CSC411H1 L0101, Fall 2018
Assignment 3

Name: JingXi, Hao
Student Number: 1000654188

Due Date: 12 October, 2018 11:59pm

1. **Robust Regression.** One problem with linear regression using squared error loss is that it can be sensitive to outliers. Another loss function we could use is the Huber loss, parameterized by a hyperparameter δ:

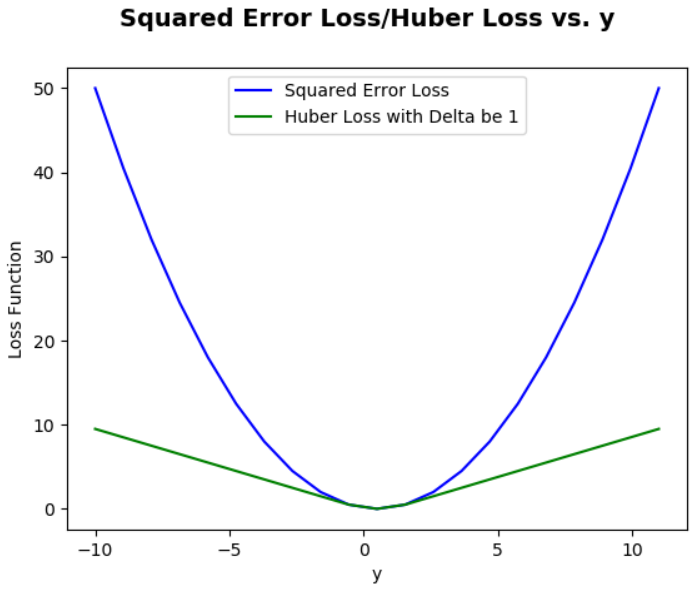
$$L_\delta(y,t) = H_\delta(y-t)$$
$$H_\delta(a) = \begin{cases} \frac{1}{2}a^2 & if\ |a| \leq \delta \\ \delta(|a| - \frac{1}{2}\delta) & if\ |a| > \delta \end{cases}$$

   (a) Sketch the Huber loss $L_\delta(y,t)$ and squared error loss $L_{SE}(y,t) = \frac{1}{2}(y-t)^2$ for $t = 0$, either by hand or using a plotting library. Based on your sketch, why would you expect the Huber loss to be more robust to outliers?

   Solution:
   For this question, we are given that $t = 0$ and we pick $\delta = 1$. Thus, we obtain that

$$H_\delta(a) = \begin{cases} \frac{1}{2}a^2 & if\ |a| \leq 1 \\ |a| - \frac{1}{2} & if\ |a| > 1 \end{cases}$$

   . Then, we sketch this piece-wise function of Huber loss, $L_\delta(y-t) = H_\delta(y-t) = H_\delta(a)$, and the squared error loss, $L_{SE}(y,t) = \frac{1}{2}(y-t)^2$ in the same plot. We show the graph below.



Based on my sketch, it is obvious that the squared error loss and the Huber loss produce the same value when $|a| = |y - t| = |y| \leq 1$ given $t = 0$. However, when $|a| = |y - t| = |y| > 1$ given $t = 0$, Huber loss is linear and squared error loss follows parabola. Therefore, the value for squared error loss is greater than the value for Huber loss when they have the same input, $|a| = |y - t| = |y| > 1$ given $t = 0$. Also, the values for squared error loss increases faster than the values for Huber loss do. For the outliers, $|a| = |y - t| = |y|$ given $t = 0$ generates larger absolute value of difference between y and t. Hence, the squared error loss potentially to produce a larger value than Huber loss based on the sketch. Therefore, for general case, we can also conclude that for $|y - t| \leq \delta$, the value for squared error loss equals to the value for Huber loss, $L_\delta(y,t) = H_\delta(y - t) = \frac{1}{2}(y - t)^2 = L_{SE}(y,t)$. In addition, for $|y - t| > \delta$, we have that squared error loss is going to produce a larger value than it produced by Huber loss since Huber loss is linear and squared error loss is parabola even though Huber loss gets closer and closer to squared error loss when δ gets bigger. For the outliers, $|y - t|$ is going to produce a larger input value, then squared error loss produces larger value than it computed by Huber loss. Thus, squared error loss is more sensitive than Huber loss to residuals, $|y - t|$, which means that Huber loss is more robust to outliers.

(b) Just as with linear regression, assume a linear model:

$$y = \mathbf{w}^T\mathbf{x} + b$$

Give formulas for the partial derivatives $\partial L_\delta/\partial \mathbf{w}$ and $\partial L_\delta/\partial b$. (We recommend you find a formula for the derivative $H'_\delta(a)$, and then give your answers in terms of $H'_\delta(y - t)$.)

Solution:

- Firstly, we find a formula for the derivative $H'_\delta(a)$, where

$$H_\delta(a) = \begin{cases} \frac{1}{2}a^2 & if\ |a| \leq \delta \\ \delta(|a| - \frac{1}{2}\delta) & if\ |a| > \delta \end{cases}$$

Therefore, we have that

$$H'_\delta(a) = \begin{cases} a & if\ |a| \le \delta \\ \frac{\delta a}{|a|} & if\ |a| > \delta \end{cases} \qquad (1)$$

Then, we are able to find $H'_\delta(y-t)$ by substituting $a = y - t$ into formula (1), which is

$$H'_\delta(y-t) = \begin{cases} y-t & if\ |y-t| \le \delta \\ \frac{\delta(y-t)}{|y-t|} & if\ |y-t| > \delta \end{cases} = \frac{\partial L_\delta}{\partial y} \qquad (2)$$

- Then, we compute $\partial L_\delta / \partial \mathbf{w}$. For $|y-t| \le \delta$,
  $\partial L_\delta / \partial \mathbf{w} = \frac{\partial L_\delta}{\partial y} \frac{\partial y}{\partial \mathbf{w}}$
  $= (y-t)\mathbf{x}$ # $y = \mathbf{w}^T\mathbf{x} + b = \sum_j w_j x_j + b \Longrightarrow \partial y / \partial w_j = x_j \Longrightarrow \partial y / \partial \mathbf{w} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix} = \mathbf{x}$ and also by formula (2)

  For $|y-t| > \delta$, we have that
  $\partial L_\delta / \partial w = \frac{\partial L_\delta}{\partial y} \frac{\partial y}{\partial \mathbf{w}}$
  $= (\frac{\delta(y-t)}{|y-t|})\mathbf{x}$ # $y = \mathbf{w}^T\mathbf{x} + b = \sum_j w_j x_j + b \Longrightarrow \partial y / \partial w_j = x_j \Longrightarrow \partial y / \partial \mathbf{w} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix} = \mathbf{x}$ and also by formula (2)

  Hence, the formula for the partial derivative $\partial L_\delta / \partial w$ is defined as following,

$$\partial L_\delta / \partial \mathbf{w} = \begin{cases} (y-t)\mathbf{x} & if\ |y-t| \le \delta \\ \frac{\delta(y-t)}{|y-t|}\mathbf{x} & if\ |y-t| > \delta \end{cases} = H'_\delta(y-t)\mathbf{x}$$

- Then, We compute $\partial L_\delta / \partial b$. For $|y-t| \le \delta$,
  $\partial L_\delta / \partial b = \frac{\partial L_\delta}{\partial y} \frac{\partial y}{\partial b}$
  $= (y-t)(1)$ # $y = \mathbf{w}^T\mathbf{x} + b = \sum_j w_j x_j + b \Longrightarrow \partial y / \partial b = 1$ and also by formula (2)
  $= y - t$

  For $|y-t| > \delta$, we have that
  $\partial L_\delta / \partial b = \frac{\partial L_\delta}{\partial y} \frac{\partial y}{\partial b}$
  $= (\frac{\delta(y-t)}{|y-t|})(1)$ # $y = \mathbf{w}^T\mathbf{x} + b = \sum_j w_j x_j + b \Longrightarrow \partial y / \partial b = 1$ and also by formula (2)
  $= \frac{\delta(y-t)}{|y-t|}$

  Hence, the formula for the partial derivative $\partial L_\delta / \partial b$ is defined as following,

$$\partial L_\delta / \partial b = \begin{cases} y-t & if\ |y-t| \le \delta \\ \frac{\delta(y-t)}{|y-t|} & if\ |y-t| > \delta \end{cases} = H'_\delta(y-t)$$

(c) Write Python code to perform (full batch mode) gradient descent on this model. Assume the training dataset is given as a design matrix $\mathbf{X}$ and target vector y. Initialize $\mathbf{w}$ and b to all zeros. Your code should be vectorized, i.e. you should not have a for loop over training examples or input dimensions. You may find the function np.where helpful.

Submit your code as *q1.py*.

Solution:

Please see the code implementation of gradient descent, *compute_gradient_descent*(), for this question in **q1.py**.

2. **Locally Weighted Regression.**

(a) Given $\{(\mathbf{x}^{(1)}, y^{(1)}), \ldots, (\mathbf{x}^{(N)}, y^{(N)})\}$ and positive weights $a^{(1)}, \ldots, a^{(N)}$ show that the solution to the weighted least square problem

$$w^* = arg\ min \frac{1}{2} \Sigma_{i=1}^{N} a^{(i)} (y^{(i)} - w^T \mathbf{x}^{(i)})^2 + \frac{\lambda}{2} ||w||^2 \tag{1}$$

is given by the formula

$$w^* = (\mathbf{X}^T \mathbf{A} \mathbf{X} + \lambda \mathbf{I})^{-1} \mathbf{X}^T \mathbf{A} y \tag{2}$$

where $\mathbf{X}$ is the design matrix (define in class) and $\mathbf{A}$ is a diagonal matrix $\mathbf{A}_{ii} = a^{(i)}$.

Solution:

Based on the formula (1) given for computing w^*, we can see that the matrix version of $\Sigma_{i=1}^{N} a^{(i)} (y^{(i)} - w^T \mathbf{x}^{(i)})^2 = (y - \mathbf{X}w)^T \mathbf{A} (y - \mathbf{X}w)$ and the matrix version of $\frac{\lambda}{2} ||w||^2 = \frac{\lambda}{2} w^T w$. Therefore, the matrix version of total loss function should be

$$\begin{aligned} L(w) &= \tfrac{1}{2} (y - \mathbf{X}w)^T \mathbf{A} (y - \mathbf{X}w) + \tfrac{\lambda}{2} w^T w \\ &= \tfrac{1}{2} (y^T \mathbf{A} y - y^T \mathbf{A} \mathbf{X} w - w^T \mathbf{X}^T \mathbf{A} y + w^T \mathbf{X}^T \mathbf{A} \mathbf{X} w) + \tfrac{\lambda}{2} w^T w \end{aligned}$$

Then, we compute the derivative of the total loss function with respect to w and we obtain that

$$\begin{aligned} \nabla L(w) &= \tfrac{1}{2} (-2(\mathbf{X^T A} y) + 2(\mathbf{X^T A X} w)) + \tfrac{\lambda}{2} (2\mathbf{I} w) \qquad \text{where } \mathbf{I} \text{ is an identity matrix} \\ &= -\mathbf{X^T A} y + \mathbf{X^T A X} w + \lambda \mathbf{I} w \end{aligned}$$

Then, let $\nabla L(w) = 0$ since we want to find value for w, call it w^*, that produces the minimum output value. Thus, we have that

$$\begin{aligned} -\mathbf{X^T A} y + \mathbf{X^T A X} w^* + \lambda \mathbf{I} w^* &= 0 \\ \mathbf{X^T A X} w^* + \lambda \mathbf{I} w^* &= \mathbf{X^T A} y \\ (\mathbf{X^T A X} + \lambda \mathbf{I}) w^* &= \mathbf{X^T A} y \\ w^* &= (\mathbf{X^T A X} + \lambda \mathbf{I})^{-1} \mathbf{X^T A} y \end{aligned}$$

Hence, we have shown that the solution to the weighted least square solution problem is the same as the formula $w^* = (\mathbf{X^T A X} + \lambda \mathbf{I})^{-1} \mathbf{X^T A} y$.

(b) Locally re-weighted least squares combines ideas from k-NN and linear regression. For each new test example x we compute distance-based weights for each training example $a^{(i)} = \frac{exp(-||\mathbf{x}-\mathbf{x}^{(i)}||^2/2\tau^2)}{\Sigma_j exp(-||\mathbf{x}-\mathbf{x}^{(i)}||^2/2\tau^2)}$, computes $w^* = arg\ min \frac{1}{2}\Sigma_{i=1}^{N} a^{(i)}(y^{(i)} - w^T\mathbf{x}^{(i)})^2 + \frac{\lambda}{2}||w||^2$ and predicts $\hat{y} = \mathbf{x}^T w^*$. Complete the implementation of locally re-weighted least squares by providing the missing parts for **q2.py**.

Important things to notice while implementing: First, do not invert any matrix, use a linear solver (numpy.linalg.solve is one example). Second, notice that $\frac{exp(\mathbf{A}_i)}{\Sigma_j exp(\mathbf{A}_j)} = \frac{exp(\mathbf{A}_i - \mathbf{B}}{\Sigma_j exp(\mathbf{A}_j - \mathbf{B})}$ but if we use $B = max_j \mathbf{A}_j$ it is much more numerically stable as $\frac{exp(\mathbf{A}_i)}{\Sigma_j exp(\mathbf{A}_j)}$ overflows / underflows easily. *This is handled automatically in the scipy package with the scipy.misc.logsumexp* function.

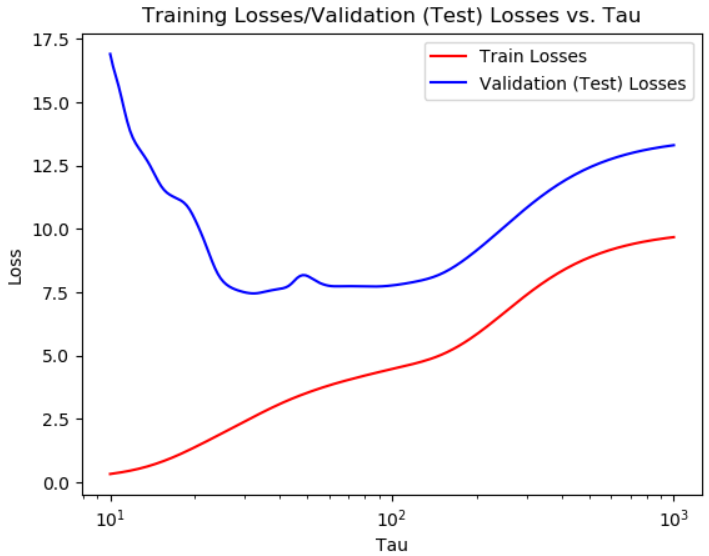
Solution:

Please see the code implementation of locally re-weighted least squares, $LRLS()$, in **q2.py**.

(c) Randomly hold out 30% of the dataset as a validation set. Compute the average loss for different values of τ in the range [10, 1000] on both the training set and the validation set. Plot the training and validation losses as a function of τ (using a log scale for τ).

Solution:

Please see code implementation of $run\_validation()$ in **q2.py**. Also, the plot of the loss values for each choice of τ is shown below.



(d) How would you expect this algorithm to behave as $\tau -> \infty$? When $\tau -> 0$? Is this what actually happened?

Solution:

As $\tau \to 0$, we can see that $-||\mathbf{x} - \mathbf{x}^{(i)}||^2/2\tau^2$ approaches to $-\infty$, which means that $exp(-||\mathbf{x} -$

$\mathbf{x}^{(i)}||^2/2\tau^2) \rightarrow 0$. Therefore, $a^{(i)}$ would approach to 0 as well based on the formula for $a^{(i)}$. Therefore, the weight, $a^{(i)}$, of the residual for each training example is potentially to approach to 0, which implies that training set is overfitted in this case. Hence, the overfitting for the training set causes the bigger error (loss) for validation set when $\tau \rightarrow 0$. As $\tau \rightarrow \infty$, $-||\mathbf{x} - \mathbf{x}^{(i)}||^2/2\tau^2$ approaches to 0, which means that $exp(-||\mathbf{x} - \mathbf{x}^{(i)}||^2/2\tau^2) \rightarrow 1$. Hence, $a^{(i)} \rightarrow \frac{1}{N}$, where N is the total number of training examples. This indicates that the weight, $a^{(i)}$, of the residual for each training example is potentially to be the same. Therefore, the total error (loss) for training set becomes flat, potentially would approach to a constant, when $\tau \rightarrow \infty$, which is the same case for the validation set loss. Based on the explanation, I would expect that this algorithm for computing training set losses would produce a value that approaches to a constant when $\tau \rightarrow \infty$ and the algorithm for calculating validation set losses would give a value that approaches to a constant when $\tau \rightarrow \infty$. In addition, as $\tau \rightarrow 0$, I would expect that the value for training loss would approach to 0 and the value for validation loss would approach to ∞ by using this algorithm. Based on the graph shown in $2(c)$, my expectation is justified, indicating that it actually happened.