Assignment 5 Solutions: Gradient Descent - Linear Regression

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import sklearn.datasets
import sklearn.model_selection
```

In [2]:
```
# Set up data
diabetes_X, diabetes_y = sklearn.datasets.load_diabetes(return_X_y = True)
# Split into train and test sets
split = sklearn.model_selection.train_test_split(diabetes_X, diabetes_y)
diabetes_X_train, diabetes_X_test, diabetes_y_train, diabetes_y_test = split
```

1. Loss Functions

In this exercise we'll be considering a simple linear model:

$$y \approx \theta x$$

The hypothesis for the model is written as

$$h(\theta) = \theta x$$

a. Fill in the following methods for the loss functions and their derivatives.

In [3]:
```
def squared_loss(X, theta, y):
    """
    Returns the squared loss

    Input:
    X: n length vector - n datapoints
    theta: scalar
    y: n length vector

    Output:
    loss: scalar
    """
    # TODO:
    loss = None
    loss = np.sum(np.square(theta*X-y))
    return loss
```

In [4]:
```
def squared_deriv(X, theta, y):
    """
    Returns the gradient wrt theta of the squared loss

    Input:
    X: n length vector - n datapoints
    theta: scalar
    y: n length vector

    Output:
    grad: scalar
    """
    # TODO:
    grad = None
    grad = np.dot(2*X, (theta*X-y))
    return grad
```

In [5]:
```
def abs_loss(X, theta, y):
    """
    Returns the absolute value loss

    Input:
    X: n length vector - n datapoints
    theta: scalar
    y: n length vector

    Output:
    loss: scalar
    """
    # TODO:
    loss = None
    loss = np.sum(np.abs(theta*X-y))
    return loss
```

In [6]:
```
def abs_deriv(X, theta, y):
    """
    Returns the gradient wrt theta of the absolute loss

    Input:
    X: n length vector - n datapoints
    theta: scalar
    y: n length vector

    Output:
    grad: scalar
    """
    # TODO:
    grad = None
    grad = np.dot(X, np.sign(theta*X-y))
    return grad
```

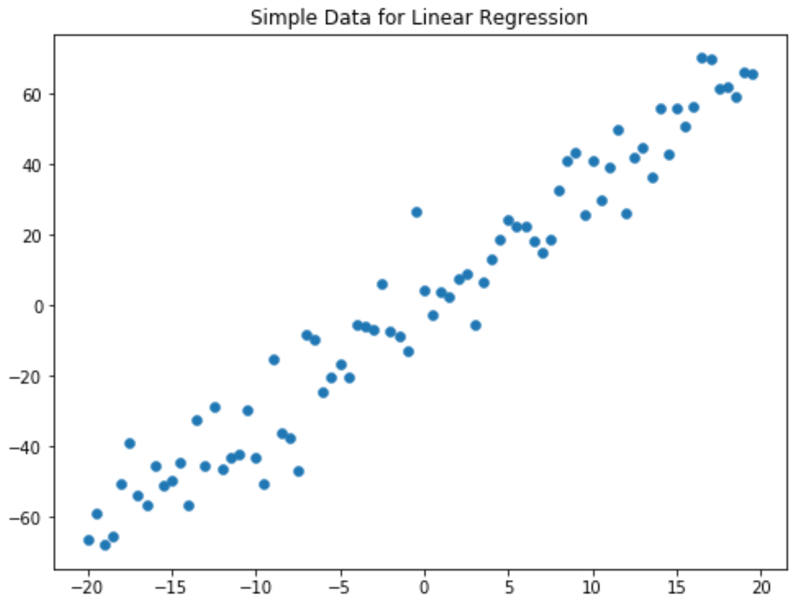
b. Plot the loss and the gradient for the provided data

In other words, compute an array of losses + gradients with pos_theta (find loss and gradient for each possible theta).

In [7]:
```
# Data you'll use with the above methods
simple_x = np.arange(-20,20,0.5)

# Yields a float between 3 and 7
true_theta = 4*np.random.random_sample()+3
# Add noise and scale y
simple_y = true_theta*simple_x + np.random.normal(scale = 10, size=simple_x.shap
e)

plt.figure(figsize = (8,6))
plt.title("Simple Data for Linear Regression")
plt.scatter(simple_x, simple_y, linewidths=0.5)
plt.show()
```



In [8]:
```
# Possible theta values (to iterate through)
pos_theta = np.arange(0, 10, 0.01)
```

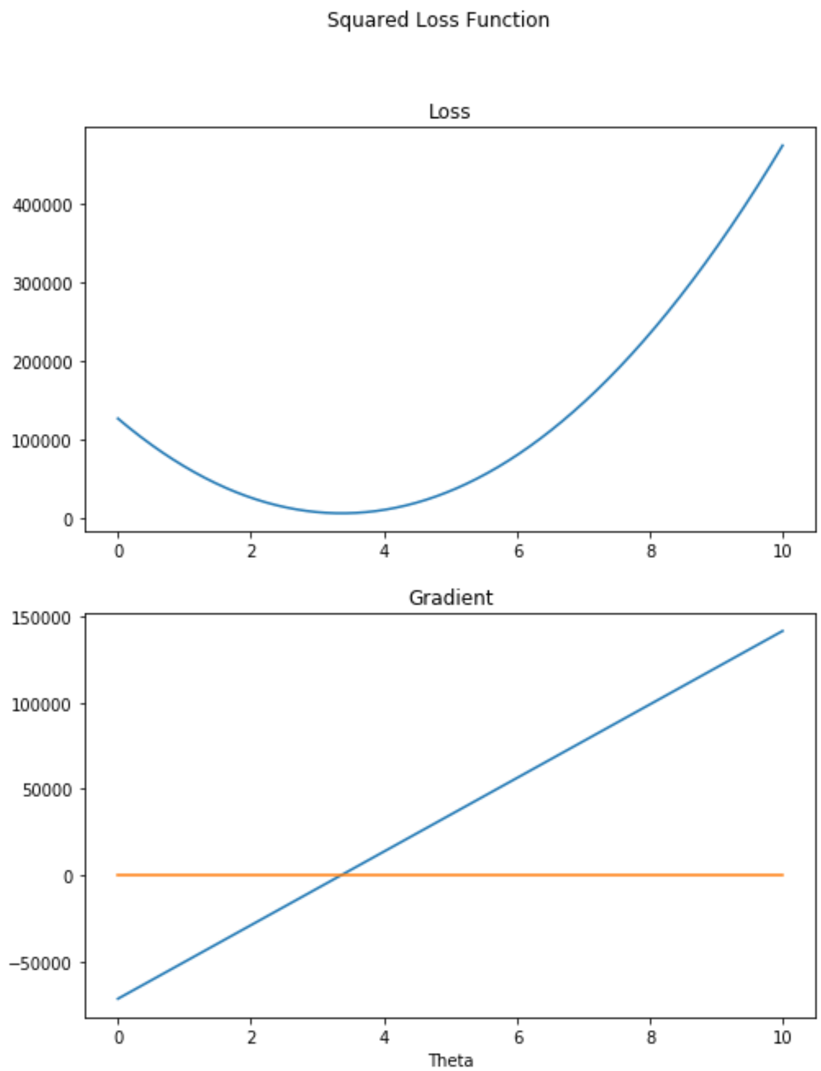
In [9]:
```
# Plot squared loss and gradient
plt.figure(figsize=(8,10))
plt.suptitle("Squared Loss Function")
plt.subplots_adjust(hspace=0.2)
plt.subplot(2,1,1)

# TODO: Find and plot loss
plt.plot(pos_theta, [squared_loss(simple_x, theta, simple_y) for theta in pos_th
eta])
plt.title("Loss")

plt.subplot(2,1,2)

# TODO: Find and plot gradient
plt.plot(pos_theta, [squared_deriv(simple_x, theta, simple_y) for theta in pos_t
heta])
plt.plot(pos_theta, np.zeros_like(pos_theta))
plt.title("Gradient")
plt.xlabel("Theta")

plt.show()
```



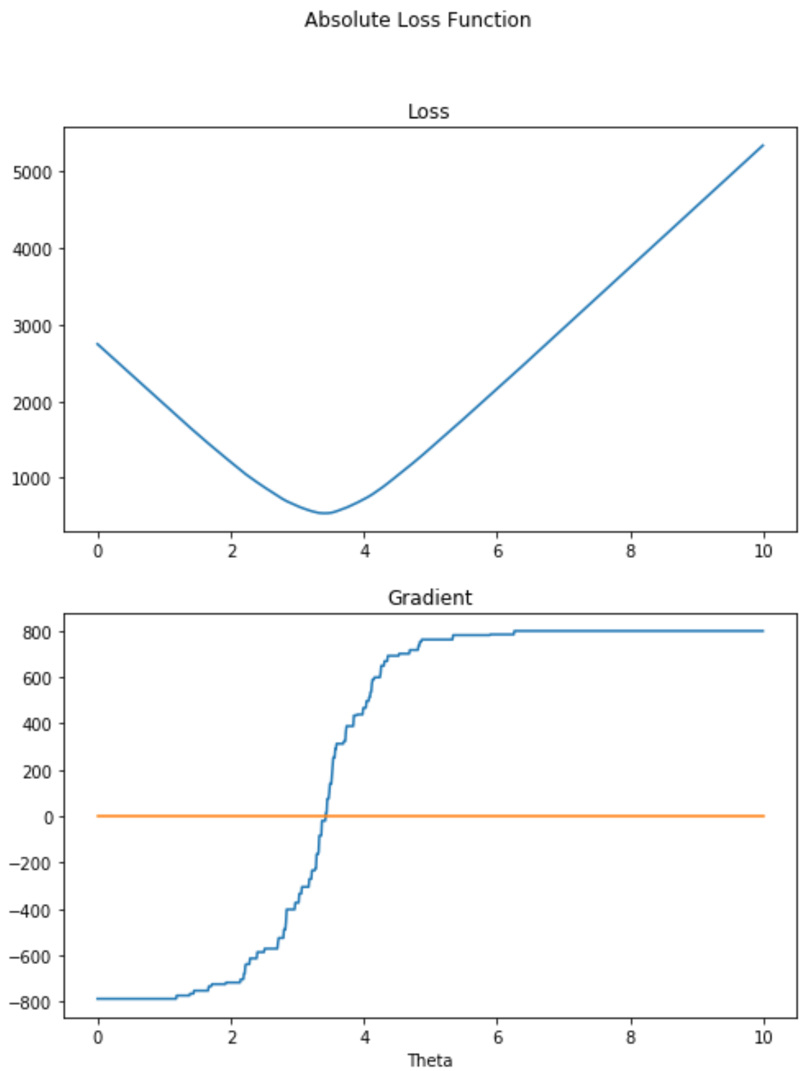
In [10]:
```
# Plot absolute loss and gradient
plt.figure(figsize=(8,10))
plt.suptitle("Absolute Loss Function")
plt.subplots_adjust(hspace=0.2)
plt.subplot(2,1,1)

# TODO: Find and plot loss
plt.plot(pos_theta, [abs_loss(simple_x, theta, simple_y) for theta in pos_thet
a])
plt.title("Loss")

plt.subplot(2,1,2)

# TODO: Find and plot gradient
plt.plot(pos_theta, [abs_deriv(simple_x, theta, simple_y) for theta in pos_thet
a])
plt.plot(pos_theta, np.zeros_like(pos_theta))
plt.title("Gradient")
plt.xlabel("Theta")

plt.show()
```



c. Given that the gradient descent algorithm uses the first derivative to find a local minimum, which of the above loss functions is preferable for linear regression using gradient descent? Briefly explain using the above plots.

Answer:

TODO:

Use squared loss as the absolute value loss function is not differentiable at the minimum.

2. Gradient Descent Linear Regression

Here you'll implement a linear regressor using gradient descent and the diabetes dataset initialized at the top of this assignment. Using the L2-norm squared loss function, the gradient descent algorithm will follow the below given formula to update the parameters and find the optimal solution.

The model:

$$y \approx Xw$$

Hypothesis:

$$h(w) = Xw$$

Gradient Descent Update Function:

$$w_{n+1} = w_n - \alpha \nabla L(w_n)$$

Due to the relatively small size of the dataset, use all datapoints for computing the gradient (also known as batch gradient descent - compare to stochastic gradient descent, an optimization over batch).

The L2-norm squared loss for this model is written as

$$L(w) = ||Xw - y||_2^2$$

---

a. Find the gradient of the loss function with respect to w.

Answer:

TODO:

$$L(w) = ||Xw - y||_2^2$$
$$L(w) = (Xw - y)^T(Xw - y)$$
$$L(w) = w^T X^T Xw - 2w^T X^T y + y^T y$$
$$\nabla L(w) = 2X^T Xw - 2X^T y$$

b. Implement the following methods to perform linear regression using gradient descent.

In [11]:
```
def gd_linreg(X, y, alpha, loss_func, derivative_func, epsilon=0.001, max_iters=
10000):
    """
    Performs linear regression on X and y using gradient descent

    Input:
    X: n x m matrix - n datapoints, m features
    y: n length vector
    alpha: step size for gradient descent update
    loss_func: method to compute loss between two quantities
    derivative_func: method to compute gradient wrt w
    epsilon: maximum difference between the w_n+1 and w_n for convergence

    Output:
    w: m length vector - weights for each feature of a data point
    losses: array of losses at each step/iteration
    """
    # TODO:
    w = np.zeros(shape=X.shape[1])
    losses = []

    for i in range(max_iters):
        w_prev = w
        w = np.subtract(w_prev, alpha * derivative_func(X, w_prev, y))
        losses.append(loss_func(X, w, y))

        diff = np.abs(w - w_prev)
        if np.max(diff) <= epsilon:
            break

    return w, losses
```

In [12]:
```
def loss_linreg(X, w, y):
    """
    Evaluates the loss function

    Input:
    X: n x m - n datapoints, m features
    w: m length vector - weights for features in X
    y: n length vector

    Output:
    loss: scalar
    """
    #TODO:
    loss = None
    diff = np.matmul(X, w)-y
    loss = np.dot(np.transpose(diff), diff)
    return loss
```

In [13]:
```
def derivative_loss_linreg(X, w, y):
    """
    Finds the derivative of the loss function wrt w

    Input:
    X: n x m - n datapoints, m features
    w: m length vector - weights for features in X
    y: n length vector

    Output:
    gradient: length m array - gradient wrt w
    """
    #TODO:
    grad = None
    grad = np.matmul(2*np.transpose(X), np.matmul(X, w) - y)
    return grad
```

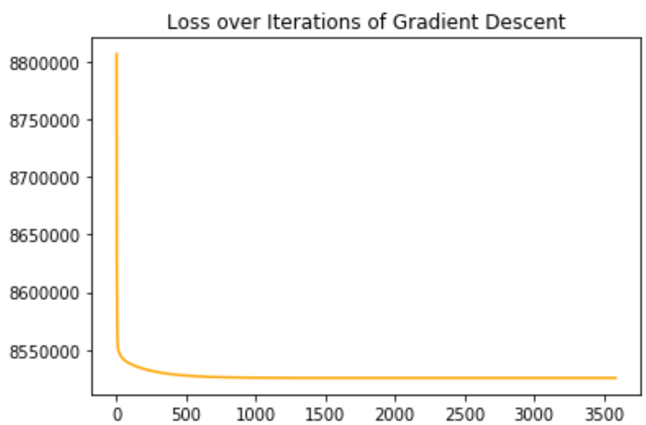
c. Run gradient descent with an appropriate step size.

In [14]:
```
# TODO: set an appropriate alpha
alpha = 0.2

diabetes_w, losses = gd_linreg(diabetes_X_train, diabetes_y_train, alpha, loss_l
inreg, derivative_loss_linreg)
```

In [15]:
```
# Plot losses (may help find a good value for alpha)
num_iter = len(losses)

plt.title("Loss over Iterations of Gradient Descent")
plt.plot(range(1, num_iter+1), losses, c = 'orange');
```



3. Evaluate your Implementation

a. Find the loss for the training set and the test set using the weights found with gradient descent.

In [16]:
```
# TODO:
gd_train_loss = losses[-1]
gd_test_loss = loss_linreg(diabetes_X_test, diabetes_w, diabetes_y_test)

print("Method: Gradient Descent")
print("Training Loss: " + str(gd_train_loss))
print("Test Loss: " + str(gd_test_loss))
```

```
Method: Gradient Descent
Training Loss: 8525634.416049298
Test Loss: 3049353.842534288
```

b. Write and implement the OLS solution for w.

The OLS solution sets the above found gradient of the loss wrt to w to 0 (from 2a) and solves for w.

Answer:

TODO:

$$\nabla L(w) = 2X^T Xw - 2X^T y = 0$$
$$w_{OLS} = (X^T X)^{-1} X^T y$$

In [17]:
```
def OLS(X, y):
    """
    Finds OLS solution to linear regression of X and y

    Input:
    X: n x m - n datapoints, m features
    y: n length vector

    Output:
    w: m length vector - weights for features in X
    """

    # TODO:
    w = None
    w = np.matmul(np.linalg.inv(np.matmul(np.transpose(X), X)), np.matmul(np.tra
nspose(X), y))
    return w
```

In [18]:
```
ols_w = OLS(diabetes_X_train, diabetes_y_train)
```

c. Find the loss for the training set and the test set using the weights found with OLS.

In [19]:

```
# TODO:
ols_train_loss = loss_linreg(diabetes_X_train, ols_w, diabetes_y_train)
ols_test_loss = loss_linreg(diabetes_X_test, ols_w, diabetes_y_test)

print("Method: OLS")
print("Training Loss: " + str(ols_train_loss))
print("Test Loss: " + str(ols_test_loss))
```

```
Method: OLS
Training Loss: 8525634.413813028
Test Loss: 3049364.7063357322
```