# ML HW1

## 1 Linear Algebra Review

Machine Learning HW1

1.

1.1

$$2x_1 + 2x_2 + 3x_3 = 1$$
$$-3x_1 + 6x_2 + 3x_3 = 6$$
$$5x_1 - 4x_2 = -5$$
$$-\ 5x_1 - 5x_2 = -5$$
$$x_2 = 0$$

$$5x_1 = -5 \Rightarrow x_1 = -1$$

$$1 + x_3 = 2 \Rightarrow x_3 = 1$$

$x_1 = -1, x_2 = 0, x_3 = 1$ //

1.2

$$\begin{bmatrix} 2 & 2 & 3 \\ 1 & -1 & 0 \\ -1 & 2 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \\ 2 \end{bmatrix}$$

1.3 3

1.4

$$\left[\begin{array}{ccc|ccc} 2 & 2 & 3 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 & 1 & 0 \\ -1 & 2 & 1 & 0 & 0 & 1 \end{array}\right]$$

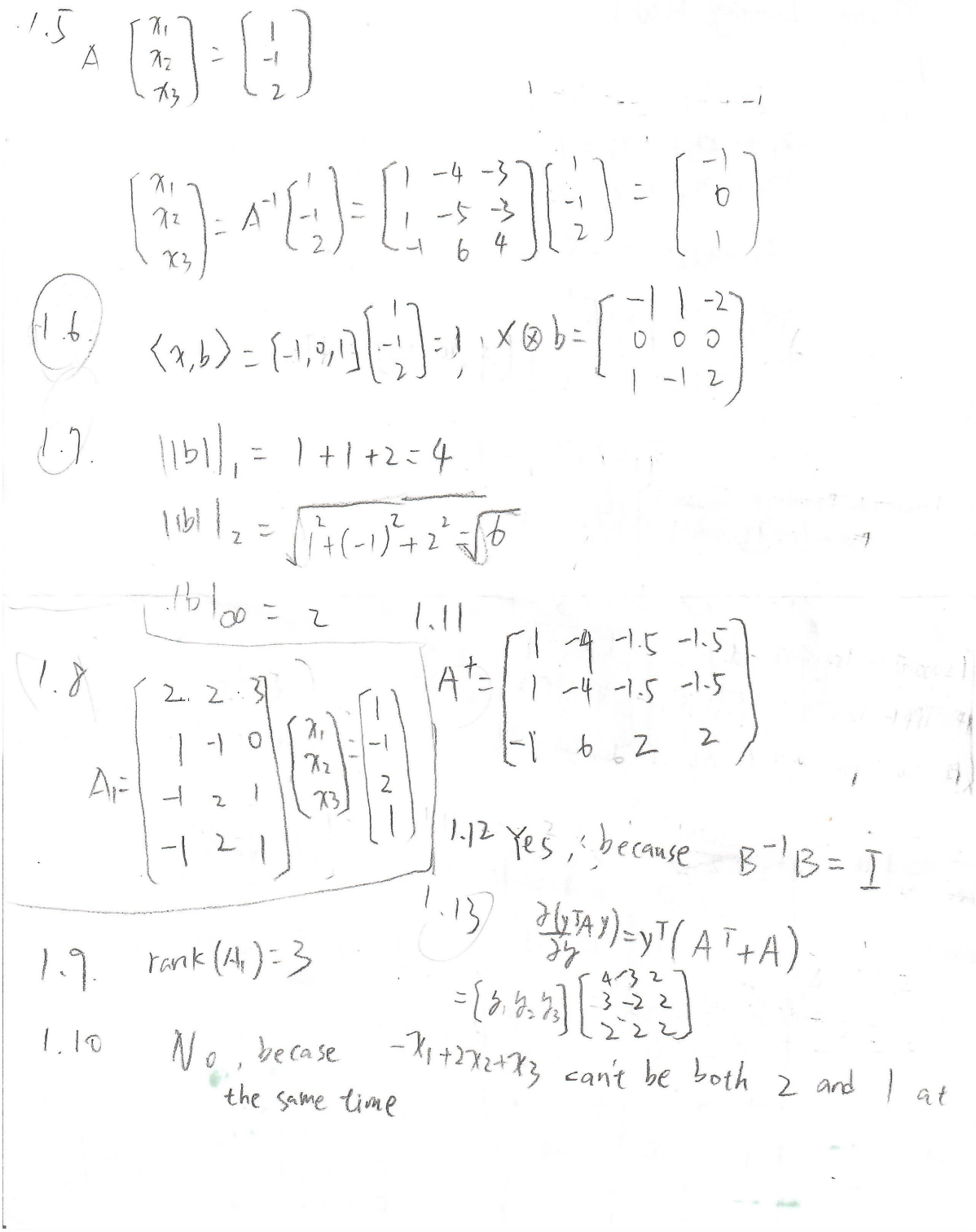
$$= \left[\begin{array}{ccc|ccc} 5 & -4 & 0 & 1 & 0 & -3 \\ 1 & -1 & 0 & 0 & 1 & 0 \\ -1 & 2 & 1 & 0 & 0 & 1 \end{array}\right]$$

$$= \left[\begin{array}{ccc|ccc} 1 & 0 & 0 & 1 & -4 & -3 \\ -1 & -1 & 0 & 0 & 1 & 0 \\ -1 & 2 & 1 & 0 & 0 & 1 \end{array}\right] = \left[\begin{array}{ccc|ccc} 1 & 0 & 0 & 1 & -4 & -3 \\ 0 & 1 & 0 & 1 & -5 & -3 \\ 0 & 2 & 1 & 1 & -4 & -2 \end{array}\right] = \left[\begin{array}{ccc|ccc} 1 & 0 & 0 & 1 & -4 & -3 \\ 0 & 1 & 0 & 1 & -5 & -3 \\ 0 & 0 & 1 & -1 & 6 & 4 \end{array}\right]$$

$$A^{-1} = \begin{bmatrix} 1 & -4 & -3 \\ 1 & -5 & -3 \\ -1 & 6 & 4 \end{bmatrix}$$

$$\det(A) = -2 + 6 - 3 - 0 - 2 = -1$$

//




## 2 Linear Regression Model Fitting (Programming)

```python
# Machine Learning HW1
import matplotlib.pyplot as plt
import numpy as np
# more imports


# Parse the file and return 2 numpy arrays


def load_data_set(filename):
    arr = np.loadtxt(filename)
    x, y = np.split(arr, [-1], axis=1)
#     plt.plot(x[:, 1], y, '.')
```

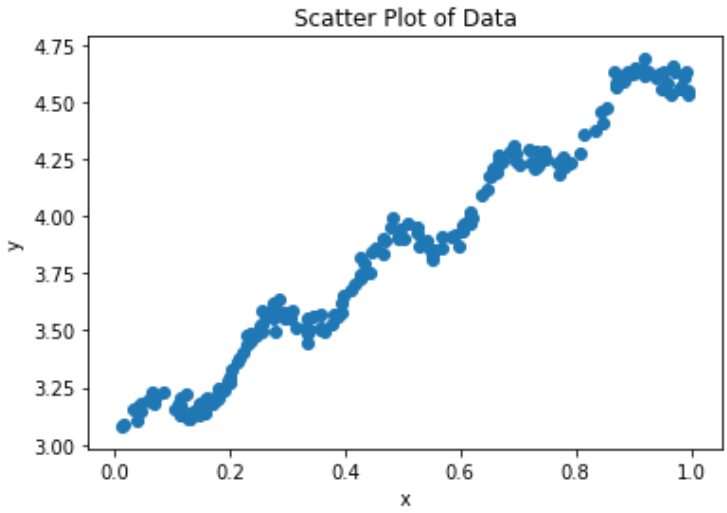
```
#     plt.show()
    return x, y

# Find theta using the normal equation


def normal_equation(x, y):
    x_t = np.transpose(x)
    theta = np.dot(np.dot(np.linalg.inv(np.dot(x_t, x)), x_t), y)
    return theta

# Find thetas using stochiastic gradient descent
# Don't forget to shuffle


def stochiastic_gradient_descent(x, y, learning_rate, num_iterations):
    # your code
    thetas = []
    theta = np.array([
        [1],
        [1]
    ])
    for _ in range(num_iterations):
        x, y = unison_shuffled_copies(x, y)
        for row, label in zip(x, y):
            theta = theta - learning_rate * \
                row[:, np.newaxis] * (np.dot(row, theta) - label)
        thetas.append(theta)
        # print(thetas)
    return thetas

# Find thetas using gradient descent


def gradient_descent(x, y, learning_rate, num_iterations):
    thetas = []
    theta = np.array([
        [1],
        [1]
    ])
    for _ in range(num_iterations):
        gradient_sum = np.array([
            [0.0],
            [0.0]
        ])
        for row, label in zip(x, y):
            gradient_sum = gradient_sum - learning_rate * \
                row[:, np.newaxis] * (np.dot(row, theta) - label)
        theta = theta + gradient_sum
        thetas.append(theta)
    return thetas


def unison_shuffled_copies(a, b):
    assert len(a) == len(b)
    p = np.random.permutation(len(a))
    return a[p], b[p]

# Find thetas using minibatch gradient descent
# Don't forget to shuffle


def minibatch_gradient_descent(x, y, learning_rate, num_iterations, batch_size):
    thetas = []
    theta = np.array([
        [1],
        [1]
    ])
    gradient_sum = np.array([
```

```
            [0],
            [0]
        ])
        count = 0
        for _ in range(num_iterations):
            x, y = unison_shuffled_copies(x, y)
            # print(x, y)
            for row, label in zip(x, y):
                count += 1
                gradient_sum = gradient_sum - learning_rate * \
                    row[:, np.newaxis] * (np.dot(row, theta) - label)
                if count == batch_size:
                    count = 0
                    theta = theta + gradient_sum
                    gradient_sum = np.array([
                        [0],
                        [0]
                    ])
                    # print(theta)
            theta = theta + gradient_sum
            thetas.append(theta)
            # print(thetas)
        return thetas

    # Given an array of x and theta predict y


    def predict(x, theta):
        # your code
        y_predict = np.dot(x, theta)
        return y_predict

    # Given an array of y and y_predict return loss


    def get_loss(y, y_predict):
        # your code
        diff = y - y_predict
        loss = np.dot(diff.T, diff) / len(y)
        return loss

    # Given a list of thetas one per epoch
    # this creates a plot of epoch vs training error


    def plot_training_errors(x, y, thetas, title):
        accuracies = []
        epochs = []
        losses = []
        epoch_num = 1
        for theta in thetas:
            losses.append(get_loss(y, predict(x, theta)))
            epochs.append(epoch_num)
            epoch_num += 1
        plt.scatter(epochs, losses)
        plt.xlabel("epoch")
        plt.ylabel("loss")
        plt.title(title)
        plt.show()

    # Given x, y, y_predict and title,
    # this creates a plot


    def plot(x, y, theta, title):
        # plot
        y_predict = predict(x, theta)
        plt.scatter(x[:, 1], y)
        plt.plot(x[:, 1], y_predict)
```

```
152       plt.xlabel("x")
153       plt.ylabel("y")
154       plt.title(title)
155       plt.show()
```

## Data Set

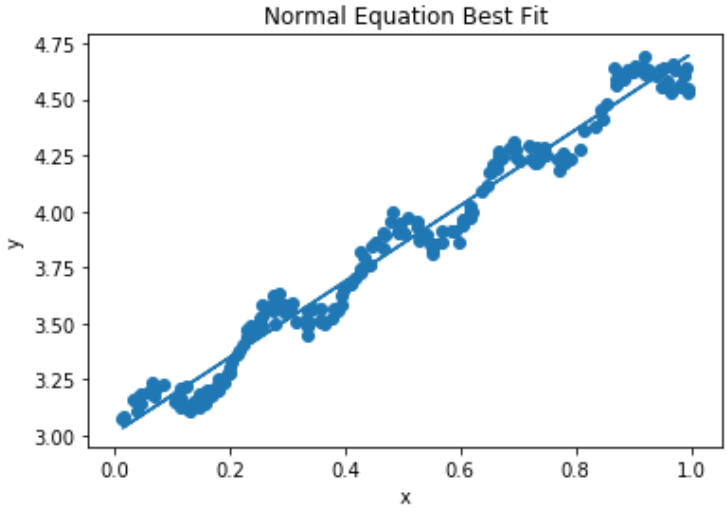
```
1  x, y = load_data_set('regression-data.txt')
2  # plot
3  plt.scatter(x[:, 1], y)
4  plt.xlabel("x")
5  plt.ylabel("y")
6  plt.title("Scatter Plot of Data")
7  plt.show()
```




## Normal Equation

```
1  theta = normal_equation(x, y)
2  print('theta={}'.format(theta))
3  plot(x, y, theta, "Normal Equation Best Fit")
```

```
1  theta=[[3.00774324]
2   [1.69532264]]
```

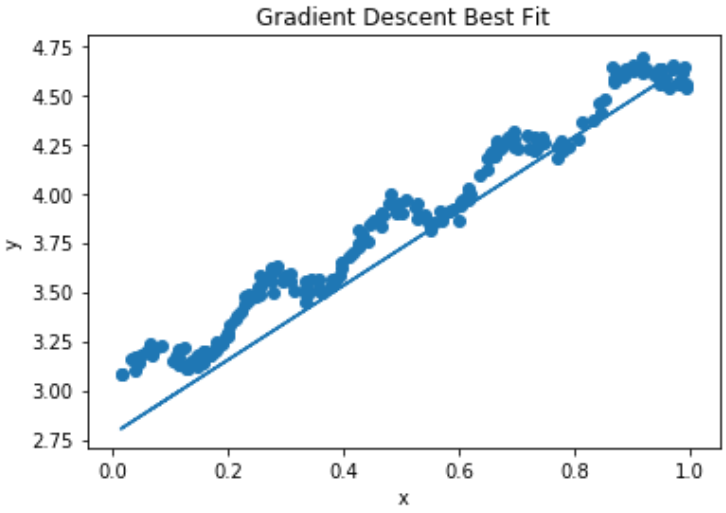


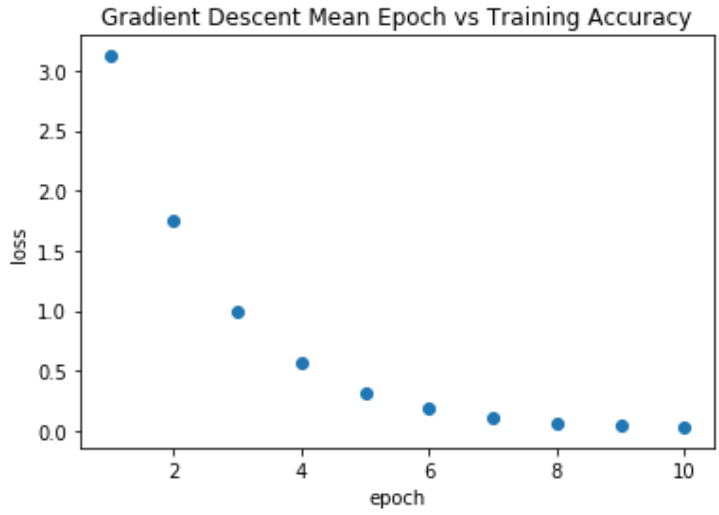

## Gradient Descent

## Results

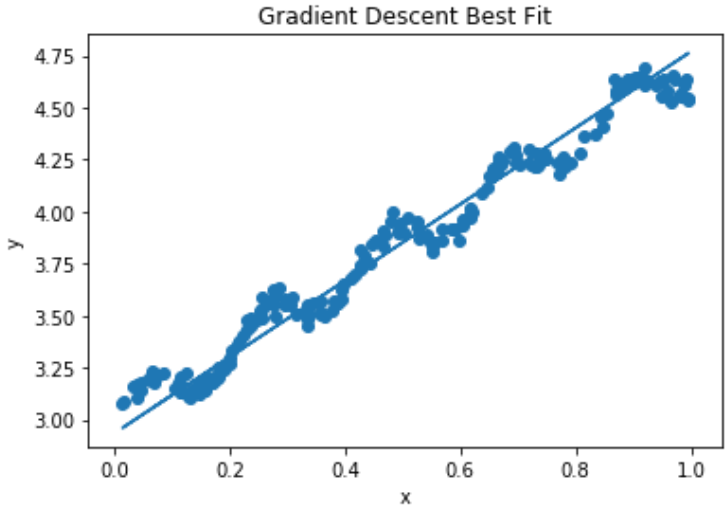
```
n_iteration_list = [10, 50]
learning_rate_list = [0.001, 0.005, 0.01, 0.05, 0.1, 0.3]
for learning_rate in learning_rate_list:
    for n_iter in n_iteration_list:
        print('learning rate: {}, number of iterations: {}'.format(learning_rate, n_iter))
        thetas = gradient_descent(x, y, learning_rate, n_iter)
        plot(x, y, thetas[-1], "Gradient Descent Best Fit")
        plot_training_errors(x, y, thetas, "Gradient Descent Mean Epoch vs Training Accuracy")
```

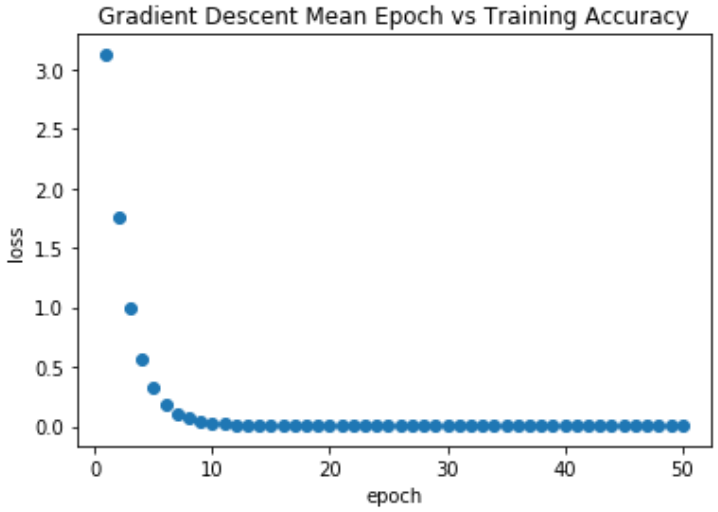
```
learning rate: 0.001, number of iterations: 10
```



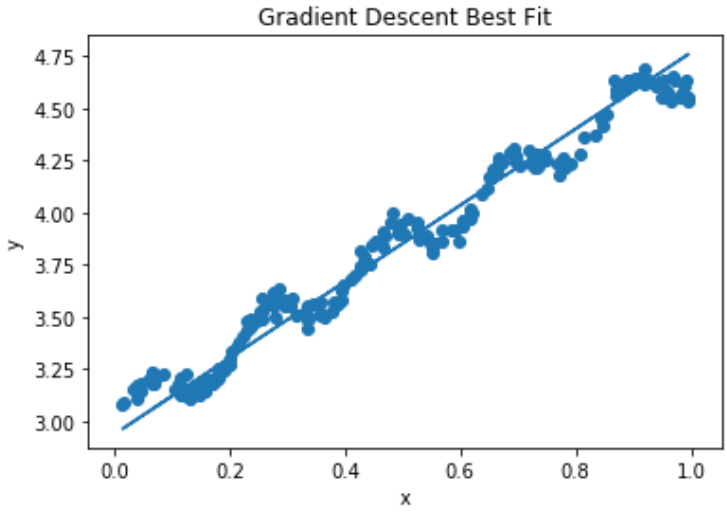



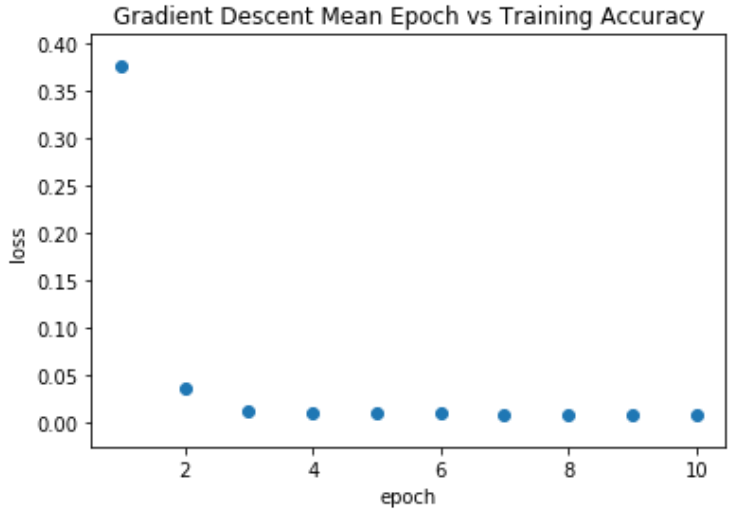

```
learning rate: 0.001, number of iterations: 50
```



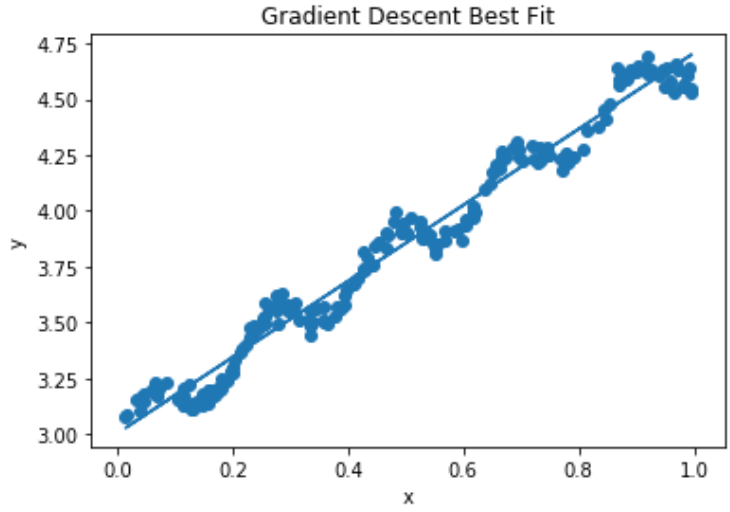



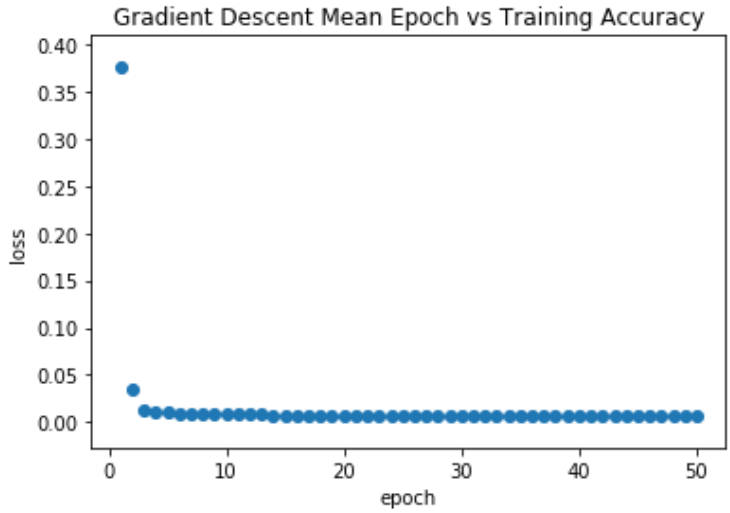

```
learning rate: 0.005, number of iterations: 10
```



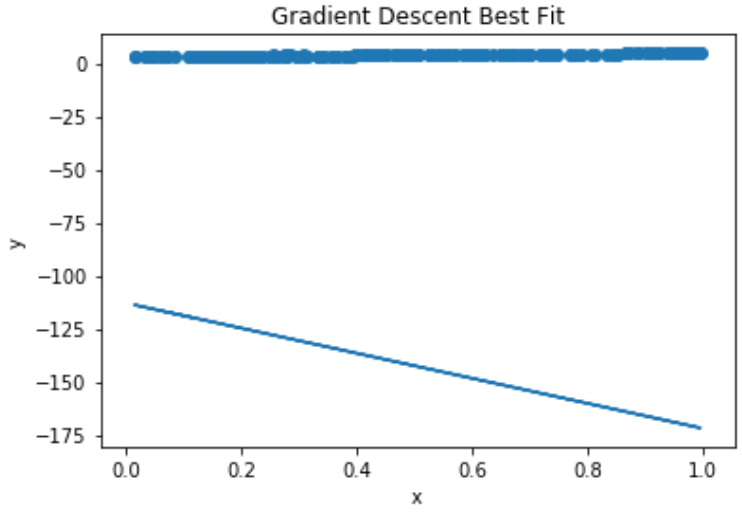


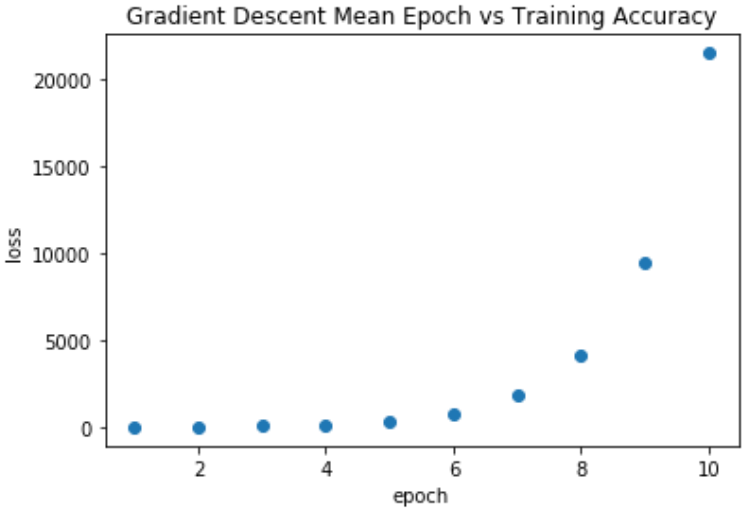

```
learning rate: 0.005, number of iterations: 50
```



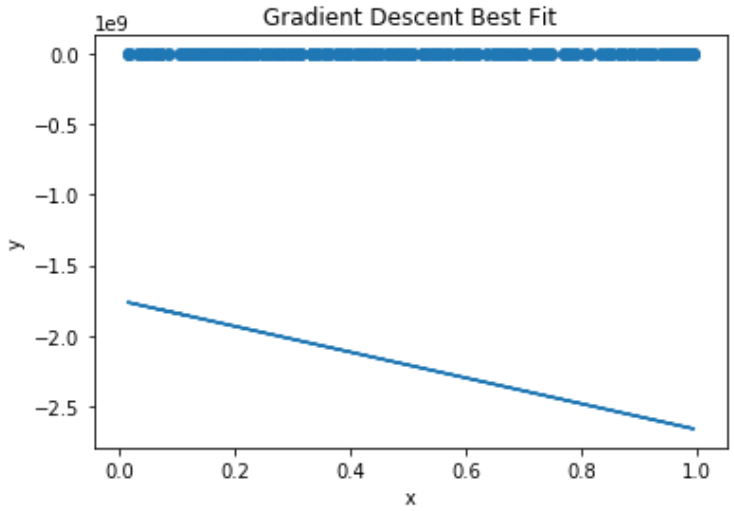



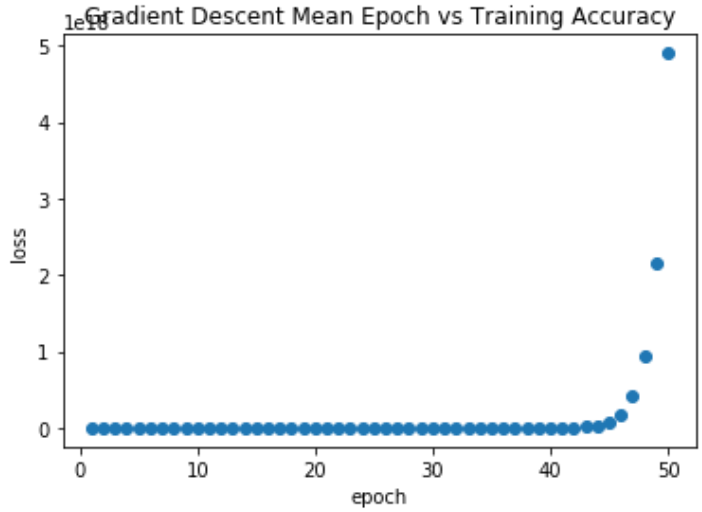

```
learning rate: 0.01, number of iterations: 10
```



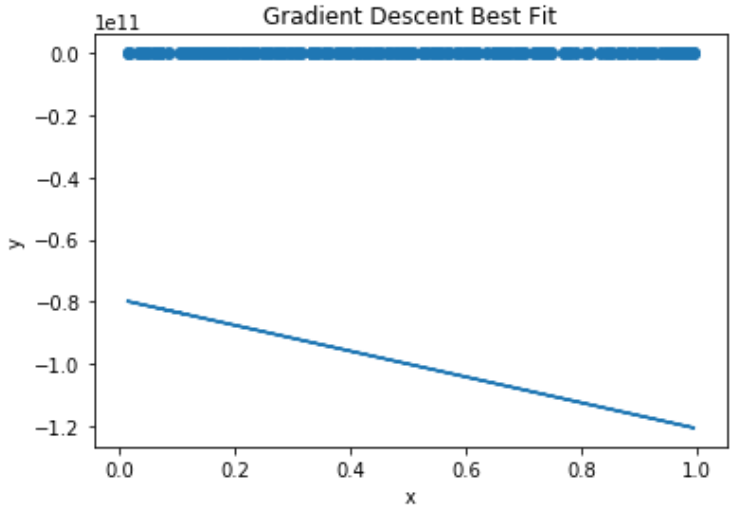



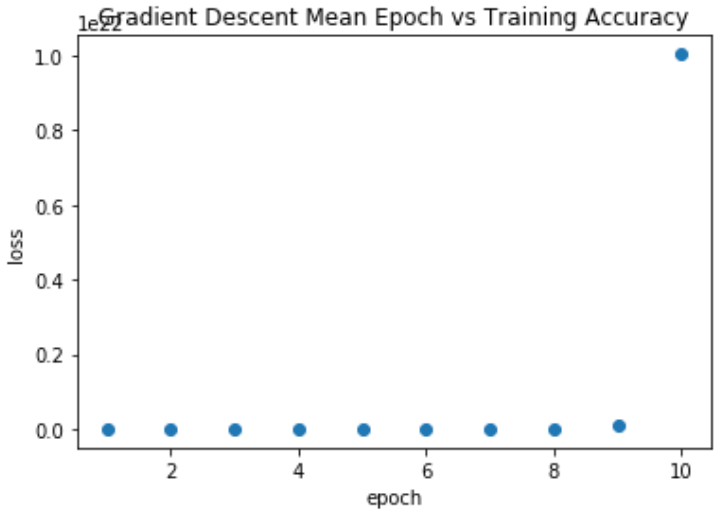

```
learning rate: 0.01, number of iterations: 50
```



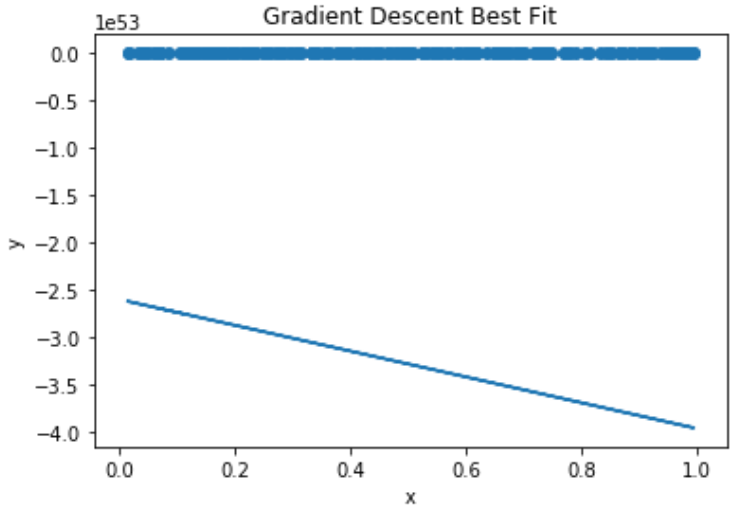


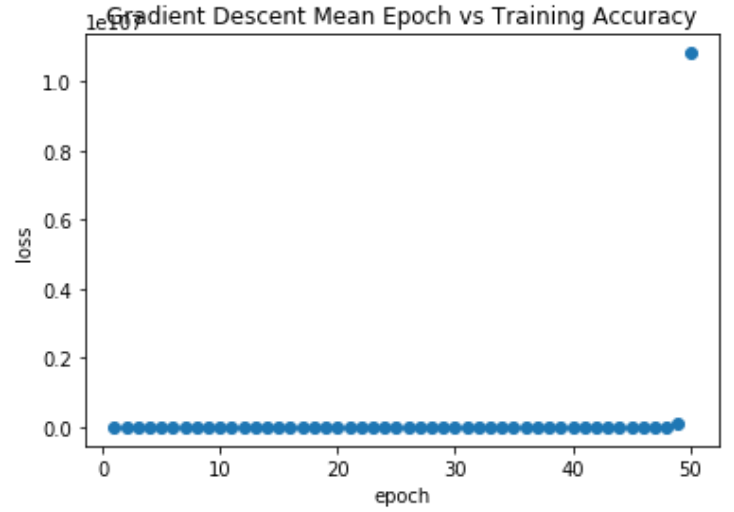

```
1  learning rate: 0.05, number of iterations: 10
```



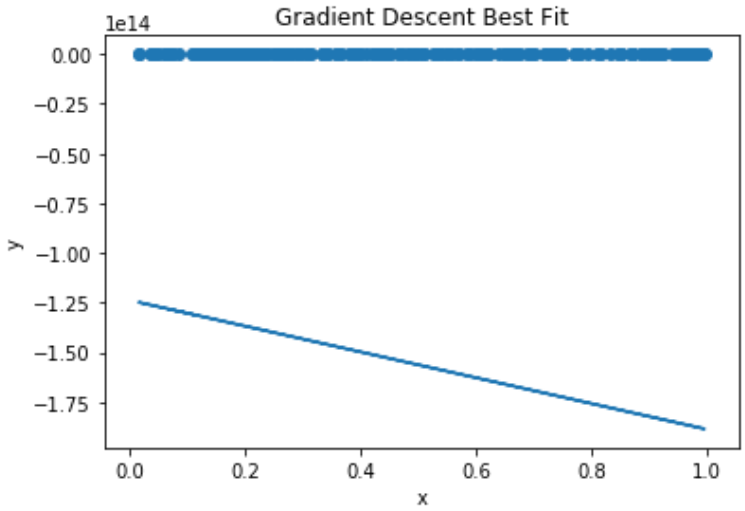



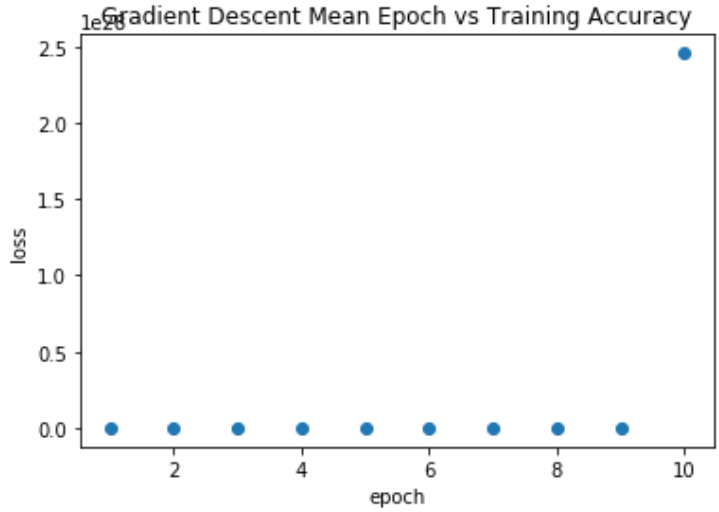

```
1  learning rate: 0.05, number of iterations: 50
```



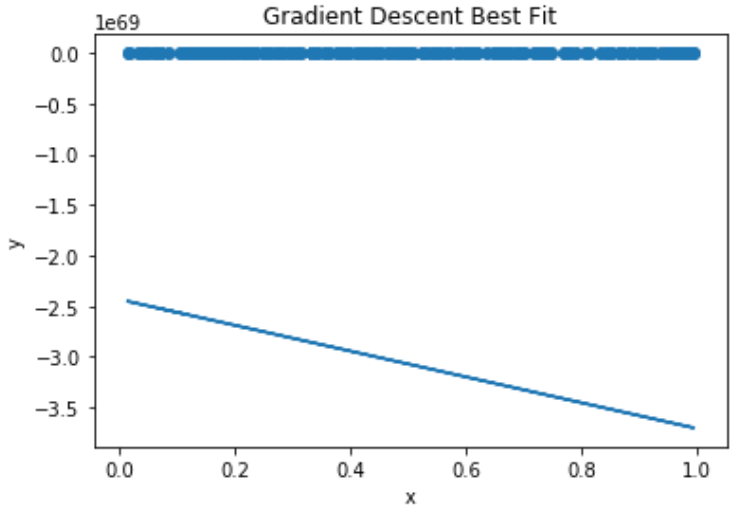



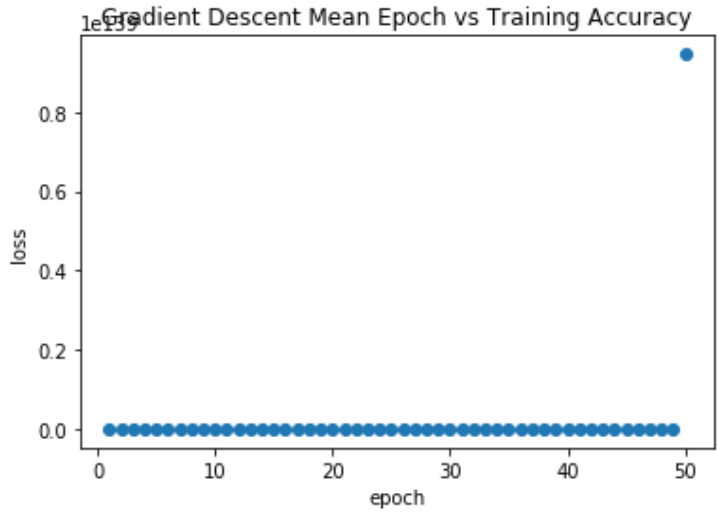

```
learning rate: 0.1, number of iterations: 10
```



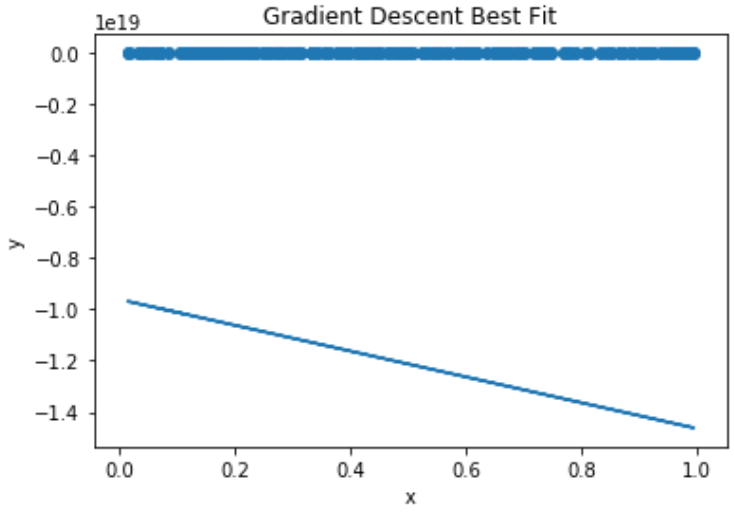


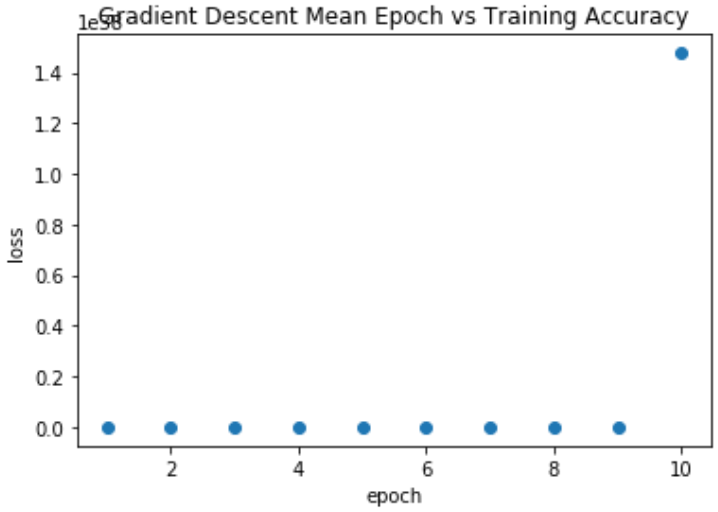

```
1  learning rate: 0.1, number of iterations: 50
```



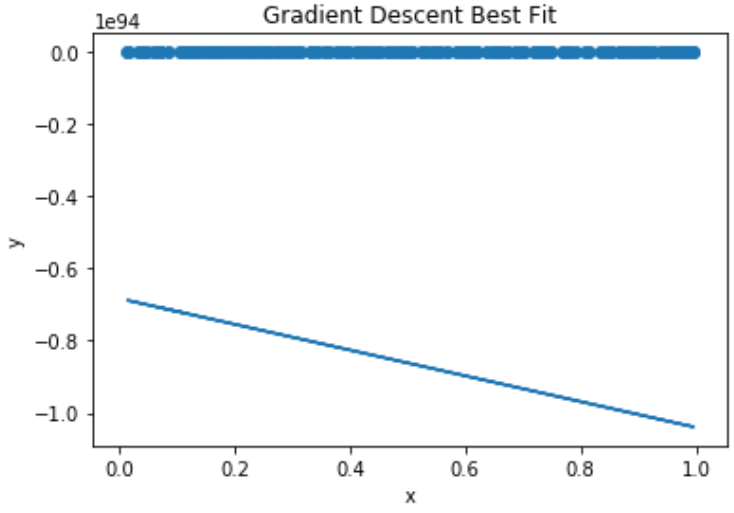



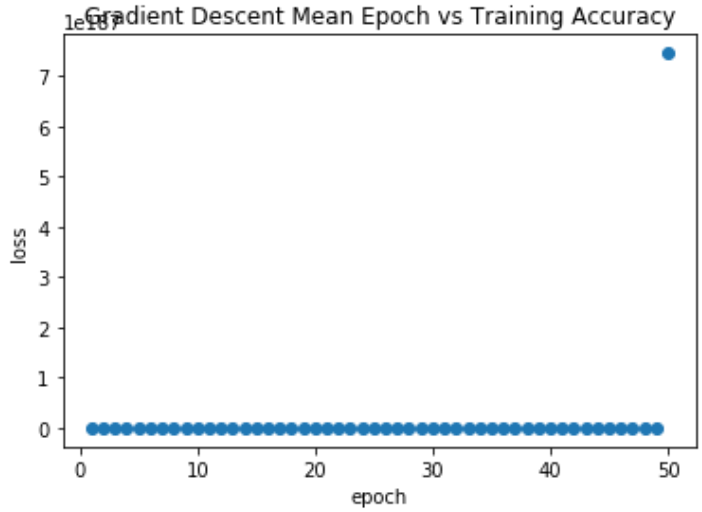

```
1  learning rate: 0.3, number of iterations: 10
```




```
learning rate: 0.3, number of iterations: 50
```






### Observation

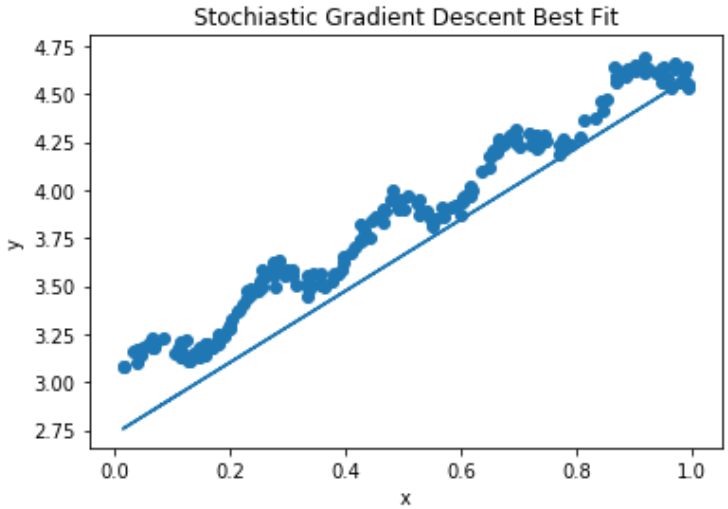
As shown above, too large a learning rate resulted in the loss function not converging, with 0.001 being a small enough learning rate for the algorithm to get very close to the minimum point.

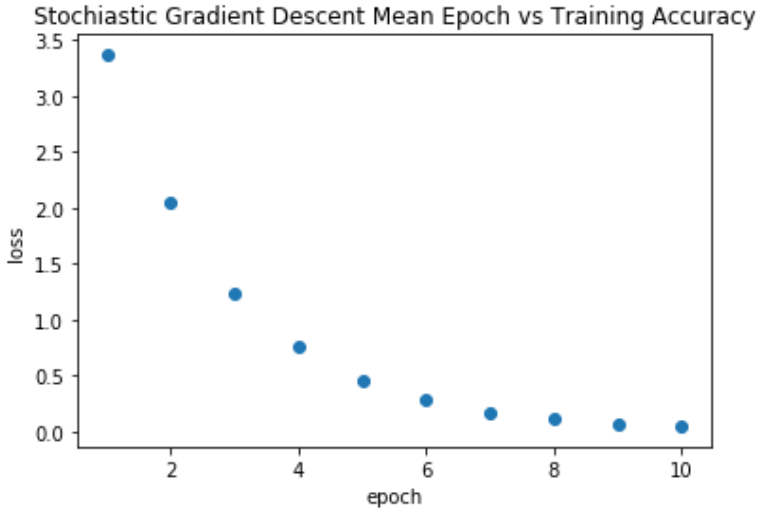
## Stochastic Gradient Descent

### Results

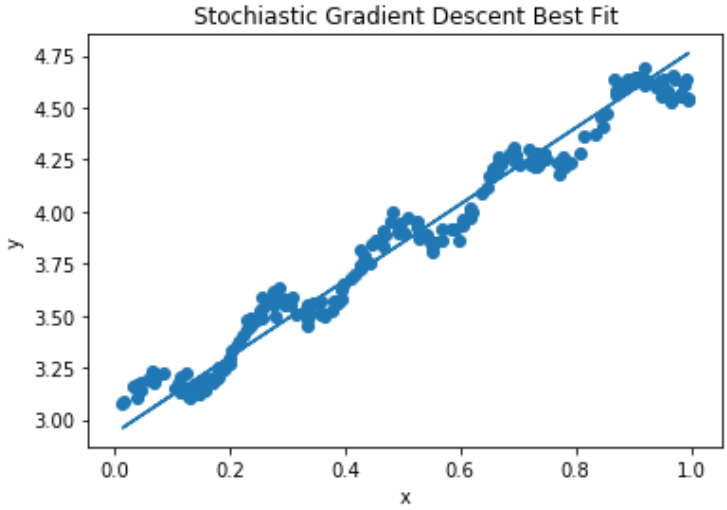
```
n_iteration_list = [10, 50]
learning_rate_list = [0.001, 0.005, 0.01, 0.05, 0.1, 0.3]
for learning_rate in learning_rate_list:
    for n_iter in n_iteration_list:
        print('learning rate: {}, number of iterations: {}'.format(learning_rate, n_iter))
        thetas = stochiastic_gradient_descent(x, y, learning_rate, n_iter)
        plot(x, y, thetas[-1], "Stochiastic Gradient Descent Best Fit")
        plot_training_errors(x, y, thetas, "Stochiastic Gradient Descent Mean Epoch vs Training Accuracy")
```

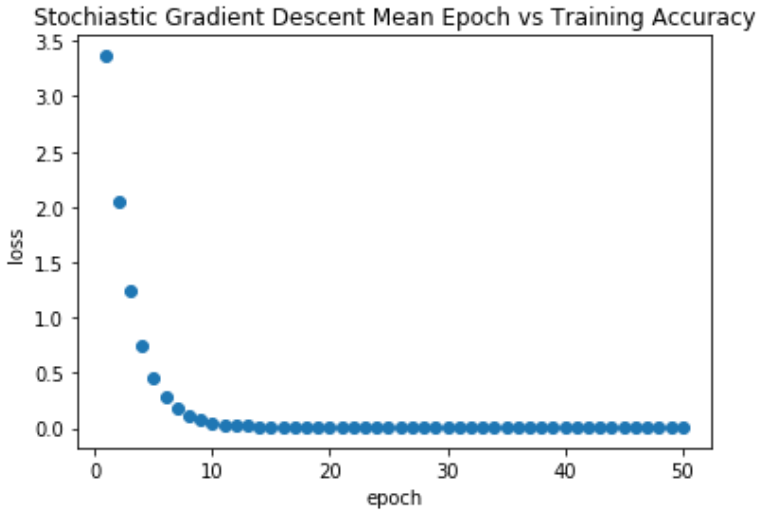
```
learning rate: 0.001, number of iterations: 10
```



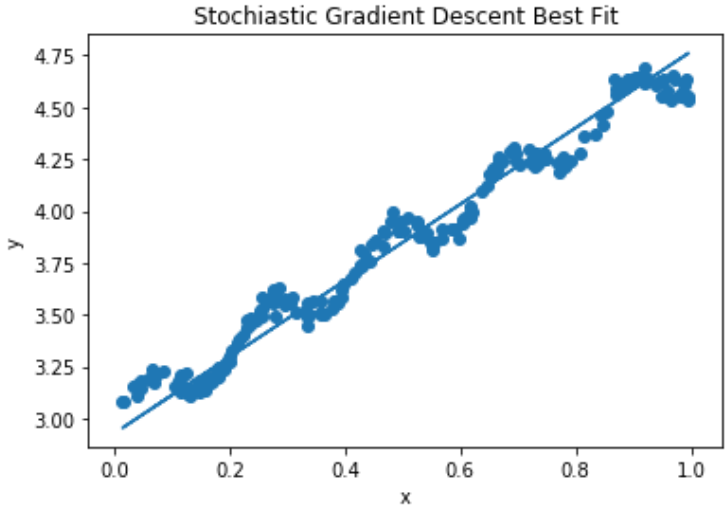


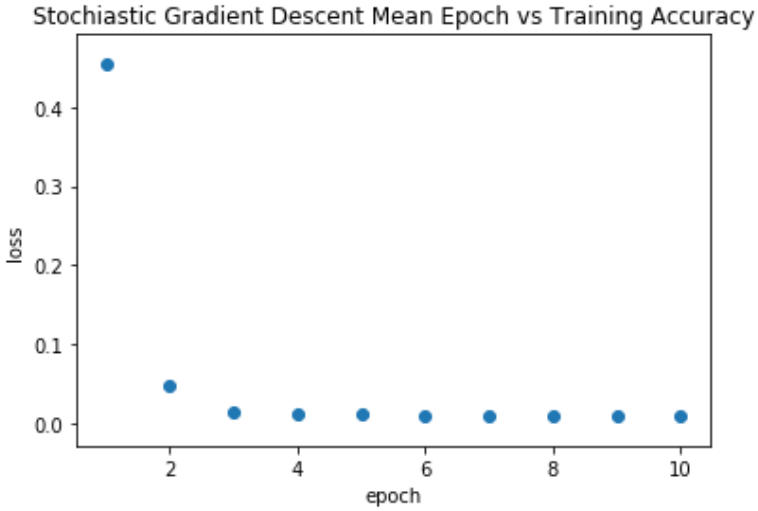

```
learning rate: 0.001, number of iterations: 50
```



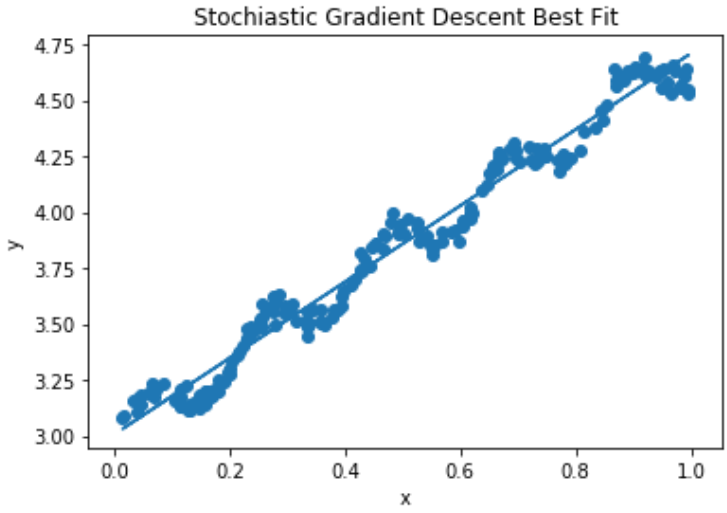



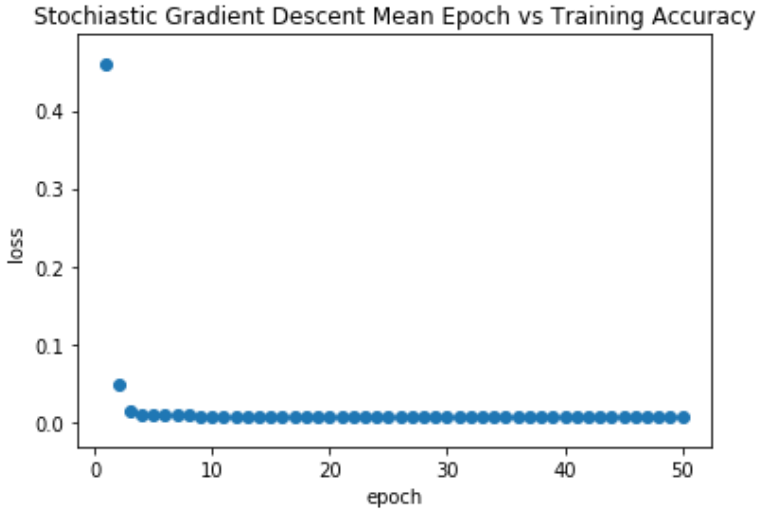

```
learning rate: 0.005, number of iterations: 10
```



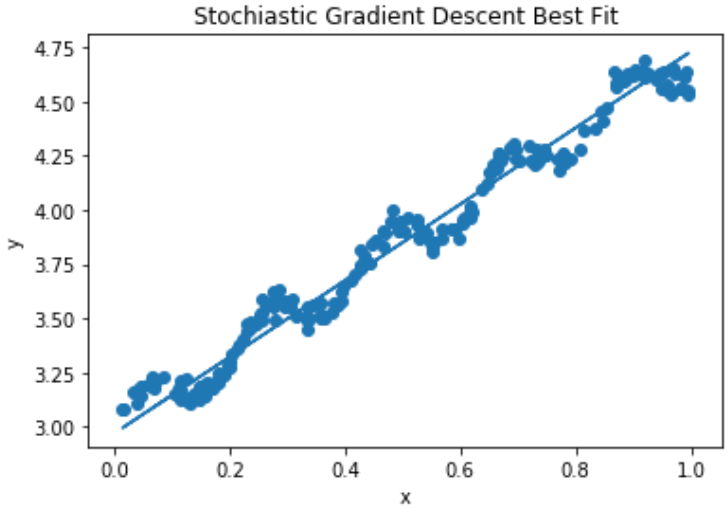



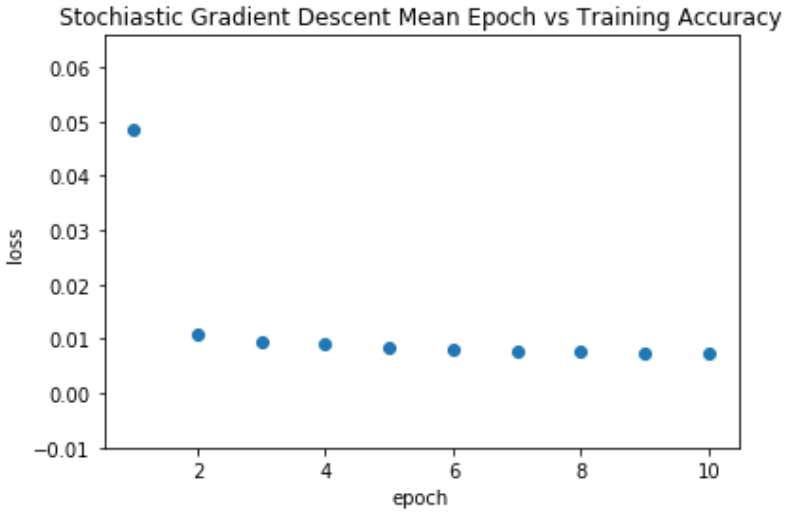

```
1  learning rate: 0.005, number of iterations: 50
```



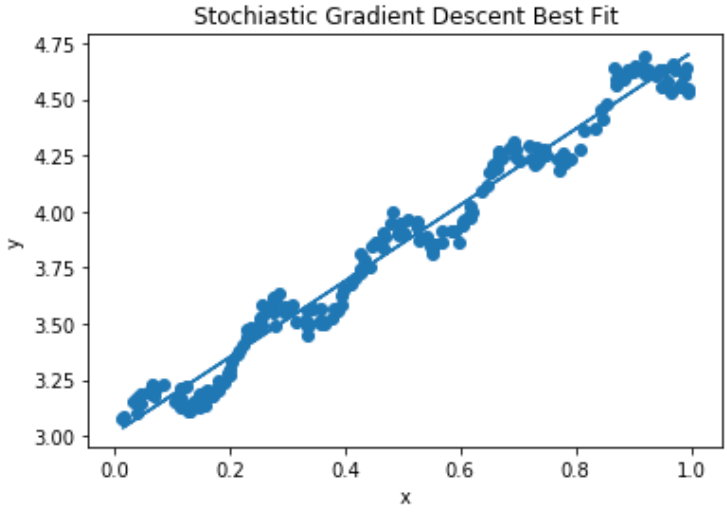


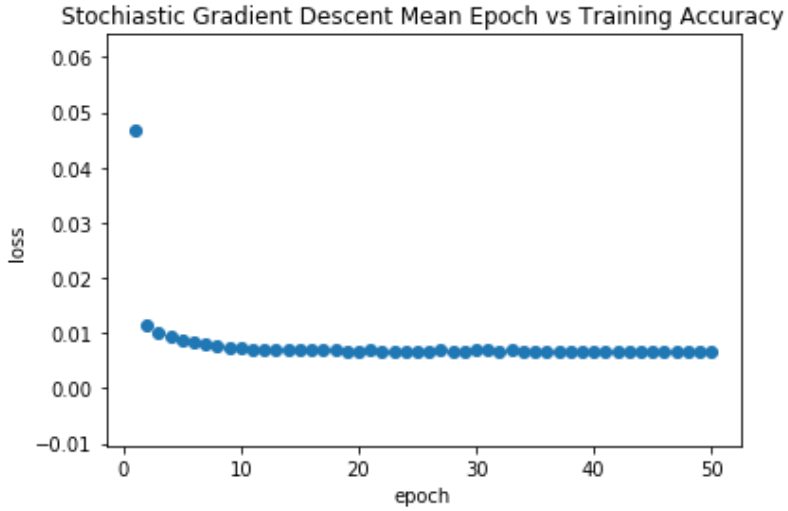

```
learning rate: 0.01, number of iterations: 10
```



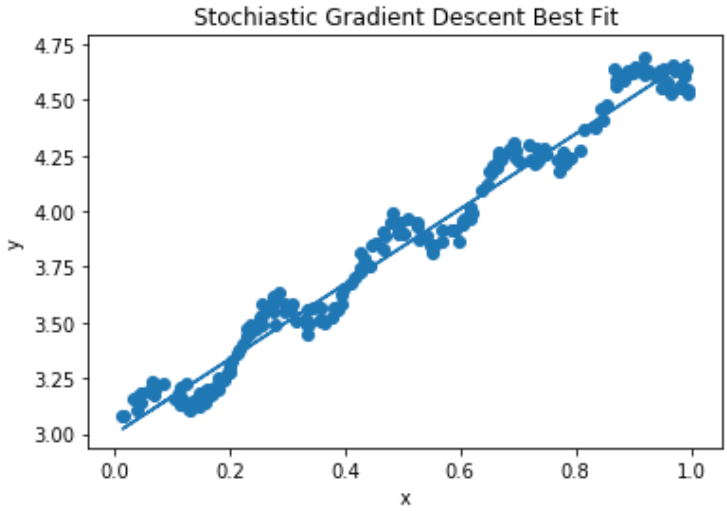



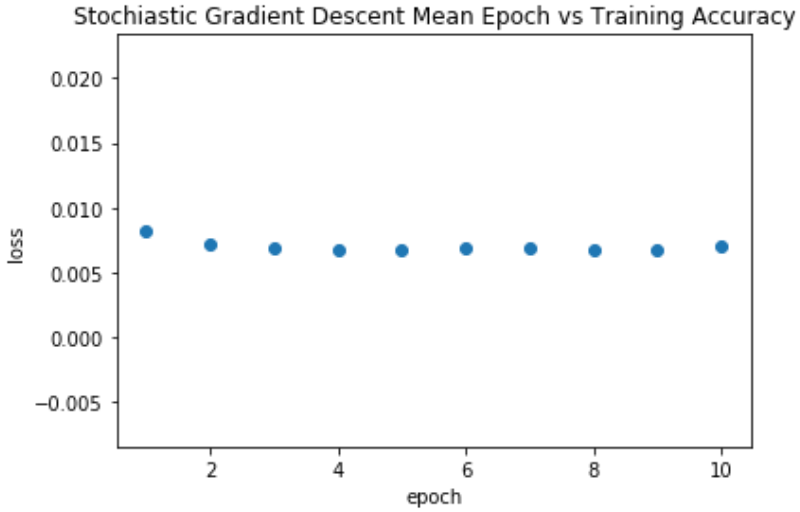

```
learning rate: 0.01, number of iterations: 50
```



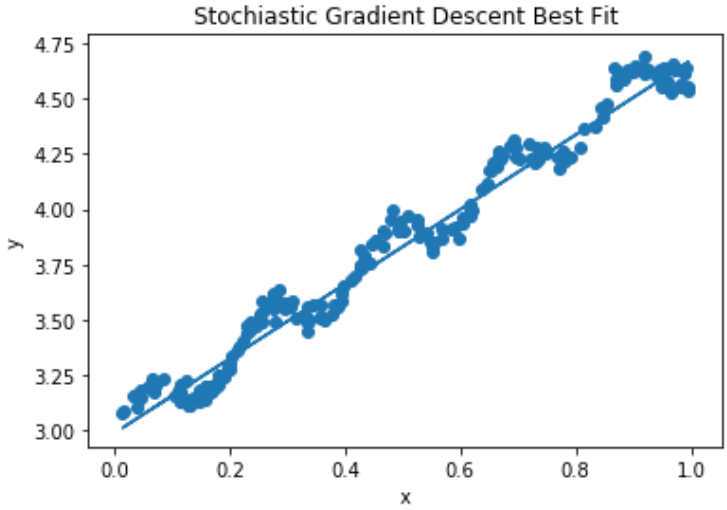



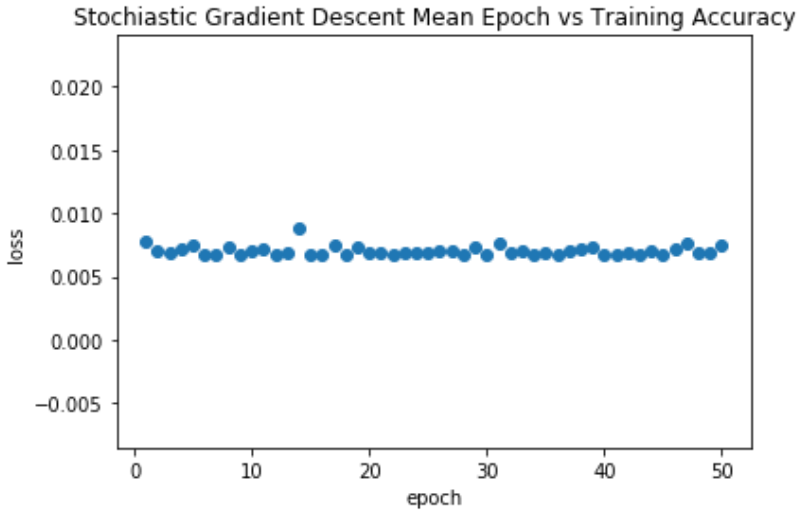

```
learning rate: 0.05, number of iterations: 10
```



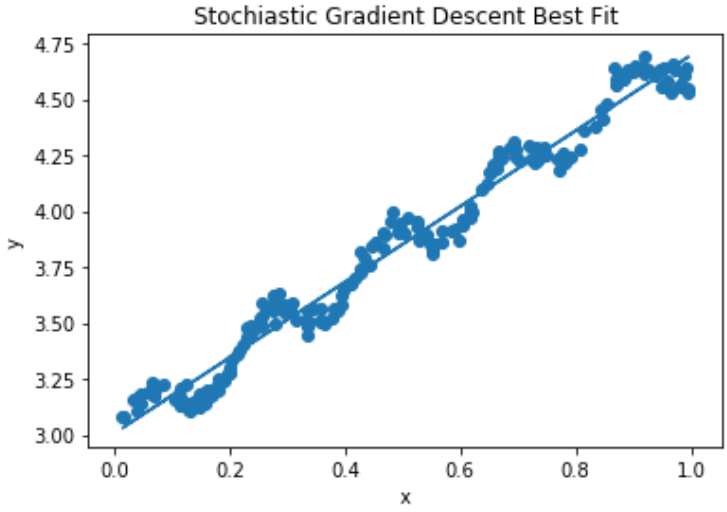


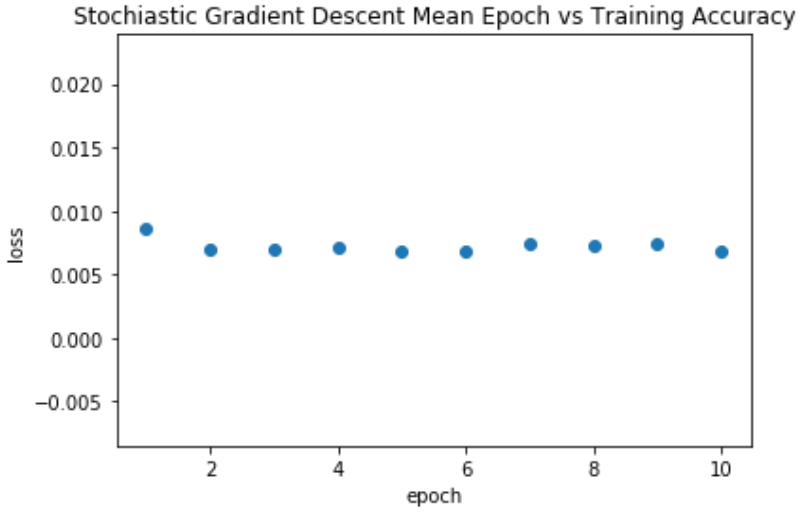

```
1  learning rate: 0.05, number of iterations: 50
```



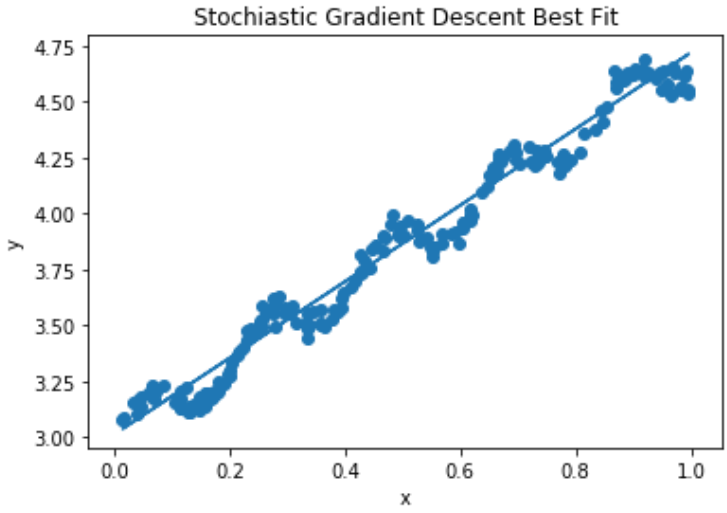



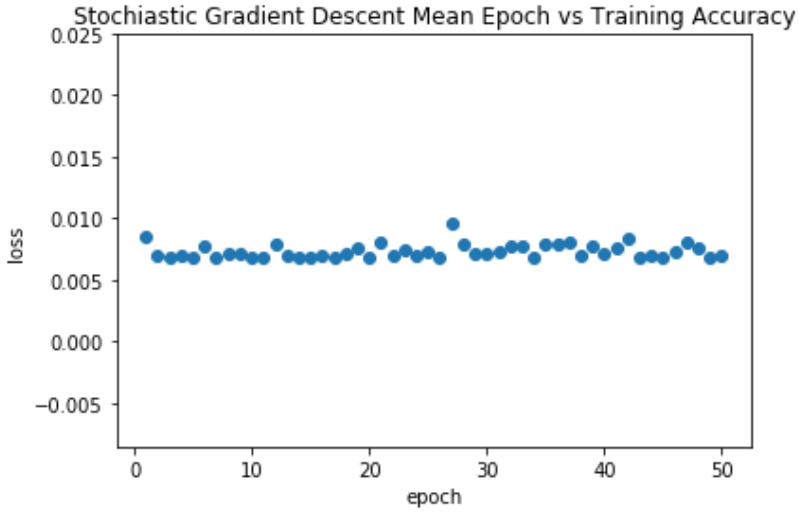

```
1  learning rate: 0.1, number of iterations: 10
```



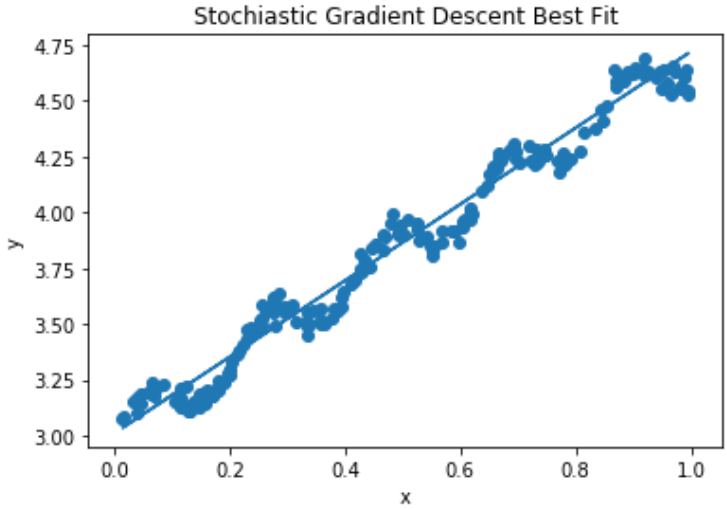



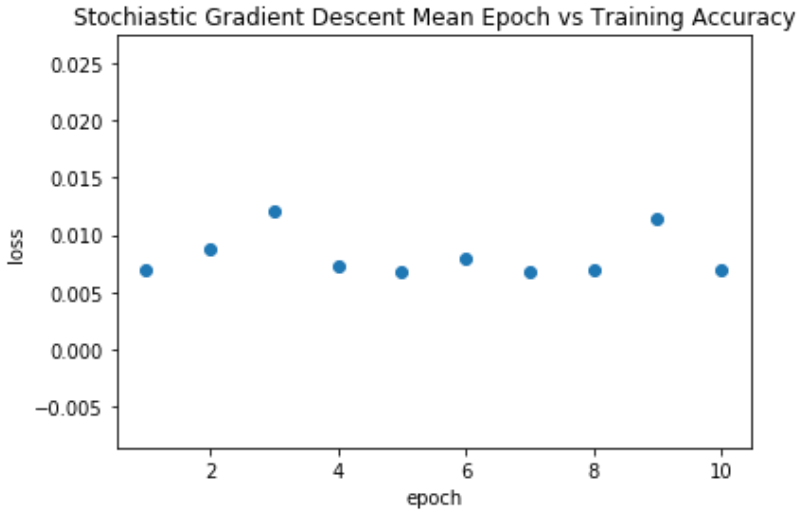

```
1  learning rate: 0.1, number of iterations: 50
```



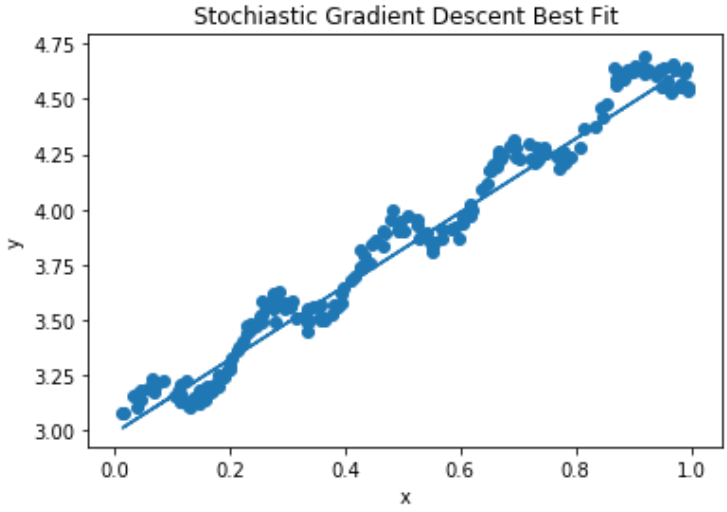


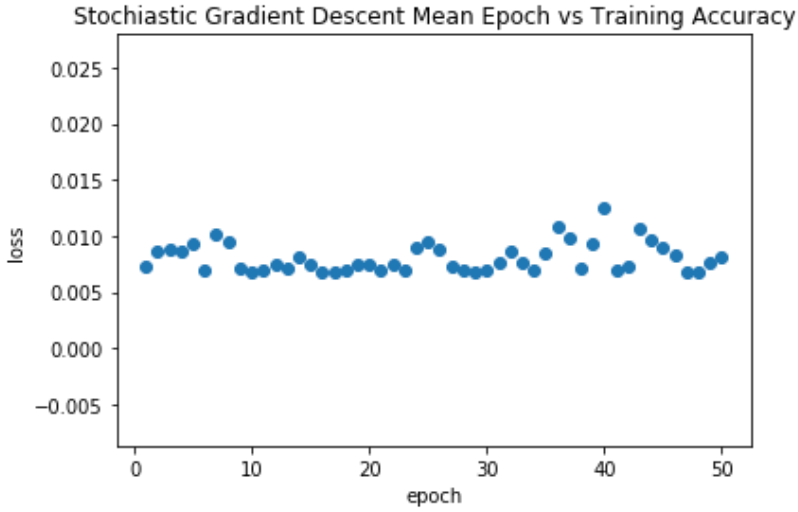

```
learning rate: 0.3, number of iterations: 10
```




```
learning rate: 0.3, number of iterations: 50
```




## Observation

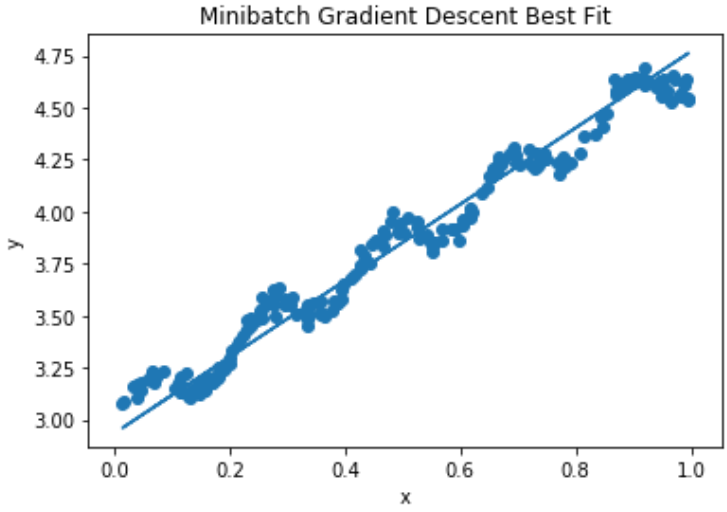
As we can see in the above figure, even with a learning rate of 0.3, SGD still managed to fit pretty well to the data points. We can also see the loss fluctuated slightly but didn't diverge too much.

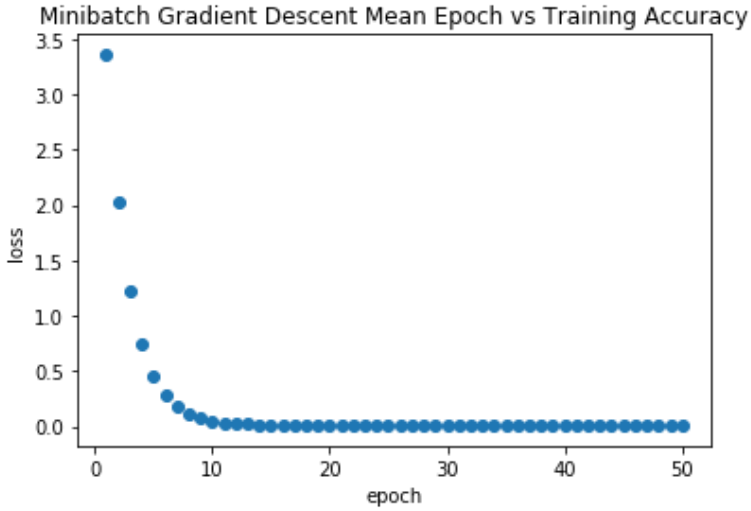
# Minibatch Gradient Descent

## Result

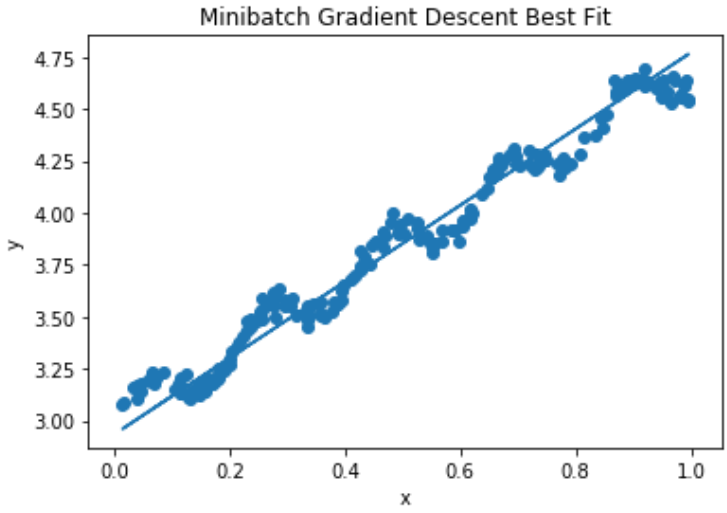
```
learning_rate_list = [0.001, 0.005, 0.01, 0.05, 0.1, 0.3]
for learning_rate in learning_rate_list:
    for batch_size in [10, 50, 100]:
        print('learning rate: {}, batch size: {}, number of iterations:
50'.format(learning_rate, batch_size))
        thetas = minibatch_gradient_descent(x, y, learning_rate, 50, batch_size)
        plot(x, y, thetas[-1], "Minibatch Gradient Descent Best Fit")
        # print(thetas)
        plot_training_errors(
            x, y, thetas, "Minibatch Gradient Descent Mean Epoch vs Training Accuracy")
```

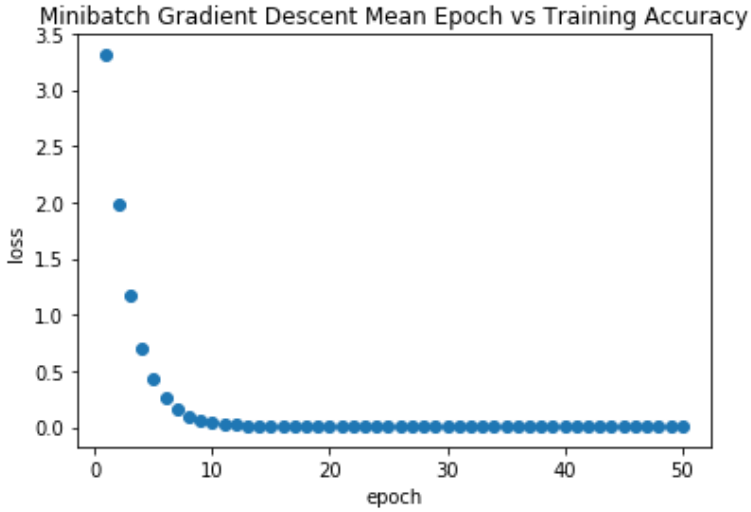
```
learning rate: 0.001, batch size: 10, number of iterations: 50
```



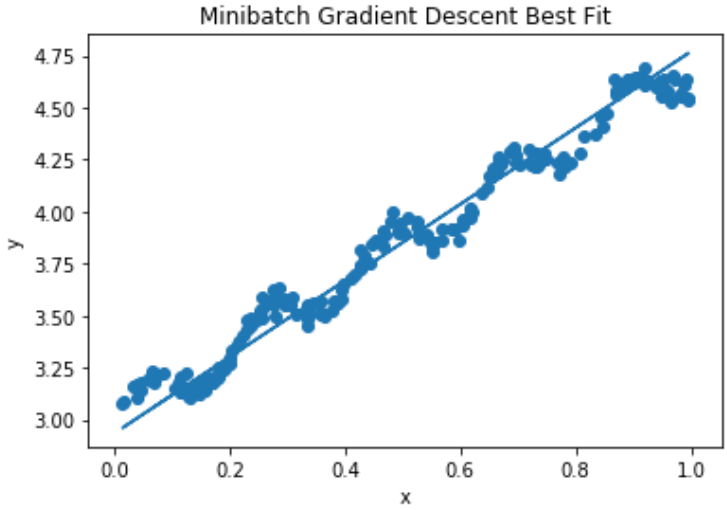



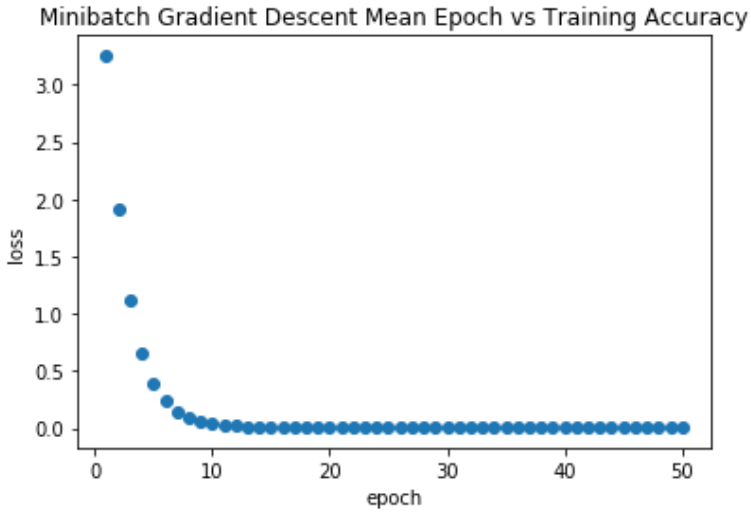

```
learning rate: 0.001, batch size: 50, number of iterations: 50
```



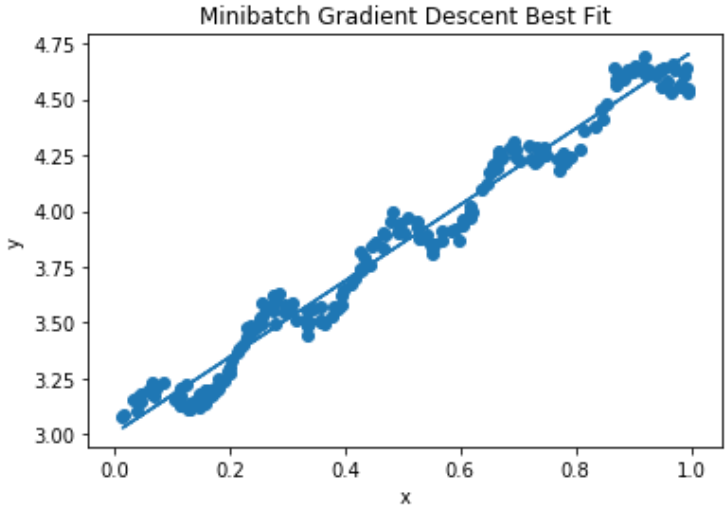


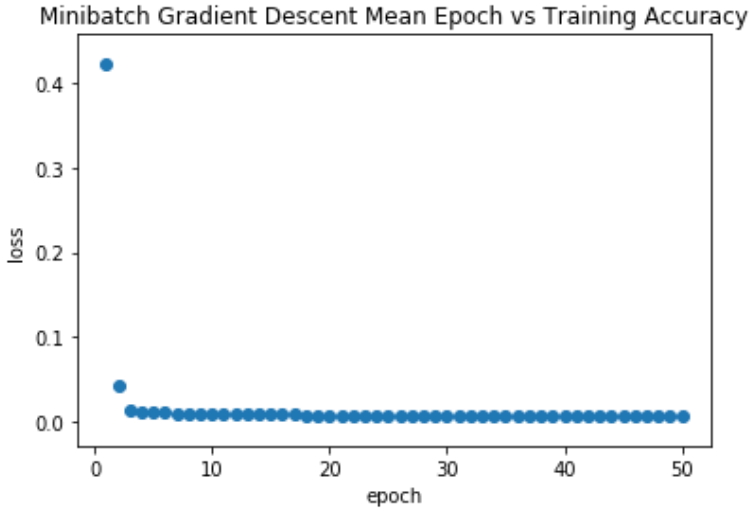

```
learning rate: 0.001, batch size: 100, number of iterations: 50
```



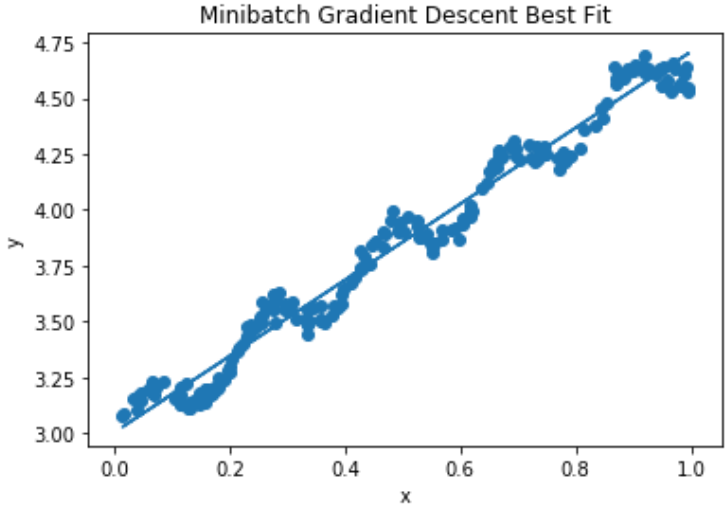



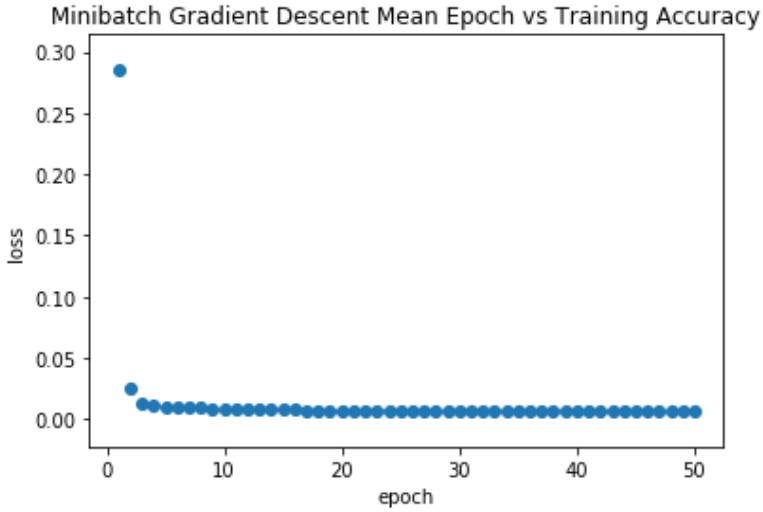

```
learning rate: 0.005, batch size: 10, number of iterations: 50
```



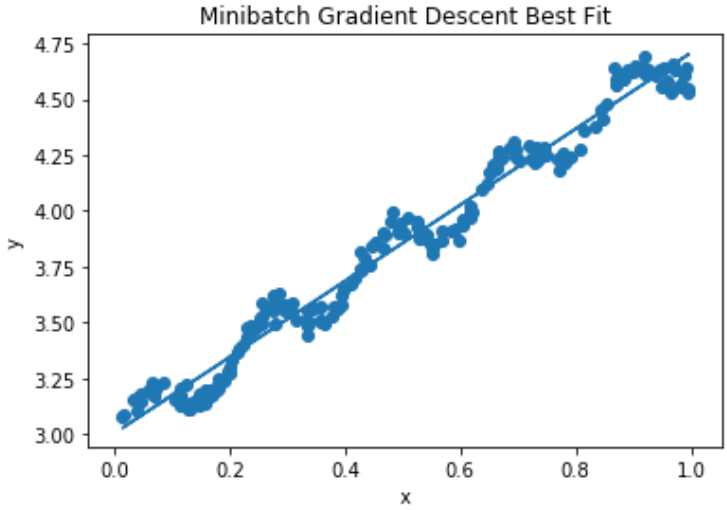



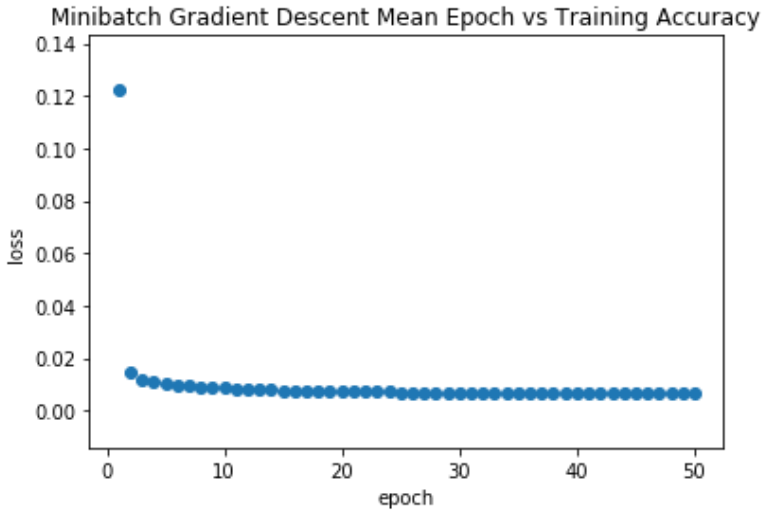

```
1  learning rate: 0.005, batch size: 50, number of iterations: 50
```



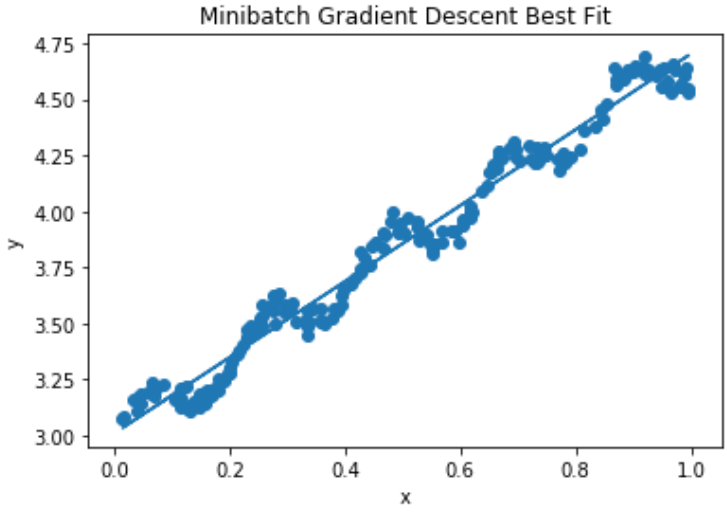


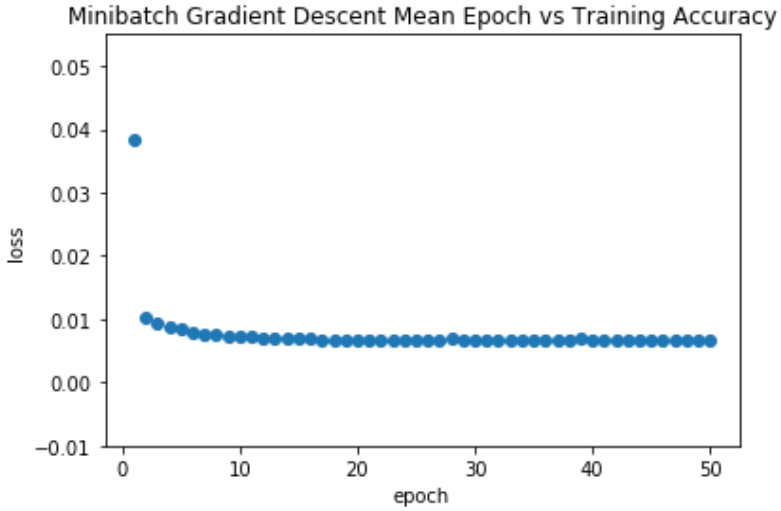

```
learning rate: 0.005, batch size: 100, number of iterations: 50
```



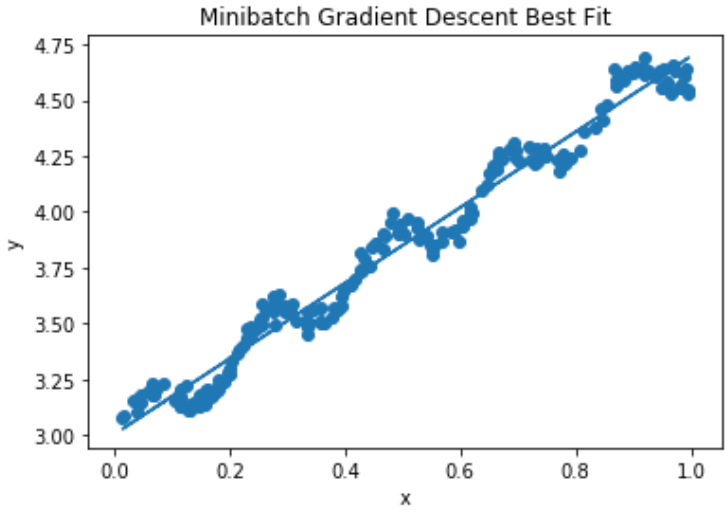



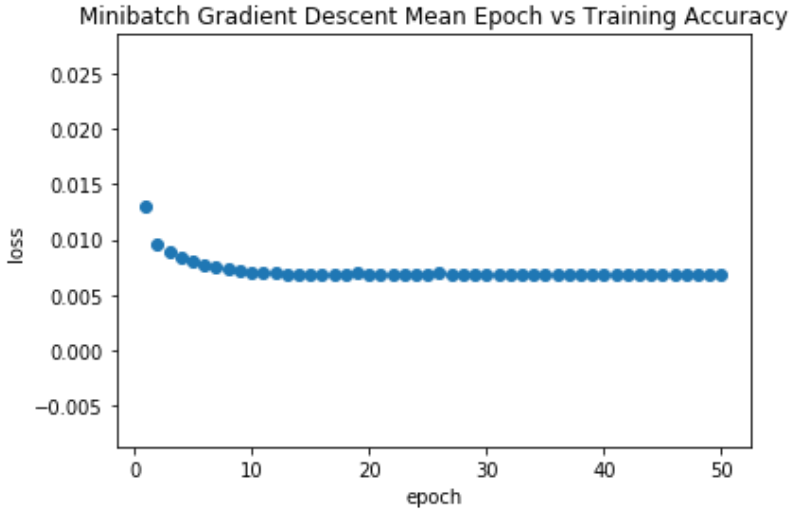

```
learning rate: 0.01, batch size: 10, number of iterations: 50
```



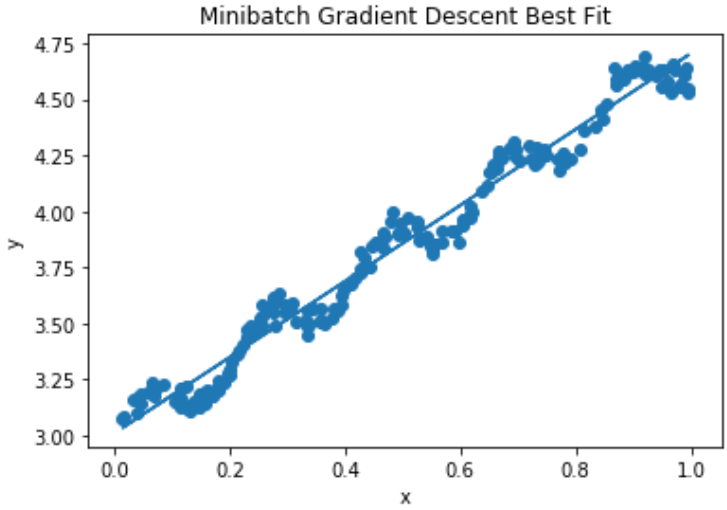



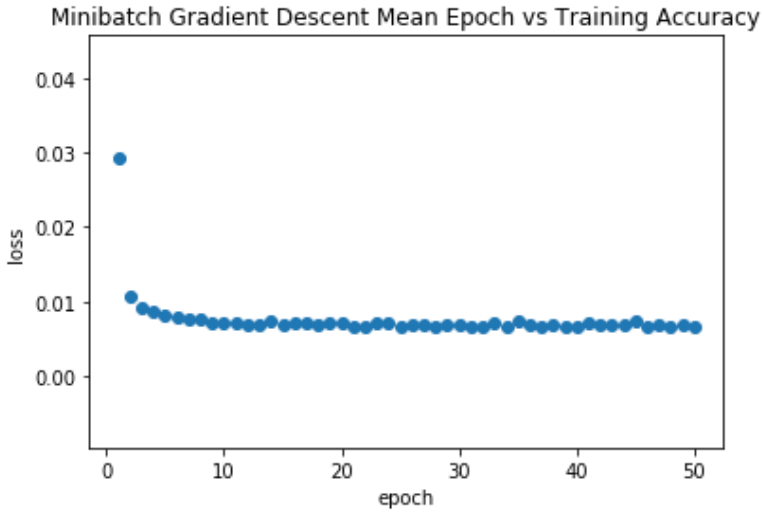

```
learning rate: 0.01, batch size: 50, number of iterations: 50
```



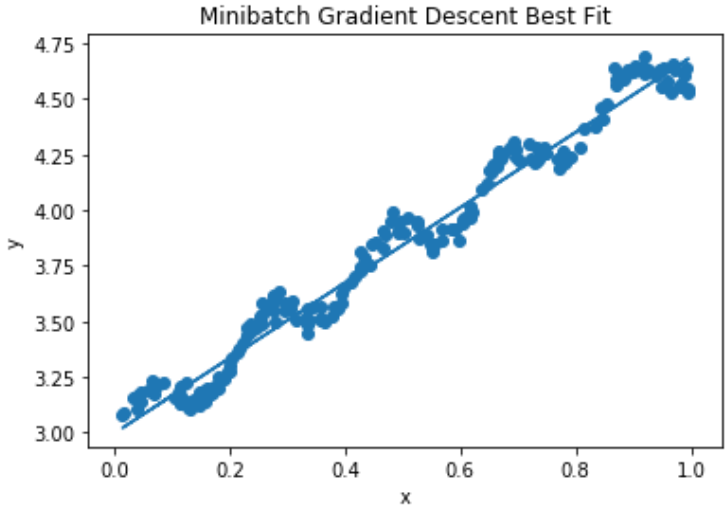


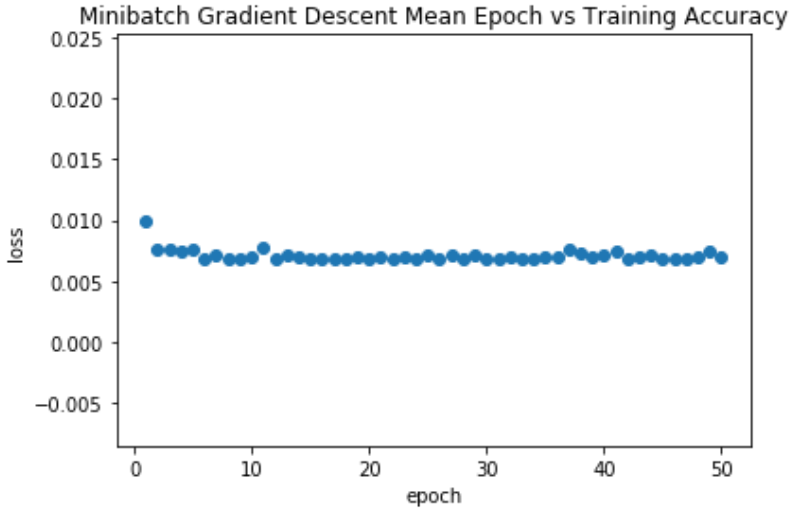

```
1  learning rate: 0.01, batch size: 100, number of iterations: 50
```



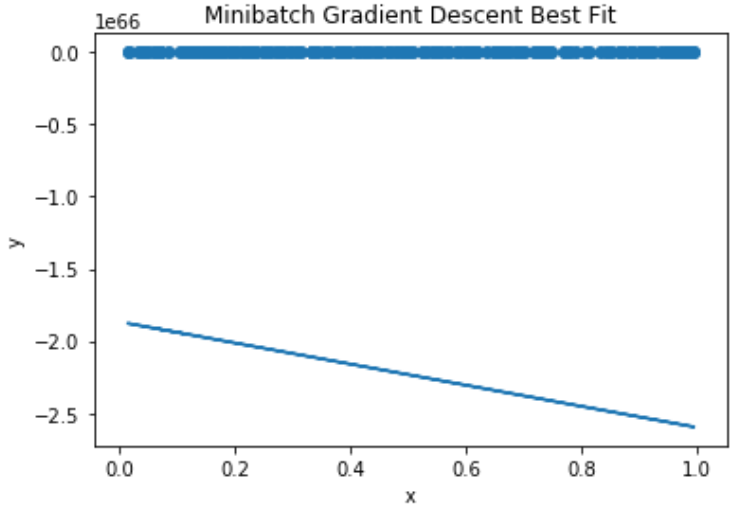



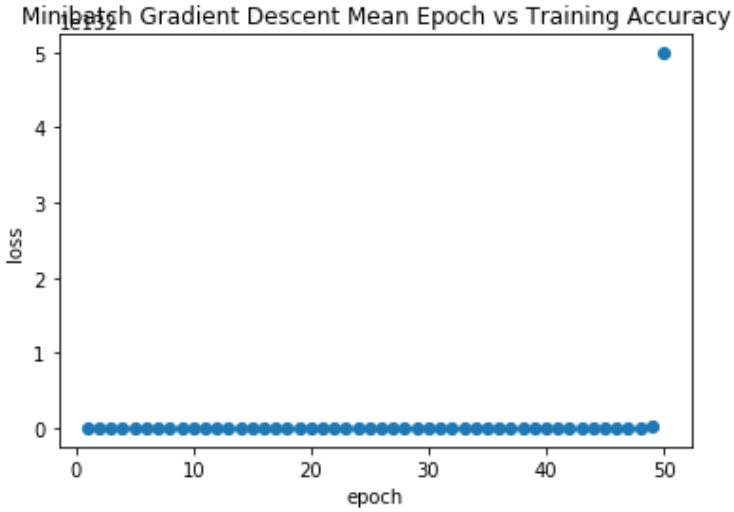

```
1  learning rate: 0.05, batch size: 10, number of iterations: 50
```



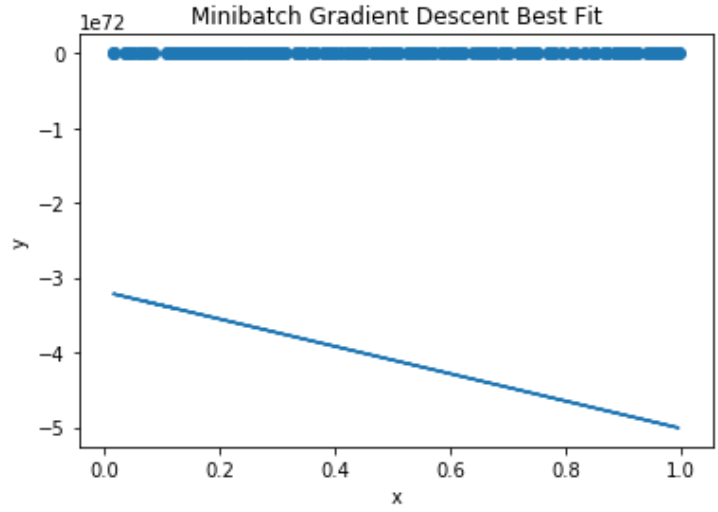



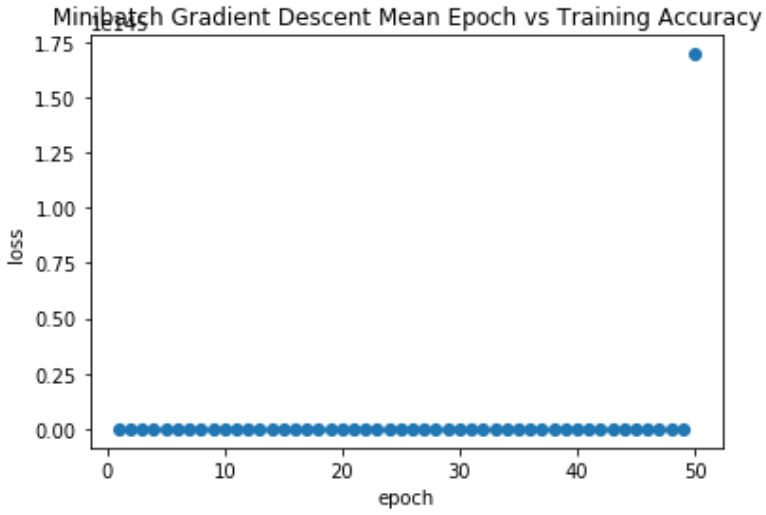

```
learning rate: 0.05, batch size: 50, number of iterations: 50
```



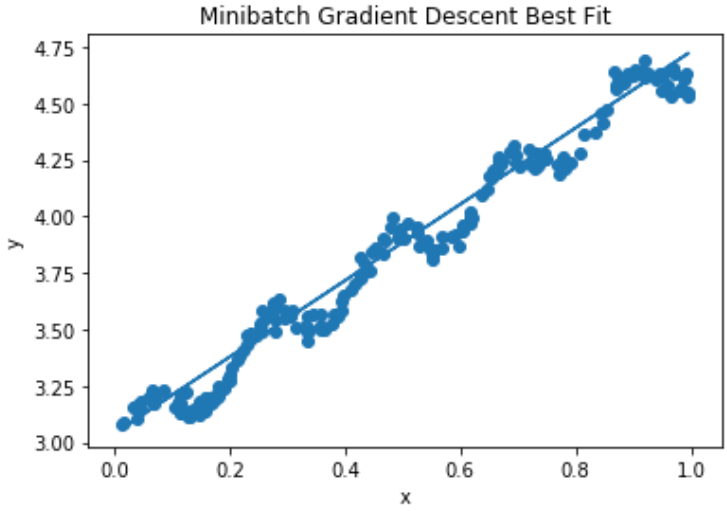


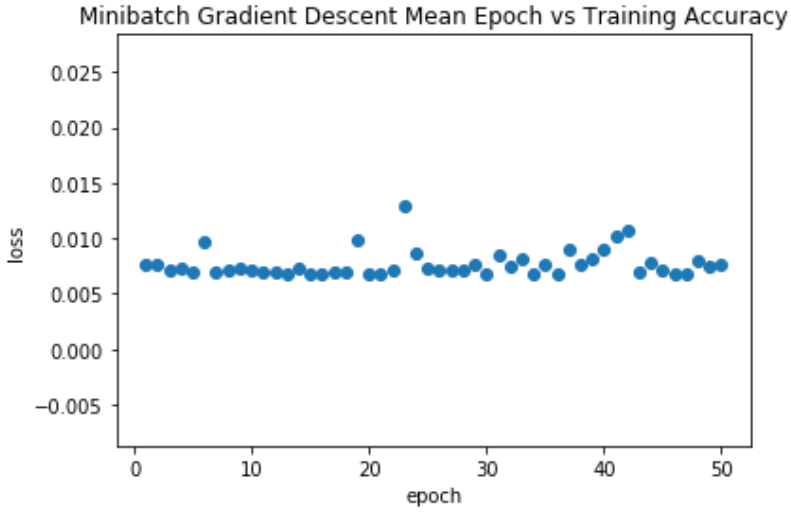

```
learning rate: 0.05, batch size: 100, number of iterations: 50
```



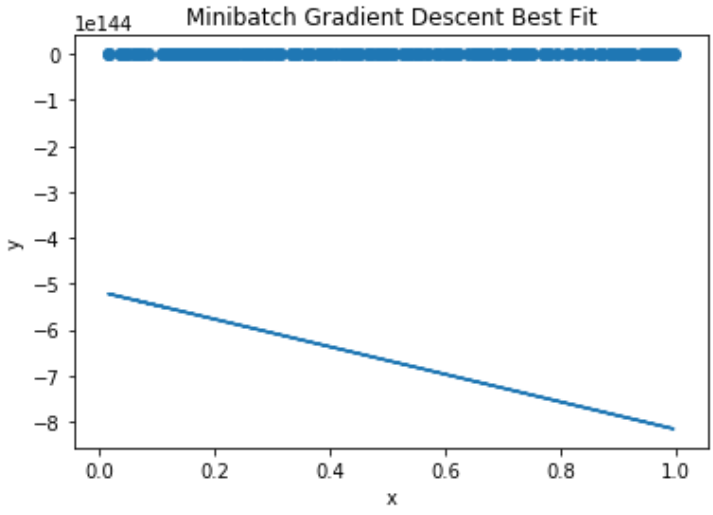



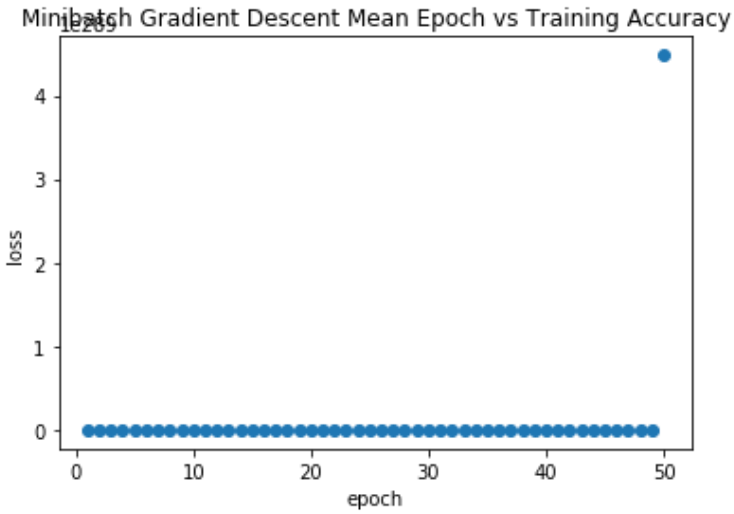

```
learning rate: 0.1, batch size: 10, number of iterations: 50
```



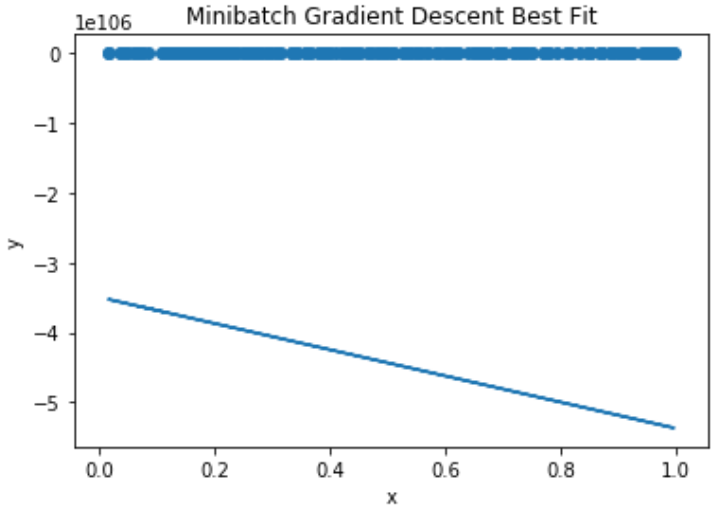



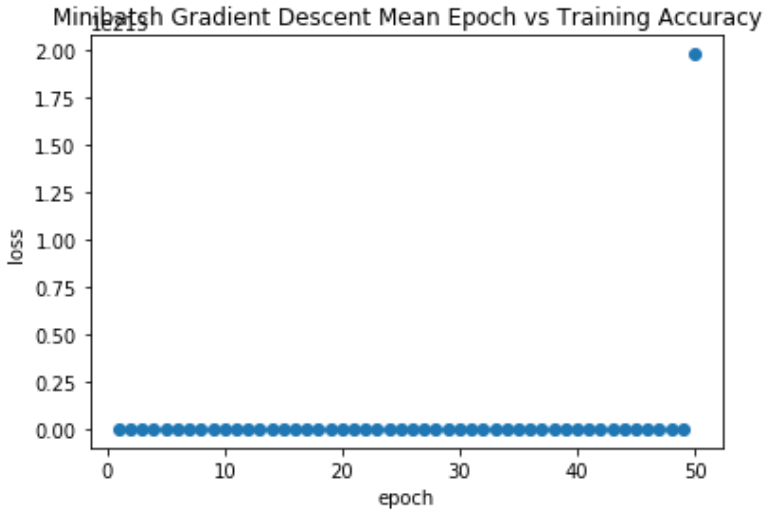

```
learning rate: 0.1, batch size: 50, number of iterations: 50
```





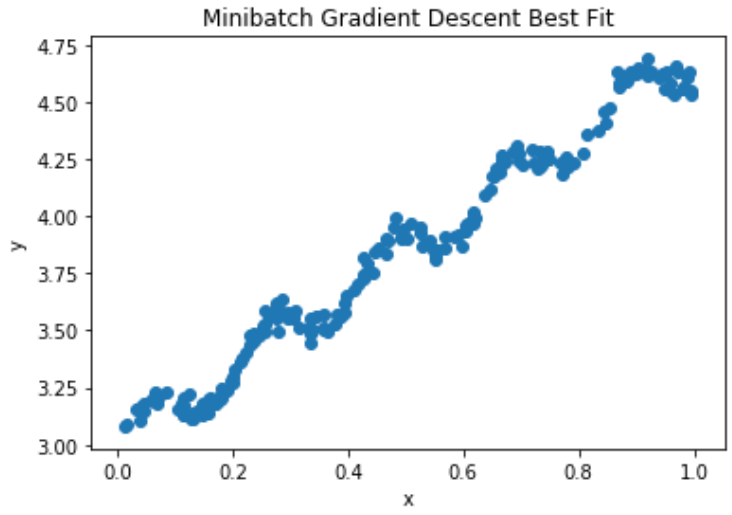

```
learning rate: 0.1, batch size: 100, number of iterations: 50
```

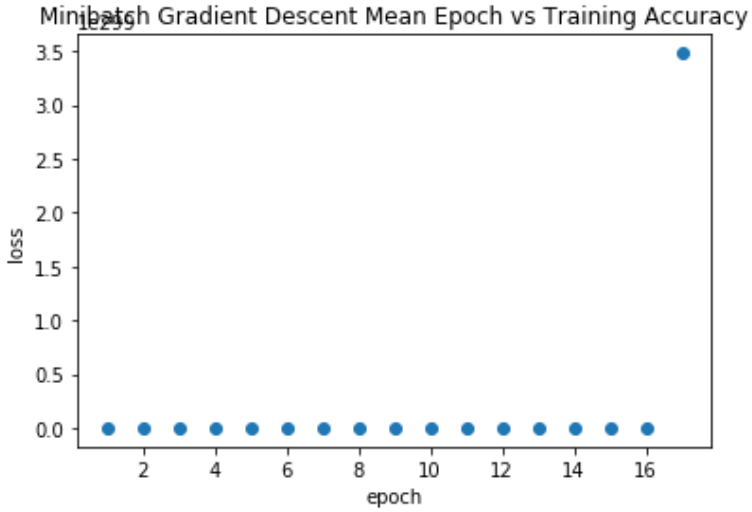


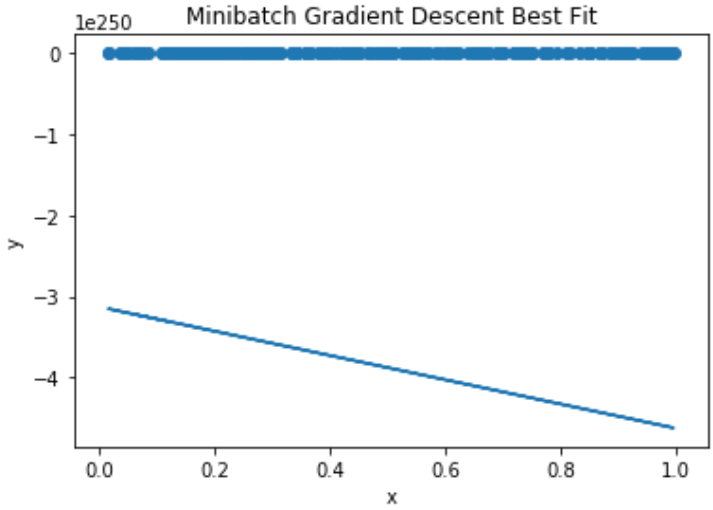

```
learning rate: 0.3, batch size: 10, number of iterations: 50
```

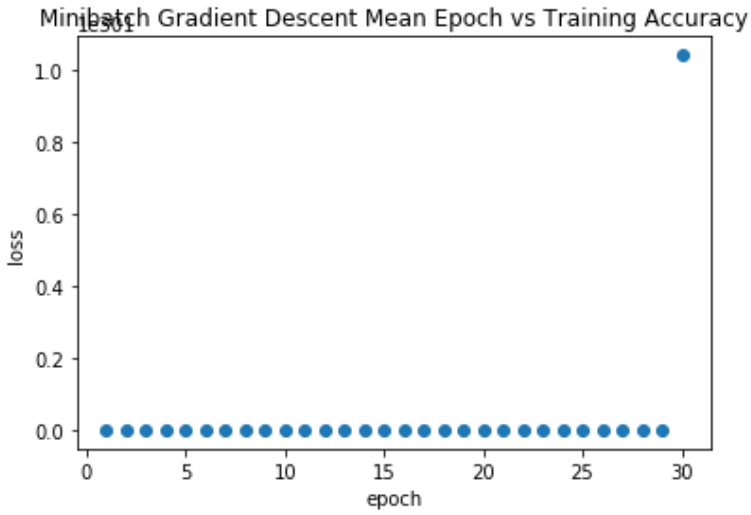
```
C:\Users\james\Anaconda3\lib\site-packages\ipykernel_launcher.py:93: RuntimeWarning:
overflow encountered in subtract
C:\Users\james\Anaconda3\lib\site-packages\ipykernel_launcher.py:96: RuntimeWarning:
invalid value encountered in add
```



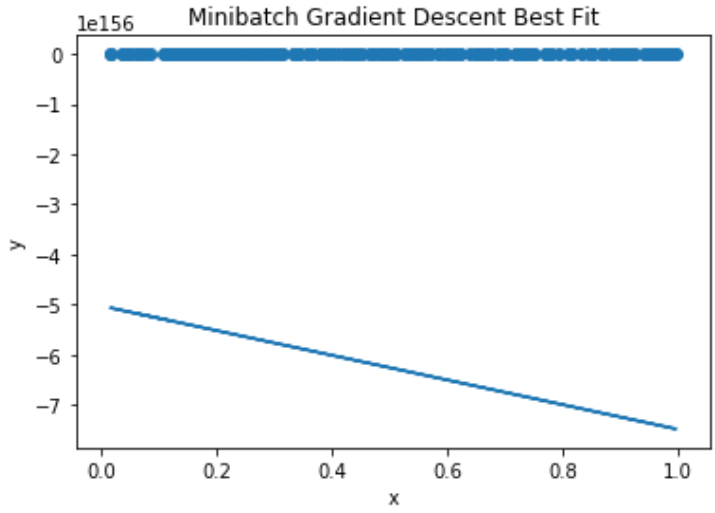



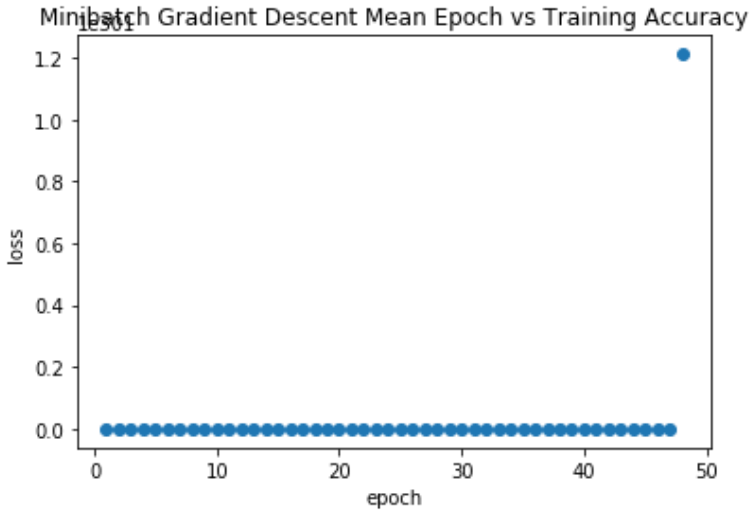

```
learning rate: 0.3, batch size: 50, number of iterations: 50
```






```
learning rate: 0.3, batch size: 100, number of iterations: 50
```




## Observation

From observing the results above, we can see when the batch size is larger, we need to tone down the learning rate in order to avoid the diverging.

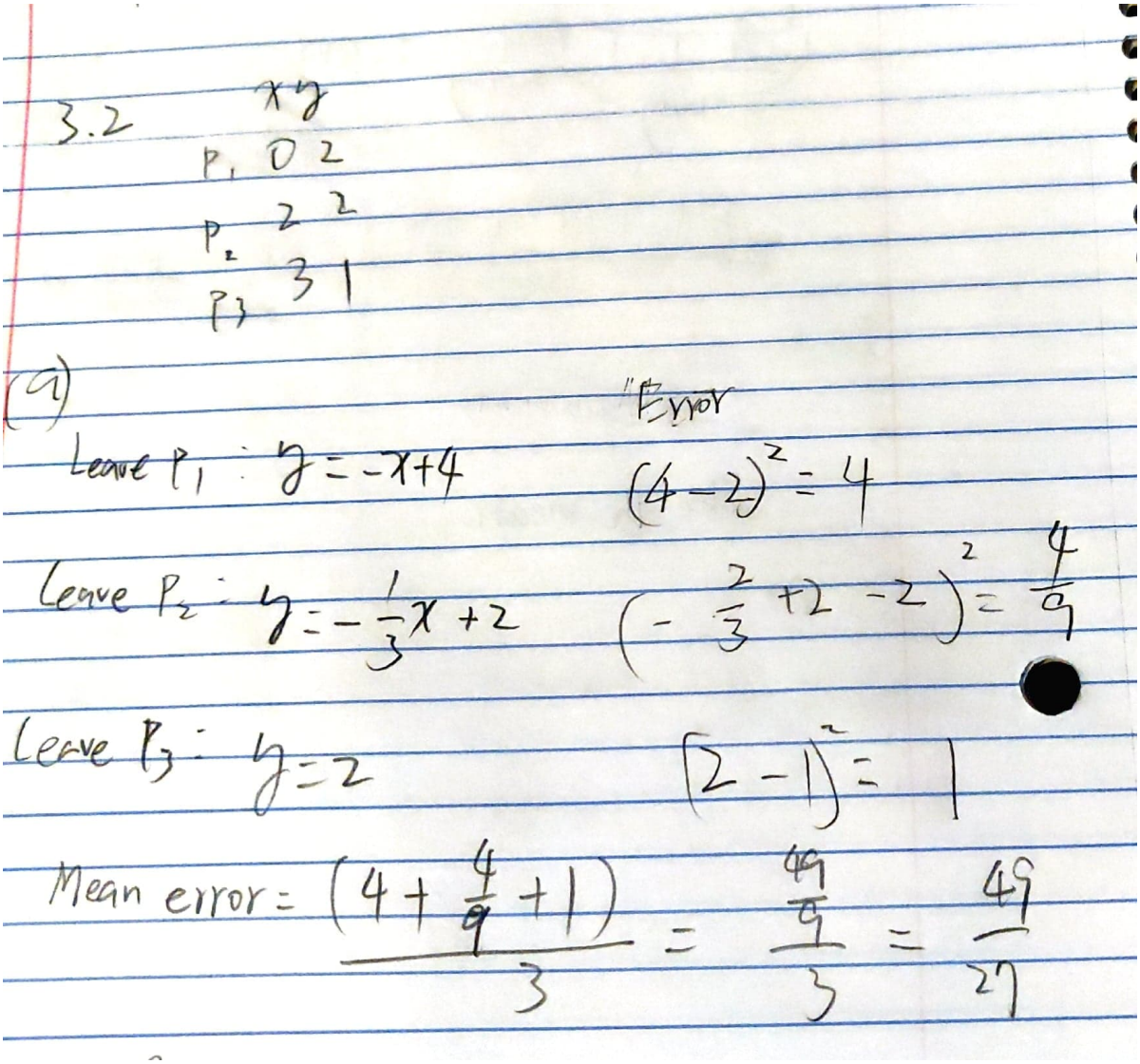
# 3 Sample Exam Questions:

## 3.1

The MSE will be 0 as every point in the data set sits perfectly on the same line, so we can use linear regression to find a line to fit the data set perfectly.

## 3.2

(a)




(b)

(b) $y = \beta_0$

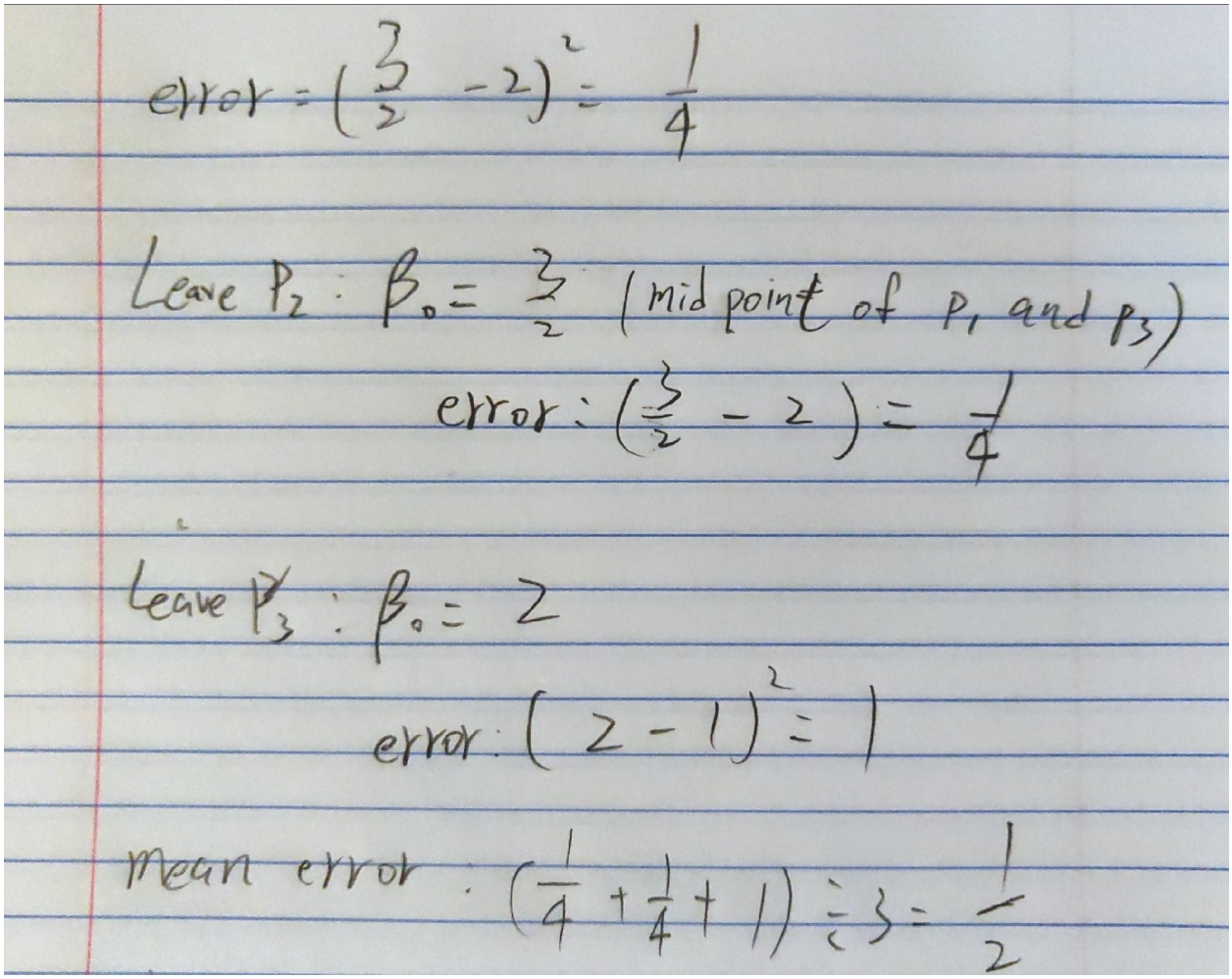
Leave $\beta_1$ $\quad J(\beta_0) = (\beta_0 - 2)^2 + (\beta_0 - 1)^2$

$$= \beta_0^2 - 4\beta_0 + 4 + \beta_0^2 - 2\beta_0 + 1$$

$$= 2\beta_0^2 - 6\beta_0 + 5$$

$$\frac{dJ}{d\beta_0} = 4\beta_0 - 6 \stackrel{\text{set}}{=} 0$$

$$\beta_0 = \frac{3}{2}$$




(c) The second model, since it produced a smaller error

## 3.3

(e) A, because the smaller the training data set is, the easier it is for us to find a line that fits the data points. Two points, for example, can be perfectly fit with 0 training error.

(f) B, because as we increase the size of the training data set, the model should generalize better and be able to make better predictions on the testing data.