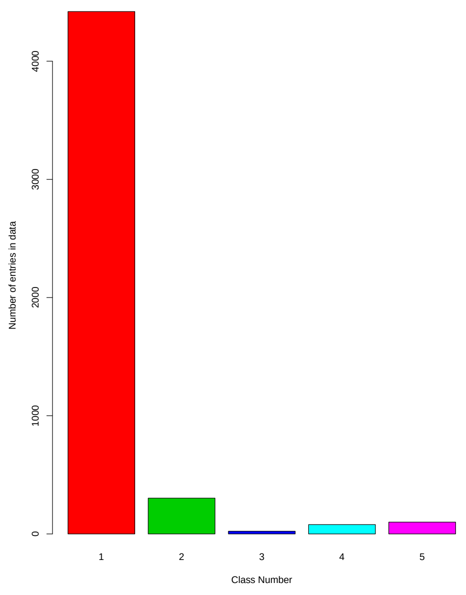


Figure 3: Unbalanced Data

2 Neural Networks

2.1 Brief Overview

A neural network is made up of multiple neurons in different layers, connected together by their respective allocated weights. This will be explored further during this section.

2.1.1 Feed Forward Network Architecture

There are three types of layers in a feed forward neural network, namely:

- **input layer**

  The input layer consists of a layer of nodes which take in the inputs(predictor variables) of the data. The number of input nodes equals the number of predictor variables

- **hidden layer**

  The hidden layers sit between the input and output layers, they are comprised of multiple neurons which take in inputs, and then multiply them by their specific weights, and then sum them together in order to be passed through an activation function

- **output layer**

  The output layer comprises of neurons which output the final predictions. The number of neurons in the output layer depends on the number of outputs being predicted.

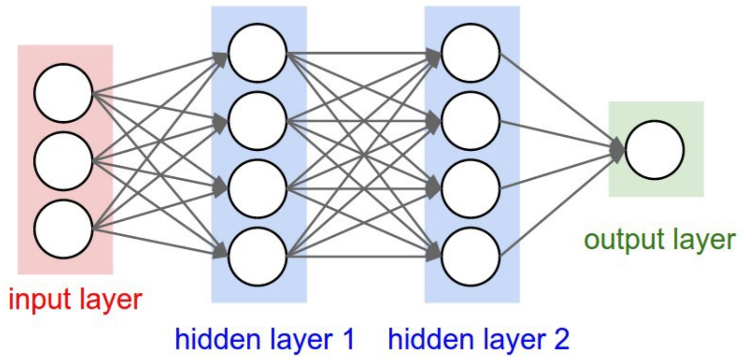


Figure 4: Feed Forward Architecture [4]

The layers can be viewed in Figure 4, the input layer is shown in red, the hidden layers in blue and the output layer in green. The information moves in a forward direction through the network from left to right, this is why it is called a feed forward network

2.1.2 Neurons

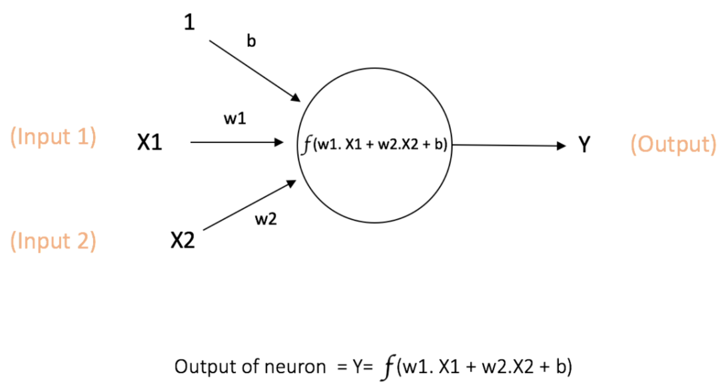


Figure 5: Architecture of a Neuron [9]

A neural network's building blocks are neurons. The architecture of a neuron can be seen in Figure 5. Each neuron takes in inputs from different neurons in the network or from input nodes and computes a single output. This computation consists of the node taking in it's given inputs and multiplying them each by their associated weight, then finds the weighted sum of these. After the weighted sum has been calculated it is passed to an activation function(discussed in the next section) which computes the final output.

2.1.3 Activation functions

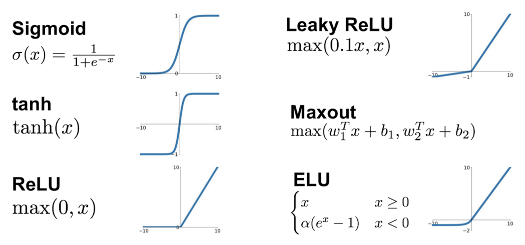


Figure 6: Activation Functions [7]

The non-linearity of a neuron's output is due to it's activation function. An activation takes in the weighted sum of a neuron and performs a mathematical function on it. Figure 6 shows the graphical plot and equation of different activation functions.

2.1.4 The Back-Propagation Algorithm

Once the feed forward network structure has been set up, the model needs to be trained, this is done using the back-propagation algorithm. The algorithm works by updating the weights according to a loss function. The loss function is chosen according to the specific problem, for example mean squared error can be used if you are targeting specific values in a regression problem. Figure 7 illustrates this process, training inputs are given to the network and the output is then compared to the expected output using the loss function 'error estimation'. The algorithm then adjusts the weighting by propagating the error backwards through the network, it repeats this process multiple times to get an accurate network.

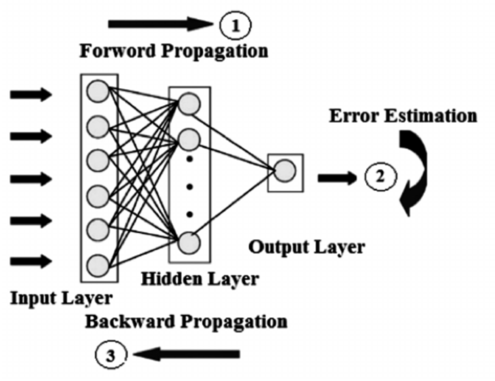


Figure 7: Back-Propagation Algorithm [5]

The number of times this process is iterated depends on the batch number and number of epochs defined. The batch number refers to the number of training observations the algorithm should skip between the training inputs. It can be beneficial to skip some of the training inputs as they could lead to over-fitting the model to the training data, it also helps reduce computation time. The number of epochs refers to the number of times the training data should be run over during the training process.

2.1.5 Gradient Descent

Gradient descent is used as an optimizer to train the network.Gradient descent attempts to minimize a function step by step moving in the direction which has the steepest gradient to find the global cost minimum - this can be seen visually in Figure 8. Most optimizers use a stochastic approach such as stochastic gradient decent which allows gradient descent to be possible with the constraints we have with computing power. It also comes with the added benefit of being random and so the model is less likely to get stuck in a local optima.

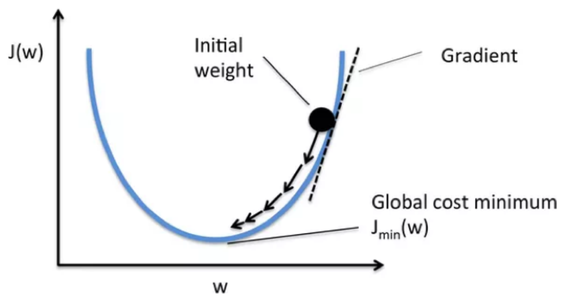


Figure 8: Gradient Descent [8]

2.2 Initial Model

The initial model was created using all of H2O's library deep learning function defaults, this will be the base model and the model to beat. The inputs were standardised and the model was fitted. Hyper parameter tuning will be done in the next section using grid search.

The train and test confusion matrices are shown below

	1	2	3	4	5
1	3053	33	2	11	25
2	18	173	0	1	1
3	3	0	11	1	0
4	1	2	0	36	0
5	15	2	5	3	51

Table 1: Neural Network Initial Model Training Set Results

	1	2	3	4	5
1	1353	24	0	5	9
2	4	77	0	0	0
3	5	0	4	0	0
4	1	0	0	22	0
5	8	1	1	0	16

Table 2: Neural Network Initial Model Test Set Results

From Table 3 it can be seen that the network struggles with predicting class 3 and 5. Class 3 has the fewest available training observations and so the network probably wasn't able learn it's classes features well enough - even though class 3 shows the most variation in some of the predictor variables as shown in the data exploration section.

Since the classification percentage is the highest for class 1 (the class with most training points) it brings up the percentage overall.

Class Number	Classification Percentage
1	97.27 %
2	95.06 %
3	44.44 %
4	95.65 %
5	61.54 %
Overall Test	96.21 %

Table 3: Neural Network Initial Model Classification Percentages

The training set accuracy percent is 96.43 %, leaving a misclassification percent of 3.57 %. The test set accuracy percent is 96.21 %, leaving a misclassification percent of 3.79 % . These figures are quite close and show the model generalizes well. This goes to show the H2O defaults work well.

2.3 Variable Significance

Figure 9 shows the variable importance plot produced by H2O, and it shows eccen as the most significant. If we refer back to the initial data exploration in section 1, it is interesting to see that some variables which didn't seem important in the data exploration are seen as important here. This just goes to show how neural networks are able to capture underlying patterns which may not be clear when initially looking into the data.

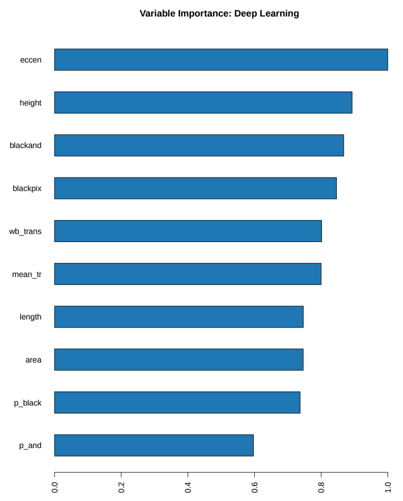


Figure 9: Neural Network Variable Importance

2.4 Grid Search

A multitude of models were explored for the purpose of class prediction, this was automated by the use of grid search. Cross validation using 10-folds and a stratified approach was used for the evaluation of the models and the best model was evaluated on the validation set error.

The biggest search criterion was the hidden layers, there needs to be enough layers to pick up the complexities of different patterns in the data and there needs to be the right amount of neurons - too many and the network will over-fit, too little and the network will under-fit.

Activation functions are important in the model as they alter how it operates. The output type decides what the output activation has to be but the activation functions in the hidden layers can be altered and can be used to improve the model.

H2O works with an adaptive learning rate, this means that it starts by taking bigger steps during gradient descent and taking smaller steps as it feels the gradient decreasing. The hyper-parameters for this rho and epsilon where tuned to make sure this worked for the problem at hand.

Regularization is extremely important when it comes to neural nets as they are very flexible, and so multiple regularization methods were tested in combination to see how the network could be coerced to generalize well. The methods used were drop out, early stopping, L1 and L2 regularization.

The batch size was also altered as the dataset is quite large and the computational requirements to train the models were quite high. By increasing the batch size it also helps the model to not over fit the training data.

The parameters explored are shown in the table below and the final values chosen for them.

Hyper-parameter	**Value Chosen**	**Reason for tuning**
activation	Maxout	Changes the inner functioning of the model
epochs	50	Changes the amount of times the algorithm runs over the training set during training
hidden	(120)	Changes the configuration and in turn the complexity of the model
input drop out ratio	0.1	The percentage drop out used on the input layer, this is useful for help with over-fitting of the input data
output dropout ratio	0.1	The percentage drop out used on the hidden layers controls over-fitting of the learned features
stopping rounds	5	Amount of iterations to average the stopping criteria over
stopping metric	logloss	What metric to evaluate the stopping condition on
stopping tolerance	0	The tolerance of the stopping metric to stop at
rho	0.99	is the similarity to prior weight updates (similar to momentum)
epsilon	1e-10	parameter that prevents the optimization to get stuck in local optima
L1	0.0001	L1 regularises the network by letting only strong weights survive
L2	0.001	L2 regularises the network by preventing any single weight from getting too big

Mini batch size	50	Helps with computation efficiency and over-fitting
Balance classes	F	Helps balance classes by over and under sampling the dataset

Table 4: Grid Search Hyper parameters [1]

Figure 10 shows how the grid search works, this just shows three parameters being tuned as it's easier to visualize, the graph shows the logloss of the different models. This is just for visualization purposes, a much more extensive grid search was performed.

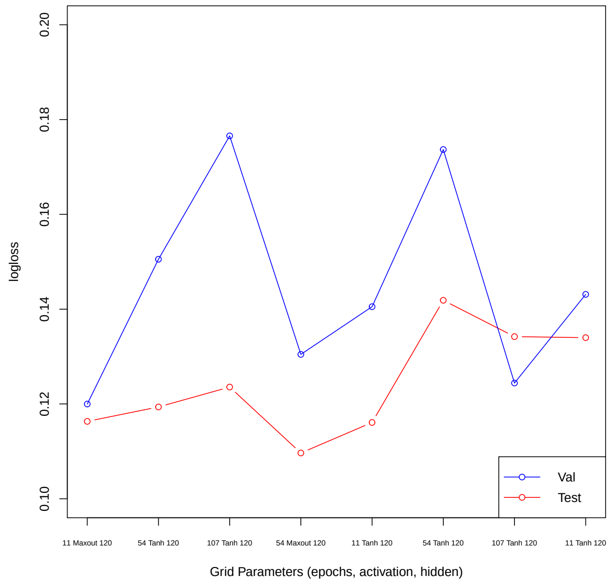


Figure 10: Grid Search Graph

2.5 Final Model

The final model's weights are very different to the base model's. This is due to the complexity of the model, the models that are fitted are vastly different every time they are fitted. This high variance of the model means it can create models with low bias. The low bias is why it is so important to have regularization when training, otherwise the model will over-fit.

	1	2	3	4	5
1	2617	20	4	6	25
2	16	161	0	1	1
3	2	0	9	0	0
4	4	3	0	43	0
5	4	2	0	0	37

Table 5: Neural Network Final Model Training Set Results

	1	2	3	4	5
1	871	11	1	2	11
2	3	60	0	0	0
3	2	0	2	0	0
4	1	0	0	12	0
5	1	0	1	0	8

Table 6: Neural Network Final Model Test Set Results

The final model still performs poorly on class 3, but has shown improvements in class 5.

Class Number	Classification Percentage
1	97.21 %
2	95.24 %
3	50.00 %
4	92.31 %
5	80.00 %
Overall Test	96.65 %

Table 7: Neural Network Final Model Classification Percentages

The training set accuracy percent is 97.02 %, leaving a misclassification percent of 2.98%. The test set accuracy percent is 96.65%, leaving a misclassification percent of 3.35%. These values are similar and therefore indicate towards the model generalizing well

The variable importance plot shown in Figure 11 is vastly different to the initial model's, this just further goes to show the variation of models that a neural net can fit to the same data.

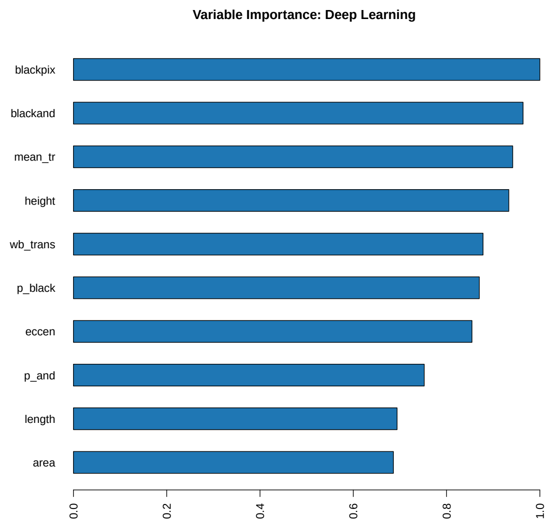


Figure 11: Neural Network Final Model Variable Importance

2.6 Plotted Network

The plotted network was too large to plot graphically, so a smaller model was fitted and plotted to view the inner workings of a neural net. This can be seen in Figure 12.

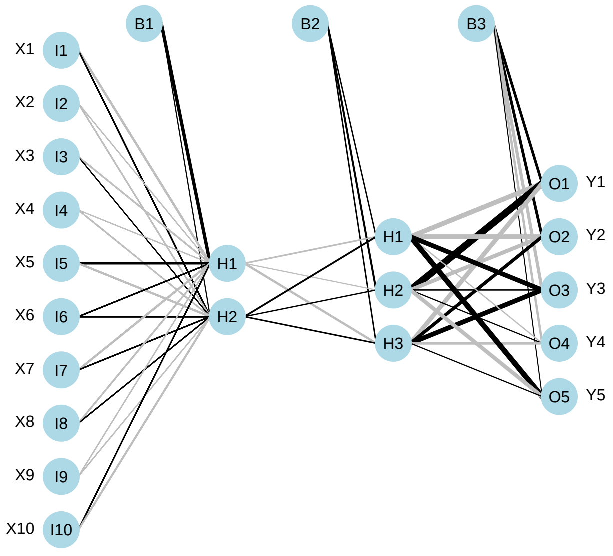


Figure 12: Neural Network Plot

3 Support Vector Machines

3.1 Brief Overview

Support Vector Machines are largely used in a classification setting, they do this by creating a hyperplane which splits the classes most accurately. Figure 13 shows the hyperplane between the blue and green classes.

3.1.1 Support Vectors

The support vectors are the points on the parallel lines which are closest to the hyperplane on each side. If these points are moved it would change the position of the hyperplane, and therefore they are extremely important to the model. Figure 13 shows the support vectors and the hyperplane they create.

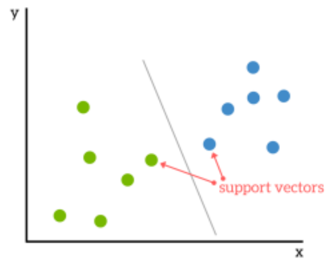


Figure 13: Hyperplane between classes [2]

3.1.2 Hyperplane

A hyperplane is a plane of n-1 dimensions that linearly separates the data. The further away a point lies away from the hyperplane, the more confident we can be that it's correctly classified.

Since it is in our best interest to to keep points as far away from the hyperplane as possible, we want to maximise the margin(distance between the hyperplane and support vectors) between the the two classes. Figure 14 shows the margin which should be maximised.

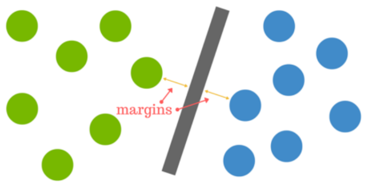


Figure 14: Margin between classes [2]

3.1.3 Non-separable Data

Unfortunately most datasets we work with are not linearly separable, such as the data shown in Figure 15

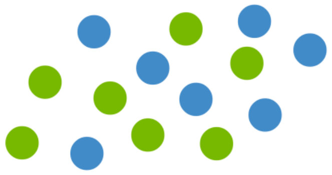


Figure 15: Non separable data [2]

A solution to this problem is a method known as kerneling.This consists of transforming the data into a higher dimension. This can be visualised in Figure 16 where a 2D data set is transformed into a 3D data set, for which a 2D hyperplane could linearly separate the dataset.

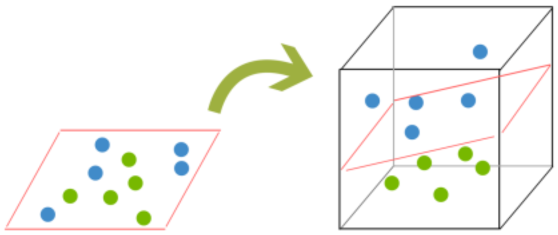


Figure 16: Kerneling [2]

3.1.4 Cost Penalty

A cost penalty can be added to the support vector classifier equation as a form of regularization. This allows incorrect points to violate the outer hyperplanes making the model less sensitive to the support vectors and more robust.

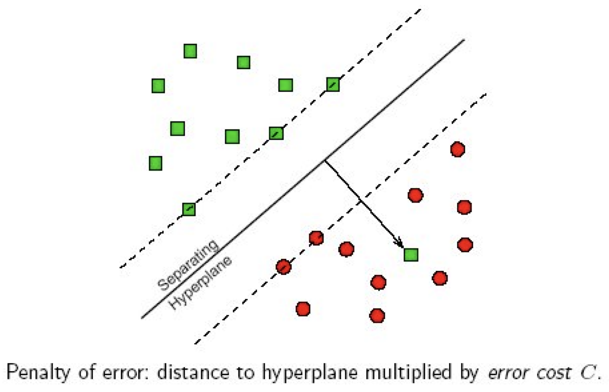
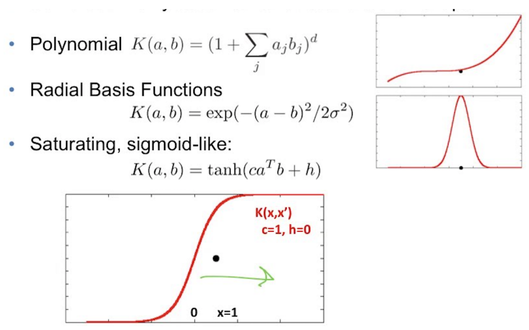

Separating Hyperplane
Penalty of error: distance to hyperplane multiplied by *error cost* C.

Figure 17: Cost violations [3]

3.1.5 Types of Kernels

Some kernels are shown in Figure 18. These each transform the data in different ways



- Polynomial $K(a,b) = (1 + \sum_j a_j b_j)^d$
- Radial Basis Functions $K(a,b) = \exp(-(a-b)^2/2\sigma^2)$
- Saturating, sigmoid-like: $K(a,b) = \tanh(ca^T b + h)$

Figure 18: Types of kernels [6]

3.1.6 Multi-class classification

SVMs can not be used directly for binary classification, but there is a way to make use of them for multi-class classification by fitting multiple SVMs. There are two methods for doing this - one-versus-one and one-versus-all.

3.2 Initial Model

The initial model was created using all of e1017's library SVM function defaults, this will be the base model and the model to beat. Parameter tuning will be done in the next section using the tuning function for different kernels.

	1	2	3	4	5
1	3079	14	0	2	3
2	30	173	0	1	0
3	9	0	6	0	0
4	11	1	0	41	0
5	44	0	0	0	33

Table 8: Support Vector Machine Initial Model Training Set Results

	1	2	3	4	5
1	1355	8	0	1	0
2	25	83	0	1	0
3	8	0	0	0	0
4	4	0	0	20	2
5	14	1	0	0	10

Table 9: Support Vector Machine Initial Model Test Set Results

The SVM predicts class 1 extremely well, whilst predicting the rest of the classes poorly. Class 3 even has a 100 % misclassification rate.

Class Number	Classification Percentage
1	99.34 %
2	76.15 %
3	00.00 %
4	76.92 %
5	40.00 %
Overall Test	95.82 %

Table 10: Support Vector Machine Initial Model Classification Percentages

The accuracy on the test set is 95.82 %, which gives a misclassification rate of 4.18 %

The accuracy on the training set is 96.66 %, which gives a misclassification rate of 3.34 %

3.3 Tuning

Multiple models are fitted with different tuning parameters and kernels. 10-fold cross validation is used to compare the models and select the best one. A comparison of the different kernels is given in Table 11.

Kernel	function
linear	$u' * v$
polynomial	$(gamma * u' * v + coef0)^d$
radial	$exp(-gamma * \|u - v\|^2)$
sigmoid	$tanh(gamma * u' * v + coef0)$

Table 11: Kernel Functions

Hyper-parameter	**Values**	**Reason for tuning**
Kernel(v)	polynomial	Transforms the data in different ways
Cost(u)	140	This value gives the inverse of the total violations allowed over the margin, the smaller the value the wider the margin and the smaller the chance is of overfitting
Gamma	0.01	This is used to tune all the kernels except for linear
Ceof0	0.1	This is used to tune the polynomial kernel
Degree	4	This specifies the degree of the polynomial kernel

Table 12: Support Vector Machine Tuning Parameters

Figure 19 shows the comparison of the different kernels on the dataset after their parameters have been tuned. From this it can be seen that the best kernel to use would be the polynomial kernel with a degree of 4. The graph shows how the error percentage decreases with the increased flexibility until it reaches degree 5 at which the model starts becoming overly complex for the underlying distribution.

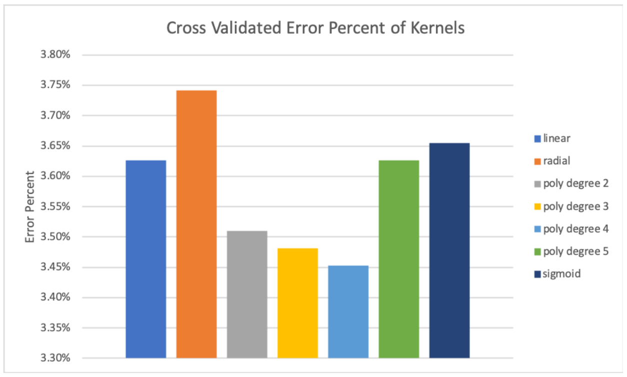
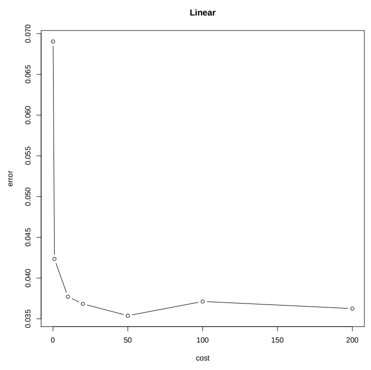
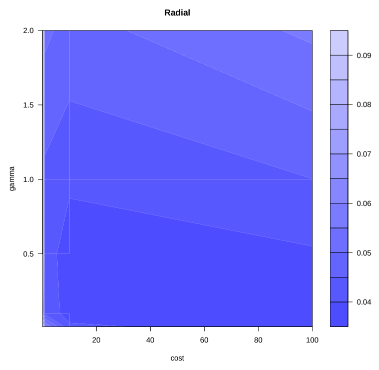


Figure 19: SVM kernels comparison

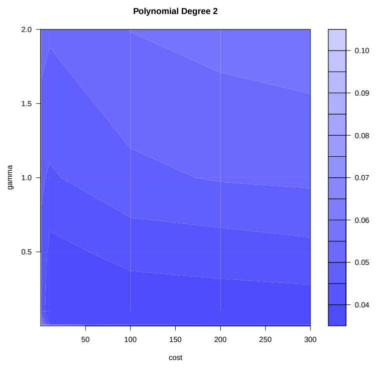
Figure 20 shows the contours of the tuning parameters (cost and gamma) for the different kernels. Other parameters were tuned but for the purpose of visualization were held constant at the values the tuning process had chosen.



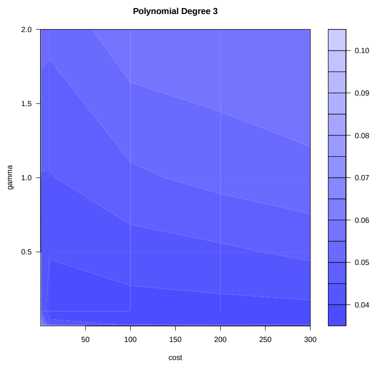
(a) linear



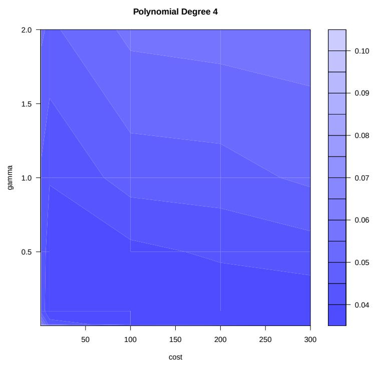
(b) radial



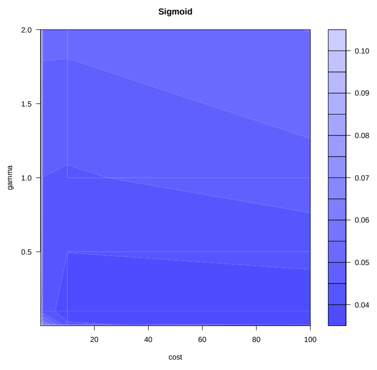
(c) polynomial degree 2



(d) polynomial degree 3



(e) polynomial degree 4



(f) sigmoid

Figure 20: Tuning Parameters

3.4 Final Model

From model tuning we can see the kernel that works best with the data is the polynomial kernel, and is the one that transforms the input space the best for linear separability. The results for this model are shown below, this can be visualised best by use of a confusion matrix.

	1	2	3	4	5
1	3089	6	0	1	2
2	67	136	0	1	0
3	9	0	6	0	0
4	14	1	0	38	0
5	47	0	0	0	30

Table 13: Support Vector Machine Final Model Training Set Results

	1	2	3	4	5
1	1363	0	0	0	0
2	37	70	0	2	0
3	3	0	4	0	1
4	5	0	0	20	1
5	14	0	1	0	10

Table 14: Support Vector Machine Final Model Test Set Results

The model got a 0 % misclassification rate on class one. There wasn't much improvement in the other classes except for class 3 now sitting at 50 percent.

Class Number	Classification Percentage
Class 1	100.00 %
Class 2	64.22 %
Class 3	50.00 %
Class 4	76.92 %
Class 5	40.00 %
Overall Test	95.76 %

Table 15: Support Vector Machine Final Model Classification Percentages

The test classification rate is 95.76, leaving the misclassification rate at 4.24 %. The train classification rate is 95.71 %, leaving the misclassification rate at 4.29 %. The cross Validation misclassification rate was 3.65 %.

Although the test classification percentage is slightly lower than the initial model's, the training and test errors are closer together indicating towards a model which will generalize better. The cross validation results also point towards this fact.

4 Model Comparison

The support vector machine is less prone to over fitting as it is less flexible. It also doesn't rely on huge amounts of hyper-parameters to be tuned. The kernels allow for a simple model to be used to capture a more complicated underlying function. Occam's razor principle is that the best solution is the simplest one, granted it is complicated enough. And so if the SVM is able to capture the complexity of the underlying function well enough, it will be a better solution as it has much lower variance.

Class Number	Classification %
1	97.21 %
2	95.24 %
3	50.00 %
4	92.31 %
5	80.00 %
Overall Test	96.65 %

Class Number	Classification %
Class 1	100.00 %
Class 2	64.22 %
Class 3	50.00 %
Class 4	76.92 %
Class 5	40.00 %
Overall Test	95.76 %

The final results of the models are shown in the tables above with neural network on the right and SVM on the left. From the final results it can be seen that the neural network works slightly better than the SVM, and is better at all around predicting the classes. This goes to show that the regularization implementations on the neural network helped the model to not over-fit. The SVM predicts class 1 extremely well, and because class 1 makes up the largest part of the test set it brings the average up substantially, despite the fact it predicts the other classes poorly. And so the model chosen to classify the unlabelled data was the neural network.

References

[1] Classification and regression with h2o deep. `http://docs.h2o.ai/h2o-tutorials/latest-stable/tutorials/deeplearning/index.html`. Accessed: 2019-03-20.

[2] Support vector machines: A simple explanation. `https://www.kdnuggets.com/2016/07/support-vector-machines-simple-explanation.html`. Accessed: 2019-04-29.

[3] Where is the cost parameter c in the rbf kernel in svm? `https://datascience.stackexchange.com/questions/5717/where-is-the-cost-parameter-c-in-the-rbf-kernel-in-svm`. Accessed: 2019-04-29.

[4] Zerium Aegeus. Artificial neural networks explained. `https://blog.goodaudience.com/artificial-neural-networks-explained-436fcf36e75`. Accessed: 2019-03-20.

[5] Niklas Donges. Recurrent neural networks and lstm. `https://towardsdatascience.com/recurrent-neural-networks-and-lstm-4b601dd822a5`, note = Accessed: 2019-03-20.

[6] Prof. Alexander Ihler. Machine learning and data mining. `https://www.youtube.com/watch?v=OmTu0fqUsQk`, note = Accessed: 2019-03-20.

[7] Krishna. Introduction to exponential linear unit. `https://medium.com/@krishnakalyan3/introduction-to-exponential-linear-unit-d3e2904b366c`. Accessed: 2019-03-20.

[8] Lachlan Miller. Machine learning week 1: Cost function, gradient descent and univariate linear regression. `https://medium.com/@lachlanmiller_52885/machine-learning-week-1-costfunction-gradient-descent-and-univariate-linear-regression-8`
note = Accessed: 2019-03-20.

[9] ujjwalkarn. A quick introduction to neural networks. `https://ujjwalkarn.me/2016/08/09/quick-intro-neural-networks//`. Accessed: 2019-03-20.

5 Appendix:

5.1 R Code: Neural Network

```r
suppressMessages(library(h2o))
suppressMessages(library(ggplot2))
suppressMessages(library(reshape2))

## initialising h2o

localH2O = h2o.init(ip = "localhost", port = 54321, startH2O
     = TRUE)
set.seed(123)

## stratified splitting of test, train and validation sets

frequencies = c(4420, 303, 23,79,100)

data_num <- read.csv("blocksTrain.csv")
data_num$class = factor(data_num$class)

ordered_data = data_num[order(data_num$class),]
training_numbers = round(frequencies*0.6,0)

sample_train_one = sample(1:frequencies[1], training_numbers
    [1])
diff_one = setdiff((1:frequencies[1]), sample_train_one)
sample_val_one = sample(diff_one, length(diff_one)/2)
sample_test_one = setdiff(diff_one, sample_val_one)
sample_train_one = sample_train_one
sample_val_one = sample_val_one
sample_test_one = sample_test_one

sample_train_two = sample(1:frequencies[2], training_numbers
    [2])
diff_two = setdiff((1:frequencies[2]), sample_train_two)
sample_val_two = sample(diff_two, length(diff_two)/2)
sample_test_two = setdiff(diff_two, sample_val_two)
sample_train_two = sample_train_two + frequencies[1]
sample_val_two = sample_val_two + frequencies[1]
sample_test_thwo = sample_test_two + frequencies[1]

sample_train_three = sample(1:frequencies[3], training_
    numbers[3])
diff_three = setdiff((1:frequencies[3]), sample_train_three)
sample_val_three = sample(diff_three, length(diff_three)/2)
sample_test_three = setdiff(diff_three, sample_val_three)
sample_train_three = sample_train_three + frequencies[1] +
    frequencies[2]
```

```
sample_val_three = sample_val_three + frequencies[1] +
    frequencies[2]
sample_test_three = sample_test_three+ frequencies[1] +
    frequencies[2]


sample_train_four = sample(1:frequencies[4], training_
    numbers[4])
diff_four = setdiff((1:frequencies[4]), sample_train_four)
sample_val_four = sample(diff_four, length(diff_four)/2)
sample_test_four = setdiff(diff_four, sample_val_four)
sample_train_four = sample_train_four + frequencies[1] +
    frequencies[2] +frequencies[3]
sample_val_four = sample_val_four + frequencies[1] +
    frequencies[2] +frequencies[3]
sample_test_four = sample_test_four + frequencies[1] +
    frequencies[2] +frequencies[3]

sample_train_five = sample(1:frequencies[5], training_
    numbers[5])
diff_five = setdiff((1:frequencies[5]), sample_train_five)
sample_val_five = sample(diff_five, length(diff_five)/2)
sample_test_five = setdiff(diff_five, sample_val_five)
sample_train_five = sample_train_five + frequencies[1] +
    frequencies[2] +frequencies[3]+frequencies[4]
sample_val_five = sample_val_five + frequencies[1] +
    frequencies[2] +frequencies[3]+frequencies[4]
sample_test_five = sample_test_five + frequencies[1] +
    frequencies[2] +frequencies[3]+frequencies[4]


training_sample = c(sample_train_one, sample_train_two,
    sample_train_three, sample_train_four,
                    sample_train_five)
validation_sample = c(sample_val_one, sample_val_two, sample
    _val_three, sample_val_four,
                      sample_val_five)
test_sample = c(sample_test_one, sample_test_two, sample_
    test_three, sample_test_four,
                sample_test_five)

train = sort(as.integer(training_sample))
val = sort(as.integer(validation_sample))
test = sort(as.integer(test_sample))
datTrain = data_num[train,]
datTrain = as.data.frame(datTrain)
datTest = data_num[test,]
datTest = as.data.frame(datTest)
datVal = data_num[val,]
datVal = as.data.frame(datVal)
```

```
datTrain_h2o <- as.h2o(datTrain)
datTest_h2o = as.h2o(datTest)
datVal_h2o = as.h2o(datVal)

## fitting the initial model

model <- h2o.deeplearning(x = 2:11,  # column numbers for
    predictors
                          y = 12,   # column number for
                              label
                          training_frame = datTrain_h2o,
                          #validation_frame = datTest_h2o,
                          nfolds = 10,
                          fold_assignment = "Stratified",
                          export_weights_and_biases = T,
                          seed = 123)


summary(model)

## plotting the variable importance

h2o.varimp_plot(model)

## Evaluating performance

yhat_train <- h2o.predict(model, datTrain_h2o)$predict
yhat_train <- as.factor(as.matrix(yhat_train))
yhat_test <- h2o.predict(model, datTest_h2o)$predict
yhat_test <- as.factor(as.matrix(yhat_test))

print(table(yhat_train, data_num[train,]$class))
print(table(yhat_test, data_num[-train,]$class))

correctRate = sum(yhat_test==datTest$class)/length(datTest$
    class)
correctRate

## plotting a neural network

model_plot <- h2o.deeplearning(x = 2:11,  # column numbers
    for predictors
                          y = 12,   # column number for
                              label
                          training_frame = datTrain_h2o,
                          #validation_frame = datTest_h2o,
                          hidden = c(2,3),
                          export_weights_and_biases = T,
                          seed = 123)
```

```r
library(NeuralNetTools)
wts <- c()

for(l in 1:(length(model_plot@allparameters$hidden)+1))
{
  wts_in <- h2o.weights(model_plot, l)
  biases <- as.vector(h2o.biases(model_plot, l))
  for (i in 1:nrow(wts_in))
  {
    wts <- c(wts, biases[i],as.vector(wts_in[i,]))
  }
}

struct <- model_plot@model$model_summary$units

plotnet(wts, struct = struct)

## grid search

epochs <- c(5,50,100,150)
activations <- c("Maxout", "Tanh", "Rectifier")
hidden <- list(c(5,5,5,5), c(30,30,30), c(60,60), c(120))
input <- c(0.1, 0.2, 0.3)
l1 <- c(0.001,0.0001)
l2 <- c(0.001, 0.001)
stopping_rounds <- c(4,5,6)
stopping_tolerance <- c(0, 0.001, 0.01)
output <- c(0.1, 0.2, 0.3)
rho <- c(0.999, 0.99, 1.1)
epsilon <- c(1e-10, 1e-8, 1e-6)
balance_classes = c(T,F)

hyper_params <- list(epochs = epochs, activation =
    activations, hidden = hidden,
                    input_dropout_ratio = input, hidden_
                        dropout_ratios = output,
                    stopping_tolerance = stopping_tolerance
                        ,
                    rho = rho, epsilon = epsilon, l1 = l1,
                        l2=l2,
                    balance_classes = balance_classes)

## running grid search

grid <- h2o.grid(algorithm = "deeplearning",
                hyper_params = hyper_params,
                x = 2:11, y = 12,
                training_frame = datTrain_h2o,
                 validation_frame = datVal_h2o,
                 standardize = T,
                 nfolds = 10,
                 seed = 1,
                 fold_assignment="Stratified",
                 mini_batch_size = 50)

## best model

grid@summary_table[1,]
best_model <- h2o.getModel(grid@model_ids[[1]])
best_model

print(best_model@allparameters)
print(h2o.performance(best_model))
print(h2o.logloss(best_model, valid=T))

h2o.confusionMatrix(best_model, valid = TRUE)

## visualising grid search

models=list()
for(i in 1:8){
  models[i] = h2o.getModel(grid@model_ids[[i]])
}


xlabel = NULL
for(i in 1:8){
  xlabel[i] = paste(round(models[[i]]@allparameters$epochs
      ,0), models[[i]]@allparameters$activation, models[[i]]
      @allparameters$hidden)
}

# TEST SET
res_devTest = NULL
for (i in 1:8) {
  res_devTest[i] <- h2o.logloss(h2o.performance(h2o.getModel
      (grid@model_ids[[i]]), newdata = datTest_h2o))
}

# TEST SET
res_devVal = NULL
for (i in 1:8) {
  res_devVal[i] <- h2o.logloss(h2o.performance(h2o.getModel(
      grid@model_ids[[i]]), newdata = datVal_h2o))
}

par(mfrow = c(1,1))
```

```r
plot(res_devTest[1:8], xaxt = "n", xlab='Grid␣Parameters␣(
    epochs,␣activation,␣hidden)', type = "b",col ="red",
      ylim = c(0.1,0.2),ylab = 'logloss')
axis(1, at=1:8, labels=xlabel[1:8], cex.axis = 0.6)
lines(res_devVal[1:8], type = "o", col ="blue")
legend("bottomright",c("Val","Test"),pch = 21, pt.bg = "
    white", lty = 1, col = c("blue", "red"))

## final model

model <- h2o.deeplearning(x = 2:11,
                          y = 12,
                          training_frame = datTrain_h2o,
                          validation_frame = datTest_h2o,
                          nfolds = 10,
                          hidden = c(120),
                          epochs = 50,
                          fold_assignment = "Stratified",
                          export_weights_and_biases = T,
                          seed = 123,
                          activation = "MaxoutWithDropout",
                          input_dropout_ratio = 0.1,
                          hidden_dropout_ratios = 0.1,
                          l1= 0.0001,
                          l2 = 0.001,
                          rho = 0.999,
                          epsilon = 1e-10)


summary(model)
print(model@parameters)
head(as.data.frame(h2o.varimp(model)))
h2o.varimp_plot(model)

## Evaluating performance

yhat_train <- h2o.predict(model, datTrain_h2o)$predict
yhat_train <- as.factor(as.matrix(yhat_train))
yhat_test <- h2o.predict(model, datTest_h2o)$predict
yhat_test <- as.factor(as.matrix(yhat_test))

print(table(yhat_train, data_num[train,]$class))
print(table(yhat_test, data_num[test,]$class))

correctRate = sum(yhat_test==datTest$class)/length(datTest$
    class)
correctRate

print(head(yhat_test))
print(head(datTest$class))
```

```
## unlabeled data

data_unlab <- read.csv("blocksTestNoLabel.csv")
datUnlab_h2o <- as.h2o(data_unlab)

yhat_unlab <- h2o.predict(model, datUnlab_h2o)$predict
yhat_unlab <- as.factor(as.matrix(yhat_unlab))

combined = cbind(data_unlab$ID,yhat_unlab)
colnames(combined) = c("ID", "Class")


write.csv(combined, file = "unlabeledpred.csv", row.names =
    FALSE)
```

5.2 R Code: Support Vector Machine

```
library(e1071)
library(kernlab)

## read in data

data <- read.csv("blocksTrain.csv")
data$class = factor(data$class)

##  class frequencies

frequencies = c(4420, 303, 23,79,100)

## barplot of frequencies

barplot(frequencies, col = c(2,3,4,5,6), xlab = "Class␣
    Number", ylab = "Number␣of␣entries␣in␣data",
         names.arg = c("1", "2", "3", "4", "5"))


set.seed(123)

## ordering data for stratisfied seperation

ordered_data = data[order(data$class),]
training_numbers = round(frequencies*0.7,0)

## row values for stratisfied seperation

sample_class_one = sample(1:frequencies[1], training_numbers
    [1])
sample_class_two = sample(1:frequencies[2], training_numbers
    [2]) + frequencies[1]
```

```
sample_class_three = sample(1:frequencies[3], training_
    numbers[3]) + frequencies[1] +frequencies[2]
sample_class_four = sample(1:frequencies[4], training_
    numbers[4]) + frequencies[1] +frequencies[2]
+ frequencies[3]
sample_class_five = sample(1:frequencies[5], training_
    numbers[5]) + frequencies[1] +frequencies[2]
+ frequencies[3] + frequencies[4]

training_sample = c(sample_class_one, sample_class_two,
    sample_class_three, sample_class_four,
                  sample_class_five)

train_data = data[training_sample,-1]
test_data = data[-training_sample,-1]

## fitting initial model and predicting

svm.model <- svm(class ~ .  , data = train_data)
svm.pred <- predict(svm.model, test_data[,-c(length(test_
    data))])

## test results
svm.model$index
mean(svm.pred==test_data$class)

table(pred = svm.pred, true = test_data$class)
correctRate = sum(svm.pred==test_data$class)/length(test_
    data$class)
misRate=1-correctRate
misRate

## train results

svm.pred.train <- predict(svm.model, train_data[,-c(length(
    train_data))])
svm.model$index
mean(svm.pred.train==train_data$class)

table(pred = svm.pred.train, true = train_data$class)
correctRate = sum(svm.pred.train ==train_data$class)/length(
    train_data$class)
misRate=1-correctRate
misRate

## tuning linear kernel

tune.out.linear <- tune(svm,class ~ . ,data = train_data,
    kernal = "linear",
                 ranges = list(cost = c
                    (0.1,1,10,20,50,100,200)))


summary(tune.out.linear)
plot(tune.out.linear, main = 'Linear')

## tuning radial kernel

tune.out.radial <- tune(svm,class ~ . ,data = train_data,
    kernal = "radial",
                 ranges = list(cost = c(0.1,1,10,100),
                               gamma = c(0.01, 0.1,0.5,1,2))
                                   )

summary(tune.out.radial)
plot(tune.out.radial, main = 'Radial')

## tuning polynomial kernels

tune.out.poly1 <- tune(svm,class ~ . ,data = train_data,
    kernal = "polynomial",
                      degree = 1, coef0 = 0.1,
                 ranges = list(cost = c
                    (0.1,1,10,100,200,300),
                               gamma = c(0.001,0.01,
                                   0.1,0.5,1,2)))

tune.out.poly2 <- tune(svm,class ~ . ,data = train_data,
    kernal = "polynomial",
                       degree = 2, coef0 = 0.1,
                       ranges = list(cost = c
                           (0.1,1,10,100,200,300),
                                      gamma = c(0.001,0.01,
                                          0.1,0.5,1,2)))

tune.out.poly3 <- tune(svm,class ~ . ,data = train_data,
    kernal = "polynomial",
                       degree = 3, coef0 = 0.1,
                       ranges = list(cost = c
                           (0.1,1,10,100,200,300),
                                      gamma = c(0.001,0.01,
                                          0.1,0.5,1,2)))

tune.out.poly4 <- tune(svm,class ~ . ,data = train_data,
    kernal = "polynomial",
                       degree = 4, coef0 = 0.1,
                       ranges = list(cost = c
                           (0.1,1,10,100,200,300),
                                      gamma = c(0.001,0.01,
                                          0.1,0.5,1,2)))
```

```
tune.out.poly5 <- tune(svm,class ~ . ,data = train_data,
    kernal = "polynomial",
                          degree = 5, coef0 = 0.1,
                          ranges = list(cost = c
                              (0.1,1,10,100,200,300),
                                          gamma = c(0.001,0.01,
                                              0.1,0.5,1,2)))

summary(tune.out.poly5)
plot(tune.out.poly5, main ="Polynomial␣Degree␣5")

par(mfrow=c(1,1))
dev.off()
plot(tune.out.poly1, main ="polynomial␣degree␣1")
plot(tune.out.poly2, main ="polynomial␣degree␣2")
plot(tune.out.poly3, main ="polynomial␣degree␣3")


## tuning sigmoid kernel

tune.out.sig <- tune(svm,class ~ . ,data = train_data,
    kernal = "sigmoid", coef0 = 0.1,
                   ranges = list(cost = c(0.1,1,10,100),
                                  gamma = c(0.001,0.01,
                                      0.1,0.5,1,2)))


summary(tune.out.sig)
plot(tune.out.sig, main = 'Sigmoid')


## bar graphs of final tuned kernels

labels = c('linear','radial' ,'polynomial␣degree␣2', '
    polynomial␣degree␣3','polynomial␣degree␣4',
            'polynomial␣degree␣5', 'sigmoid')
errors = c(0.03626643 , 0.03741827 , 0.03510195, 0.03481547,
     0.03452393 , 0.03626643 , 0.03655207 )

## final kernel tuning

tune.out.poly4 <- tune(svm,class ~ . ,data = train_data,
    kernal = "polynomial",
                          degree = 4, coef0 = 0.1,
                          ranges = list(cost = 140,
                                          gamma = 0.01))

summary(tune.out.poly4)
```

```
## initial model cross validation

tune.out.initial <- tune(svm,class ~ . ,data = train_data,
                       ranges = list(kernel = "linear"))

summary(tune.out.initial)


## final model

svm.model <- svm(class ~ .  , data = train_data, kernel = "
   polynomial",
                 degree = 4, coef0 = 0.1, cost = 140, gamma
                     = 0.01)
svm.pred <- predict(svm.model, test_data[,-c(length(test_
   data))])


## test results
svm.model$index
mean(svm.pred==test_data$class)

table(pred = svm.pred, true = test_data$class)
correctRate = sum(svm.pred==test_data$class)/length(test_
   data$class)
misRate=1-correctRate
misRate

## train results

svm.pred.train <- predict(svm.model, train_data[,-c(length(
   train_data))])
svm.model$index
mean(svm.pred.train==train_data$class)

table(pred = svm.pred.train, true = train_data$class)
correctRate = sum(svm.pred.train ==train_data$class)/length(
   train_data$class)
misRate=1-correctRate
misRate
```