Statistical Learning: Support Vector Machines and Neural Networks

Lecturers:
Şebnem Er
Stefan Britz

STA5077A - Supervised Learning

Due: 2019/05/02

Chris Maree - MRXCHR013

Abstract

This paper presents Support Vector Machines and Neural Networks as two classification techniques to label different document page layouts. A number of different support vector kernels were implemented, with the best proving to be radial and polynomial. Cross validated grid based tuning was implemented to refine the ideal kernels prediction accuracy, with the best model yielding a validation accuracy of 97.46%. Neural networks were created and refined using the `H2o` package with the best model being a model with three hidden layers with 500 neurons per layer. An ensemble model was created, combining the best models from both the support vector machine and neural network collections which yielded a validation accuracy of 97.56%.

Contents

1 Introduction

This report consists of two main sections that explore classification of blocks of page layout based on Support Vector Machine (SVM) and Neural Network (NN). This kind of analysis is important in separating text from graphics. The report begins by first breaking down the data provided and then moves onto each classification method in turn. Each section contains a theoretical description of each technique, followed by an implementation and test result discussion. Different hyper-parameter and regularization techniques are motivated and then finally the two classes of methods are compared and the best model is presented. The source code associated with each question can be found in the appendix in the form of commented and annotated R markdown notebooks. In addition, this source code can be found on Github here.

SVMs and NNs are two sophisticated methods of creating classification models. They build on other simpler methods of classification and offer a range of advantages which are explored to give context of their value amongst the myriad of other classification techniques. Both techniques do however have their own drawbacks, which are also explored below.

Advantages of SVMs are that they deliver a unique solution, as a result of the optimization methodology being convex, making them very robust over different samples[1]. Importantly, SVMs support kernels, enabling them to model non-linear relations in data. The main disadvantage with SVMs is that they can't perform feature selection as they do not identify important variables through weighted magnitudes.

NNs tend to have a higher classification accuracy than many more traditional classification techniques. This has a lot to do with the ability for NNs to model non-linear and complex relationships which is important when modeling real-life data. NNs can also generalize well when compared to other methods making over fitting hard. NNs do not impose any assumptions about the distribution of the data and so can better model heteroskedastic data[2]. Some disadvantages with NNs are that they can potentially have multiple solutions associated with different local minima and so are not always robust over different samples. They are also extremely hard to interpret as they model relationships as a black box and so no significance can be inferred from the weightings. They tend to rely on more data than traditional models, but this is becoming less of an issue with ever growing datasets.

2 Data Exploration

The training dataset provided contains 4925 observations from 54 distinct documents. Each observation consists of 10 different attributes that are used to describe the document. In addition to the training data, a set of 549 rows of unlabeled test data was also provided. The attributes for both sets can be seen in Table 1 below, along with their description.

Attribute	**Description**
height	Height of the block.
length	Length of the block.
area	Area of the block (height length).
eccen	Eccentricity of the block (length / height).
p_black	Percentage of black pixels within the block (blackpix / area).
p_and	Percentage of black pixels after the application of Run Length Smoothing (RLSA) (blackand/area).
mean tr	Mean number of white-black transitions (blackpix / wb trans).
blackpix	Total number of black pixels in the original bitmap of the block.
blackand	Total number of black pixels in the bitmap of the block after the RLSA.
wb_trans	Number of white-black transitions in the original bitmap of the block.

Table 1: All Attributes and their descriptions for the provided datasets. All attributes are numeric.

The provided data has five classes, of `text`, `horizontal line`, `picture`, `vertical line` and `graphic`. All classes were encoded in R to a factor using the `as.factor` function to assign numerical values from $1 \rightarrow 5$ for each class.

In order to fairly compare different models a validation set was created which is held out until the very end and used to quantify the quality of the best model chosen. 20% of the data was selected for this purpose.

In order to capture the testing and validation results for all different experiments conducted, a two custom function was created to add the testing and validation results to a `data.frame()` that is reused between experiments. This data frame can then be printed later on to generate tables that have results from all used models. In order to standardize the inputs to this function it expects the output of a `confusionMatrix` function call from the Caret package.

3 Support Vector Machines

SVM classification models are implemented in this section to identify page layouts and content within documents. First a technical overview of the underlying theory is explored, laying the theoretical understanding upon which the implementation builds. After the theory the implementation is discussed followed by an analysis of the results.

3.1 Support Vector Machines: Theory

Fundamentally SVM approach a two-class classification problem by finding a plane that separates the classes in the features space. A p dimensional space can be partitioned by a $p-1$ dimensional hyperplane described by the linear Equation 1. In a two dimensional case, this hyperplane will be a line. In a three dimensional case, this hyperplane will be a flat plane.

$$\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_p X_p = 0 \tag{1}$$

If it is possible to construct a hyperplane that perfectly separates observations according to class, such that points above the hyperplane are positive and those below the hyperplane are negative, then a *separating hyperplane* classifier exists. Numerically, this occurs as a result of the sum of squares of all β_p values being 1, meaning that the direction vector β from the origin to the hyperplane is a unit vector.

If observations are labeled between each class (in the binary case) as $y_i = 1$ and $y_i = -1$ for two different classes then a separating hyperplane can be shown to be represented as Equation 2 for each observation y_i.

$$y_i(\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_p X_p) > 0 \tag{2}$$

Identifying the best separating hyperplane can be then done using a *maximal margin classifier*. Selecting the best hyperplane among all possible hyperplanes involves finding the one that makes the largest gap between the two classes. This is done by constructing an optimization problem to maximize the margin M between the classes, subject to $\sum_{j=1}^{p} \beta_j^2 = 1$ due to the restriction imposed by the unit vector β. This ensures that the magnitude of the values for each class corresponds to the distance from the hyperplane, as shown in Equation 2. Additionally, as the maximal margin classifier ensures that no observations fall on the wrong side of the margin, an additional restriction is imposed as $y_i(\beta_0+\beta_1 X_1+\beta_2 X_2+\ldots+\beta_p X_p) \geq M$ for all $i = 1, \ldots, N$ to force all observations to fall on the correct side of the hyperplane. This method however is restricted to situations where data is linearly separable, which is not always the case.

To deal with non-separable data (as well as noise in the data) the maximal margin classifier can be extended to consider a *soft margin* which relaxes the idea of the separating hyperplane. This implementation allows for a threshold of observations to land on the wrong side of the margin (and even the

hyperplane). Adjusting the size of the soft margin acts as a form of regularization as allowing some points to lie on the wrong side of the margin enables the margin to be determined by more than just the closest points.

The formulation of the optimization problem defined for the maximal margin is now modified to accommodate this soft margin. Fundamentally the problem still maximizes the margin M, subject to $\sum_{j=1}^{p} \beta_j^2 = 1$. The second restriction implemented on the magnitude of the sum of all points now includes a discount factor ϵ_i as $y_i(\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_p X_p) \geq M(1 - \epsilon_i)$ to allow some slack, enabling some points to fall on the wrong side of the margin. The sum of all slack values ϵ_i is restricted through the use of a budget C defined by $\epsilon_i \geq 0, \sum_{j=1}^{p} \epsilon_i \leq C$. In other words, the ϵ_i tells us how much each point is allowed to be on the wrong side of the margin, constrained within the total allowed budget of C. C is considered a regularization parameter that will increase (or decrease) the size of the soft margin around the hyperplane, and is experimented with later when an ideal SVM model is generated. In general, the larger the C value the more stable the margin becomes resulting in a bias variance trade off.

In order to deal with non-linear data the feature space can be enlarged by polynomial dimension transformations. This takes the p-dimensional space to a higher dimensional space allowing for separations to form within the data at the higher dimensions. A linear support vector classifier is then fitted in the enlarged feature space, resulting in a non-linear decision boundary in the original space.

This process of feature expansion is however computationally expensive and so *kernels* are introduced to greatly reduce the cost in feature space expansion. Kernels leverage the notion of inner products, as the sum of cross products of the individual components of the vector represented for two vectors x_i and x_i' as $\langle x_i, x_{i'} \rangle = \sum_{j=1}^{p} x_{ij} x_{i'j}$. The use of kernels enables the feature space to be expanded without having to consider all possible terms in the expansion.

Equation 3 shows a bivariate kernel function which computes the inner-product needed for a d dimensional polynomial feature expansion. Using kernels with a $d > 1$ leads to higher dimensional support vector classification, resulting in a much more flexible decision boundary.

$$K(x_i, x_{i'}) = \left(1 + \sum_{j=1}^{p} x_{ij} x_{i'j}\right)^d \tag{3}$$

This bivariate kernel can then be used to find the support vector of the form shown in Equation 4. Importantly $\hat{\alpha}_i$ are non zero for only terms that are within the support set $\mathcal{S}$, the points that are immediately adjacent to the support vector and margin. These are the points that are used to define the support vetor its self.

$$f(x) = \beta_0 + \sum_{i \in \mathcal{S}} \hat{\alpha}_i K(x, x_i) \tag{4}$$

An example of another popular kernel function is the radial kernel with the form shown in Equation 5. This kernel would then be used with Equation 4 to generate the support vectors. This kernel leverages elements from a multivariate Gaussian distribution and includes a tuning parameter γ. This kernel amounts to using an inner product in an abstract, infinite dimensional feature space enabling the kernel to capture much of the possible non-linearity within a data-set.

$$K(x_i, x_{i'}) = e^{\left(-\gamma \sum_{j=1}^{p} (x_{ij} - x_{i'j})^2\right)} \tag{5}$$

The γ parameter defines how far the influence of a single training example reaches. A low value results in a far reach and higher values result in closer influence. It can be thought of as 1 over the standard deviation of the Gaussian distribution that the radial kernel leverages. This is another tuning parameter that is included in the models discussed later on in section 3.4.

Finding the ideal β values involves maximizing the classifier margin which is equivalent to minimizing the *hinge loss* function define in Equation 6. This function is similar in implementation to the loss function for Ridge and Lasso except it is capable of returning zero values for observations on the correct side of the margin. How this loss function is minimized is out of the scope of this paper.

$$L(\boldsymbol{X}, y, \beta) = \sum_{i=1}^{n} \max[0, 1 - y_i(\beta_0 + \beta_1 x_{i1} + \ldots + \beta_p x_{ip})] \tag{6}$$

The theory outlined up to this point only considers SVM's with two classes (binary). For contexts with more than two classes (such as the dataset provided for this assignment) there are two different possible approaches, as outlined below:

- One versus All (OVA) fits K different 2-class SVM classifiers against all other classes. In this implementation all other classes are grouped together and assigned a value of -1 and the class currently being considered is assigned a value of $+1$. This results in K different functions, with K different classifiers for all K classes. When a new point is classified it is applied to all K classifiers and the class that yields the highest value is chosen as the output class.
- One versus One (OVO) fits $\binom{K}{2}$ pairwise classifiers for each class against all other classes. When a new point is classified it is applied to all classifiers and the class that wins the most pairwise competitions is the chosen class.

3.2 Support Vector Machines: Feature Normalization

Before model fitting can occur the data needs to be normalized as SVMs assume that the data it uses is within a standard range. To this end all numeric predictor variables were standardized using the `scale` function in R.

3.3 Support Vector Machines: Implementation in R

The simplest implementation of a SVM in R can be achieved using the `SVM` function which is part of the e1071 package. This function takes in the variable relationships to be modeled, the data set and importantly a kernel and its associated tuning parameters. As was discussed in the theory section, the main kernel tuning parameters to consider here are cost (c) and gamma (γ) values which define the size of the soft margin and the influence of a single training example respectively. The implementation for this question can be found in Appendix 1.

To create the classification model one can either specify the formula as a symbolic description of the model to be fitted as `class~.` indicating that all variables are fit against `class` or specific `x`'s and `y`'s can be defined. The `SVM` function from e1071 contains four basic kernels: linear, polynomial, radial and sigmoid. Each were experimented with and the results compared later on in Section 3.4.

As the e1071 package has a relatively limited number of kernels available for model building the ksvm package was also used to gain access to a few other kernels. While there is clearly no guarantee that this will produce superior results to the limited set of kernels, it is still advantageous to experiment with a few more to understand the ideal configuration for this provided data set.

3.4 Support Vector Machines: Model Refinement And Tuning

Next the hyper parameters associated with SVM's are tuned. The tuning parameters of interest are the cost c which define the size of the soft margin and gamma (γ) which defines how far the influence of a single training example reaches. This hyper parameter applies specifically to polynomial kernels. Radial kernels tuning hyper parameter is sigma (σ) and this acts in a similar way to what γ does to polynomial

kernels. There is also another hyper parameter for SVMs, ϵ but this only applies to regression based SVMs and so is clearly not used in this classification context[3].

Tuning was initially preformed with the `tune` function which is a generic function for tuning hyperparameters of statistical methods using a grid search. The `tune` function uses randomized cross-validation by default when selecting ideal hyper parameters. A range of `cost` and gamma `gamma` values were provided to the tuning grid to identify the ideal combination. Fundamentally the values of c are γ and interconnected within a SVM meaning that you can't tune the one, then tune the other. Rather, they need to be tuned together in a grid like fashion where an ideal combination can be found[4]. To this end, a range of values were selected for `cost` and `gamma` with the best pair being selected with the best cross-validation accuracy. An exponentially growing sequence of values for c and γ was used in increasing exponents of 10. The details of these ranges and what values proved best is outlined in Section 3.4 and implementation can be found in Appendix 1.

In an effort to achieve a higher quality model the `train` function from the caret package was also implemented which utilizes repeated random cross validation in tuning and can be found in Appendix 1. Caret also implements cross-validation in parallel by using `foreach` in the backend and therefore makes the tuning of hyper parameters far quicker with a multicore CPU.

3.5 Testing, Results and Model Refinement

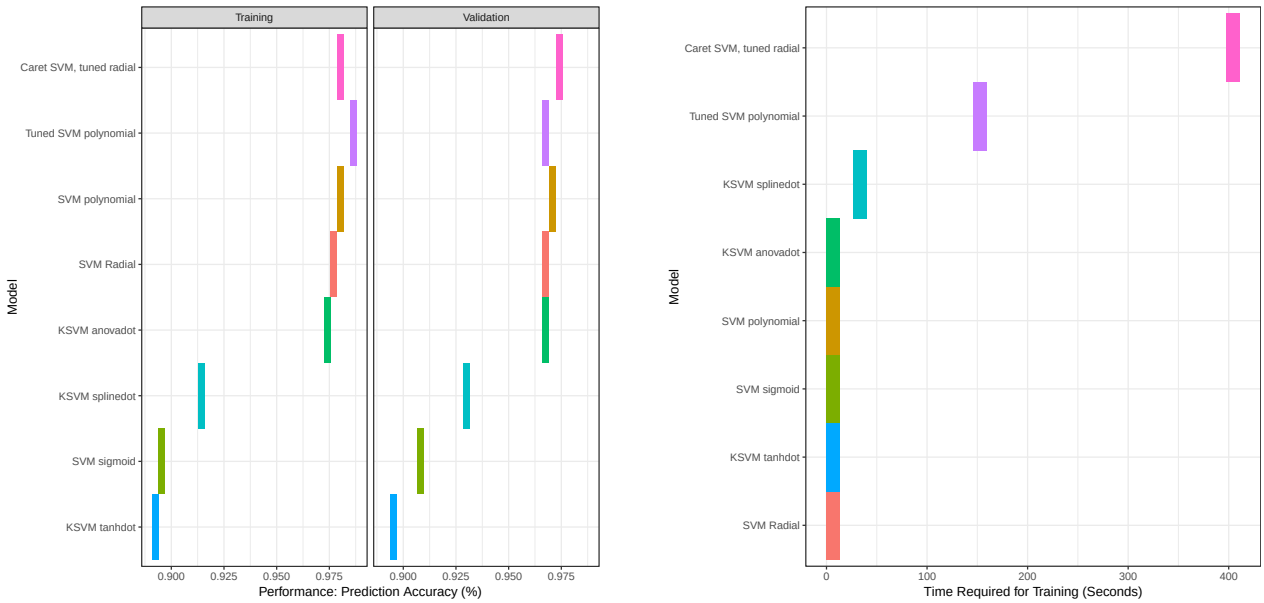
The training and validation results for the different SVM tuning parameters and kernels previously discussed are now broken down. Table 2 shows the key accuracy results for each kind of kernel and tuning method used in testing. Both training and validation accuracy is shown. Figure 1a shows the performance metrics for each kind of kernel implemented. Figure 1b shows the time that each kernel took to generate the respective SVM. From these results it is clear that radial, polynomial and anova kernels are the best preforming. Anova however took much longer to find a SVM of similar performance.

Model name	**Traing accuracy**	**Validation accuracy**	**Run time(s)**
Tuned SVM polynomial	0.9878	0.9675	158.95
Caret SVM, tuned radial	0.9817	0.9746	395.73
SVM polynomial	0.9812	0.9695	1.49
SVM Radial	0.9782	0.9675	0.9
KSVM anovadot	0.9749	0.9685	8.82
KSVM splinedot	0.9142	0.931	38.1
SVM sigmoid	0.8944	0.9076	0.88

Table 2: SVM training and validation accuracy and run time

The two best preforming SVM method in terms of training and validation accuracy were built using tuned kernels, as shown by two grey rows in Table 2. This is to be expected as finding these ideal values influences how model generalize. Interestingly the tuned SVM polynomial which was made using the `tune` function resulted in the highest testing accuracy, but not the highest validation accuracy. This leads one expect that this model has slightly over fit onto the data set. This is visually clear from Figure 1a where the best preforming model on the training set is the `Tuned SVM polynomial`. This model's accuracy is clearly higher than the other models, even the other tuned model. Then, on the validation side of the figure, the model has a noticeably lower value for it's accuracy, indicating over fitting.

The other tuned model used a radial kernel and was built using the `train` function from caret. This function enables the use of a `trainControl` which was implemented to preform repeated, random cross validation. This additional process of repeated cross validation could account for the increase in validation accuracy as the model has generalized better than the one created using `tune`. This repeated cross validation process will also account for why this model took so long to build, as shown in the second row of Table 2 and is clearly viable in Figure 1b.



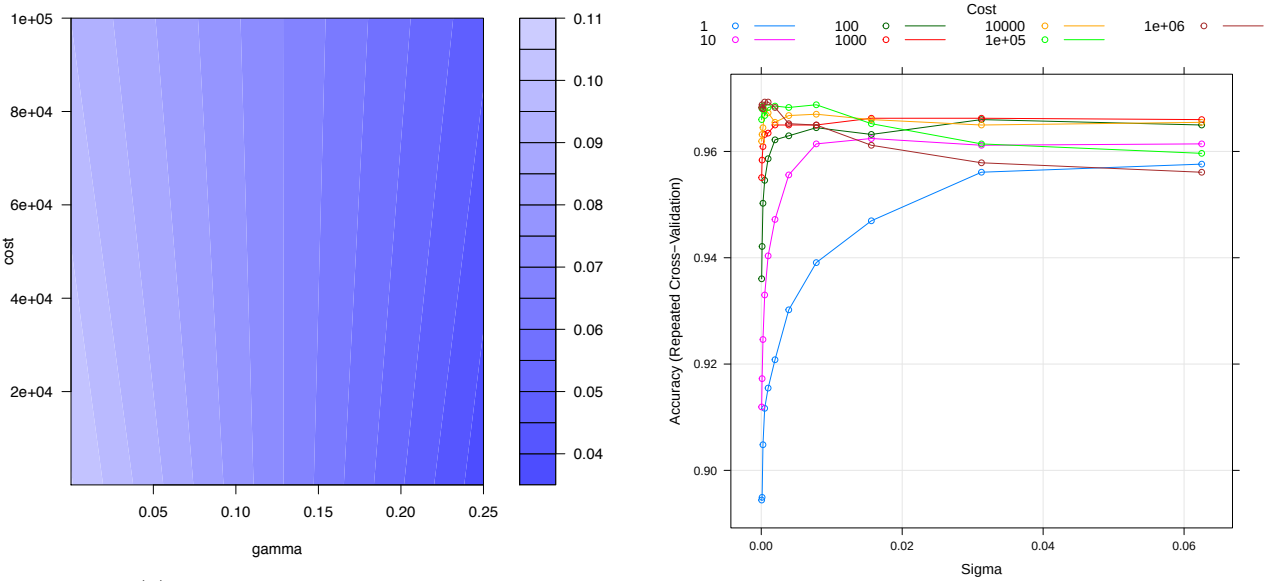
(a) Kernel accuracy, ordered by training accuracy

(b) SVM kernel compute time comparison

Figure 1: Plots comparing different kernel performance metrics

Figure 2a shows the accuracy vs γ and cost for the `tune` implemented version of the `polynomial` kernel. The γ tuning parameter is present(and not σ) as this is the relevant parameter (other than cost) for polynomial kernels. This implementation used a range of γ and cost values with γ ranging from $\gamma = 2^{-13} \rightarrow 2^{-2}$ and cost ranging from $2^0 \rightarrow 2^{13}$. This is a large range but enabled the ideal values to be found which was a value of gamma $\gamma = 0.25$ and cost $C = 8192$.

Figure 2b shows accuracy vs σ for different cost values for the `radial` kernel implementation. The σ tuning parameter only applies to radial kernels and so is the modified value in this case. A range of values for these tuning parameter were used with sigma $\sigma = 2^{-14} \rightarrow 2^{-4}$ and cost $C = 10^0 \rightarrow 10^6$. From this plot it can be seen that a good model should use a low σ and a higher cost value. The ideal model had sigma $\sigma = 0.000976$ and cost $C = 10^6$.



(a) Polynomial kernel tuning

(b) Radial kernel tuning

Figure 2: Kernel hyper parameter tuning and refinement

Training SVMs proved to be a rather slow and computationally expensive process as can be seen in figure 2b. The more refinement a model required the slower it took to build. There were some configurations that did not end up generating a model at all because they ran for too long and the training had to be stopped.

4 Neural Networks

This section explores the use of neural networks in creating a classification model used to detect to identify page layouts and content within documents. A high level overview of the underlying theory behind neural networks is first explored followed by implementation and testing in R.

4.1 Neural Networks: Theory

NNs learn by example and process information in a way that is largely inspired by the biological neuron system. NNs comprise a large number of highly interconnected processing neurons which together make collective decisions about a given input. Neurons take in a set of inputs, apply a weighting and bias process to them and then apply a non-linear activation function. The bias are only included in each non-output layer of the network. A generic implementation of a neuron can be seen in Figure 3.

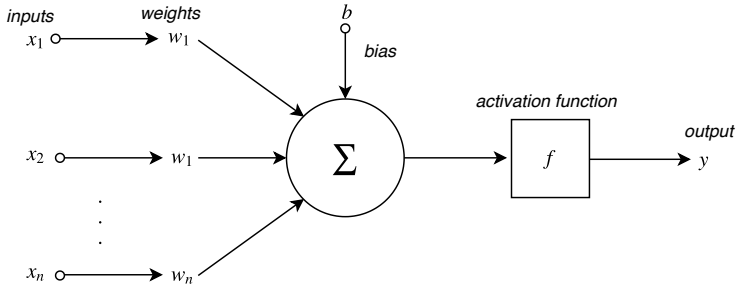


Figure 3: Inner workings of a basic sigmoid neuron

The activation function f can change based on the context of what the NN is trying to model but a common example is the sigmoid function which produces a value between 0 and 1 based on the magnitude of the inputs. This sigmoid activation function is defined by Equation 7 for a given input z. Other popular activation functions include identify, binary step, ramp and ReLu functions, some of wich are explored later.

$$\sigma(z) \equiv \frac{1}{1 + e^{-z}} \tag{7}$$

If the weighting and summing process is introduced into the sigmoid function 7 then the output y for given inputs $x_1, x_2, \ldots$ and weights $w_1, w_2, \ldots$ becomes Equation 8.

$$y = \frac{1}{1 + e^{-\sum_{i=0}^{n} w_i x_i - b}} \tag{8}$$

Multiple neurons can now be combined together to form a network of neurons consisting of an input layer to match the feature space, multiple hidden layers and an output layer to match the output space. The weighting w when multiple neurons are connected together defines the influence of one neuron on another. This configuration is known as a *multi-layer feedfoward neural network* and a simple example can be seen in Figure 4.

The weighting linking neurons together and the biases within each neuron fully determine the output of the neural network. Learning is therefore a process of changing these weightings and bias to minimize the *miss-classification quadratic cost function*, as shown in Equation 9. This is very similar to a mean squared error (MSE). This cost function is applied for the collection of all weights w, all biases b and n total training inputs. a is a vector representing the output of the network with a given input x.

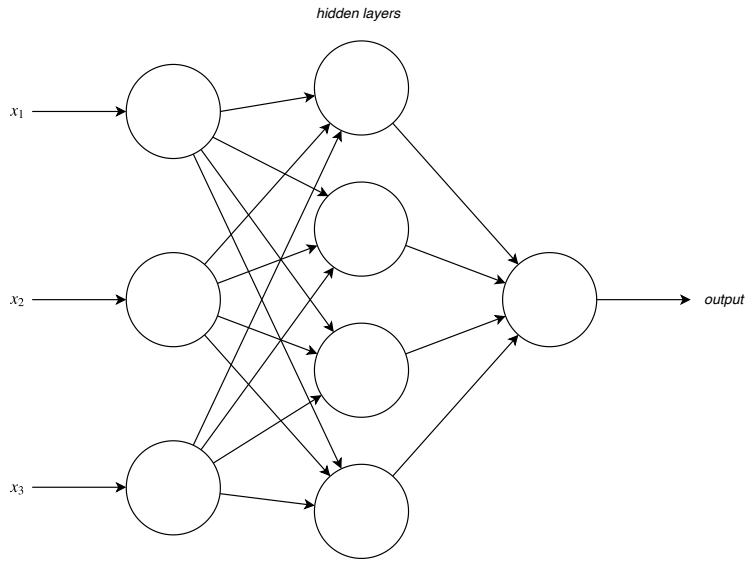


Figure 4: Simple three layered neural network

$$C(w,b) \equiv \frac{1}{2n} |y(x) - a|^2 \tag{9}$$

Networks that preform well in classifying the provided input will have $C(w,b) \approx 0$. It is therefore the goal of the optimization function to get this cost function as low as possible and in doing so select the optimal values for w and b. The minimization of this cost function to find the ideal weights and biases is achieved through a process called *gradient decent* which utilizes the backpropagation algorithm.

The backpropagation algorithm involves iterating through many cycles of two discrete processes to find the ideal weights and biases that yield the minimum cost value. This two step process consist of a forward phase where an input propagates through the network with each neuron applying it's weighting, bias and activation function to it, until a final output is generated at the final layer. The backward phase then compares this output to the true target value and uses this to iterate backwards through the network to modify the connection weightings and biases within the network to reduce errors.

4.2 Implementation in R

NNs were implemented in R using the `h2o.deeplearning` h2o function. A number of different model refinement and regularization techniques were then applied to identify the ideal implementation. Each implemented model with each regularization technique is repeated five times to account for random variance in the model building and regularization process. For each technique a common number of epochs was used. This defines the number of times to iterate over the data set while building the models. Each method applied is now discussed and the associated implementation can be found in Appendix 2.

4.2.1 Vanilla Neural Networks

Neural networks from h2o are implemented natively within the package as Multi-Layer Perceptron (MLP)[5]. This function's interface takes in a number of different parameters, including a number that can be used for tuning purposes. The basic implementation in `R` involves defining the number of hidden layers to build up within the neural network as well as the number of neurons within each layer. Four neural networks of this vanilla variety were created with all tuning parameters left at zero while the number of hidden layers and number of nodes within each layer was modified. The four ranges of hidden layers were defined by the column vector fed into the `h2o.deeplearning` function as `hidden=...` for columns consisting of: (50,50,50), (200,200,200), (200,200,200,200) and finally (500,500,500,500). In each case the number corresponds to the number of neurons within each hidden layer.

Model refinement and tuning was implemented for a number of different regularization techniques. Each implemented model with each regularization technique is repeated five times to account for random variance in the model building and regularization process. For each technique a common number of epochs was used. This defines the number of times to iterate over the data set while building the models.

4.2.2 Model Regularization and Generalization

Model regularization acts to reduce variance within models and therefore makes them better at generalizing unseen data sets. This results in models that are less likely to over fit. Three common methods for regularization in building neural networks are discussed including parameter penalization through L^1 and L^2 regularization, dropout regularization and early stopping.

4.2.3 L^1 and L^2 Regularization

The loss function defined before in Equation 6 can be extended to use a regularization penalty for large neuron weights. This modification to the loss function is shown in Equation 10 and is referred to as the L^2 regularization function. This loss function is then minimized in the process of stochastic gradient decent when building neural networks[6].

$$\begin{aligned} C &= -1\frac{1}{n}\sum_{jx}\left[y_j \ln a_j^L + (1-y_i)\ln\left(1-a_j^L\right)\right] + \frac{\lambda}{2n}\sum_w w^2 \\ C &= C_0 + \frac{\lambda}{2n}\sum_w w^2 \end{aligned} \tag{10}$$

The first term in this expression is the standard cross-entropy value which is minimized when finding a normal set of weightings for a NN and can be referred to as C_0 for simplicity sake, as shown in the second line of Equation 10. The second term is called the regularization parameter and acts as a penalty for large weights. Ranges of this regularization parameter λ are explored later on as a tunable hyper parameter.

L^1 regularization is built on the same basic idea that underlies L^2 but, instead of using a squared loss value for weights, an absolute is taken as shown in Equation 11 below. The major difference here is that L^1 regularization acts as a feature selection method as a result of the absolute being taken for each weighting, rather than the square.

$$C = C_0 + \frac{\lambda}{2n}\sum_w |w| \tag{11}$$

Note that L^1 penalties can't be used in gradient-based approaches of optimization as it is not differentiable, unlike the form of L^2[7]. This makes L^2 more computationally efficient as it has a closed form solution, while L^2 does not. L^2 regularization can help multicollinearity within a given set by constraining the coefficient norm and keeping all the variables (does not preform variable selection). L^1 regularization is similar to lasso and L^2 is similar to ridge regression in both process and form. In general L^2 regularization tends to provide better prediction accuracy than L^1 regularization in classification problems[6].

A range of values for L^1 and L^2 λ's were tested and compared using the `hyper_params` parameter within the `h2o.grid` function. Value ranges for L^1 and L^2 were defined as $(0, 0.00001, 0.0001, 0.001, 0.01)$. In order to find the ideal set of λ's for both penalty functions a range of different activation functions were also implemented for this regularization technique. Specifically `Rectifier`, `Maxout` and `Tanh` where implemented. As running a grid search over such a wide range of values can take a long time, a value of `max_runtime_secs` was defined as 600 seconds to keep the model building time under 10 minutes for each set of models to be built. No explicit values for `cost` or γ were specified at this point but rather `RandomDiscrete` was used to search over a random subset of the key hyper parameters. This extensive, deep grid based method might result in over fitting, something that is discussed in section 4.3 in more detail.

4.2.4 Dropout Regularization

Dropout regularization enables networks to be built in a generalized, regularized way without having to modify the cost function. The main idea behind dropout selection is to randomly remove neurons from the network during training. This acts to prevent each neuron from co-adapting together. Co-adapting occurs when neurons become coupled and dependent on other neurons within the network and therefore do not generalize well to unseen testing data. By introducing dropouts in the building process the co-adaption of neurons can be minimized.

Dropout regularization modifies the backpropogation algorithm by randomly removing half the hidden neurons in the network. An example of this is shown in Figure 5. Forward propagation of the input is done through the modified network, followed by standard back propagation to update neuron weightings and biases. This process is then repeated a number of times by choosing a new random subset of neurons to remove when each model is built. The final model consists of all neurons, resulting in twice as many as during training. As a result all neuron's weights are halved after building the network.

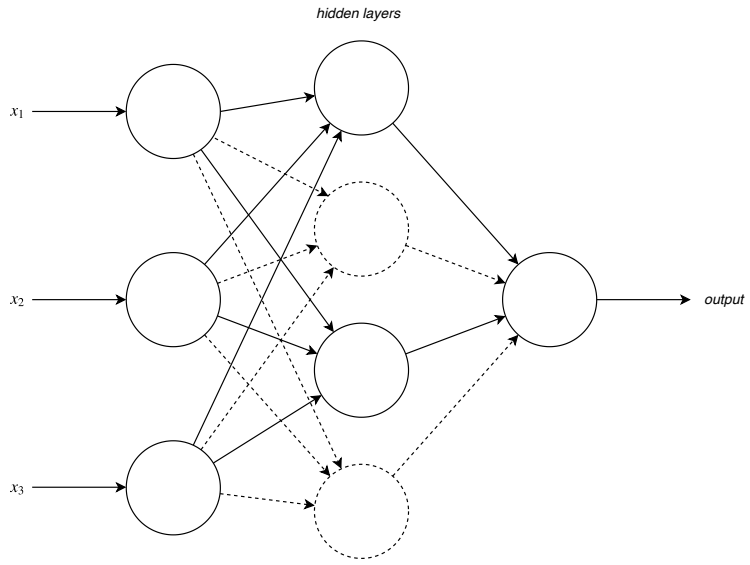


Figure 5: Dropout regularization network example

Dropout is configured in h2o using the `hidden_dropout_ratios` variable. The standard ratio for this is 0.5, meaning that half are dropped during model building as is the standard. Drop out rate values of $(0.5, 0.5, 0.5)$ were defined for all dropout model types built. `input_dropout_ratio` was also experimented with to try and improve generalization of the model. This rate specifies the number of input layers that should be dropped. Rates of 0.1 and 0.2 were experimented with, within the suggested range according to the documentation. The performance results for the different dropout methods can be seen in Section 4.3.

4.2.5 Early Stopping Regularization

Early stopping is a mechanism to automatically prevent over fitting when training a neural network. Early stopping works by computing the classification accuracy at the end of each epoch (iteration in building the network) and compares this to the previous accuracy. When the accuracy stops improving the learning process is terminated. This simplifies the process of setting epocs to an automatic process[8].

If the building process is stopped as soon as accuracy stops decreasing then the final model will most probably miss the ideal solution as it stopped while there are still more improvements to be had. A more general rule is to stop building when there has been a number of iterations that have not improved.

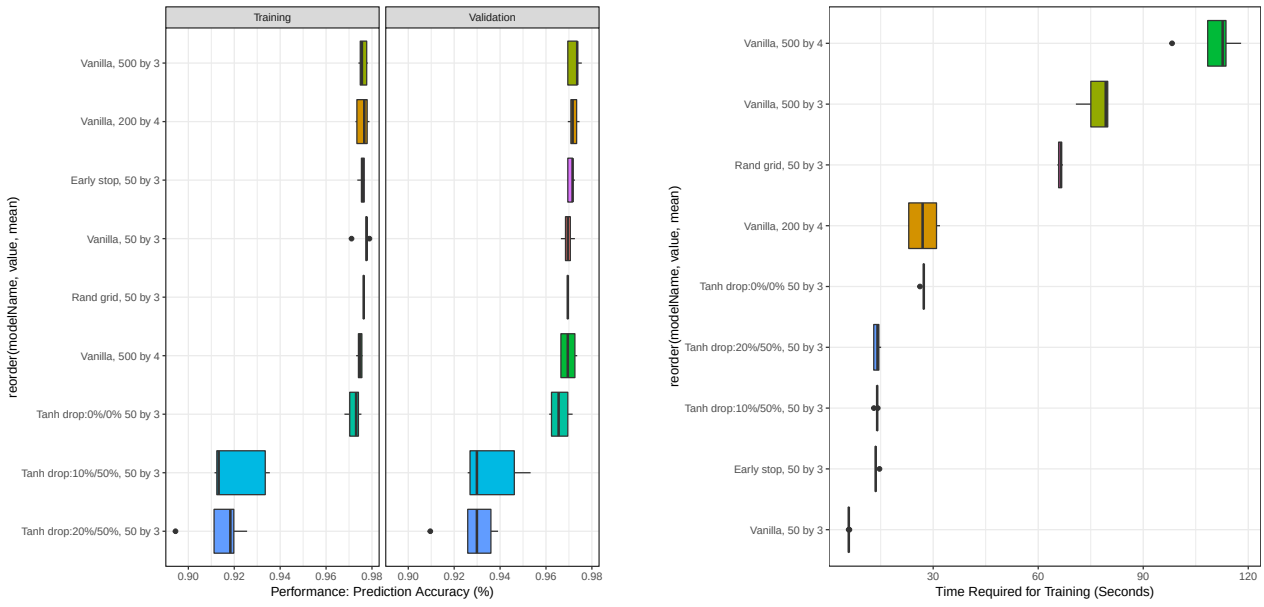
h2o implements the notation of early stopping through a few main tuning parameters. `stopping_rounds` defines the number of rounds that the training must not improve over to stop training. This value was set to 5. This is based off the `stopping_metric` criteria which is used to define how the error improvement rate is quantified. As this paper's implementation is for a classification algorithm this value is set to `"misclassification"` to quantify the misclassifications within the constructed model. Lastly a `stopping_tolerance` can be specified which defines the tolerance for the stopping criteria. This value was set to $1e-3$ to try and force the model to generalize. The testing results for this implementation can be seen in Section 4.3.

4.3 Testing, Results and Model Refinement

Each kind of neural network discussed was generated 5 times to produce a range of values for training, validating and computational times. These results can be seen in Table 3 and graphically in Figure 6. As method involved building up 5 networks Table 3 shows the mean for each network over all 5 experiments. This was done in R using the `aggregate` function in R which acts almost like an `GROUP BY` from SQL to aggregate values based off a common property, in this case the model names. 'Vanilla' models shown are networks that don't contain any specific regularization technique but rather simply change the number of hidden layers and neurons per hidden layer as the testing variable. Models with dropout indicate the ratio based off input vs hidden layers dropout.

Model name	Training accuracy	Validation accuracy	Run time(s)
Vanilla, 50 by 3	0.97656	0.96954	5.918
Rand grid, 50 by 3	0.97640	0.96950	66.300
Vanilla, 500 by 3	0.97610	0.97236	76.972
Vanilla, 200 by 4	0.97598	0.97198	27.158
Early stop, 50 by 3	0.97572	0.97096	13.786
Vanilla, 500 by 4	0.97472	0.96974	110.188
Tanh drop:0%/0% 50 by 3	0.97218	0.96608	27.088
Tanh drop:10%/50%, 50 by 3	0.92120	0.93644	13.858
Tanh drop:20%/50%, 50 by 3	0.91386	0.92810	13.938

Table 3: Different neural network design training and validation accuracy



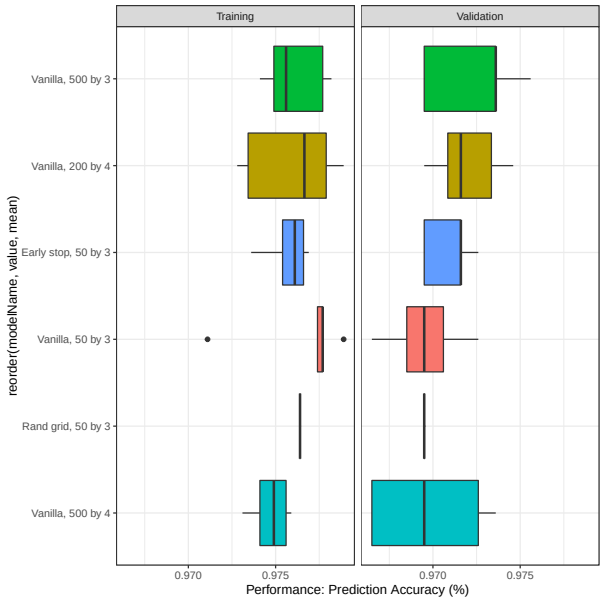
(a) Neural network hyperparameter accuracy comparison

(b) Further Refined SVM Tuning

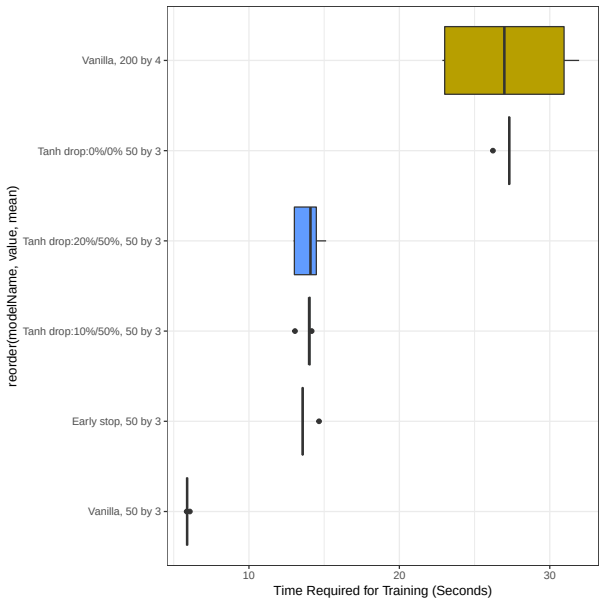
Figure 6: Neural network hyperparameter compute time comparison

There are clearly a few models in terms of both accuracy and training time that are far worse than

the others. From an accuracy perspective `tahn with dropout` had a low relative accuracy (and a high variance between models) and from a compute time perspective the `random grid` was way slower than other methods. The poor preforming models are now removed and the refined set without the poor preforming models can be seen in Figure 7.



(a) Neural network hyperparameter accuracy comparison refined



(b) Further Refined SVM Tuning

Figure 7: Network hyperparameter compute time comparison refined

There are a number of interesting observations that can be drawn from the results. Firstly all regularized models, irrespective of method, performed worse than non-regularized models. This was true for both the training and validation sets. This result is curious as one would expect that some level of regularization would have resulted with models that could generalize better and as a result have a higher validation accuracy.

Of the regularized models the best performing was the early stop and the random grid (which used L^1 and L^2 regularization). This can be seen in Figure 7a. Comparing these models to the vanilla models in fact shows very similar testing and training results in terms of mean values. However the regularized methods had far lower variance between multiple tests, as shown in the size of the box plot, when compared to the vanilla methods. This means that the regularization process preformed when building the model resulted in a more stable and consistent model output when compared to non-regularized methods. This can be seen when looking at the `Early Stop` and `Rand grid` models which performed relatively well when compared to the vanilla models but had distinctly lower variance between their training and testing results as shown by smaller box plot sizes indicating more stable models. This is a strong indication that this kind of model would generalize better to unseen testing data.

Each of the vanilla models have differing number of hidden layers and neurons per layer. Naively one might expect that models with the most neurons and with the most hidden layers would preform best. This was shown to not strictly hold, as can be seen in Figures 6 and 7. Interestingly the models that had more neurons and more hidden layers tended to be less stable between models as well as less stable when generalizing on the validation set. This phenomena can be seen with the `Vanilla, 500 by 4` model which shows the largest variation in validation accuracy when compared to its training accuracy indicating that having such a high number of neurons and hidden layers does not lend its self well to generalizing.

5 Method Comparison and Best Model Selection

SVM and NN models have now been extensively discussed and analyzed. The best setups for SVMs and NNs can now be compared and a final model selected. When looking at the accuracy of models on the training set SVMs preformed better across the board. However this could mean that SVMs were in general more over fit so to corroborate this result one needs to look at the validation error. The best SVM method was the Caret tuned SVM model using a radial kernel with a 97.46% validation accuracy. The best preforming Neural network was a non-tuned vanilla 3 layer neural network with 500 nodes per layer which achieved a validation accuracy of 97.23%. These accuracy's are very close to each other so picking one method as definitively better than the other is hard when we only consider prediction accuracy. The model building times are however not comparable, with the SVM taking $\approx 5\times$ longer to build than the NN. As the prediction accuracy's are very similar to each other potentially a hybrid model could yield the most ideal results.

5.1 Creation of an Ensemble Model

An ensemble model is a hybrid machine learning method that use multiple learning algorithms to create a better overall predictive model than could be created from any single model. This hybrid was created by taking the best three SVMs and the best four NN and combining them to create one larger model that takes a weighted vote from all 7 sub models. The mode of all votes is taken as the selected class. The mode is used as each model is given an equal weighting so simply the class with the most votes is taken as the output class. The implementation in `R` is shown in Figure 8.

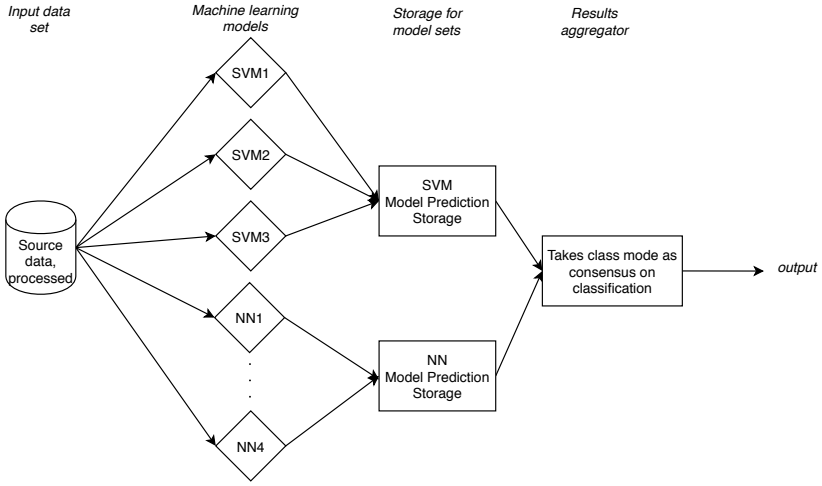


Figure 8: Ensamble network of machine learning methods

In `R` the ensemble model was implemented by first capturing the classifications from each of the 7 generated methods and storing them as CSV files. Then, these files were read in by a separate note book and the matrices of results joined to give one matrix of 7 rows(for each model) by n columns for n predictions. Next, the mode was calculated for each column in the matrix to get an aggregate vote for that particular classification. This was done using a custom `Mode` function which calculates the mode of a provided vector of numbers. This function was wrapped within an `apply` to apply it to all columns in the results aggregate part of Figure 8. This value is then taken as the voted sum. The output was then written to a CSV file.

This method yielded a validation accuracy of 97.56% which is slightly higher than all other methods tested thus far, showing a degree of stability achieved through running the ensemble method. As a result the ensemble model will be used for the unlabeled dataset and is selected as the best possible model combining both SVMs and NNs.

6 Generating Results for Unlabeled Dataset

A set of unlabeled data was provided for this assignment. The ensemble model generated before was used to classify this unlabeled dataset with the hope that taking the vote from a number of different models will yield a more stable result that generalizes well. This was done by first building a model on the full set of data (no training and validation split) and then predicting the classes on this unlabeled set. These predicted class labels were exported to a CSV file with no labels and no headings and submitted along with the report. The code implementation of this prediction can be seen in Appendix 3. The notebook books submitted along with this paper and those on Github assume you want to find the validation error. To recreate the unlabeled dataset classification one needs to change the name of the CSV files being read in and modify the split for training and validation at the top of each scripts.

7 Conclusion

This paper examined a number of different powerful machine learning methods, namely support vector machines and neural networks. For each a detailed theoretical foundation was laid that outlined the justification for model building, tuning and hyper parameters selection. For the support vector machines a collection of kernels were experimented and the best two were refined through a tuning process. For the neural networks a range of different regularization techniques were experimented with to find the ideal setup for the provided data. A hybrid approach was proposed, involving the joining of three support vector machines along with four neural networks to create an ensemble model that aimed to capture the best elements from multiple models without over fitting. This method worked well but if the compute time of building up all the models within the ensemble is considered it is a very inefficient process involving a lot of repeated iteration.

References

[1] D. Keshav, "Support Vector Machine (SVM) Theory — CommonLounge," 2015. [Online]. Available: https://www.commonlounge.com/discussion/a49bcd907bdc4824ae53483c060f0259

[2] C. Paul and G. K. Vishwakarma, "Back propagation neural networks and multiple regressions in the case of heteroskedasticity," *Communications in Statistics - Simulation and Computation*, vol. 46, no. 9, pp. 6772–6789, oct 2017. [Online]. Available: https://www.tandfonline.com/doi/full/10.1080/03610918.2016.1212066

[3] V. Cherkassky and Y. Ma, "Practical Selection of SVM Parameters and Noise Estimation for SVM Regression," Tech. Rep., 2010. [Online]. Available: http://people.ece.umn.edu/users/cherkass/N2002-SI-SVM-13-whole.pdf

[4] C.-W. Hsu, C.-C. Chang, and C.-J. Lin, "A Practical Guide to Support Vector Classification," Tech. Rep., 2016. [Online]. Available: http://www.csie.ntu.edu.tw/{~}cjlin

[5] H2o, "Deep Learning (Neural Networks) H2O 3.24.0.2 documentation." [Online]. Available: http://docs.h2o.ai/h2o/latest-stable/h2o-docs/data-science/deep-learning.html{#}introduction

[6] S. Mazilu and J. Iria, "L1 vs. L2 Regularization in Text Classification when Learning from Labeled Features," in *2011 10th International Conference on Machine Learning and Applications and Workshops*. IEEE, dec 2011, pp. 166–171. [Online]. Available: http://ieeexplore.ieee.org/document/6146963/

[7] T. Evgeniou, M. Pontil, and T. Poggio, "Regularization Networks and Support Vector Machines," Tech. Rep., 1999. [Online]. Available: http://pages.cs.wisc.edu/{~}brecht/cs838docs/99.evgeniou.reg.pdf

[8] L. Prechelt, "Early Stopping — but when?" Tech. Rep. [Online]. Available: https://page.mi.fu-berlin.de/prechelt/Biblio/stop{_}tricks1997.pdf

Assignment 2, question 1

This appendix stores all the code for implemented Support Vector Machine (SVMs) in R. Each section corresponds to a part of the discussion both directly and indirectly of that done in the main body of the report.

Setup and basic data processing

First all the required libraries are installed and data is imported.

```
#setup work space, install packages and import libs
suppressMessages(library(e1071))
suppressMessages(library(kernlab))
suppressMessages(library(caret))
suppressMessages(library(ggplot2))
suppressMessages(library(reshape2))
rm(list=ls())
fulldata <- read.csv("blocksTrain.csv")
fulldata <- fulldata[, -1]
```

Next, data processing to define the training and validation sets.

```
#convert class to a factor
fulldata$class <- as.factor(fulldata$class)
#normalize variables
fulldata[,1:10] <- scale(fulldata[,1:10])
#extract traning and validation stes
set.seed(42)
train <- sample(seq_len(nrow(fulldata)),
                size = ceiling(dim(fulldata)[1]*0.8)) #~80% of the set
Blocks.train <- fulldata[train, ]
Blocks.validation <- fulldata[-train, ]
```

Generic functions

Next generic functions are defined to store model building accuracy.

```
modelPreformance <- data.frame()
storeModelPreformance <- function(modelName, performance){
  modelPreformance <<- rbind(
    modelPreformance,
    data.frame(
      modelName,
      performance
    )
  )
}

ensamble <- data.frame()
storeEnsembleValue <- function(values){
  ensamble <<- rbind(
```

```
    ensamble,values)
}
```

The use of the confusionMatrix function from caret is the same as generating a table type confusion matrix. Syntactically this would look like this: `tab <- table(pred = svm.pred, true = Blocks.train$class)` to get the accuracy one would use this: `sum(diag(tab))/sum(tab)`. Miss classification rate would be 1-this number

```
simple.svm <- svm(class~.,
                  data = Blocks.train,
                  kernel = "radial",
                  gamma = 1,
                  cost = 1)
svm.pred <- predict(simple.svm, Blocks.validation)
confusionMatrix(svm.pred, Blocks.validation$class)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   1   2   3   4   5
##          1 886   7   3   5   7
##          2   7  51   0   0   0
##          3   0   0   0   0   0
##          4   1   0   0  12   0
##          5   2   0   0   0   4
##
## Overall Statistics
##
##                Accuracy : 0.9675
##                  95% CI : (0.9544, 0.9777)
##     No Information Rate : 0.9096
##     P-Value [Acc > NIR] : 5.242e-13
##
##                   Kappa : 0.794
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: 1 Class: 2 Class: 3 Class: 4 Class: 5
## Sensitivity            0.9888  0.87931 0.000000  0.70588 0.363636
## Specificity            0.7528  0.99245 1.000000  0.99897 0.997947
## Pos Pred Value         0.9758  0.87931      NaN  0.92308 0.666667
## Neg Pred Value         0.8701  0.99245 0.996954  0.99486 0.992850
## Prevalence             0.9096  0.05888 0.003046  0.01726 0.011168
## Detection Rate         0.8995  0.05178 0.000000  0.01218 0.004061
## Detection Prevalence   0.9218  0.05888 0.000000  0.01320 0.006091
## Balanced Accuracy      0.8708  0.93588 0.500000  0.85242 0.680791
```

Itterative hyper parameter search and plotting

Define a generic timer function to calculate how long training takes

```
start_timer <- function() return(proc.time())
stop_timer <- function(time_start) {
  time_diff <- proc.time() - time_start
  time_diff <- time_diff[1] + time_diff[2] + time_diff[3]
  return(round(as.numeric(time_diff), digits = 2))
}
```

Next, a number of diffrent kernals are experimented with and stored in a data structure plot later.

```
tt <- start_timer()
## Train the model
model <- svm(class~.,
                data = Blocks.train,
                kernel = "radial",
                gamma = 1,
                cost = 1)

## Evaluate performance
yhat_train <- predict(model, Blocks.train)
yhat_validation <- predict(model, Blocks.validation)


results <- data.frame(Training = round(confusionMatrix(yhat_train,
                                                       Blocks.train$class)$overall[1], 4),
                      Validation = round(confusionMatrix(yhat_validation,
                                                         Blocks.validation$class)$overall[1], 4),
                      Duration = round(stop_timer(tt), 2))
storeModelPreformance("SVM Radial", results)
```

```
tt <- start_timer()
## Train the model
model <- svm(class~.,
                 data = Blocks.train,
                 kernel = "polynomial",
                 gamma = 1,
                 cost = 1)

## Evaluate performance
yhat_train <- predict(model, Blocks.train)
yhat_validation <- predict(model, Blocks.validation)


results <- data.frame(Training = round(confusionMatrix(yhat_train,
                                                       Blocks.train$class)$overall[1], 4),
                      Validation = round(confusionMatrix(yhat_validation,
                                                         Blocks.validation$class)$overall[1], 4),
                      Duration = round(stop_timer(tt), 2))
storeModelPreformance("SVM polynomial", results)
storeEnsembleValue(yhat_validation)
```

```
tt <- start_timer()
## Train the model
model <- svm(class~.,
                 data = Blocks.train,
                 kernel = "sigmoid",
```

```
                  gamma = 1,
                  cost = 1)

## Evaluate performance
yhat_train <- predict(model, Blocks.train)
yhat_validation <- predict(model, Blocks.validation)


results <- data.frame(Training = round(confusionMatrix(yhat_train,
                                                       Blocks.train$class)$overall[1], 4),
                      Validation = round(confusionMatrix(yhat_validation,
                                                         Blocks.validation$class)$overall[1], 4),
                      Duration = round(stop_timer(tt), 2))
storeModelPreformance("SVM sigmoid", results)
```

```
tt <- start_timer()
## Train the model
model <- ksvm(class~.,
                  data = Blocks.train,
                  type = "C-svc",
                  kernel = 'anovadot',
                  C = 1, scaled = c())
```

```
##  Setting default kernel parameters
```

```
## Evaluate performance
yhat_train <- predict(model, Blocks.train)
yhat_validation <- predict(model, Blocks.validation)


results <- data.frame(Training = round(confusionMatrix(yhat_train,
                                                       Blocks.train$class)$overall[1], 4),
                      Validation = round(confusionMatrix(yhat_validation,
                                                         Blocks.validation$class)$overall[1], 4),
                      Duration = round(stop_timer(tt), 2))
storeModelPreformance("KSVM anovadot", results)
```

```
tt <- start_timer()
## Train the model
model <- ksvm(class~.,
                  data = Blocks.train,
                  type = "C-svc",
                  kernel = 'splinedot',
                  C = 1, scaled = c())
```

```
##  Setting default kernel parameters
```

```
## Evaluate performance
yhat_train <- predict(model, Blocks.train)
yhat_validation <- predict(model, Blocks.validation)


results <- data.frame(Training = round(confusionMatrix(yhat_train,
                                                       Blocks.train$class)$overall[1], 4),
                      Validation = round(confusionMatrix(yhat_validation,
                                                         Blocks.validation$class)$overall[1], 4),
```

```
                      Duration = round(stop_timer(tt), 2))
storeModelPreformance("KSVM splinedot", results)
```

```
tt <- start_timer()
## Train the model
model <- ksvm(class~.,
                 data = Blocks.train,
                 type = "C-svc",
                 kernel = 'tanhdot',
                 C = 1,
                 scaled = c())
```

```
##  Setting default kernel parameters
```

```
## Evaluate performance
yhat_train <- predict(model, Blocks.train)
yhat_validation <- predict(model, Blocks.validation)

results <- data.frame(Training = round(confusionMatrix(yhat_train,
                                                       Blocks.train$class)$overall[1], 4),
                      Validation = round(confusionMatrix(yhat_validation,
                                                         Blocks.validation$class)$overall[1], 4),
                      Duration = round(stop_timer(tt), 2))
storeModelPreformance("KSVM tanhdot", results)
```

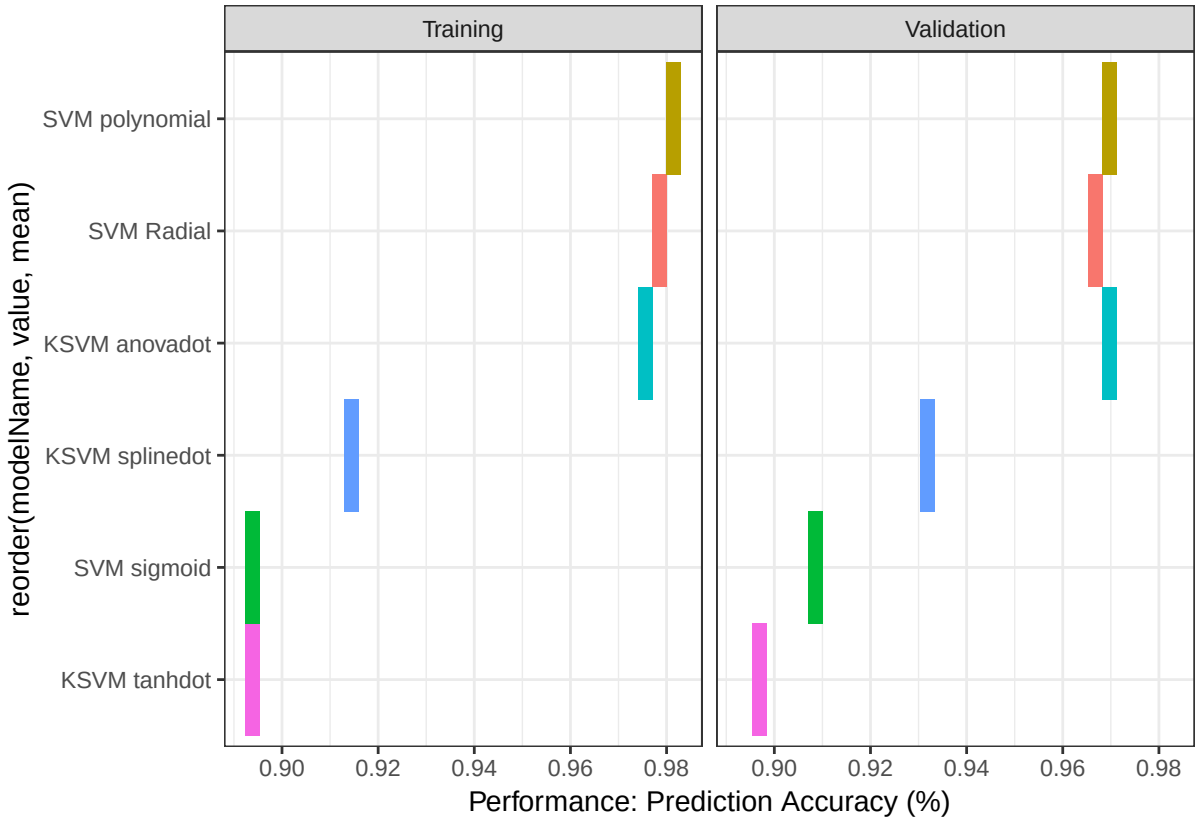
Print average results in a table. Order by training

```
modelPreformance[order(modelPreformance$Training),]
```

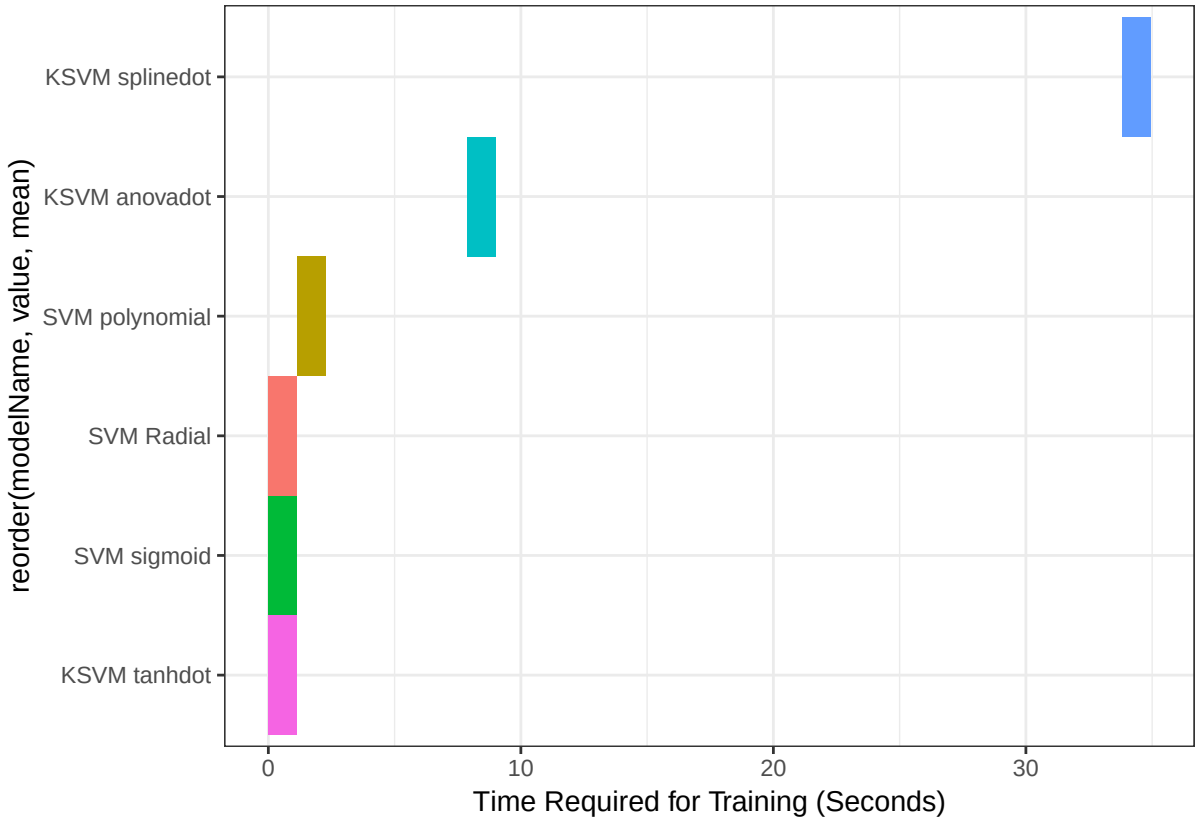
```
##                modelName Training Validation Duration
## Accuracy5   KSVM tanhdot   0.8937     0.8964     0.80
## Accuracy2    SVM sigmoid   0.8944     0.9076     0.81
## Accuracy4 KSVM splinedot   0.9142     0.9310    34.61
## Accuracy3  KSVM anovadot   0.9749     0.9685     8.22
## Accuracy       SVM Radial   0.9782     0.9675     0.85
## Accuracy1 SVM polynomial   0.9812     0.9695     1.45
```

Lastly create some graphs for our results!

```
res_acc <- melt(modelPreformance,
                id.vars = c('modelName'),
                measure.vars = c('Training', 'Validation'))

res_time <- melt(modelPreformance,
                 id.vars = c('modelName'),
                 measure.vars = c('Duration'))

## Prediction Accuracy
ggplot(data = res_acc,
       aes(x = reorder(modelName, value, mean), y = value, fill = modelName)) +
  geom_bin2d() +
  facet_grid(~ variable) +
  ylab("Performance: Prediction Accuracy (%)")+
  coord_flip() +
  guides(fill = FALSE, color = FALSE, linetype = FALSE, shape = FALSE)+
  theme_bw()
```




```
## Time Required for Training
ggplot(data = res_time,
       aes(x = reorder(modelName,value, mean), y = value, fill = modelName)) +
  geom_bin2d() +
  ylab("Time Required for Training (Seconds)")+
    coord_flip() +
  guides(fill = FALSE, color = FALSE, linetype = FALSE, shape = FALSE)+
  theme_bw()
```



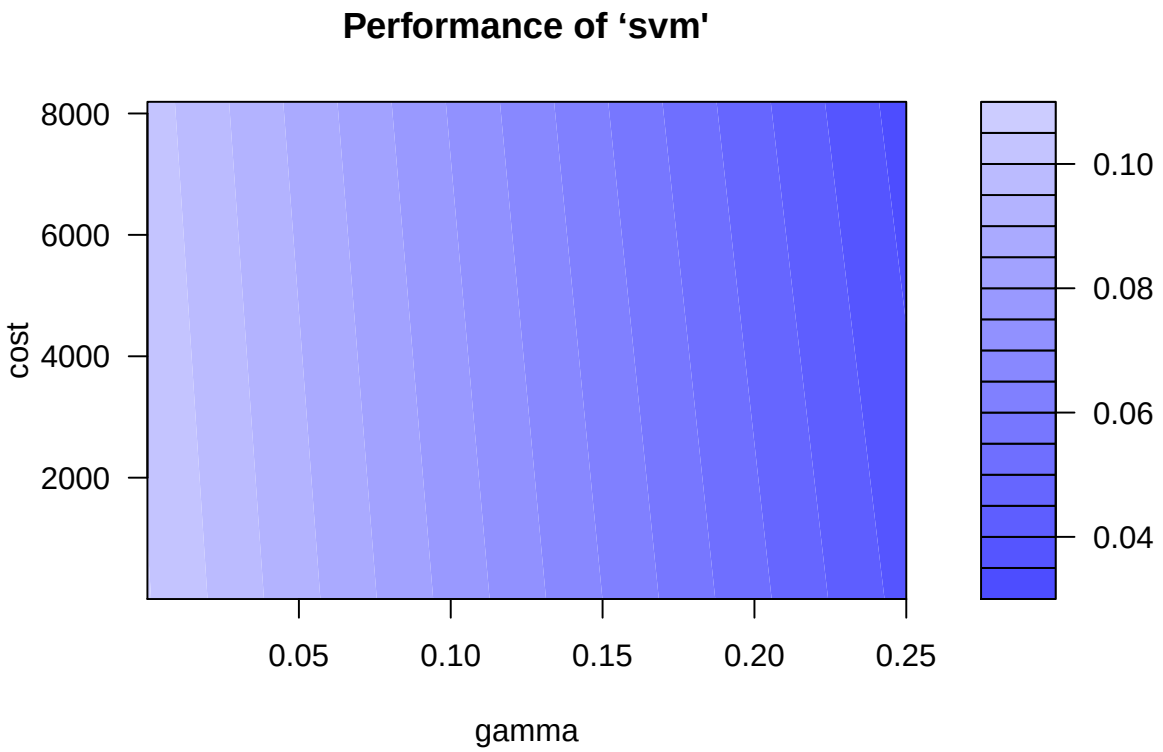
Tuning or Hyper parameter optomization.

If the cost is too high then it will mean a high penalty for non-seperable points and as a result the model may store too many support vectors which will lead to over fitting. however, if the cost value is too small then the model may end up with too few support vectors and the model then will not be accurate.

The γ (gamma) has to be tuned to better fit the hyperplane to the data. It is responsible for the linearity degree of the hyperplane, and for that, it is not present when using linear kernels. The smaller γ is, the more the hyperplane is going to look like a straight line. If γ is too great, the hyperplane will be more curvy and might delineate the data too well and lead to overfitting.

We also plot the tune output. Darker regions represent lower misclasification error.

```
tt <- start_timer()
tune.out <- tune(svm, class~.,
                 data = Blocks.train,
                 kernel = "polynomial",
                 ranges = list(gamma = c(2^-13,2^-2), cost = c(2^-0,2^13)))

plot(tune.out)
```




```
# show best model
best.model <- tune.out$best.model
yhat_train <- predict(best.model, Blocks.train)
yhat_validation <- predict(best.model, Blocks.validation)
storeEnsembleValue(yhat_validation) # store the votes from this model to form part of the ensamble sele

results <- data.frame(Training = round(confusionMatrix(yhat_train,
                                                       Blocks.train$class)$overall[1], 4),
                      Validation = round(confusionMatrix(yhat_validation,
                                                         Blocks.validation$class)$overall[1], 4),
                      Duration = round(stop_timer(tt), 2))
storeModelPreformance("Tuned SVM polynomial", results)
```

We can also try using the caret train function to expand over a grid of posible values.

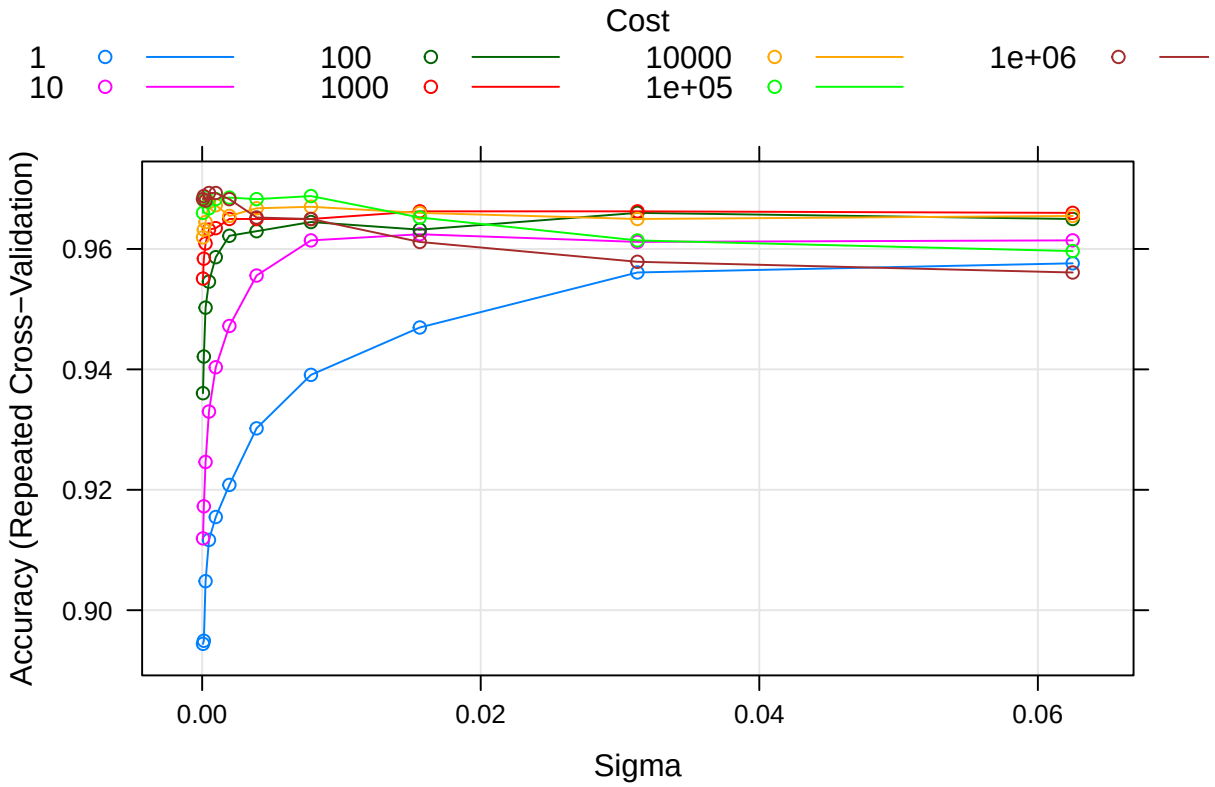
```
library(caret)
library(caretEnsemble)
```

```
##
## Attaching package: 'caretEnsemble'
```

```
## The following object is masked from 'package:ggplot2':
##
##     autoplot
```

```
tt <- start_timer()
TrainCtrl1 <- trainControl(method = "repeatedcv",
                           number = 5,
                           repeats = 1,
                           verbose = FALSE)
SVMgrid <- expand.grid(sigma = 2^(-14:-4),
```

```
                        C = 10^(0:6))

modelSvmRRB <- train(class~.,
                     data = Blocks.train,
                     method="svmRadial",
                     trControl=TrainCtrl1,
                     tuneGrid = SVMgrid,
                     verbose=FALSE)
plot(modelSvmRRB)
```




```
best.model <- modelSvmRRB$finalModel
yhat_train <- predict(modelSvmRRB, Blocks.train)
yhat_validation <- predict(modelSvmRRB, Blocks.validation)
storeEnsembleValue(yhat_validation) # store the votes from this model to form part of the ensamble sele

results <- data.frame(Training = round(confusionMatrix(yhat_train,
                                                       Blocks.train$class)$overall[1], 4),
                      Validation = round(confusionMatrix(yhat_validation,
                                                         Blocks.validation$class)$overall[1], 4),
                      Duration = round(stop_timer(tt), 2))
storeModelPreformance("Caret SVM, tuned radial", results)
```

Print average results in a table. Order by training

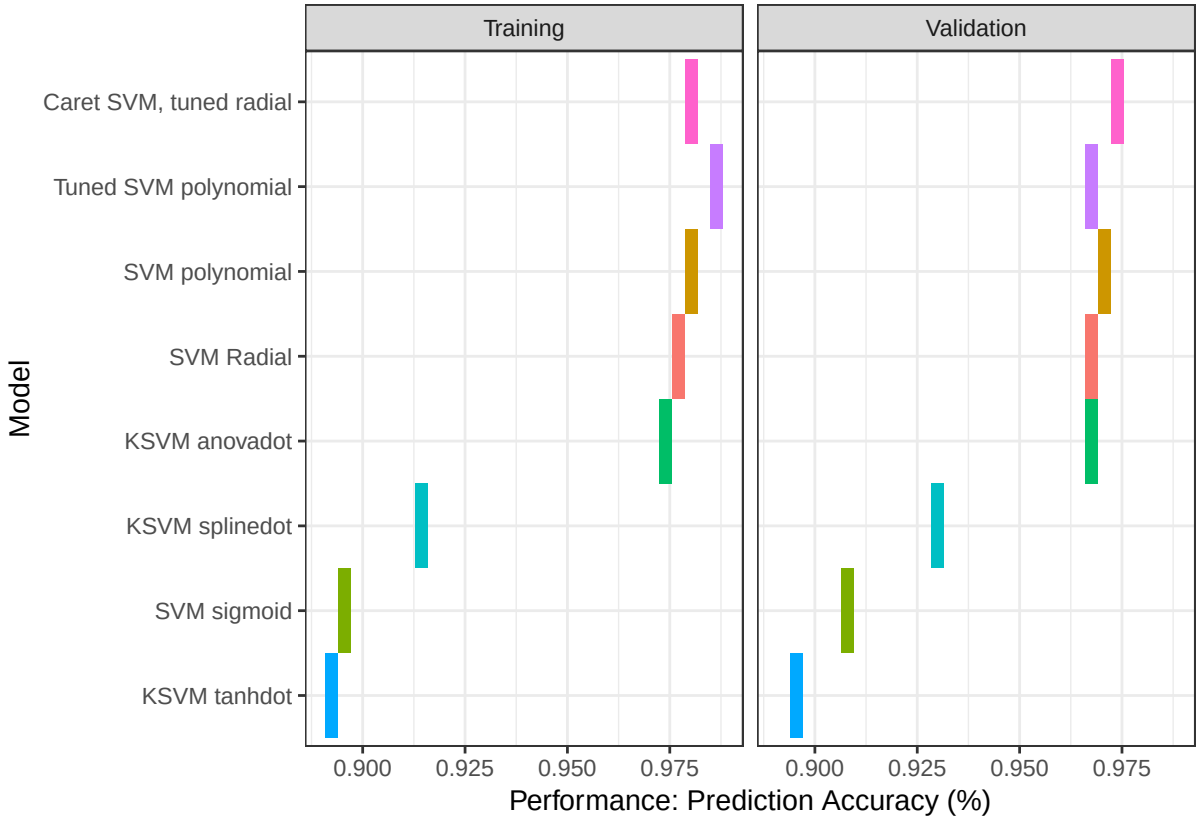
```
modelPreformance[order(modelPreformance$Training, decreasing = TRUE),]
```

```
##                          modelName Training Validation Duration
## Accuracy6    Tuned SVM polynomial   0.9878     0.9675   150.95
## Accuracy7 Caret SVM, tuned radial   0.9817     0.9746   401.86
```

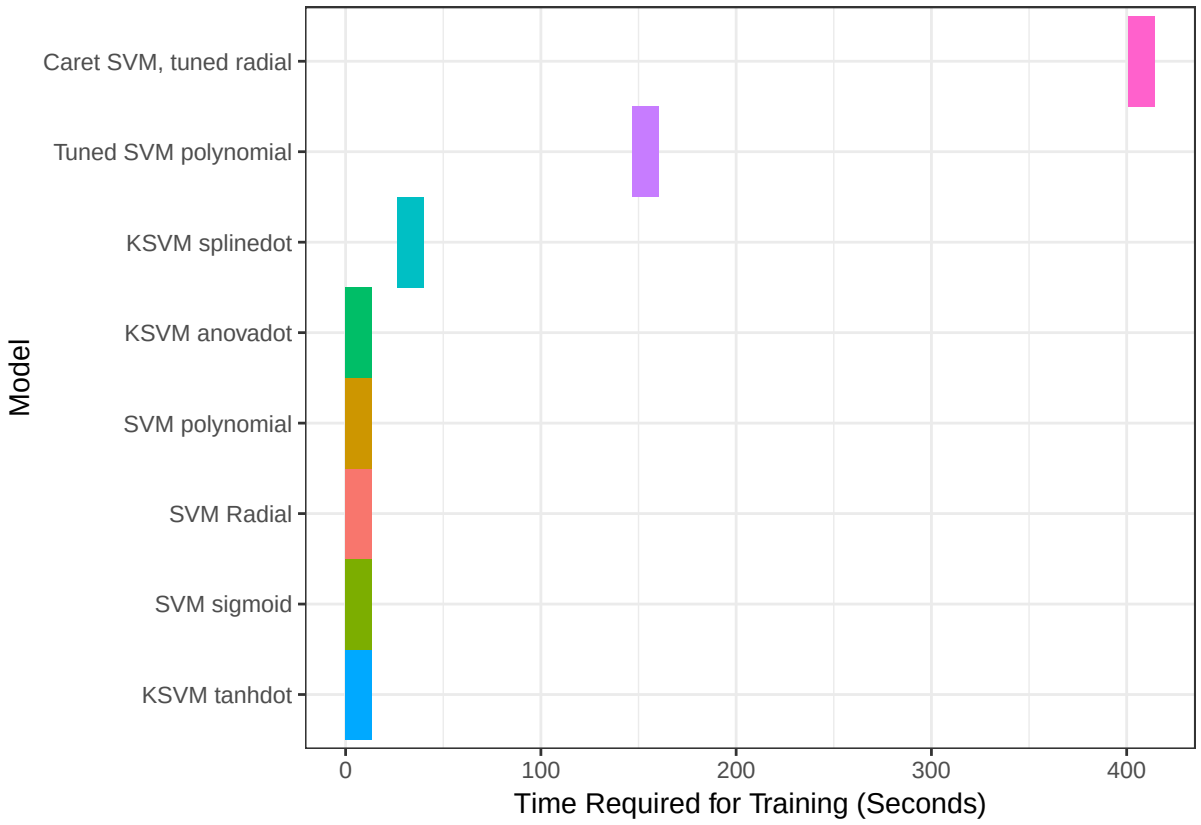
```
## Accuracy1          SVM polynomial   0.9812     0.9695     1.45
## Accuracy               SVM Radial   0.9782     0.9675     0.85
## Accuracy3           KSVM anovadot   0.9749     0.9685     8.22
## Accuracy4          KSVM splinedot   0.9142     0.9310    34.61
## Accuracy2             SVM sigmoid   0.8944     0.9076     0.81
## Accuracy5            KSVM tanhdot   0.8937     0.8964     0.80
```

Lastly create some graphs for our results!

```
res_acc <- melt(modelPreformance,
                id.vars = c('modelName'),
                measure.vars = c('Training', 'Validation'))

res_time <- melt(modelPreformance,
                 id.vars = c('modelName'),
                 measure.vars = c('Duration'))

## Prediction Accuracy
ggplot(data = res_acc,
       aes(x = reorder(modelName, value, mean), y = value, fill = modelName)) +
  geom_bin2d() +
  facet_grid(~ variable) +
  ylab("Performance: Prediction Accuracy (%)")+
  xlab("Model")+
  coord_flip() +
  guides(fill = FALSE, color = FALSE, linetype = FALSE, shape = FALSE)+
  theme_bw()
```



```
## Time Required for Training
ggplot(data = res_time,
       aes(x = reorder(modelName,value, mean), y = value, fill = modelName)) +
  geom_bin2d() +
  ylab("Time Required for Training (Seconds)")+
  xlab("Model")+
    coord_flip() +
  guides(fill = FALSE, color = FALSE, linetype = FALSE, shape = FALSE)+
  theme_bw()
```




```
write.csv(ensamble, "EnsambleVotesQuestion1.csv")
```

Question 2

Question 2: Neural Networks

This appendix starts with importing the required libraries then preforming data processing. Next, a basic example using the h2o library is shown, outlining a simple use case. After this a number of diffrent model refinement techniques are applied and stored in a datastructure to be printed and plotted later.

Libraries

```
#setup work space, install packages and import libs
suppressMessages(library(h2o))
suppressMessages(library(caret))
suppressMessages(library(mlbench))
suppressMessages(library(ggplot2))
suppressMessages(library(reshape2))
suppressMessages(library(DEEPR))
rm(list=ls())
fulldata <- read.csv("blocksTrain.csv")
fulldata <- fulldata[, -1]

suppressMessages(h2o.init(max_mem_size = "15g"))
```

```
## Warning in h2o.clusterInfo():
## Your H2O cluster version is too old (4 months and 4 days)!
## Please download and install the latest version from http://h2o.ai/download/
```

```
knitr::opts_chunk$set(message = FALSE)
```

Data Processing

Read data, process it and create a h2o data object to be used later on.

```
#convert class to a factor
fulldata$class <- as.factor(fulldata$class)
#normalize variables
fulldata[,1:10] <- scale(fulldata[,1:10])
#extract traning and validation stes
set.seed(42)
train <- sample(seq_len(nrow(fulldata)),
                size = ceiling(dim(fulldata)[1]*0.8)) #~80% of the set
Blocks.train <- fulldata[train, ]
Blocks.validation <- fulldata[-train, ]

datTrain_h2o <- as.h2o(Blocks.train)
datValidation_h2o = as.h2o(Blocks.validation)

ensamble <- data.frame()
storeEnsembleValue <- function(values){
```

```
  ensamble <<- rbind(
    ensamble,values)
}
```

Generic Functions

A generic function is defined to store the preformance of each model. This is the same as I did it in the first project. It acts to store the name of a model along with some preformance charireristics that can be viewed later on.

```
modelPreformance <- data.frame()
storeModelPreformance <- function(modelName, performance){
  modelPreformance <<- rbind(
    modelPreformance,
    data.frame(
      modelName,
      performance
    )
  )
}

ensamble <- data.frame()
storeEnsembleValue <- function(values){
  ensamble <<- rbind(
    ensamble,values)
}
```

Simple Neural network

First a simple nerual network is defined to show the simple use example of h2o.

```
model <- h2o.deeplearning(x = 1:10,
                          y = 11,
                          training_frame = datTrain_h2o,
                          hidden = c(50,50,50),
                          seed = 1,)

yhat_train <- h2o.predict(model, datTrain_h2o)$predict
yhat_train <- as.factor(as.matrix(yhat_train))
yhat_validation <- h2o.predict(model, datValidation_h2o)$predict
yhat_validation <- as.factor(as.matrix(yhat_validation))

sprintf("Training accuracy: %s",
        round(confusionMatrix(yhat_train, Blocks.train$class)$overall[1],4))
sprintf("Validation accuracy: %s",
        round(confusionMatrix(yhat_validation, Blocks.validation$class)$overall[1],4))
```

Itterative hyper parameter search and plotting

Define a generic timer function to calculate how long training takes

```
start_timer <- function() return(proc.time())
stop_timer <- function(time_start) {
  time_diff <- proc.time() - time_start
  time_diff <- time_diff[1] + time_diff[2] + time_diff[3]
  return(round(as.numeric(time_diff), digits = 2))
}
```

Core Settings for the Experiments

```
n_run <- 5
n_epochs <- 50
```

Vanilla Model, 50 by 3

```
res_tmp <- data.frame(Trial = 1:n_run, Training = NA, Validation = NA, Duration = NA)
for (n in 1:n_run) {
  tt <- start_timer()
  ## Train the model
  model <- h2o.deeplearning(x = 1:10,
                            y = 11,
                            training_frame = datTrain_h2o,
                            hidden = c(50,50,50),
                            seed = 1,
                            epochs = n_epochs)

  ## Evaluate performance
  yhat_train <- h2o.predict(model, datTrain_h2o)$predict
  yhat_train <- as.factor(as.matrix(yhat_train))
  yhat_validation <- h2o.predict(model, datValidation_h2o)$predict
  yhat_validation <- as.factor(as.matrix(yhat_validation))

  ## Store Results
  res_tmp[n, 1] <- n
  res_tmp[n, 2] <- round(caret::confusionMatrix(yhat_train,
                                                Blocks.train$class)$overall[1], 4)
  res_tmp[n, 3] <- round(caret::confusionMatrix(yhat_validation,
                                                Blocks.validation$class)$overall[1], 4)
  res_tmp[n, 4] <- round(stop_timer(tt), 2)
}

## Store overall results
storeModelPreformance("Vanilla, 50 by 3", res_tmp)
```

Vanilla Model, 200 by 3

```
res_tmp <- data.frame(Trial = 1:n_run, Training = NA, Validation = NA, Duration = NA)
for (n in 1:n_run) {
  tt <- start_timer()

  ## Train the model
  model <- h2o.deeplearning(x = 1:10,
                            y = 11,
                            training_frame = as.h2o(Blocks.train),
                            hidden = c(200,200,200),
                            seed = 1,
                            epochs = n_epochs)

  ## Evaluate performance
  yhat_train <- h2o.predict(model, datTrain_h2o)$predict
  yhat_train <- as.factor(as.matrix(yhat_train))
  yhat_validation <- h2o.predict(model, datValidation_h2o)$predict
  yhat_validation <- as.factor(as.matrix(yhat_validation))

  ## Store Results
  res_tmp[n, 1] <- n
  res_tmp[n, 2] <- round(caret::confusionMatrix(yhat_train,
                                                Blocks.train$class)$overall[1], 4)
  res_tmp[n, 3] <- round(caret::confusionMatrix(yhat_validation,
                                                Blocks.validation$class)$overall[1], 4)
  res_tmp[n, 4] <- round(stop_timer(tt), 2)
}

## Store overall results
storeModelPreformance("Vanilla, 200 by 4", res_tmp)
storeEnsembleValue(yhat_validation)
```

Vanilla Model, 500 by 3

```
res_tmp <- data.frame(Trial = 1:n_run, Training = NA, Validation = NA, Duration = NA)
for (n in 1:n_run) {
  tt <- start_timer()

  ## Train the model
  model <- h2o.deeplearning(x = 1:10,
                            y = 11,
                            training_frame = as.h2o(Blocks.train),
                            hidden = c(500,500,500),
                            seed = 1,
                            epochs = n_epochs)

  ## Evaluate performance
  yhat_train <- h2o.predict(model, datTrain_h2o)$predict
  yhat_train <- as.factor(as.matrix(yhat_train))
  yhat_validation <- h2o.predict(model, datValidation_h2o)$predict
  yhat_validation <- as.factor(as.matrix(yhat_validation))
```

```
  ## Store Results
  res_tmp[n, 1] <- n
  res_tmp[n, 2] <- round(caret::confusionMatrix(yhat_train,
                                                Blocks.train$class)$overall[1], 4)
  res_tmp[n, 3] <- round(caret::confusionMatrix(yhat_validation,
                                                Blocks.validation$class)$overall[1], 4)
  res_tmp[n, 4] <- round(stop_timer(tt), 2)
}

## Store overall results
storeModelPreformance("Vanilla, 500 by 3", res_tmp)
storeEnsembleValue(yhat_validation)
```

Vanilla Model, 200 by 4

```
res_tmp <- data.frame(Trial = 1:n_run, Training = NA, Validation = NA, Duration = NA)
for (n in 1:n_run) {
  tt <- start_timer()

  ## Train the model
  model <- h2o.deeplearning(x = 1:10,
                            y = 11,
                            training_frame = as.h2o(Blocks.train),
                            hidden = c(200,200,200,200),
                            seed = 1,
                            epochs = n_epochs)

  ## Evaluate performance
  yhat_train <- h2o.predict(model, datTrain_h2o)$predict
  yhat_train <- as.factor(as.matrix(yhat_train))
  yhat_validation <- h2o.predict(model, datValidation_h2o)$predict
  yhat_validation <- as.factor(as.matrix(yhat_validation))

  ## Store Results
  res_tmp[n, 1] <- n
  res_tmp[n, 2] <- round(caret::confusionMatrix(yhat_train,
                                                Blocks.train$class)$overall[1], 4)
  res_tmp[n, 3] <- round(caret::confusionMatrix(yhat_validation,
                                                Blocks.validation$class)$overall[1], 4)
  res_tmp[n, 4] <- round(stop_timer(tt), 2)
}

## Store overall results
storeModelPreformance("Vanilla, 200 by 4", res_tmp)
```

Vanilla Model, 500 by 4

```
res_tmp <- data.frame(Trial = 1:n_run, Training = NA, Validation = NA, Duration = NA)
for (n in 1:n_run) {
  tt <- start_timer()
```

```
  ## Train the model
  model <- h2o.deeplearning(x = 1:10,
                            y = 11,
                            training_frame = as.h2o(Blocks.train),
                            hidden = c(500,500,500,500),
                            seed = 1,
                            epochs = n_epochs)

  ## Evaluate performance
  yhat_train <- h2o.predict(model, datTrain_h2o)$predict
  yhat_train <- as.factor(as.matrix(yhat_train))
  yhat_validation <- h2o.predict(model, datValidation_h2o)$predict
  yhat_validation <- as.factor(as.matrix(yhat_validation))

  ## Store Results
  res_tmp[n, 1] <- n
  res_tmp[n, 2] <- round(caret::confusionMatrix(yhat_train,
                                                Blocks.train$class)$overall[1], 4)
  res_tmp[n, 3] <- round(caret::confusionMatrix(yhat_validation,
                                                Blocks.validation$class)$overall[1], 4)
  res_tmp[n, 4] <- round(stop_timer(tt), 2)
}

## Store overall results
storeModelPreformance("Vanilla, 500 by 4", res_tmp)
```

\subsubsection{Tanh drop:0%/0% 50 by 3}

```
## Create an empty data frame for results
res_tmp <- data.frame(Trial = 1:n_run, Training = NA, Validation = NA, Duration = NA)

## Train model and evaluate performance for n times

for (n in 1:n_run) {
  tt <- start_timer()

  ## Train the model
  model <- h2o.deeplearning(x = 1:10,  # column numbers for predictors
                            y = 11,   # column number for label
                            training_frame = datTrain_h2o,
                            activation = "Tanh",
                            balance_classes = TRUE,
                            hidden = c(50,50,50),  ## three hidden layers
                            #nfolds = 10,
                            epochs = n_epochs)

  ## Evaluate performance
  yhat_train <- h2o.predict(model, datTrain_h2o)$predict
  yhat_train <- as.factor(as.matrix(yhat_train))
  yhat_validation <- h2o.predict(model, datValidation_h2o)$predict
  yhat_validation <- as.factor(as.matrix(yhat_validation))

  ## Store Results
  res_tmp[n, 1] <- n
```

```
  res_tmp[n, 2] <- round(caret::confusionMatrix(yhat_train,
                                                Blocks.train$class)$overall[1], 4)
  res_tmp[n, 3] <- round(caret::confusionMatrix(yhat_validation,
                                                Blocks.validation$class)$overall[1], 4)
  res_tmp[n, 4] <- round(stop_timer(tt), 2)
}

## Store overall results
storeModelPreformance("Tanh drop:0%/0% 50 by 3", res_tmp)
```

\subsubsection{Tanh drop:0%/50%, 50 by 3}

```
## Create an empty data frame for results
res_tmp <- data.frame(Trial = 1:n_run, Training = NA, Validation = NA, Duration = NA)

for (n in 1:n_run) {
  tt <- start_timer()

  model <- h2o.deeplearning(x = 1:10,  # column numbers for predictors
                            y = 11,   # column number for label
                            training_frame = datTrain_h2o,
                            activation = "TanhWithDropout",
                            input_dropout_ratio = 0,
                            hidden_dropout_ratios = c(0.5,0.5,0.5),
                            balance_classes = TRUE,
                            hidden = c(50,50,50),  ## three hidden layers
                            epochs = n_epochs)

  yhat_train <- h2o.predict(model, datTrain_h2o)$predict
  yhat_train <- as.factor(as.matrix(yhat_train))
  yhat_validation <- h2o.predict(model, datValidation_h2o)$predict
  yhat_validation <- as.factor(as.matrix(yhat_validation))

  res_tmp[n, 1] <- n
  res_tmp[n, 2] <- round(confusionMatrix(yhat_train,
                                         Blocks.train$class)$overall[1], 4)
  res_tmp[n, 3] <- round(confusionMatrix(yhat_validation,
                                         Blocks.validation$class)$overall[1], 4)
  res_tmp[n, 4] <- round(stop_timer(tt), 2)
}
```

```
## Warning in levels(reference) != levels(data): longer object length is not a
## multiple of shorter object length
```

```
## Warning in confusionMatrix.default(yhat_train, Blocks.train$class): Levels
## are not in the same order for reference and data. Refactoring data to
## match.
```

```
## Warning in levels(reference) != levels(data): longer object length is not a
## multiple of shorter object length
```

```
## Warning in confusionMatrix.default(yhat_validation,
## Blocks.validation$class): Levels are not in the same order for reference
## and data. Refactoring data to match.
```

```
## Warning in levels(reference) != levels(data): longer object length is not a
## multiple of shorter object length
```

```
## Warning in confusionMatrix.default(yhat_validation,
## Blocks.validation$class): Levels are not in the same order for reference
## and data. Refactoring data to match.

## Warning in levels(reference) != levels(data): longer object length is not a
## multiple of shorter object length

## Warning in confusionMatrix.default(yhat_validation,
## Blocks.validation$class): Levels are not in the same order for reference
## and data. Refactoring data to match.

## Warning in levels(reference) != levels(data): longer object length is not a
## multiple of shorter object length

## Warning in confusionMatrix.default(yhat_train, Blocks.train$class): Levels
## are not in the same order for reference and data. Refactoring data to
## match.

## Warning in levels(reference) != levels(data): longer object length is not a
## multiple of shorter object length

## Warning in confusionMatrix.default(yhat_validation,
## Blocks.validation$class): Levels are not in the same order for reference
## and data. Refactoring data to match.
```

```
res_b <- data.frame(Model = "Tanh drop:0%/50%, 50 by 3", res_tmp[, -1])
```

\subsubsection{Tanh drop:10%/50%, 50 by 3}

```
res_tmp <- data.frame(Trial = 1:n_run, Training = NA, Validation = NA, Duration = NA)

for (n in 1:n_run) {
  tt <- start_timer()

  ## Train the model
  model <- h2o.deeplearning(x = 1:10,  # column numbers for predictors
                            y = 11,   # column number for label
                            training_frame = datTrain_h2o,
                            activation = "TanhWithDropout",
                            input_dropout_ratio = 0.1,
                            hidden_dropout_ratios = c(0.5,0.5,0.5),
                            balance_classes = TRUE,
                            hidden = c(50,50,50),  ## three hidden layers
                            epochs = n_epochs)

  yhat_train <- h2o.predict(model, datTrain_h2o)$predict
  yhat_train <- as.factor(as.matrix(yhat_train))
  yhat_validation <- h2o.predict(model, datValidation_h2o)$predict
  yhat_validation <- as.factor(as.matrix(yhat_validation))

  res_tmp[n, 1] <- n
  res_tmp[n, 2] <- round(confusionMatrix(yhat_train,
                                         Blocks.train$class)$overall[1], 4)
  res_tmp[n, 3] <- round(confusionMatrix(yhat_validation,
                                         Blocks.validation$class)$overall[1], 4)
  res_tmp[n, 4] <- round(stop_timer(tt), 2)
}
```

```
## Warning in levels(reference) != levels(data): longer object length is not a
## multiple of shorter object length

## Warning in confusionMatrix.default(yhat_train, Blocks.train$class): Levels
## are not in the same order for reference and data. Refactoring data to
## match.

## Warning in levels(reference) != levels(data): longer object length is not a
## multiple of shorter object length

## Warning in confusionMatrix.default(yhat_validation,
## Blocks.validation$class): Levels are not in the same order for reference
## and data. Refactoring data to match.

## Warning in levels(reference) != levels(data): longer object length is not a
## multiple of shorter object length

## Warning in confusionMatrix.default(yhat_train, Blocks.train$class): Levels
## are not in the same order for reference and data. Refactoring data to
## match.

## Warning in levels(reference) != levels(data): longer object length is not a
## multiple of shorter object length

## Warning in confusionMatrix.default(yhat_validation,
## Blocks.validation$class): Levels are not in the same order for reference
## and data. Refactoring data to match.

## Warning in levels(reference) != levels(data): longer object length is not a
## multiple of shorter object length

## Warning in confusionMatrix.default(yhat_train, Blocks.train$class): Levels
## are not in the same order for reference and data. Refactoring data to
## match.

## Warning in levels(reference) != levels(data): longer object length is not a
## multiple of shorter object length

## Warning in confusionMatrix.default(yhat_validation,
## Blocks.validation$class): Levels are not in the same order for reference
## and data. Refactoring data to match.

## Warning in levels(reference) != levels(data): longer object length is not a
## multiple of shorter object length

## Warning in confusionMatrix.default(yhat_train, Blocks.train$class): Levels
## are not in the same order for reference and data. Refactoring data to
## match.

## Warning in levels(reference) != levels(data): longer object length is not a
## multiple of shorter object length

## Warning in confusionMatrix.default(yhat_validation,
## Blocks.validation$class): Levels are not in the same order for reference
## and data. Refactoring data to match.

## Warning in levels(reference) != levels(data): longer object length is not a
## multiple of shorter object length

## Warning in confusionMatrix.default(yhat_train, Blocks.train$class): Levels
## are not in the same order for reference and data. Refactoring data to
## match.
```

```
## Warning in levels(reference) != levels(data): longer object length is not a
## multiple of shorter object length
```

```
## Warning in confusionMatrix.default(yhat_validation,
## Blocks.validation$class): Levels are not in the same order for reference
## and data. Refactoring data to match.
```

```
storeModelPreformance("Tanh drop:10%/50%, 50 by 3", res_tmp)
```

\subsubsection{Tanh drop:20%/50%, 50 by 3}

```
res_tmp <- data.frame(Trial = 1:n_run, Training = NA, Validation = NA, Duration = NA)
for (n in 1:n_run) {
  tt <- start_timer()

  model <- h2o.deeplearning(x = 1:10,  # column numbers for predictors
                            y = 11,   # column number for label
                            training_frame  = datTrain_h2o,
                            activation = "TanhWithDropout",
                            input_dropout_ratio = 0.2,
                            hidden_dropout_ratios = c(0.5,0.5,0.5),
                            balance_classes = TRUE,
                            hidden = c(50,50,50),  ## three hidden layers
                            epochs = n_epochs)


  yhat_train <- h2o.predict(model, datTrain_h2o)$predict
  yhat_train <- as.factor(as.matrix(yhat_train))
  yhat_validation <- h2o.predict(model, datValidation_h2o)$predict
  yhat_validation <- as.factor(as.matrix(yhat_validation))


  res_tmp[n, 1] <- n
  res_tmp[n, 2] <- round(confusionMatrix(yhat_train,
                                         Blocks.train$class)$overall[1], 4)
  res_tmp[n, 3] <- round(confusionMatrix(yhat_validation,
                                         Blocks.validation$class)$overall[1], 4)
  res_tmp[n, 4] <- round(stop_timer(tt), 2)

}
```

```
## Warning in levels(reference) != levels(data): longer object length is not a
## multiple of shorter object length
```

```
## Warning in confusionMatrix.default(yhat_train, Blocks.train$class): Levels
## are not in the same order for reference and data. Refactoring data to
## match.
```

```
## Warning in levels(reference) != levels(data): longer object length is not a
## multiple of shorter object length
```

```
## Warning in confusionMatrix.default(yhat_validation,
## Blocks.validation$class): Levels are not in the same order for reference
## and data. Refactoring data to match.
```

```
## Warning in levels(reference) != levels(data): longer object length is not a
## multiple of shorter object length
```

```
## Warning in confusionMatrix.default(yhat_train, Blocks.train$class): Levels
## are not in the same order for reference and data. Refactoring data to
## match.

## Warning in levels(reference) != levels(data): longer object length is not a
## multiple of shorter object length

## Warning in confusionMatrix.default(yhat_validation,
## Blocks.validation$class): Levels are not in the same order for reference
## and data. Refactoring data to match.

## Warning in levels(reference) != levels(data): longer object length is not a
## multiple of shorter object length

## Warning in confusionMatrix.default(yhat_train, Blocks.train$class): Levels
## are not in the same order for reference and data. Refactoring data to
## match.

## Warning in levels(reference) != levels(data): longer object length is not a
## multiple of shorter object length

## Warning in confusionMatrix.default(yhat_validation,
## Blocks.validation$class): Levels are not in the same order for reference
## and data. Refactoring data to match.

## Warning in levels(reference) != levels(data): longer object length is not a
## multiple of shorter object length

## Warning in confusionMatrix.default(yhat_train, Blocks.train$class): Levels
## are not in the same order for reference and data. Refactoring data to
## match.

## Warning in levels(reference) != levels(data): longer object length is not a
## multiple of shorter object length

## Warning in confusionMatrix.default(yhat_validation,
## Blocks.validation$class): Levels are not in the same order for reference
## and data. Refactoring data to match.

## Warning in levels(reference) != levels(data): longer object length is not a
## multiple of shorter object length

## Warning in confusionMatrix.default(yhat_train, Blocks.train$class): Levels
## are not in the same order for reference and data. Refactoring data to
## match.

## Warning in levels(reference) != levels(data): longer object length is not a
## multiple of shorter object length

## Warning in confusionMatrix.default(yhat_validation,
## Blocks.validation$class): Levels are not in the same order for reference
## and data. Refactoring data to match.
```

```
storeModelPreformance("Tanh drop:20%/50%, 50 by 3", res_tmp)
```

Early stop, 50 by 3

```
res_tmp <- data.frame(Trial = 1:n_run, Training = NA, Validation = NA, Duration = NA)

for (n in 1:n_run) {
```

```
  tt <- start_timer()

  ## Train the model
  model <- h2o.deeplearning(x = 1:10,
                            y = 11,
                            training_frame = as.h2o(Blocks.train),
                            hidden = c(50,50,50),
                            seed = 1,
                            epochs = n_epochs,
                            nfolds = 3,                            #used for early stopping
                            score_interval = 1,                    #used for early stopping
                            stopping_rounds = 5,                   #used for early stopping
                            stopping_metric = "misclassification", #used for early stopping
                            stopping_tolerance = 1e-3,             #used for early stopping
                            )

  yhat_train <- h2o.predict(model, datTrain_h2o)$predict
  yhat_train <- as.factor(as.matrix(yhat_train))
  yhat_validation <- h2o.predict(model, datValidation_h2o)$predict
  yhat_validation <- as.factor(as.matrix(yhat_validation))

  res_tmp[n, 1] <- n
  res_tmp[n, 2] <- round(confusionMatrix(yhat_train,
                                         Blocks.train$class)$overall[1], 4)
  res_tmp[n, 3] <- round(confusionMatrix(yhat_validation,
                                         Blocks.validation$class)$overall[1], 4)
  res_tmp[n, 4] <- round(stop_timer(tt), 2)
}

storeModelPreformance("Early stop, 50 by 3", res_tmp)
storeEnsembleValue(yhat_validation)
```

Rand grid, 50 by 3

```
res_tmp <- data.frame(Trial = 1:n_run, Training = NA, Validation = NA, Duration = NA)

  for (n in 1:n_run) {
    tt <- start_timer()

   activation_opt <- c("Rectifier", "Maxout", "Tanh")
  l1_opt <- c(0, 0.00001, 0.0001, 0.001, 0.01)
  l2_opt <- c(0, 0.00001, 0.0001, 0.001, 0.01)

  hyper_params <- list(activation = activation_opt, l1 = l1_opt, l2 = l2_opt)
  search_criteria <- list(strategy = "RandomDiscrete", max_runtime_secs = 60)

  splits <- h2o.splitFrame(as.h2o(Blocks.train), ratios = 0.8, seed = 1)

  dl_grid <- h2o.grid("deeplearning",
                      x = 1:10,
                      y = 11,
                      grid_id = "dl_grid",
```

```
                      training_frame = splits[[1]],
                      validation_frame = splits[[2]],
                      seed = 1,
                      epochs = n_epochs,
                      hidden = c(50,50,50),
                      hyper_params = hyper_params,
                      search_criteria = search_criteria)

  dl_gridperf <- h2o.getGrid(grid_id = "dl_grid",
                             sort_by = "accuracy",
                             decreasing = TRUE)
  print(dl_gridperf)

  best_dl_model_id <- dl_gridperf@model_ids[[1]]
  model <- h2o.getModel(best_dl_model_id)

  yhat_train <- h2o.predict(model, datTrain_h2o)$predict
  yhat_train <- as.factor(as.matrix(yhat_train))
  yhat_validation <- h2o.predict(model, datValidation_h2o)$predict
  yhat_validation <- as.factor(as.matrix(yhat_validation))

  res_tmp[n, 1] <- n
  res_tmp[n, 2] <- round(confusionMatrix(yhat_train,
                                         Blocks.train$class)$overall[1], 4)
  res_tmp[n, 3] <- round(confusionMatrix(yhat_validation,
                                         Blocks.validation$class)$overall[1], 4)
  res_tmp[n, 4] <- round(stop_timer(tt), 2)
}

storeModelPreformance("Rand grid, 50 by 3", res_tmp)
storeEnsembleValue(yhat_validation)
```

Results output and plotting

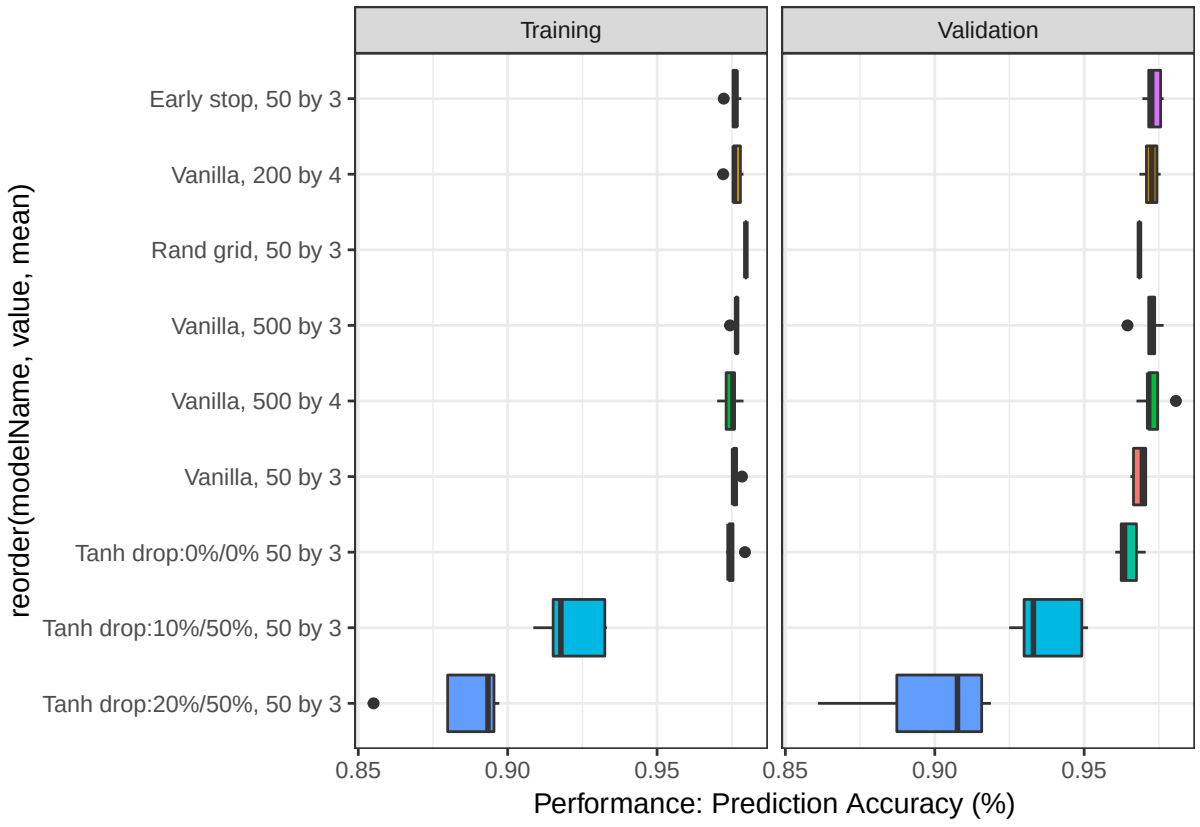
Print average results in a table. this acts like an `AVG` then `ORDERY` in `SQL`.

```
aggrigated <- aggregate(modelPreformance[,3:5], list(modelPreformance$modelName), mean)
aggrigated[order(aggrigated$Training, decreasing = TRUE),]
```

```
##                       Group.1 Training Validation Duration
## 9          Rand grid, 50 by 3  0.97970    0.96850   75.884
## 3           Vanilla, 500 by 3  0.97634    0.97178   75.768
## 1            Vanilla, 50 by 3  0.97632    0.96854    6.946
## 2           Vanilla, 200 by 4  0.97628    0.97238   27.245
## 8         Early stop, 50 by 3  0.97578    0.97318   13.586
## 5    Tanh drop:0%/0% 50 by 3  0.97522    0.96488   28.390
## 4           Vanilla, 500 by 4  0.97468    0.97320  107.610
## 6 Tanh drop:10%/50%, 50 by 3  0.92146    0.93766   13.880
## 7 Tanh drop:20%/50%, 50 by 3  0.88420    0.89806   13.906
```

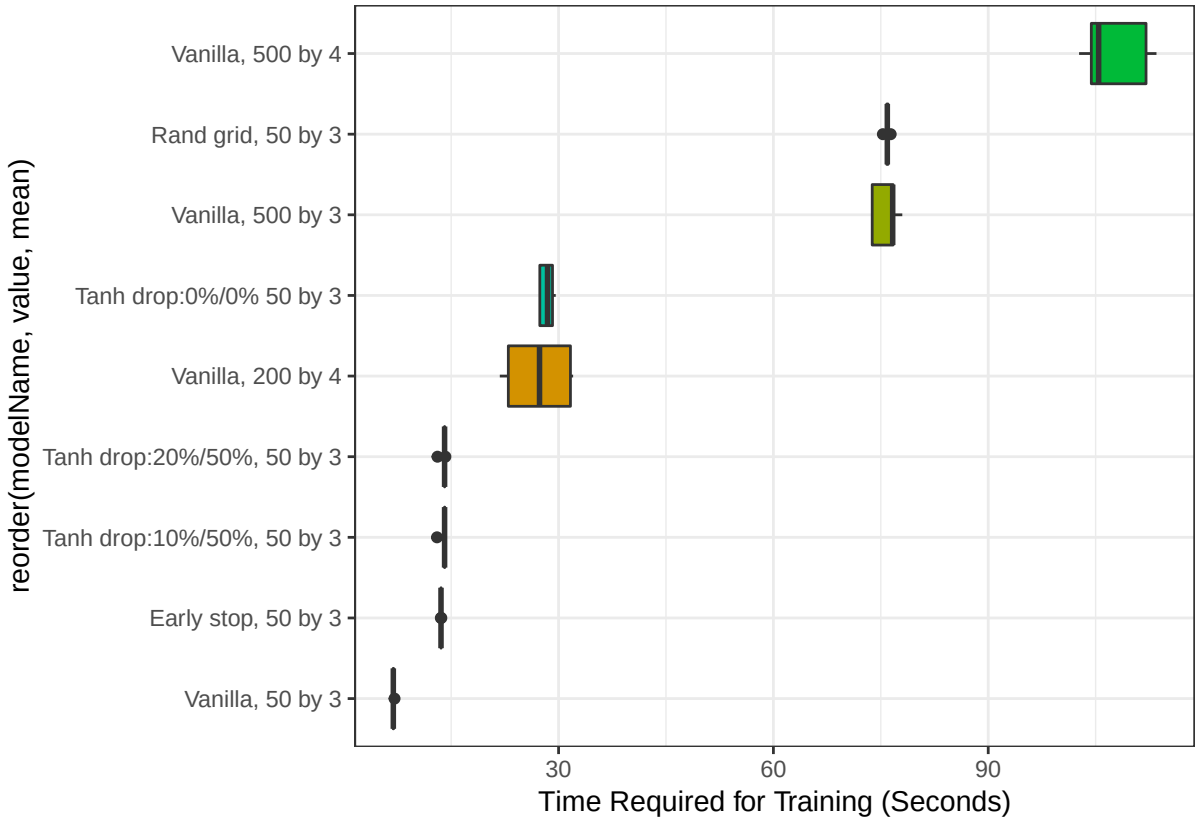
Model vizualization

```
res_acc <- melt(modelPreformance,
                id.vars = c('modelName'),
                measure.vars = c('Training', 'Validation'))

res_time <- melt(modelPreformance,
                 id.vars = c('modelName'),
                 measure.vars = c('Duration'))

## Prediction Accuracy
ggplot(data = res_acc,
       aes(x = reorder(modelName, value, mean), y = value, fill = modelName)) +
  geom_boxplot() +
  facet_grid(~ variable) +
  ylab("Performance: Prediction Accuracy (%)")+
  coord_flip() +
  guides(fill = FALSE, color = FALSE, linetype = FALSE, shape = FALSE)+
  theme_bw()
```




```
## Time Required for Training
ggplot(data = res_time,
       aes(x = reorder(modelName,value, mean), y = value, fill = modelName)) +
  geom_boxplot() +
  ylab("Time Required for Training (Seconds)")+
    coord_flip() +
```

```
  guides(fill = FALSE, color = FALSE, linetype = FALSE, shape = FALSE)+
  theme_bw()
```

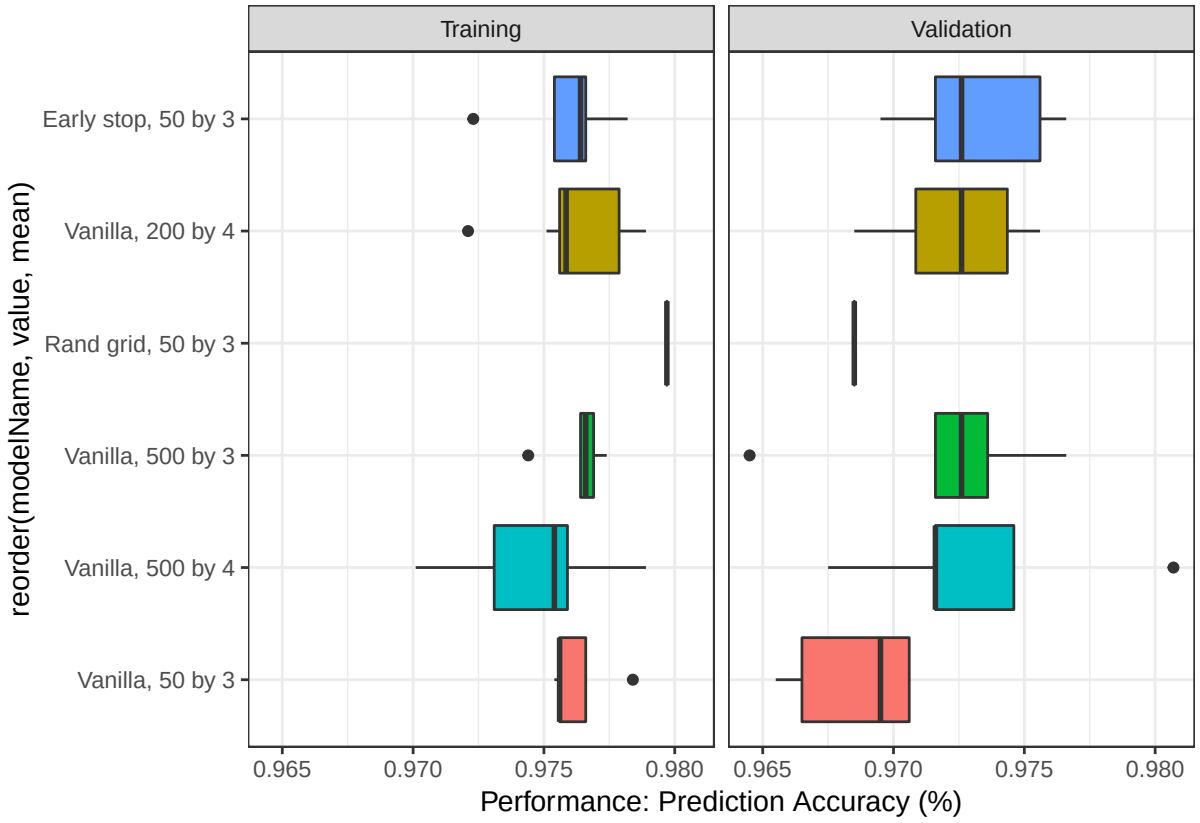


Refined Vizualization

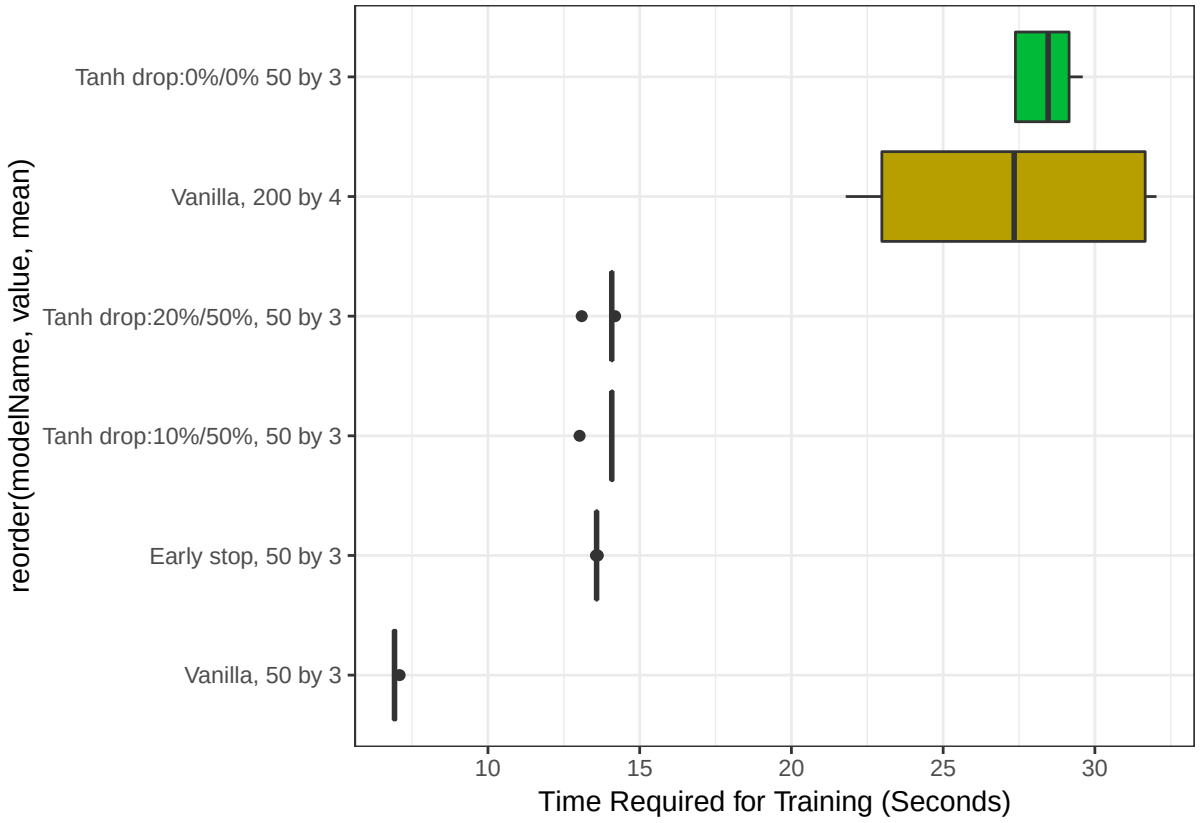
Remove all the rows that have the word dropout in them as these were bad preformers on the preformance and and remove all the rows that have random grid or the c(500,500,500,500) layered as these were bad preformeres in the training time.

```
res_acc <- melt(modelPreformance[!grepl('drop', modelPreformance$modelName),],
                id.vars = c('modelName'),
                measure.vars = c('Training', 'Validation'))


res_time <- melt(modelPreformance[!(grepl('Rand grid', modelPreformance$modelName)
                                    + grepl('500', modelPreformance$modelName)),],
                 id.vars = c('modelName'),
                 measure.vars = c('Duration'))

## Prediction Accuracy
ggplot(data = res_acc,
       aes(x = reorder(modelName, value, mean), y = value, fill = modelName)) +
  geom_boxplot() +
  facet_grid(~ variable) +
  ylab("Performance: Prediction Accuracy (%)")+
```

```
  coord_flip() +
  guides(fill = FALSE, color = FALSE, linetype = FALSE, shape = FALSE)+
  theme_bw()
```




```
## Time Required for Training
ggplot(data = res_time,
       aes(x = reorder(modelName,value, mean), y = value, fill = modelName)) +
  geom_boxplot() +
  ylab("Time Required for Training (Seconds)")+
    coord_flip() +
  guides(fill = FALSE, color = FALSE, linetype = FALSE, shape = FALSE)+
  theme_bw()
```




```
write.csv(ensamble, "EnsambleVotesQuestion2.csv")
```

Ensemble Joiner

This notebook takes in the "votes" from the two sets of methods, stored as csv files and joines them together taking the mean values.

```
suppressMessages(library(caret))
Mode <- function(x) {
  ux <- unique(x)
  ux[which.max(tabulate(match(x, ux)))]
}
```

```
svm_predictions <- read.csv("EnsambleVotesQuestion1.csv")
svm_predictions <- svm_predictions[, -1]
svm_predictions <- as.matrix(unname(svm_predictions))

nn_predictions <- read.csv("EnsambleVotesQuestion2.csv")
nn_predictions <- nn_predictions[, -1]
nn_predictions <- as.matrix(unname(nn_predictions))

combindMatrix <- rbind(svm_predictions, nn_predictions)

modevalues <- apply(combindMatrix, 2, Mode)
```

Next we need to get the prediction results so we can get our confusion matrix. Here we will read in the data the same way I did for each question. Note that if the input to the ensemblerBuilder is an unlabled set then you can skip this step and rather directly run the saving of the csv file with the generated lables. This is done as follows: `write.csv(modevalues, file = "NoLables_PredictedResults.csv")`

```
fulldata <- read.csv("blocksTrain.csv")
fulldata <- fulldata[, -1]

#convert class to a factor
fulldata$class <- as.factor(fulldata$class)
#normalize variables
#extract traning and validation stes
set.seed(42)
train <- sample(seq_len(nrow(fulldata)),
                size = ceiling(dim(fulldata)[1]*0.8)) #~80% of the set
Blocks.train <- fulldata[train, ]
Blocks.validation <- fulldata[-train, ]

confusionMatrix(as.factor(modevalues), Blocks.validation$class)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   1   2   3   4   5
##          1 880   3   0   4   4
##          2  10  54   0   0   0
##          3   0   0   3   0   1
##          4   1   0   0  13   0
##          5   5   1   0   0   6
##
## Overall Statistics
```

```
##
##                Accuracy : 0.9706
##                  95% CI : (0.958, 0.9802)
##     No Information Rate : 0.9096
##     P-Value [Acc > NIR] : 1.744e-14
##
##                   Kappa : 0.8298
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: 1 Class: 2 Class: 3 Class: 4 Class: 5
## Sensitivity            0.9821  0.93103 1.000000  0.76471 0.545455
## Specificity            0.8764  0.98921 0.998982  0.99897 0.993840
## Pos Pred Value         0.9877  0.84375 0.750000  0.92857 0.500000
## Neg Pred Value         0.8298  0.99566 1.000000  0.99588 0.994861
## Prevalence             0.9096  0.05888 0.003046  0.01726 0.011168
## Detection Rate         0.8934  0.05482 0.003046  0.01320 0.006091
## Detection Prevalence   0.9046  0.06497 0.004061  0.01421 0.012183
## Balanced Accuracy      0.9293  0.96012 0.999491  0.88184 0.769647
```