DSC 140A - Homework 02

Due: Wednesday, January 22

Instructions: Write your solutions to the following problems either by typing them or handwriting them on another piece of paper or on an iPad/tablet. Show your work or provide justification unless otherwise noted; submissions that don't show work might lose credit. If you write code to solve a problem, include the code by copy/pasting or as a screenshot. You may use `numpy`, `pandas`, `matplotlib` (or another plotting library), and any standard library module, but no other third-party libraries unless specified. Submit homeworks via Gradescope by 11:59 PM.

A LaTeX template is provided at `http://dsc140a.com`, next to where you found this homework. Using it is totally optional, but encouraged if you plan to go to grad school. See this video for a quick introduction to LaTeX.

Problem 1.

Let $(\vec{x}^{(1)}, y_1), \ldots, (\vec{x}^{(n)}, y_n)$ be a set of n training examples, where $\vec{x}^{(i)} \in \mathbb{R}^d$ and $y_i \in \mathbb{R}$

Recall that the mean squared error of a linear predictor is defined to be

$$R(\vec{w}) = \frac{1}{n} \sum_{i=1}^{n} (\text{Aug}(\vec{x}^{(i)}) \cdot \vec{w} - y_i)^2.$$

In lecture, we saw two equivalent expressions for the gradient of $R(\vec{w})$:

$$\frac{dR}{d\vec{w}} = \frac{2}{n} \sum_{i=1}^{n} (\text{Aug}(\vec{x}^{(i)}) \cdot \vec{w} - y_i) \, \text{Aug}(\vec{x}^{(i)}),$$

and, in matrix-vector form:

$$\frac{dR}{d\vec{w}} = \frac{2}{n} X^T (X\vec{w} - \vec{y}),$$

where X is the $n \times (d+1)$ *design matrix* whose ith row is $\text{Aug}(\vec{x}^{(i)})$, and $\vec{y}$ is the $n \times 1$ vector whose ith entry is y_i.

Show that these two expressions are equivalent.

Solution: First, focus on $X\vec{w}$. Since the rows of X are $\text{Aug}(\vec{x}^{(i)})$, we have

$$X\vec{w} = \begin{pmatrix} \text{Aug}(\vec{x}^{(1)}) \cdot \vec{w} \\ \vdots \\ \text{Aug}(\vec{x}^{(n)}) \cdot \vec{w} \end{pmatrix}.$$

Therefore:

$$X\vec{w} - \vec{y} = \begin{pmatrix} \text{Aug}(\vec{x}^{(1)}) \cdot \vec{w} - y_1 \\ \vdots \\ \text{Aug}(\vec{x}^{(n)}) \cdot \vec{w} - y_n \end{pmatrix}$$

This means that:

$$X^T(X\vec{w} - \vec{y}) = X^T \begin{pmatrix} \text{Aug}(\vec{x}^{(1)}) \cdot \vec{w} - y_1 \\ \vdots \\ \text{Aug}(\vec{x}^{(n)}) \cdot \vec{w} - y_n \end{pmatrix}$$

$$= \begin{pmatrix} \text{Aug}(\vec{x}^{(1)}) & \cdots & \text{Aug}(\vec{x}^{(n)}) \\ \downarrow & \ddots & \downarrow \end{pmatrix} \begin{pmatrix} \text{Aug}(\vec{x}^{(1)}) \cdot \vec{w} - y_1 \\ \vdots \\ \text{Aug}(\vec{x}^{(n)}) \cdot \vec{w} - y_n \end{pmatrix}$$

Here we use a fact from the first discussion: the product $A\vec{u}$ is a vector equal to u_1 times the first column of A, plus u_2 times the second column of A, and so on. Therefore:

$$\begin{pmatrix} \text{Aug}(\vec{x}^{(1)}) & \cdots & \text{Aug}(\vec{x}^{(n)}) \\ \downarrow & \ddots & \downarrow \end{pmatrix} \begin{pmatrix} \text{Aug}(\vec{x}^{(1)}) \cdot \vec{w} - y_1 \\ \vdots \\ \text{Aug}(\vec{x}^{(n)}) \cdot \vec{w} - y_n \end{pmatrix} = \vec{x}^{(1)}(\text{Aug}(\vec{x}^{(1)}) \cdot \vec{w} - y_1) + \cdots + \vec{x}^{(n)}(\text{Aug}(\vec{x}^{(n)}) \cdot \vec{w} - y_n)$$

$$= \sum_{i=1}^{n} \text{Aug}(\vec{x}^{(i)})(\text{Aug}(\vec{x}^{(i)}) \cdot \vec{w} - y_i)$$

Therefore, $\frac{2}{n} X^T(X\vec{w} - \vec{y}) = \frac{2}{n} \sum_{i=1}^{n} \text{Aug}(\vec{x}^{(i)})(\text{Aug}(\vec{x}^{(i)}) \cdot \vec{w} - y_i)$.

Problem 2.

Let $f(\vec{z}) = e^{z_1} + e^{z_2} + e^{z_3} + (z_1 - 1)^2 + \|\vec{z}\|^2$ be a function of $\vec{z} = (z_1, z_2, z_3)^T$.

Find a minimizer of this function using gradient descent (implemented with code; don't run GD by hand!). Report:

- The initial point, $\vec{z}^{(0)}$;
- your choice of step size;
- the stopping criterion you chose and number of iterations taken to reach the criterion;
- the minimizer you found;
- the minimum value of the function.

Include your code.

Solution: First, we do a little calculus to find the gradient of our function. The $\|\vec{z}\|^2$ term might throw you off, but remember that $\|\vec{z}\|^2 = z_1^2 + z_2^2 + z_3^2$. So we have:

$$f(\vec{z}) = e^{z_1} + e^{z_2} + e^{z_3} + (z_1 - 1)^2 + z_1^2 + z_2^2 + z_3^2.$$

With this, the partial derivatives are:

$$\begin{aligned}\frac{\partial f}{\partial z_1} &= e^{z_1} + 2(z_1 - 1) + 2z_1 \\ &= e^{z_1} + 4z_1 - 2\end{aligned}$$

$$\frac{\partial f}{\partial z_2} = e^{z_2} + 2z_2$$

$$\frac{\partial f}{\partial z_3} = e^{z_3} + 2z_3$$

Next, we code this up into a `gradient` function that computes the gradient.

```
def gradient(z):
    z_1, z_2, z_3 = z
    return np.array([
        np.exp(z_1) + 4*z_1 - 2,
        np.exp(z_2) + 2*z_2,
        np.exp(z_3) + 2*z_3
    ])
```

From here, we use the `gradient_descent` function from lecture with a starting location of $(0, 0, 0)$, and a step size of 0.1. As our stopping criterion, we'll stop when the norm of the gradient is less than 10^{-6}.

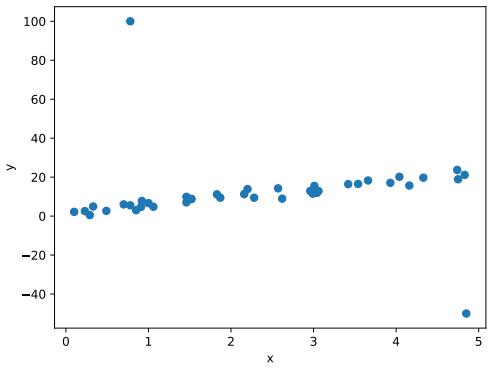
```
def gradient_descent(
        gradient, z_0, learning_rate, stop_threshold
        ):
    z = z_0
    i = 1
    while True:
        print(f"Iteration {i}:\t{z = }\t{f(z) = }")
        i += 1
        z_new = z - learning_rate * gradient(z)
        if np.linalg.norm(z_new - z) < stop_threshold:
            break
        z = z_new
    return z_new

z_opt = gradient_descent(gradient, np.array([0, 0, 0]), 0.1, 1e-6)
```

With this, we find a minimizer of approximately $(0.196, -0.352, -0.352)^T$ in 39 iterations. At this point, the function value is approximately 3.556.

Problem 3.

In a previous homework, you performed least squares regression on the data set shown below:



This data set can be downloaded again at the following link:

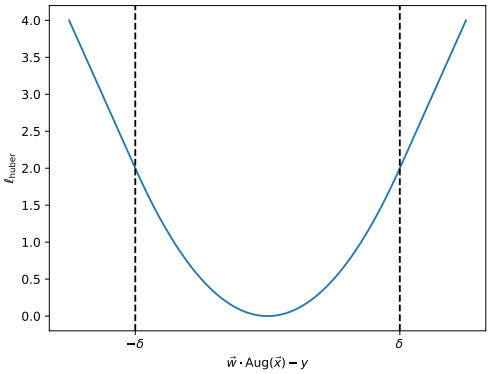
`https://f000.backblazeb2.com/file/jeldridge-data/002-regression_outlier/data.csv`

As you can see, the data contains outliers which may affect our regression. In the earlier homework, you should have found that least squares regression is not robust to these outliers.

The *Huber loss* is a loss function used in regression that is less sensitive to outliers than the square loss. It can be thought of as a "mix" between the square loss and the absolute loss. For linear prediction functions $H(\vec{x}) = \vec{w} \cdot \text{Aug}(\vec{x})$, the Huber loss is defined as:

$$\ell_{\text{huber}}(\text{Aug}(x) \cdot \vec{w}, y) = \begin{cases} \frac{1}{2}(\text{Aug}(x) \cdot \vec{w} - y)^2, & \text{if } |\,\text{Aug}(x) \cdot \vec{w} - y| \leq \delta, \\ \delta(\text{Aug}(x) \cdot \vec{w} - y) - \frac{1}{2}\delta^2, & \text{if } \text{Aug}(x) \cdot \vec{w} - y > \delta, \\ -\delta(\text{Aug}(x) \cdot \vec{w} - y) - \frac{1}{2}\delta^2, & \text{if } \text{Aug}(x) \cdot \vec{w} - y < -\delta, \end{cases}$$

A plot of the Huber loss is shown below.



Note that, despite being a piecewise function, the Huber loss is differentiable (unlike the absolute loss). However, there is no closed-form solution for the minimizer of the empirical risk with respect to the Huber loss, so if we want to train a regression model using this loss we need to use an iterative optimization algorithm, like gradient descent.

a) Write a function to compute the risk of a linear prediction function $w_0 + w_1 x$ with respect to the Huber loss (with $\delta = 1$) for the data given above. Use your function to compute the risk for $\vec{w} = (20, -1)$. Report the risk, and include your code.

Solution:

```
def huber_loss(w, x, y, delta=1):
    """Computes the Huber loss."""
```

```
    z = w @ x - y
    if np.abs(z) <= delta:
        return 0.5 * z**2
    elif z > delta:
        return delta * (z - 0.5 * delta)
    else:  # np.abs(z) < -delta
        return -delta * (z + 0.5 * delta)


def huber_risk(w):
    """Computes the risk w.r.t. the Huber loss on the data set."""
    return np.mean([huber_loss(w, X[i], y[i]) for i in range(len(X))])
```

By running `huber_risk([20, -1])`, we find that the risk of the linear prediction function $20 - x$ with respect to the Huber loss is approximately 11.085.

b) The gradient of the Huber loss with respect to $\vec{w}$ is also a piecewise function. Compute this gradient.

Note that since the Huber loss is differentiable, the gradient can be computed by separately computing the gradient within each piece of the piecewise function. You may use any of the matrix-vector calculus rules we've seen in class to help you compute the gradient. For example, you may use the fact that $\frac{d}{d\vec{w}} \vec{w} \cdot \text{Aug}(\vec{x}) = \text{Aug}(\vec{x})$.

Solution: In the first case, where $|\text{Aug}(x) \cdot \vec{w} - y| \leq \delta$, the Huber loss is equal to $\frac{1}{2}(\text{Aug}(x) \cdot \vec{w} - y)^2$. Differentiating using the chain rule, we have:

$$\begin{aligned} \frac{d}{d\vec{w}} \frac{1}{2}(\text{Aug}(x) \cdot \vec{w} - y)^2 &= (\text{Aug}(x) \cdot \vec{w} - y) \frac{d}{d\vec{w}} (\text{Aug}(x) \cdot \vec{w} - y) \\ &= (\text{Aug}(x) \cdot \vec{w} - y) \, \text{Aug}(x) \end{aligned}$$

We recognize this as being the same as the gradient of the square loss (which it should be, since the Huber loss is equal to the square loss when $|\text{Aug}(x) \cdot \vec{w} - y| \leq \delta$).

For the second case, where $\text{Aug}(x) \cdot \vec{w} - y > \delta$, the Huber loss is $\delta(\text{Aug}(x) \cdot \vec{w} - y) - \frac{1}{2}\delta^2$. Differentiating this, we have:

$$\frac{d}{d\vec{w}} \delta(\text{Aug}(x) \cdot \vec{w} - y) - \frac{1}{2}\delta^2 = \delta \, \text{Aug}(x)$$

Finally, for the third case, where $\text{Aug}(x) \cdot \vec{w} - y < -\delta$, the Huber loss is $-\delta(\text{Aug}(x) \cdot \vec{w} - y) - \frac{1}{2}\delta^2$. Differentiating this, we have:

$$\frac{d}{d\vec{w}} - \delta(\text{Aug}(x) \cdot \vec{w} - y) - \frac{1}{2}\delta^2 = -\delta \, \text{Aug}(x)$$

So, putting it all together, the gradient of the Huber loss with respect to $\vec{w}$ is:

$$\frac{d}{d\vec{w}} \ell_{\text{huber}}(\text{Aug}(x) \cdot \vec{w}, y) = \begin{cases} (\text{Aug}(x) \cdot \vec{w} - y) \, \text{Aug}(x), & \text{if } |\text{Aug}(x) \cdot \vec{w} - y| \leq \delta, \\ \delta \, \text{Aug}(x), & \text{if } \text{Aug}(x) \cdot \vec{w} - y > \delta, \\ -\delta \, \text{Aug}(x), & \text{if } \text{Aug}(x) \cdot \vec{w} - y < -\delta, \end{cases}$$

c) Write a function that computes the gradient of the empirical risk with respect to the Huber loss with $\delta = 1$ for a linear prediction function $w_0 + w_1 x$. Use your function to compute the gradient for

$\vec{w} = (20, -1)$. Show your code.

Solution:

```
def gradient_of_huber_loss(w, x, y, delta=1):
    """Computes the gradient of the Huber loss."""
    z = w @ x - y
    if np.abs(z) <= delta:
        return z * x
    elif z > delta:
        return delta * x
    else:  # np.abs(z) < -delta
        return -delta * x


def gradient_of_empirical_risk(w):
    """Computes the gradient of the risk w.r.t. the Huber loss on the data set.

    Remember, the gradient of the empirical risk is the average of the gradients
    of the individual losses.

    """
    return np.mean(
        [gradient_of_huber_loss(w, X[i], y[i]) for i in range(len(X))], axis=0
```

Computing the gradient at $\vec{w} = (20, -1)$ is done by evaluating:

```
gradient_of_empirical_risk([20, -1])
```

This gives us the gradient $\frac{dR}{d\vec{w}}(20, -1) = (0.532, 0.477)^T$.

d) Implement and run gradient descent to minimize the empirical risk with respect to the Huber loss (using $\delta = 1$) on the data set above. In addition to your code, include:

- The optimal solution found by gradient descent;
- A plot of the empirical risk of $\vec{w}^{(t)}$ at each iteration. In other words, include a plot whose horizontal axis is iteration number, t, and whose vertical axis measures the empirical risk of $\vec{w}^{(t)}$.
- A plot of the data and the linear prediction function corresponding to the optimal solution.

Solution:

```
def gradient_descent(gradient, w_0, learning_rate, stop_threshold):
    w = w_0
    i = 1
    risk_history = []
    while True:
        current_risk = huber_risk(w)
        risk_history.append(current_risk)
        print(f"Iteration {i}:\t{w = }\t{current_risk = }")
        i += 1
        w_new = w - learning_rate * gradient(w)
        if np.linalg.norm(w_new - w) < stop_threshold:
            break
```

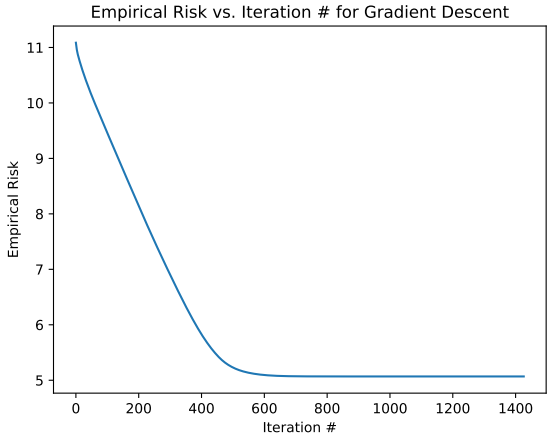
```
        w = w_new
    return w_new, risk_history
```

We run gradient descent with:

```
w_gd, gd_risk_history = gradient_descent(
    gradient_of_empirical_risk, np.array([20, -1]), 0.1, 1e-6
)
```

We find a solution of $(2.287, 3.887)^T$ with an empirical risk of 5.068 in about 1,400 iterations.

The plot of risk at each iteration is shown below:



e) Implement and run *stochastic* gradient descent with a batch size of 8 to minimize the empirical risk with respect to the Huber loss (using $\delta = 1$) on the data set above.

Include:

- The optimal solution found by SGD;
- A plot of the empirical risk of $\vec{w}^{(t)}$ at each iteration. In other words, include a plot whose horizontal axis is iteration number, t, and whose vertical axis measures the empirical risk of $\vec{w}^{(t)}$.

Note that you should plot the risk with respect to the whole data set, and not the risk with respect to the random batch (the latter will be very noisy, and it's the former that we're trying to optimize).

Solution: There's a trick to implementing stochastic gradient descent: we just implement a `stochastic_gradient` function that computes the gradient for a random batch, and pass this function to `gradient_descent` in place of the usual `gradient` function.

Here is what that looks like:

```
def stochastic_gradient(w, batch_size=8):
    """Computes a stochastic gradient from `batch_size` random examples."""
    # select a random batch of data points
    batch = np.random.choice(len(X), batch_size, replace=True)
    X_batch = X[batch]
    y_batch = y[batch]

    return np.mean(
        [
            gradient_of_huber_loss(w, X_batch[i], y_batch[i])
            for i in range(len(X_batch))
```

```
        ],
        axis=0,
    )
```

We run stochastic gradient descent with:

```
w_sgd, sgd_risk_history = gradient_descent(
    stochastic_gradient, np.array([20, -1]), 0.001, 1e-6
)
```

We find a solution of $(2.319, 3.871)^T$ with an empirical risk of 5.069 in 60,000 iterations.

The plot of risk at each iteration is shown below: