Grading

This week's lab doesn't have any auto-graded components. Each question in this notebook has an accompanying Peer Review question. Although the lab shows as being ungraded, you need to complete the notebook to answer the Peer Review questions.
DO NOT CHANGE VARIABLE OR METHOD SIGNATURES

Validate Button

This week's lab doesn't have any auto-graded components. Each question in this notebook has an accompanying Peer Review question. Although the lab shows as being ungraded, you need to complete the notebook to answer the Peer Review questions.

You do not need to use the Validate button for this lab since there are no auto-graded components. If you hit the Validate button, it will time out given the number of visualizations in the notebook. Cells with longer execution times cause the validate button to time out and freeze. ***This notebook's Validate button time-out does not affect the final submission grading.***

Homework 2. Stochastic Gradient Descent

In this assignment we'll implement a rudimentary Stochastic Gradient Descent algorithm to learn the weights in simple linear regression. Then we'll see if we can make it more efficient. Finally, we'll investigate some graphical strategies for diagnosing convergence and tuning parameters.

Note: The cell below has some helper functions. Scroll down and evaluate those before proceeding.

In [1]:
```python
!pip install pytest
import numpy as np
import matplotlib.pylab as plt
import pytest
%matplotlib inline
```

```
Requirement already satisfied: pytest in /opt/conda/lib/python3.7/site-p
ackages (7.4.4)
Requirement already satisfied: iniconfig in /opt/conda/lib/python3.7/sit
e-packages (from pytest) (2.0.0)
Requirement already satisfied: exceptiongroup>=1.0.0rc8 in /opt/conda/li
b/python3.7/site-packages (from pytest) (1.2.1)
Requirement already satisfied: tomli>=1.0.0 in /opt/conda/lib/python3.7/
site-packages (from pytest) (2.0.1)
Requirement already satisfied: importlib-metadata>=0.12 in /opt/conda/li
b/python3.7/site-packages (from pytest) (1.6.0)
Requirement already satisfied: pluggy<2.0,>=0.12 in /opt/conda/lib/pytho
n3.7/site-packages (from pytest) (1.2.0)
Requirement already satisfied: packaging in /opt/conda/lib/python3.7/sit
e-packages (from pytest) (20.1)
Requirement already satisfied: zipp>=0.5 in /opt/conda/lib/python3.7/sit
e-packages (from importlib-metadata>=0.12->pytest) (3.1.0)
Requirement already satisfied: six in /opt/conda/lib/python3.7/site-pack
ages (from packaging->pytest) (1.14.0)
Requirement already satisfied: pyparsing>=2.0.2 in /opt/conda/lib/python
3.7/site-packages (from packaging->pytest) (2.4.7)
WARNING: You are using pip version 21.3.1; however, version 24.0 is avai
lable.
You should consider upgrading via the '/opt/conda/bin/python3 -m pip ins
tall --upgrade pip' command.
```

In [2]:
```
mycolors = {"blue": "steelblue", "red":"#a76c6e",  "green":"#6a9373", "sm
oke": "#f2f2f2"}

def eval_RSS(X, y, b0, b1):
    rss = 0
    for ii in range(len(df)):
        xi = df.loc[ii, "x"]
        yi = df.loc[ii, "y"]
        rss += (yi - (b0 + b1 * xi)) ** 2
    return rss

def plotsurface(X, y, bhist=None):
    xx, yy = np.meshgrid(np.linspace(-3, 3, 300), np.linspace(-1, 5, 30
0))
    Z = np.zeros((xx.shape[0], yy.shape[0]))
    for ii in range(X.shape[0]):
        Z += (y[ii] - xx - yy * X[ii,1]) ** 2
    fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(10,10))
    levels = [125, 200] + list(range(400,2000,400))
    CS = ax.contour(xx, yy, Z, levels=levels)
    ax.clabel(CS, CS.levels, inline=True, fontsize=10)
    ax.set_xlim([-3,3])
    ax.set_ylim([-1,5])
    ax.set_xlabel(r"$\beta_0$", fontsize=20)
    ax.set_ylabel(r"$\beta_1$", fontsize=20)
    if bhist is not None:
        for ii in range(bhist.shape[0]-1):
            x0 = bhist[ii][0]
            y0 = bhist[ii][1]
            x1 = bhist[ii+1][0]
            y1 = bhist[ii+1][1]
            ax.plot([x0, x1], [y0, y1], color="black", marker="o", lw=
1.5, markersize=5)
```

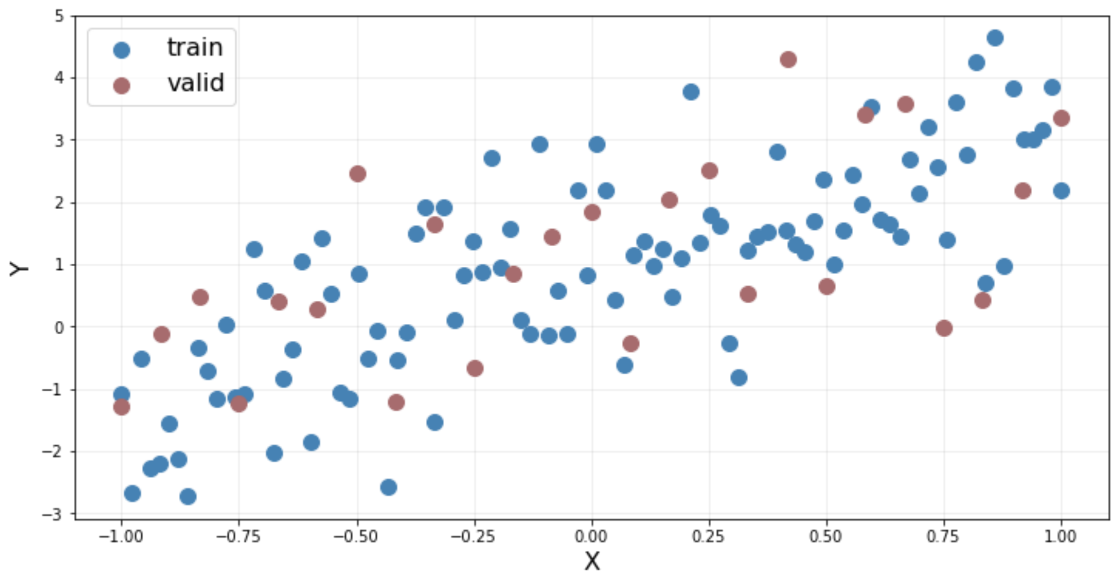
Part 1: Setting Up Simulated Data and a Sanity Check

We'll work with simulated data for this exercise where our generative model is given by

$$Y = 1 + 2X + \epsilon \text{ where} \epsilon \sim N(0, \sigma^2)$$

Part A: The following function will generate data from the model. We'll grab a training set of size $n = 100$ and a validation set of size $n = 50$.

In [3]:
```
def dataGenerator(n, sigsq=1.0, random_state=1236):
    np.random.seed(random_state)
    x_train = np.linspace(-1, 1, n)
    x_valid = np.linspace(-1, 1, int(n / 4))
    y_train = 1 + 2 * x_train + np.random.randn(n)
    y_valid = 1 + 2 * x_valid + np.random.randn(int(n / 4))
    return x_train, x_valid, y_train, y_valid

x_train, x_valid, y_train, y_valid = dataGenerator(100)

fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(12,6))
ax.scatter(x_train, y_train, color="steelblue", s=100, label="train")
ax.scatter(x_valid, y_valid, color="#a76c6e", s=100, label="valid")
ax.grid(alpha=0.25)
ax.set_axisbelow(True)
ax.set_xlabel("X", fontsize=16)
ax.set_ylabel("Y", fontsize=16)
ax.legend(loc="upper left", fontsize=16);
```



Part B: Since we're going to be implementing things ourselves, we're going to want to prepend the data matrices with a column of ones so we can fit a bias term. We can do this using numpy's column_stack (https://docs.scipy.org/doc/numpy/reference/generated/numpy.column_stack.html) function.

In [4]:
```
X_train = np.column_stack((np.ones_like(x_train), x_train))
X_valid = np.column_stack((np.ones_like(x_valid), x_valid))
```

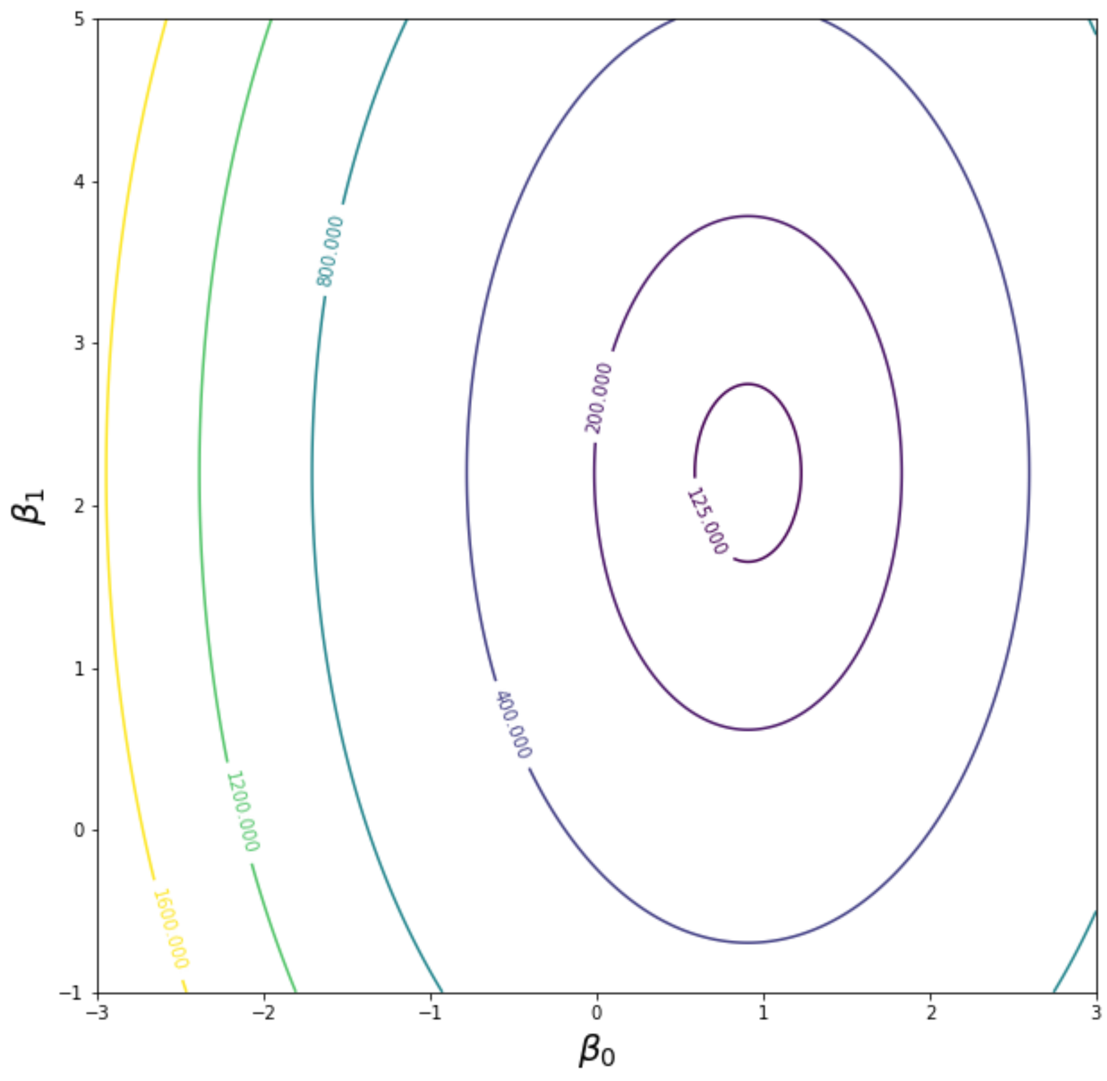
Part C: Finally, let's fit a linear regression model with sklearn's LinearRegression class and print the coefficients so we know what we're shooting for.

In [5]:
```
from sklearn.linear_model import LinearRegression
reg = LinearRegression(fit_intercept=False)
reg.fit(X_train, y_train)
print("sklearn says the coefficients are ", reg.coef_)
```

```
sklearn says the coefficients are  [0.90918343 2.20093262]
```

Part D: The last thing we'll do is visualize the surface of the RSS, of which we're attempting to find the minimum. Does it looks like the parameters reported by sklearn lie at the bottom of the RSS surface?

```
In [6]: plotsurface(X_train, y_train)
```



Part 2: Implementing and Improving SGD

Part A: Now it's time to implement Stochastic Gradient Descent. Most of the code in the function sgd has been written for you. Your job is to fill in the values of the partial derivatives in the appropriate places.
Recall that the update scheme is given by

$$\beta_0 \leftarrow \beta_0 - \eta \cdot 2 \cdot [(\beta_0 + \beta_1 x_i) - y_i]$$
$$\beta_1 \leftarrow \beta_1 - \eta \cdot 2 \cdot [(\beta_0 + \beta_1 x_i) - y_i]\, x_i$$

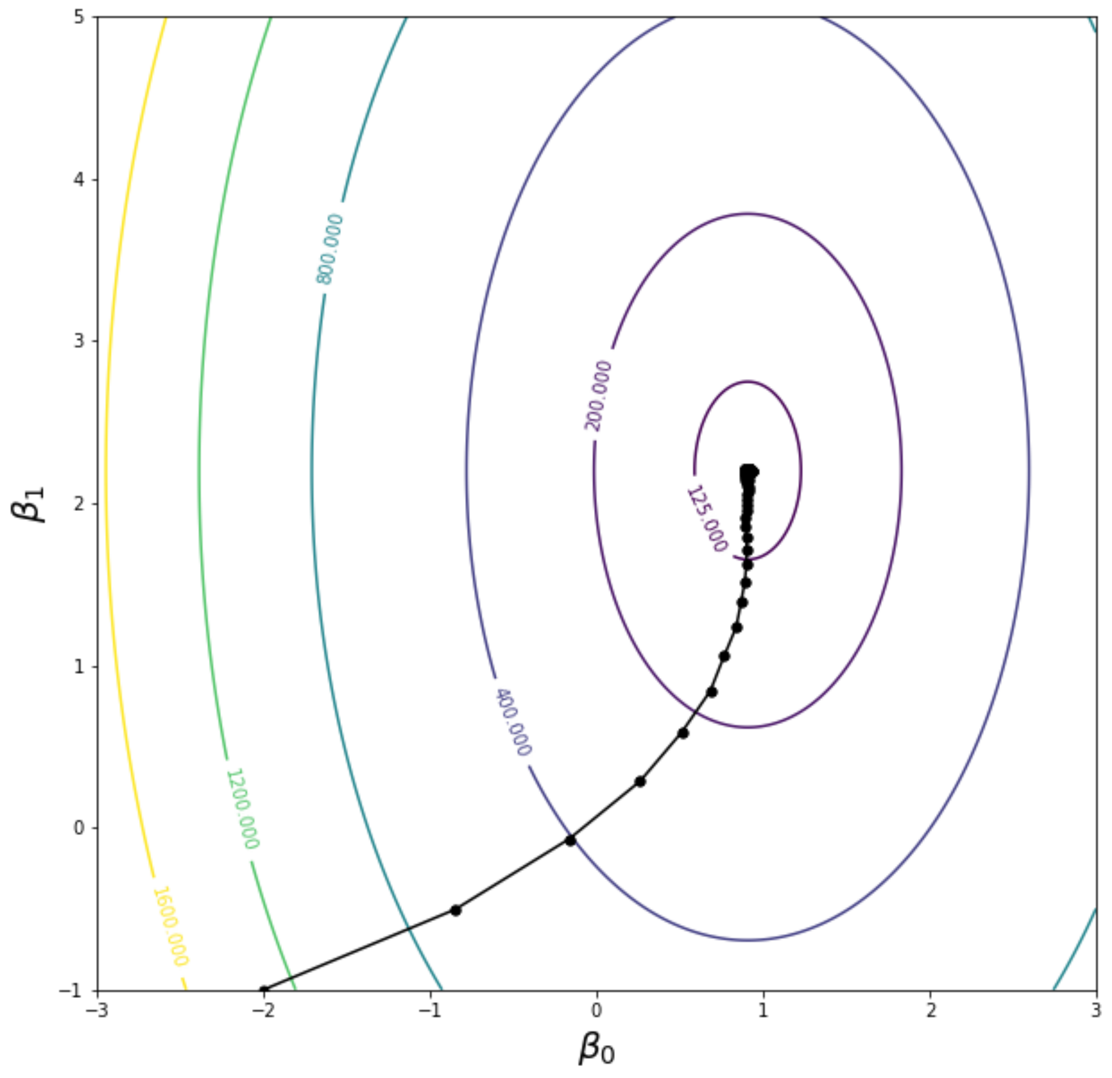
Note that the function parameter beta is a numpy array containing the initial guess for the solve. The numpy array bhist stores the approximation of the betas after each iteration for plotting and diagnostic purposes.
Look at the Peer Review assignment for a question about this section.

```
In [7]: def sgd(X, y, beta, eta=0.1, num_epochs=100):
            """
            Peform Stochastic Gradient Descent

            :param X: matrix of training features
            :param y: vector of training responses
            :param beta: initial guess for the parameters
            :param eta: the learning rate
            :param num_epochs: the number of epochs to run
            """

            # initialize history for plotting
            bhist = np.zeros((num_epochs+1, len(beta)))
            bhist[0,0], bhist[0,1] = beta[0], beta[1]

            # perform steps for all epochs
            for epoch in range(1, num_epochs+1):

                # shuffle indices (randomly)
                shuffled_inds = list(range(X.shape[0]))
                np.random.shuffle(shuffled_inds)

                # TODO: loop over training examples, update beta (beta[0] and bet
        a[1]) as per the above formulas
                # your code here
                for i in shuffled_inds:
                    xi = X[i, 1]  # feature value
                    yi = y[i]     # target value
                    y_pred = beta[0] + beta[1] * xi
                    error = y_pred - yi
                    beta[0] -= eta * 2 * error
                    beta[1] -= eta * 2 * error * xi

                # save history
                bhist[epoch, :] = beta

            # return bhist. Last row
            # are the learned parameters.
            return bhist
```

```
In [8]: # SGD Test for 2 features
        np.random.seed(42)

        mock_X = np.array([[ 1., -1.], [ 1., -0.97979798], [ 1., -0.95959596], [
        1., -0.93939394]])
        mock_y = np.array([-1.09375848, -2.65894663, -0.51463485, -2.27442244])
        mock_beta_start = np.array([-2.0, -1.0])

        mock_bhist_exp = np.array([[-2., -1.], [-2.01174521, -0.98867152], [-2.02
        304238, -0.97777761], [-2.03400439, -0.96720934]])
        mock_bhist_act = sgd(mock_X, mock_y, beta=mock_beta_start, eta=0.0025, nu
        m_epochs=3)

        for exp, act in zip(mock_bhist_exp, mock_bhist_act):
            assert pytest.approx(exp, 0.0001) == act, "Check sgd function"
```

```
In [9]: # Start at (-2,1)
        beta_start = np.array([-2.0, -1.0])

        # Training
        %time bhist = sgd(X_train, y_train, beta=beta_start, eta=0.0025, num_epoc
        hs=1000) # old = 0.0025

        # Print and Plot
        print("beta_0 = {:.5f}, beta_1 = {:.5f}".format(bhist[-1][0], bhist[-1]
        [1]))
        plotsurface(X_train, y_train, bhist=bhist)
```

```
CPU times: user 197 ms, sys: 8.3 ms, total: 205 ms
Wall time: 201 ms
beta_0 = 0.91899, beta_1 = 2.20488
```



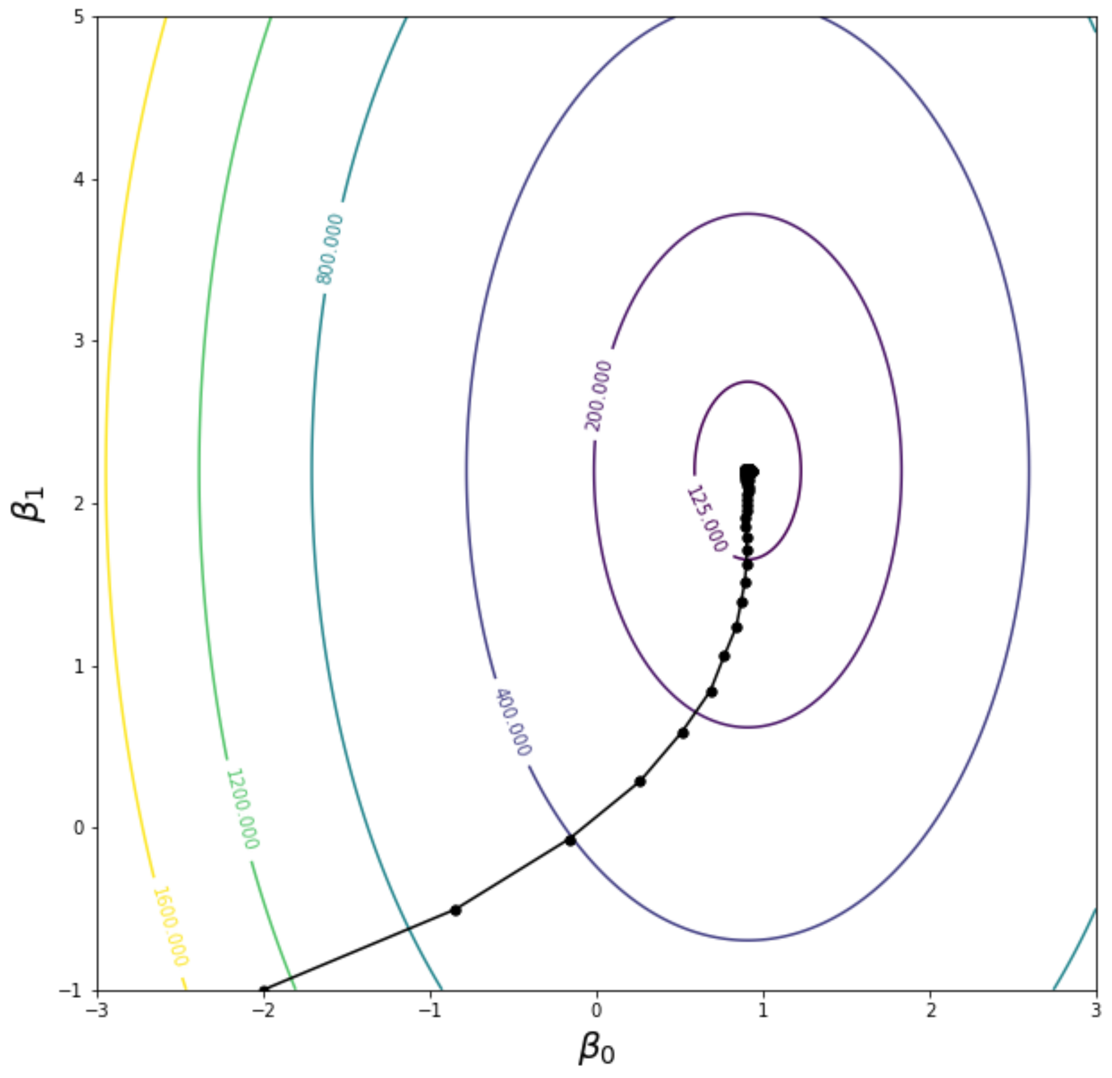
Part B: Thinking about the case where we have more than two features, can you think of a way to vectorize the stochastic gradient update of the parameters? When you see it, go back to the sgd function and improve it.

In [10]:
```
## TODO: rewrite/modify the sgd function below. Do not modify the previou
s sgd function, but write a new one here.
## Do not change the function name.
## The previous question worked for 2 features and this function is for m
ore than 2 features so update the earlier
# logic to work for any number of features
# your code here

def sgd(X, y, beta, eta=0.1, num_epochs=100):
    """
    Perform Stochastic Gradient Descent for any number of features.

    :param X: matrix of training features
    :param y: vector of training responses
    :param beta: initial guess for the parameters
    :param eta: the learning rate
    :param num_epochs: the number of epochs to run
    """

    # initialize history for plotting
    bhist = np.zeros((num_epochs+1, len(beta)))
    bhist[0, :] = beta

    # perform steps for all epochs
    for epoch in range(1, num_epochs+1):

        # shuffle indices (randomly)
        shuffled_inds = np.random.permutation(X.shape[0])

        # loop over training examples, update beta using vectorized opera
tions
        for i in shuffled_inds:
            xi = X[i, :]  # feature values
            yi = y[i]     # target value
            y_pred = np.dot(xi, beta)
            error = y_pred - yi
            beta -= eta * 2 * error * xi

        # save history
        bhist[epoch, :] = beta

    # return bhist. Last row contains the learned parameters.
    return bhist
```

```
In [11]: # SGD Test for more than 2 features
         np.random.seed(42)

         mock_X = np.array([[ 1., -1.], [ 1., -0.97979798], [ 1., -0.95959596], [
         1., -0.93939394]])
         mock_y = np.array([-1.09375848, -2.65894663, -0.51463485, -2.27442244])
         mock_beta_start = np.array([-2.0, -1.0])

         mock_bhist_exp = np.array([[-2., -1.], [-2.01174521, -0.98867152], [-2.02
         304238, -0.97777761], [-2.03400439, -0.96720934]])
         mock_bhist_act = sgd(mock_X, mock_y, beta=mock_beta_start, eta=0.0025, nu
         m_epochs=3)

         for exp, act in zip(mock_bhist_exp, mock_bhist_act):
             assert pytest.approx(exp, 0.0001) == act, "Check sgd function"
```

```
In [12]: # Start at (-2,1)
         beta_start = np.array([-2.0, -1.0])

         # Training
         %time bhist = sgd(X_train, y_train, beta=beta_start, eta=0.0025, num_epoc
         hs=1000)

         # Print and Plot
         print("beta_0 = {:.5f}, beta_1 = {:.5f}".format(bhist[-1][0], bhist[-1]
         [1]))
         plotsurface(X_train, y_train, bhist=bhist)
```

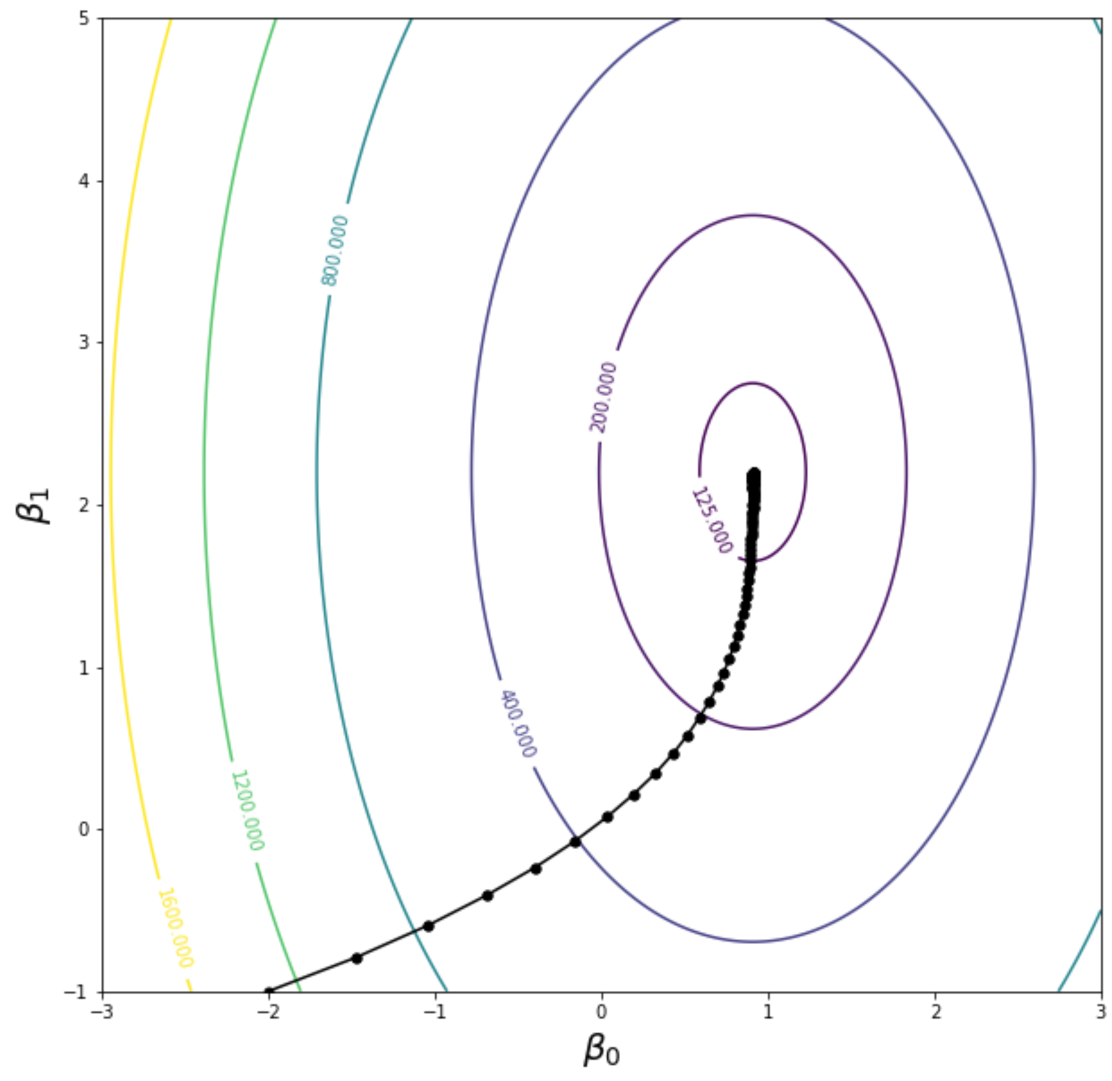
```
CPU times: user 580 ms, sys: 12.4 ms, total: 592 ms
Wall time: 581 ms
beta_0 = 0.91899, beta_1 = 2.20488
```

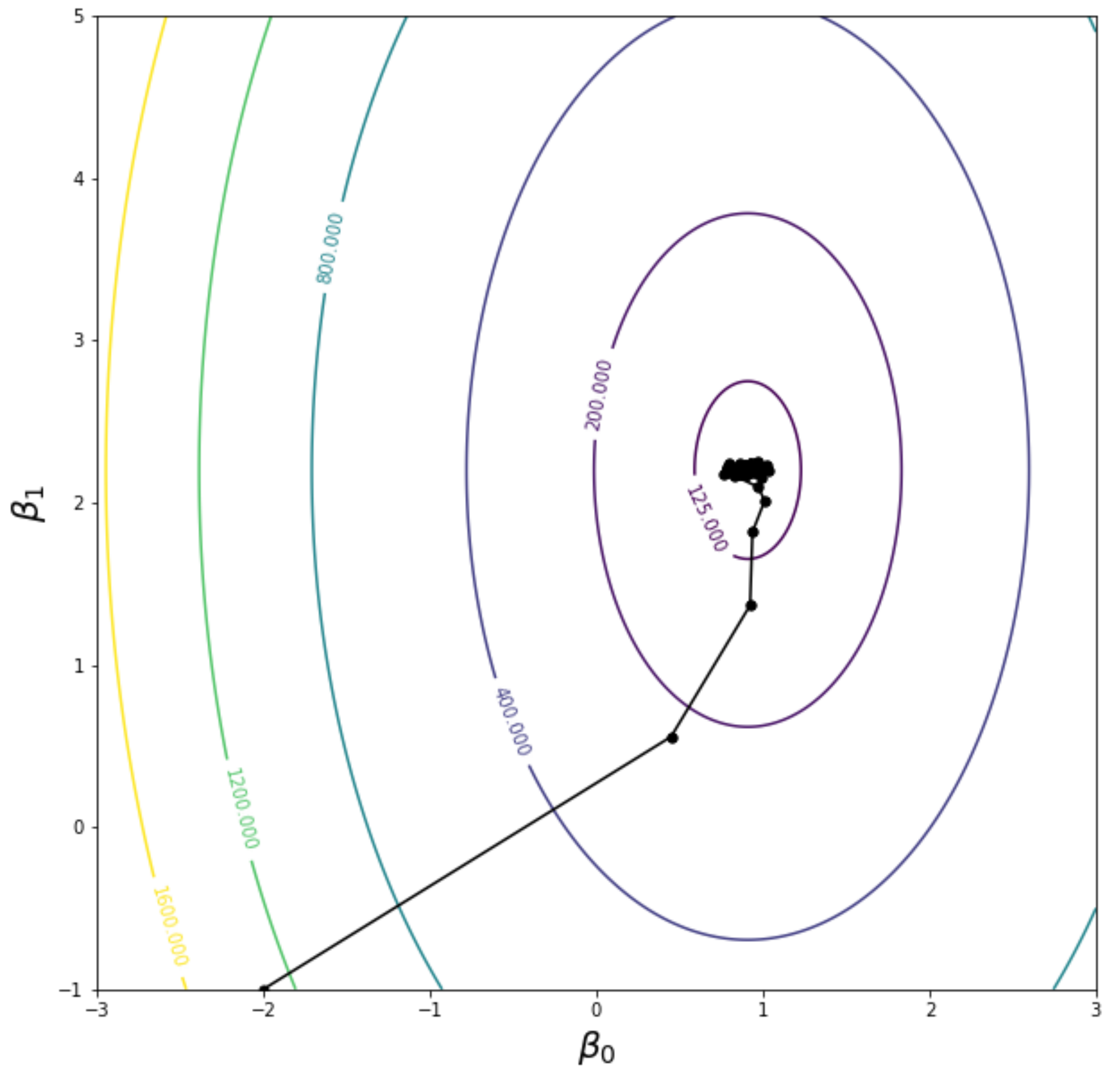


Part C: Now that you have created this beautiful solver, go back and break it by playing with the learning rate. Does the learning rate have the effect on convergence that you expect when visualized in the surface plot?

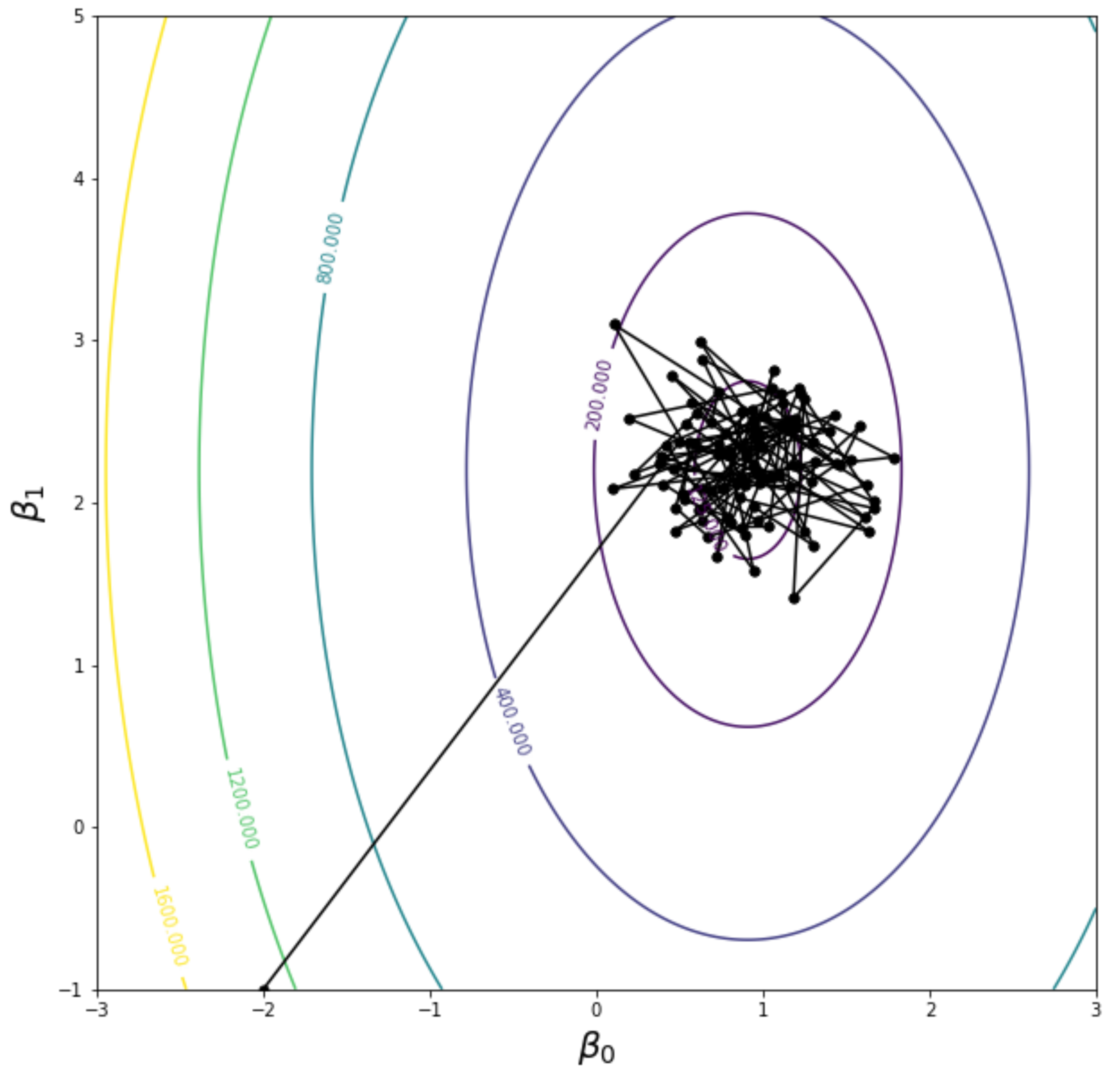
In [13]:
```python
# Generate synthetic data
x_train, x_valid, y_train, y_valid = dataGenerator(100)
X_train = np.column_stack((np.ones_like(x_train), x_train))
X_valid = np.column_stack((np.ones_like(x_valid), x_valid))

# Initial guess for beta
beta_start = np.array([-2.0, -1.0])

# Different learning rates to try
learning_rates = [0.001, 0.01, 0.1, 0.5]

for eta in learning_rates:
    # Run SGD with the given learning rate
    bhist = sgd(X_train, y_train, beta_start.copy(), eta=eta, num_epochs=
100)
    final_beta = bhist[-1]

    # Print final coefficients
    print(f"Learning rate: {eta}")
    print(f"SGD coefficients are: beta_0 = {final_beta[0]:.5f}, beta_1 =
{final_beta[1]:.5f}\n")

    # Plot the parameter updates
    plotsurface(X_train, y_train, bhist)
```

```
Learning rate: 0.001
SGD coefficients are: beta_0 = 0.91102, beta_1 = 2.19727

Learning rate: 0.01
SGD coefficients are: beta_0 = 0.89399, beta_1 = 2.19294

Learning rate: 0.1
SGD coefficients are: beta_0 = 1.05816, beta_1 = 2.15312

Learning rate: 0.5
SGD coefficients are: beta_0 = 0.78261, beta_1 = 0.61233
```

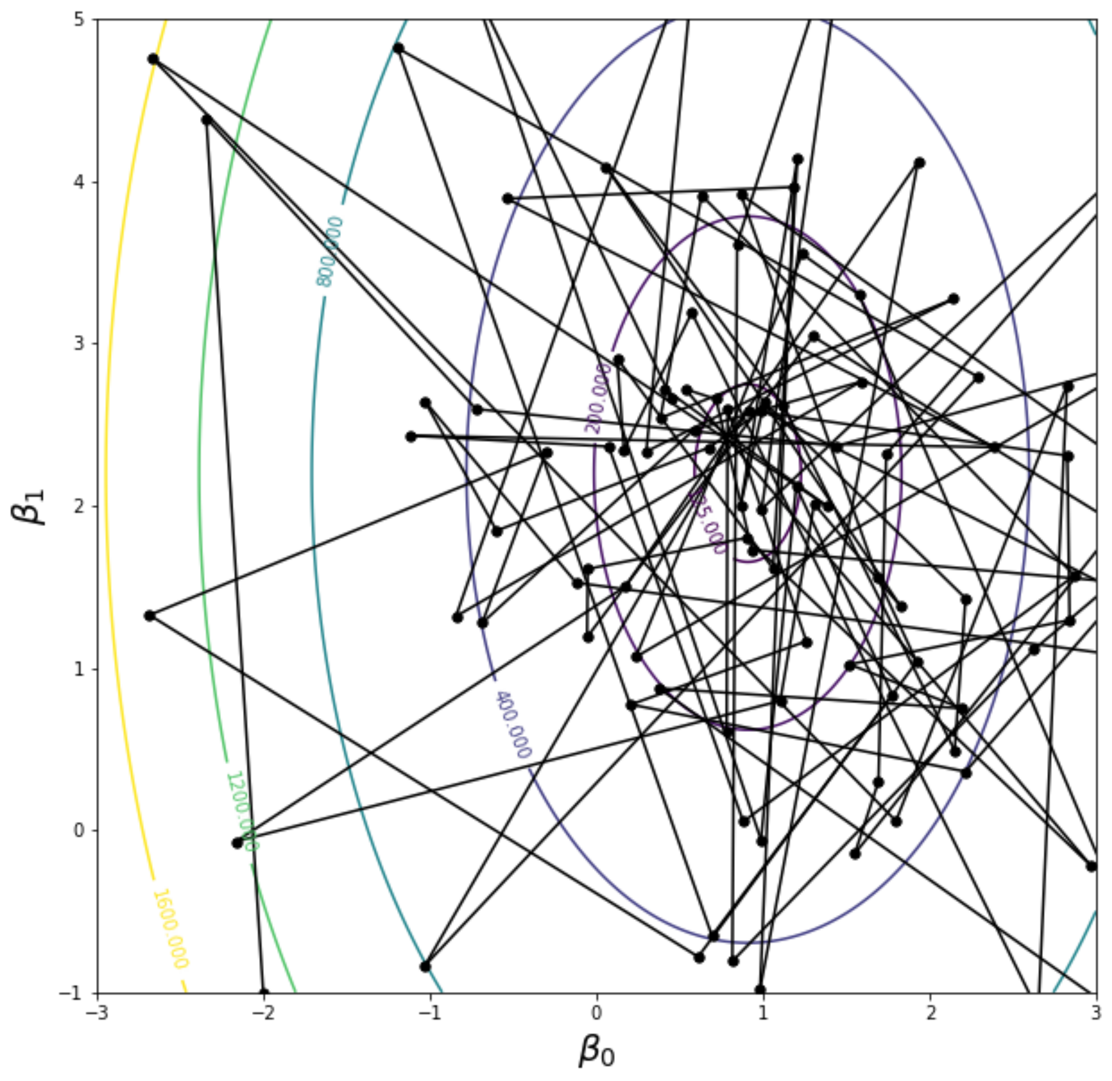




5
4
3
2
1
0
−1
−3
−2
−1
0
1
2
3
β_1
β_0
800.000
200.000
125.000
400.000
1200.000
1600.000



β_1
β_0
800.000
1200.000
1600.000
400.000
200.000
5
4
3
2
1
0
−1
−3
−2
−1
0
1
2
3



In [14]:
```
# Start at (-2,1)
# your code here
```

In [15]:
```
# Start at (-2,1)
# your code here
```

In [16]:
```
# Start at (-2,1)
# your code here
```

In [17]:
```
# Start at (-2,1)
# your code here
```

Expected Effects:

- Small Learning Rate: Convergence will be slow, and the parameters will take small steps towards the optimal values.
- Moderate Learning Rate: Convergence will be faster, and the parameters will reach the optimal values more quickly.
- High Learning Rate: The updates might overshoot, causing the parameters to oscillate or even diverge instead of converging to the optimal values.

Part 3: Graphical Diagnosis of Convergence

A common way to monitor the convergence of SGD and to tune hyperparameters (like learning rate and regularization strength) is to make a plot of how the loss function evolves during the training process. That is, we plot the value of the loss function periodically and see if it looks like it's reached a minimum, or see if it's jumping around a lot. Normally we'd record the value of the loss function as we train, but we'll use the beta histories returned by our solver. Finally, using the MSE instead of the RSS is a popular choice, so we'll do that.

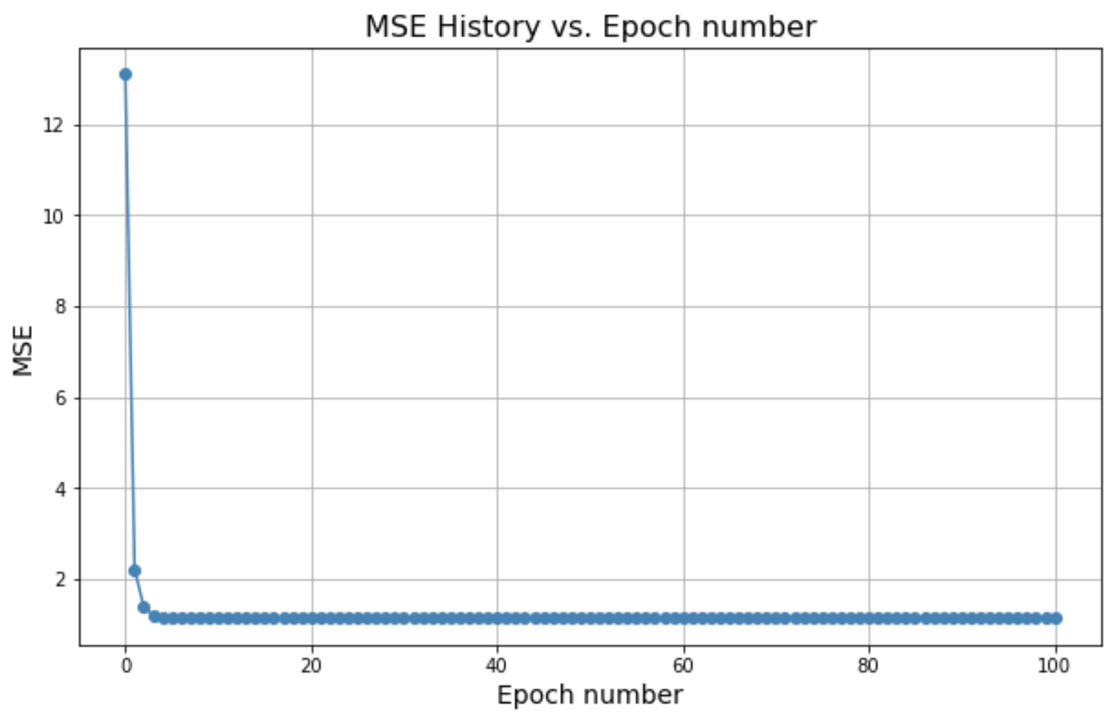
Part A: Modify the function below to take in a beta history and a data set and return a vector of MSE values for each epoch.

In [18]:
```
def MSE_hist(X, y, bhist):
    mse = np.zeros(bhist.shape[0])
    for epoch in range(bhist.shape[0]):
        beta = bhist[epoch]
        predictions = X.dot(beta)
        errors = predictions - y
        mse[epoch] = np.mean(errors ** 2)
    return mse
```

In [19]:
```
# MSE Tests
mock_X = np.array([[ 1., -1.], [ 1., -0.97979798], [ 1., -0.95959596], [
1., -0.93939394]])
mock_y = np.array([-1.09375848, -2.65894663, -0.51463485, -2.27442244])
mock_bhist = np.array([[-2., -1.], [-2.01174521, -0.98867152], [-2.023042
38, -0.97777761], [-2.03400439, -0.96720934]])

mock_mse_exp = np.array([1.1110145, 1.0840896, 1.05916951, 1.03590509])
mock_mse_act = MSE_hist(mock_X, mock_y, mock_bhist)

assert pytest.approx(mock_mse_exp, 0.0001 ) == mock_mse_act, "Check MSE_h
ist function"
```

Part B: Next we'll take the MSE history that we just computed and plot it vs epoch number. Based on your plot, would you say that your MSE has converged?

In [20]:
```
# plot MSE history vs. epoch number
# your code here
# Generate synthetic data
x_train, x_valid, y_train, y_valid = dataGenerator(100)
X_train = np.column_stack((np.ones_like(x_train), x_train))
X_valid = np.column_stack((np.ones_like(x_valid), x_valid))

# Initial guess for beta
beta_start = np.array([-2.0, -1.0])

# Parameters
learning_rate = 0.01
num_epochs = 100

# Run SGD
bhist = sgd(X_train, y_train, beta_start.copy(), eta=learning_rate, num_e
pochs=num_epochs)

# Compute MSE history
mse_history = MSE_hist(X_train, y_train, bhist)

# Plot MSE history vs. epoch number
plt.figure(figsize=(10, 6))
plt.plot(range(len(mse_history)), mse_history, color="steelblue", marker
="o")
plt.xlabel("Epoch number", fontsize=14)
plt.ylabel("MSE", fontsize=14)
plt.title("MSE History vs. Epoch number", fontsize=16)
plt.grid(True)
plt.show()

# Print final beta values
final_beta = bhist[-1]
print("SGD coefficients are: beta_0 = {:.5f}, beta_1 = {:.5f}".format(fin
al_beta[0], final_beta[1]))
```

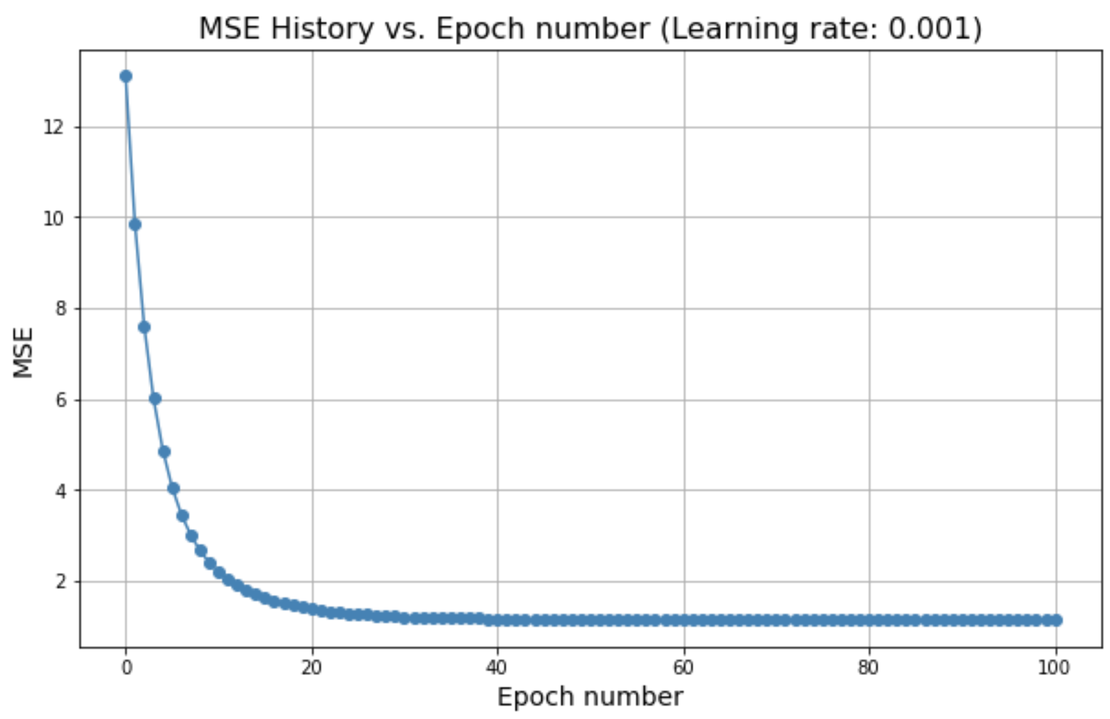


```
SGD coefficients are: beta_0 = 0.96463, beta_1 = 2.19088
```

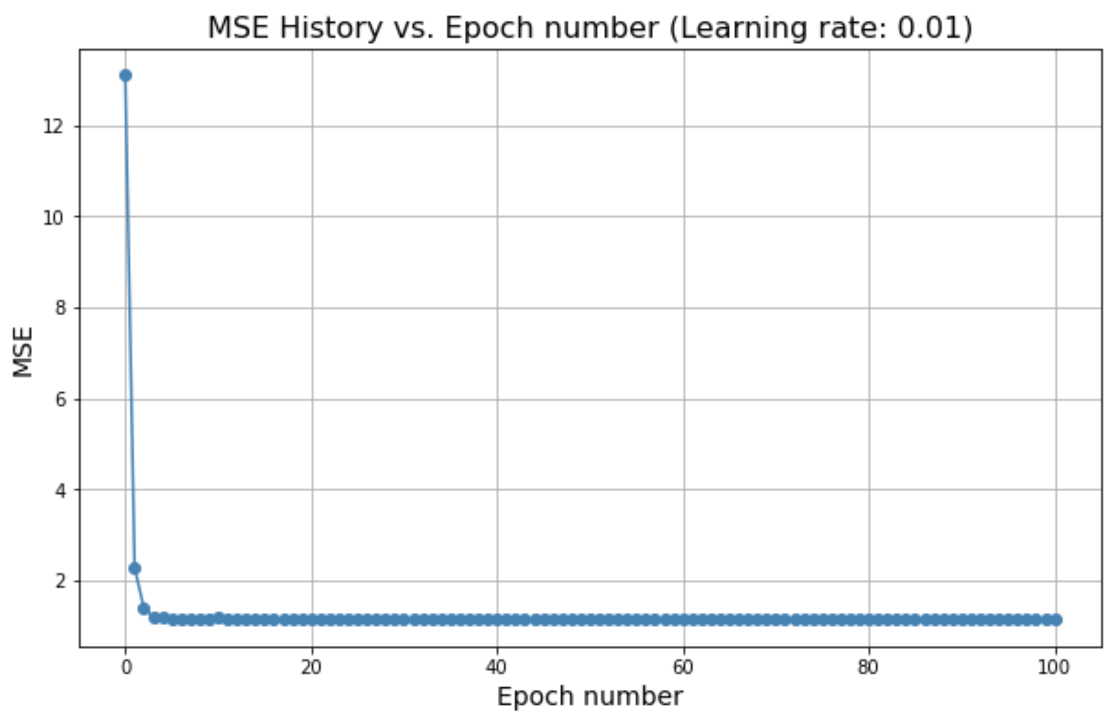
By running this code, we can observe how the MSE changes over the epochs and determine if it has converged. If the MSE stabilizes and stops decreasing significantly, it indicates that the model has converged. If the MSE fluctuates widely, it suggests that the learning rate might be too high, or the model hasn't converged properly.

Part C: Go back up and change the value of the learning rate to bigger and smaller values (you might also have to adjust the max epochs). Do the different learning rates have the effect on the MSE plots that you would expect?

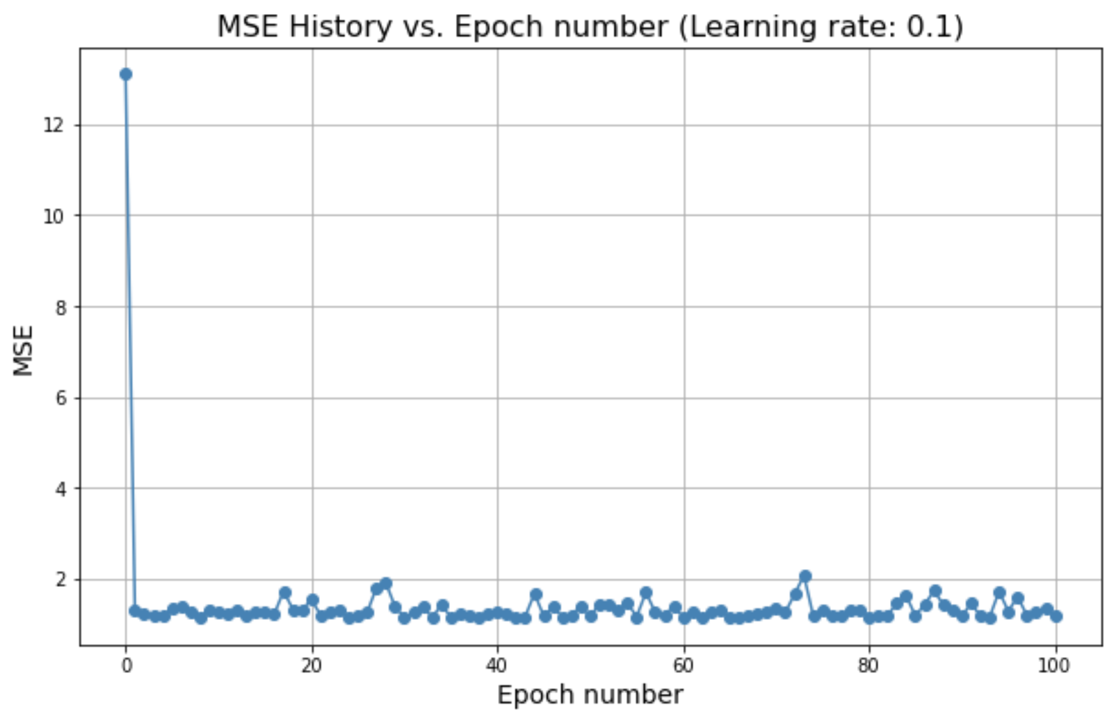
In [21]:
```
# TODO: change the value of the learning rate to bigger and smaller value
s, consider adjusting max epochs
# test plots
# your code here
# Generate synthetic data
x_train, x_valid, y_train, y_valid = dataGenerator(100)
X_train = np.column_stack((np.ones_like(x_train), x_train))
X_valid = np.column_stack((np.ones_like(x_valid), x_valid))

# Initial guess for beta
beta_start = np.array([-2.0, -1.0])

# Different learning rates to try
learning_rates = [0.001, 0.01, 0.1, 0.5]
num_epochs = 100

for eta in learning_rates:
    # Run SGD with the given learning rate
    bhist = sgd(X_train, y_train, beta_start.copy(), eta=eta, num_epochs=
num_epochs)

    # Compute MSE history
    mse_history = MSE_hist(X_train, y_train, bhist)

    # Plot MSE history vs. epoch number
    plt.figure(figsize=(10, 6))
    plt.plot(range(len(mse_history)), mse_history, color="steelblue", mar
ker="o")
    plt.xlabel("Epoch number", fontsize=14)
    plt.ylabel("MSE", fontsize=14)
    plt.title(f"MSE History vs. Epoch number (Learning rate: {eta})", fon
tsize=16)
    plt.grid(True)
    plt.show()

    # Print final beta values
    final_beta = bhist[-1]
    print(f"Learning rate: {eta}")
    print(f"SGD coefficients are: beta_0 = {final_beta[0]:.5f}, beta_1 =
{final_beta[1]:.5f}\n")
```



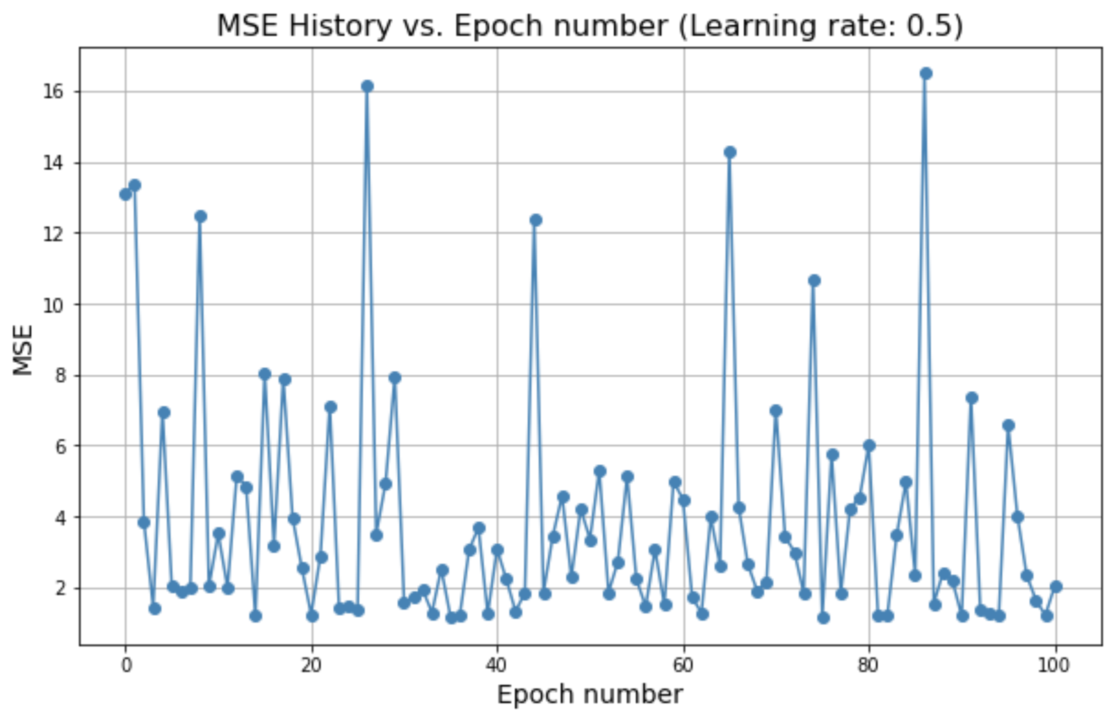
```
Learning rate: 0.001
SGD coefficients are: beta_0 = 0.91102, beta_1 = 2.19727
```



```
Learning rate: 0.01
SGD coefficients are: beta_0 = 0.89399, beta_1 = 2.19294
```



```
Learning rate: 0.1
SGD coefficients are: beta_0 = 1.05816, beta_1 = 2.15312
```

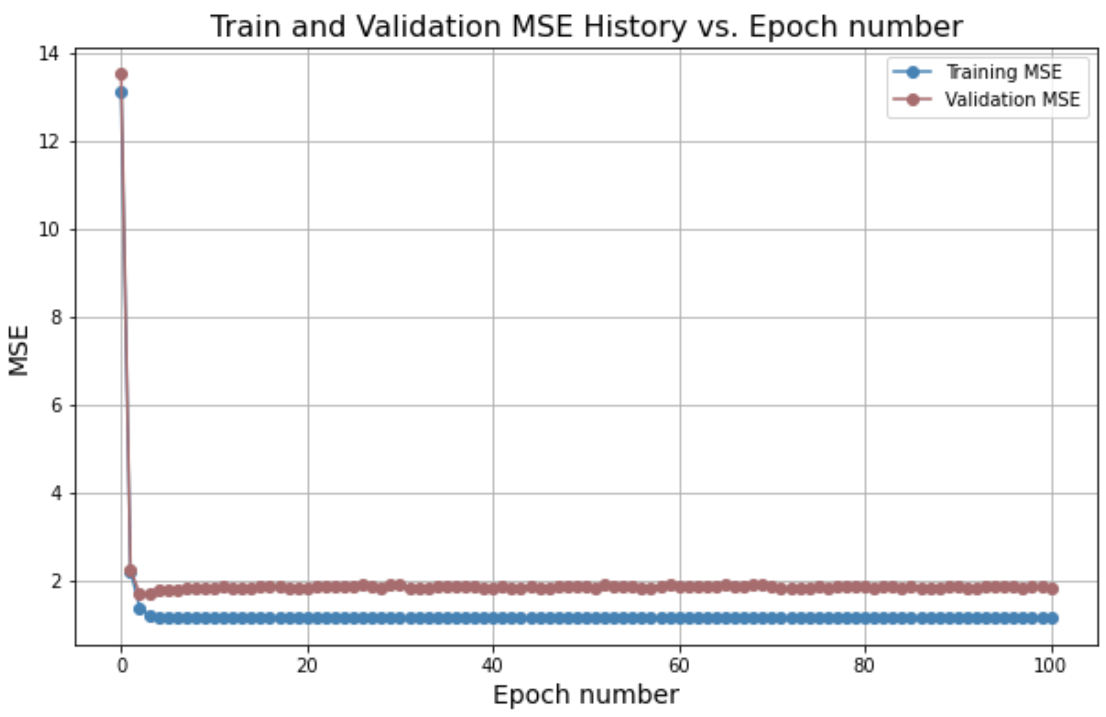


```
Learning rate: 0.5
SGD coefficients are: beta_0 = 0.78261, beta_1 = 0.61233
```

In [22]:
```
# continue testing plots for C
# your code here
```

Part D: Is the MSE on the training data the best thing to look at when deciding if our training algorithm has converged? Plot the train and validation MSE as a function of epochs. Discuss the result.

In [23]:
```
# plot train and validation MSE as function of epochs
# Start at (-2,1)
# your code here
def MSE_hist(X, y, bhist):
    mse = np.zeros(bhist.shape[0])
    for epoch in range(bhist.shape[0]):
        beta = bhist[epoch]
        predictions = X.dot(beta)
        errors = predictions - y
        mse[epoch] = np.mean(errors ** 2)
    return mse

# Generate synthetic data
x_train, x_valid, y_train, y_valid = dataGenerator(100)
X_train = np.column_stack((np.ones_like(x_train), x_train))
X_valid = np.column_stack((np.ones_like(x_valid), x_valid))

# Initial guess for beta
beta_start = np.array([-2.0, -1.0])

# Parameters
learning_rate = 0.01
num_epochs = 100

# Run SGD
bhist = sgd(X_train, y_train, beta_start.copy(), eta=learning_rate, num_e
pochs=num_epochs)

# Compute MSE history for both training and validation data
train_mse_history = MSE_hist(X_train, y_train, bhist)
valid_mse_history = MSE_hist(X_valid, y_valid, bhist)

# Plot MSE history for both training and validation data vs. epoch number
plt.figure(figsize=(10, 6))
plt.plot(range(len(train_mse_history)), train_mse_history, color="steelbl
ue", marker="o", label='Training MSE')
plt.plot(range(len(valid_mse_history)), valid_mse_history, color="#a76c6
e", marker="o", label='Validation MSE')
plt.xlabel("Epoch number", fontsize=14)
plt.ylabel("MSE", fontsize=14)
plt.title("Train and Validation MSE History vs. Epoch number", fontsize=1
6)
plt.legend()
plt.grid(True)
plt.show()
```


Train and Validation MSE History vs. Epoch number
MSE
Epoch number
Training MSE
Validation MSE
0
20
40
60
80
100
2
4
6
8
10
12
14

Discussion of Train and Validation MSE Results

The plot shows the Mean Squared Error (MSE) for both the training and validation datasets as a function of the number of epochs. Here's a detailed discussion of the results:

1. **Initial MSE Spike**:

    - There is a noticeable spike in the MSE values at the very beginning. This is typical because the initial beta values are likely far from the optimal values. The first few updates in SGD can cause significant changes in the model's predictions, leading to high initial errors.

1. **Convergence and Stability**:

    - After the initial spike, both the training and validation MSEs decrease rapidly and then stabilize. This indicates that the Stochastic Gradient Descent (SGD) algorithm is successfully minimizing the loss function and converging towards an optimal solution.
    - The training MSE stabilizes at a slightly lower value than the validation MSE. This is expected as the model is trained to minimize the training error specifically.
2. **Training vs. Validation MSE**:

    - The gap between the training and validation MSEs is small, which suggests that the model generalizes well to unseen data. If the validation MSE were significantly higher than the training MSE, it would indicate overfitting, where the model performs well on the training data but poorly on new data.
    - The slight increase and fluctuations in the validation MSE are normal and can be attributed to the inherent variability in the validation data and the stochastic nature of the SGD algorithm.
3. **Model Generalization**:

    - The plot indicates that the model has converged to a solution that generalizes well to the validation data. The stable and low validation MSE suggests that the model has learned the underlying pattern in the data and is not just memorizing the training examples.

Conclusion

The MSE plot effectively demonstrates the convergence of the SGD algorithm. The model appears to be well-tuned with the given learning rate and number of epochs. Both the training and validation MSEs are low and stable, indicating good generalization. Monitoring both training and validation MSEs is crucial for diagnosing convergence and ensuring that the model does not overfit the training data.