

# 1 Problem 1

## 1.1 Part a

We are given

$$h_j = \sum_{k=1}^K \left( w_k^{(j)} \right)^2 \quad (1)$$

and asked to prove that  $0 \leq h_j \leq 1$  and  $\sum_{j=1}^D h_j = K$ . To start, I will prove the first fact. Since, by definition  $h_j$  is the square of a vector, it will clearly be  $\geq 0$ . This trivial fact need not be explicitly proved, but is shown merely by the definition of  $h_j$ . Decomposing  $h_j$  and knowing that  $w_k = U^T a_k$ , we see that

$$h_j = u_j^T (a_1 a_1^T + \cdots + a_K a_K^T) u_j \quad (2)$$

It's also easy to see, given  $A = [a_1 \dots a_K]$ , that

$$AA^T = a_1 a_1^T + \cdots + a_K a_K^T \quad (3)$$

and therefore

$$h_j = u_j^T AA^T u_j \quad (4)$$

With this definition, we can decompose  $AA^T$  via its singular value decomposition

$$A = U_2 \Lambda_2 V_2^T \quad (5)$$

$$\Rightarrow AA^T = U_2 \Lambda_2 V_2^T (U_2 \Lambda_2 V_2^T)^T = U_2 \Lambda_2 \Lambda_2^T U_2^T \quad (6)$$

where  $U_2^T U_2 = U_2 U_2^T = I$  from the definition of SVD. Therefore, our definition of  $h_j$  can be reformulated as:

$$h_j = u_j^T U_2 \Lambda_2 \Lambda_2^T U_2^T u_j \quad (7)$$

and we must try and prove

$$u_j^T U_2 \Lambda_2 \Lambda_2^T U_2^T u_j \leq I \quad (8)$$

where  $u_j^T u_j = U_2^T U_2 = I$ . We now focus on  $\Lambda_2 \Lambda_2^T$ , the eigenvalues of  $AA^T$ . From numerous sources and linear systems definitions, it is true that any non-zero eigenvalue of  $AA^T$  is also an eigenvalue of  $A^T A$ . To prove this fact, consider the eigendecomposition of  $AA^T$  such that, for eigenvalues  $\lambda$  and eigenvectors  $v$

$$AA^T v = \lambda v \quad (9)$$

multiply both side by  $A^T$ , and see

$$A^T A (A^T v) = \lambda (A^T v) \quad (10)$$

therefore, the eigenvectors are rotated via the matrix  $A^T$  such that  $\bar{v} = A^T v$  while the eigenvalues remain the same. This will continue to hold for all non-zero eigenvalues of  $AA^T$ . Therefore, we

can say that all non-zero eigenvalues of  $AA^T$  are 1 (since  $A^T A = I$ ) and  $D - K$  eigenvalues will be zero. It is pretty easy to say, given this fact, that  $\Lambda_2 \Lambda_2^T \leq I$  due to the zero eigenvalues. Therefore,

$$u_j^T U_2 \Lambda_2 \Lambda_2^T U_2^T u_j \leq u_j^T U_2 I U_2^T u_j \quad (11)$$

$$= u_j^T U_2 U_2^T u_j \quad (12)$$

$$= u_j^T I u_j \quad (13)$$

$$= I \quad (14)$$

and therefore

$$u_j^T U_2 \Lambda_2 \Lambda_2^T U_2^T u_j \leq I \quad (15)$$

and the proof is complete.

To prove  $\sum_{j=1}^D h_j = K$ , consider taking the transpose of  $h_j$ , which can be done since  $h_j$  is a scalar. The transpose will be

$$h_j = a_1^T u_j u_j^T a_1 + \cdots + a_K^T u_j u_j^T a_K \quad (16)$$

With this definition, the sum of all  $h_j$  will be

$$\sum_{j=1}^D h_j = (a_1^T u_1 u_1^T a_1 + \cdots + a_K^T u_1 u_1^T a_K) + \cdots + (a_1^T u_D u_D^T a_1 + \cdots + a_K^T u_D u_D^T a_K) \quad (17)$$

$$= a_1^T (u_1 u_1^T + \cdots + u_D u_D^T) a_1 + \cdots + a_K^T (u_1 u_1^T + \cdots + u_D u_D^T) a_K \quad (18)$$

At this point, it is easy to see  $UU^T = u_1 u_1^T + \cdots + u_D u_D^T = I$  from the definition of singular value decomposition. Therefore, the above can be simplified to

$$\sum_{j=1}^D h_j = a_1^T a_1 + \cdots + a_K^T a_K = \sum_{i=1}^K a_i^T a_i = \sum_{i=1}^K 1 = K \quad (19)$$

$a_1^T a_1 = 1$  since  $A^T A = I$  and therefore

$$a_i^T a_j = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} \quad (20)$$

and the above sum holds. Therefore, both proofs are complete.

## 1.2 Part b

We are asked to prove that

$$\max_{W^T W = I_K} \sum_{k=1}^K w_k^T \Lambda w_k = \max_{h_j} \sum_{j=1}^D h_j \lambda_j \quad (21)$$

To prove this, first see that the condition  $W^T W = I_K$  is the same in both optimizations. Additionally, since  $h_j \geq 0$  and is a convex function of  $W$ , maximizing  $W$  is the same as maximizing  $h_j$ .

Now, we must simply prove  $\sum_{k=1}^K w_k^T \Lambda w_k = \sum_{j=1}^D h_j \lambda_j$ . To do so, we see that

$$w_j^T \Lambda w_j = \begin{bmatrix} w_j^{(1)} \\ \vdots \\ w_j^{(D)} \end{bmatrix} \begin{bmatrix} \lambda_1 & & \\ & \lambda_2 & \\ & & \ddots \\ & & & \lambda_D \end{bmatrix} \begin{bmatrix} w_j^{(1)} & \dots & w_j^{(D)} \end{bmatrix} \quad (22)$$

$$= \lambda_1 \left(w_j^{(1)}\right)^2 + \dots + \lambda_D \left(w_j^{(D)}\right)^2 \quad (23)$$

and therefore

$$\sum_{k=1}^K w_k^T \Lambda w_k = \left( \lambda_1 \left(w_1^{(1)}\right)^2 + \dots + \lambda_D \left(w_1^{(D)}\right)^2 \right) + \dots + \left( \lambda_1 \left(w_K^{(1)}\right)^2 + \dots + \lambda_D \left(w_K^{(D)}\right)^2 \right) \quad (24)$$

$$= \lambda_1 \left( \left(w_1^{(1)}\right)^2 + \dots + \left(w_K^{(1)}\right)^2 \right) + \dots + \lambda_D \left( \left(w_1^{(D)}\right)^2 + \dots + \left(w_K^{(D)}\right)^2 \right) \quad (25)$$

$$= \sum_{i=1}^D \lambda_i \left( \left(w_1^{(i)}\right)^2 + \dots + \left(w_K^{(i)}\right)^2 \right) \quad (26)$$

$$= \sum_{i=1}^D \lambda_i h_i \quad (27)$$

since, from earlier

$$h_j = \sum_{k=1}^K \left(w_k^{(j)}\right)^2 \quad (28)$$

### 1.3 Part c

We are asked to find the optimal  $h_j$  in

$$\max_{\substack{h_j \\ w^T w = I_K}} \sum_{j=1}^D h_j \lambda_j \quad (29)$$

and show that  $a_k = u_k (k = 1, \dots, K)$  is a solution to the above. To do so, I will first find the optimal  $h_j$ . To find the optimal points, consider three important facts. First,  $0 \leq h_j \leq 1$ , second  $\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_D$ , and third  $\sum_{j=1}^D h_j = K$ . With these three facts, the optimal solution becomes trivial. It becomes a linear problem in our eigenvalues  $\lambda$ . Clearly, if you could choose you would want  $h_1 = h_2 = \dots = h_K = 1$  and  $h_{K+1} = \dots = h_D = 0$ . This way, the above optimization becomes

$$\sum_{j=1}^K \lambda_j \quad (30)$$

i.e., the sum of the  $K$  largest eigenvalues. If there were  $h_j$  values at any index larger than  $K$ , there would be a contribution from a  $\lambda$  value which was smaller than  $\lambda_K$ . This would clearly lower the overall objective value and be a non-optimal solution, since  $\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_D$ .

To check  $a_k = u_k (k = 1, \dots, K)$  is a solution to the above, remember the definition of  $h_j$ , where

$$h_j = a_1^T u_j u_j^T a_1 + \dots + a_K^T u_j u_j^T a_K \quad (31)$$

with the above definition, the overall sum of  $h$  values will be

$$\sum_{j=1}^D h_j \lambda_j = \lambda_1 (u_1^T u_1 u_1^T u_1 + \dots + u_K^T u_1 u_1^T u_K) + \dots \quad (32)$$

$$+ \lambda_K (u_1^T u_K u_K^T u_1 + \dots + u_K^T u_K u_K^T u_K) + \dots + \lambda_D (u_1^T u_D u_D^T u_1 + \dots + u_K^T u_D u_D^T u_K) \quad (33)$$

$$= \lambda_1 (1 + 0 + \dots + 0) + \dots + \lambda_K (0 + 0 + \dots + 1) + \lambda_{K+1} (0 + \dots + 0 + 0) + \dots \quad (34)$$

$$= \sum_{j=1}^K \lambda_j \quad (35)$$

which is the same optimal solution I showed above.

## 1.4 Part d

We are asked if the solution to part c is unique. For the subspace of the solution  $\{a_1^*, \dots, a_K^*\}$  to be unique we need

$$\lambda_K > \lambda_{K+1} \quad (36)$$

And therefore  $\{a_1^*, \dots, a_K^*\}$  is unique  $\Leftrightarrow \lambda_K > \lambda_{K+1}$ . Consider the case when  $\lambda_K = \lambda_{K+1}$ . In this case, given  $h_1, \dots, h_{K-1} = 1$  the optimization cost would not change if  $h_K = 1$ ,  $h_{K+1} = 0$  or  $h_K = 0$ ,  $h_{K+1} = 1$ . Therefore, we have more than one optimal solution and the solution above is not unique.

## 1.5 Part e

In this section, we are asked to show that if  $a_k$  ( $k = 2, \dots, K$ ) is orthogonal to  $u_1, u_2, \dots, u_{k-1}$  then  $a_k = u_k$  is a solution of

$$\begin{aligned} \max_{\substack{a_k \\ a_k^T a_k = 1 \\ a^T u_i = 0, i=1, \dots, k-1}} a_k^T S a_k \end{aligned} \quad (37)$$

We first notice that

$$a_k^T S a_k = a_k^T U \Lambda U^T a_k \quad (38)$$

$$= a_k^T \begin{bmatrix} u_1 & u_2 & \dots & u_D \end{bmatrix} \begin{bmatrix} \lambda_1 & & & \\ & \lambda_2 & & \\ & & \ddots & \\ & & & \lambda_D \end{bmatrix} \begin{bmatrix} u_1^T \\ u_2^T \\ \vdots \\ u_D^T \end{bmatrix} a_k \quad (39)$$

$$= \begin{bmatrix} a_k^T u_1 & \dots & a_k^T u_D \end{bmatrix} \begin{bmatrix} \lambda_1 & & & \\ & \lambda_2 & & \\ & & \ddots & \\ & & & \lambda_D \end{bmatrix} \begin{bmatrix} u_1^T a_k \\ \vdots \\ u_D^T a_k \end{bmatrix} \quad (40)$$

$$= a_k^T u_1 \lambda_1 u_1^T a_k + \dots + a_k^T u_D \lambda_D u_D^T a_k \quad (41)$$

Now, given  $a_k$  ( $k = 2, \dots, K$ ) is orthogonal to  $u_1, u_2, \dots, u_{k-1}$ , we get

$$0 + 0 + \dots + a_k^T u_k \lambda_k u_k^T a_k + \dots + a_k^T u_D \lambda_D u_D^T a_k \quad (42)$$

$$= a_k^T (\lambda_k u_k u_k^T + \dots + \lambda_D u_D u_D^T) a_k \quad (43)$$

Now, we aim to maximize the above value. We are given that  $\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_D$  and  $a_k^T a_k = 1$ . Let's say that  $a_k = 0.5u_k + 0.5u_j$  where  $j > k$ . The coefficients (0.5, 0.5) must add to one since  $a_k^T a_k = 1$  is a constraint. Nonetheless, with this formulation of  $a_k$  we see our objective function becomes  $0.5\lambda_k + 0.5\lambda_j \leq \lambda_k$  since  $\lambda_j \leq \lambda_k$ . This will continue to hold for different number of and values of coefficient. Clearly, the optimal solution is when the objective function is  $\lambda_k$ . This will only hold when  $a_k = u_k$ , and therefore this is a solution of the optimization problem.

## 2 Problem 2

We are given linear matrices  $A \in \mathbb{R}^{K \times M+1}$ ,  $S \in \mathbb{R}^{D \times K+1}$ ,  $W \in \mathbb{R}^{N \times D+1}$  such that:

$$h(t) = \frac{1}{1 + e^{-t}} \quad (44)$$

$$y_i = h \left( a_{i,M+1} + \sum_{j=1}^M a_{i,j} x_j \right) \quad (45)$$

$$z_i = h \left( s_{i,K+1} + \sum_{j=1}^K s_{i,j} y_j \right) \quad (46)$$

$$P_i = \frac{\exp \left( \sum_{j=1}^D w_{i,j} z_j + w_{i,D+1} \right)}{\sum_{k=1}^N \exp \left( \sum_{j=1}^D w_{k,j} z_j + w_{k,D+1} \right)} \quad (47)$$

$$(48)$$

and the loss function is defined, for a data point  $x$  and label  $t$ , as

$$E(W, S, A) = - \sum_{i=1}^N 1(t=i) \log(P_i) \quad (49)$$

### 2.1 Part a: Derive the gradients $\nabla_W E, \nabla_S E, \nabla_A E$

#### 2.1.1 $\nabla_W E$

To find  $\nabla_W E$ , we see:

$$E(W, S, A) = - \sum_{i=1}^N 1(t=i) \left[ \left( \sum_{j=1}^D w_{i,j} z_j + w_{i,D+1} \right) - \log \left( \sum_{k=1}^N \exp \left( \sum_{j=1}^D w_{k,j} z_j + w_{k,D+1} \right) \right) \right] \quad (50)$$

Taking this expression term by term, we find the partial derivative of the first term, i.e.,

$$\frac{\partial E_1(W, S, A)}{\partial w_{i,j}} = \frac{\partial}{\partial w_{i,j}} \left( - \sum_{i=1}^N 1(t=i) \left[ \left( \sum_{j=1}^D w_{i,j} z_j + w_{i,D+1} \right) \right] \right) \quad (51)$$

$$= -1(t=i) z_j \quad (52)$$

What this equates to, is each row of  $\frac{\partial E_1(W, S, A)}{\partial w_{i,j}}$  is the  $z$  components corresponding to the column, i.e.,

$$\frac{\partial E_1(W, S, A)}{\partial w_{i,j}} = \begin{bmatrix} z_1 & z_2 & \dots & z_D \\ \vdots & & & \vdots \\ z_1 & z_2 & \dots & z_D \end{bmatrix} \quad (53)$$

However, we also have the indicator function, which means all rows of  $\frac{\partial E(W, S, A)}{\partial w_{i,j}}$  are equal to zero with the exception of the  $t^{th}$  row. For the second term in the equation, i.e.

$$\frac{\partial E_2(W, S, A)}{\partial w_{i,j}} = \frac{\partial}{\partial w_{i,j}} \left( \sum_{i=1}^N 1(t=i) \left[ \log \left( \sum_{k=1}^N \exp \left( \sum_{j=1}^D w_{k,j} z_j + w_{k,D+1} \right) \right) \right] \right) \quad (54)$$

$$(55)$$

the derivative is more complicated. The first thing to note is that the summation inside the logarithm is non-dependent on the summation with the indicator function. Since  $t$  is just one value, the logarithm term will only appear once, at the  $t^{th}$  point. Therefore, we drop the indicator function and rather only consider the derivative at  $w_{i,j}$ , which will only matter at  $w_{t,j}$ . From the properties of logarithm, we see this is equal to

$$\frac{\partial E_2(W, S, A)}{\partial w_{i,j}} = \frac{z_j \exp \left( \sum_{j=1}^D w_{i,j} z_j + w_{i,D+1} \right)}{\sum_{k=1}^N \exp \left( \sum_{j=1}^D w_{k,j} z_j + w_{k,D+1} \right)} = z_j P_i \quad (56)$$

and therefore, the overall gradient is:

$$\frac{\partial E(W, S, A)}{\partial w_{i,j}} = -1(t=i) (z_j - z_j P_i) \quad (57)$$

$$= - \begin{bmatrix} z_1 - z_1 P_1 & z_2 - z_2 P_1 & \dots & z_D - z_D P_1 \\ \vdots & & & \vdots \\ z_1 - z_1 P_N & z_2 - z_2 P_N & \dots & z_D - z_D P_N \end{bmatrix} \quad (58)$$

and

$$\frac{\partial E(W, S, A)}{\partial w_{i,D+1}} = - \begin{bmatrix} 1 - P_1 \\ 1 - P_2 \\ \vdots \\ 1 - P_D \end{bmatrix} \quad (59)$$

where the only row that won't be zero is the row  $i$  where  $t = i$ .

### 2.1.2 $\nabla_S E$

To find this gradient, we must implement the chain rule. To do so, we see:

$$\frac{\partial E}{\partial s_{k,j}} = \frac{\partial E}{\partial z_k} \frac{\partial z_k}{\partial t_k} \frac{\partial t_k}{\partial s_{k,j}} \quad (60)$$

To find this, I will separate each partial derivative out and compute it individually.

$$\frac{\partial E}{\partial z_k} :$$

From earlier, we see

$$E(W, S, A) = - \sum_{i=1}^N 1(t=i) \left[ \left( \sum_{j=1}^D w_{i,j} z_j + w_{i,D+1} \right) - \log \left( \sum_{k=1}^N \exp \left( \sum_{j=1}^D w_{k,j} z_j + w_{k,D+1} \right) \right) \right] \quad (61)$$

Now, we must take the partial with respect to  $z_k$ . Once again, I separate out this term into two parts, and see

$$E_1(W, S, A) = \left( - \sum_{i=1}^N 1(t=i) \left[ \left( \sum_{j=1}^D w_{i,j} z_j + w_{i,D+1} \right) \right] \right) \quad (62)$$

$$\frac{\partial E_1(W, S, A)}{\partial z_k} = \frac{\partial}{\partial z_k} \left( - \sum_{i=1}^N 1(t=i) \left[ \left( \sum_{j=1}^D w_{i,j} z_j + w_{i,D+1} \right) \right] \right) \quad (63)$$

$$= - \sum_{i=1}^N 1(t=i) w_{i,k} \quad (64)$$

The second term,  $E_2$ , is defined as:

$$\frac{\partial E_2(W, S, A)}{\partial z_k} = \frac{\partial}{\partial z_k} \left( \sum_{i=1}^N 1(t=i) \left[ \log \left( \sum_{m=1}^N \exp \left( \sum_{j=1}^D w_{m,j} z_j + w_{m,D+1} \right) \right) \right] \right) \quad (65)$$

$$= \sum_{i=1}^N 1(t=i) \frac{\sum_{m=1}^N w_{m,k} \exp \left( \sum_{j=1}^D w_{m,j} z_j + w_{m,D+1} \right)}{\sum_{m=1}^N \exp \left( \sum_{j=1}^D w_{m,j} z_j + w_{m,D+1} \right)} \quad (66)$$

and therefore:

$$\frac{\partial E}{\partial z_k} = - \sum_{i=1}^N 1(t=i) \left( w_{i,k} - \frac{\sum_{m=1}^N w_{m,k} \exp \left( \sum_{j=1}^D w_{m,j} z_j + w_{m,D+1} \right)}{\sum_{m=1}^N \exp \left( \sum_{j=1}^D w_{m,j} z_j + w_{m,D+1} \right)} \right) \quad (67)$$

$\frac{\partial z_k}{\partial t_k}$  :

Next, we attempt to find  $\frac{\partial z_k}{\partial t_k} = \frac{\partial h(t_k)}{\partial t_k}$ , where:

$$z_k = h(t_k) \quad (68)$$

From the definition of sigmoids, we know:

$$\frac{\partial h(t_k)}{\partial t_k} = h(t_k)[1 - h(t_k)] = z_k[1 - z_k] \quad (69)$$

$\frac{\partial t_k}{\partial s_{k,j}}$  :

Lastly, we find  $\frac{\partial t_k}{\partial s_{k,j}}$ . This proves to be a much easier proposition. Remember

$$t_k = s_{i,K+1} + \sum_{j=1}^K s_{i,j} y_j \quad (70)$$



and therefore

$$\frac{\partial t_k}{\partial s_{k,j}} = y_j \quad (71)$$

Therefore, our overall partial derivative is:

$$\frac{\partial E}{\partial s_{k,j}} = \left( - \sum_{i=1}^N 1(t=i) \left( w_{i,k} - \frac{\sum_{m=1}^N w_{m,k} \exp \left( \sum_{j=1}^D w_{m,j} z_j + w_{m,D+1} \right)}{\sum_{m=1}^N \exp \left( \sum_{j=1}^D w_{m,j} z_j + w_{m,D+1} \right)} \right) \right) z_k [1 - z_k] y_j \quad (72)$$

and

$$\frac{\partial E}{\partial s_{k,K+1}} = \left( - \sum_{i=1}^N 1(t=i) \left( w_{i,k} - \frac{\sum_{m=1}^N w_{m,k} \exp \left( \sum_{j=1}^D w_{m,j} z_j + w_{m,D+1} \right)}{\sum_{m=1}^N \exp \left( \sum_{j=1}^D w_{m,j} z_j + w_{m,D+1} \right)} \right) \right) z_k [1 - z_k] \quad (73)$$

### 2.1.3 $\nabla_A E$

As in the previous vein, we see, via the chain rule,

$$\frac{\partial E}{\partial a_{i,j}} = \frac{\partial E}{\partial z} \frac{\partial z}{\partial t} \frac{\partial t}{\partial y_i} \frac{\partial y_i}{\partial t_i} \frac{\partial t_i}{\partial a_{i,j}} \quad (74)$$

We start at the first two partials, i.e.,  $\frac{\partial E}{\partial z} \frac{\partial z}{\partial t} \frac{\partial t}{\partial y_i}$ . Since we are now considering the entire vector  $z$ , via the properties of the chain rule we see:

$$\frac{\partial E}{\partial z} \frac{\partial z}{\partial t_{k1}} \frac{\partial t_{k1}}{\partial y_i} = \sum_{k=1}^D \frac{\partial E}{\partial z_k} \frac{\partial z_k}{\partial t} \frac{\partial t}{\partial y_i} \quad (75)$$

$$= \sum_{k=1}^D \left( - \sum_{i=1}^N 1(t=i) \left( w_{i,k} - \frac{\sum_{m=1}^N w_{m,k} \exp \left( \sum_{j=1}^D w_{m,j} z_j + w_{m,D+1} \right)}{\sum_{m=1}^N \exp \left( \sum_{j=1}^D w_{m,j} z_j + w_{m,D+1} \right)} \right) \right) z_k [1 - z_k] s_{k,i} \quad (76)$$

Using the previous definition of a derivative of a sigmoid and the definition of  $y_i$ , we see:

$$\frac{\partial y_i}{\partial t_{k2}} \frac{\partial t_{k2}}{\partial a_{k,j}} = y_i [1 - y_i] x_j \quad (77)$$

and  $\frac{\partial E}{\partial a_{i,j}} =$

$$\left( \sum_{k=1}^D \left( - \sum_{i=1}^N 1(t=i) \left( w_{i,k} - \frac{\sum_{m=1}^N w_{m,k} \exp \left( \sum_{j=1}^D w_{m,j} z_j + w_{m,D+1} \right)}{\sum_{m=1}^N \exp \left( \sum_{j=1}^D w_{m,j} z_j + w_{m,D+1} \right)} \right) \right) z_k [1 - z_k] s_{k,i} \right) y_i [1 - y_i] x_j \quad (78)$$

and  $\frac{\partial E}{\partial a_{i,M+1}} =$

$$\left( \sum_{k=1}^D \left( - \sum_{i=1}^N 1(t=i) \left( w_{i,k} - \frac{\sum_{m=1}^N w_{m,k} \exp \left( \sum_{j=1}^D w_{m,j} z_j + w_{m,D+1} \right)}{\sum_{m=1}^N \exp \left( \sum_{j=1}^D w_{m,j} z_j + w_{m,D+1} \right)} \right) \right) z_k [1 - z_k] s_{k,i} \right) y_i [1 - y_i] \quad (79)$$

which can be extrapolated for each term in  $A$ .

## 2.2 Part b

To implement the forward and backward algorithms, I used the starter code with some adjustments. The code is shown below:

```
import numpy as np
from itertools import izip
import random

epsilon = 10e-4
M = 100
K = 50
D = 30
N = 10
nCheck = 1000

def sigmoid(x):
    return 1 / (1+np.exp(-x))

def forwardprop(x, t, A, S, W):
    # ----- make your implementation here -----

    y = np.transpose(sigmoid(A[:,M]+np.transpose(np.dot(A[:,0:M],x))))
    z = np.transpose(sigmoid(S[:,K]+np.transpose(np.dot(S[:,0:K],y))))
    P = np.transpose(np.exp(W[:,D]+np.transpose(np.dot(W[:,0:D],z))))
    P_den = np.sum(P)
    P = P/P_den
    J = -np.log(P[t])
    # -----

    return y, z, P, J, P_den

def backprop(x, t, A, S, W):
```

```

y, z, P, J, P_den = forwardprop(x, t, A, S, W)
I = np.zeros((N, 1), dtype=np.float)
I[t] = 1

# ----- make your implementation here -----

grad_W = np.zeros(np.shape(W))

for i in range(0,D):
    grad_W[t,i] = -z[i] + z[i]*P[t]
    grad_W[t,D] = -1+P[t]

grad_S = np.zeros(np.shape(S))
dEdzk=0

dEdzk = np.zeros((D))
dEdzk2 = np.zeros((D))

for k in range(0,D):
    for i in range(0,N):
        dEdzk[k] += W[i,k]*(np.exp(W[i,D]+np.dot(W[i,0:D],z)))/P_den
    for j in range(0,K):
        grad_S[k,j] = -(W[t,k]-dEdzk[k])*z[k]*(1-z[k])*y[j]
        grad_S[k,K] = -(W[t,k]-dEdzk[k])*z[k]*(1-z[k])

dEdz=0
dEdz2=0

grad_A = np.zeros(np.shape(A))

for i in range(0,K):
    dEdz=0
    for k in range(0,D):
        dEdz += -(W[t,k] - dEdzk[k])*z[k]*(1-z[k])*S[k,i]
    for j in range(0,M):
        grad_A[i,j] = dEdz*y[i]*(1-y[i])*x[j]
        grad_A[i,M] = dEdz*y[i]*(1-y[i])

# -----

return grad_A, grad_S, grad_W

```

```

def gradient_check():
    A = np.random.rand(K, M+1)*0.1-0.05
    S = np.random.rand(D, K+1)*0.1-0.05
    W = np.random.rand(N, D+1)*0.1-0.05
    x, t = np.random.rand(M, 1)*0.1-0.05, np.random.choice(range(N), 1)[0]

    grad_A, grad_S, grad_W = backprop(x, t, A, S, W)
    errA, errS, errW = [], [], []

    for i in range(nCheck):

        # ----- make your implementation here -----
        idx_x, idx_y = random.randint(0,K-1), random.randint(0,M)
        # numerical gradient at (idx_x, idx_y)
        A[idx_x,idx_y]+=epsilon
        y, z, P, J, P_den = forwardprop(x, t, A, S, W)
        A[idx_x,idx_y]-=2*epsilon
        y, z, P, J2, P_den = forwardprop(x, t, A, S, W)

        numerical_grad_A = (J-J2)/(2*epsilon)
        A[idx_x,idx_y]+=epsilon

        errA.append(np.abs(grad_A[idx_x, idx_y] - numerical_grad_A))

    idx_x, idx_y = random.randint(0,D-1), random.randint(0,K)

    # numerical gradient at (idx_x, idx_y)
    S[idx_x,idx_y]+=epsilon
    y, z, P, J, P_den = forwardprop(x, t, A, S, W)
    S[idx_x,idx_y]-=2*epsilon
    y, z, P, J2, P_den = forwardprop(x, t, A, S, W)

    numerical_grad_S = (J-J2)/(2*epsilon)
    S[idx_x,idx_y]+=epsilon

    errS.append(np.abs(grad_S[idx_x, idx_y] - numerical_grad_S))

    idx_x, idx_y = random.randint(0,N-1), random.randint(0,D)
    # numerical gradient at (idx_x, idx_y)
    W[idx_x,idx_y]+=epsilon
    y, z, P, J, P_den = forwardprop(x, t, A, S, W)

```

```

W[idx_x,idx_y]-=2*epsilon
y, z, P, J2, P_den = forwardprop(x, t, A, S, W)
numerical_grad_W = (J-J2)/(2*epsilon)

W[idx_x,idx_y]+=epsilon

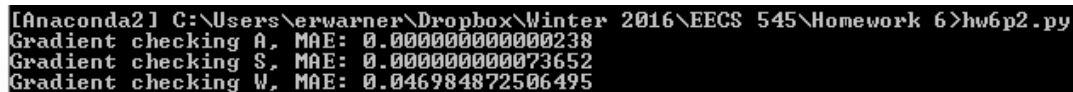
errW.append(np.abs(grad_W[idx_x, idx_y] - numerical_grad_W))
# -----

print 'Gradient checking A, MAE: %.8f' % np.mean(errA)
print 'Gradient checking S, MAE: %.8f' % np.mean(errS)
print 'Gradient checking W, MAE: %.8f' % np.mean(errW)

if __name__ == '__main__':
    gradient_check()

```

With the following output:



```

[Anaconda2] C:\Users\erwarner\Dropbox\Winter 2016\EECS 545\Homework 6>hw6p2.py
Gradient checking A, MAE: 0.0000000000000238
Gradient checking S, MAE: 0.000000000073652
Gradient checking W, MAE: 0.046984872506495

```

As you can see, the lowest error occurs in the gradient of  $E$  with respect to  $A$ . The error gets larger for  $S$  and is largest at  $W$ . This makes sense to me, as  $A$  is the furthest into the backward algorithm and therefore will not change quite as much as the other two matrices.

### 3 Problem 3

The code for Kaggle is shown below. I used an aggregate of three classifiers, with the median taken as my solution.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris, load_digits
from sklearn.tree import DecisionTreeClassifier
from sklearn.cross_validation import StratifiedKFold, train_test_split
from sklearn.ensemble import BaggingClassifier, RandomForestClassifier
from sklearn import metrics
import cPickle as pickle
import pandas
from sklearn.ensemble import ExtraTreesClassifier
from scipy import stats

#-----Training Data-----
train = open('train.pkl','rb')
Train = pickle.load(train)

X = Train['data']
Y = Train['labels']

#-----Test Data-----

test = open('test.pkl','rb')
Test = pickle.load(test)

X_test = np.array(Test['data'])

X_train = np.array(X)
y_train = np.array(Y)

ERF = ExtraTreesClassifier(n_estimators=1000,max_depth=None,
    min_samples_split=1, random_state=0)
ERF.fit(X_train,y_train)
accuracy = ERF.score(X_train,y_train)
ERF_test = ERF.predict(X_test)
print accuracy

RF = ExtraTreesClassifier(n_estimators=1000)
RF.fit(X_train,y_train)
accuracy = RF.score(X_train,y_train)
RF_test = RF.predict(X_test)
print accuracy

BC = BaggingClassifier(n_estimators=200)
```

```
BC.fit(X_train,y_train)
accuracy = BC.score(X_train,y_train)
BC_test = BC.predict(X_test)
print accuracy

Y_test = []
for i in range(0,len(X_test)):
    Test = np.array([BC_test[i],RF_test[i],ERF_test[i]])
    med = np.median(Test)
    med2 =int(med)
    Y_test.append(med2)

#Write to a Dataframe
df = pandas.DataFrame( Y_test,range(1,len(Y_test)+1),columns=['category'])

df.to_csv('erwaner_total_100.csv')
```

## 4 Problem 4

### 4.1 Part a

For one iteration of the Adaboost, I found using the code:

```
After one iteration, alpha is 0.458145365937
After one iteration, W is [ 0.1  0.25  0.1  0.25  0.1  0.1  0.1 ]
```

```
import numpy as np
import math
import random

X = np.array([[0.22,0.19],[0.30,0.17],[0.45,0.20],[0.20,0.18],[0.12,0.17],[0.10,0.15],[0.30,0.15]])
y = np.array([-1,-1,-1,1,1,1,1])

W0 = 1.0/len(X)*np.ones((1,7))
W= W0[0]
n=7

alpha = np.ones((7,1))
theta1 = np.array([0,-1])
theta0 = 0.17
classifier = np.zeros((n,1))

#-----PART A-----#

epsilon =0

for i in range(0,n):
    classifier[i,0] = np.sign(X[i][0]*theta1[0]+X[i][1]*theta1[1]+theta0)
    if classifier[i] != y[i]:
        epsilon += W[i]

alpha = 1.0/2.0*np.log((1.0-epsilon)/epsilon)
for i in range(0,n):
    W[i] = W[i]*math.exp(-classifier[i]*alpha*y[i])

W = W/np.sum(W)
print "After one iteration, alpha is ",alpha
print "After one iteration, W is", W
```

### 4.2 Part b

Using a value of  $M = 10,000$ , I was able to find a boosted classifier. The code to do so is shown below:

```
#-----PART B-----#
```



```

theta_array = np.array([2,-1,0.17])
M          = 10

classifier = np.zeros((n,1))
f = np.zeros((n,1))
W0 = 1.0/len(X)*np.ones((1,7))
W= W0[0]

m=0
hT=0

while m<M:

    min = 100
    thetaT = np.array([-1,1])
    b0      = np.array([np.linspace(0.1, 0.5, 20),np.linspace(0.14, 0.2, 20)])

        #Find the optimal theta,b,k
    for k in [0,1]:
        for b in b0[k]:
            for theta in [-1,1]:
                epsilon=0
                for i in range(0,n):
                    f[i] =(X[i][k]*theta+b)
                    classifier[i] = np.sign(f[i]+0.0000001)

                if classifier[i] != y[i]:
                    epsilon += W[i]
                if epsilon<min:
                    min=epsilon
                    theta_min = theta
                    k_min      = k
                    b_min      = b

    theta1 = theta_min
    b       = b_min
    k       = k_min

    epsilon = 0
    j=0
        #Find the classifier values of the optimal parameters
    for i in range(0,n):
        f[i] =(X[i][k]*theta1+b)
        classifier[i] = np.sign(f[i]+0.0000001)

```

```

if classifier[i] != y[i]:
    epsilon += W[i]
j+=1

#Solve for alpha
alpha = 1.0/2.0*np.log((1.0-epsilon)/epsilon)

for i in range(0,n):
    W[i] = W[i]*math.exp(-classifier[i]*alpha*y[i])
W = W/np.sum(W)

hT += alpha*classifier
m+=1

EL =0

for i in range(0,len(hT)):
    EL += math.exp(-y[i]*(hT[i]))

print "The exponential loss is",EL
print "The hT output is",hT

```

As a note, the exponential loss was defined as  $\sum_{i=1}^N \exp(-y^{(t)}h(\bar{x}^{(t)}))$ , where  $h(\bar{x}^{(t)})$  was the sign of the classifier, as long as the classifier was completely accurate the exponential loss would be the same number. This number would be

$$n * \exp(-1) \tag{80}$$

which in our case is  $7 * \exp(-1) = 2.5752$ . Instead, I used the

### 4.3 Part c

The classifier that minimizes the exponential loss is the one that correctly classifies the most points. It does not matter where these correctly classified points are located, as long as they are correctly classified. Therefore, all of the boosted classifier will have the same exponential loss if they correctly classify the training set.

## 5 Problem 5

### 5.1 Part a

Adjusting the code for *plot\_error* to the below:

```
def cv_error(clf, X, y, k=5):
    """
    Splits the data, X and y, into k-folds. Trains a classifier on K-1 folds and
    testing on the remaining fold. Calculates the cross validation train and test
    error for classifier clf by averaging the performance across folds.
    Input:
        clf- an instance of SVC()
        X- (n,d) array of feature vectors, where n is # of examples
            and d is # of features
        y- (n,) array of binary labels {1,-1}
        k- int, # of folds (default=5)

    Returns: average train and test error across the k folds as np.float64
    """
    skf = StratifiedKFold(y, k)
    train_scores, test_scores = [], []
    for train, test in skf:
        X_train, X_test, y_train, y_test = X[train], X[test], y[train], y[test]
        clf.fit(X_train, y_train)
        train_scores.append(clf.score(X_train, y_train))
        test_scores.append(clf.score(X_test, y_test))

    return 1 - np.array(train_scores).mean(), 1 - np.array(test_scores).mean()

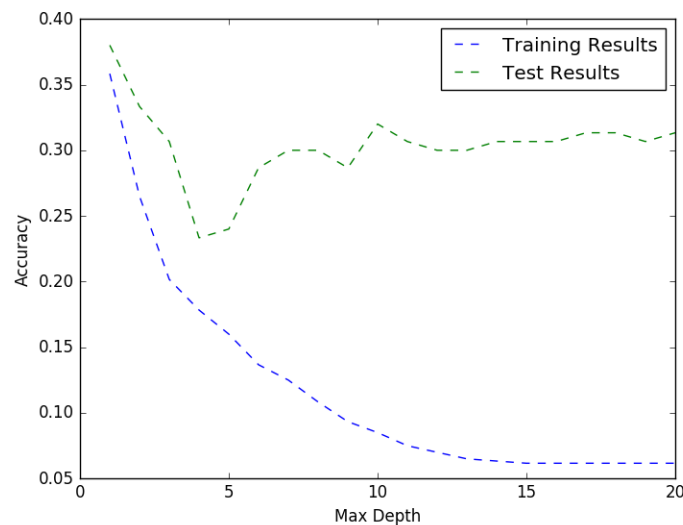
def plot_error(X, y):
    """
    Plots the variation of 5-fold cross-validation error w.r.t. maximum
    depth of the decision tree
    X- (n, d) array of feature vectors, where n is # of examples
        and d is # of features
    y- (n, ) array of labels corresponding to X
    """
    # ----- make your implementation here-----
    TrainT, TestT = [], []
    for i in range(1, 21):
        clf = DecisionTreeClassifier(criterion='entropy', max_depth=i)
        (Train, Test) = cv_error(clf, X, y)
        TrainT.append(Train)
        TestT.append(Test)

    plt.figure()
```

```

plt.plot(range(1, 21), TrainT, '--', label = 'Training Results')
plt.plot(range(1, 21), TestT, '--', label = 'Test Results')
plt.xlabel('Max Depth')
plt.ylabel('Accuracy')
plt.legend(loc='upper right')
plt.show()
return TrainT, TestT

```



As far as training error, as the depth increases the training error decreases. This is expected, as the more components in the regressor or classifier usually corresponds to lower training error. For the test data, there seems to be an optimal point (depth = 4) where the test error is lowest. Therefore, overall I would say the classifier performs best when the max depth is 4.

## 5.2 Part b

Implementing the bagging ensemble and random forest, I used to code:

```

def bagging_ensemble(X_train, y_train, X_test, y_test, m=None, n_clf = 10):
    '''
    Returns accuracy on the test set X_test with corresponding labels y_test
    using a bagging ensemble classifier with n_clf decision trees trained with
    training examples X_train and training labels y_train.
    Input:
        X_train- (n_train, d) array of training feature vectors, where n_train
            is # of examples and d is # of features
        y_train- (n_train, ) array of labels corresponding to X_train
        X_test- (n_test, d) array of testing feature vectors, where n_test is
            # of examples and d is # of features
        y_test- (n_test, ) array of labels corresponding to X_test
        m - int, # of features to consider when looking for the best split
        n_clf- # of decision tree classifiers in the bagging ensemble, default
            value of n_clf is 10
    '''

```

Output:

```
    Accuracy of the bagging ensemble classifier on X_test
'''
```

```
# ----- make your implementation here-----
BagClas = BaggingClassifier(n_estimators=n_clf)
BagClas.fit(X_train,y_train)
accuracy = BagClas.score(X_test,y_test)
return accuracy

# -----
```

```
def random_forest(X_train, y_train, X_test, y_test, m=None, n_clf = 10):
'''
```

Returns accuracy on the test set X\_test with corresponding labels y\_test using a random forest classifier with n\_clf decision trees trained with training examples X\_train and training labels y\_train.

Input:

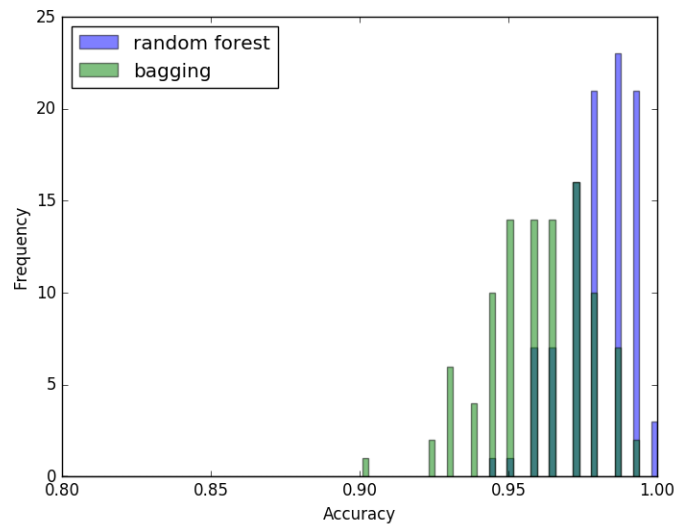
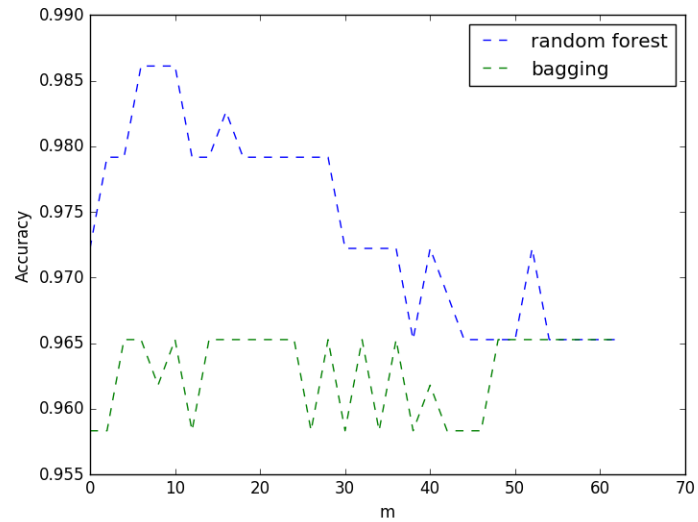
```
    X_train- (n_train, d) array of training feature vectors, where n_train
              is # of examples and d is # of features
    y_train- (n_train, ) array of labels corresponding to X_train
    X_test- (n_test, d) array of testing feature vectors, where n_test is
            # of examples and d is # of features
    y_test- (n_test, ) array of labels corresponding to X_test
    n_clf- # decision tree classifiers in the random forest, default
           value of n_clf is 10
```

Output:

```
    Accuracy of the random forest classifier on X_test
'''
```

```
# ----- make your implementation here-----
RF = RandomForestClassifier(n_estimators=n_clf,max_features=m)
RF.fit(X_train,y_train)
accuracy = RF.score(X_test,y_test)
return accuracy
# -----
```

With the following outputs:



As we increase  $m$ , the accuracy for the random forest seems to decrease. It eventually levels out at around 0.965, but before that point seems to decrease. The bagging classifier does not really change, its accuracy bounces between 0.96 and 0.965. However, both algorithms seem to settle out around 0.965 and stay at that point.