Jacob Thieret S01984343 – Undergraduate
I enjoyed this class. The material was interesting, and projects were very fun to work on and learn from. Thank you.

I. Introduction

Overview

The objective of this project is to explore and comprehend gradient-based optimization algorithms, specifically focusing on Gradient Descent (GD) and Stochastic Gradient Descent (SGD) and mini-batch descent. We achieved this by designing polynomial models with varying degrees and employing these optimization techniques to fit our models to a predefined dataset. Our work primarily involved:

1. Implementation of GD, SGD, and mini-batch SGD optimizers.
2. Utilizing these optimizers to fit polynomial models to the provided dataset.

Objective

Our task involved fitting a curve to 100 pairs of (X, Y) points, which were supplied in the "Part1_x_y_Values.txt" file. We accomplished this using 2nd (quadratic), 3rd (cubic), and 4th-degree polynomials, equivalent to models with 3, 4, and 5 features, respectively. An added challenge was to explore the effects of varying learning rates on the performance of our model.

II. Methodology

Initializing X Y coordinates + Normalizing

```
with open('Project 3/Part1_x_y_Values.txt', 'r') as f:
    lines = f.readlines()
    for line in lines[1:]:
        # Remove parentheses and split string at comma
        x, y = line.strip()[1:-1].split(', ')
        x = x.replace(',','')
        y = y.replace(',','')
        # Append values to respective lists
        x_values.append(float(x))
        y_values.append(float(y))
# np array to store cleaned x y values
X = np.array(x_values)
Y = np.array(y_values)
```

X Y are saved into NumPy arrays, as it is conveniently easy to perform standard calculations over the array such as mean and standard deviation. Very convenient in our case.

Normalizing:

```
X = (X - np.mean(X)) / np.std(X)
Y = (Y - np.mean(Y)) / np.std(Y)
```

The main objective of normalization is to change the values in the dataset to a common scale, without distorting differences in the ranges of values or losing information. The expression `(X - np.mean(X)) / np.std(X)` subtracts the mean from each value in 'X' (which effectively makes the new average of 'X' to be zero), and then divides each value by the standard deviation (which scales the values such that their new standard deviation is 1). This process results in 'X' having a mean of 0 and a standard deviation of 1, and likewise for ‘Y’. Some machine learning algorithms require the input data to have such a standard scale (zero mean and unit variance) to function correctly. Retrieved from developers.google.com/machine-learning/data-prep/transform/normalization.

II. Methodology

Hypothesis and Cost Functions

Hypothesis Function

The hypothesis function formulates the mathematical relationship we hypothesize exists between the input data (x-values) and the output data (y-values). We implemented polynomial functions of degrees 2, 3, and 4 as part of our experimentations.

For a model with **three features (a, b, c)**, the hypothesis function was quadratic:

$$Y = aX^2 + bX + c$$

```
def quadratic_hypothesis(x, a, b, c):
    return a * (x ** 2) + b * x + c
```

For a model with **four features (a, b, c, d)**, the hypothesis function was cubic:

$$Y = aX^3 + bX^2 + cX + d$$

```
def hypothesis(x, a, b, c, d):
    return a * (x ** 3) + b * (x ** 2) + c * x + d
```

For a model with **five features (a, b, c, d, e),** the hypothesis function was of degree 4:

$$Y = aX^4 + bX^3 + cX^2 + dX + e$$

```
def hypothesis(x, a, b, c, d, e):
    return a * (x ** 4) + b * (x ** 3) + c * (x ** 2) + d * x + e
```

Cost Function

The cost function measures how well the hypothesis function is doing at correctly predicting the output for a given input. In our case, we used the mean squared error cost function, which calculates the average of the squares of the differences between the actual output (Y) and the predicted output (our hypothesis function).

3 Features:

```
def cost_function(X, Y, a, b, c):
    return np.mean((Y - quadratic_hypothesis(X, a, b, c)) ** 2)
```

4 Features:

```
def cost_function(X, Y, a, b, c, d):
    return np.mean((Y - hypothesis(X, a, b, c, d)) ** 2)
```

5 Features:

```
def cost_function(X, Y, a, b, c, d, e):
    return np.mean((Y - hypothesis(X, a, b, c, d, e)) ** 2)
```

Gradient Descent - 3 Features

In batch gradient descent, all training examples are used to compute the gradient of the cost function for each iteration of the training algorithm. This method is computationally expensive, but it's deterministic and produces a stable error gradient and convergence.

In our gradient descent implementation, we used a quadratic hypothesis function to represent a model with three features. The initial weights for the three features a, b, c were set to 0.0. We set a learning rate of 0.01 and set the algorithm to run for 100 epochs. Initialization of variables and calling the function is identical for the other descent functions.

```
a, b, c = 0.0, 0.0, 0.0
learning_rate = 0.01
epochs = 100
a_gd, b_gd, c_gd, cost_history_gd = gradient_descent(X, Y, a, b, c
learning_rate, epochs, ax, fig)
```

The weight updating process involves calculating the gradients of the cost function with respect to each feature. This is done by using the following formulas:

```
def gradients(X, Y, a, b, c):
    da = -2 * np.mean((Y - quadratic_hypothesis(X, a, b, c)) * X ** 2)
    db = -2 * np.mean((Y - quadratic_hypothesis(X, a, b, c)) * X)
    dc = -2 * np.mean(Y - quadratic_hypothesis(X, a, b, c))
    return da, db, dc
```

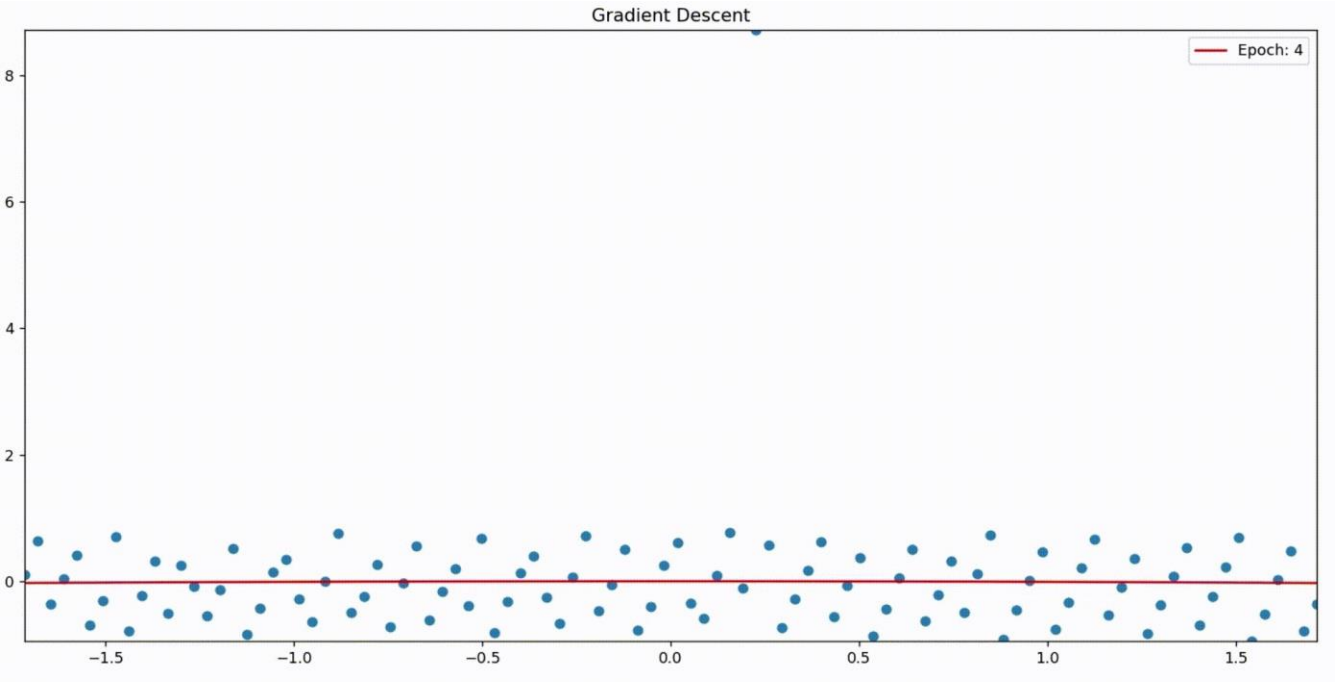
Here, da, db, and dc represent the gradients of the cost function with respect to the features a, b, and c respectively. The quadratic_hypothesis function calculates the predicted output based on the current weights a, b, and c and the input data X. The difference between the actual output Y and the predicted output is then used to calculate the gradients.

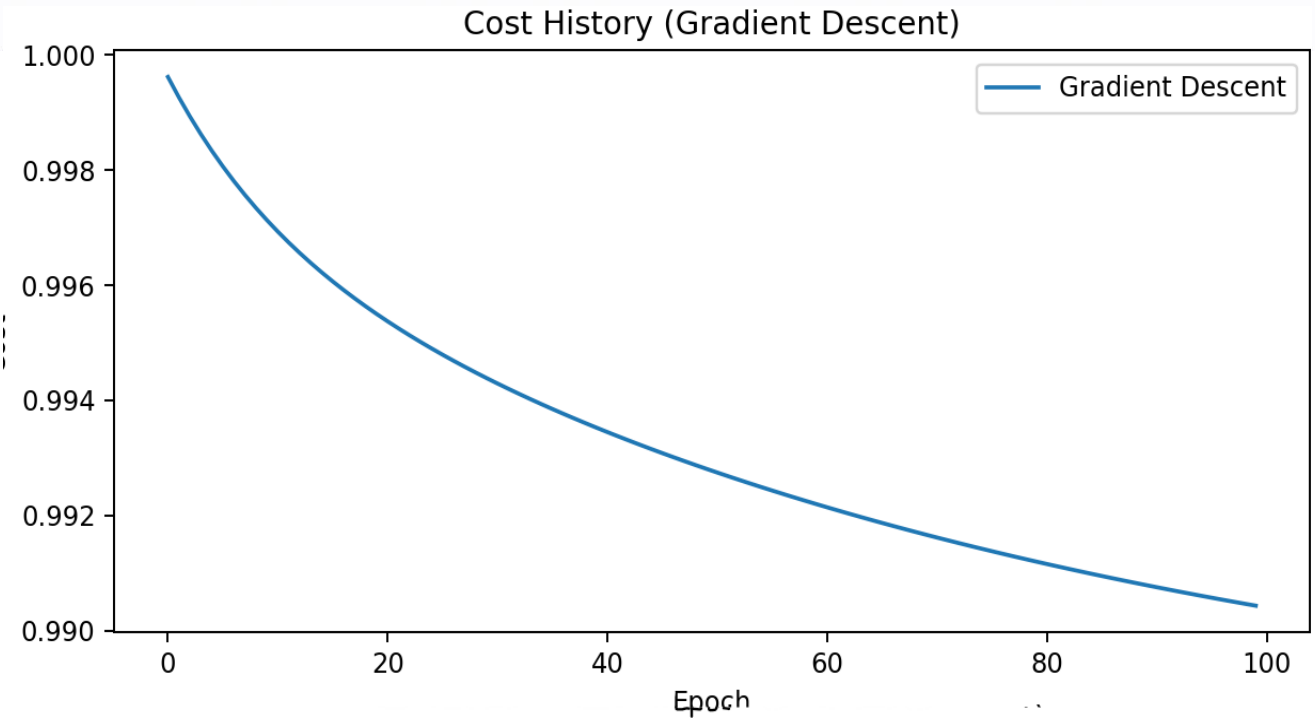
In each epoch, these gradients are used to update the weights a, b, and c. The learning rate of 0.01 is used to control the step size in the direction of the negative gradient. This process is repeated for each epoch, and the history of the cost function is recorded for later analysis.

```
def gradient_descent(X, Y, a, b, c, learning_rate, epochs, ax, fig):
    cost_history = []
    for epoch in range(epochs):
        da, db, dc = gradients(X, Y, a, b, c)
        a -= learning_rate * da
        b -= learning_rate * db
        c -= learning_rate * dc
        cost_history.append(cost_function(X, Y, a, b, c))
    return a, b, c, cost_history
```

We also visualized the fitted quadratic curve and the data points in each epoch using matplotlib. The curve is updated in each epoch to show the current state of the model.

Live Gif: Not available in pdf format. Available in the .docx version of this document or in the project folders as a gif or mp4.





The Gradient Descent has produced a fairly stable error gradient and convergence as we expected we would see from before.

Stochastic Gradient Descent - 3 Features

In SGD, each iteration uses only one training example to compute the gradient of the cost function. SGD is much faster, but the error gradient and convergence are much less stable and much noisier because only a single random example from the dataset is used at each iteration.

Just like in the gradient descent implementation, we started with initial weights of 0.0 for the features a, b, and c. The learning rate was set to 0.01, and the algorithm was set to run for 100 epochs.

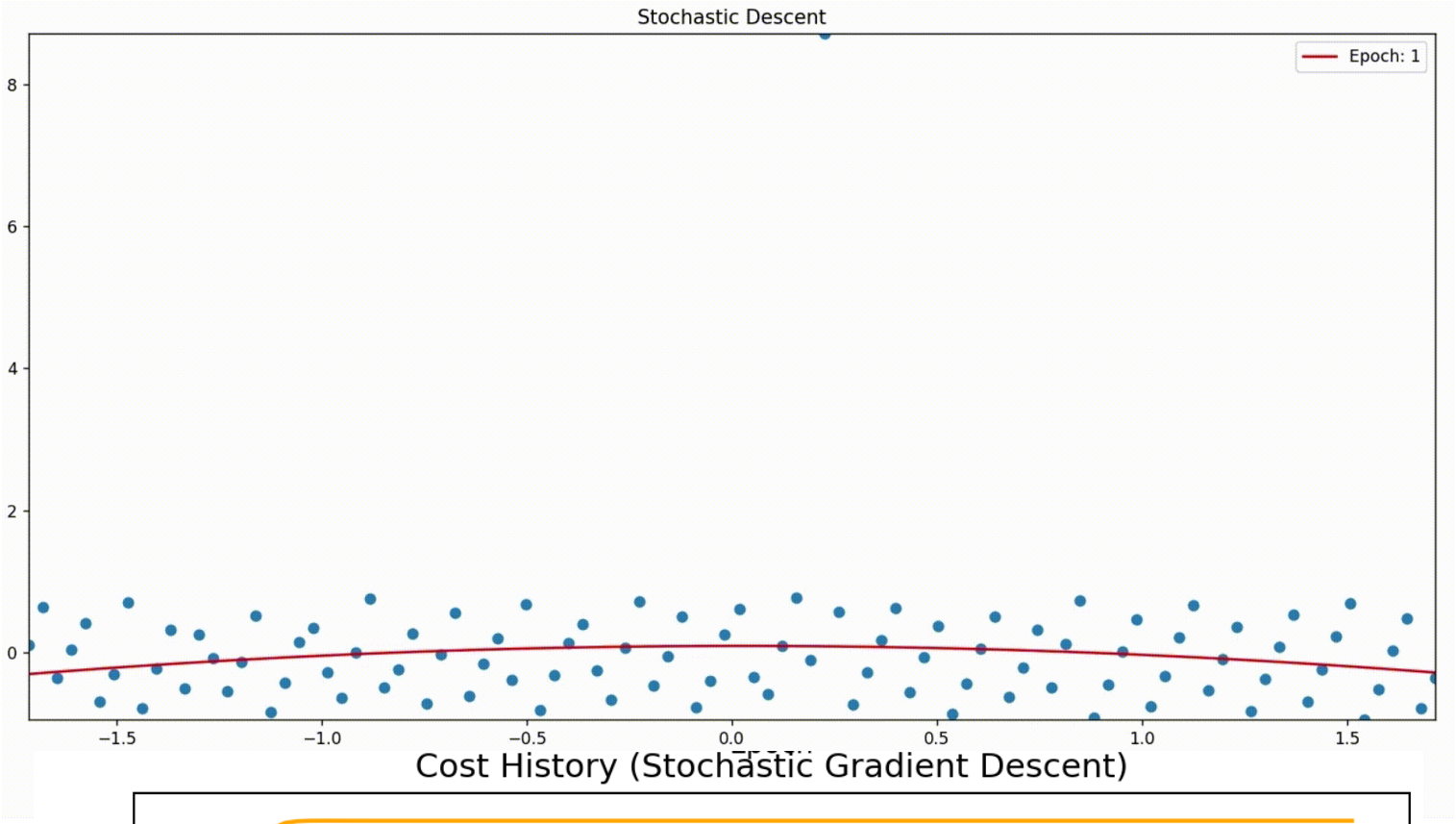
The weight updating process in SGD, however, differed from that of the gradient descent. In each epoch, for each data point in our dataset, we calculated the gradients of the cost function with respect to each feature. These gradients were calculated using the stochastic_gradients function, which uses the same formulas for the gradients as in the gradient descent case but applies them to individual data points rather than the entire dataset.

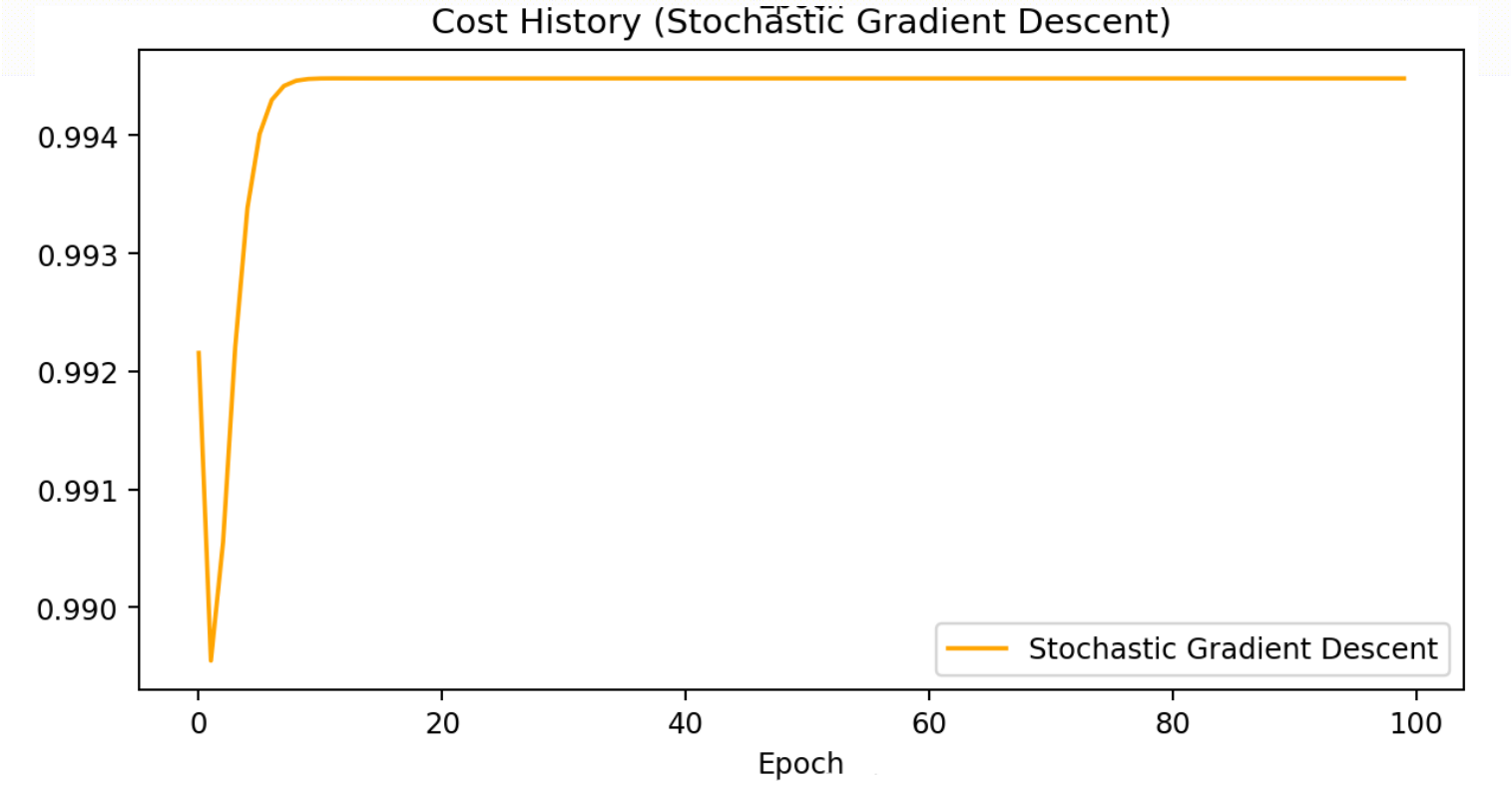
```
def stochastic_gradients(x, y, a, b, c):
    da = -2 * (y - quadratic_hypothesis(x, a, b, c)) * (x ** 2)
    db = -2 * (y - quadratic_hypothesis(x, a, b, c)) * x
    dc = -2 * (y - quadratic_hypothesis(x, a, b, c))
    return da, db, dc
```

Here, x and y correspond to individual data points in the dataset. The gradients da, db, and dc were then used to update the corresponding weights a, b, and c. This process was repeated for every data point in the dataset, and for each epoch.

```
def stochastic_gradient_descent(X, Y, a, b, c, learning_rate, epochs, ax,
fig):
    cost_history = []
    for epoch in range(epochs):
        for i in range(len(X)):
            da, db, dc = stochastic_gradients(X[i], Y[i], a, b, c)
            a -= learning_rate * da
            b -= learning_rate * db
            c -= learning_rate * dc
        cost_history.append(cost_function(X, Y, a, b, c))
      return a, b, c, cost_history
```

Live Gif: Not available in pdf format. Available in the .docx version of this document or in the project folders as a gif or mp4.





Mini-Batch Gradient Descent - 3 Features

Our third approach, mini-batch gradient descent, is a variant of the previous two techniques (gradient descent and stochastic gradient descent) that aims to balance the advantages of both. Mini-batch gradient descent is a compromise between batch gradient descent and SGD. It uses a mini-batch of 'n' training examples at each step to compute the gradient of the cost function. This method reduces the variance of the parameter updates, which can lead to more stable convergence. It can also take advantage of vectorized operations for computational efficiency.

We again initiated the feature weights a, b, and c to 0.0, with a learning rate of 0.01 over 100 epochs and a batch size of 32. The batch size was another parameter we introduced in this model.

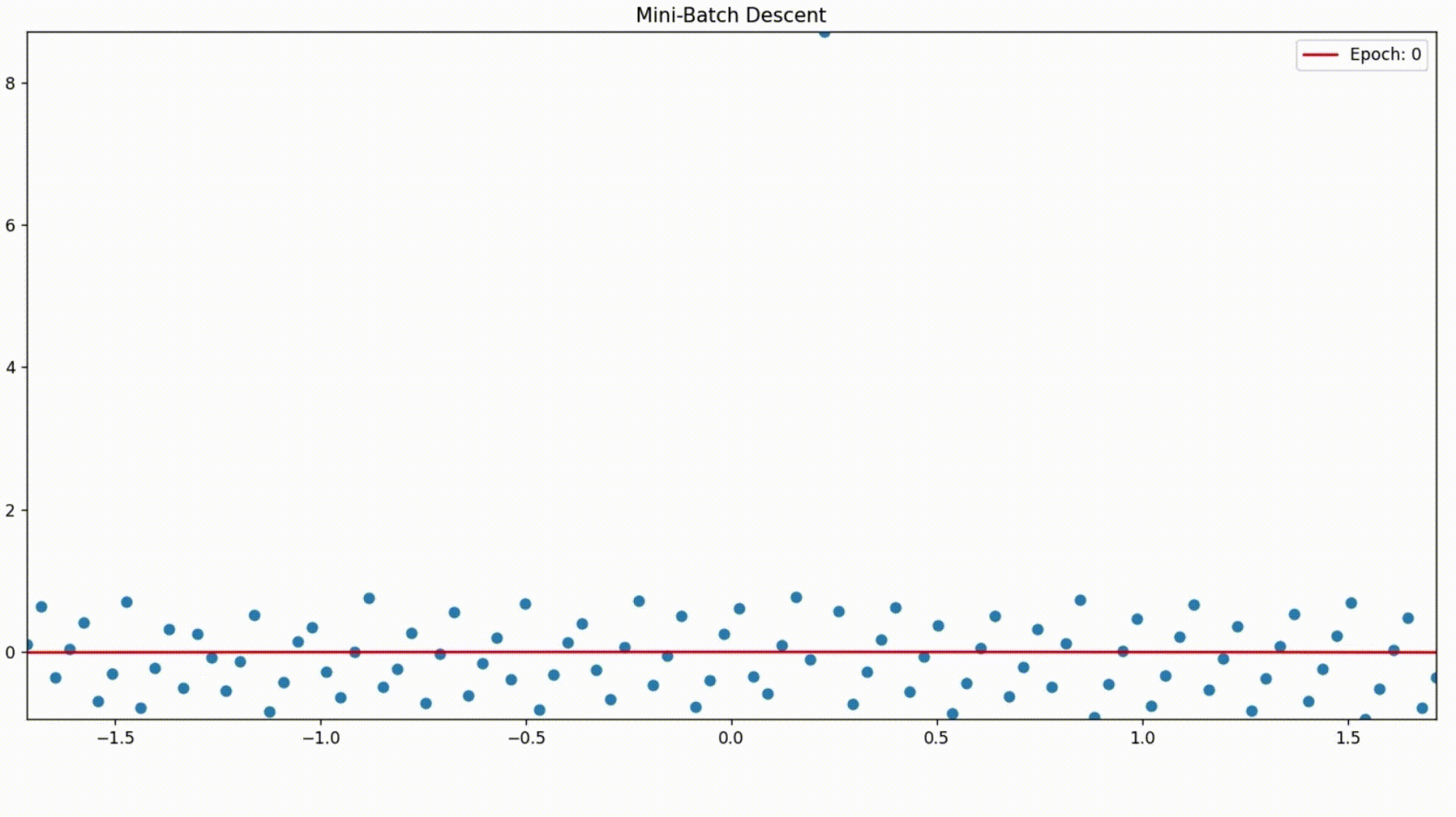
The mini_batch_gradients function randomly selected a subset of data (of size batch_size) from the entire dataset at each iteration. This was done using the np.random.choice function. These mini-batches of data were then used to compute the gradients and update the model weights.

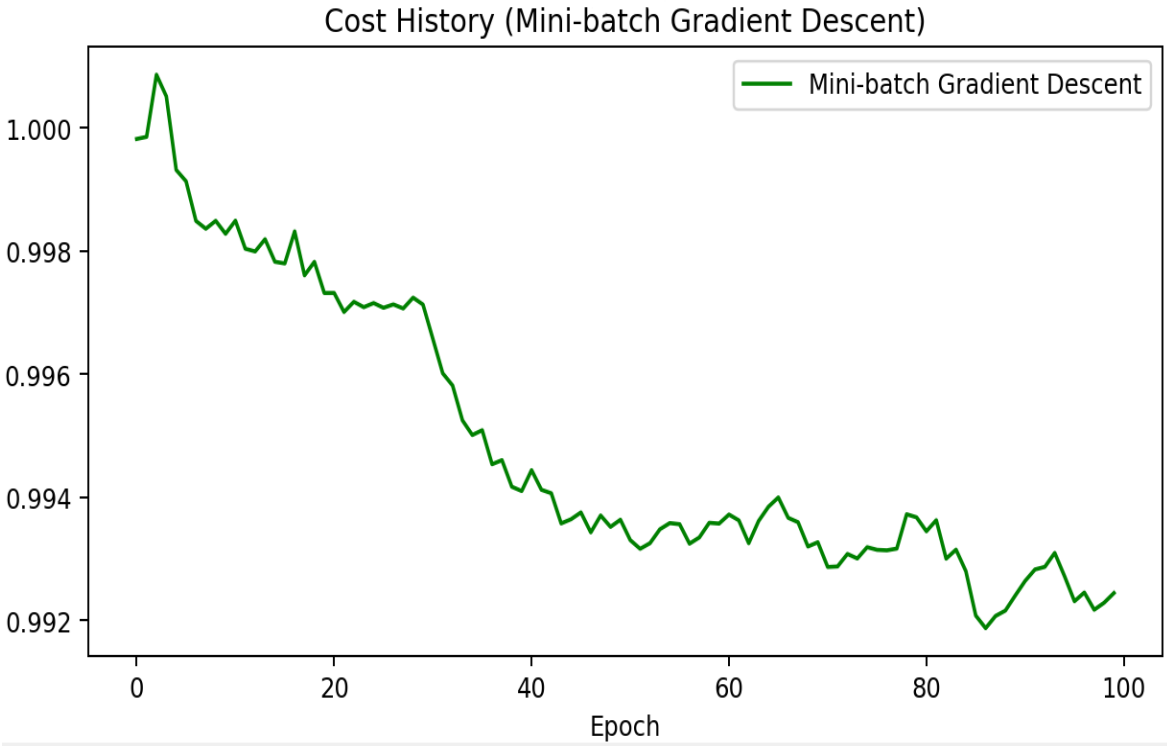
```
def mini_batch_gradients(X, Y, a, b, c, batch_size):
    indices = np.random.choice(len(X), batch_size)
    X_batch = X[indices]
    Y_batch = Y[indices]
    da = -2 * np.mean((Y_batch - quadratic_hypothesis(X_batch, a, b, c)) *
X_batch ** 2)
    db = -2 * np.mean((Y_batch - quadratic_hypothesis(X_batch, a, b, c)) *
X_batch)
    dc = -2 * np.mean(Y_batch - quadratic_hypothesis(X_batch, a, b, c))
    return da, db, dc
```

For each epoch, the mini-batch gradient descent algorithm calculated the gradients of the cost function with respect to each feature (a, b, and c) based on the selected mini-batch of data, and then updated the weights correspondingly.

```
def mini_batch_gradient_descent(X, Y, a, b, c, learning_rate, epochs,
batch_size, ax, fig):
    cost_history = []
    for epoch in range(epochs):
        da, db, dc = mini_batch_gradients(X, Y, a, b, c, batch_size)
        a -= learning_rate * da
        b -= learning_rate * db
        c -= learning_rate * dc
        cost_history.append(cost_function(X, Y, a, b, c))
```

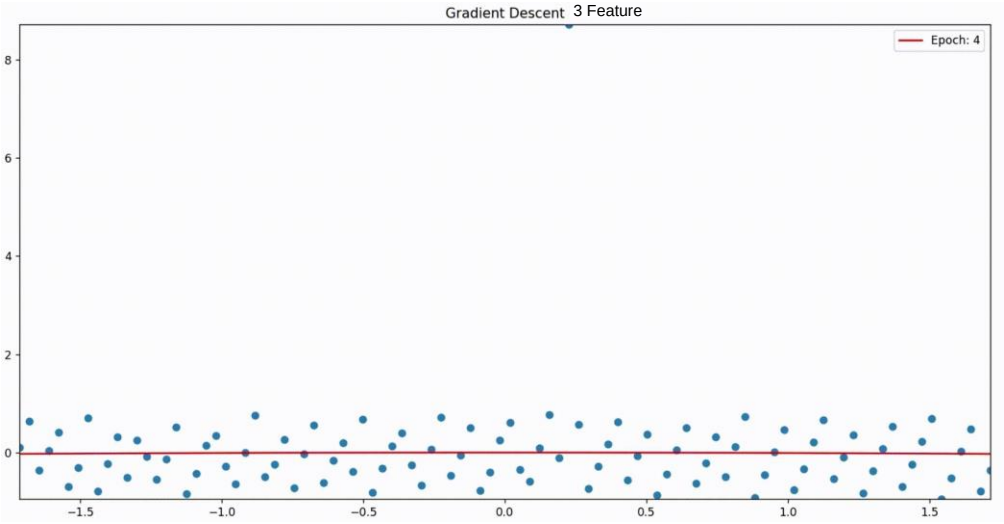
Live Gif: Not available in pdf format. Available in the .docx version of this document or in the project folders as a gif or mp4.

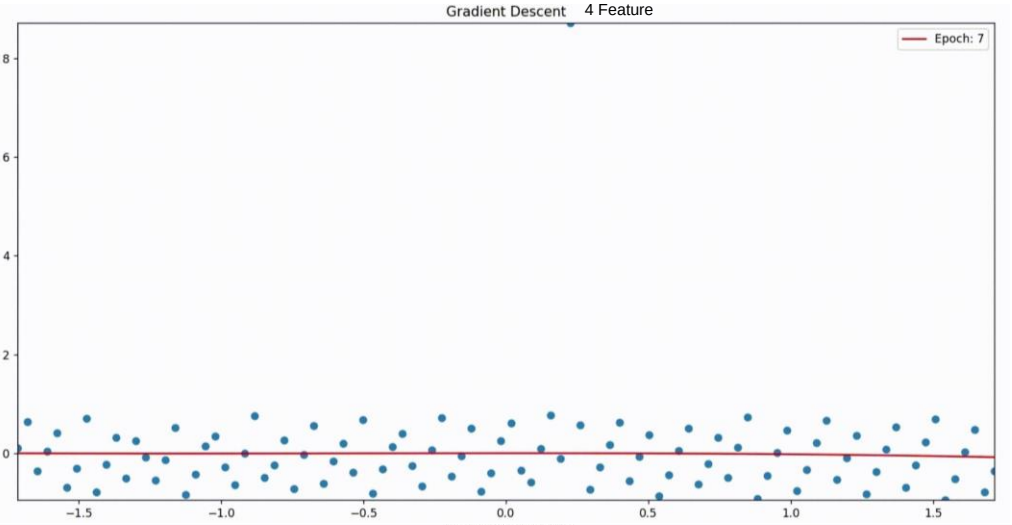


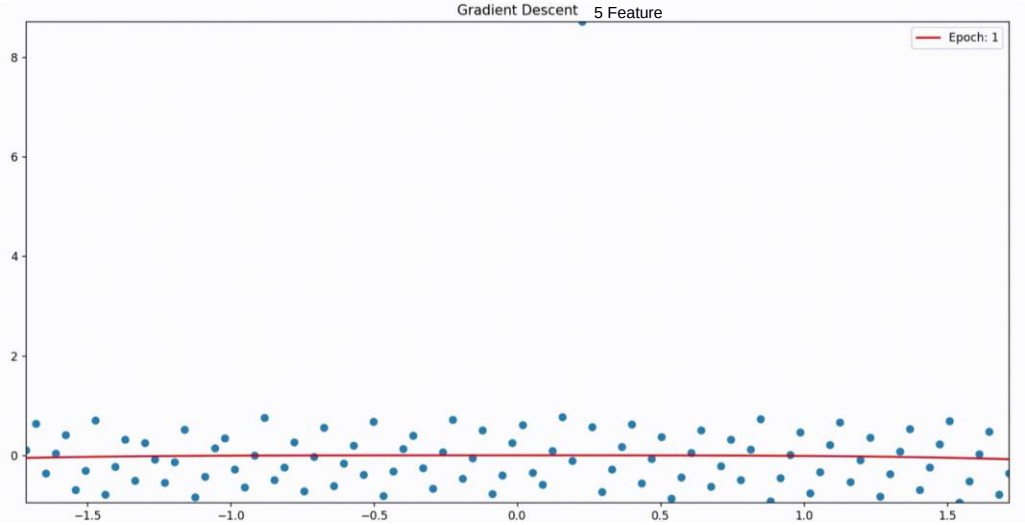


III. Results LIVE PREVIEW NOT AVAILABLE IN PDF FORMAT!

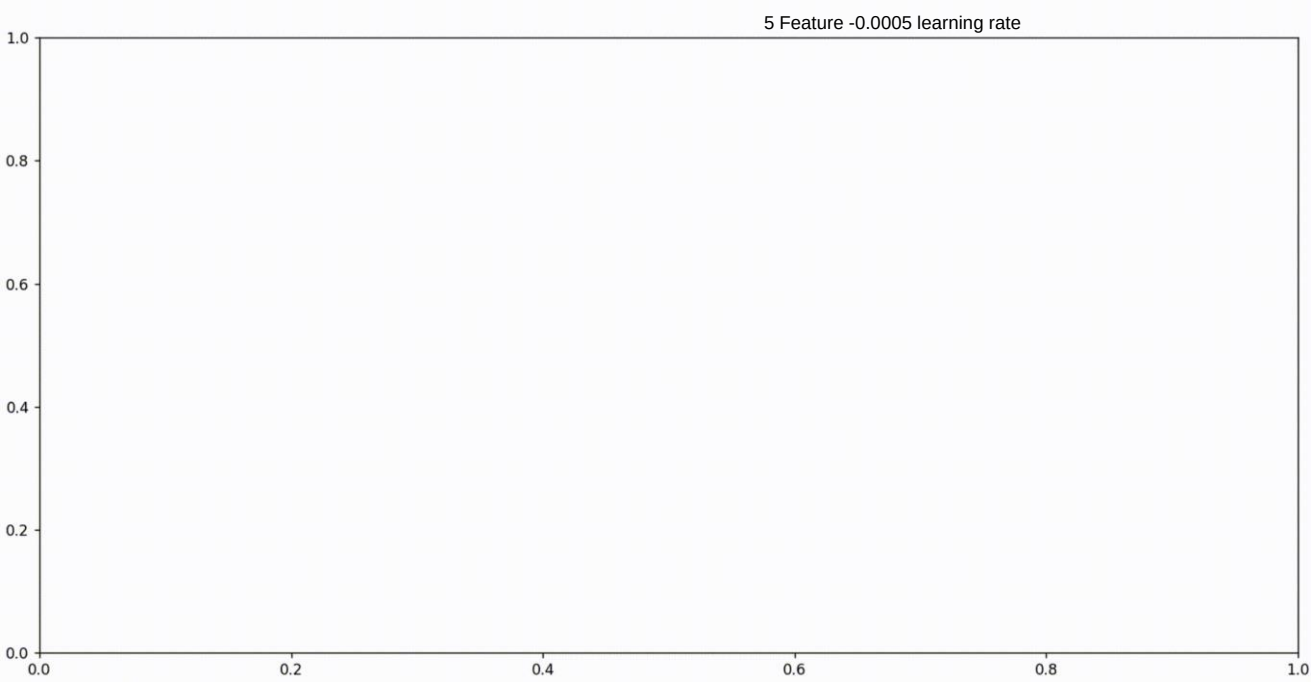
Gradient Descent 3,4,5 Feature Runs

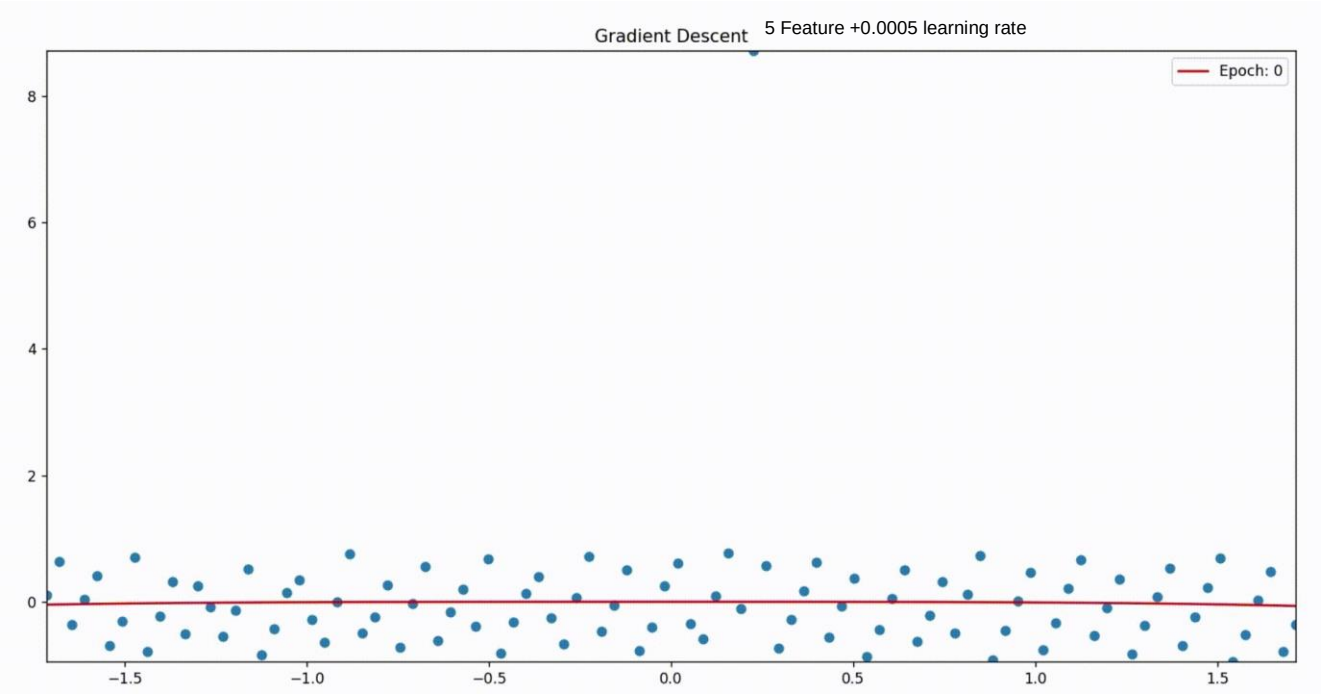




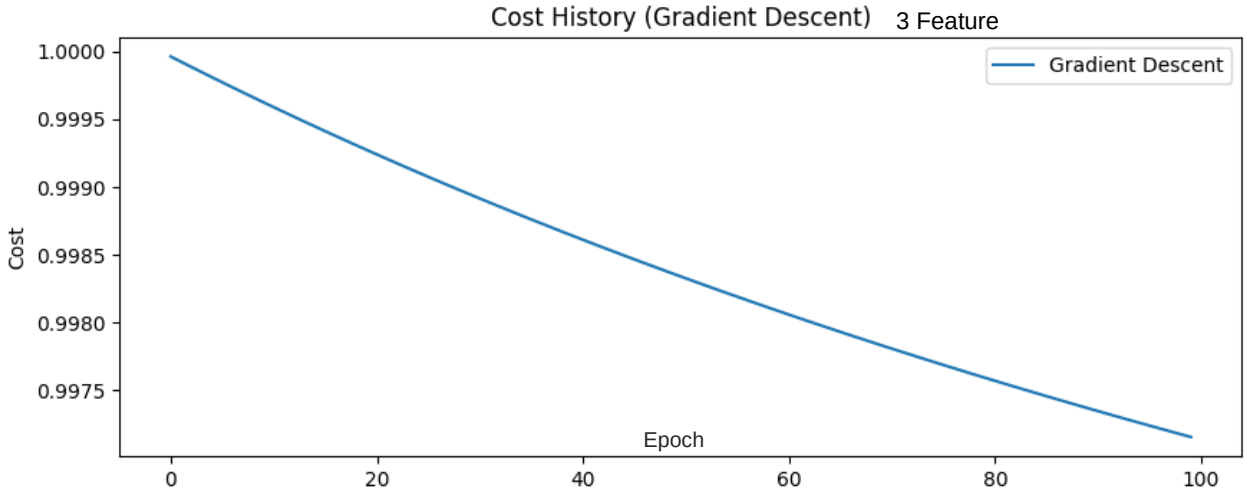


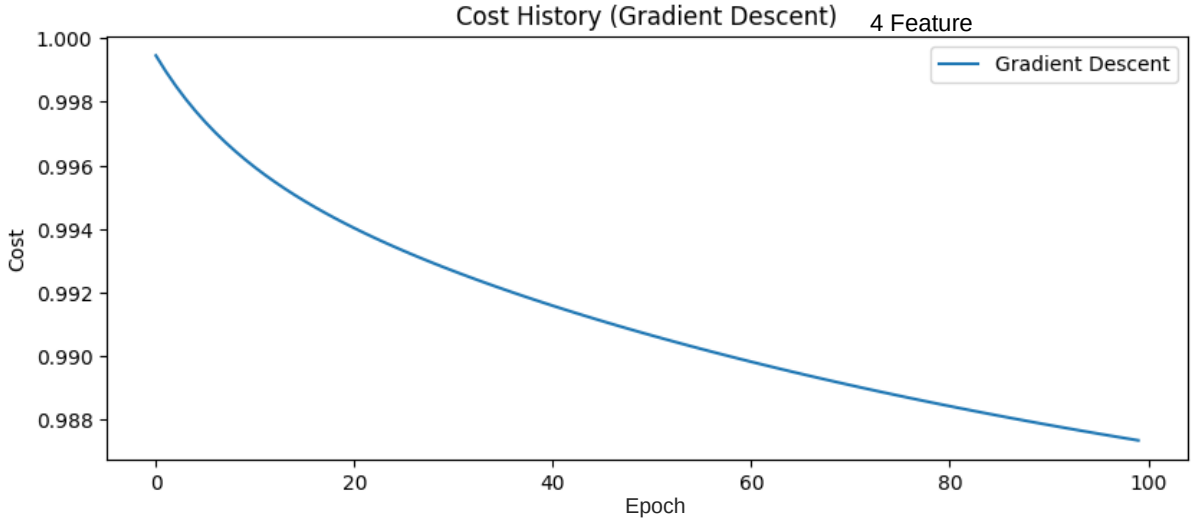
Gradient Descent 5 Feature -/+ 0.005 Learning Rate Runs

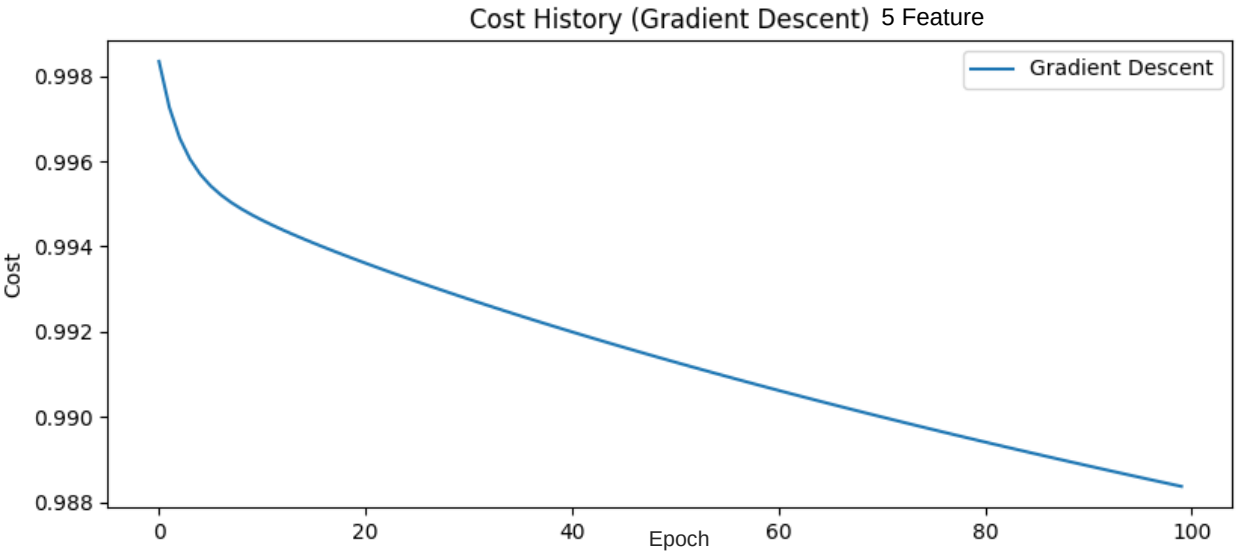




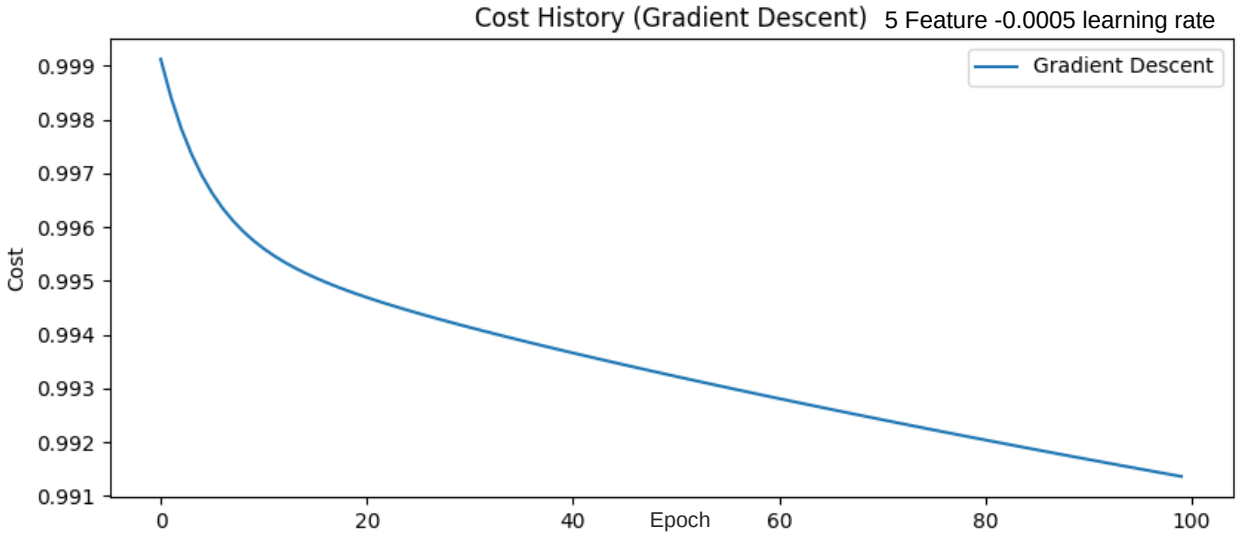
Gradient Descent 3,4,5 Feature Cost Results

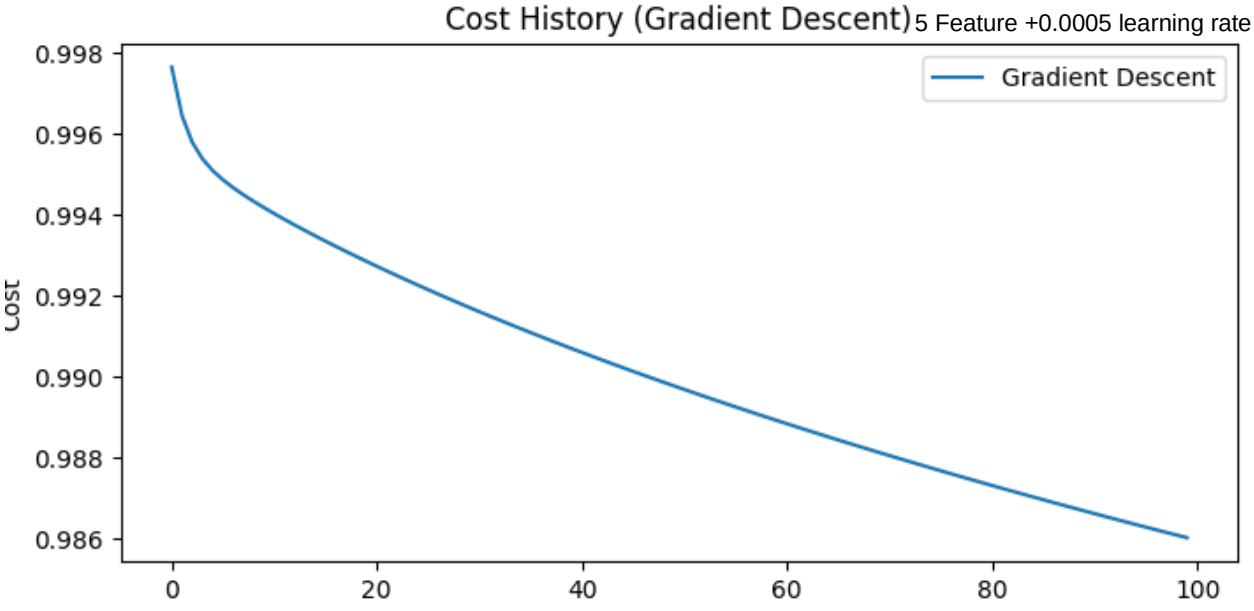




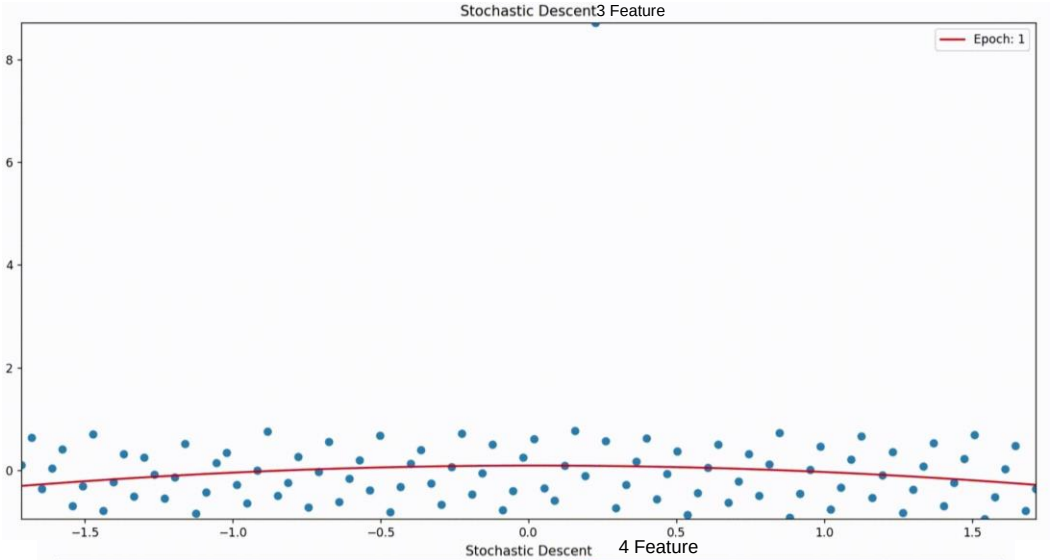


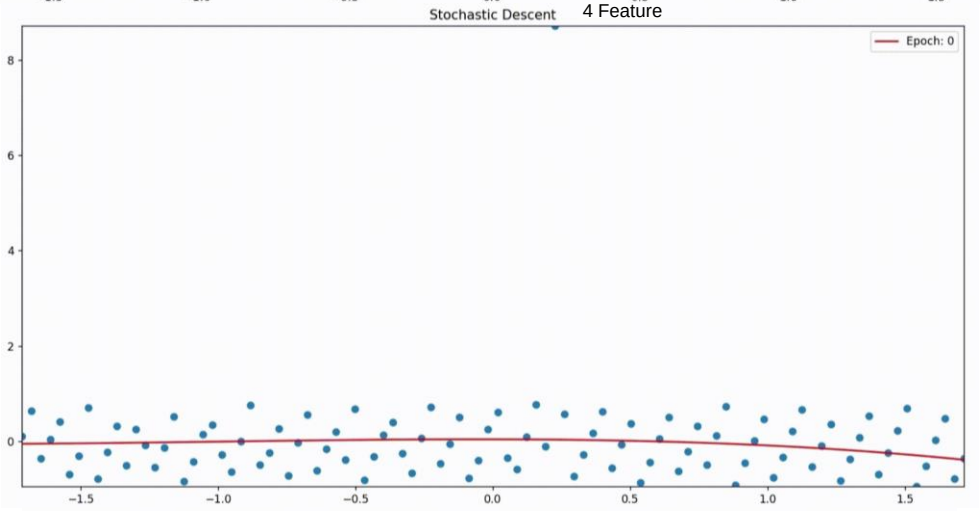
Gradient Descent 5 Feature -/+ 0.005 Learning Rate Results

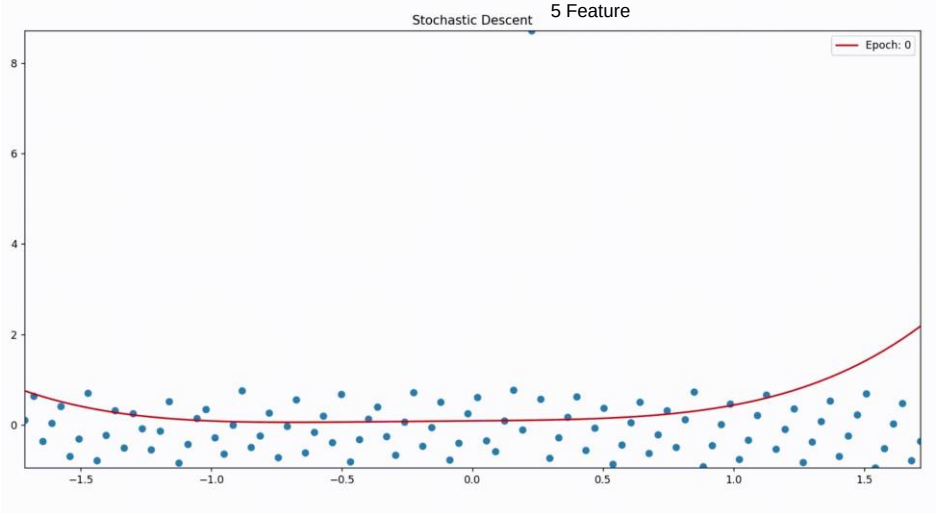




Stochastic Descent 3,4,5 Feature Runs



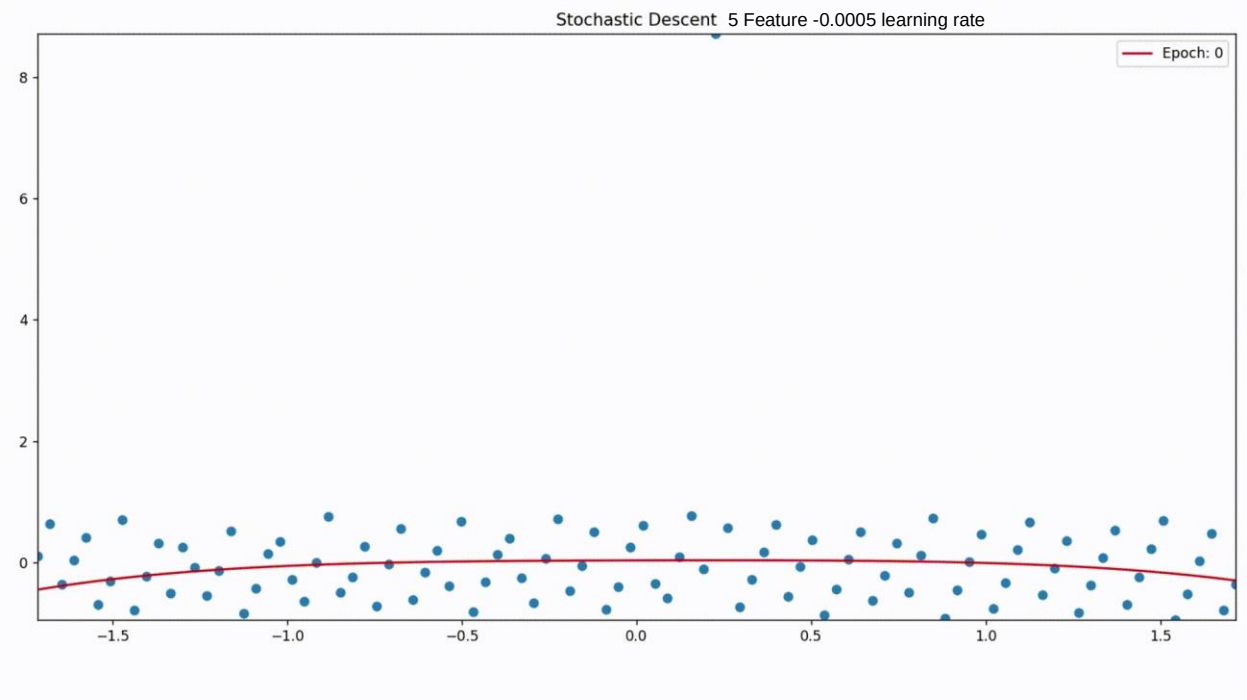


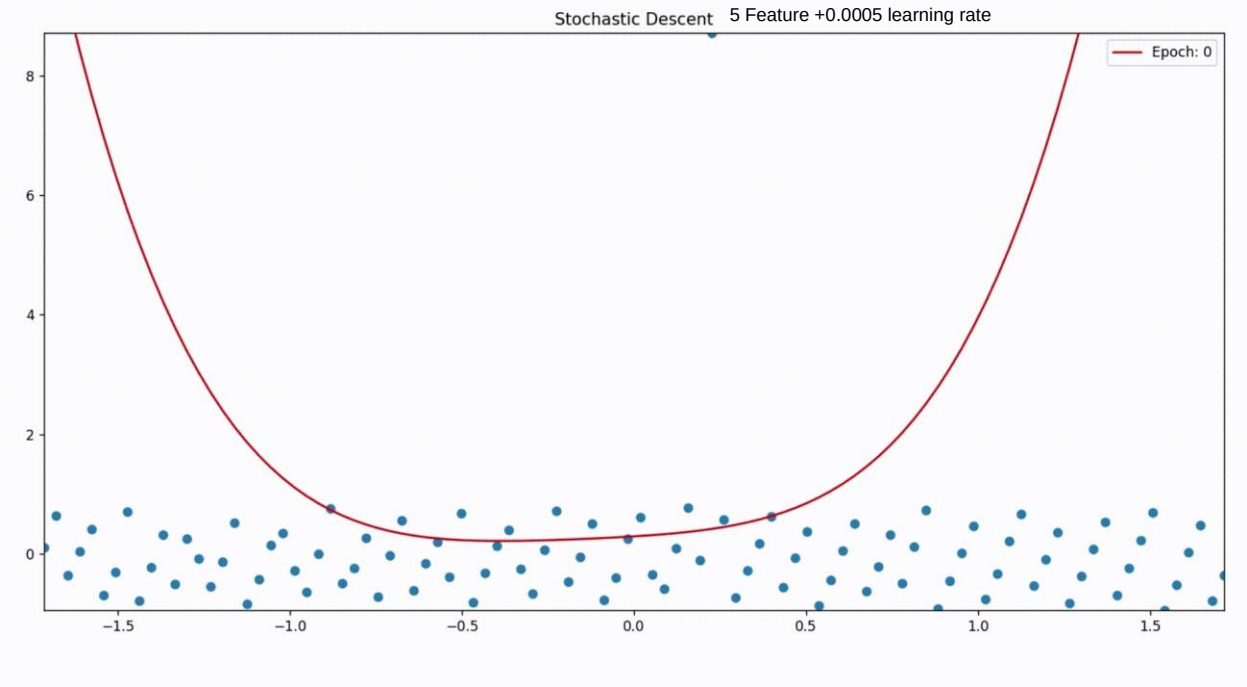


The feature 5 graph displayed some unusual action unalike the prior two feature graphs. The beginning and end seem to be curving up as the epochs go on. It could be because we are using a polynomial function of degree 4. Higher-degree polynomials can sometimes lead to this kind of behavior, where the function fits the data well in some

regions but performs poorly at the extremes. This is especially true if the degree of the polynomial is higher than necessary for the underlying data.

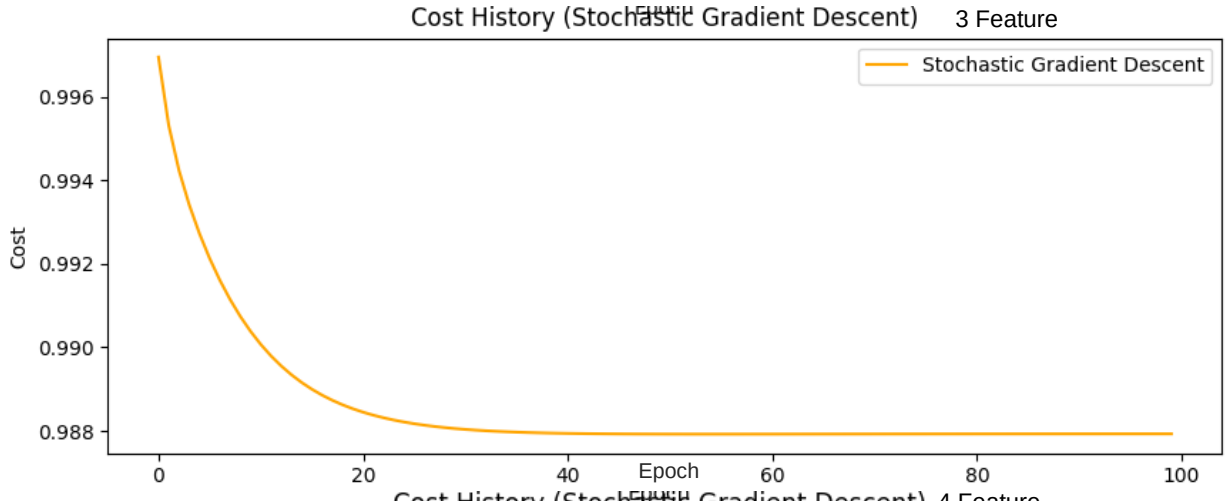
Stochastic Descent 5 Feature -/+ 0.005 Learning Rate Runs



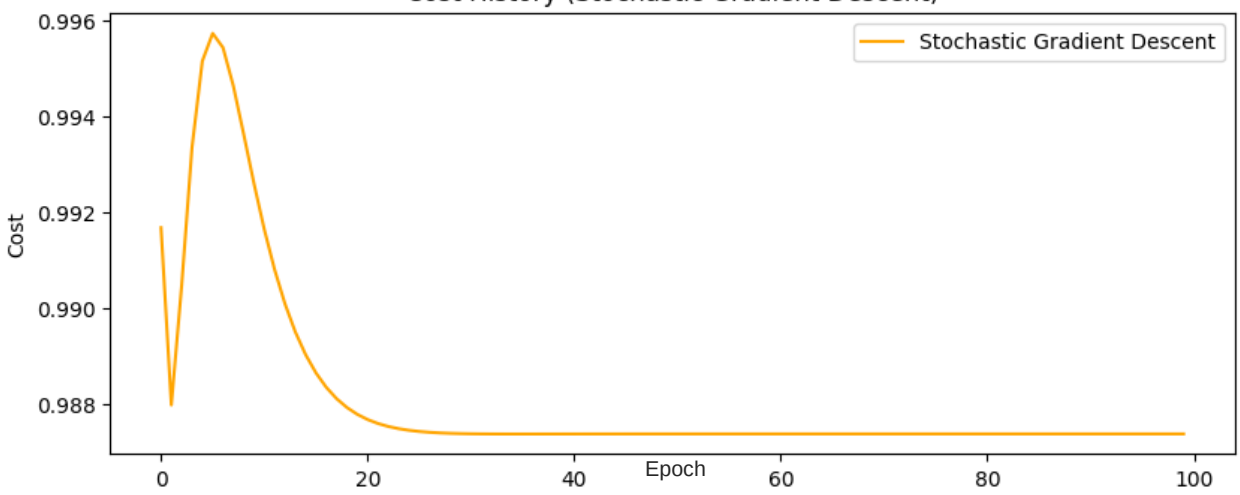


Very interesting result here. I have run it multiple times to receive a similar result. Whenever you increase the learning rate, the model parameters are being updated in larger steps. Usually, when a model that was already reaching it's optimal solution prior to the buff in learning rate, the model can overshoot the minimum of the cost function causing an increase in cost.

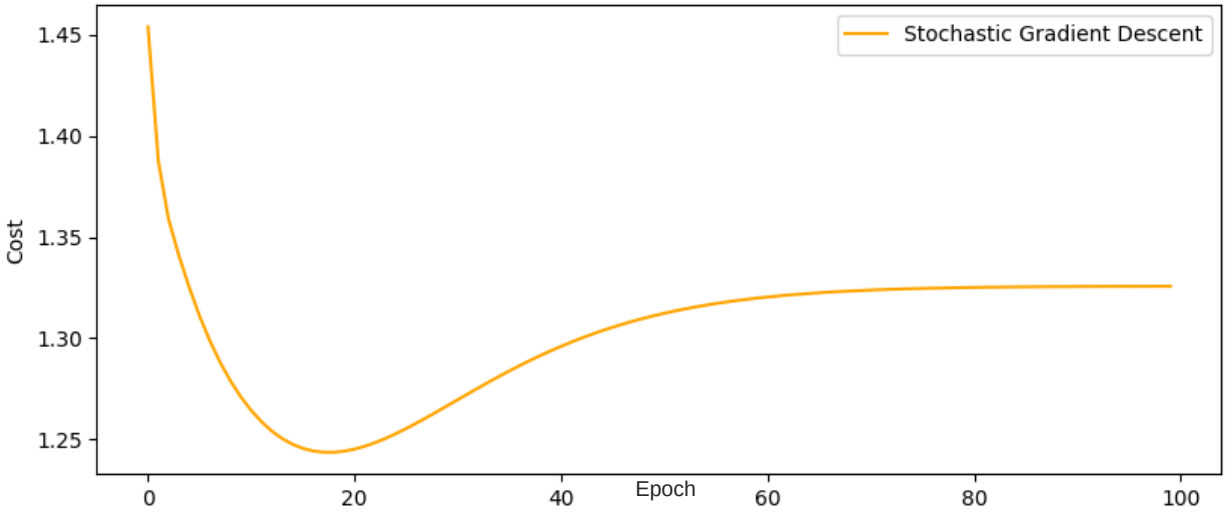
Stochastic Descent 3,4,5 Feature Cost Results



3 Feature

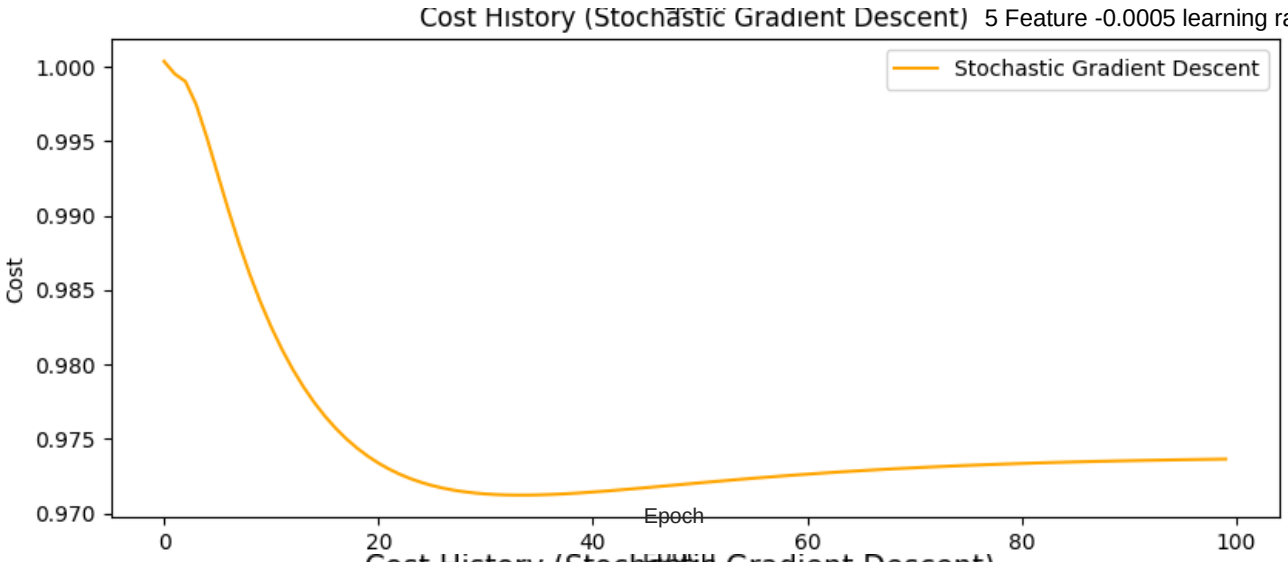


4 Feature

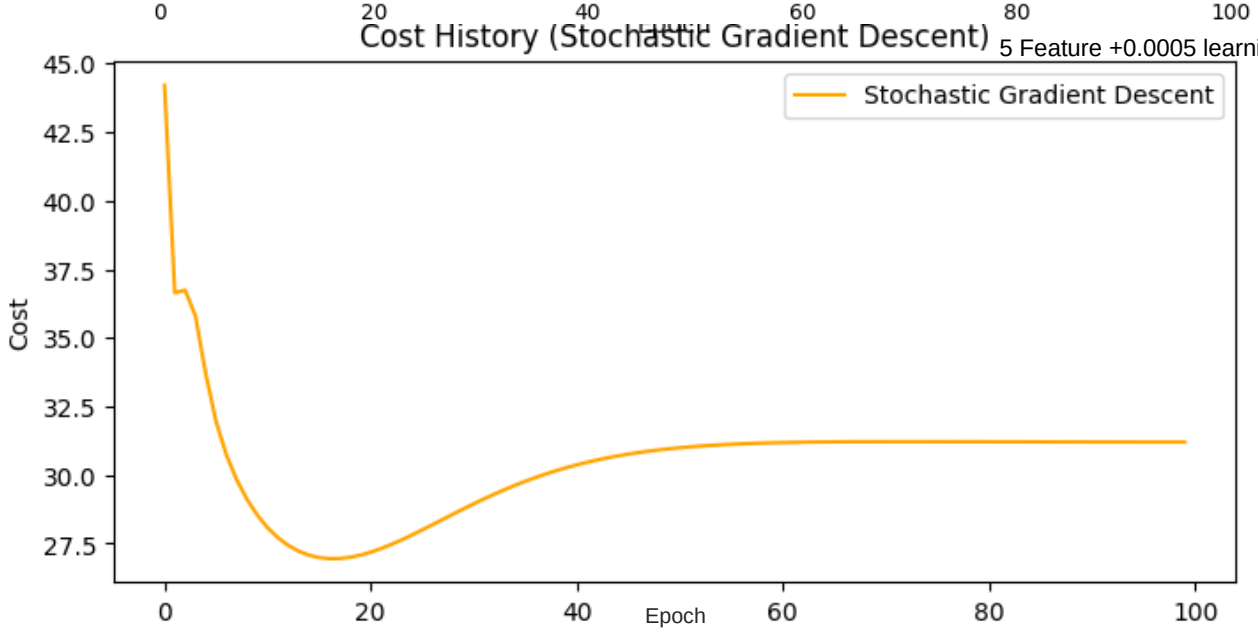


5 Feature

Stochastic Descent 5 Feature -/+ 0.005 Learning Rate Results

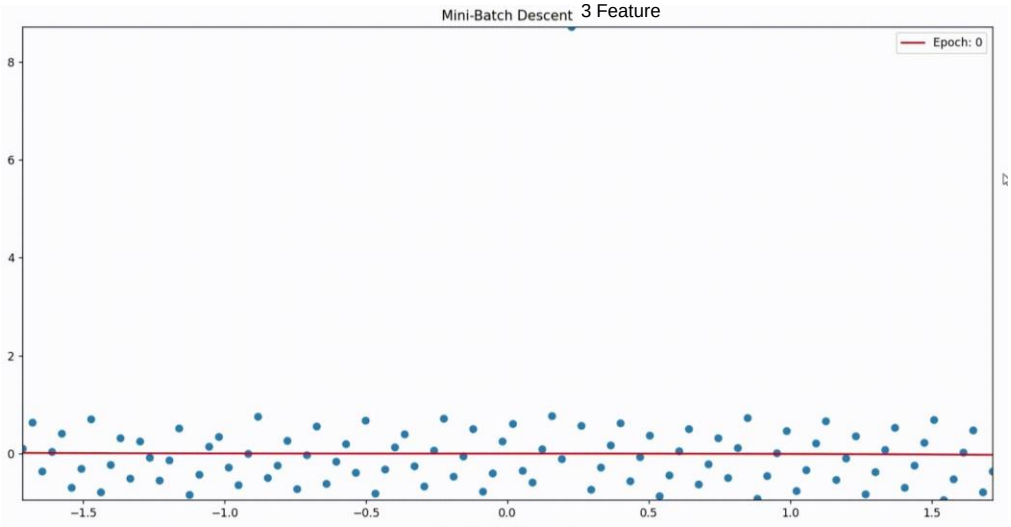


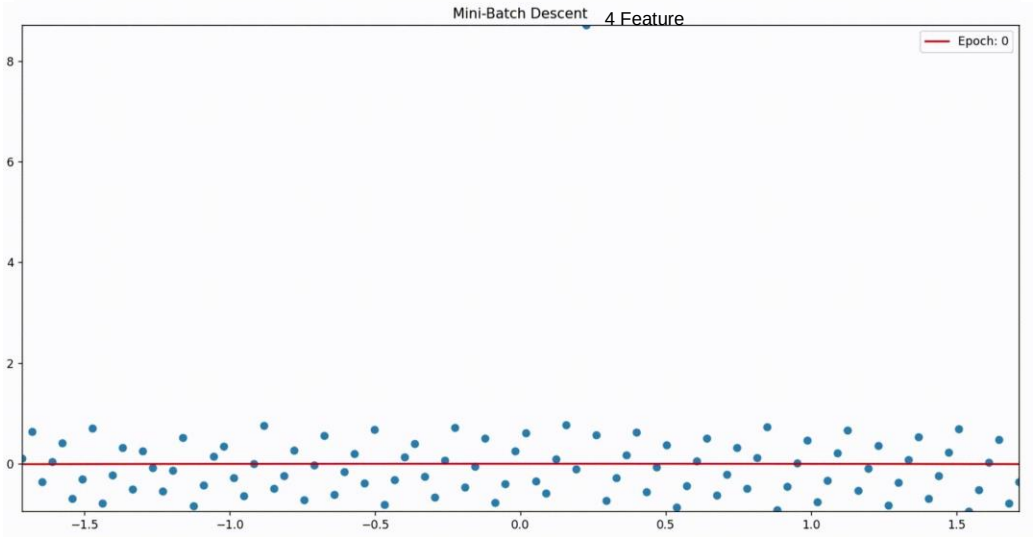
5 Feature -0.0005 learning rate

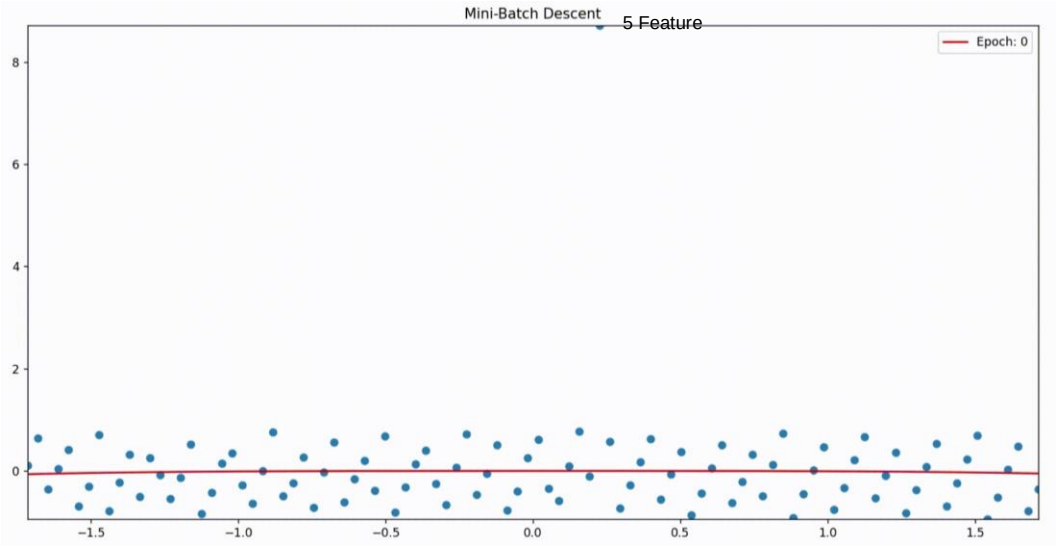


5 Feature +0.0005 learning rate

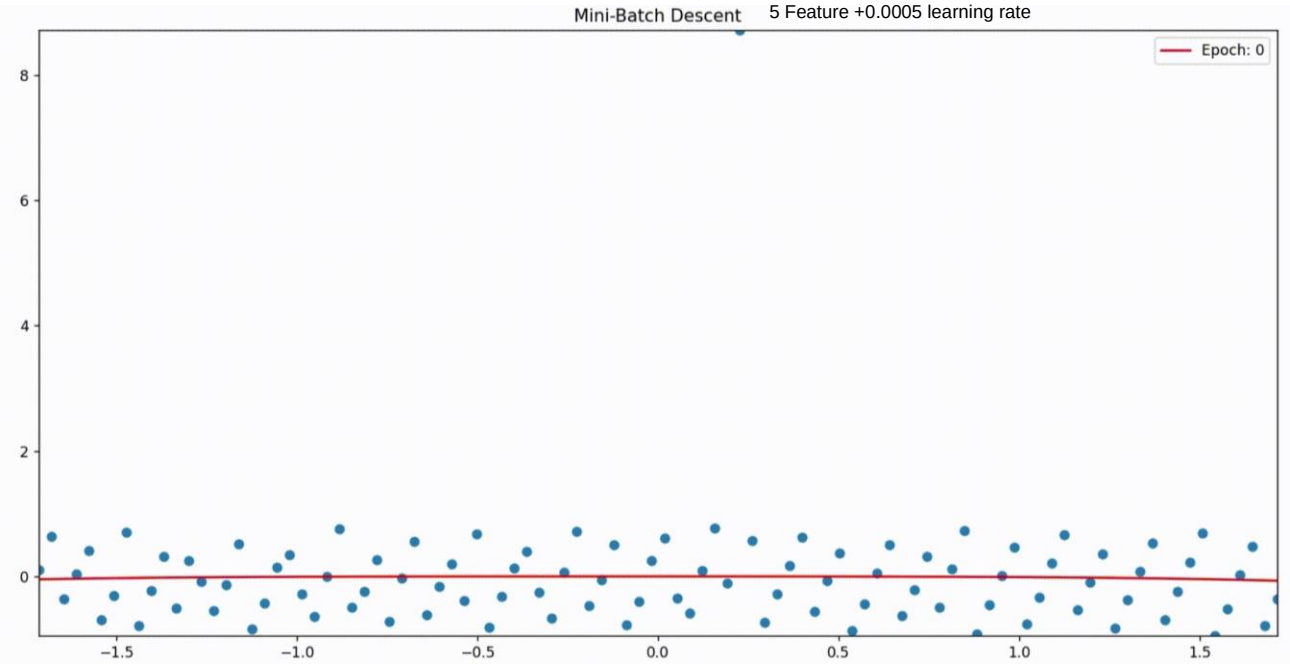
Mini-Batch Descent 3,4,5 Feature Runs

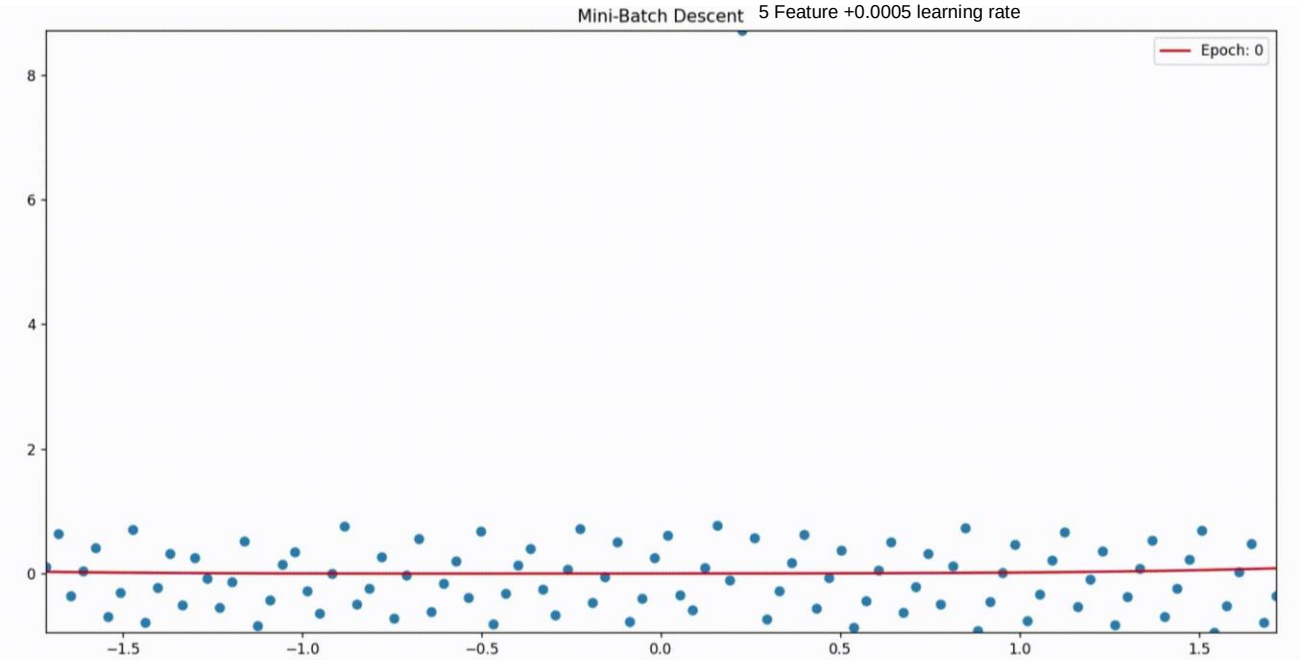




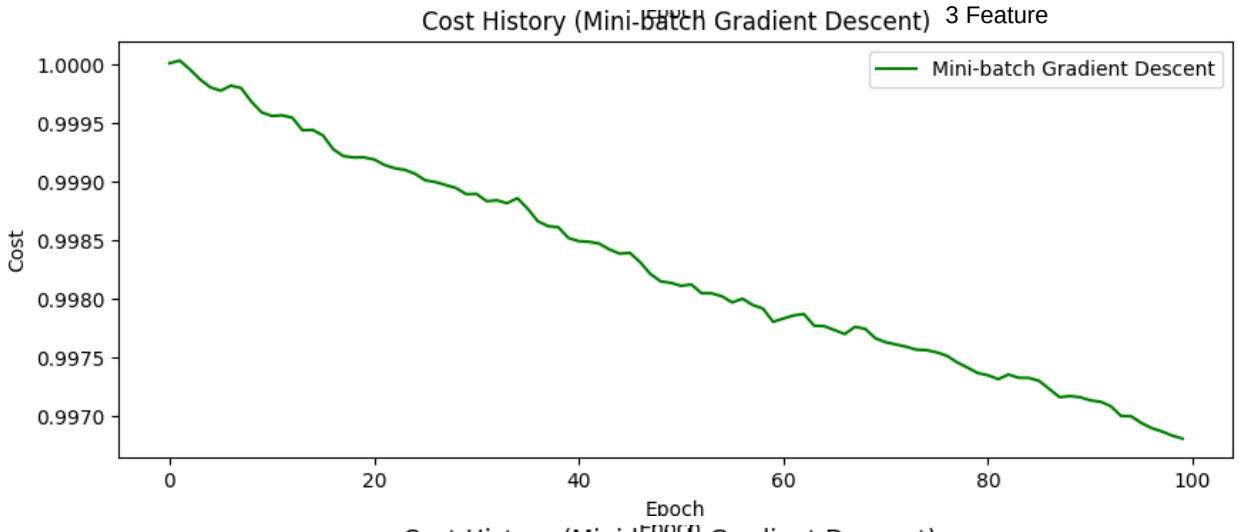


Mini-Batch Descent 5 Feature -/+ 0.005 Learning Rate Runs

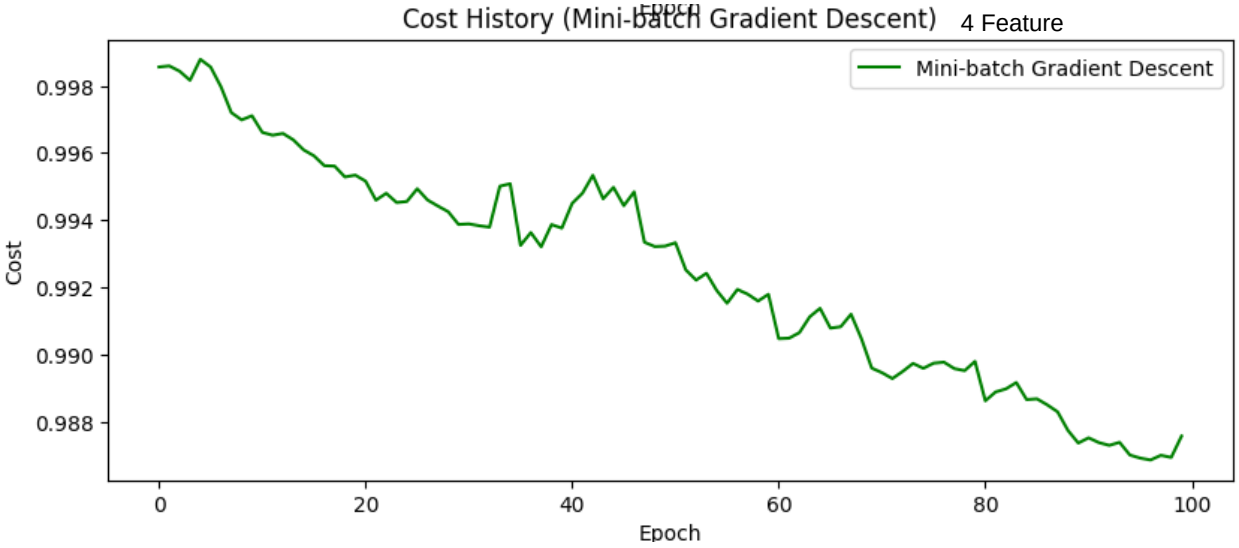




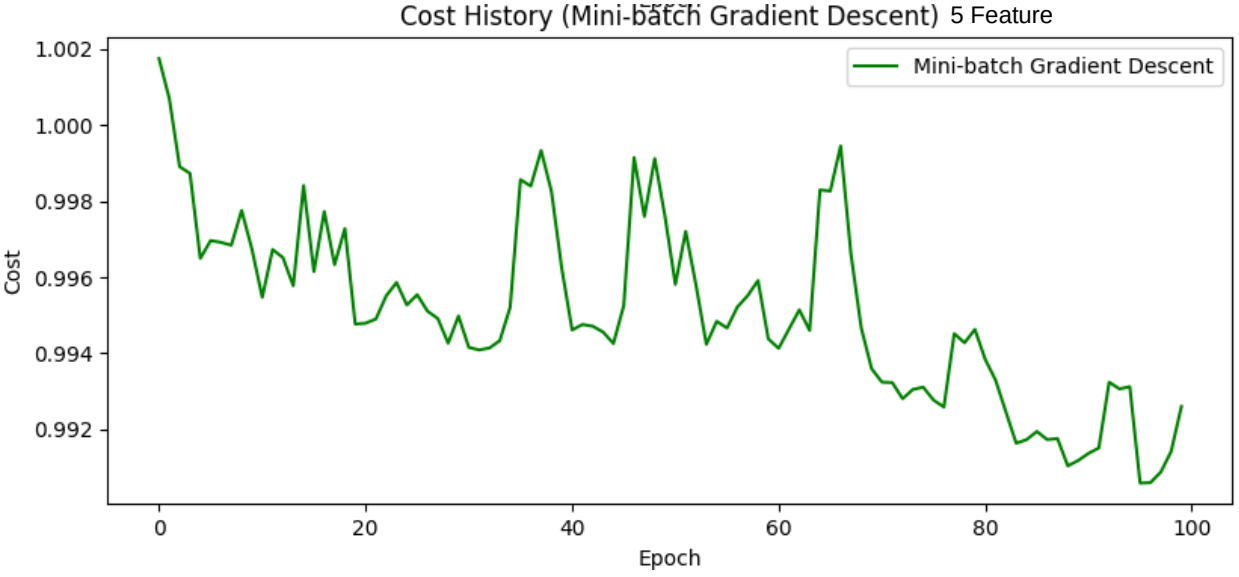
Mini-Batch Descent 3,4,5 Feature Cost Results



3 Feature

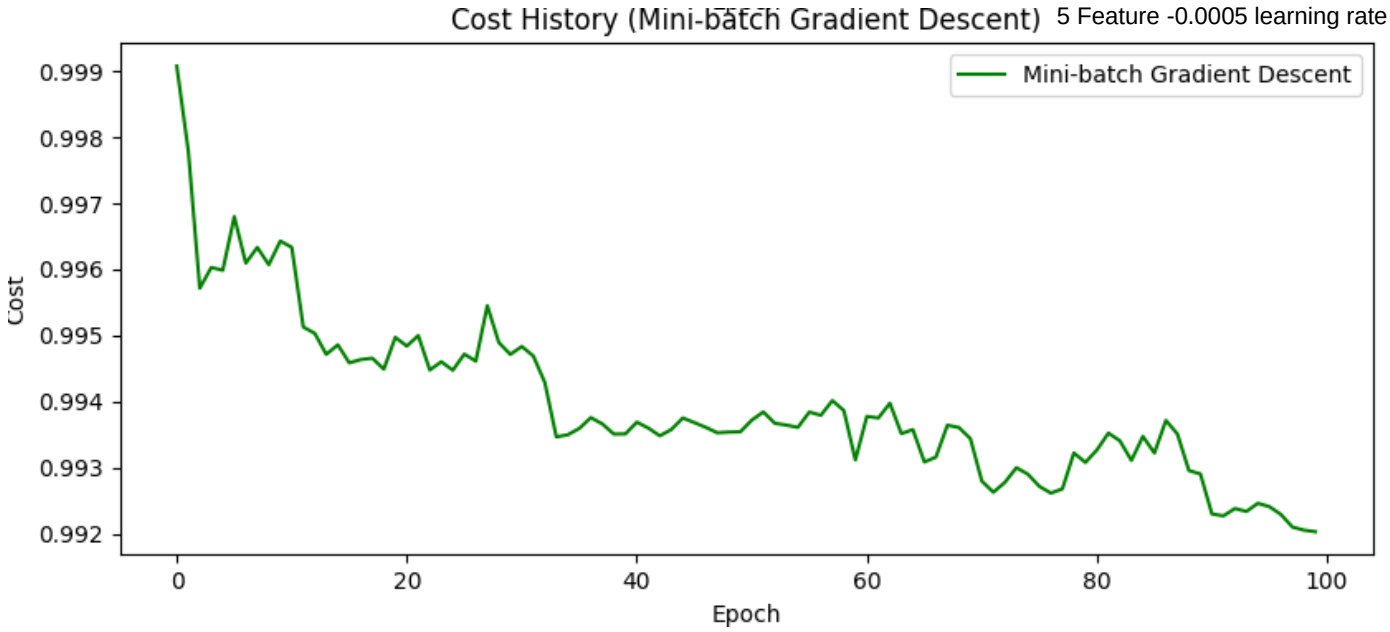


4 Feature

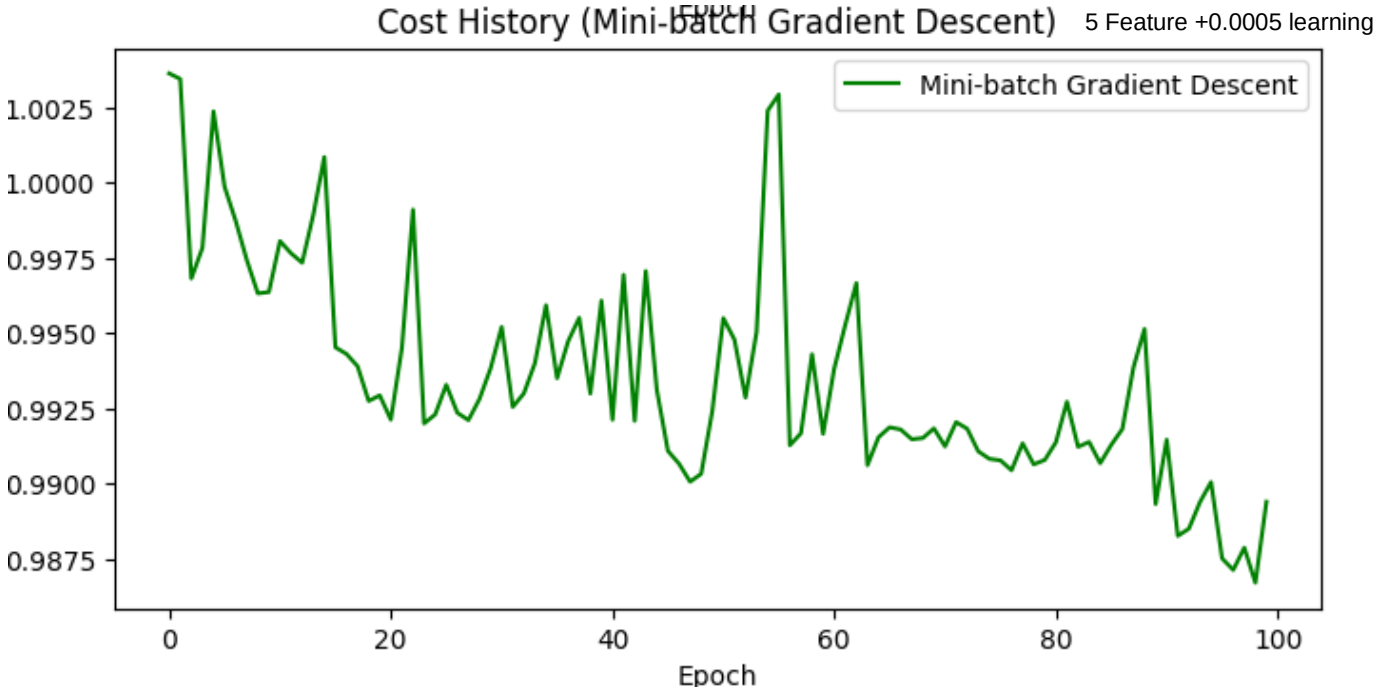


5 Feature

Mini-Batch Descent 5 Feature -/+ 0.005 Learning Rate Results



5 Feature -0.0005 learning rate



5 Feature +0.0005 learning

Works Cited:

"Normalization | Machine Learning | Google for Developers." *Google*, 18 June 2022, developers.google.com/machine-learning/data-prep/transform/normalization.