Lab 08 – Optimization Methods: Gradient Descent

Author: Nigel Nelson

Introduction:

- This lab acted as an introduction to using gradient descent as optimization method. This is done largely to discover the pros and cons of gradient descent in comparison to optimization methods that have been used previously in this course, such as using grid search as well as manually adjusting parameters. The first portion of this lab is implementing a gradient descent algorithm in the Optimizer class. This implementation builds on the gradient function created for lab07, but uses the gradient as a heuristic to determine which direction to step the parameters towards. The first experiment of this lab showcases the potential power of gradient descent as it correctly identifies the local minima from two local extrema, indicating it is capable of finding optima parameters. However, in the third experiment, the potential shortcomings of the Optimizer is seen as there are two minima in the quartic model, one local and one global. From this experiment it was realized that the starting location of the gradient descent algorithm can have a profound impact on the optima parameters found. Lastly, experiment 3 is conducted to explore the Optimizer’s ability to optimize the Gaussian distribution, which has been visited several time for this course and allows for a strong intuition of the Optimizer’s abilities. This final experiment results in parameters that creates predicted Gaussian Distribution values with an extremely low MSE, much lower than the values that were found using the grid search method as well as manually selecting for parameters.

Results:

- Experiment 1 used a Cubic model to represent a cost function and used gradient descent to search the corresponding parameter space for feature values that corresponded to local minima. Ultimately, the Optimizer’s gradient descent was able to find a single optima parameter representing a local minima in the cubic function at **x = ~8.0**, requiring **27 iterations** to find this optima parameter. In experiment 2, a quartic model was used, one which had a local minima as well as a global minima. Using 3 different starting points, the gradient descent implementation found both the local minima, and the global minima, indicating the importance of starting point placement for gradient descent. The starting point at x = 6 found the global minima at **x = 4.646**, requiring **30 iterations** to converge. The starting point at x = 3 found the global minima at **x = 4.646**, requiring **36 iterations** to converge. The starting point at x = -2 found the local minima at **x = -0.646**, requiring **176 iterations** to converge. Lastly, gradient descent was used to optimize parameters for the Gaussian Distribution model. Ultimately, the Optimizer found parameters that corresponded to near perfect predictions. The Optimizer found in **3031 iterations** that the optima parameters for the Gaussian Distribution were **Mu = 5.490** and **Sigma = 1.251**, which corresponds to a MSE of **5e-07**.

Questions:

1. Reflect on the form and organization of our optimizer API. Specifically, discuss each of the methods and what role they serve. This discussion should include what arguments they accept, what the method returns, and why we might choose to separate out these specific methods into helper methods.
   - **__init__** Served the purpose of initializing the Optimizer class and accepting all of the parameters required to specify the behavior of our gradient decent implementation. This included taking the step size, max iterations, tolerance, and the delta all as parameters so that these values could be saved and used as needed in the Optimizer class.
   - **__calculate_change** Served the purpose of calculating the distance between the new set of parameters and the old set of parameters. This was an important step as this distance was compared to the input tolerance parameter, and if this distance was less than the tolerance, then that indicated convergence and that the optimum parameter values had been found.
   - **__gradient** Served the purpose of calculating the approximate derivative of the provided cost function, at the point specified by the provided parameters. The gradient indicates whether the set of parameters is at a point such that they're either increasing away from a minima, or decreasing towards a minima, or are approximately at a zero meaning that they are at an approximate extrema. This gradient acts as a heuristic in our optimizer and guides our parameters towards a local minima according to the supplied cost function.
   - **__update** Served the purpose of calculating the next set of parameters to evaluate using the cost function. This function accepts the old params and the gradient, this is so that the gradient can be multiplied by the step_size, so that this value can be subtracted from the old params. This is done so that the parameters are adjusted according to the specified step_size, using the gradient as a heuristic to determine to what degree the parameters need to be adjusted positively, or negatively, such that the new parameters will descend another step towards a minima.
   - **optimize** Serves the purpose of using all of the above listed helper functions to determine an optimized set of parameters, and returns these optimized parameters along with the number of iterations that was needed to attain those parameters. This method accepts a cost function and the starting set of parameters as arguments. This is done so that while the max iterations has not been exceeded, and the tolerance has not been met, this method can use the cost function and provided parameters in order to calculate the gradient, which is then used to calculate the new parameters, which is then used by the calculate_change method to determine if the tolerance has been met. The helper methods that are defined in this class are used in this class to allow for a fluent API which allows the optimizer function to have a highly interpretable main loop, that simply consists of assignments and is void of complex calculations. In addition, by separating many of the calculations to helper functions, debugging the optimize becomes a much simpler task as each individual calculation is compartmentalized in a helper function.
2. For experiment 1:
   A. how many optima did you find? Hint: Discuss the significance of places where the derivative is equal to 0.

   - A single optima was found for experiment 1. The significance behind this is that while there were two local extrema for the cubic model, one represented a local maxima and one represented a local minima. Due to the fact that this cubic model was evaluated as a cost function, the Optimizer sought a local minima, which would represent parameters with the lowest amount of loss. It is for that reason experiment 1 returned a single optima.

B. When you used the optimizer you started at x = 0. How many optima did your optimizer return? Was it a minima or maxima? Was it a global or local optima? By looking at the gradient descent algorithm find what term pointed you toward the minimum. Describe how it did this. Can you think of a way to find the gradient. function's maxima?

- When the optimizer started at x = 0, the Optimizer class found a single optima. This optima represented a local minima. The term that pointed towards the minima was the gradient. This is due to the fact that the gradient represents to what degree the cost function is increasing or decreasing. The way the gradient could be used to find the function's maxima would be to add (gradient * step_size) to the existing params. This is because if the params were on an upwards slope towards a local maxima, the gradient would be positive and thus adding to the params would represent a step towards the local maxima. If however the params were on a downwards slope away from a local maxima, adding the gradient would result in a negative step back towards the local maxima.

3. For experiment 2:
   A. how many optima did you find?

   - 2 optima were found by the Optimizer function.

   B. Describe the different starting locations that you used to solve for optima. Was the found optima different for any of these starting locations and were they the global or local optima? If it was, can you explain why the optimizer found different solutions?

   - 3 different starting points were attempted. These starting points were -2, 3, and 6. The starting point of -2 found a local minima that represented a plateau in the quartic model. However, the starting points of 3 and 6 both found the same global minima. The reason that the Optimizer found different solutions is due to the fact that this Optimizer searches for local minima by traversing down which ever slope it is on towards a point where the derivative is approximately zero. The reason -2 found a different local minima is due to the fact that the slope -2 was on corresponded to different local minima than the slopes that the other starting points were placed on.
4. For experiment 3:
   A. how many optima did you find?

   - A single optima was found by the Optimizer in the 3rd experiment

   B. Look back at the heatmaps you generated in Lab06 for the gaussian distribution. Describe what the optimizer is doing using the heatmap visualization.

- The heatmap produced in Lab06 uses Mu and Sigma values on the x and y axis's, and the resulting heatmap results in a circular pattern that descends to lower and lower values of MSE towards the center of this circle. Once assigned a starting point, what the Optimizer is doing is seeing this heatmap as a funnel towards optima parameters. At a given point in the funnel, the Optimizer detects which direction leads further down the funnel, and takes a step in that direction, and continues this evaluation and step series until it finds itself at the very bottom of the funnel.

Imports:

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from optim import *
from test_optim import *
from cost_functions import *
```

Verifying Optimizer implementation:

In [2]:
```
!python test_optim.py
```

```
......
----------------------------------------------------------------------
Ran 6 tests in 0.002s

OK
```

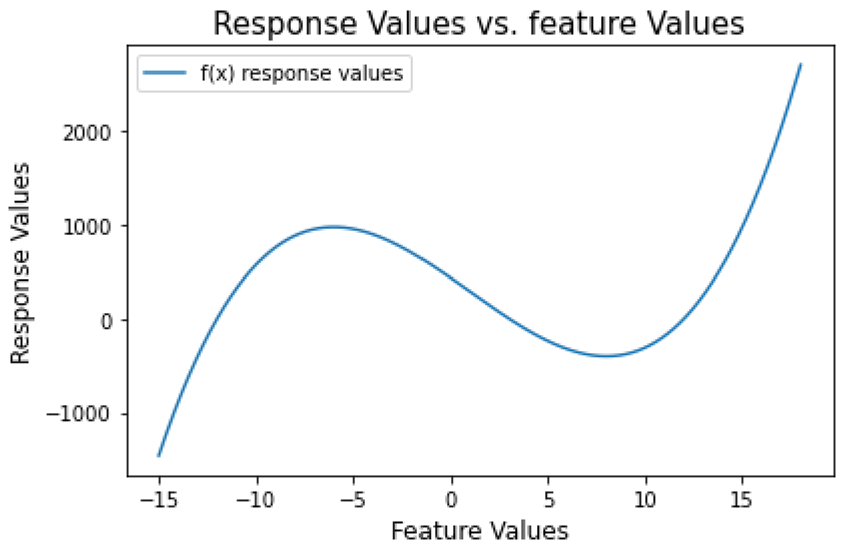
Experiment 1: Cubic Model

Defining the cost function & derivative of the cost function:

In [3]:
```
def f(x):
    return x**3 - 3*x**2 - 144*x + 432

def f_prime(x):
    return 2*x**2 - 6*x - 144
```

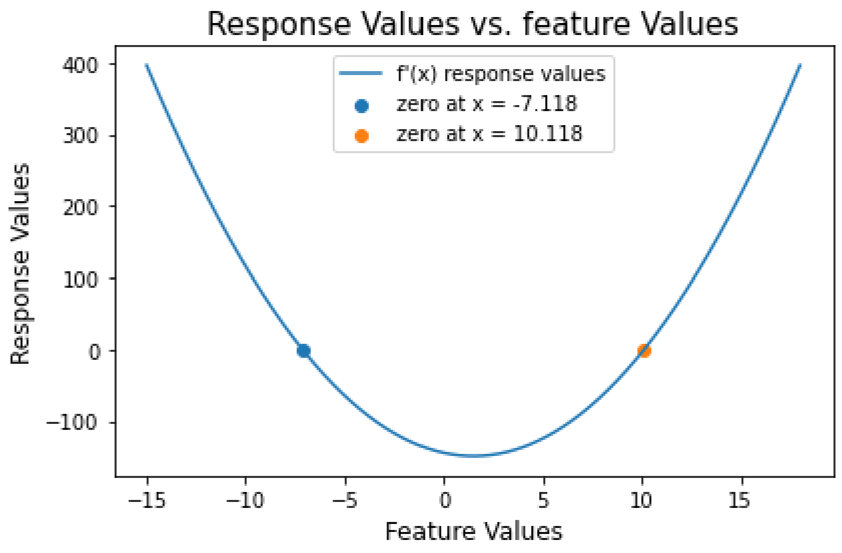
Plotting f(x):

In [4]:
```
x_values = np.linspace(-15,18, 3300)

plt.plot(x_values, f(x_values), label='f(x) response values')
plt.title("Response Values vs. feature Values", fontsize=15)
plt.xlabel("Feature Values", fontsize=12)
plt.ylabel("Response Values", fontsize=12)
plt.legend()
plt.tight_layout()
```



Plotting f'(x) with zeros:

In [5]:
```
f_prime_values = f_prime(x_values)
zeros = x_values[np.argsort((f_prime_values**2))[:2]]

plt.plot(x_values, f_prime_values, label="f'(x) response values")
plt.scatter(zeros[0], f_prime(zeros[0]), label="zero at x = -7.118")
plt.scatter(zeros[1], f_prime(zeros[1]), label="zero at x = 10.118")
plt.title("Response Values vs. feature Values", fontsize=15)
plt.xlabel("Feature Values", fontsize=12)
plt.ylabel("Response Values", fontsize=12)
plt.legend()
plt.tight_layout()
```



Creating cubic objective function class:

In [6]:
```
class CubicObjectiveFunction:
    def cost(self, params):
        return params**3 - 3*params**2 - 144*params + 432
```

Initializing Optimizer:

In [7]:
```
step_size = 0.01
max_iter = 100
tol = 1*10**-5
delta = 1*10**-4

optimizer = Optimizer(step_size, max_iter, tol, delta)
```

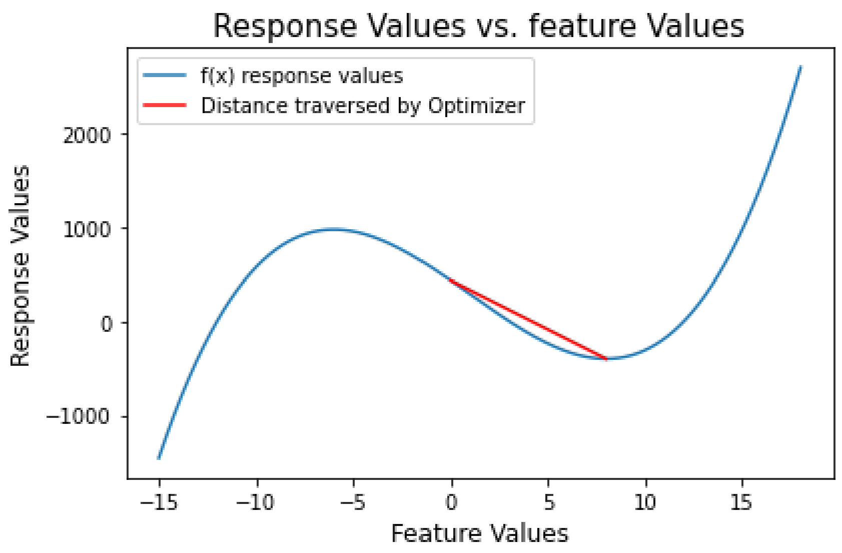
Calculating optimum parameters using the Optimizer:

In [8]:
```
starting_params = np.zeros(1)
optim_params, iters = optimizer.optimize(CubicObjectiveFunction(), starting_p
print(f'Optimum parameters found: {optim_params[0]}')
print(f'Required iterations: {iters}')
```

```
Optimum parameters found: 7.999927191312395
Required iterations: 27
```

Plotting feature space covered by gradient descent optimizer:

In [9]:
```
x_values = np.linspace(-15,18, 3300)

plt.plot(x_values, f(x_values), label='f(x) response values')
plt.plot([0, optim_params[0]], [f(0), f(optim_params[0])], c='r', label="Dist
plt.title("Response Values vs. feature Values", fontsize=15)
plt.xlabel("Feature Values", fontsize=12)
plt.ylabel("Response Values", fontsize=12)
plt.legend()
plt.tight_layout()
```



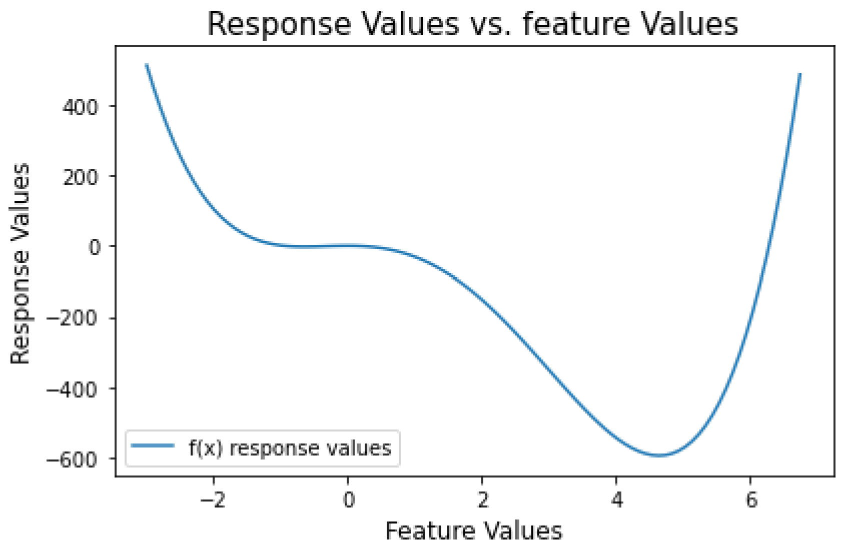
Experiment 2: Quartic Model

Defining quartic model & associated derivative model:

In [10]:
```
def f(x):
    return 3*x**4 - 16*x**3 - 18*x**2

def f_prime(x):
    return 12*x**3 - 48*x**2  - 36*x
```

Plotting quartic model:

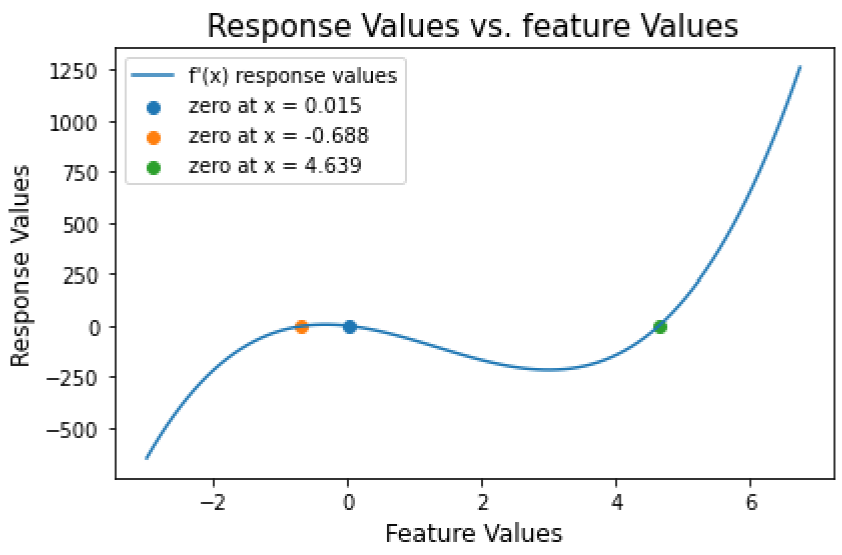
In [11]:
```python
x_values = np.linspace(-3, 6.75, 98)

plt.plot(x_values, f(x_values), label='f(x) response values')
plt.title("Response Values vs. feature Values", fontsize=15)
plt.xlabel("Feature Values", fontsize=12)
plt.ylabel("Response Values", fontsize=12)
plt.legend()
plt.tight_layout()
```



Plotting derivative of quartic model with zeros:

In [12]:
```
f_prime_values = f_prime(x_values)
zeros = x_values[np.argsort((f_prime_values**2))[:3]]
print(zeros)
plt.plot(x_values, f_prime_values, label="f'(x) response values")
plt.scatter(zeros[0], f_prime(zeros[0]), label="zero at x = 0.015")
plt.scatter(zeros[1], f_prime(zeros[1]), label="zero at x = -0.688")
plt.scatter(zeros[2], f_prime(zeros[2]), label="zero at x = 4.639")
plt.title("Response Values vs. feature Values", fontsize=15)
plt.xlabel("Feature Values", fontsize=12)
plt.ylabel("Response Values", fontsize=12)
plt.legend()
plt.tight_layout()
```

```
[ 0.01546392 -0.68814433  4.63917526]
```



Creating quartic objective function class:

In [13]:
```
class QuarticObjectiveFunction:
    def cost(self, params):
        return 3*params**4 - 16*params**3 - 18*params**2
```

Initializing Optimizer:

In [14]:
```
step_size = 0.001
max_iter = 1000
tol = 1*10**-5
delta = 1*10**-4

optimizer = Optimizer(step_size, max_iter, tol, delta)
```

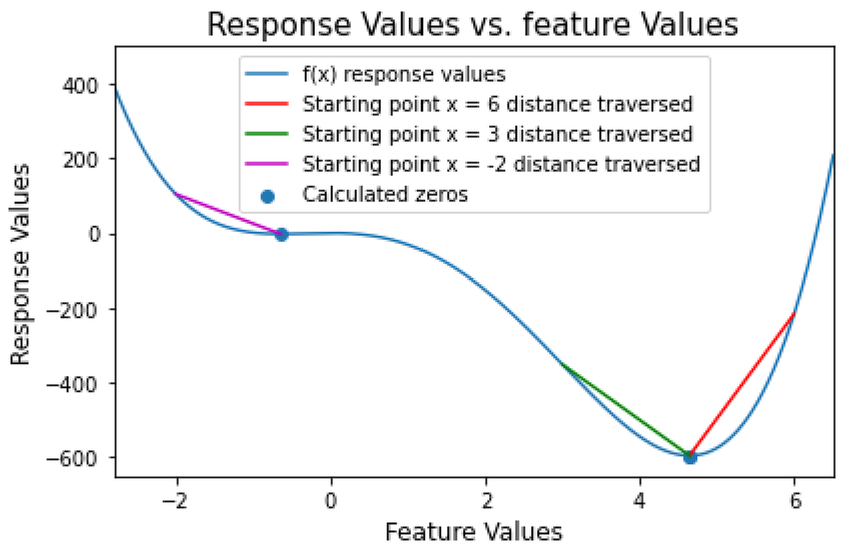
Calculating optimum parameters using the Optimizer:

In [15]:
```
starting_points = np.array([6., 3., -2.])
optim_params = []

for point in starting_points:
    optim_param, iters = optimizer.optimize(QuarticObjectiveFunction(), np.ar
    optim_params.append(optim_param[0])
    print(f'Starting point: {point}')
    print(f'Optimum parameters found: {optim_param[0]}')
    print(f'Required iterations: {iters}\n')
```

```
Starting point: 6.0
Optimum parameters found: 4.645728617403847
Required iterations: 30

Starting point: 3.0
Optimum parameters found: 4.645673715540852
Required iterations: 36

Starting point: -2.0
Optimum parameters found: -0.6460408353327978
Required iterations: 176
```

Plotting feature space covered by gradient descent optimizer:

In [16]:
```
x_values = np.linspace(-15,18, 3300)

plt.plot(x_values, f(x_values), label='f(x) response values')
plt.scatter(optim_params, f(np.array(optim_params)), label="Calculated zeros"
plt.plot([6, optim_params[0]], [f(6), f(optim_params[0])], c='r', label="Star
plt.plot([3, optim_params[1]], [f(3), f(optim_params[1])], c='g', label="Star
plt.plot([-2, optim_params[2]], [f(-2), f(optim_params[2])], c='m', label="St
plt.title("Response Values vs. feature Values", fontsize=15)
plt.xlabel("Feature Values", fontsize=12)
plt.ylabel("Response Values", fontsize=12)
plt.legend()
plt.xlim(-2.8, 6.5)
plt.ylim(-650, 500)
plt.tight_layout()
```



Experiment 3: Gaussian Model

Loading gaussdist.csv:

In [17]:
```
data = np.loadtxt(open("gaussdist.csv", "rb"), delimiter=",")
data[:5]
```

Out[17]:
```
array([[6.99000000e+00, 1.56842355e-01],
       [8.90000000e+00, 7.89692304e-03],
       [9.58000000e+00, 1.55088416e-03],
       [5.46000000e+00, 3.18990459e-01],
       [1.38000000e+00, 1.39635489e-03]])
```

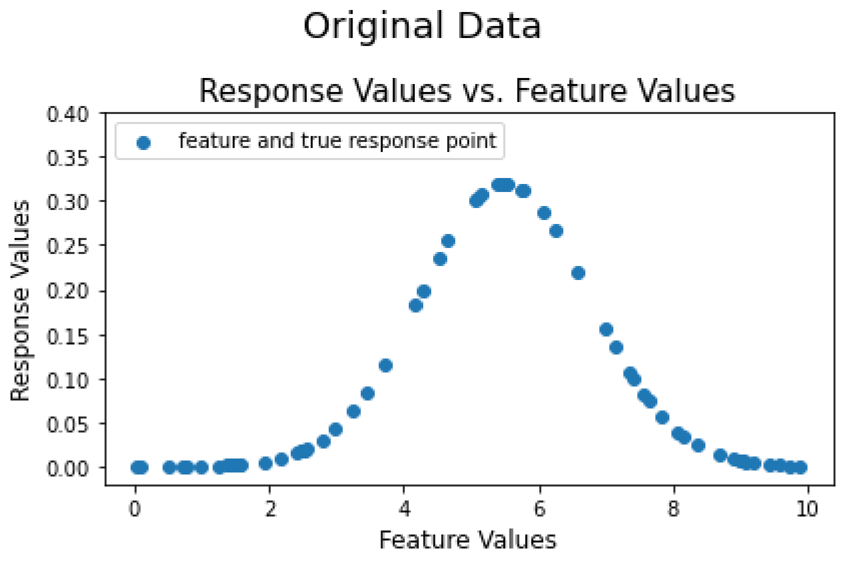
- **Identifying response variable and feature variable:**
  - Based from the Gaussian Distribution data that is loaded in the above cell, it is clear that the second column is not monotonic, which is a characteristic of the output for a Gaussian Distribution. As such, the first column are the feature variable values, and the second column are the response variable values.

Instantiating GaussianCostFunction using the provided data:

In [18]:
```
features = data[:, 0]
responses = data[:, 1]

gaussian_func = GaussianCostFunction(features, responses)
```

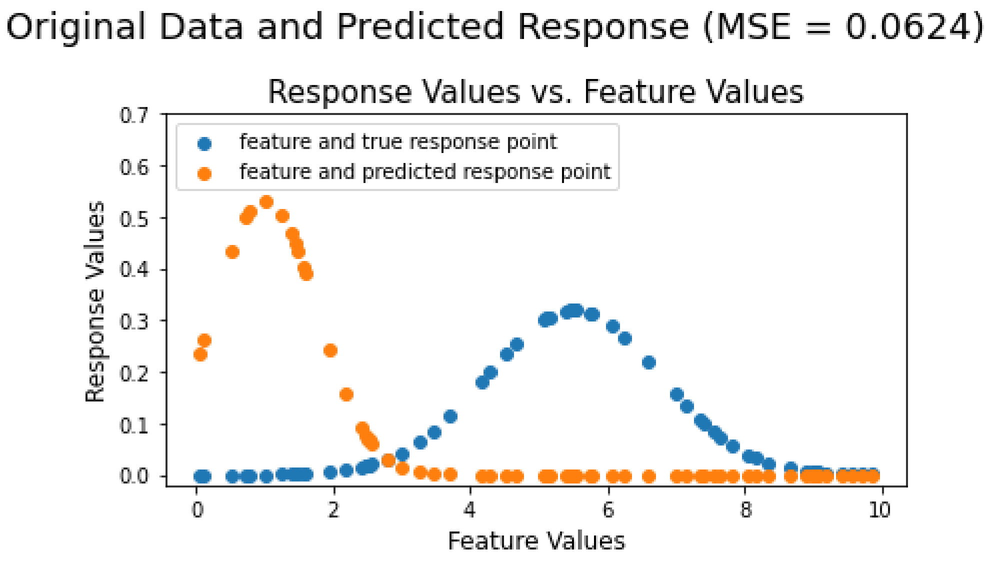
Plotting the provided data:

In [19]:
```
plt.scatter(features, responses, label='feature and true response point')
plt.title("Response Values vs. Feature Values", fontsize=15)
plt.xlabel("Feature Values", fontsize=12)
plt.ylabel("Response Values", fontsize=12)
plt.suptitle('Original Data', fontsize=18)
plt.legend(loc=2)
plt.ylim(-.02, .4)
plt.tight_layout()
```



Plotting the provided data alongside predicted output:

In [20]:
```
pred_y = gaussian_func.predict(features, np.array([1, 0.75]))
mse_score = gaussian_func.cost(np.array([1, 0.75]))

plt.scatter(features, responses, label='feature and true response point')
plt.scatter(features, pred_y, label='feature and predicted response point')
plt.title("Response Values vs. Feature Values", fontsize=15)
plt.xlabel("Feature Values", fontsize=12)
plt.ylabel("Response Values", fontsize=12)
plt.suptitle(f'Original Data and Predicted Response (MSE = {round(mse_score,
plt.legend(loc=2)
plt.ylim(-.02, .7)
plt.tight_layout()
```



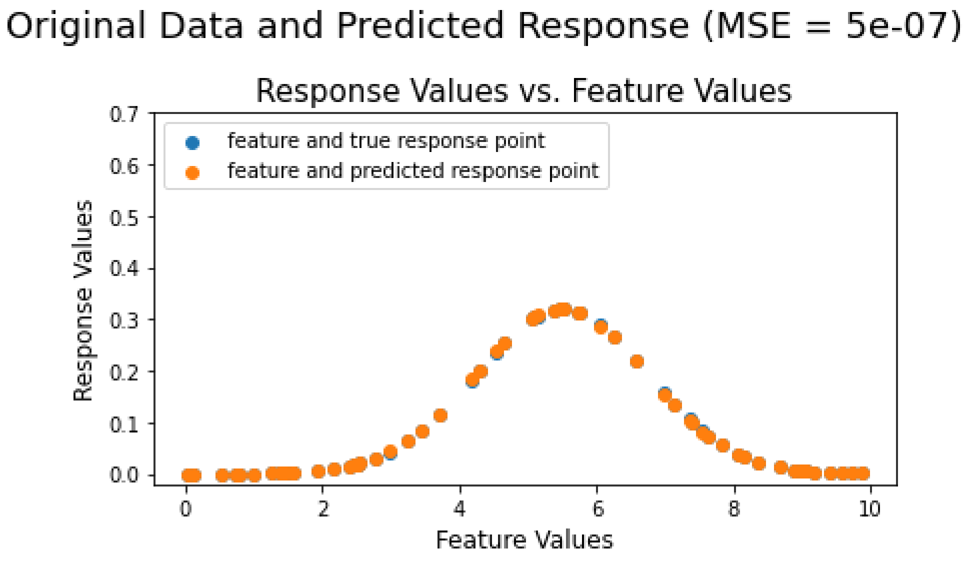
Initializing Optimizer and optimizing parameters:

In [21]:
```
step_size = 1
max_iter = 5000
tol = 1*10**-4
delta = 1*10**-3

optimizer = Optimizer(step_size, max_iter, tol, delta)
optim_params, iters = optimizer.optimize(gaussian_func, np.array([1., 0.75]))

print(f'Optimized Mu value: {optim_params[0]}')
print(f'Optimized Sigma value: {optim_params[1]}')
print(f'Iterations required to optimize parameters: {iters}')
```

```
Optimized Mu value: 5.490361029529459
Optimized Sigma value: 1.2507025179253015
Iterations required to optimize parameters: 3031
```

Plotting Gaussian function using optimized parameters:

In [22]:
```
pred_y = gaussian_func.predict(features, optim_params)
mse_score = gaussian_func.cost(optim_params)

plt.scatter(features, responses, label='feature and true response point')
plt.scatter(features, pred_y, label='feature and predicted response point')
plt.title("Response Values vs. Feature Values", fontsize=15)
plt.xlabel("Feature Values", fontsize=12)
plt.ylabel("Response Values", fontsize=12)
plt.suptitle(f'Original Data and Predicted Response (MSE = {round(mse_score,
plt.legend(loc=2)
plt.ylim(-.02, .7)
plt.tight_layout()
```


Original Data and Predicted Response (MSE = 5e-07)
Response Values vs. Feature Values
feature and true response point
feature and predicted response point
Response Values
Feature Values