Homework 5: Modeling, Estimation and Gradient Descent

Due Date: Saturday 11/23, 11:59 PM

We will explore modeling through estimation and prediction. Along the way, we will get experience with an iterative method for guess-and-check called gradient descent. These approaches are helpful throughout data science particularly the field of machine learning. By completing Homework 5, you should take away...

- Practice reasoning about a model and building intuition for loss functions.
- Determining the gradient of a loss function with respect to model parameters and using the calculations for gradient descent.

We will apply these skills in Homework 6 to a real-world dataset.

Submission Instructions

For this assignment, you will submit a copy to Gradescope. Follow these steps

1. Download as HTML (`File->Download As->HTML(.html)`).
2. Open the HTML in the browser. Print to .pdf
3. Upload to Gradescope. Tag your answers.

Note that

- Please map your answers to our questions. Otherwise you may lose points. Please see the rubric below.
- You should break long lines of code into multiple lines. Otherwise your code will extend out of view from the cell. Consider using `\` followed by a new line.
- For each textual response, please include relevant code that informed your response. For each plotting question, please include the code used to generate the plot.
- You should not display large output cells such as all rows of a table. Instead convert the input cell from Code to Markdown back to Code to remove the output cell.

Moreover you will submit a copy on Jupyter Hub under Assignments Tab. You cannot access the extension in JupyterLab. So if the URL ends with lab, then please change it to tree

```
https://pds-f19.jupyter.hpc.nyu.edu/user/[Your NetID]/tree
```

Consult the instructional video

```
https://nbgrader.readthedocs.io/en/stable/_images/student_assignment.gif
```

for steps to...

1. fetch
2. modify
3. optionally validate
4. submit your project

Failure to follow these guidelines for submission could mean the deduction of two points. See the rubric below.

Collaboration Policy

Data science is a collaborative activity. While you may talk with others about the homework, we ask that you **write your solutions individually**. If you do discuss the assignments with others please **include their names** at the top of your solution.

Rubric

Question	Points
Gradescope	2
Question 1a	1
Question 1b	1
Question 1c	1
Question 1d	1
Question 1e	1
Question 2a	2
Question 2b	1
Question 2c	1
Question 2d	1
Question 2e	1
Question 2f	1
Question 3a	1
Question 3b	3
Question 3c	2
Question 4a	3
Question 4b	1
Question 4c	1
Question 4d	1
Question 4e	1
Question 5a	2
Question 5b	1
Question 5c	1
Question 5d	0

Getting Started

In [1]:
```
from IPython.display import display, Markdown

# Import standard packages
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import csv
import re
import seaborn as sns

# Set some parameters
plt.rcParams['figure.figsize'] = (12, 9)
plt.rcParams['font.size'] = 16
np.set_printoptions(4)
```

In [2]:
```
# Import a specialized plotting package useful for 3d
import plotly
import plotly.graph_objs as go
plotly.offline.init_notebook_mode(connected=True)
from hw5_utils import plot_3d

# Note that you may need to install plotly to import these packages
# Either execucte the next cell or complete the assignment on JupyterHub
```

In [3]:
```
# !pip install plotly
# !conda install -c conda-forge jupyterlab-plotly-extension
```

Load Data

Load the data.csv file into a pandas dataframe.
Note that we are reading the data directly from the URL address.

In [4]:
```
# Run this cell to load our sample data
data = pd.read_csv("data.csv", index_col=0)
data.head()
```

Out[4]:

	x	y
0	-5.000000	-8.262369
1	-4.966555	-7.692411
2	-4.933110	-7.358698
3	-4.899666	-8.067488
4	-4.866221	-7.834256

1: A Simple Model

Let's start by examining our data and creating a simple model that can represent this data.

Question 1

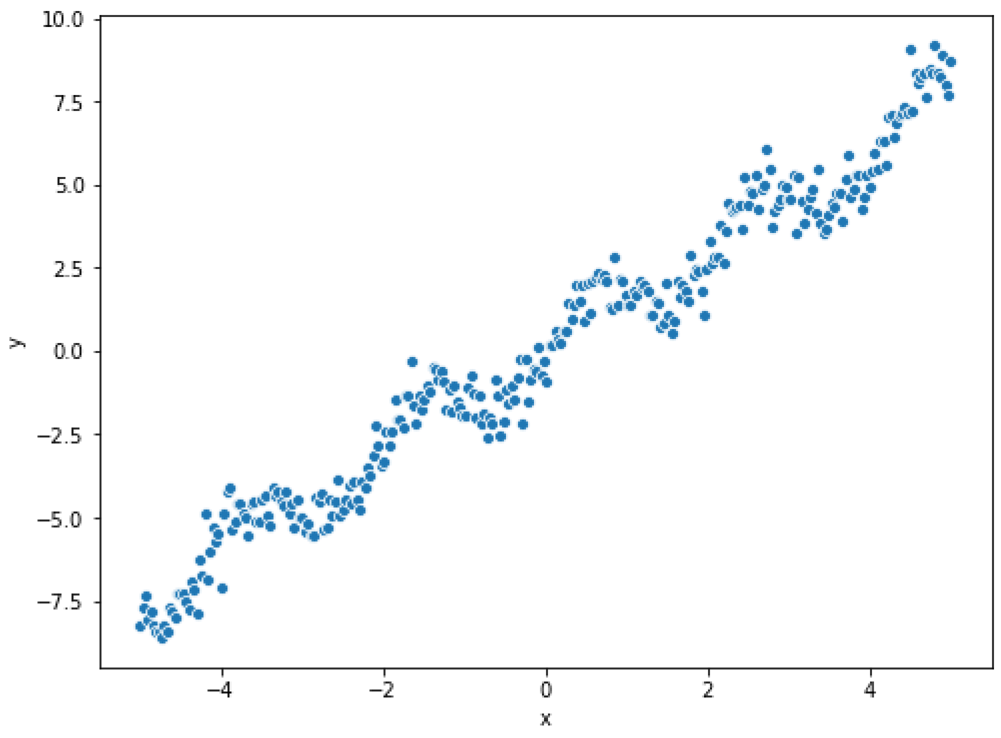
Question 1a

First, let's visualize the data in a scatter plot. After implementing the `scatter` function below, you should see something like this:

scatter

In [5]:
```
def scatter(x, y):
    """
    Generate a scatter plot using x and y

    Keyword arguments:
    x -- the vector of values x
    y -- the vector of values y
    """
    plt.figure(figsize=(8, 6))
    plt.figure(figsize=(8, 6))
    return sns.scatterplot(x,y)
    raise NotImplementedError()

x = data['x']
y = data['y']
scatter(x,y)
```

Out[5]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x2b7dc9da1f28>

<Figure size 576x432 with 0 Axes>
```



Question 1b

Describe any significant observations about the distribution of the data. How can you describe the relationship between x and y?

There seems to be a linear positive relationship, with a dip around every 2 increments of x.

Question 1c

The data looks roughly linear. However it moves up and down away from the line. For now, let's assume that the data follows some underlying linear model. We define the underlying linear model that predicts the value y using the value x as: $f_{\theta^*}(x) = \theta^* \cdot x$

Since we cannot find the value of the population parameter θ^* exactly, we will assume that our dataset approximates our population and use our dataset to estimate θ^*. We denote our estimation with θ, our fitted estimation with $\hat{\theta}$, and our model as:

$$f_\theta(x) = \theta \cdot x$$

Based on this equation, define the linear model function `linear_model` below to estimate $\mathbf{y}$ (the y-values) given $\mathbf{x}$ (the x-values) and θ. This model is similar to the model you defined in Lab 5: Modeling and Estimation.

In [6]:
```
def linear_model(x, theta):
    """
    Returns the estimate of y given x and theta

    Keyword arguments:
    x -- the vector of values x
    theta -- the scalar theta
    """
    y = theta*x
    return y
    raise NotImplementedError()
```

In [7]:
```
assert linear_model(0, 1) == 0
assert np.sum(linear_model(np.array([3, 5]), 3)) == 24
```

In [ ]:
```

```

Question 1d

In class, we learned that the square loss, sometimes called the L^2 loss. Let's use L^2 loss to evaluate our estimate θ, which we will use later to identify an optimal θ, represented as $\hat{\theta}$. Define the L^2 loss function `l2_loss` below.

In [8]:
```
def l2_loss(y, y_hat):
    """
    Returns the average l2 loss given y and y_hat

    Keyword arguments:
    y -- the vector of true values y
    y_hat -- the vector of predicted values y_hat
    """
    return np.mean((y - y_hat)**2)
```

In [9]:
```
assert l2_loss(2, 1) == 1
assert l2_loss(2, 0) == 4
assert l2_loss(np.array([5, 6]), np.array([1, 1])) == 20.5
```

In [ ]:
```
```

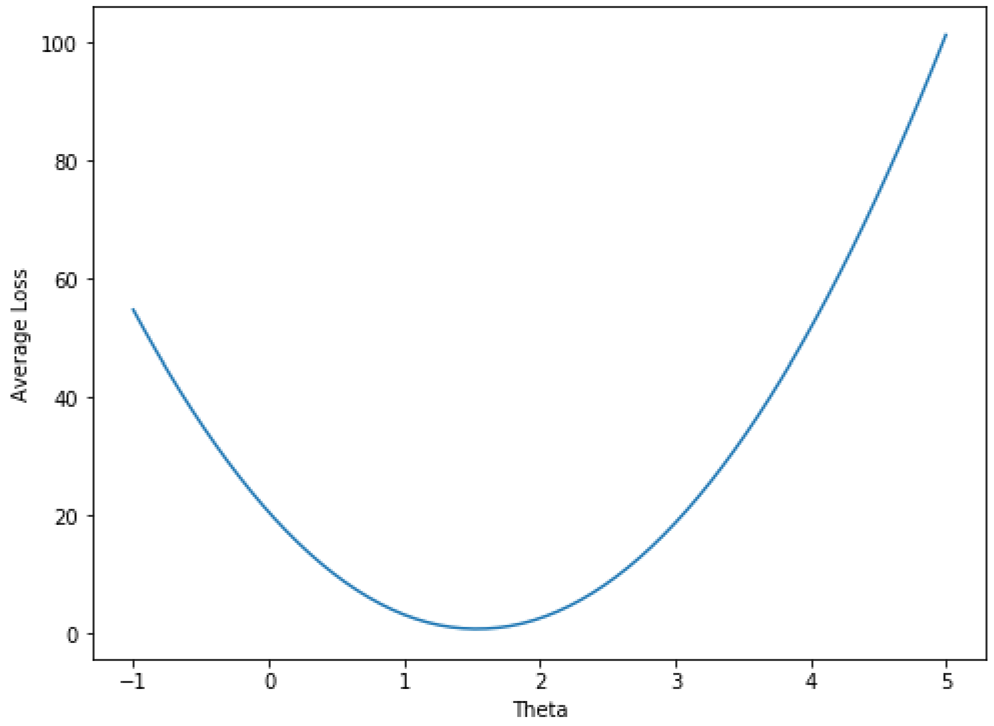
Question 1e

First, visualize the L^2 loss as a function of θ, where several different values of θ are given. Be sure to label your axes properly. You plot should look something like this:

avg_l2

What looks like the optimal value, $\hat{\theta}$, based on the visualization? Set `theta_star_guess` to the value of θ that appears to minimize our loss.

In [10]:
```python
def visualize(x, y, thetas):
    """
    Plots the average l2 loss for given x, y as a function of theta.
    Use the functions you wrote for linear_model and l2_loss.

    Keyword arguments:
    x -- the vector of values x
    y -- the vector of values y
    thetas -- an array containing different estimates of the scalar theta
    """
    avg_loss = np.array([l2_loss(y, linear_model(theta,x)) for theta in thetas
])

    plt.figure(figsize=(8,6))
    plt.xlabel(r"Theta")
    plt.ylabel(r"Average Loss")

    return plt.plot(thetas, avg_loss)
    raise NotImplementedError()

thetas = np.linspace(-1, 5, 70)
visualize(x, y, thetas)

theta_star_guess = 1.5
# YOUR CODE HERE
#raise NotImplementedError()
```



In [11]:
```python
assert l2_loss(3, 2) == 1
assert l2_loss(0, 10) == 100
```

In [ ]:
```python
```

2: Fitting our Simple Model

Now that we have defined a simple linear model and loss function, let's begin working on fitting our model to the data.

Question 2

Let's confirm our visual findings for optimal $\hat{\theta}$.

Question 2a

First, find the analytical solution for the optimal $\hat{\theta}$ for average L^2 loss. Write up your solution in the cell below using LaTex.

Hint: notice that we now have $\mathbf{x}$ and $\mathbf{y}$ instead of x and y. This means that when writing the loss function $L(\mathbf{x}, \mathbf{y}, \theta)$, you'll need to take the average of the squared losses for each y_i, $f_\theta(x_i)$ pair.

For tips on getting started, see [chapter 10 (https://www.textbook.ds100.org/ch/10/modeling_loss_functions.html)](https://www.textbook.ds100.org/ch/10/modeling_loss_functions.html) of the textbook. Note that if you check the [source (https://github.com/DS-100/textbook/tree/master/content/ch/10)](https://github.com/DS-100/textbook/tree/master/content/ch/10), you can access the LaTeX code of the book chapter, which you might find handy for typing up your work. Show your work, i.e. don't just write the answer.

$$n = number of datapoints$$

$$L(x, y, \theta) = \sum \frac{((y - f(x;\theta))^2)}{n}$$

$$\frac{dL}{d\theta} = \sum \frac{(2*(y - f(x;\theta)) * f'(x;\theta))}{n} = 0$$

Question 2b

Now that we have the analytic solution for $\hat{\theta}$, implement the function `find_theta` that calculates the numerical value of $\hat{\theta}$ based on our data $\mathbf{x}$, $\mathbf{y}$.

```
In [12]: def find_theta(x, y):
             """
             Find optimal theta given x and y

             Keyword arguments:
             x -- the vector of values x
             y -- the vector of values y
             """
             theta_opt = np.sum(x*y)/np.sum(x**2)
             return theta_opt
             raise NotImplementedError()
```

In [13]:
```
t_hat = find_theta(x, y)
print(f'theta_opt = {t_hat}')

assert 1.4 <= t_hat <= 1.6
```

```
theta_opt = 1.5372874874787739
```

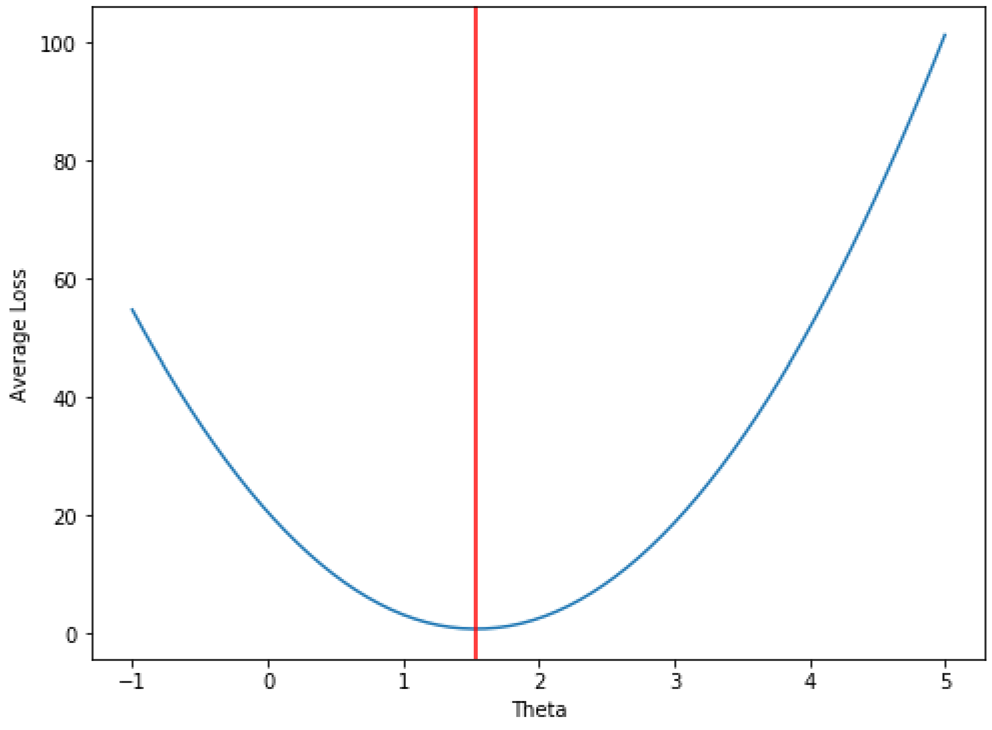
Question 2c

Now, let's plot our loss function again using the `visualize` function. But this time, add a vertical line at the optimal value of theta (plot the line $x = \hat{\theta}$). Your plot should look something like this:

vertical_linear

In [14]:
```
### BEGIN SOLUTION
theta_opt = find_theta(x,y)
visualize(x,y,thetas)
plt.axvline(theta_opt, color="red")
```

Out[14]:
```
<matplotlib.lines.Line2D at 0x2b7dcd33db70>
```



Question 2d

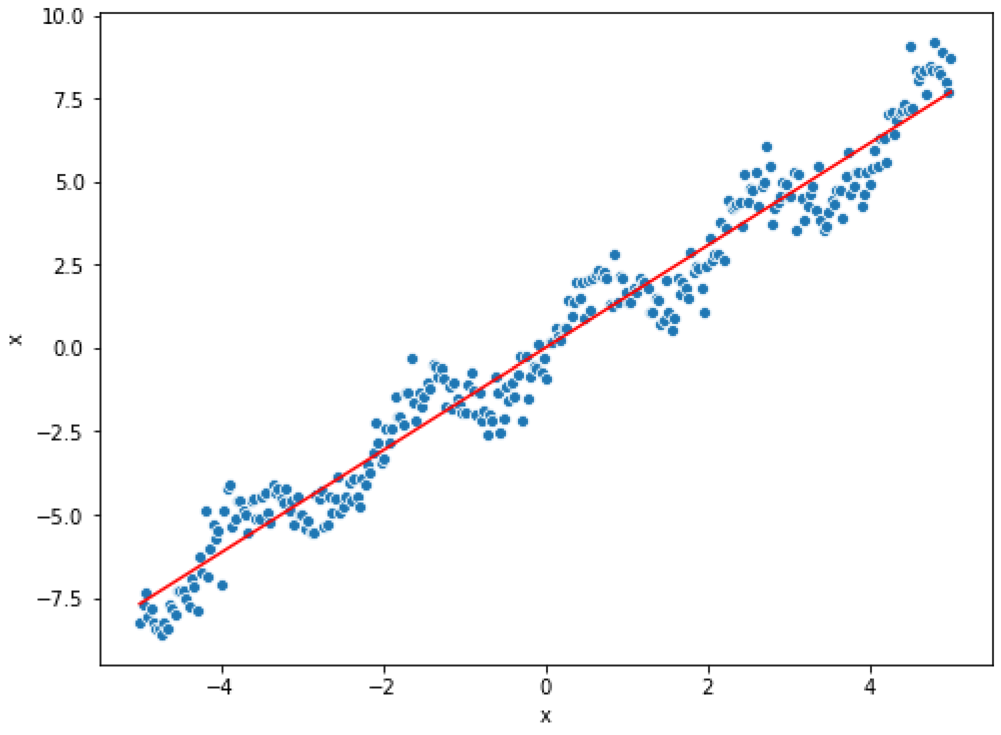
We now have an optimal value for θ that minimizes our loss. In the cell below, plot the scatter plot of the data from Question 1a (you can reuse the `scatter` function here). But this time, add the line $f_{\hat{\theta}}(x) = \hat{\theta} \cdot \mathbf{x}$ using the $\hat{\theta}$ you computed above. Your plot should look something like this:

scatter_with_line

In [15]:
```
theta_opt = find_theta(x, y)
scatter(x,y)
sns.lineplot(x=x,y=theta_opt*x, color='red')
```

Out[15]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x2b7dcd31aac8>
```

```
<Figure size 576x432 with 0 Axes>
```



Question 2e

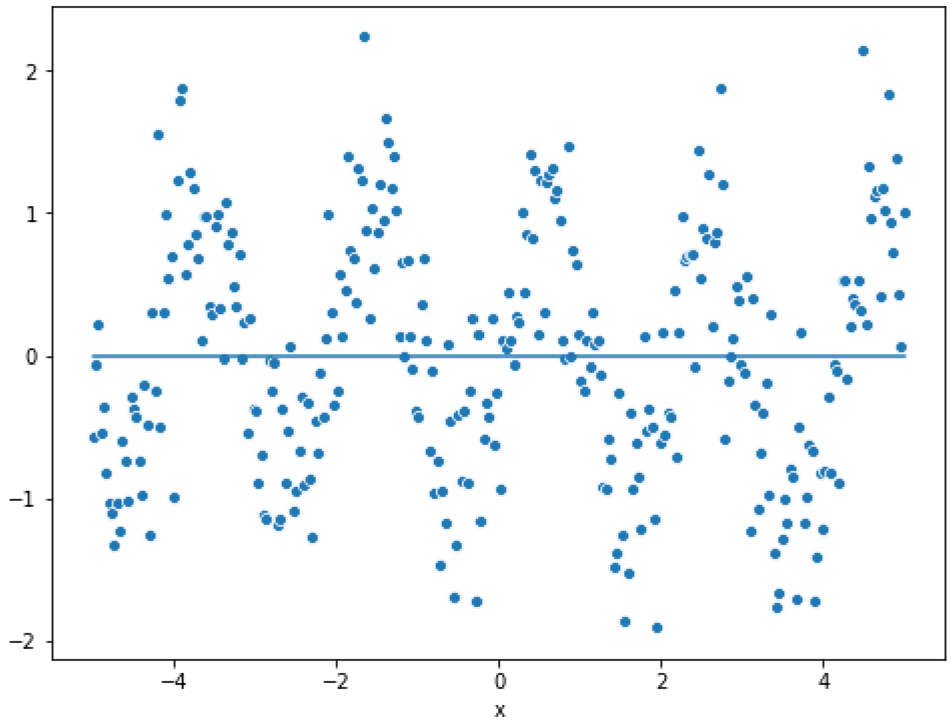
Great! It looks like our estimator $f_{\hat{\theta}}(x)$ is able to capture a lot of the data with a single parameter θ. Now let's try to remove the linear portion of our model from the data to see if we missed anything.

The remaining data is known as the residual, $\mathbf{r} = \mathbf{y} - \hat{\theta} \cdot \mathbf{x}$. Below, write a function to find the residual and plot the residuals corresponding to x in a scatter plot. Plot a horizontal line at $y = 0$ to assist visualization.

In [16]:
```
def visualize_residual(x, y):
    """
    Plot a scatter plot of the residuals, the remaining
    values after removing the linear model from our data.

    Keyword arguments:
    x -- the vector of values x
    y -- the vector of values y
    """
    theta_opt = find_theta(x, y)
    scatter(x,(y-theta_opt*x))
    return sns.lineplot(x=x,y=0)
    raise NotImplementedError()

visualize_residual(x, y)
```

Out[16]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x2b7dcd3885c0>
```

```
<Figure size 576x432 with 0 Axes>
```



Question 2f

What does the residual look like? Do you notice a relationship between x and r?

Notice the residuals increase in magnitude for every dip in the linear regression model. This looks like a shifted sin curve.

3: Increasing Model Complexity

It looks like the remaining data is sinusoidal, meaning our original data follows a linear function and a sinusoidal function. Let's define a new model to address this discovery and find optimal parameters to best fit the data:

$$f_{\boldsymbol{\theta}}(x) = \theta_1 x + sin(\theta_2 x)$$

Now, our model is parameterized by both θ_1 and θ_2, or composed together, $\boldsymbol{\theta}$.

Note that a generalized sine function $a\sin(bx+c)$ has three parameters: amplitude scaling parameter a, frequency parameter b and phase shifting parameter c. Looking at the residual plot above, it looks like the residual is zero at x = 0, and the residual swings between -1 and 1. Thus, it seems reasonable to effectively set the scaling and phase shifting parameter (a and c in this case) to 1 and 0 respectively. While we could try to fit a and c, we're unlikely to get much benefit. When you're done with the homework, you can try adding a and c to our model and fitting these values to see if you can get a better loss.

Question 3a

As in Question 1, fill in the `sin_model` function that predicts $\mathbf{y}$ (the y-values) using $\mathbf{x}$ (the x-values), but this time based on our new equation.

Hint: Try to do this without using for loops. The `np.sin` function may help you.

In [17]:
```
def sin_model(x, theta_1, theta_2):
    """
    Predict the estimate of y given x, theta_1, theta_2

    Keyword arguments:
    x -- the vector of values x
    theta_1 -- the scalar value theta_1
    theta_2 -- the scalar value theta_2
    """
    y = theta_1*x +np.sin(theta_2*x)
    return y
    raise NotImplementedError()
```

In [18]:
```
assert np.isclose(sin_model(1, 1, np.pi), 1.0000000000000002)
# Check that we accept x as arrays
assert len(sin_model(x, 2, 2)) > 1
```

Question 3b

Use the average L^2 loss to compute $\frac{\partial L}{\partial \theta_1}, \frac{\partial L}{\partial \theta_2}$.

First, we will use LaTex to write $L(\mathbf{x}, \mathbf{y}, \theta_1, \theta_2)$, $\frac{\partial L}{\partial \theta_1}$, and $\frac{\partial L}{\partial \theta_2}$ given $\mathbf{x}$, $\mathbf{y}$, $\boldsymbol{\theta}$.

You don't need to write out the full derivation. Just the final expression is fine.

$\frac{dL]{d\theta1} = -2(y-\theta1x - sin(\theta2*x))x$

$\frac{dL}{d\theta 2} = -2(y - \theta 1x - sin(\theta 2 * x) * cos(\theta 2 * x) * x$

Question 3c

Now, implement the functions `dt1` and `dt2` , which should compute $\frac{\partial L}{\partial \theta_1}$ and $\frac{\partial L}{\partial \theta_2}$ respectively. Use the formulas you wrote for $\frac{\partial L}{\partial \theta_1}$ and $\frac{\partial L}{\partial \theta_2}$ in the previous exercise. In the functions below, the parameter `theta` is a vector that looks like (θ_1, θ_2).

Note: To keep your code a bit more concise, be aware that `np.mean` does the same thing as `np.sum` divided by the length of the numpy array.

In [19]:
```
def dt1(x, y, theta):
    """
    Compute the numerical value of the partial of l2 loss with respect to thet
a_1

    Keyword arguments:
    x -- the vector of all x values
    y -- the vector of all y values
    theta -- the vector of values theta
    """
    return np.mean(-2*(y-theta[0]*x-np.sin(theta[1]*x))*x)
    raise NotImplementedError()
```

In [20]:
```
def dt2(x, y, theta):
    """
    Compute the numerical value of the partial of l2 loss with respect to thet
a_2

    Keyword arguments:
    x -- the vector of all x values
    y -- the vector of all y values
    theta -- the vector of values theta
    """
    return np.mean(-2*(y-theta[0]*x-np.sin(theta[1]*x))*x*np.cos(theta[1]*x))
    raise NotImplementedError()
```

In [21]:
```
# This function calls dt1 and dt2 and returns the gradient dt. It is already i
mplemented for you.
def dt(x, y, theta):
    """
    Returns the gradient of l2 loss with respect to vector theta

    Keyword arguments:
    x -- the vector of values x
    y -- the vector of values y
    theta -- the vector of values theta
    """
    return np.array([dt1(x,y,theta), dt2(x,y,theta)])
```

In [22]:
```
assert np.isclose(dt1(x, y, [0, np.pi]), -25.376660670924529, rtol=0.5)
```

In [ ]:
```
```

4: Gradient Descent

We cannot try to solve for the optimal $\hat{\theta}$ exactly. We resort to an algorithm for guess-and-check to iteratively find an approximate solution. So let's try implementing gradient descent.

Question 4

Question 4a

Implement the `grad_desc` function that performs gradient descent for a finite number of iterations. This function takes in an array for $\mathbf{x}$ (`x`), an array for $\mathbf{y}$ (`y`), and an initial value for θ (`theta`). `alpha` will be the learning rate (or step size, whichever term you prefer). In this part, we'll use a static learning rate that is the same at every time step.

At each time step, use the gradient and `alpha` to update your current `theta` . Also at each time step, be sure to save the current `theta` in `theta_history` , along with the L^2 loss (computed with the current `theta`) in `loss_history` .

Hints:

- Write out the gradient update equation (1 step). What variables will you need for each gradient update? Of these variables, which ones do you already have, and which ones will you need to recompute at each time step?
- You may need a loop here to update `theta` several times
- Recall that the gradient descent update function follows the form:

$$\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} - \alpha \left(\nabla_{\boldsymbol{\theta}} \mathbf{L}(\mathbf{x}, \mathbf{y}, \boldsymbol{\theta}^{(t)}) \right)$$

In [23]:
```
# Run me
def init_t():
    """Creates an initial theta [0, 0] of shape (2,) as a starting point for g
radient descent"""
    return np.zeros((2,))
```

In [24]:
```
def grad_desc(x, y, theta, num_iter=20, alpha=0.1):
    """
    Run gradient descent update for a finite number of iterations and static l
earning rate

    Keyword arguments:
    x -- the vector of values x
    y -- the vector of values y
    theta -- the vector of values theta to use at first iteration
    num_iter -- the max number of iterations
    alpha -- the learning rate (also called the step size)

    Return:
    theta -- the optimal value of theta after num_iter of gradient descent
    theta_history -- the series of theta values over each iteration of gradien
t descent
    loss_history -- the series of loss values over each iteration of gradient
 descent
    """
    theta_history = []
    loss_history = []

    for i in range(num_iter):
        change = dt(x,y,theta)
        theta = theta - (change*alpha)
        theta_history.append(theta)
        L = y - (theta[0]*x + np.sin(theta[1]*x))
        loss_history.append(sum((L[:] * L[:]))/len(x))

    return theta, theta_history, loss_history
    raise NotImplementedError()
```

In [25]:
```
t = init_t()
t_est, ts, loss = grad_desc(x, y, t, num_iter=20, alpha=0.1)

assert len(ts) == len(loss) == 20 # theta history and loss history are 20 item
s in them
assert ts[0].shape == (2,) # theta history contains theta values
assert np.isscalar(loss[0]) # loss history is a list of scalar values, not vec
tor
assert loss[1] - loss[-1] > 0 # loss is decreasing
```

In [ ]:

Question 4b

Now, let's try using a decaying learning rate. Implement `grad_desc_decay` below, which performs gradient descent with a learning rate that decreases slightly with each time step. You should be able to copy most of your work from the previous part, but you'll need to tweak how you update `theta` at each time step.

By decaying learning rate, we mean instead of just a number α, the learning should be now $\frac{\alpha}{i+1}$ where i is the current number of iteration. (Why do we need to add '+ 1' in the denominator?)

In [26]:
```
def grad_desc_decay(x, y, theta, num_iter=20, alpha=0.1):
    """
    Run gradient descent update for a finite number of iterations and decaying
learning rate

    Keyword arguments:
    x -- the vector of values x
    y -- the vector of values y
    theta -- the vector of values theta
    num_iter -- the max number of iterations
    alpha -- the learning rate

    Return:
    theta -- the optimal value of theta after num_iter of gradient descent
    theta_history -- the series of theta values over each iteration of gradien
t descent
    loss_history -- the series of loss values over each iteration of gradient
 descent
    """
    theta_history = []
    loss_history = []

    for i in range(num_iter):
        change = dt(x,y,theta)
        theta = theta - (change*(alpha/(i+1)))
        theta_history.append(theta)
        L = y - (theta[0]*x + np.sin(theta[1]*x))
        loss_history.append(sum((L[:] * L[:]))/len(x))

    return theta, theta_history, loss_history
    raise NotImplementedError()
```

In [27]:
```
t = init_t()
t_est_decay, ts_decay, loss_decay = grad_desc_decay(x, y, t, num_iter=20, alph
a=0.1)

assert len(ts_decay) == len(loss_decay) == 20 # theta history and loss history
are 20 items in them
assert ts_decay[0].shape == (2,) # theta history contains theta values
assert np.isscalar(loss[0]) # loss history should be a list of values, not vec
tor
assert loss_decay[1] - loss_decay[-1] > 0 # loss is decreasing
```

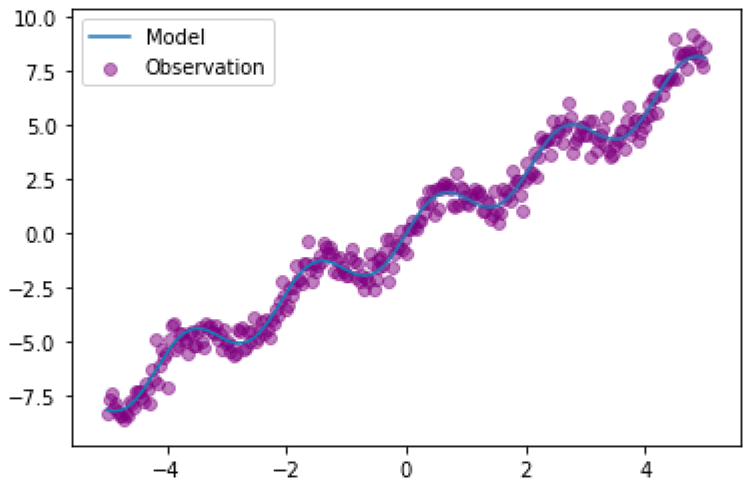
In [ ]:

Question 4c

Let's visually inspect our results of running gradient descent to optimize $\boldsymbol{\theta}$. Plot our x-values with our model's predicted y-values over the original scatter plot. Did gradient descent successfully optimize $\boldsymbol{\theta}$?

In [28]:
```
# Run me
t = init_t()
t_est, ts, loss = grad_desc(x, y, t)

t = init_t()
t_est_decay, ts_decay, loss_decay = grad_desc_decay(x, y, t)
```

In [29]:
```
y_pred = sin_model(x, t_est[0], t_est[1])

plt.plot(x, y_pred, label='Model')
plt.scatter(x, y, alpha=0.5, label='Observation', color='purple')
plt.legend();
```



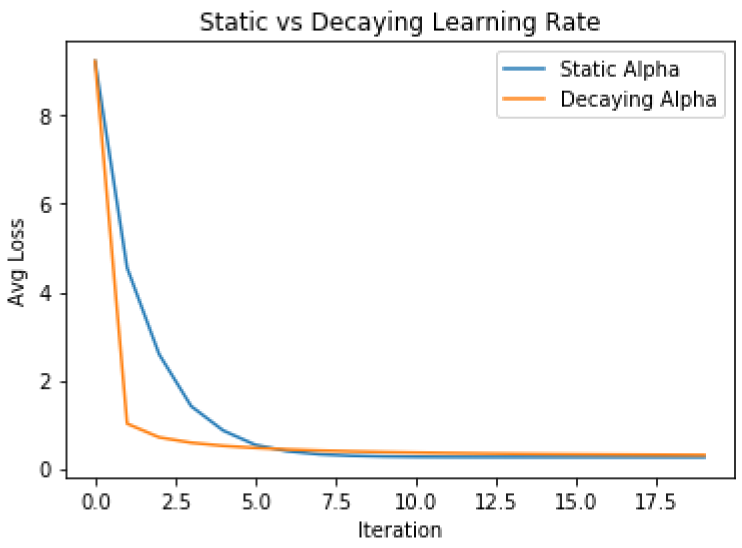
In [ ]:

Question 4d

Let's compare our two gradient descent methods and see how they differ. Plot the loss values over each iteration of gradient descent for both static learning rate and decaying learning rate.

In [30]:
```
plt.plot(np.arange(len(loss)), loss, label='Static Alpha')
plt.plot(np.arange(len(loss)), loss_decay, label='Decaying Alpha')
plt.xlabel('Iteration')
plt.ylabel('Avg Loss')
plt.title('Static vs Decaying Learning Rate')
plt.legend()
```

Out[30]:
```
<matplotlib.legend.Legend at 0x2b7dcd8af0f0>
```



Question 4e

Compare and contrast the performance of the two gradient descent methods. Which method begins to converge more quickly?

Decaying alpha allows method to converge more quickly.

5: Visualizing Loss

Question 5:

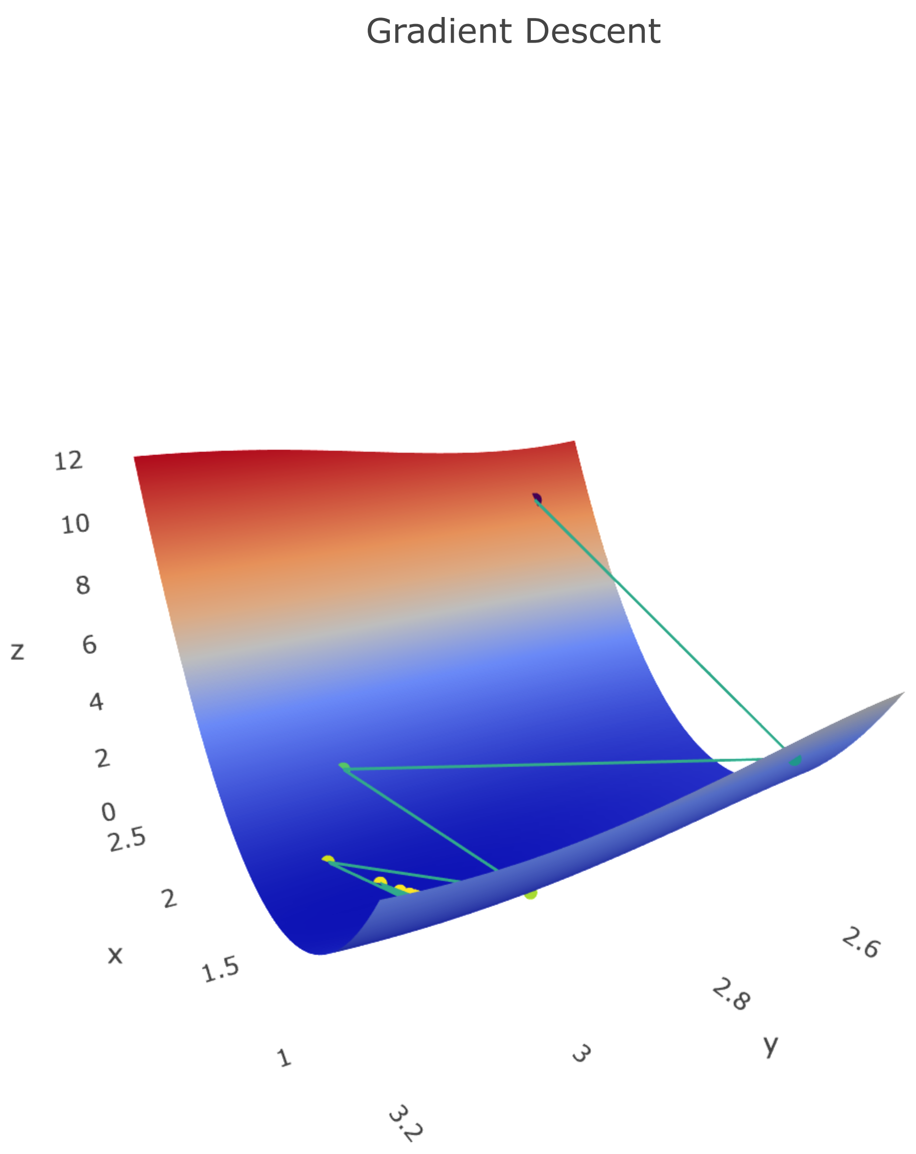
Let's visualize our loss functions and gain some insight as to how gradient descent and stochastic gradient descent are optimizing our model parameters.

Question 5a:

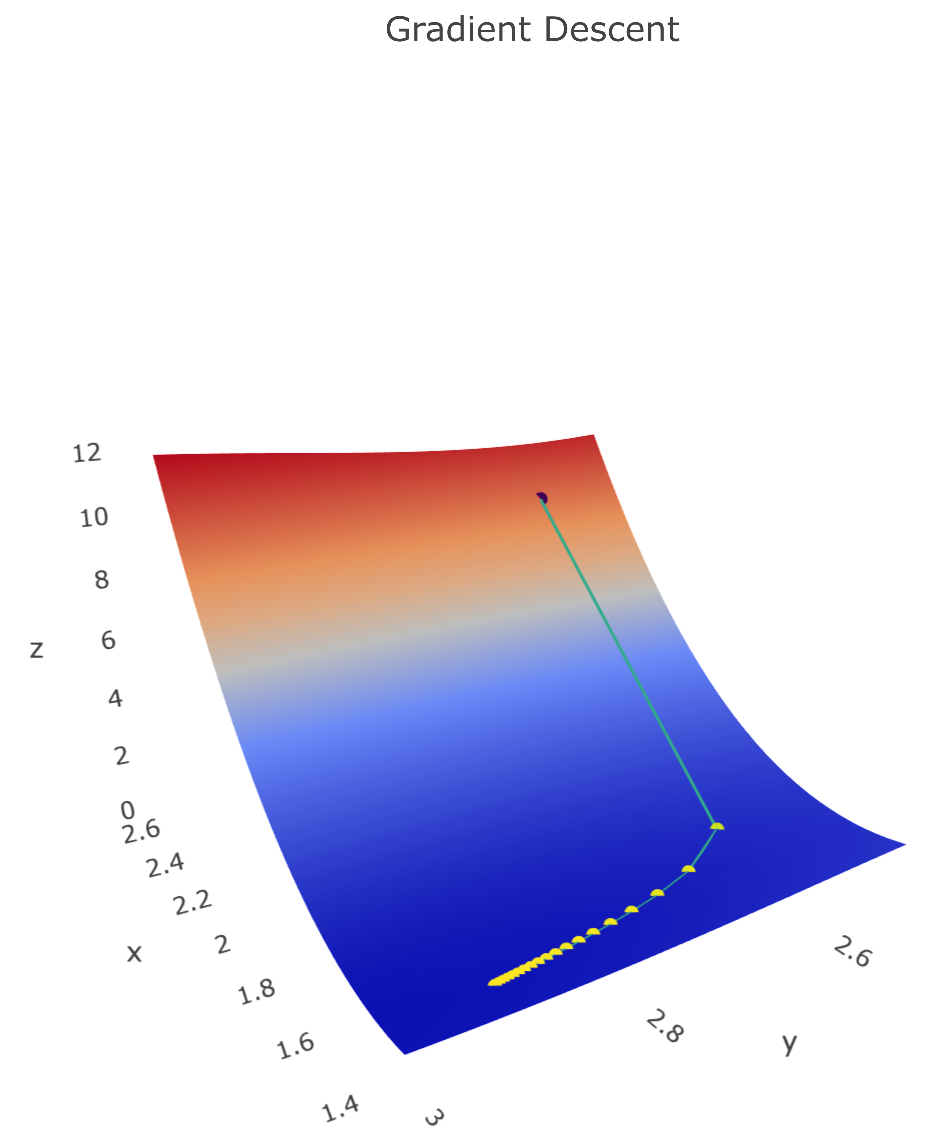
In the previous plot is about the loss decrease over time, but what exactly is path the theta value? Run the following three cells.

In [31]:
```
# Run me
ts = np.array(ts).squeeze()
ts_decay = np.array(ts_decay).squeeze()
loss = np.array(loss)
loss_decay = np.array(loss_decay)
```

In [32]:
```
# Run me to see a 3D plot (gradient descent with static alpha)
plot_3d(ts[:, 0], ts[:, 1], loss, l2_loss, sin_model, x, y)
```



In [33]:
```
# Run me to see another 3D plot (gradient descent with decaying alpha)
plot_3d(ts_decay[:, 0], ts_decay[:, 1], loss_decay, l2_loss, sin_model, x, y)
```



In the following cell, write 1-2 sentences about the differences between using a static learning rate and a learning rate with decay for gradient descent. Use the loss history plot as well as the two 3D visualization to support your answer.

The two gradient descent methods differs in ...

Again we see that using a static learning rate would lead to a slower convergence to the loss minimizer. We also see that the static learning rate oversteps and bounces around until it reaches the point to which it converges.

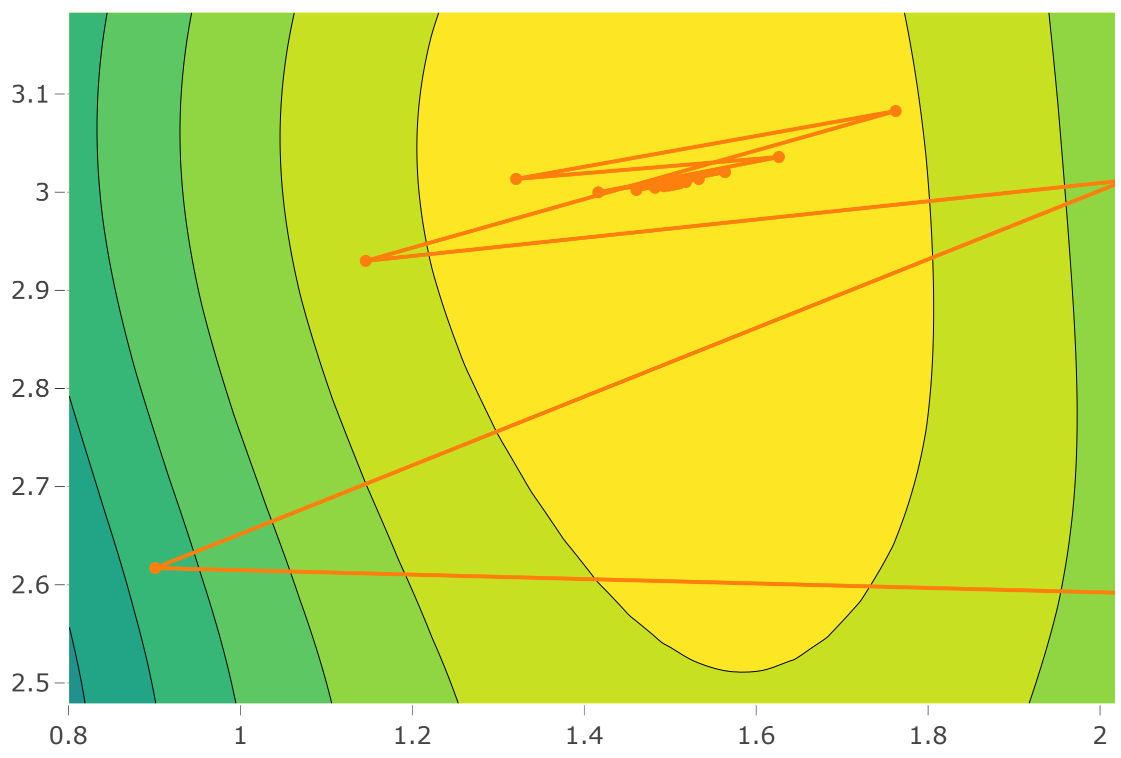
Question 5b:

Another common way of visualizing 3D dynamics is with a *contour* plot. Please run the following cell.

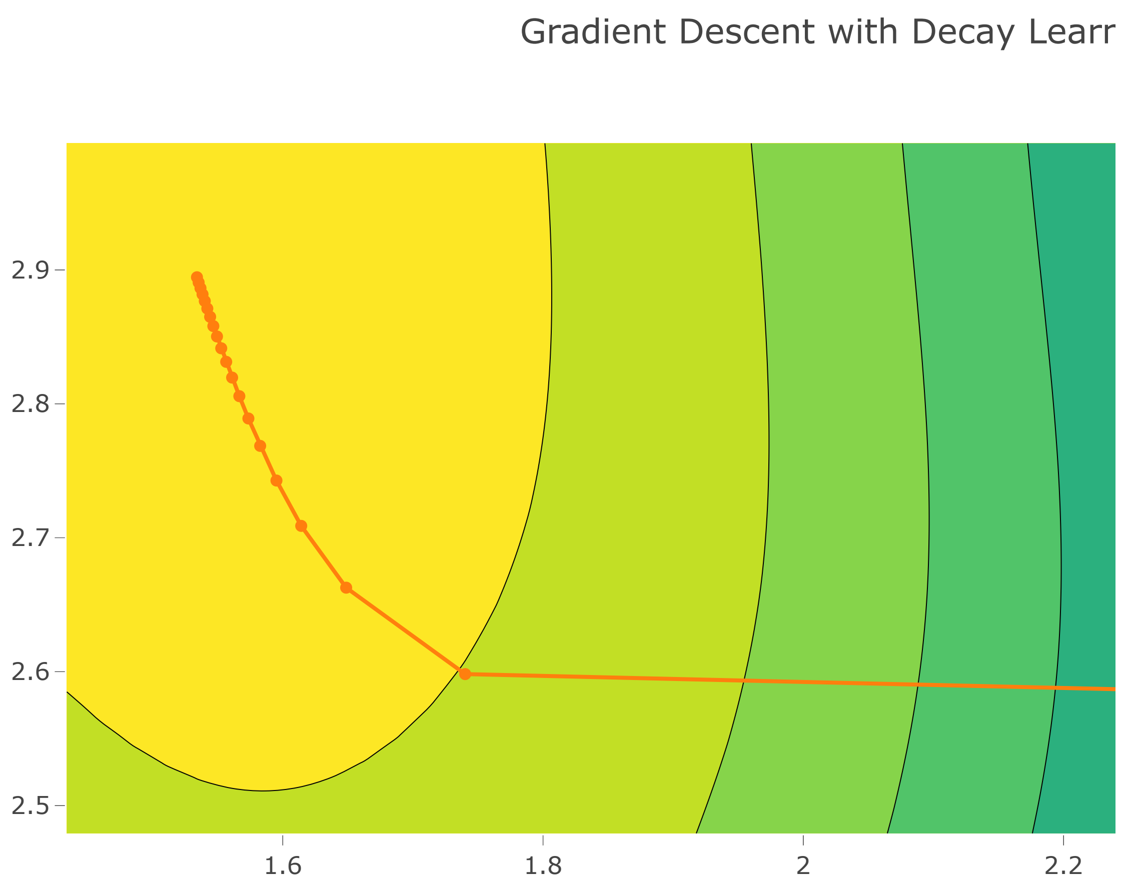
In [34]:
```
def contour_plot(title, theta_history, loss_function, model, x, y):
    """
    The function takes the following as argument:
        theta_history: a (N, 2) array of theta history
        loss: a list or array of loss value
        loss_function: for example, l2_loss
        model: for example, sin_model
        x: the original x input
        y: the original y output
    """
    theta_1_series = theta_history[:,0] # a list or array of theta_1 value
    theta_2_series = theta_history[:,1] # a list or array of theta_2 value

    # Create trace of theta point
    # Uncomment the following lines and fill in the TODOS
#     thata_points = go.Scatter(name="Theta Values",
#                               x=..., #TODO
#                               y=..., #TODO
#                               mode="lines+markers")

    ## In the following block of code, we generate the z value
    ## across a 2D grid
    t1_s = np.linspace(np.min(theta_1_series) - 0.1, np.max(theta_1_series) +
0.1)
    t2_s = np.linspace(np.min(theta_2_series) - 0.1, np.max(theta_2_series) +
0.1)

    x_s, y_s = np.meshgrid(t1_s, t2_s)
    data = np.stack([x_s.flatten(), y_s.flatten()]).T
    ls = []
    for t1, t2 in data:
        l = loss_function(model(x, t1, t2), y)
        ls.append(l)
    z = np.array(ls).reshape(50, 50)

    # Create the contour
    # Uncomment the following lines and fill in the TODOS
#     lr_loss_contours = go.Contour(x=..., #TODO
#                                   y=..., #TODO
#                                   z=..., #TODO
#                                   colorscale='Viridis', reversescale=True)


    thata_points = go.Scatter(name="Theta Values",
                             x=theta_1_series,
                             y=theta_2_series,
                             mode="lines+markers")
    lr_loss_contours = go.Contour(x=t1_s,
                                 y=t2_s,
                                 z=z,
                                 colorscale='Viridis', reversescale=True)

    plotly.offline.iplot(go.Figure(data=[lr_loss_contours, thata_points], layo
ut={'title': title}))
```

In [35]:
```
# Run this
contour_plot('Gradient Descent with Static Learning Rate', ts, l2_loss, sin_mo
del, x, y)
```

Gradient Descent with Static Learn



In [36]:
```
## Run me
contour_plot('Gradient Descent with Decay Learning Rate', ts_decay, l2_loss, s
in_model, x, y)
```



In the following cells, write down the answer to the following questions:

- How do you interpret the two contour plots?
- Compare contour plot and 3D plot, what are the pros and cons of each?

From reading these two contour plots, I can see that...

The contour plots show the exact same thing as the 3D plots. While,the 3D plots look cool, contour plots are easier to read and notice how much more slowly a static learning gradient descent would converge as well as it flickers between numerous points. With a decay learning rate, the gradient descent converges more quickly.

Question 5c

Try adding the two additional model parameters for phase and amplitude that we ignored (see 3a). What are the optimal phase and amplitude values for your four parameter model? Do you get a better loss?

I think...

We get a better loss with

optimal θ_1 = 1.5037

optimal θ_2 = 3.0074

optimal amp = 0.9584

optimal phase = -0.035

```
In [46]: def new_sin_model(x, theta_1, theta_2, amp, phase):
             """
             Predict the estimate of y given x, theta_1, theta_2

             Keyword arguments:
             x -- the vector of values x
             theta_1 -- the scalar value theta_1
             theta_2 -- the scalar value theta_2
             """
             y = theta_1*x +amp*np.sin(theta_2*x + phase)
             return y
             raise NotImplementedError()
```

```
In [47]: def dt1(x, y, theta):
             """
             Compute the numerical value of the partial of l2 loss with respect to thet
         a_1

             Keyword arguments:
             x -- the vector of all x values
             y -- the vector of all y values
             theta -- the vector of values theta
             """
             return np.mean(2*(theta[0]*x+theta[2]*np.sin(theta[1]*x+theta[3])-y)*x)
             raise NotImplementedError()
```

In [48]:
```
def dt2(x, y, theta):
    """
    Compute the numerical value of the partial of l2 loss with respect to thet
a_2

    Keyword arguments:
    x -- the vector of all x values
    y -- the vector of all y values
    theta -- the vector of values theta
    """
    return np.mean(2*(theta[0]*x+theta[2]*np.sin(theta[1]*x+theta[3])-y)*(x*np
.cos(theta[1]*x+theta[3])))
    raise NotImplementedError()
```

In [49]:
```
def dt3(x, y, theta):
    """
    Compute the numerical value of the partial of l2 loss with respect to thet
a_2

    Keyword arguments:
    x -- the vector of all x values
    y -- the vector of all y values
    theta -- the vector of values theta
    """
    return np.mean(2*(theta[0]*x+theta[2]*np.sin(theta[1]*x+theta[3])-y)*np.si
n(theta[1]*x+theta[3]))
    raise NotImplementedError()
```

In [50]:
```
def dt4(x, y, theta):
    """
    Compute the numerical value of the partial of l2 loss with respect to thet
a_2

    Keyword arguments:
    x -- the vector of all x values
    y -- the vector of all y values
    theta -- the vector of values theta
    """
    return np.mean(2*(theta[0]*x+theta[2]*np.sin(theta[1]*x+theta[3])-y)*theta
[2]*np.cos(theta[1]*x+theta[3]))
    raise NotImplementedError()
```

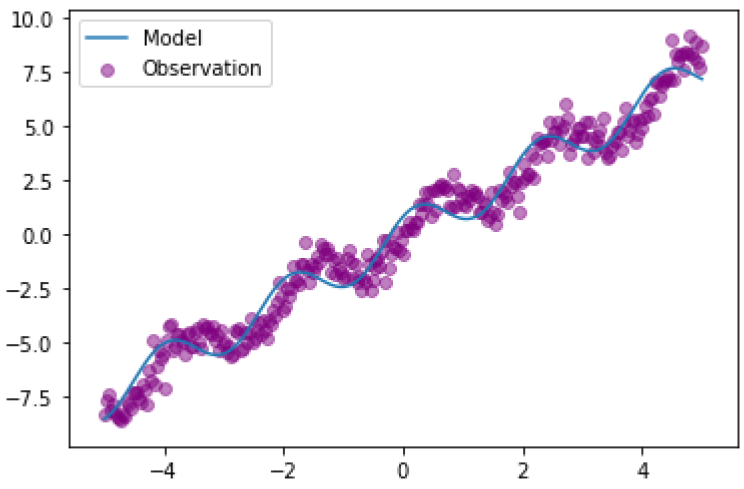
In [51]:
```
# This function calls dt1 and dt2 and returns the gradient dt. It is already i
mplemented for you.
def dt(x, y, theta):
    """
    Returns the gradient of l2 loss with respect to vector theta

    Keyword arguments:
    x -- the vector of values x
    y -- the vector of values y
    theta -- the vector of values theta
    """
    return np.array([dt1(x,y,theta), dt2(x,y,theta), dt3(x,y,theta), dt4(x,y,t
heta)])
```

In [52]:
```
def grad_desc(x, y, theta, num_iter=20, alpha=0.1):
    """
    Run gradient descent update for a finite number of iterations and static l
earning rate

    Keyword arguments:
    x -- the vector of values x
    y -- the vector of values y
    theta -- the vector of values theta to use at first iteration
    num_iter -- the max number of iterations
    alpha -- the learning rate (also called the step size)

    Return:
    theta -- the optimal value of theta after num_iter of gradient descent
    theta_history -- the series of theta values over each iteration of gradien
t descent
    loss_history -- the series of loss values over each iteration of gradient
 descent
    """
    theta_history = []
    loss_history = []

    for i in range(num_iter):
        change = dt(x,y,theta)
        theta = theta - (change*alpha)
        theta_history.append(theta)
        L = y - (theta[0]*x + theta[2]*np.sin(theta[1]*x + theta[3]))
        loss_history.append((sum(L[:] * L[:]))/len(x))

    return theta, theta_history, loss_history
    raise NotImplementedError()
```

In [55]:
```
t = [0, 0, 1, 0]
t_est, ts, loss = grad_desc(x, y, t, num_iter=20, alpha=0.1)
t_est
```

Out[55]:
```
array([ 1.5037,  3.0074,  0.9584, -0.035 ])
```

In [54]:
```
y_pred = new_sin_model(x, t_est[0], t_est[1], 1, 1)

plt.plot(x, y_pred, label='Model')
plt.scatter(x, y, alpha=0.5, label='Observation', color='purple')
plt.legend();
```



Question 5d (optional)

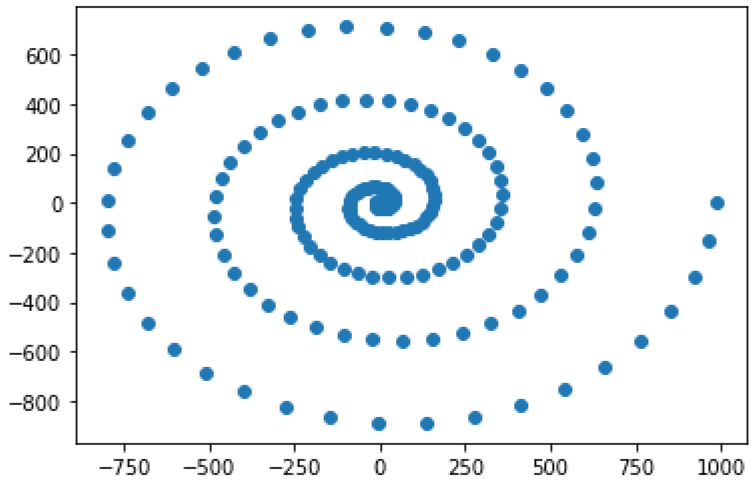
It looks like our basic two parameter model, a combination of a linear function and sinusoidal function, was able to almost perfectly fit our data. It turns out that many real world scenarios come from relatively simple models.

At the same time, the real world can be incredibly complex and a simple model wouldn't work so well. Consider the example below; it is neither linear, nor sinusoidal, nor quadratic.

Optional: Suggest how we could iteratively create a model to fit this data and how we might improve our results.

Extra optional: Try and build a model that fits this data.

In [49]:
```
x = []
y = []
for t in np.linspace(0,10*np.pi, 200):
    r = ((t)**2)
    x.append(r*np.cos(t))
    y.append(r*np.sin(t))

plt.scatter(x,y)
plt.show()
```



YOUR ANSWER HERE