sta314 HW2

shimmy Nauenberg

October 2021

1 Question 1

We want to show that every linear regression is a linear function. Therefore, given any linear function

$$f(x) = w^\top X, w \in \mathbb{R}^d$$

for all $x, y \in \mathbb{R}^D, f(x+y) = f(x)+f(y)$ and for all $x \in \mathbb{R}^d, a \in \mathbb{R} f(ax) = af(x)$. We will begin by showing the first fact. Fix $x, y \in \mathbb{R}^D$, we will show that $f(x+y) = f(x) + f(y)$

$$\begin{aligned} f(x+y) &= w^\top (x+y), w \in \mathbb{R}^D \\ &= \sum w_j(x_j + y_j) \text{per the definition provided} \\ &= \sum (w_j x_j + w_j y_j) \text{ by distributing } w_j \\ &= \sum w_j x_j + \sum w_j y_j \text{ per rules of splitting up sums} \\ &= f(x) + f(y) \end{aligned}$$

Now we will fix $x \in \mathbb{R}^D$ and $a \in \mathbb{R}$ and show that $f(ax) = a(fx)$

$$\begin{aligned} f(ax) &= w^\top (ax) \\ &= \sum w_j(a)x_j \text{ per definition provide} \\ &= a \sum w_j x_j \text{ as a is constant we can put it in front of the sum} \\ &= af(x) \end{aligned}$$

Therefore, we have shown that both condition for a function to be linear is satisfied by any linear regression predictor $\square$

2 question 2

2.1 2a

we can notice that a function which uses the Huber loss will treat the penalty for outliers, those greater than δ away from y, as linear instead of quadratic. As quadratic functions have a greater gradient than linear functions, at least for larger values, then the penalty applied for these outliers won't be as strong and hence won't influence the model as much.

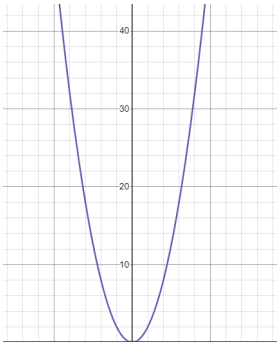

Figure 1: graph of $.5x^2$

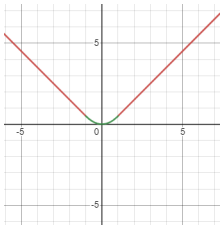

Figure 2: graph of huber loss when $\delta = 1$

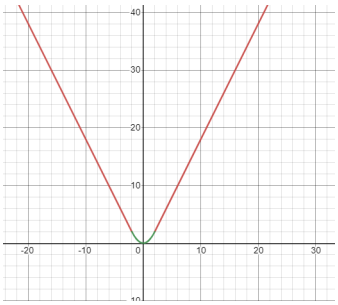

Figure 3: graph of huber loss when $\delta = 2$

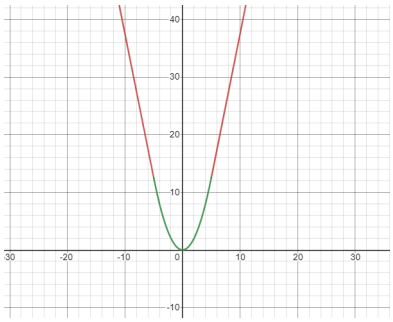

Figure 4: graph of huber loss when $\delta = 5$

2.2 2b

Let us begin by finding the derivative of $H_\delta(\alpha)$

$$\partial_\alpha H_\delta(\alpha) = \alpha; |\alpha| \leq \delta \tag{1}$$

$$\partial_\alpha H_\delta(\alpha) = \delta\alpha\frac{1}{|\alpha|}; |\alpha| > \delta \tag{2}$$

As an important note we can recognize that this function is differentiable at $\alpha = \delta$ as (1) would equal (2).
Now let us notice that

$$\hat{R} = \frac{1}{N}\sum H_\delta(y - t)$$

. Now we will solve for $\partial_{y^{(i)}}\hat{R}$,the partial with respect to each $y^{(i)}$ Therefore, we are able to apply the chain rule to solve for $\partial_{y^{(i)}}\hat{R}$.

$$\partial_{y^{(i)}}\hat{R} = \partial_{H_\delta}\hat{R} \cdot \partial_{y^{(i)}} H_\delta \tag{3}$$

$$= \frac{1}{N}H'_\delta(y^{(i)} - t^{(i)})\partial_{y^{(i)}} H_\delta \tag{4}$$

$$= \frac{1}{N}H'_\delta(y^{(i)} - t^{(i)}) * 1 \tag{5}$$

Therefore, since we have solved the derivative with respect to each $^{(i)}$ then

$$\partial_y \hat{R} = \frac{1}{N} \begin{pmatrix} H_\delta^{'}(y^{(1)} - t^{(1)}) \\ \vdots \\ H_\delta^{'}(y^{(N)} - t^{(N)}) \end{pmatrix} \tag{6}$$

$$= \frac{1}{N} H_\delta^{'}(y - t) \text{asumming } H_\delta^{'} \text{ behaves like } H_\delta \text{ and takes derivaties component wise} \tag{7}$$

Now since we are interested in also solving for in also solving for $\partial_w \hat{R}$, we can notice that $\partial_w \hat{R} = \partial_y \hat{R} \cdot \partial_w y$, where $y = X^\top w$. Therefore,

$$\partial_w \hat{R} = \frac{1}{N} H_\delta^{'}(y - t) \cdot \partial_w y \tag{8}$$

$$= \frac{1}{N} X^\top H_\delta^{'}(X^\top w - t) \tag{9}$$

Therefore our final requested answer would be

$$y = X^\top w \tag{10}$$

$$\partial_y \hat{R} = \frac{1}{N} H_\delta^{'}(y - t) \tag{11}$$

$$\partial_w \hat{R} = \frac{1}{N} X^\top H_\delta^{'}(X^\top w - t) \tag{12}$$

2.3 question 2c, question 2d

```
"""
STA314, 2021 Fall, University of Toronto
"""

import numpy as np

def robust_regression_grad(X, t, w, delta):
    """
    Compute the gradient of the average Huber (NOT squared error) loss for robus

    Parameters
    ----------
        X: numpy array
            N x (D+1) numpy array for the train inputs (with dummy variables)
        t: numpy array
            N x 1 numpy array for the train targets
        w: numpy array
            (D+1) x 1 numpy array for the weights
        delta: positive float
            parameter for huber loss.


    Returns
    -------
        dw: numpy array
            (D+1) x 1 numpy array, the gradient of the huber loss in w
    Valuable methods
    ----------
    np.where, np.abs, np.dot, np.shape, and np.sign
    """
    # ====== YOUR CODE GOES HERE (delete 'pass') ======
    y = X @ w
    # derivative of H'_delta(y-t)
    # dividiing by N
    less = (y-t)
    more =  delta * np.sign( y-t)
    # because derivative is y -t / abs(y-t) we can choose use the sign of (y-t)
    # derivative with respect to w
    dw = X.T @ np.where(
        np.abs(y-t) <= delta, less, more)/ np.shape(t)[0]

    return dw
    # ================================================
```

```
def optimization(X, t, delta, lr, num_iterations=10000):
    """
    Compute (nearly) optimal weights for robust linear regression.

    Parameters
    ----------
        X: numpy array
            N x (D+1) numpy array for the train inputs (with dummy variables)
        t: numpy array
            N x 1 numpy array for the train targets
        delta: positive float
            parameter for huber loss.
        lr: positive float
            learning rate or step-size.

    Returns
    -------
        w: numpy array
            (D+1) x 1 numpy array, (nearly) optimal weights for robust linear re
    """

    # some initialization for robust regression parameters
    w = np.zeros((X.shape[1], 1))
    for i in range(num_iterations):
        # ====== YOUR CODE GOES HERE (delete 'pass') ======
        #update the w throughout by calculating the gradient and using grad desc
        grad = robust_regression_grad(X, t, w, delta)
        #the adjustement for w.
        l_t = -lr*grad
        w += l_t
        # ================================================
    return w

def squared_error(y, t):
    """
    Compute the average squared error (NOT huber) loss:

    sum_{i=1}^N (y^i - t^i)^2 / N

    Parameters
    ----------
        y: numpy array
            N x 1 numpy array for the predictions
        t: numpy array
```

```
            N x 1 numpy array for the train targets

    Returns
    -------
        cost: float
            the average squared error loss
    """
    dif = y - t
    L = 0.5 * np.power(dif, 2)
    return L.mean()

def linear_regression_optimal_weights(X, t):
    """
    Compute the optimal weights for linear regression.

    Parameters
    ----------
        X: numpy array
            N x (D+1) numpy array for the train inputs (with dummy variables)
        t: numpy array
            N x 1 numpy array for the train targets

    Returns
    -------
        w: numpy array
            (D+1) x 1 numpy array, optimal weights for linear regression

    """
    w = np.linalg.solve(np.dot(X.T, X), np.dot(X.T, t))
    return w


def main():
    # load data
    X = np.load("hw2_X.npy")
    t = np.load("hw2_t.npy")
    t = np.expand_dims(t, 1)
    (N, D) = X.shape

    # this code adds the "dummy variable"
    ones_vector = np.ones((N, 1))
    X = np.concatenate((ones_vector, X), axis=1)

    # train, validation, test split using numpy indexing
    X_train, X_val, X_test = X[30:], X[15:30], X[:15]
    t_train, t_val, t_test = t[30:], t[15:30], t[:15]
```

```
    lr = 0.01  # learning rate

    # these are the deltas we will try for robust regression
    deltas = [0.1, 0.5, 1, 5, 10]

    # Report the validation squared error loss using standard linear regression
    w_linreg = linear_regression_optimal_weights(X_train, t_train)
    predictions = np.dot(X_val, w_linreg)
    val_loss = squared_error(predictions, t_val)
    print(f"linear regression validation loss: {val_loss}")

    for i, delta in enumerate(deltas):

        # Optimize the parameters based on the train data for robust regression.
        w = optimization(X_train, t_train, delta, lr)

        # Report the validation and training squared error loss for this delta.
        val_pred = np.dot(X_val, w)
        val_loss = squared_error(t_val, val_pred)
        train_pred = np.dot(X_train, w)
        train_loss = squared_error(t_train, train_pred)
        print(f"delta: {delta}, valid. squared error loss: {val_loss}, train squ


if __name__ == "__main__":
    main()
```

2.3.1 results from code

linear regression validation loss: 28.442028967907056
delta: 0.1, valid. squared error loss: 4.909228730093659, train squared error loss: 38.04144442529115
delta: 0.5, valid. squared error loss: 4.018972761905309, train squared error loss: 38.31116689103212
delta: 1, valid. squared error loss: 4.200164017924182, train squared error loss: 37.99343986336346
delta: 5, valid. squared error loss: 23.636105018763654, train squared error loss: 28.847567784752847
delta: 10, valid. squared error loss: 26.4204677294915, train squared error loss: 27.798297431509038

2.4 2e

When we are training our model we are obtaining a model which will underestimate the value of the outliers and hence when we apply the squared error it will have a very bad estimate of the outliers in the training set and hence will have a very bad estimate of the outlier and hence a large error when predicting with the training set. Now it will go down towards the error of the estimate of the regression model because as $\delta \to \infty$ the huber regression approaches the square error loss. However, this would not be true for the validation set. The validation set does not have the same outliers as the training set and therefore, when we are determining the delta we are hoping to determine the delta which accurately divides the data set into the points which are from the underlying data generating distribution and which are not. As the validation set would not have the outliers, there would be some delta which would accurately divide the training set to predict the validation set to a minimum which would be the delta which accurately divides the outliers and the non-outliers. For this delta, the predictor will most accurately predict the data points in the underlying distribution and hence in the validation set.

3 question 3

3.1 3a

Let us begin by assuming that there exists some linear classify H, such that it correctly classifies all three points. Now let us notice that if two points have the same prediction value according to the classifier H, then every point on a line which connects those two points must also have the same classification as the sets of points must be convex. However, let us notice that if H classifies both $x^{(1)}$ and $x^{(3)}$ as 1, then it must also classify $x^{(2)}$ as 1 because the line segment that connects these two points goes through $x^{(2)}$ as is obvious. However, $x^{(2)}$ is classified as 0. Therefore, we have arrived at a contradiction. Therefore we have shown that such an H doesn't exist that manages to classify all points according to their predicted values and hence the data set is not separable. □

3.2 3b

let us first make a table of all of our values.

x $^{(i)}$	$\psi_1(x^{(i)})$	$\psi_2(x^{(i)})$	t $^{(i)}$
-1	-1	1	1
1	1	1	0
3	3	9	1

Now let we want to find w_1, w_2 such that all observations are corectly classified. Therefore,

$$-w_1 + w_2 \geq 0 \implies w_2 \geq w_1 \tag{13}$$
$$w_1 + w_2 < 0 \implies w_1 < -w_2 \tag{14}$$
$$3w_1 + 9w_2 \geq 0 \implies w_1 \geq -3w_2 \tag{15}$$

Now let take $w_1 = -2$, $w_2 = 1$ We can nptoce that $1 \geq -2$ and therefore (13) is satisfied. we can notice that $-2 < -1$ which implies that (14) is satisfied. And we can notice that $-2 \geq -3 \times 2 = -6$ which implies that (15) is satisfied. Therefore, the weights $(w_1, w_2) = (-2, 1)$ correctly classify our dataset under the feature map $\psi(x) = (\psi_1(x), \psi_2(x))$