Problem_1

January 20, 2023

1 Imports

[ ]:

```
import numpy as np
import matplotlib.pyplot as plt
import os
from GD import BatchGradientDescent, MiniBatchGradientDescent, SteepestDescent

plt.rcParams['figure.figsize'] = (10.0, 7.0)
plt.rcParams["font.size"] = 16
plt.rcParams["font.family"] = "Serif"
plt.rcParams["grid.linestyle"] = "--"
```

[ ]:

```
DATA_DIR = 'data'
SAVE_DIR = "plots"
```

2 Loading Data

[ ]:

```
data = np.loadtxt(os.path.join(DATA_DIR, "prob1data.txt"), delimiter=",")
data = data.T
data.shape
```

[ ]:

```
(70, 2)
```

[ ]:

```
t = data[:, 0]
y = data[:, 1]
```

3 The Problems

The problems we need to solve are:

1. Plot the training data. Write a code in Python to perform nonlinear regression on the given data. Implement batch gradient descent algorithm for optimization. (Choose $\alpha = 0.01$, number of iterations $= 50000$)
2. Implement stochastic gradient descent for optimization of weights. Plot cost history (J) vs number of iterations for both cases batch gradient descent and stochastic gradient descent. Comment on the difference, if any.

3. Plot the cost history (J) vs number of iterations for different learning rates ($\alpha = 0.1, 0.5, 0.01, 0.05$). Write your inferences from the plot.
4. Implement line search method (Secant method) to find learning rate (α). Optimize the weights using batch gradient descent and plot the cost history (J) vs number of iterations for variable learning rate. Comment on the difference between implementing line search method and choosing arbitrary α.

3.1 The Approach

First, we'll define the original problem statement in a way which is easier to solve in Python. We'll start by the given data.

3.1.1 The Hypothesis Function

The data is not linear, for obvious region. The general equation of a particle moving under a constant gravity is:

$$y = y_0 + v_0 t - \frac{1}{2} g t^2$$

Where θ is the projectile angle, y_0 is initial position and v_0 is the initial velocity and g is the acceleration due to gravity.

Note that the problem can be rewritten as:

$$y = w_0 + w_1 t + w_2 t^2$$

So, this is just a linear regression with variables $[1, t, t^2]$. This is the approach we'll take to solve the problem. This means that our hypothesis function is:

$$h(w_i) = w_0 + w_1 t + w_2 t^2$$

3.1.2 Notations

Some notations which we will follow throught the assignment (and future assignments too) are:

1. A scalar is denoted by regular lowercase letter, eg. x, y, t, w.
2. A vector, which is denoted by boldface lowercase letter, eg. $\mathbf{x}, \mathbf{y}, \mathbf{w}$.
3. A vector, if not mentioned otherwise, will be a column vector, like

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}$$

4. A matrix is denoted by a regular uppercase letter, eg. X, W.
5. Number of training examples is denoted by m.
6. Number of features is denoted by n.
7. $X \in \mathbb{R}^{m \times n}$ is the input matrix.
8. $X^{(i)} \in \mathbb{R}^n$ is the i^{th} example in the input matrix.
9. $\mathbf{w} \in \mathbb{R}^n$ denotes the coeficient vector and b is the intercept.

10. $\hat{y}$ is the predicted value.
11. The hypothesis function is denoted by $h(\theta)$ where $\theta \in \mathbb{R}^{n+1}$ includes $\mathbf{w}$ and b.
12. The loss/cost function for the whole data is denoted by $J(X, \mathbf{y}, \mathbf{w}, b)$ or $J(\hat{y}, \mathbf{y})$.
13. ∂J_{w_i} denotes the partial derivative of J with respect to w_i, namely, $\frac{\partial J}{\partial w_i}$. Same goes for b.

3.1.3 Reformulation of the Problem

Using the notations above, we can reformulate the problem. The hypothesis function is:

$$h(\mathbf{w}, b) = h(\theta) = b + w_1 x + w_2 x^2 = \mathbf{w}^T \mathbf{x} + b$$

with $\mathbf{w} = [w_1, w_2]$ and $\mathbf{x} = [x, x^2]$.

For the loss function, we'll use the mean squared error loss function:

$$J(\hat{y}, \mathbf{y}) = \frac{1}{2m} \sum_{i=1}^{m} (\hat{y}^{(i)} - y^{(i)})^2$$

We are using a factor of 2 instead of 1 to make the derivatives simpler.

The gradient descent algorithm is:

$$\mathbf{w} := \mathbf{w} - \alpha \frac{\partial J}{\partial \mathbf{w}} b := b - \alpha \frac{\partial J}{\partial b}$$

Where α is the learning rate.

The partial derivatives can easily be calculated as:

$$\frac{\partial J}{\partial \mathbf{w}} = \frac{1}{m} \sum_{i=1}^{m} (\hat{y}^{(i)} - y^{(i)}) \mathbf{x}^{(i)} \frac{\partial J}{\partial b} = \frac{1}{m} \sum_{i=1}^{m} (\hat{y}^{(i)} - y^{(i)})$$

3.1.4 Including b in $\mathbf{w}$

Note that we can include the bias term b in $\mathbf{w}$, by adding a column of 1's to X. This will make the code simpler. We'll do this in the code. Using this, the X matrix will be:

$$X = \begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ \vdots & \vdots & \vdots \\ 1 & x_m & x_m^2 \end{bmatrix}$$

While, the gradient descent algorithm will become:

$$\mathbf{w} := \mathbf{w} - \alpha \frac{\partial J}{\partial \mathbf{w}}$$

with $\mathbf{w} \in \mathbb{R}^3$, $\mathbf{w} = [b, w_1, w_2]^T = [w_0, w_1, w_2]^T$.

Much simpler, isn't it?

4 A General Gradient Descent Algorithm

Instead of writing a gradient descent of the problem defined above, we'll implement a generalized gradient descent algorithm. This will be useful for future assignments as well as other problems in this assignment. The above problem will then just be a special case.

4.1 Formalism of the Algorithm

Suppose we have the input vector $X \in \mathbb{R}^{n \times m}$ with n number of features and m number of examples. The hypothesis function will be:

$$h(\mathbf{w}) = \mathbf{w}^T X$$

where $\mathbf{w} \in \mathbb{R}^n$ or $\mathbf{w} \in \mathbb{R}^{n+1}$ (if bias term is included) is a column vector.

We can see that the hypothesis function $h(\mathbf{w}) \in \mathbb{R}^m$.

The cost function is:

$$\begin{aligned} J(\hat{y}, \mathbf{y}) &= \frac{1}{2m} \sum_{i=1}^{m} (\hat{y}^{(i)} - y^{(i)})^2 \\ &= \frac{1}{2m} \sum_{i=1}^{m} (h(\mathbf{w})^{(i)} - y^{(i)})^2 \end{aligned}$$

The gradient given by:

$$\frac{\partial J}{\partial \mathbf{w}} = \frac{1}{m} \sum_{i=1}^{m} (\hat{y}^{(i)} - y^{(i)}) \mathbf{x}^{(i)}$$

The gradient descent update rule is:

$$\begin{aligned} \mathbf{w} &:= \mathbf{w} - \alpha \frac{\partial J}{\partial \mathbf{w}} \\ &:= \mathbf{w} - \alpha \frac{1}{m} \sum_{i=1}^{m} (\hat{y}^{(i)} - y^{(i)}) \mathbf{x}^{(i)} \end{aligned}$$

Note that we have included b in $\mathbf{w}$.

4.2 Implementing Gradient Descent

The implementation of gradient descent can be found in the module `GD.py`. The code is structured as follow:

> To increase the reusability of code, I've used inheritance, where one class inherits some of the properties from the parent class. This is done because there are a lot of similarity among linear gradient descent, batch gradient descent, stochastic gradient descent as well as their logistic regression counterparts. Using inheritence means I will be able to save hundreds of lines of code and make the code more readable as well as more maintainable.

4.2.1 `BatchGradientDescent` class

This is the base class for all gradient descent algorithms. Any gradient descent algorithm, inheriting from this base class get some attributes and methods for free.

The class is instantiated with the following parameters:

- `fit_intercept`: If `True`, the bias term b is included in $\mathbf{w}$.
- `tol`: The tolerance for the stopping criterion. The algorithm stops when the change in the loss function is less than `tol`.

By default, `fit_intercept` is set to `True` and `tol` is set to `None`, which means that the stopping criterion is not used.

Apart from just being the base class for gradient descent, the class can be used for a multilinear regression. The class has a `fit` method which takes the input matrix X, the target variable y along with `learning_rate`, `epochs` and some other parameters. The `fit` method fits the model. The learned parameters are stored in the `self.weights` attribute. The `predict` method takes the input matrix X and returns the predicted values. These two are the main methods of the class.

Using `callback` The `fit` method accepts a `callable` which takes in input parameters, `self`, which is the model in question, `weights`, which is the weight of the model at the current iteration and `epoch`, which is the current epoch. This is useful for plotting the loss function as well as the weights as the model is being trained.

4.2.2 `MiniBatchGradientDescent` class

This class inherits from the `BatchGradientDescent` class. The `fit` method is the same as the `BatchGradientDescent` class. The `predict` method is also the same. The only difference is that the `fit` method uses the batch gradient descent algorithm. and hence accepts a parameter `batch_size` which controls what batch size to use.

4.2.3 Using Stochastic Gradient Descent

Stochastic gradient descent is nothing but a batch gradient descent with batch size of 1. So, SGD can be implemented by setting the `batch_size` to 1 for the `MiniBatchGradientDescent` class. I've not written a separate class for SGD.

5 Solving the First Problem

5.1 Problem 1.1

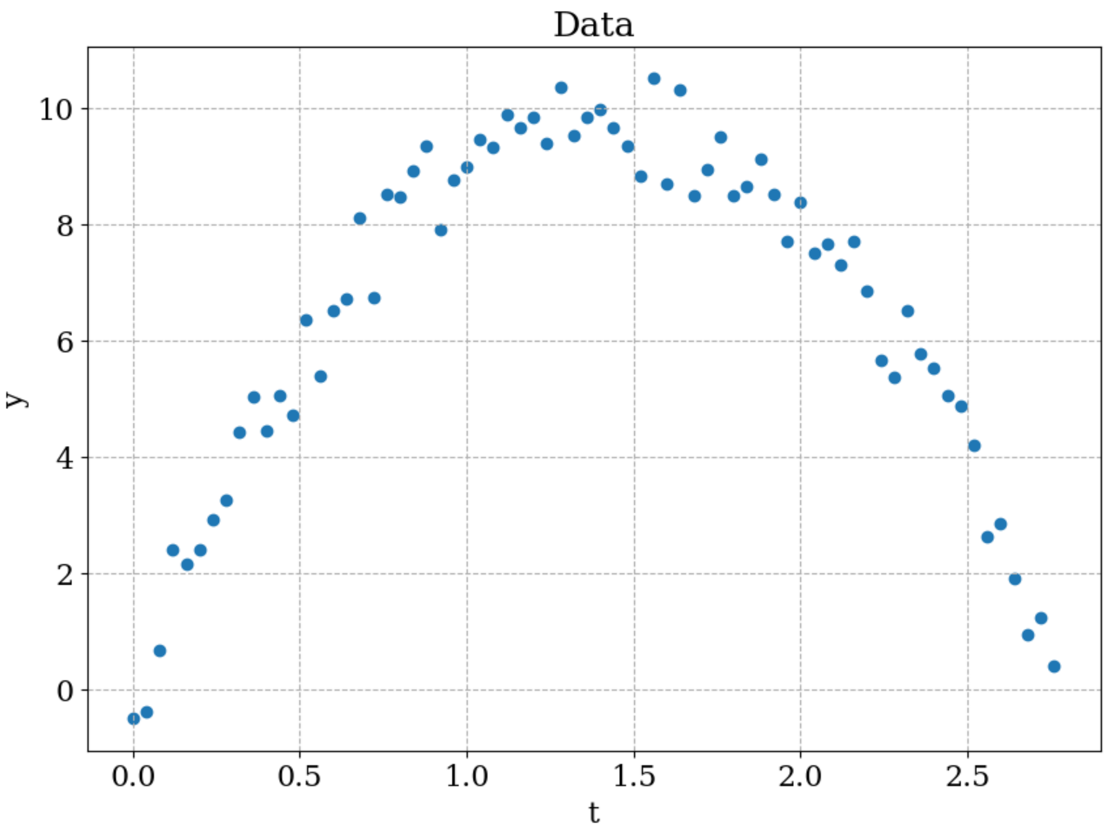
The first problem is : Plot the training data. Write a code in Python to perform nonlinear regression on the given data. Implement batch gradient descent algorithm for optimization. (Choose $\alpha = 0.01$, number of iterations = 50000)

5.1.1 Plotting y with t

[ ]:

```
fig, ax = plt.subplots()

ax.scatter(t, y)
ax.set_xlabel("t")
ax.set_ylabel("y")
ax.grid()
ax.set_title("Data")
plt.savefig(os.path.join(SAVE_DIR, "0101.png"));
```



The graph is a parabola as it should be. Since the free fall is a parabola. See section 3.1.1.

5.1.2 Batch Gradient Descent

We need to add the second order term. This can be done easily. We'll also rename these variables.

[ ]:

```
t = t.reshape(-1, 1)
X = np.concatenate((t, t**2), axis=1)
X.shape
```

[ ]:

```
(70, 2)
```

[ ]:

```
bgd = BatchGradientDescent(fit_intercept=True, tol=None)


bgd.fit(X, y, learning_rate=0.01, epochs=50000, verbose=-1)
bgd_losses = bgd._losses
```

Let's see the coefficients learned by the model.

[ ]:

```
bgd_params = bgd.weights
print(bgd_params)
```

```
[-0.13575302 14.20039596 -5.04170007]
```

Let's calculate the R^2 score.

[ ]:

```
r2_bgd = bgd.score(X, y)
print(f"R2 score for BGD: {r2_bgd}")
```

```
R2 score for BGD: 0.9684610571559654
```

5.2 Problem 1.2

5.2.1 Stochastic Gradient Descent

Stochastic gradient descent is nothing but mini batch gradient descent with batch size of 1. So, we'll use the `MiniBatchGradientDescent` class with `batch_size` set to 1.

[ ]:

```
sgd = MiniBatchGradientDescent(fit_intercept=True, tol=None)


sgd.fit(X, y, learning_rate=0.01, epochs=50000, batch_size=1, verbose=0)
sgd_losses = sgd._losses
```

```
50000/50000 [=================== ] 100.0%
```

[ ]:

```
sgd_params = sgd.weights
print(sgd_params)
```

```
[-0.12448906 14.12396319 -5.03803759]
```

let's see the R^2 score.

[ ]:

```
r2_sgd = sgd.score(X, y)
print(f"R2 score for SGD: {r2_sgd}")
```

```
R2 score for SGD: 0.9673424422990577
```

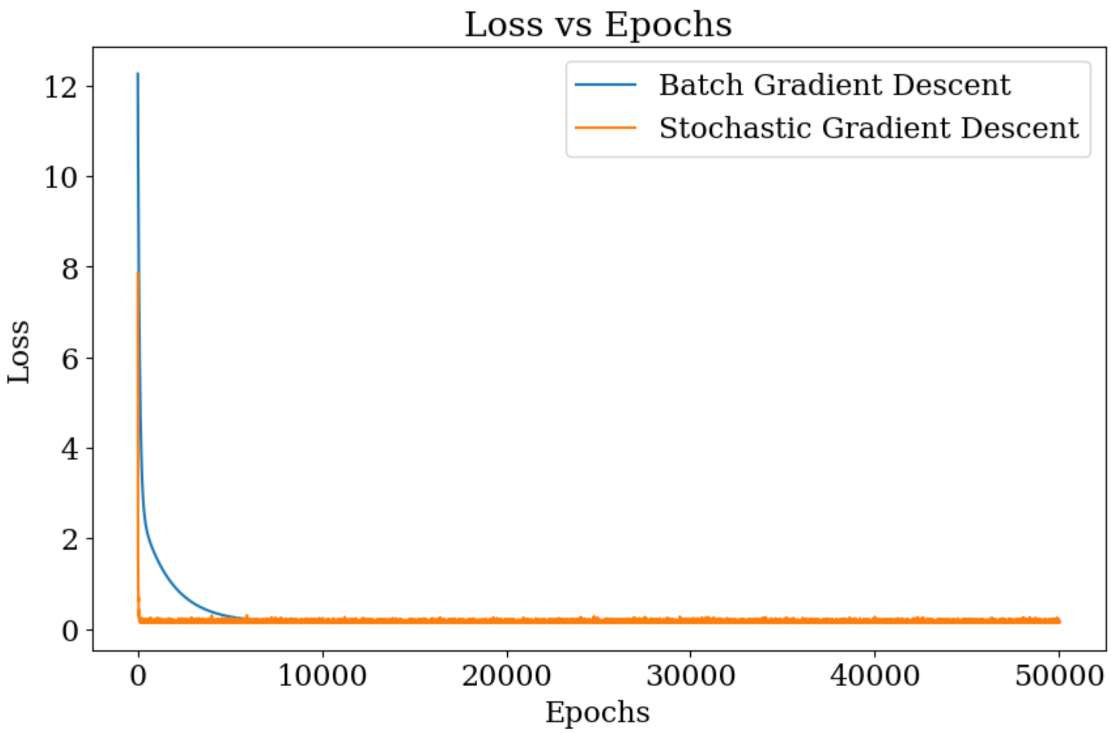
5.2.2 Plotting the Losses

[ ]:

```
fig, ax = plt.subplots(figsize = (10, 6))

epochs = np.arange(1, 50001)
ax.plot(epochs, bgd_losses, label="Batch Gradient Descent")
ax.plot(epochs, sgd_losses, label="Stochastic Gradient Descent")

ax.set_xlabel("Epochs")
ax.set_ylabel("Loss")
ax.set_title("Loss vs Epochs")
ax.legend()
plt.savefig(os.path.join(SAVE_DIR, "0102.png"));
```



5.2.3 Some Differences Between SGD and BGD

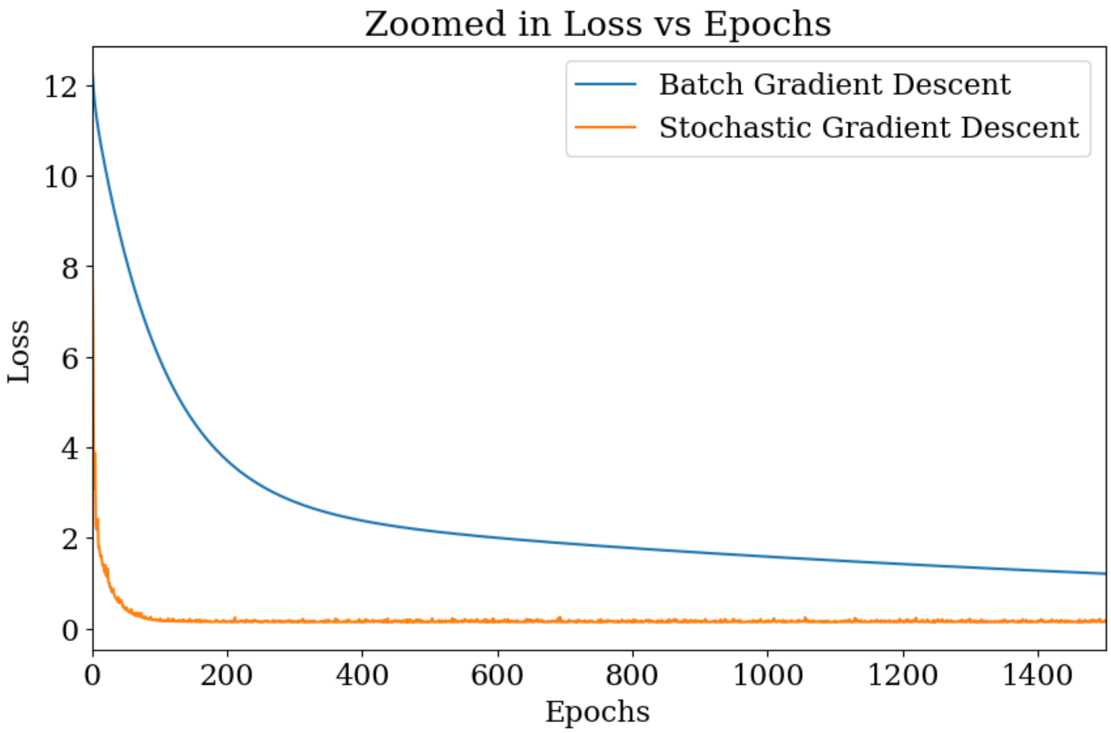
1. For BGD, the loss is decreasing only up to 2000 epoch. After that, change in loss is almost negligible.
2. For SGD this starts happening even before 200 epoch!
3. SGD starts with much lower loss that the BGD.

Let's zoom to the initial epochs.

[ ]:

```
ax.set_xlim(0, 1500)
ax.set_title("Zoomed in Loss vs Epochs")
fig.savefig(os.path.join(SAVE_DIR, "0103.png"))
fig
```

[ ]:



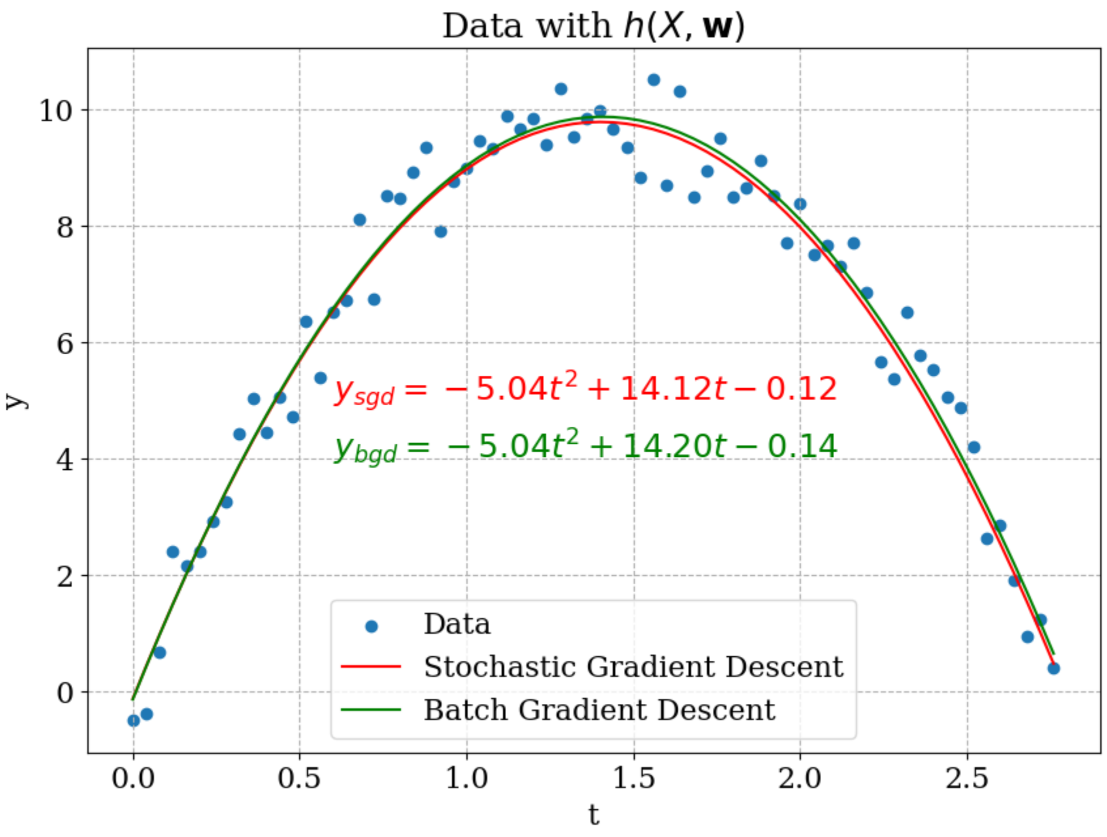
We can see that there is a little fluctuation is the loss by the SGD, as it should.

Next, we'll plot the function learned by the models on the original data.

[ ]:

```
y_pred_sgd = sgd.predict(X)
y_pred_bgd = bgd.predict(X)

fig, ax = plt.subplots()
ax.scatter(t, y, label="Data")
ax.plot(t, y_pred_sgd, label="Stochastic Gradient Descent", color="red")
ax.plot(t, y_pred_bgd, label="Batch Gradient Descent", color="green")
#annotate the function learned
ax.text(0.60, 5, fr"$y_{{sgd}} = {sgd_params[2]:.2f}t^2 + {sgd_params[1]:.2f}t {sgd_params[0]:.2f}$", fontsize=18, color="red")
ax.text(0.60, 4, fr"$y_{{bgd}} = {bgd_params[2]:.2f}t^2 + {bgd_params[1]:.2f}t {bgd_params[0]:.2f}$", fontsize=18, color="green")
ax.set_xlabel("t")
ax.set_ylabel("y")
ax.grid()
ax.legend()
ax.set_title("Data with $h(X, \mathbf{w})$")
plt.savefig(os.path.join(SAVE_DIR, "0104.png"));
```



5.3 Problem 1.3

5.3.1 J with α

We'll use the `BatchGradientDescent` for this as this is the most fast to train. Also, since we saw that the model is mostly converged to about 1000 epochs, we'll use just 1000 epochs for the training.

```
[ ]: def one_alpha(alpha):
         gd = BatchGradientDescent(fit_intercept=True, tol=None)
         gd.fit(X, y, learning_rate=alpha, epochs=1000, verbose=-1)
         gd_losses = gd._losses
         return gd_losses
```

```
[ ]: alphas = [0.1, 0.5, 0.01, 0.05]
     loss_dict = {}
     for alpha in alphas:
```

```
    print(f"alpha = {alpha}", end="\r")
    losses = one_alpha(alpha)
    loss_dict[alpha] = losses
```

```
alpha = 0.05
```

```
/home/hari31416/anaconda3/envs/data-science/lib/python3.9/site-
packages/numpy/core/fromnumeric.py:86: RuntimeWarning: overflow encountered in
reduce
  return ufunc.reduce(obj, axis, dtype, out, **passkwargs)
/media/hari31416/Hari_SSD/Users/harik/Desktop/APL745/Assignments/Assignment_2/GD
.py:52: RuntimeWarning: overflow encountered in square
  return np.sum((y_hat - y_true) ** 2) / (2 * m)
/media/hari31416/Hari_SSD/Users/harik/Desktop/APL745/Assignments/Assignment_2/GD
.py:39: RuntimeWarning: overflow encountered in matmul
  res = np.matmul((y_hat - y_true), X)
```

[ ]:

```
for alpha in alphas:
    print(alpha, max(loss_dict[alpha]))
```

```
0.1 16.684610305415177
0.5 inf
0.01 14.32390952269674
0.05 17.942936565766516
```

Okay, it seems we are getting overflow for some values of α. This is happening because the value of α is too high and the gradient descent is diverging.

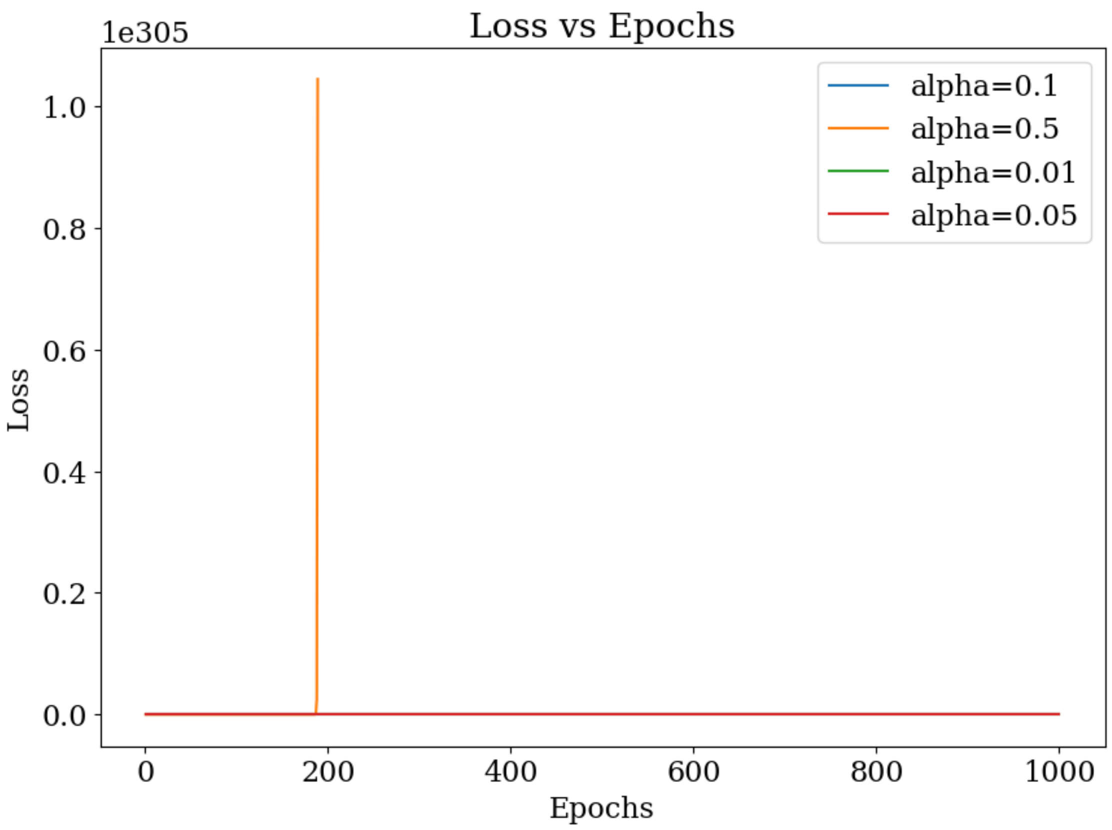
So, this is happening for $\alpha = 0.5$. Let's plot the losses. However, we need to exclude the values for $\alpha = 0.5$ otherwise we won't be able to make any inference. See the figure below for example.

[ ]:

```
fig, ax = plt.subplots()

epochs = np.arange(1, 1001)
for alpha in alphas:
    ax.plot(epochs, list(loss_dict[alpha]), label=f"alpha={alpha}")

ax.set_xlabel("Epochs")
ax.set_ylabel("Loss")
ax.set_title("Loss vs Epochs")
ax.legend()
fig.savefig(os.path.join(SAVE_DIR, "0105.png"));
```



```
[ ]: loss_dict[0.5][:15]
```

```
[ ]: [17.852495334643752,
      251.73813011404698,
      10132.623560900614,
      420571.7811636781,
      17465747.223886114,
      725334546.3748773,
      30122404456.437798,
      1250952761392.6108,
      51950793424495.63,
      2157463511599822.2,
      8.959726112075296e+16,
      3.720882952216258e+18,
      1.5452447732117933e+20,
      6.41724407836086e+21,
      2.6650160722215383e+23]
```

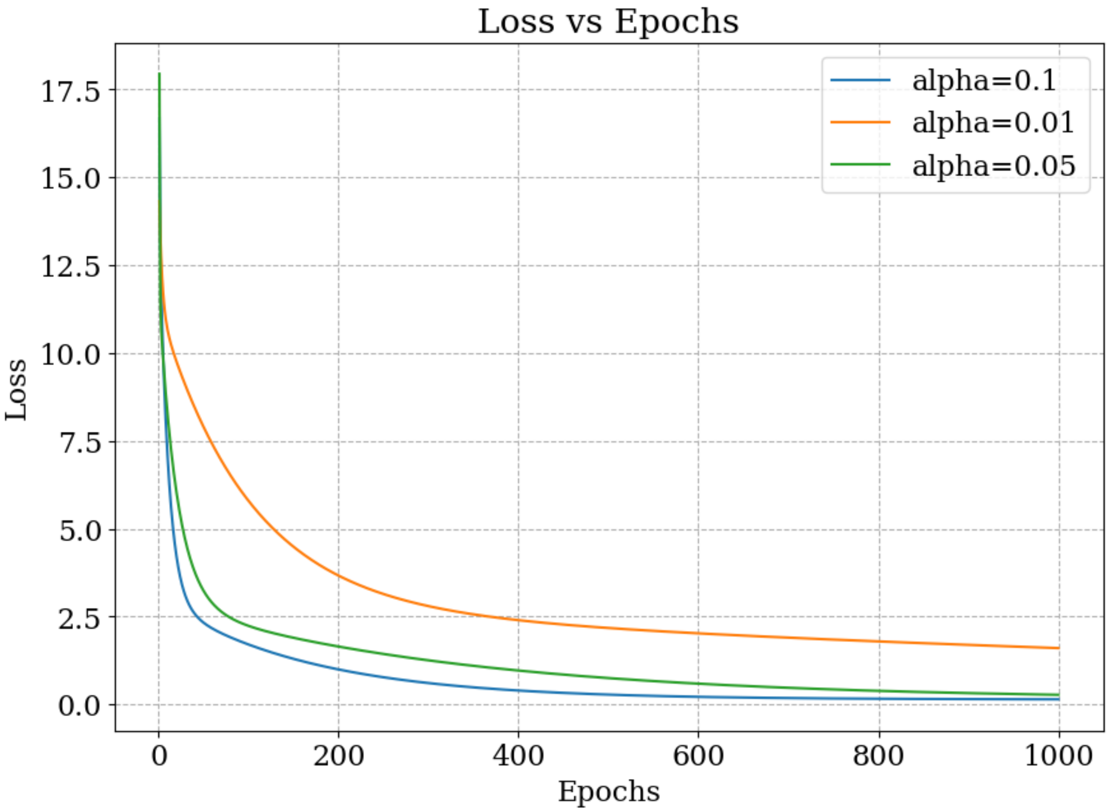
Here are the top 15 values corresponding to the loss by $\alpha = 0.5$. We can see that the loss is increasing exponentially and becoming incomprehesibly large in just a few tens of iterations.

[ ]:

```
fig, ax = plt.subplots()

for alpha in alphas:
    if alpha == 0.5:
        continue
    ax.plot(epochs, list(loss_dict[alpha]), label=f"alpha={alpha}")

ax.set_xlabel("Epochs")
ax.set_ylabel("Loss")
ax.set_title("Loss vs Epochs")
ax.grid()
ax.legend()
fig.savefig(os.path.join(SAVE_DIR, "0106.png"));
```



5.4 Problem 1.4

5.4.1 The Secant Method

For this, we need to implement the line search algorithm for finding α. The code for this is in the class `SteepestDescent` in the same `GD.py` file. What I've done is this:

Created a function `phi` which represents the function

$$\phi(\alpha) = J(\mathbf{w} - \alpha \nabla J)$$

discussed in the lecture.

In the assignment, we are supposed to find the α which minimizes $\phi(\alpha)$. This has to be done by the secant method. For this, we need the derivative of $\phi(\alpha)$, as at local minima, the gradient is zero. The update equation is:

$$\alpha_{k+1} = \alpha_k - \phi'(\alpha_k)\frac{\alpha_k - \alpha_{k-1}}{\phi'(\alpha_k) - \phi'(\alpha_{k-1})}$$

For this, I've first defined a function which calculates the derivative of $\phi(\alpha)$ using the central difference method. Then, I've defined a function which implements the secant method. The secant method is implemented in the `secant` method of the `SteepestDescent` class.

Finally, I've written a final wrapper function `_get_lr` which calculates α using the `secant` function and does some preprocessing on the learning rate. See the code for more details.

[ ]:

```
np.random.seed(42)
alphas = []

def callback(model, weights, epoch, lr):
    alphas.append(lr)
sd = SteepestDescent(fit_intercept=True, tol=1e-6)

sd.fit(X,y, epochs = 1000, verbose=0, callback=callback)
sd_losses = sd._losses
```

```
Converged at epoch 826    ] 82.6%
Loss: 0.14247026521113368
```

[ ]:

```
sd_params = sd.weights
print(sd_params)
```

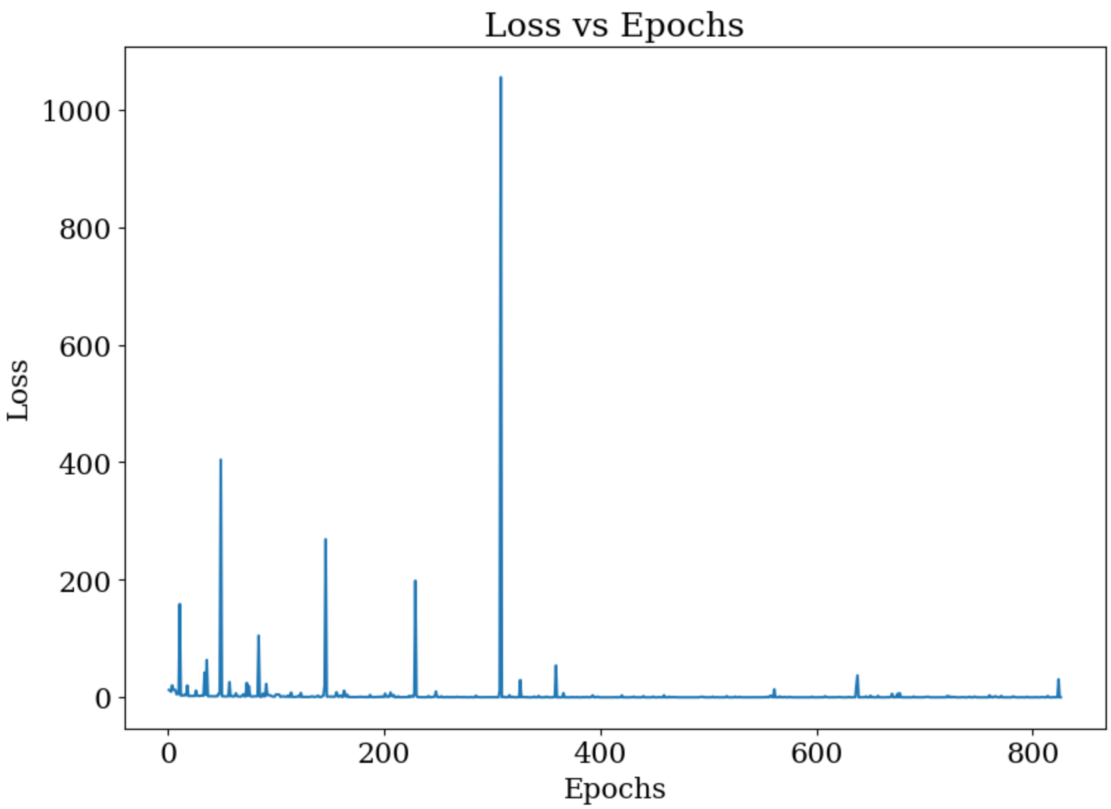
```
[-0.06189022 14.06248588 -4.99873017]
```

Let's plot the loss.

[ ]:

```
fig, ax = plt.subplots()

epochs = np.arange(1, len(sd_losses)+1)
ax.plot(epochs, sd_losses)
ax.set_xlabel("Epochs")
ax.set_ylabel("Loss")
ax.set_title("Loss vs Epochs")
fig.savefig(os.path.join(SAVE_DIR, "0107.png"));
```



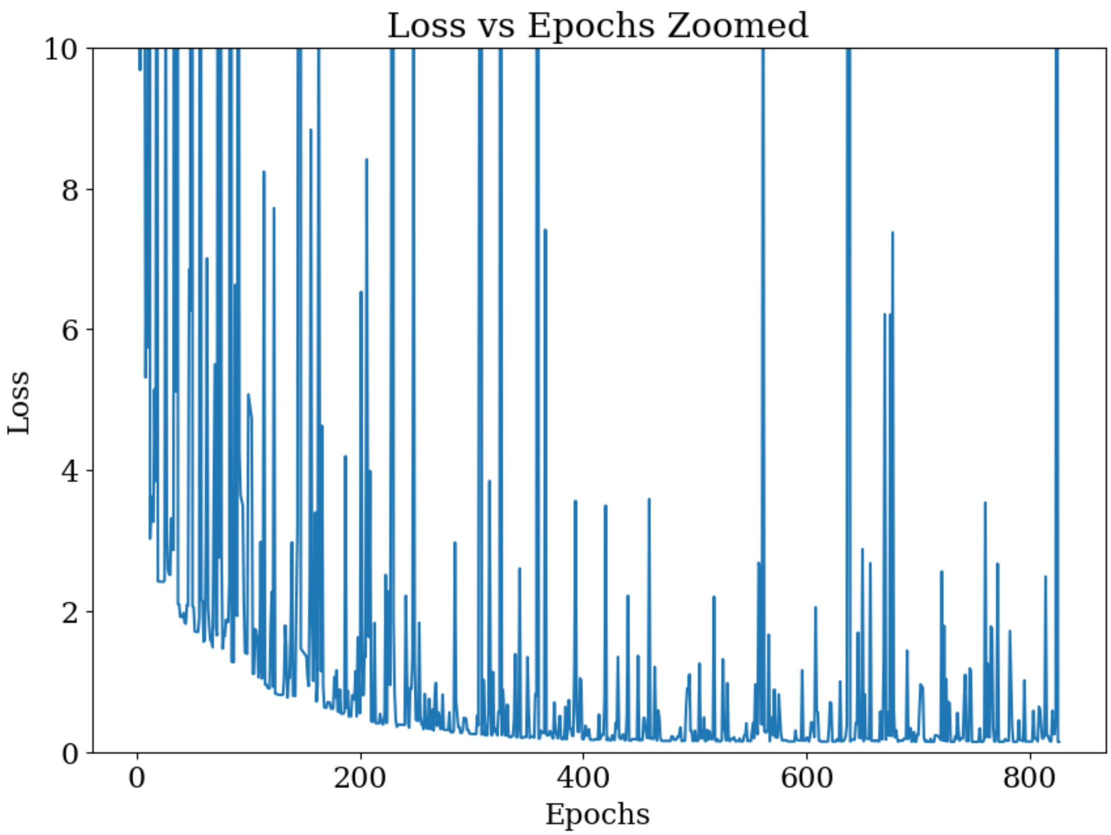
We can see that there are some huge values of loss for some epochs. This is because the learning rate is too high for that epoch and the gradient descent is diverging.

Let's set a limit on y-axis to see the loss better.

[ ]:

```
fig, ax = plt.subplots()

epochs = np.arange(1, len(sd_losses)+1)
ax.plot(epochs, sd_losses)
ax.set_xlabel("Epochs")
ax.set_ylabel("Loss")
ax.set_title("Loss vs Epochs Zoomed")
ax.set_ylim(0, 10)
fig.savefig(os.path.join(SAVE_DIR, "0108.png"));
```

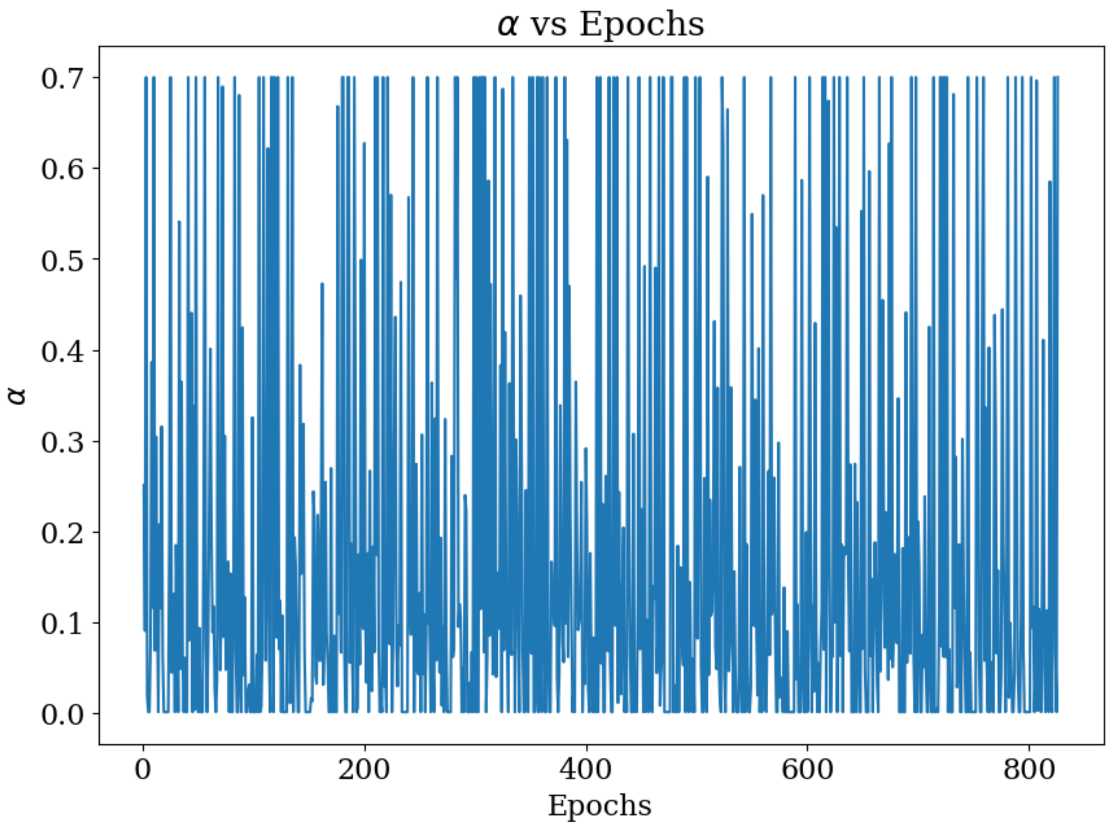


Let’s plot the learning rates.

[ ]:

```
fig, ax = plt.subplots()

epochs = np.arange(1, len(sd_losses)+1)
ax.plot(epochs, alphas)
ax.set_xlabel("Epochs")
ax.set_ylabel(r"$\alpha$")
ax.set_title(r"$\alpha$ vs Epochs")
fig.savefig(os.path.join(SAVE_DIR, "0109.png"));
```

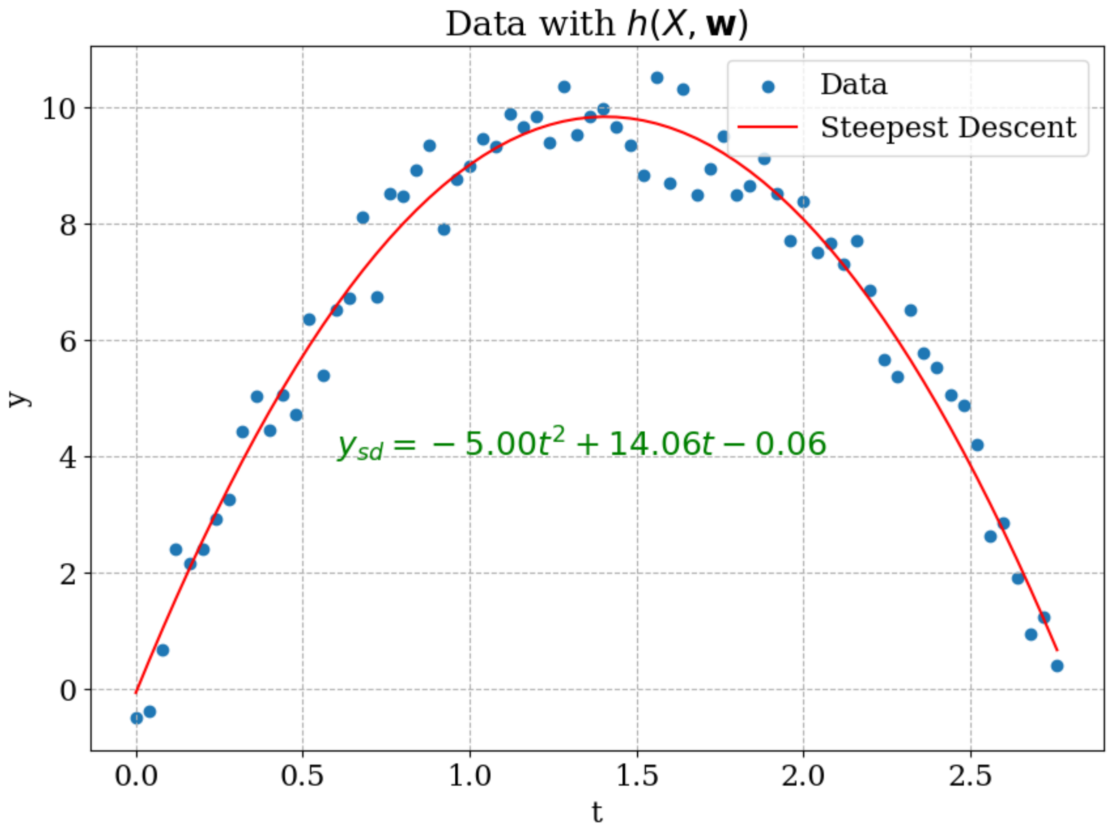


We can see that there are very few times where the learning rate is not too high, neither too low. This is the reason why the loss is fluctuating so much.

Finally, we'll plot all the hypothesis functions learned by this model.

[ ]:
```
y_pred_sd = sd.predict(X)

fig, ax = plt.subplots()
ax.scatter(t, y, label="Data")
ax.plot(t, y_pred_sd, label="Steepest Descent", color="red")
#annotate the function learned
ax.text(0.60, 4, fr"$y_{{sd}} = {sd_params[2]:.2f}t^2 + {sd_params[1]:.2f}t ␣
  ↪{sd_params[0]:.2f}$", fontsize=18, color="green")
ax.set_xlabel("t")
ax.set_ylabel("y")
ax.grid()
ax.legend()
ax.set_title("Data with $h(X, \mathbf{w})$")
plt.savefig(os.path.join(SAVE_DIR, "0110.png"));
```


Data with $h(X, \mathbf{w})$
Data
Steepest Descent
$y_{sd} = -5.00t^2 + 14.06t - 0.06$
y
t
0
2
4
6
8
10
0.0
0.5
1.0
1.5
2.0
2.5