Stats 315B: Homework 3

Joe Higgins, Austin Wang, Jessica Wetstone

Due 5/20/2018

Question 1

Consider a multi-hidden layer neural network trained by sequential steepest-descent using the weight updating formula $w_t = w_{t-1} - \eta G(w_{t-1})$ Here t labels the observations presented in sequence (time) and $G(w)$ is the gradient of the squared-error criterion evaluated at w. Derive a recursive "back-propagation" algorithm for updating all of the network weights at each step. With this algorithm the update for an input weight to a particular hidden node is computed using only the value of its corresponding input (that it weights), the value of the output of the hidden node to which it is input, and an "error signal" from each of the nodes in the next higher layer to which this node is connected. Thus, each node in the network can update its input weights using information provided only by the nodes to which it is connected.

Per Piazza, we assume that the network has only one output node, and that the activation function for the output node is simply the identity $I(\zeta) = \zeta$. We also assume that the activation functions for all of the interior nodes are sigmoids: $S(\zeta) = \frac{1}{1+e^{-\zeta}}$

The loss of the network for a particular observation x_i can be written $L(y_i, \hat{F}(x_i)) = \frac{1}{2}(y_i - \hat{F}(x_i))^2$.

We use the following notation:

- $O_i^{(j)}$ is the output of the i^{th} node of layer j.
- $I_j^{(k)}$ is the number of inputs to the j^{th} node of layer k
- $W_{ij}^{(k)}$ is the weight connecting the i^{th} node in layer $k-1$ to the j^{th} node of layer k
- $n^{(l)}$ is the number of nodes in layer l.

The output of the final node o is:

$$\hat{F}(x_i) = \sum_{k=1}^{I_o^{(L)}} W_{ko}^{(L)} O_k^{(L-1)}$$

The output of each internal node is (where S is a sigmoid function):

$$O_j^{(l)} = S(\sum_{k=1}^{I_j^{(l)}} W_{kj}^{(l)} O_k^{(l-1)})$$

We want to derive a recursive back-propagation algorithm for updating all of the network weights at each time step. At each time step, we calculate not only the gradient with respect to the weights of node j in layer l but also with respect to the outputs of the nodes in layer $l-1$ that are connected to node j.

First, we need to derive the gradient for the base case (the output node). The gradient of the loss with

respect to the weight of the output node is:

$$\frac{\partial L(y_i, \hat{F}(x_i))}{\partial W_{ko}^{(L)}} = \frac{\partial L(y_i, \hat{F}(x_i))}{\partial \hat{F}(x_i)} * \frac{\partial \hat{F}(x_i)}{\partial W_{ko}^{(L)}}$$

$$\frac{\partial L(y_i, \hat{F}(x_i))}{\hat{F}(x_i)} = \frac{\partial}{\partial \hat{F}(x_i)} \big[\frac{1}{2}(y_i - \hat{F}(x_i))^2\big] = (y_i - \hat{F}(x_i))$$

$$\frac{\partial \hat{F}(x_i)}{\partial W_{ko}^{(L)}} = \frac{\partial}{\partial W_{ko}^{(L)}} \sum_{j=1}^{I_o^{(L)}} W_{jo}^{(L)} O_j^{(L-1)} = O_k^{(L-1)}$$

$$\frac{\partial L(y_i, \hat{F}(x_i))}{\partial W_{ko}^{(L)}} = (y_i - \hat{F}(x_i)) * O_k^{(L-1)}$$

At the same time, we also calculate the gradient with respect to the output of each node in the $(L-1)^{th}$ layer connected to the output node:

$$\frac{\partial L(y_i, \hat{F}(x_i))}{\partial O_k^{(L-1)}} = \frac{\partial L(y_i, \hat{F}(x_i))}{\partial \hat{F}(x_i)} * \frac{\partial \hat{F}(x_i)}{\partial O_k^{(L-1)}}$$
$$= (y_i - \hat{F}(x_i)) * W_{ko}^{(L)}$$

For the weights of each node in the l^{th} layer, assuming we have already calculated the gradients of the output of the l^{th} layer, we calculate the gradient:

$$\frac{\partial L(y_i, \hat{F}(x_i))}{\partial W_{kj}^{(l)}} = \frac{\partial L(y_i, \hat{F}(x_i))}{\partial O_j^{(l)}} * \frac{\partial O_j^{(l)}}{\partial \zeta} * \frac{\partial \zeta}{\partial W_{kj}^{(l)}}$$

$\frac{\partial L(y_i, \hat{F}(x_i))}{\partial O_j^{(l)}}$ was calculated in the previous step

$$\frac{\partial O_j^{(l)}}{\partial \zeta} = \frac{\partial S(\zeta)}{\partial \zeta} = S(\zeta) * (1 - S(\zeta)), \text{ where } \zeta = \sum_{r=1}^{I_j^{(l)}} W_{rj}^{(l)} O_r^{(l-1)}$$

$$\frac{\partial \zeta}{\partial W_{kj}^{(l)}} = \frac{\partial}{\partial W_{kj}^{(l)}} \sum_{r=1}^{I_j^{(l)}} W_{rj}^{(l)} O_r^{(l-1)} = O_k^{(l-1)}$$

$$\frac{\partial L(y_i, \hat{F}(x_i))}{\partial W_{ij}^{(l)}} = \frac{\partial L(y_i, \hat{F}(x_i))}{\partial O_k^{(l)}} * S(\sum_{r=1}^{I_j^{(l)}} W_{rj}^{(l)} O_r^{(l-1)}) * (1 - S(\sum_{r=1}^{I_j^{(l)}} W_{rj}^{(l)} O_r^{(l-1)})) * O_k^{(l-1)}$$

And finally for the outputs of the $(l-1)^{th}$ layer:

$$\frac{\partial L(y_i, \hat{F}(x_i))}{\partial O_k^{(l-1)}} = \frac{\partial L(y_i, \hat{F}(x_i))}{\partial O_j^{(l)}} * \frac{\partial O_j^{(l)}}{\partial \zeta} * \frac{\partial \zeta}{\partial O_k^{(l-1)}}$$

$$\frac{\partial L(y_i, \hat{F}(x_i))}{\partial O_k^{(l-1)}} = \frac{\partial L(y_i, \hat{F}(x_i))}{\partial O_j^{(l)}} * S(\sum_{r=1}^{I_j^{(l)}} W_{rj}^{(l)} O_r^{(l-1)}) * (1 - S(\sum_{r=1}^{I_j^{(l)}} W_{rj}^{(l)} O_r^{(l-1)})) * W_{kj}^{(l)}$$

Using the derivations above, we can express the backpropagation algorithm in pseudocode as follows:

procedure BACKPROP
  for layer $l = 1$ ... L **do**
    for node $j = 1$ ... $n^{(l)}$ **do**
      for input node $i = 1$ ... $I_j^{(L)}$ **do**
        $W_{ij}^{(l)} \leftarrow$ initial guess
        gradient_$W_{ij}^{(l)} \leftarrow 0$
        gradient_$O_j^{(l)} \leftarrow 0$
        prev_$W_{ij}^{(l)} \leftarrow 0$
      end for
    end for
  end for
  loop
    for observation $i = 1$ ... N **do**
      for layer $l = L$ ... 1 **do**
        for node $j = 1$ ... $n^{(l)}$ **do**
          for input node $k = 1$ ... $I_j^{(L)}$ **do**
            prev_$W_{kj}^{(l)} \leftarrow W_{kj}^{(l)}$
            if $l == L$ **then**
              gradient_$W_{kj}^{(l)} \leftarrow (y_i - \hat{F}(x_i)) * O_k^{(L-1)}$
              gradient_$O_k^{(l-1)} \leftarrow (y_i - \hat{F}(x_i)) * W_{ko}^{(L)}$
            else
              gradient_$W_{kj}^{(l)} \leftarrow$ gradient_$O_j^{(l)} * S(\sum_{r=1}^{I_j^{(l)}} W_{rj}^{(l)} O_r^{(l-1)}) * (1 - S(\sum_{r=1}^{I_j^{(l)}} W_{rj}^{(l)} O_r^{(l-1)})) * O_k^{(l-1)}$
              gradient_$O_k^{(l-1)} \leftarrow$ gradient_$O_j^{(l)} * S(\sum_{r=1}^{I_j^{(l)}} W_{rj}^{(l)} O_r^{(l-1)}) * (1 - S(\sum_{r=1}^{I_j^{(l)}} W_{rj}^{(l)} O_r^{(l-1)})) * W_{kj}^{(l)}$
            end if
            $W_{kj}^{(l)} \leftarrow W_{kj}^{(l)} - \eta$ * gradient_$W_{kj}^{(l)}$
          end for
        end for
      end for
    end for
  end loop
  until $|\mathbf{W} - \mathbf{prev\_W}| <$ threshold
end procedure

Question 2

Consider a radial basis function network with spherical Gaussian basis of the form $B(x|\mu_m, \sigma_m) = \left(-\frac{1}{2\sigma_m^2}\sum_{j=1}^{n}(x_j - \mu_{jm})^2\right)$, with the function approximation given by $\hat{F}(x) = \sum_{m=1}^{M} a_m B(x|\mu_m, \sigma_m)$ and sum-of-squares error criterion. Derive expressions for the gradient $G(x)$ with respect to all (types of) parameters in the network.

Question 3

Consider a ("elliptical") radial basis function network with elliptically symmetric Gaussian basis $B(\mathbf{x}|\mu_m, \Sigma) = exp(-\frac{1}{2}(\mathbf{x} - \mu_m)^T\Sigma(\mathbf{x} - \mu_m))$ where Σ is a positive definite matrix. Show that the output of such a network is equivalent to that of one composed of spherically symmetric Gaussian basis functions (Problem 2) with $\sigma_m = 1$, provided the input vector is first transformed by an appropriate linear transformation. Find expressions relating the transformed input vector $\bar{\mathbf{x}}$ and the transformed basis function centers $\bar{\mu}_m$ to the corresponding original vectors $\mathbf{x}$ and μ_m.

We will show that we can simply rescale the parameters $\mathbf{x}$ and μ_m by $\Sigma^{-\frac{1}{2}}$. We know that $\Sigma^{-\frac{1}{2}}$ exists because Σ is a symmetric, positive definite matrix so it has exactly one positive definite square root, and that square root is invertible.

Let $\bar{\mathbf{x}} = \Sigma^{-\frac{1}{2}}\mathbf{x}$ and $\bar{\mu}_m = \Sigma^{-\frac{1}{2}}\mu_m$. Then plugging those values in to the elliptically symmetric Gaussian basis function:

$$
\begin{aligned}
B(\bar{\mathbf{x}}|\bar{\mu}_m, \Sigma) &= e^{-\frac{1}{2}(\bar{\mathbf{x}}-\bar{\mu}_m)^T\Sigma(\bar{\mathbf{x}}-\bar{\mu}_m)} \\
&= e^{-\frac{1}{2}(\Sigma^{-\frac{1}{2}}(\mathbf{x}-\mu_m))^T\Sigma(\Sigma^{-\frac{1}{2}}(\mathbf{x}-\mu_m)} \\
&= e^{-\frac{1}{2}((\mathbf{x}-\mu_m)^T{\Sigma^{-\frac{1}{2}}}^T\Sigma\Sigma^{-\frac{1}{2}}(\mathbf{x}-\mu_m)}
\end{aligned}
$$

Since $\Sigma^{-\frac{1}{2}}$ is symmetric :

$$
= e^{-\frac{1}{2}((\mathbf{x}-\mu_m)^T\Sigma^{-\frac{1}{2}}\Sigma\Sigma^{-\frac{1}{2}}(\mathbf{x}-\mu_m)}
$$

Substituting $\Sigma = \Sigma^{\frac{1}{2}}\Sigma^{\frac{1}{2}}$:

$$
\begin{aligned}
&= e^{-\frac{1}{2}((\mathbf{x}-\mu_m)^T\Sigma^{-\frac{1}{2}}\Sigma^{\frac{1}{2}}\Sigma^{\frac{1}{2}}\Sigma^{-\frac{1}{2}}(\mathbf{x}-\mu_m)} \\
&= e^{-\frac{1}{2}((\mathbf{x}-\mu_m)^T(\mathbf{x}-\mu_m)}
\end{aligned}
$$

So $B(\bar{\mathbf{x}}|\bar{\mu}_m, \Sigma)$ is equivalent to the spherical basis network with parameters $\mathbf{x}$, μ_m, $\sigma_m^2 = 1$. ■

Question 4

Now consider a more general "elliptical" radial basis function network with Gaussian basis functions $B(\mathbf{x}|\mu_m, \Sigma_m) = exp(-\frac{1}{2}(\mathbf{x} - \mu_m)^T\Sigma_m(\mathbf{x} - \mu_m))$. Here the matrix Σ_m is allowed to be different for each basis function. In standard feed—forward neural networks the "hidden units" compute (transfer) functions that vary in only one direction in the input space. Characterize the type of matrices Σ_m that would cause radial basis functions to have this property also. In this sense general (elliptical) radial basis functions networks can be viewed as a generalization of standard feed-forward networks as well.

First, we assume that $\mu_m = 0$ to simplify the notation. Otherwise we can define $\bar{\mathbf{x}} = \mathbf{x} - \mu_m$ and follow the reasoning with $\bar{\mathbf{x}}$ below.

Since Σ_m is symmetric, we know $\Sigma_m = V\Lambda V^T = \sum_{j=1}^{p} \lambda_j v_j v_j^T$, where V is an orthonormal basis for $\mathbb{R}^p$, and $\mathbf{x} \in \mathbb{R}^p$.

We are interested in defining Σ_m such that the basis function network only allows x to change in 1 direction, which we call $\mathbf{v}_{j^*}$. Using the basis V from above, we are interested in finding the Σ_m such that for all vectors $\mathbf{v}_j \in V, j \neq j^*$,

$$B(\mathbf{x} + \mathbf{v}_j|0, \Sigma_m)) = B(\mathbf{x}|0, \Sigma_m)) \tag{1}$$

For (1) to happen, Σ_m must be such that:

$$\begin{aligned}
B(\mathbf{x} + \mathbf{v}_j|0, \Sigma_m)) &= B(\mathbf{x}|0, \Sigma_m)) \\
e^{-\frac{1}{2}(\mathbf{x}+\mathbf{v}_j)^T\Sigma_m(\mathbf{x}+\mathbf{v}_j)} &= e^{-\frac{1}{2}\mathbf{x}^T\Sigma_m\mathbf{x}}
\end{aligned}$$

Taking the log of both sides:

$$\begin{aligned}
-\frac{1}{2}(\mathbf{x} + \mathbf{v}_j)^T\Sigma_m(\mathbf{x} + \mathbf{v}_j) &= -\frac{1}{2}\mathbf{x}^T\Sigma_m\mathbf{x} \\
(\mathbf{x} + \mathbf{v}_j)^T\Sigma_m(\mathbf{x} + \mathbf{v}_j) &= \mathbf{x}^T\Sigma_m\mathbf{x} \\
(\mathbf{x}^T + \mathbf{v}_j^T)\Sigma_m(\mathbf{x} + \mathbf{v}_j) &= \mathbf{x}^T\Sigma_m\mathbf{x} \\
(\mathbf{x}^T\Sigma_m + \mathbf{v}_j^T\Sigma_m)(\mathbf{x} + \mathbf{v}_j) &= \mathbf{x}^T\Sigma_m\mathbf{x} \\
\mathbf{x}^T\Sigma_m\mathbf{x} + \mathbf{x}^T\Sigma_m\mathbf{v}_j + \mathbf{v}_j^T\Sigma_m\mathbf{x} + \mathbf{v}_j^T\Sigma_m\mathbf{v}_j &= \mathbf{x}^T\Sigma_m\mathbf{x} \\
\mathbf{x}^T\Sigma_m\mathbf{x} + 2\mathbf{x}^T\Sigma_m\mathbf{v}_j + \mathbf{v}_j^T\Sigma_m\mathbf{v}_j &= \mathbf{x}^T\Sigma_m\mathbf{x} \\
2\mathbf{x}^T\Sigma_m\mathbf{v}_j + \mathbf{v}_j^T\Sigma_m\mathbf{v}_j &= 0 \\
(2\mathbf{x} + \mathbf{v}_j)^T\Sigma_m\mathbf{v}_j &= 0
\end{aligned}$$

So now we need to find a Σ_m such that the above property holds.

Recall that $\Sigma_m = \sum_{j=1}^{p} \lambda_j v_j v_j^T$. We assert that if Σ_m is a rank 1 matrix of the form $\Sigma_m = \lambda_{j^*}\mathbf{v}_{j*}\mathbf{v}_{j^*}^T$ – i.e. assume that for all values $j \neq j^*$, $\lambda_j = 0$ – then the above property does hold.

Assume that $\Sigma_m = \lambda_{j^*}\mathbf{v}_{j*}\mathbf{v}_{j^*}^T$. Then for any $\mathbf{v}_j$, $j \neq j^*$:

$$\begin{aligned}
(2\mathbf{x} + \mathbf{v}_j)^T\Sigma_m\mathbf{v}_j &= \sum_{j=1}^{p} \lambda_j(2\mathbf{x} + \mathbf{v}_j)^T\mathbf{v}_j\mathbf{v}_j^T\mathbf{v}_j \\
&= \lambda_{j^*}(2\mathbf{x} + \mathbf{v}_j)^T\mathbf{v}_{j^*}\mathbf{v}_{j^*}^T\mathbf{v}_j \\
\text{Since } V \text{ is an orthonormal basis, for all } j \neq j^*, \mathbf{v}_{j^*}^T\mathbf{v}_j = 0: & \\
&= \lambda_{j^*}(2\mathbf{x} + \mathbf{v}_j)^T\mathbf{v}_{j^*}(0) \\
&= 0
\end{aligned}$$

Therefore we have shown that for $\Sigma_m = \lambda_{j^*}\mathbf{v}_{j*}\mathbf{v}_{j^*}^T$, $B(\mathbf{x} + \mathbf{v}_j|0, \Sigma_m)) = B(\mathbf{x}|0, \Sigma_m))$ holds for all $j \neq j^*$, and the basis function can only vary in the direction v_{j^*}

Question 5

Describe K—fold cross-validation. What is it used for. What are the advan- tages/disadvantages of using more folds (increasing K). When does cross—validation estimate the performance of the actual predicting function being used.

In K—fold cross-validation, we split our training data into disjoint subsets of m/K examples each, where m is the total number of training examples. We then train the model K times, leaving out a different m/K as a cross-validation set each time. We estimate the model's error as the average cross-validation error over each

of the K folds. If you are evaluating multiple models, you often pick the model with the lowest average error from K-fold cross validation and retrain it on the entire dataset. The extreme of this procedure is $K = m$, also known as Leave-One-Out cross validation.

Disadvantages of using more folds (as $K \to N$):

- As more and more of the data is included in the training set, the answer for the error will have higher variance as the training sets become more similar to each other.
- More computation time to execute the full K-fold test

Advantages of using more folds:

- You can use more of your available data to train, meaning that your solution will have less bias (Depending on how sensitive your problem is to the size of the training set used)
- Assuming that your final model will be the one trained on the full dataset, your cross-validation estimator will be approximately unbiased for the expected prediction error of your final model

In general, cross-validation measures the error of the modeling procedure (i.e. how good is a neural network with hyperparameters {H} at accomplishing a particular task) as opposed to the actual model or "predicting function" that you will use to predict the target on future data. This is because for each fold, you learn a new set of values for the parameters. Cross-validation only estimates the performance of the actual predicting function to be used if the parameter weights learned for each fold are identical.

Question 6

Suppose there are several outcome variables $\{y_1, y_2, ..., y_M\}$ associated with a common set of predictor variables $x = \{x_1, x_2, ..., x_n\}$. One could train separate single output neural networks for each outcome y_m or train a single network with multiple outputs, one for each y_m. What are the relative advantages/disadvantages of these two respective approaches. In what situations would one expect each to be better than the other.

Advantages of training multiple networks:

- You can finetune each network to predict a particular outcome variable very well
- If the outcomes are unrelated and the important subspace of the predictor space is very different for each, then you will probably hurt your learning by training on all of the outcomes in a single network with multiple outputs.

Disadvantages:

- Requires more computation time - you have to train M networks as opposed to just training 1.
- More parameters means that you can end up in more local minima (more likely to overfit)
- For each of the individual networks, you are not using the information that you have that $\mathbf{x}$ also predicts $\{y_j\}, j \neq k$ where y_k is the outcome variable of your network. Training a network with auxiliary losses can be thought of us a form of regularization – encouraging representations in the lower levels of the network that are more meaningful (And create a network that is more generalizable).

Question 7

Spam Email. The data sets spam_stats315B_train.csv, spam_stats315B_test.csv and documentation for this problem are the same as in Homework 2 and can be found in the class web page. You need first to standardize predictors and choose all the weights starting values at random in the interval [-0.5, 0.5].

see: https://piazza.com/class/jfehm8n4ied2w9?cid=256

```
rm(list = ls())

#Utility functions

#get overall accuracy of a model for a given threeshold
get_accuracy <- function(y_hat, y, threshold){
  y_hat[y_hat >  threshold] <- 1
  y_hat[y_hat <= threshold] <- 0
  correct <- y_hat == y
  pct_correct = sum(correct)/length(correct)
  return(pct_correct)
}

#returns misclassification rates for overall and each class for a given threshold
get_misclassification_rates <- function(model, threshold){
  y_hat <- predict(model, test_scaled_X)
  y_hat[y_hat >  threshold] <- 1
  y_hat[y_hat <= threshold] <- 0

  correct <- y_hat == test_y
  correct_spam    <- correct[test_y == 1]
  correct_nonspam <- correct[test_y == 0]

  misclassification_rate <- 1 - sum(correct)/length(correct)
  spam_misclassification_rate <- 1 - sum(correct_spam)/length(correct_spam)
  nonspam_misclassification_rate <- 1 - sum(correct_nonspam)/length(correct_nonspam)

  return(c(
    misclassification_rate,
    spam_misclassification_rate,
    nonspam_misclassification_rate)
  )
}

#display numeric as percent
percent <- function(x, digits = 2, format = "f", ...) {
  paste0(formatC(100 * x, format = format, digits = digits, ...), "%")
}

set_seed <- function(){
  set.seed(123)
}
```

```
#remove all variables except for functions
rm(list = setdiff(ls(), lsf.str()))
```

```
#data labels
rflabs<-c("make", "address", "all", "3d", "our", "over", "remove",
  "internet","order", "mail", "receive", "will",
  "people", "report", "addresses","free", "business",
  "email", "you", "credit", "your", "font","000","money",
  "hp", "hpl", "george", "650", "lab", "labs",
  "telnet", "857", "data", "415", "85", "technology", "1999",
  "parts","pm", "direct", "cs", "meeting", "original", "project",
  "re","edu", "table", "conference", ";", "(", "[", "!", "$", "#",
  "CAPAVE", "CAPMAX", "CAPTOT","type")

#load the data
data_path <- paste(getwd(),'/data',sep='')
setwd(data_path)
train <- read.csv(file="spam_stats315B_train.csv", header=FALSE, sep=",")
test <- read.csv(file="spam_stats315B_test.csv", header=FALSE, sep=",")
colnames(train)<-rflabs
colnames(test)<-rflabs

#scale the predictors (X) in train and test by the train data
num_cols <- dim(train)[2]
train_X <- train[,c(1:(num_cols-1))]
train_X_means <- apply(train_X,2,mean)
train_X_sds <- apply(train_X,2,sd)

train_scaled_X <- train_X
train_scaled_X <- sweep(train_scaled_X, 2, train_X_means, "-")
train_scaled_X <- sweep(train_scaled_X, 2, train_X_sds, "/")

test_X <- test[,c(1:(num_cols-1))]
test_scaled_X <- test_X
test_scaled_X <- sweep(test_scaled_X, 2, train_X_means, "-")
test_scaled_X <- sweep(test_scaled_X, 2, train_X_sds, "/")

#create y vectors
train_y <- data.frame(train[,'type'])
test_y  <- data.frame(test[,'type'])
```

(a) Fit on the training set one hidden layer neural networks with 1, 2,..., 10 hidden units and different sets of starting values for the predictors (obtain in this way one model for each number of units). Which structural model performs best at classifying on the test set?

The structural model with 4 hidden units performs best at classifying the test set, achieving an estimated (test set) accuracy of 92.24% . The second best was 3 hidden units, achieving 91.66% accuracy.

reference: https://piazza.com/class/jfehm8n4ied2w9?cid=235

```
#remove all except functions and data
to_keep <- ls()[!(ls() %in% c('train_scaled_X','train_y','test_scaled_X','test_y'))]
rm(list = setdiff(to_keep, lsf.str()))

#set parameters
num_neurons <- seq(1:10)
num_reps <- 10
wt_rang = 0.5
```

```r
threshold <- .50
accuracies <- c()

#for each structural model
for(size in num_neurons){

  sum_accuracy <- 0

  #average accuracy over num_reps random initializations
  for(i in c(1:num_reps)){

    set_seed()

    model <- nnet(
        train_scaled_X, train_y, size=num_neurons[size],
        linout = FALSE, entropy = FALSE, softmax = FALSE,
        censored = FALSE, skip = FALSE, rang = wt_rang, decay = 0,
        maxit = 100, Hess = FALSE, trace = FALSE
    )
    y_hat <- predict(model, test_scaled_X)
    sum_accuracy <- sum_accuracy + get_accuracy(y_hat, test_y, threshold)
  }

  accuracies <- c(accuracies, sum_accuracy/num_reps)
}

#get the num_neurons corresponding with model that produced highest average accuracy
best_performing_idx <- which.max(accuracies)
best_performing_num_neurons <- num_neurons[best_performing_idx]
best2_performing_idx <- which.max(accuracies[accuracies!=max(accuracies)])
best2_performing_num_neurons <- num_neurons[best2_performing_idx]

#Report output
cat("Best performing number of hidden layer neurons: ",
    best_performing_num_neurons, "\n",
    "Accuracy: ",
    percent(accuracies[best_performing_idx]),"\n")

cat("2nd best performing number of hidden layer neurons: ",
    best2_performing_num_neurons, "\n",
    "Accuracy: ",
    percent(accuracies[best2_performing_idx]),"\n")
```

(b) Choose the optimal regularization (weight decay for parameters 0,0.1,...,1) for the structural model found above by averaging your estimators of the misclassification error on the test set. The average should be over 10 runs with different starting values. Describe your final best model obtained from the tuning process: number of hidden units and the corresponding value of the regularization parameter. What is an estimation of the misclassification error of your model?

Our best model uses 4 neurons, and a weight decay of 0.1. It has an overall estimated (test set) misclassification rate of 4.69% (spam 5.34%, and nonspam 4.26%). We see this is an improvement over the unregularized version, which had an estimated misclassification rate of 7.76%.

see: https://piazza.com/class/jfehm8n4ied2w9?cid=223

```r
#remove all except functions, data and best structural info
to_keep <- ls()[
  !(ls() %in% c('train_scaled_X','train_y','test_scaled_X','test_y','best_performing_num_neurons'))
]
rm(list = setdiff(to_keep, lsf.str()))

#set parameters
weight_decays <- seq(0,1,.1)
num_reps <- 10
wt_rang = 0.5
threshold <- .50
accuracies <- c()

#for each weight decay
for(weight_decay in weight_decays){

  sum_accuracy <- 0

  #average over several random initializations
  for(i in c(1:num_reps)){

    set_seed()

    model <- nnet(
        train_scaled_X, train_y, size= best_performing_num_neurons,
        linout = FALSE, entropy = FALSE, softmax = FALSE,
        censored = FALSE, skip = FALSE, rang = wt_rang, decay = weight_decay,
        maxit = 100, Hess = FALSE, trace = FALSE
    )

    y_hat <- predict(model, test_scaled_X)
    sum_accuracy <- sum_accuracy + get_accuracy(y_hat, test_y, threshold)
  }

  accuracies <- c(accuracies, sum_accuracy/num_reps)
}

#get the weight decay corresponding with model that produced highest average accuracy
best_performing_idx <- which.max(accuracies)
best_performing_weight_decay <- weight_decays[best_performing_idx]

#get misclassification rates for our best chosen parameters
set_seed()
model <- nnet(
    train_scaled_X, train_y, size= best_performing_num_neurons,
    linout = FALSE, entropy = FALSE, softmax = FALSE,
    censored = FALSE, skip = FALSE, rang = wt_rang, decay = best_performing_weight_decay,
    maxit = 100, Hess = FALSE, trace = FALSE
)
misclassification_rates <- get_misclassification_rates(model, threshold)

#Report output
cat("Best number of hidden units: ", best_performing_num_neurons, "\n")
```

```
cat("Best weight decay for chosen structure: ", best_performing_weight_decay, "\n\n")

cat("Misclassifiction rates: ","\n")
cat(percent(misclassification_rates[1]), ": Overall","\n")
cat(percent(misclassification_rates[2]), ": Spam","\n")
cat(percent(misclassification_rates[3]), ": Nonspam","\n")
```

(c) As in the previous homework the goal now is to obtain a spam filter. Repeat the previous point requiring this time the proportion of misclassified good emails to be less than 1%.

Our best filter requiring the proportion of misclassified nonspam to be less than 1% has a hidden layer size of 6 and weight decay of 0.2. This achieves an overall misclassification rate of 6.58% (spam: 14.89%, nonspam: 0.98%). We find the best filter by fitting a model for each combination of a number of hidden units in $\{1, 2, ..., 10\}$ and the weight decay in $\{0.0, 0.1, ..., 1.0\}$. We then find the minimum threshold that misclassifies at most 1% of nonspam emails for each model. By applying to each model it's own threshold, we find the model and threshold combination that yields the lowest overall misclassification rate while maintaining a nonspam misclassification rate of at most 1%.

See: https://piazza.com/class/jfehm8n4ied2w9?cid=257

```
#remove all except functionsand data
to_keep <- ls()[!(ls() %in% c('train_scaled_X','train_y','test_scaled_X','test_y'))]
rm(list = setdiff(to_keep, lsf.str()))

#function that finds threshold for a 1% or lower nonspam misclassification rate
#for a given model
find_threshold <- function(model){

  thresholds <- seq(0,1,0.01)
  y_hat <- predict(model, test_scaled_X)

  for(thresh in thresholds){
    y_hat_nonspam <- y_hat[test_y == 0]
    y_hat_nonspam[y_hat_nonspam >  thresh] <- 1
    y_hat_nonspam[y_hat_nonspam <= thresh] <- 0
    nonspam_misclassification_rate <- sum(y_hat_nonspam)/length(y_hat_nonspam)

    if(nonspam_misclassification_rate <= 0.01) {break}
  }
  return(thresh)
}

#set parameters
num_neurons <- seq(1:10)
weight_decays <- seq(0,1,.1)
wt_rang = 0.5

#fit a model to each set of parameters
models <- list()
for(i in c(1:length(num_neurons))){
  for(j in c(1:length(weight_decays))){

    set_seed()

    model <- nnet(
```

```
        train_scaled_X, train_y, size=num_neurons[i],
        linout = FALSE, entropy = FALSE, softmax = FALSE,
        censored = FALSE, skip = FALSE, rang = wt_rang, decay = weight_decays[j],
        maxit = 100, Hess = FALSE, trace = FALSE
    )
    models[[paste(i,j,sep="_")]] <- model
  }
}

#find the threshold that returns < 1% nonspam misclassification rate for each model
model_thresholds <- lapply(models, function(x) {find_threshold(x)})

#get the overall, spam, and non-spam misclassification rates at each threshold
misclassification_rates <- mapply(get_misclassification_rates, models, model_thresholds)

#find the model with the lowest overall misclassification rate
#(using forced < 1% nonspam threshold)
best_model_idx <- which.min(misclassification_rates[1,])
best_model <- models[[best_model_idx]]
best_misclassification_rates <- misclassification_rates[,best_model_idx]

#report best model
cat("Best model: ","\n")
cat("Hidden layer size: ", best_model$n[2], "\n")
cat("Decay: ", best_model$decay, "\n\n")

cat("Misclassifiction rates: ","\n")
cat(percent(best_misclassification_rates[1]), ": Overall","\n")
cat(percent(best_misclassification_rates[2]), ": Spam","\n")
cat(percent(best_misclassification_rates[3]), ": Nonspam","\n")
```