Stats 315B Homework

Rachael Caelie (Rocky) Aikens, Daniel Sosa, Christine Tataru

Problem 1

First, some notation. Let $K^{(l)}$ denote the number of nodes in the l^{th} layer of a neural net, and let $l = 0...L$ index the layers of the neural net, with $l = 0$ denoting the input layer and $l = L$ denoting the output layer. Additionally, we define:

$$\delta_j^{(l)} = \frac{\partial q}{\partial a_j^{(l)}},$$

Where $a_j^{(l)} = \sum_{j=0}^{K^{(l)}} w_{ij} o_i^{(l-1)}$.

Here, I am assuming that we are applying stochastic gradient descent and I have dropped the subscript (t) denoting the observation we are using, for the purposes of this derivation. I also assume that $o_0^{(l)} = 1$ for each layer, denoting the intercept inputs to each layer

To define the update rule, we need to calculate $G(w_{ij}^{(l)}) = \frac{\partial q}{\partial w_{ij}^{(l)}}$. Applying the chain rule, this is:

$$\begin{aligned}\frac{\partial q}{\partial w_{ij}^{(l)}} &= \frac{\partial q}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial w_{ij}^{(l)}} \\ &= \delta_j^{(l)} o_i^{(l-1)}\end{aligned}$$

Moreover, we already know that $\delta^{(L)} = y - o^{(L)}$, the error for this example from forward propagation. Applying the chain rule, we can recursively calculate $\delta_j^{(l)}$ backwards through the network as:

$$\delta_j^{(l)} = S'(a_j^{(l)}) \sum_{k=1}^{K^{(l+1)}} \delta_k^{(l+1)} w_{jk}.$$

For the sigmoid activation function, this is:

$$\delta_j^{(l)} = o_j^{(l)}(1 - o_j^{(l)}) \sum_{k=1}^{K^{(l+1)}} \delta_k^{(l+1)} w_{jk}.$$

Now we can write an algorithm for backpropagation (let e denote the error from forward propagation, O the outputs from each node in forward propagation, x the example we are using, and W the weights for the whole network).

Backprop(e, O, W, x):

$\delta^{(l)} = e$

$g(w_i^{(l)}) = \delta^{(l)} o_i^{(l)}$

For $l = L - 1$ to 1:

$\{\delta_j^{(l)} = S'(a_j^{(l)}) \sum_{k=1}^{K^{(l+1)}} \delta_k^{(l+1)} w_{jk}$ for each $j = 1...K^{(l)}\}$

$\{g(w_{ij}^{(l)}) = \delta_j^{(l)} o_i^{(l-1)}$ for each $j = 1...K^{(l)}, i = 0...K^{(l-1)}\}$

$\{w_{ij}^{(l)} \leftarrow w_{ij} - \eta g(w_{ij}^{(l)})$ for each $j = 1...K^{(l)}, i = 0...K^{(l-1)}, l = 1...L\}$

Problem 2

$$\frac{\partial \hat{F}}{\partial a_m} = B(\mathbf{x}|\mu_m, \sigma_m) \tag{1}$$

$$\frac{\partial \hat{F}}{\partial \sigma_m} = \Sigma_{m=1}^{M} a_m (\frac{1}{\sigma_m^3} \Sigma_{j=1}^{n} (x_j - \mu_{jm})^2) B(\mathbf{x}|\mu_m, \sigma_m) \tag{2}$$

$$\frac{\partial \hat{F}}{\partial \mu_{mj}} = \Sigma_{m=1}^{M} (\frac{a_m}{\sigma_m^2} (x_j - \mu_{jm})) B(\mathbf{x}|\mu_m, \sigma_m) \tag{3}$$

Problem 3

We can show that the ellipsoidal radial basis function can be written as the spherical gaussian basis function:

$$\exp\left\{-\frac{1}{2}\sum_{j=1}^{p}(\tilde{x}_j - \tilde{\mu}_j)^2\right\},$$

For some transformed $\tilde{x}$ and $\tilde{\mu}$.

Simply notice that Σ is a positive semidefinite matrix, so it has a positive semidefinite square root, $\Sigma^{1/2}$. Moreover, any positive semidefinite matrix is symmetric and invertible. We can then manipulate the ellipsoidal basis function as follows:

$$\begin{aligned} B(x|\mu_m, \Sigma) &= \exp\left\{-\frac{1}{2}(x - \mu_m)^T \Sigma (x - \mu_m)\right\} \\ &= \exp\left\{-\frac{1}{2}(x - \mu_m)^T \Sigma^{1/2} \Sigma^{1/2} (x - \mu_m)\right\} \\ &= \exp\left\{-\frac{1}{2}(\Sigma^{1/2}(x - \mu_m))^T (\Sigma^{1/2}(x - \mu_m))\right\} \end{aligned}$$

Letting $\Sigma^{1/2} x = \tilde{x}$ and $\Sigma^{1/2} \mu_m = \tilde{\mu}$, we retrieve the spherical basis function desired. Note also that $\Sigma^{1/2}$ is invertible, so we can easily convert back to the original x and μ_m with the inverse transformation $x = \Sigma^{-1/2} \tilde{x}$ and $\mu_m = \Sigma^{-1/2} \tilde{\mu}$.

Problem 4

For the elliptical radial basis function to vary only in one direction, Σ must be some matrix so that $B(x|\mu_m, \Sigma)$ is proportional to some univariate normal p.d.f.. This requires that Σ must be rank 1. If we require that Σ is restricted to the set of symmetric matrices (this not explicit in the problem statement but is necessary to maintain the analogy to the gaussian p.d.f.), this is simply the set of all $p \times p$ matrices with a single nonzero diagonal entry and all other entries equal to zero.

Problem 5

K-fold validation is a method used for cross-validation, or to attempt to find regularization parameters that sufficiently generalize a model in question to other data (favoring bias over high variance). In this procedure, the data is randomly divided into K sets, whereby the model is built and trained on $K-1$ sets and the left out set is used for validation/regularization parameter tuning. This procedure is repeated K times and the parameters are the average parameters found from each iteration of training and validating.

Increasing K...

Advantages:

1. Averaging over more folds to find optimal parameters. Averaging over more folds decreases variance, so we're more confident in the parameters chosen.
2. Training set is larger, each fold creates a model that better estimates the training data likely.

Disadvantages:

1. Increasing number of folds decreases the size of the data used for training in each fold (each fold produces a worse model).
2. Validation set is smaller, estimate of parameters may be poor
3. As a result of 2, the KFCV may fail to produce parameter settings that don't overfit the data
4. More computationally expensive.

Cross-validation will estimate the performance of the actual predicting function when the data held out for validation is drawn from the same distribution as the rest of the training data, otherwise the CV may be skewed by influential outliers for example.

Problem 6

Training separate neural nets for each y_m...

Advantages:

1. A more accurate model for each y_m.
2. Perhaps different input data is important for different y_m's and by separating the neural nets, the learned parameters are more robust to uninformative variables.

Disadvantages:

1. Computationally very expensive potentially
2. Inefficient way to make a model if there do exist dependencies between the y_m's
3. Not attempting to capture inherent hierarchical "structure" from the input data (as in the case with filters in image processing for example).

Training separate neural nets might make sense when the y_m's are dependent on different variables entirely and have little correlation. Training with the same neural net would make sense when there's more of a dependency structure between the y_m's.

Problem 7

```
library(dplyr)
spam_train <- read.csv("spam_stats315B_train.csv", header=F)
spam_test <- read.csv("spam_stats315B_test.csv", header=F)
rflabs<-c("make", "address", "all", "threed", "our", "over", "remove",
  "internet","order", "mail", "receive", "will",
  "people", "report", "addresses","free", "business",
  "email", "you", "credit", "your", "font","zerozerozero","money",
  "hp", "hpl", "george", "sixfifty", "lab", "labs",
  "telnet", "eightfiftyseven", "data", "fourfifteen", "eightyfive", "technology", "nineteenninetynine",
  "parts","pm", "direct", "cs", "meeting", "original", "project",
  "re","edu", "table", "conference", "semicolon", "left_bracket_round", "left_bracket_square", "exclaim
  "CAPAVE", "CAPMAX", "CAPTOT","type")
  # Names for predictors and response
colnames(spam_train) <- colnames(spam_test) <- rflabs


#normalize all columns by mean and variance
train_type = spam_train$type
spam_train <- data.frame(apply(select(spam_train, -type), 2,
                               function(vec) return((vec - mean(vec)) / sd(vec))))
spam_train$type = train_type

test_type = spam_test$type
spam_test <- data.frame(apply(select(spam_test, -type), 2,
                              function(vec) return((vec - mean(vec)) / sd(vec))))
spam_test$type = 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).

```
#install.packages("neuralnet")
library(nnet)


spam_train$type = as.factor(spam_train$type)

find_opt_hidden_units <- function(minimize_false_pos = F){

  #train all nets
  nets = list()
  for(num_units in 1:10){
    net = nnet(type ~ ., data = spam_train, size = num_units, trace = F, rang = 0.5, maxit = 500) #weig
    nets[[paste("h", num_units, sep = "")]] = net
  }


  #select best net
  class_preds_list = list()
  errors = list()
  false_pos_rates = list()

  good_emails <- spam_test$type == 0
```

```
  for(name in names(nets)){
    net = nets[[name]]
    class_probs <- predict(net, select(spam_test, -type), type = "raw")
    if(minimize_false_pos == F){
      thresh = 0.5
    }
    if(minimize_false_pos == T){
      thresh = limit_false_pos(false_pos_limit = 0.01, class_probs, good_emails, spam_test$type)
    }
    class_preds <- ifelse(class_probs < thresh, 0, 1)
    error = sum(class_preds != spam_test$type) / length(class_preds)
    false_pos_rate <- sum(class_preds[good_emails] != 0) / sum(good_emails) #our predictions on good em

    class_preds_list[[`name`]] <- class_preds
    errors[[`name`]] <- error
    false_pos_rates[[`name`]] <- false_pos_rate
  }

  opt_hidden_units = as.numeric(which(unlist(errors) == min(unlist(errors))))

  return(list(opt_hidden_units, min(unlist(errors))))
}


limit_false_pos <- function(false_pos_limit = .01, class_probs, good_emails, y){
    thresh = 0.5
    class_preds <- ifelse(class_probs < thresh, 0, 1) #higher threshold means fewer email classified as
    false_pos_rate <- sum(class_preds[good_emails] != 0) / sum(good_emails) #our predictions on good em
    rates <- c()
    while(false_pos_rate > false_pos_limit){
      #for each net, find the threshold s.t. false positive rate is under 1%
      thresh <- thresh + .001
      class_preds <- ifelse(class_probs < thresh, 0, 1) #higher threshold means fewer email classified
      false_pos_rate <- sum(class_preds[good_emails] != 0) / sum(good_emails) #our predictions on good
      rates <- c(rates, false_pos_rate)
    }
  return(thresh)
}
```

which structural model performs best at classifying on the test set?

```
set.seed(0)
tmp <- find_opt_hidden_units(minimize_false_pos = F)
opt_hidden_units <- tmp[[1]]
error <- tmp[[2]]
print(paste("Optimal number of hidden units: ", opt_hidden_units, "with error: ", error))
```

```
## [1] "Optimal number of hidden units:  7 with error:  0.0462842242503259"
```

b)Choose the optimal regularization (weight decay for parameters 0, 0.1, . . . , 1) for the structural model found above by averageing your estimators of the misclassification error on the test set. The average should be over 10 runs with different starting values.

```
find_opt_decay_rate <- function(opt_hidden_units, minimize_false_pos = F){
  good_emails <- spam_test$type == 0
```

```
  errors <- list()
  for(decay in seq(0,1, by = 0.1)){
    errors_tmp <- c()
    for(i in 1:10){
       net = nnet(type ~ ., data = spam_train, size = opt_hidden_units, decay = decay, trace = F, rang =
       class_probs <- predict(net, select(spam_test, -type), type = "raw")

       if(minimize_false_pos == T){
          thresh = limit_false_pos(false_pos_limit = 0.01, class_probs, good_emails, spam_test$type)
       }
       if(minimize_false_pos == F){
          thresh = 0.5
       }
       class_preds <- ifelse(class_probs < thresh, 0, 1)
       error = sum(class_preds != spam_test$type) / length(class_preds)
       false_pos_rate <- sum(class_preds[good_emails] != 0) / sum(good_emails) #our predictions on good
       errors_tmp <- c(errors_tmp, error)
    }
    errors[[paste("decay", decay, sep = "")]] <- mean(errors_tmp)
  }

  opt_decay = seq(0,1, by = 0.1)[which(unlist(errors) == min(unlist(errors)))]
  return(list(opt_decay, min(unlist(errors))))
}
```

```
tmp <- find_opt_decay_rate(opt_hidden_units, minimize_false_pos = F)
opt_decay <- tmp[[1]]
min_error <- tmp[[2]]
print(paste("Optimal number of hidden units: ", opt_hidden_units))
```

```
## [1] "Optimal number of hidden units:  7"
```

```
print(paste("Optimal decay rate: " ,opt_decay))
```

```
## [1] "Optimal decay rate:  0.1"
```

```
print(paste("With minimum error of: ", min_error))
```

```
## [1] "With minimum error of:  0.0452411994784876"
```

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?

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

```
#opt_hidden_units <- find_opt_hidden_units(minimize_false_pos = T)
#print(paste("Optimal number of hidden units: ", opt_hidden_units))

tmp <- find_opt_decay_rate(opt_hidden_units, minimize_false_pos = T)
opt_decay <- tmp[[1]]
min_error <- tmp[[2]]

print(paste("After mandating the false positive rate be less than 1 % and network take the optimal numb
```

```
## [1] "After mandating the false positive rate be less than 1 % and network take the optimal number of
```

The misclassification error is obviously much higher after mandating that the false positive rate be low,

because this causes us to misclassify more spam emails as good emails.