Statistical Machine Learning
Statistics GR 5241 — Spring 2022

Homework 4

Collaboration policy: Collaboration on solving the homework is allowed, after you have thought about the problems on your own. It is also OK to get clarification (but not solutions) from books or online resources, again after you have thought about the problems on your own. There are two requirements: first, cite your collaborators fully and completely (e.g., "Jane explained to me what is asked in Question 3 (a)"). Second, write your solution *independently*: close the book and all of your notes, and send collaborators out of the room, so that the solution comes from you only.
The following problems are due on Friday, April 22nd, 11:59pm.

1. **Training Error vs. Test Error**

   In this problem, we want to use the least squares estimator to illustrate the point that the training error is generally an underestimate of the prediction error (or test error).

   (15 points) Consider a linear regression model with p parameters,

   $$\mathbf{y} = \mathbf{X}\beta + \epsilon, \text{ where } \epsilon_i \overset{iid}{\sim} \mathcal{N}(0, \sigma^2).$$

   We fit the model by least squares to a set of trainning data $(x_1, y_1), \ldots, (x_N, y_N)$ drawn independently from a population. Let $\hat{\beta}$ be the least squares estimate obtained from the training data. Suppose we have some test data $(\tilde{x}_1, \tilde{y}_1), \cdots, (\tilde{x}_M, \tilde{y}_M)$ $(N \geq M > p)$ drawn at random from the same population as the training data. If $R_{tr}(\beta) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \beta^T x_i)^2$ and $R_{te}(\beta) = \frac{1}{M}\sum_{i=1}^{M}(\tilde{y}_i - \beta^T \tilde{x}_i)^2$, prove that

   $$\mathbb{E}[R_{tr}(\hat{\beta})] \leq \mathbb{E}[R_{te}(\hat{\beta})],$$

   where the expectations are over all that is random in each expression.

   Hints:

   - Consider the least squares estimate $\tilde{\beta}$ based on the test data.
   - The expection of residual sum-of-squres $\sum_{i=1}^{N}\mathbb{E}(y_i - \hat{\beta}^T x_i)^2$ is $(N - p - 1)\sigma^2$.

2. **Strongly convex functions**

Strongly convex functions is a special kind of convex function with good convergence property. A differentiable function f is called strongly convex with parameter $m > 0$ if the following inequality holds for all points x, y in its domain:

$$(\nabla f(x) - \nabla f(y))^\top (x - y) \geq m\|x - y\|_2^2. \tag{1}$$

(a) (10 points) Prove that (1) is equivalent to

$$f(y) \geq f(x) + \nabla f(x)^\top (y - x) + \frac{m}{2}\|y - x\|_2^2.$$

(b) (10 points) If f is twice differentiable, then (1) equivalent to

$$\nabla^2 f(x) \succeq mI.$$

3. **Hidden Markov chain models**

Your manager lives a simple life. Some days he is Angry and some days he is Happy. But he hides his emotional state, and so all you can observe is whether he smiles, frowns, laughs, or yells. Starting on day 1, he is in the Happy state and there is only one transition per day.

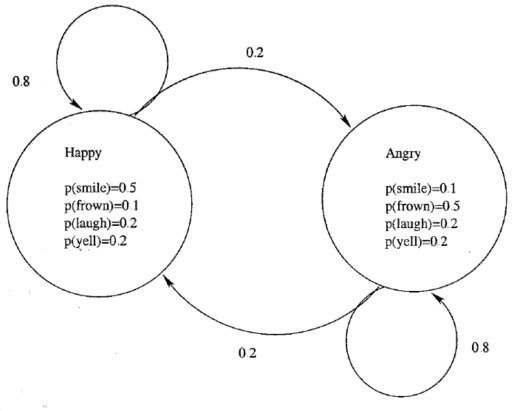


Figure 1: HMM model for your manager.

We define: q_t as state on day t and O_t as observation on day t.

(a) (3 points) What is $Pr(q_2 = Happy)$?

(b) (3 points) What is $Pr(O_2 = frown)$?

(c) (4 points) What is $Pr(q_2 = Happy | O_2 = frown)$?

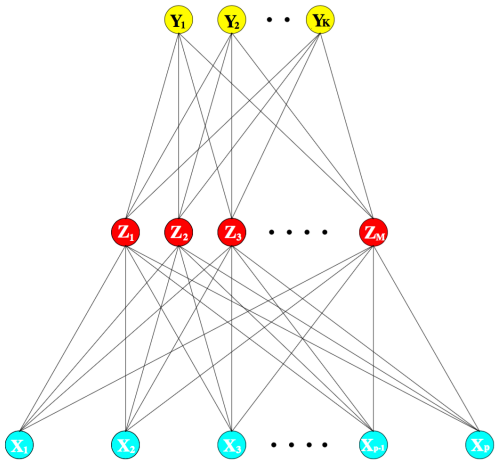
(d) (5 points) What is $Pr(O_{100} = yell)$?

(e) (5 points) Assume that $O_1 = O_2 = O_3 =$ frown. What is the most likely sequence of the states? (i.e, what are q_1, q_2 and q_3).

4. **Neural networks**

Assume that we fit a single layer hidden neural network in a regression problem on $\mathbb{R}^p$. Recall our model is:

$$\begin{aligned} Z_m &= \sigma(\alpha_{0m} + \alpha_m^T X), \quad m = 1, \ldots, M. \\ f_k(X) &= \beta_{0k} + \beta_k^T Z, \quad k = 1, \ldots, K. \end{aligned}$$



The parameters of the model are $\theta = \{\alpha_{0m}, \alpha_m, \beta_{0k}, \beta_k\}$ (each α_m is a p-dimensional vector and β_k is an M-dimensional vector. We use gradient descent to minimize the squared error loss

$$R(\theta) = \sum_{i=1}^{n} \sum_{k=1}^{K} (y_{ik} - f_k(x_i))^2.$$

A gradient update at the $(r+1)$st iteration has the form

$$\beta_{km}^{(r+1)} = \beta_{km}^{(r)} - \frac{\partial R}{\partial \beta_{km}^{(r)}}, \quad \alpha_{ml}^{(r+1)} = \alpha_{ml}^{(r)} - \frac{\partial R}{\partial \alpha_{ml}^{(r)}}.$$

An issue of neural networks is that they have too many weights and will overfit the data. Therefore, regularization is necessary. Instead of minimizing the empirical risk $R(\theta)$, we add a penalty $J(\theta)$ to it with the form

$$J(\theta) = \sum_{k,m} \beta_{km}^2 + \sum_{m,l} \alpha_{ml}^2.$$

Now the object function of the optimization problem becomes

$$R(\theta) + \lambda J(\theta).$$

(a) (15 points) Write down the gradient update for this regularized problem. (You need to calculate $\frac{\partial R}{\partial \beta_{km}}$ and $\frac{\partial R}{\partial \alpha_{ml}}$)

(b) (5 points) Discuss why stochastic gradient descent will help when n is large.

5. **Implementation of SVM via Gradient Descent**

In the problem, you will implement the soft margin SVM using different gradient descent methods. You can use either R or Python for this problem. Our training data consists of n pairs $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$, with $x_i \in \mathbb{R}^d$ and $y_i \in \{-1, 1\}$. Define a hyperplane by

$$\{x : f(x) = x^T \mathbf{w} + b = 0\}.$$

A classification rule induced by $f(x)$ is

$$H(x) = \text{sign}(x^T \mathbf{w} + b).$$

To recap, to estimate the $\mathbf{w}$, b of the soft margin SVM, we can minimize the cost:

$$f(\mathbf{w}, b) = \frac{1}{2} \sum_{j=1}^{d} (w^{(j)})^2 + C \sum_{i=1}^{n} \max \left\{ 0, 1 - y_i \left(\sum_{j=1}^{d} w^{(j)} x_i^{(j)} + b \right) \right\}. \tag{2}$$

Define $L(\mathbf{w}, b; x_i, y_i) = \max\{0, 1 - y_i(\sum_{j=1}^{d} w^{(j)} x_i^{(j)} + b)\}$. In order to minimize the cost function, we first obtain the gradient with respect to $w^{(j)}$, the jth item in the vector $\mathbf{w}$, and b as follows:

$$\begin{aligned} \nabla_{w^{(j)}} f(\mathbf{w}, b) &= \frac{\partial f(\mathbf{w}, b)}{\partial w^{(j)}} = w_j + C \sum_{i=1}^{n} \frac{\partial L(\mathbf{w}, b; x_i, y_i)}{\partial w^{(j)}}, \\ \nabla_b f(\mathbf{w}, b) &= \frac{\partial f(\mathbf{w}, b)}{\partial b} = C \sum_{i=1}^{n} \frac{\partial L(\mathbf{w}, b; x_i, y_i)}{\partial b}, \end{aligned} \tag{3}$$

where

$$\begin{aligned} \frac{\partial L(\mathbf{w}, b; x_i, y_i)}{\partial w^{(j)}} &= \begin{cases} 0 & \text{if } y_i(x_i^T \mathbf{w} + b) \geq 1 \\ -y_i x_i^{(j)} & \text{otherwise.} \end{cases} \\ \frac{\partial L(\mathbf{w}, b; x_i, y_i)}{\partial b} &= \begin{cases} 0 & \text{if } y_i(x_i^T \mathbf{w} + b) \geq 1 \\ -y_i & \text{otherwise.} \end{cases} \end{aligned}$$

Now, we will implement and compare the following gradient descent techniques:

- **Batch gradient descent:** Iterate through the entire dataset and update the parameters as follows:

  $k = 0$
  while convergence criteria not reached **do**
    for $j = 1, \ldots, d$ **do**
      Update $w^{(j)} \leftarrow w^{(j)} - \eta \nabla_{w^{(j)}} f(\mathbf{w}, b)$
    end for

Update $b \leftarrow b - \eta \nabla_b f(\mathbf{w}, b)$

Update $k \leftarrow k + 1$

end while

where,

n is the number of samples in the training data,

d is the dimensions of $\mathbf{w}$,

η is the learning rate of the gradient descent, and

$\nabla_{w^{(j)}} f(\mathbf{w}, b)$ and $\nabla_b f(\mathbf{w}, b)$ are the values compuated from equation (3).

The *convergence criteria* for the above algorithm is $\Delta_{\%cost} < \epsilon$, where

$$\Delta_{\%cost} = \frac{|f_{k-1}(\mathbf{w}, b) - f_k(\mathbf{w}, b)| \times 100}{f_{k-1}(\mathbf{w}, b)}. \tag{4}$$

Here,

$f_k(\mathbf{w}, b)$ is the value of equation (2) at kth iteration,

$\Delta_{\%cost}$ is computed at the end of each iteration of the while loop.

Initialize $\mathbf{w} = 0$, $b = 0$ and compute $f_0(\mathbf{w}, b)$ with these values.

For this method, use $\eta = 0.0000003$, $\epsilon = 0.25$.

- **Stochastic gradient descent:** Go through the dataset and update the parameters, one training sample at a time, as follows:

  Randomly shuffle the training data

  $i = 1$, $k = 0$

  while convergence criteria not reached **do**

    for $j = 1, \ldots, d$ **do**

      Update $w^{(j)} \leftarrow w^{(j)} - \eta \nabla_{w^{(j)}} f_i(\mathbf{w}, b)$

    end for

    Update $b \leftarrow b - \eta \nabla_b f_i(\mathbf{w}, b)$

    Update $i \leftarrow (i \bmod n) + 1$

    Update $k \leftarrow k + 1$

  end while

  where,

  n is the number of samples in the training data,

  d is the dimension of $\mathbf{w}$,

  η is the learning rate and

  $\nabla_{w^{(j)}} f_i(\mathbf{w}, b)$ is defined for a single training sample as follows:

$$\nabla_{w^{(j)}} f_i(\mathbf{w}, b) = \frac{\partial f_i(\mathbf{w}, b)}{\partial w^{(j)}} = w_j + C \frac{\partial L(\mathbf{w}, b; x_i, y_i)}{\partial w^{(j)}}$$

  $\nabla_b f_i(\mathbf{w}, b)$ is similar.

The *convergence criteria* here is $\Delta_{cost}^{(k)} < \epsilon$, where

$$\Delta_{cost}^{(k)} = 0.5\Delta_{cost}^{(k-1)} + 0.5\Delta_{\%cost},$$

where,

k is the iteration number, and

$\Delta_{\%cost}$ is the same as above (from equation 4).

Calcuate Δ_{cost}, $\Delta_{\%cost}$ at the end of each iteration of the while loop.

Initialize $\Delta_{cost} = 0$, $\mathbf{w} = 0$, $b = 0$ and compute $f_0(\mathbf{w}, b)$ with these values.

For this method, use $\eta = 0.0001, \epsilon = 0.001$.

- **Mini batch gradient descent:** Go through the dataset in batches of predetermined size and update the parameters, one training sample at a time, as follows:

  Randomly shuffle the training data

  $l = 1$, $k = 0$

  while convergence criteria not reached **do**

    for $j = 1, \ldots, d$ **do**

      Update $w^{(j)} \leftarrow w^{(j)} - \eta \nabla_{w^{(j)}} f_l(\mathbf{w}, b)$

    end for

    Update $b \leftarrow b - \eta \nabla_b f_l(\mathbf{w}, b)$

    Update $l \leftarrow (l + 1) \bmod ((n + batch\_size - 1)/batch\_size)$

    Update $k \leftarrow k + 1$

  end while

  where,

  n is the number of samples in the training data,

  d is the dimension of $\mathbf{w}$,

  η is the learning rate and

  batch_size is the number of training samples considered in each batch, and $\nabla_{w^{(j)}} f_l(\mathbf{w}, b)$ is defined for a batch of training sample as follows:

$$\nabla_{w^{(j)}} f_l(\mathbf{w}, b) = \frac{\partial f_l(\mathbf{w}, b)}{\partial w^{(j)}} = w_j + C \sum_{i=l*batch\_size+1}^{\min\{n,(l+1)*batch\_size\}} \frac{\partial L(\mathbf{w}, b; x_i, y_i)}{\partial w^{(j)}}$$

  The *convergence criteria* here is $\Delta_{cost}^{(k)} < \epsilon$, where

$$\Delta_{cost}^{(k)} = 0.5\Delta_{cost}^{(k-1)} + 0.5\Delta_{\%cost},$$

  where,

  k is the iteration number, and

  $\Delta_{\%cost}$ is the same as above (equation 4).

Calcuate Δ_{cost}, $\Delta_{\%cost}$ at the end of each iteration of the while loop.
Initialize $\Delta_{cost} = 0$, $\mathbf{w} = 0$, $b = 0$ and compute $f_0(\mathbf{w}, b)$ with these values.
For this method, use $\eta = 0.00001, \epsilon = 0.01$, batch_size = 20.

(a) (10 points) Implement the SVM algorithm for all the the above mentioned gradient descent techniques. Use $C = 100$ for all the techniques. For all other parameters, use the values specified in the description of the technique. **Note:** update w in iteration $k+1$ using the values computed in iteration k. Do not update using values computed in the current iteration!

(b) (10 points) Run you implementation on the dataset, which contains:

1. `features.txt:` Each line contains features (comma-separated values) for a single datapoint. It have 6414 datapoints (rows) and 122 features (columns).
2. `target.txt:` Each line contains the response variable (y = -1 or 1) for the corresponding row in `features.txt`.

(c) (5 points) Plot the value of the cost function $f(\mathbf{w}, b)$ after each iteration vs. the number of iteration (k). Report the total time taken for convergence by each of the gradient descent techniques. What do you infer from the plots and the time for convergence?
The diagram should have graphs from all the three techniques on the same plot.