LECTURE 12

Gradient Descent, sklearn

Optimization methods to analytically and numerically minimize loss functions.

Data 100/Data 200, Spring 2022 @ UC Berkeley

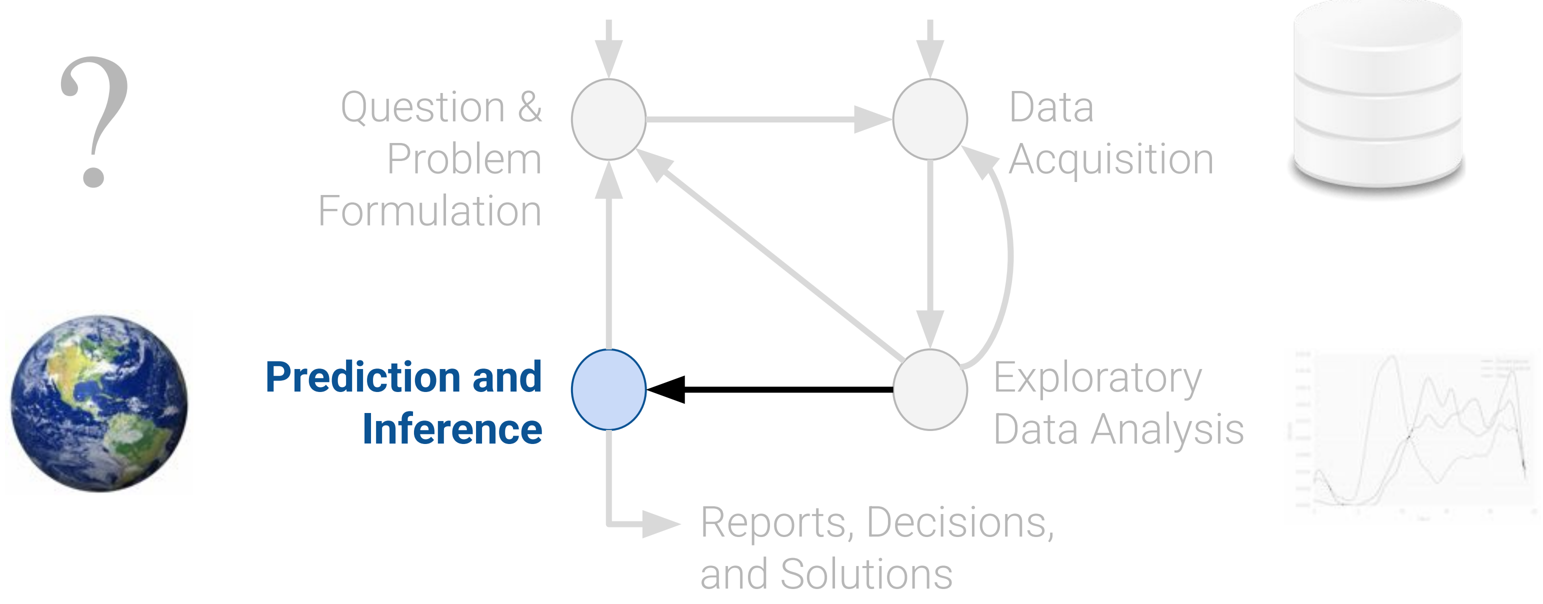
Josh Hug and Lisa Yan

Quick Note

Today's lecture is largely in a Jupyter notebook!

- I've made copies of the most important parts of the notebook in the slides, but the full details are available in code.de.

Plan for next two lectures: Model Implementation



(today)

Model Implementation I:
sklearn
Gradient Descent

Model Implementation II:
Gradient descent
Feature Engineering

Today's Goal: Ordinary Least Squares Numerically

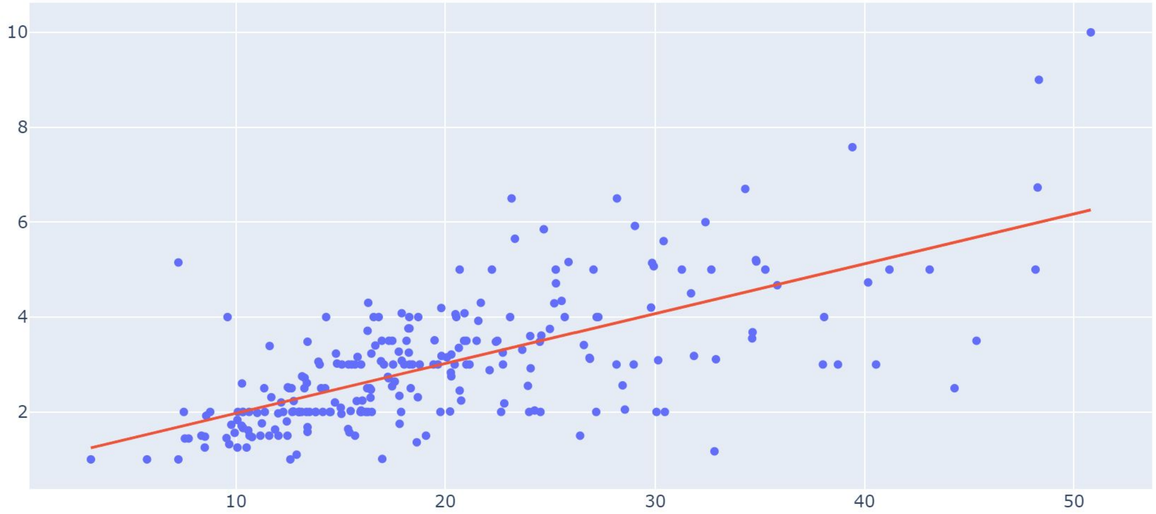
1. Choose a model	Multiple Linear Regression	For each of our n datapoints: $\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_p x_p$
2. Choose a loss function	L2 Loss Mean Squared Error (MSE)	$\hat{\mathbb{Y}} = \mathbb{X}\theta$
3. Fit the model	Minimize average loss with ~~calculus~~ ~~geometry~~ **numerical methods**	
4. Evaluate model performance	MSE	

Today's Roadmap

Lecture 12, Data 100 Spring 2022

- Simple Linear Regression
    - Using Derived Formulas
    - Using sklearn
- Multiple Linear Regression
    - Using sklearn
    - Using Derived Formulas
- Minimizing an Arbitrary 1D Function
    - Gradient Descent Example
    - Gradient Descent Implementation
- Gradient Descent on a 1D Model
- Gradient Descent in Higher Dimensions

Simple Linear Regression



$$\hat{a} = \bar{y} - \hat{b}\bar{x}$$

$$\hat{b} = r\frac{\sigma_y}{\sigma_x}$$

$$r = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{x_i - x}{\sigma_x}\right)\left(\frac{y_i - y}{\sigma_y}\right)$$

```
y = df["tip"]
x = df["total_bill"]
y_bar = np.mean(y)
x_bar = np.mean(x)
sigma_y = np.std(y)
sigma_x = np.std(x)
r = np.sum((x - x_bar) / sigma_x * (y - y_bar) / sigma_y) / len(x)
b_hat = r * sigma_y / sigma_x
a_hat = y_bar - b_hat * x_bar
```

Simple Linear Regression Using Derived Formulas

Lecture 12, Data 100 Spring 2022

Simple Linear Regression Using sklearn

Lecture 12, Data 100 Spring 2022

```python
from sklearn.linear_model import LinearRegression
model = LinearRegression()
model.fit(df[["total_bill"]], df["tip"])
df["sklearn_predictions"] = model.predict(df[["total_bill"]])
```

	total_bill	tip	sex	smoker	day	time	size	predicted_tip	residual	sklearn_predictions
0	16.99	1.01	Female	No	Sun	Dinner	2	2.704636	-1.694636	2.704636
1	10.34	1.66	Male	No	Sun	Dinner	3	2.006223	-0.346223	2.006223
2	21.01	3.50	Male	No	Sun	Dinner	3	3.126835	0.373165	3.126835
3	23.68	3.31	Male	No	Sun	Dinner	2	3.407250	-0.097250	3.407250
4	24.59	3.61	Female	No	Sun	Dinner	4	3.502822	0.107178	3.502822
5	25.29	4.71	Male	No	Sun	Dinner	4	3.576340	1.133660	3.576340
6	8.77	2.00	Male	No	Sun	Dinner	2	1.841335	0.158665	1.841335
7	26.88	3.12	Male	No	Sun	Dinner	4	3.743329	-0.623329	3.743329
8	15.04	1.96	Male	No	Sun	Dinner	2	2.499838	-0.539838	2.499838
9	14.78	3.23	Male	No	Sun	Dinner	2	2.472532	0.757468	2.472532

Multiple Linear Regression Using sklearn

Lecture 12, Data 100 Spring 2022

```python
model_2d = LinearRegression()
model_2d.fit(df[["total_bill", "size"]], df["tip"])
```

```python
model_2d.predict([[10, 3]])
```

```
array([2.17387149])
```

Multiple Linear Regression Using Derived Formulas

Lecture 12, Data 100 Spring 2022

2D Linear Model Equation

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2$$

```
print(f"theta0: {model_2d.intercept_}")
print(f"theta1 and theta2: {model_2d.coef_}")
```

```
theta0: 0.6689447408125027
theta1 and theta2: [0.09271334 0.19259779]
```

```
0.6689 + 0.0927 * 10 + 0.1926 * 3
```

```
2.1737
```

Predictions for Linear Model of Arbitrary Dimensions Using Matrix Notation

$$\hat{\mathbb{Y}} = \mathbb{X}\theta$$

```
theta = np.array([[0.6689, 0.0927, 0.1926]]).T
theta
```

```
array([[0.6689],
       [0.0927],
       [0.1926]])
```

```
X = np.array([[1, 10, 3]])
X
```

```
array([[ 1, 10,  3]])
```

```
X @ theta
```

```
array([[2.1737]])
```

Multiple Predictions for Linear Model of Arbitrary Dimensions Using Matrix Notation

$$\hat{\mathbb{Y}} = \mathbb{X}\theta$$

```python
X = df[["total_bill", "size"]].head(4)
X["bias"] = [1, 1, 1, 1]
X = X[["bias", "total_bill", "size"]]
X
```

	bias	total_bill	size
0	1	16.99	2
1	1	10.34	3
2	1	21.01	3
3	1	23.68	2

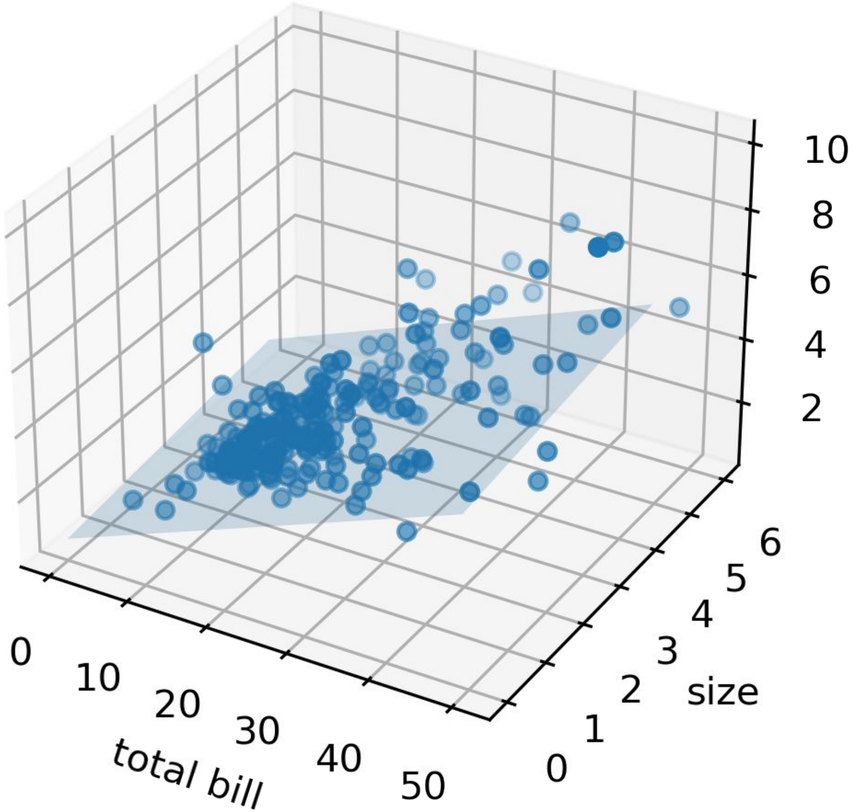
```python
theta = np.array([[0.6689, 0.0927, 0.1926]]).T
theta
```

```
array([[0.6689],
       [0.0927],
       [0.1926]])
```

```python
X @ theta
```

	0
0	2.629073
1	2.205218
2	3.194327
3	3.249236

Visualizing the Predictions of a 2D Model



$$\hat{\mathbb{Y}} = \mathbb{X}\theta$$

Predictions of our 2D linear model lie in a plane.

Finding Optimal Parameters Using the Normal Equation

$$\hat{\theta} = \left(\mathbb{X}^T \mathbb{X}\right)^{-1} \mathbb{X}^T \mathbb{Y}$$

```
X = df[["total_bill", "size"]].copy()
X["bias"] = np.ones(len(X))
X = X[["bias", "total_bill", "size"]]
X.head(4)
```

	bias	total_bill	size
0	1.0	16.99	2
1	1.0	10.34	3
2	1.0	21.01	3
3	1.0	23.68	2

```
Y = df[["tip"]]
Y.head(4)
```

	tip
0	1.01
1	1.66
2	3.50
3	3.31

```
theta_using_normal_equation = np.linalg.inv(X.T @ X) @ X.T @ Y
theta_using_normal_equation.values
```

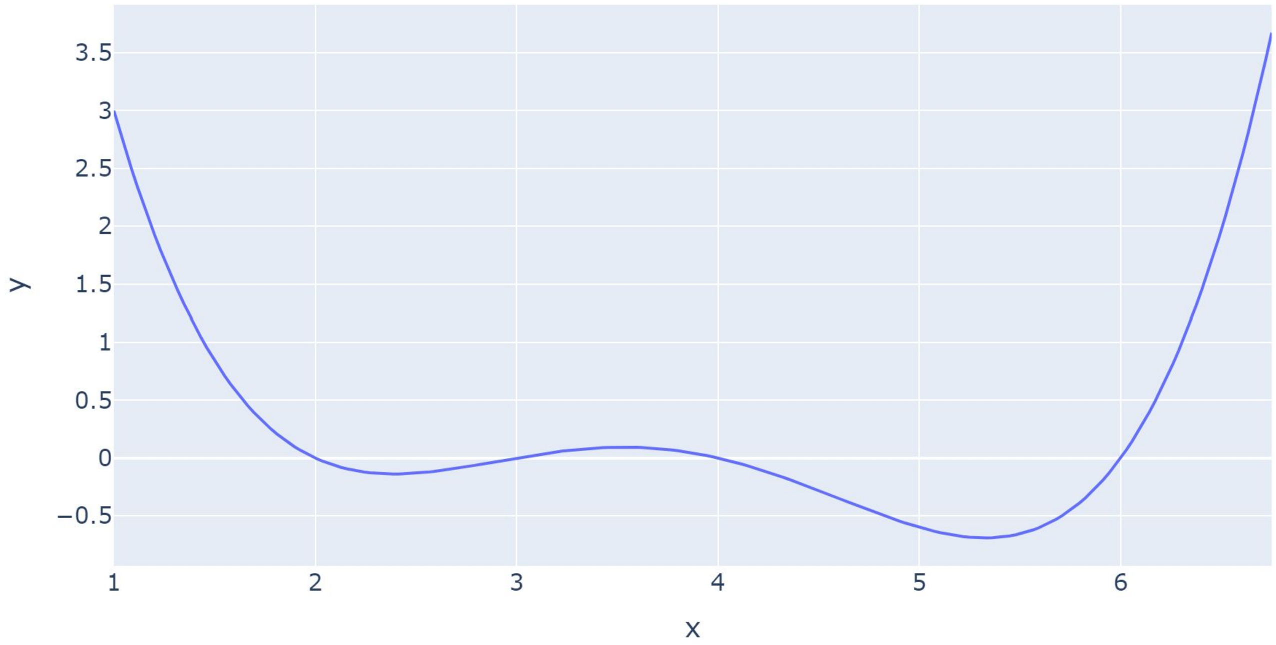
```
array([[0.66894474],
       [0.09271334],
       [0.19259779]])
```

Minimizing an Arbitrary 1D Function

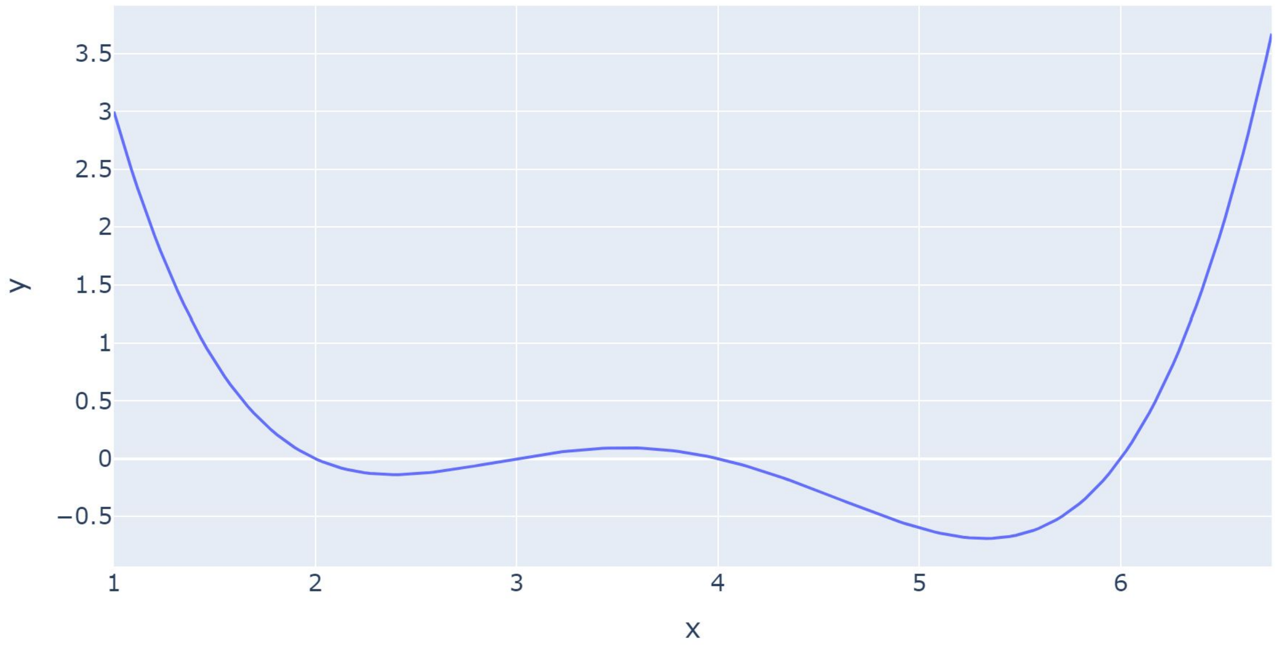
Lecture 12, Data 100 Spring 2022

Arbitrary Function of Interest

```python
def arbitrary(x):
    return (x**4 - 15*x**3 + 80*x**2 - 180*x + 144)/10

x = np.linspace(1, 6.75, 200)
fig = px.line(y = arbitrary(x), x = x)
```



Minimizing this Function using scipy.optimize.minimize



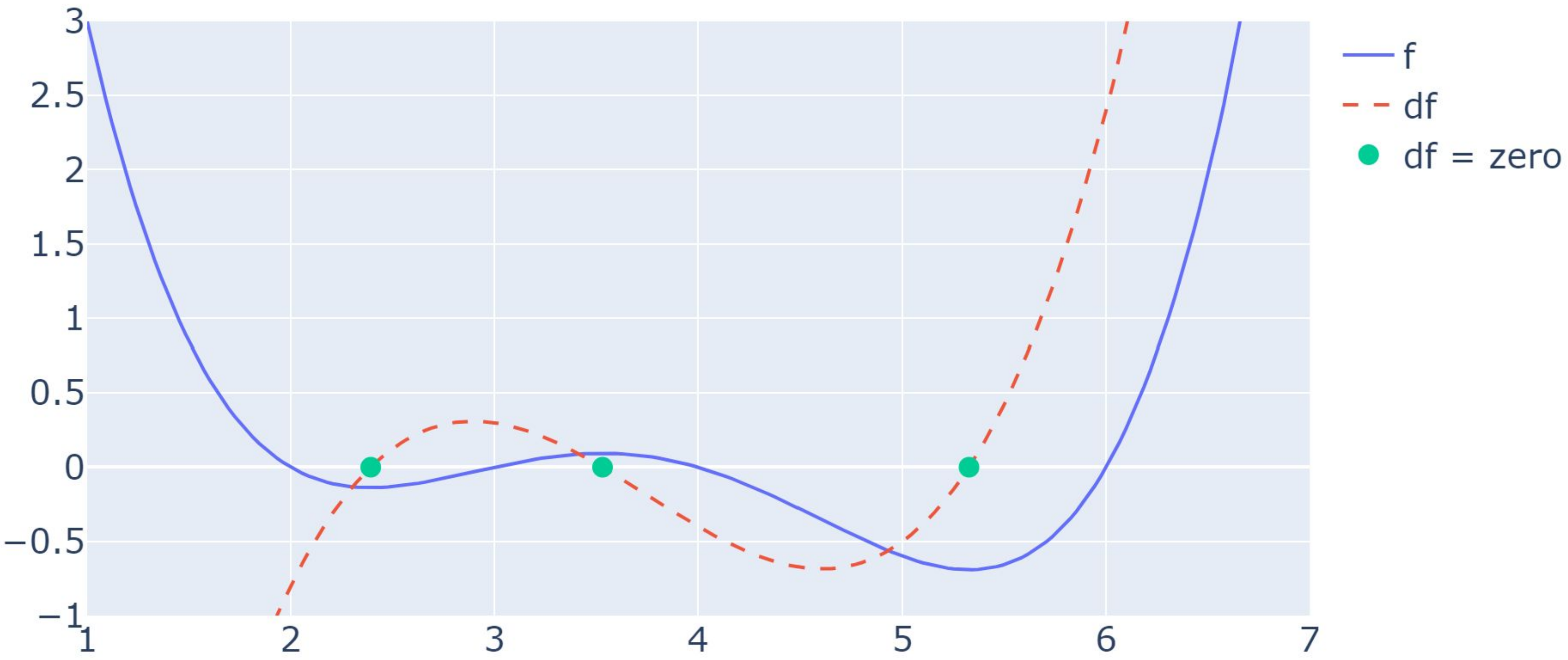

```
from scipy.optimize import minimize

minimize(arbitrary, x0 = 6)
```

```
minimize(arbitrary, x0 = 1)
```

Minimizing an Arbitrary 1D Function - Gradient Descent Example

Lecture 12, Data 100 Spring 2022

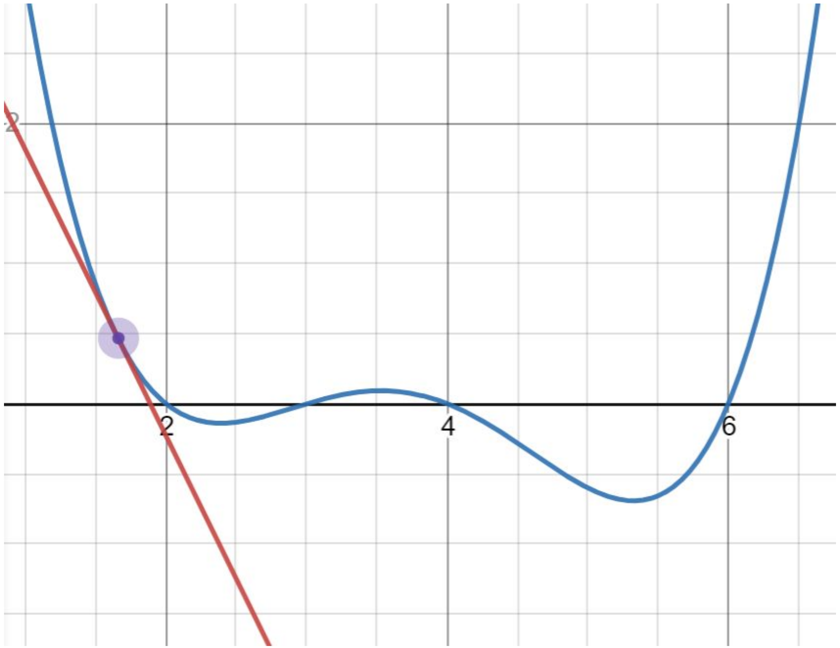
Function, Roots, and Derivatives



Derivative Tells Us Which Way to Go

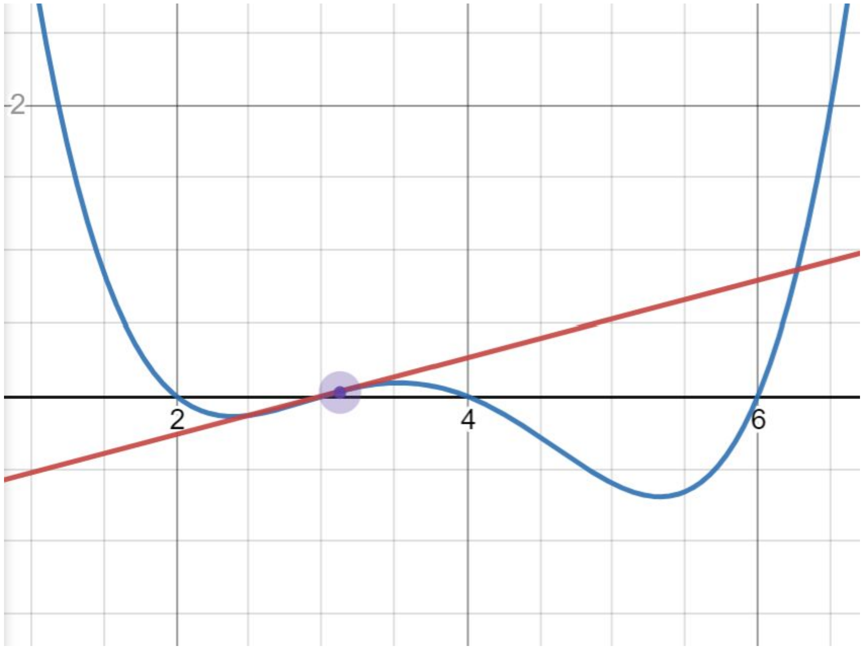
Derivative is negative, so go right.

- Follow the slope down.



Derivative is positive, so go left.

- Follow the slope down.



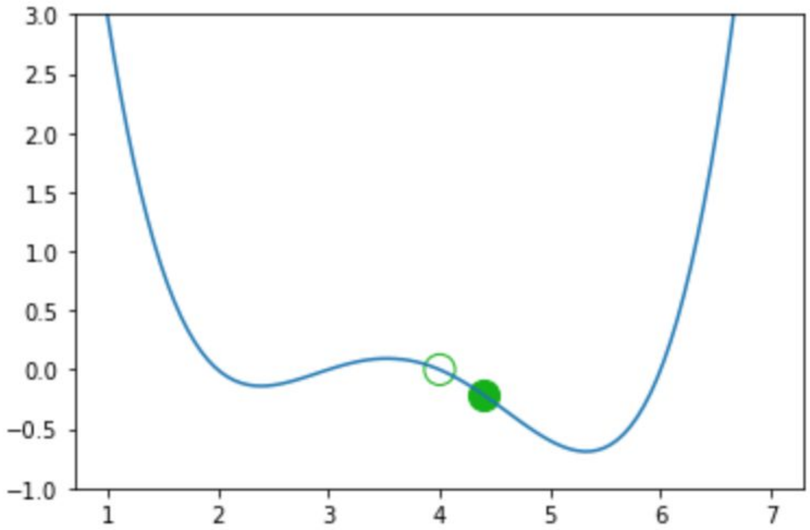
Link: https://www.desmos.com/calculator/twpnylu4lr

Manual Gradient Descent

```python
def plot_one_step(x):
    new_x = x - derivative_arbitrary(x)
    plot_arbitrary()
    plot_x_on_f(arbitrary, new_x)
    plot_x_on_f_empty(arbitrary, x)
    print(f'old x: {x}')
    print(f'new x: {new_x}')
```

```python
plot_one_step(4)
```

```
old x: 4
new x: 4.4
```

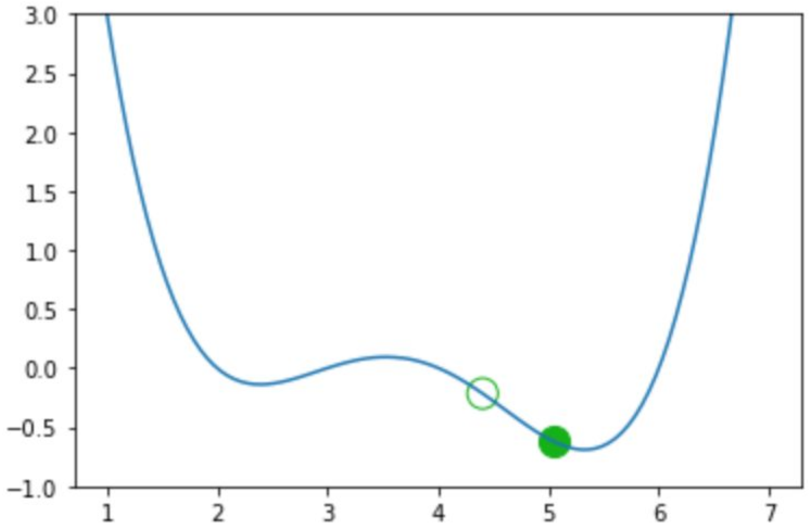


Manual Gradient Descent

```python
def plot_one_step(x):
    new_x = x - derivative_arbitrary(x)
    plot_arbitrary()
    plot_x_on_f(arbitrary, new_x)
    plot_x_on_f_empty(arbitrary, x)
    print(f'old x: {x}')
    print(f'new x: {new_x}')
```

```python
plot_one_step(4.4)
```

```
old x: 4.4
new x: 5.0464000000000055
```

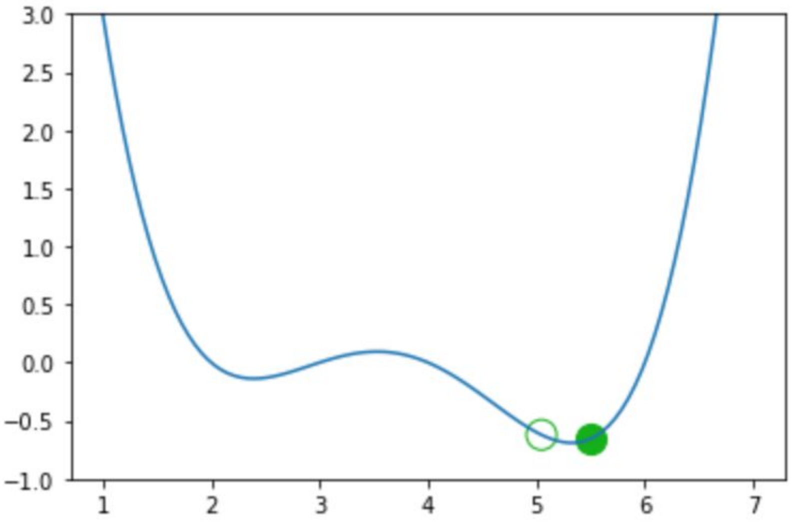


Manual Gradient Descent

```python
def plot_one_step(x):
    new_x = x - derivative_arbitrary(x)
    plot_arbitrary()
    plot_x_on_f(arbitrary, new_x)
    plot_x_on_f_empty(arbitrary, x)
    print(f'old x: {x}')
    print(f'new x: {new_x}')
```

```python
plot_one_step(5.0464)
```

```
old x: 5.0464
new x: 5.49673060106241
```

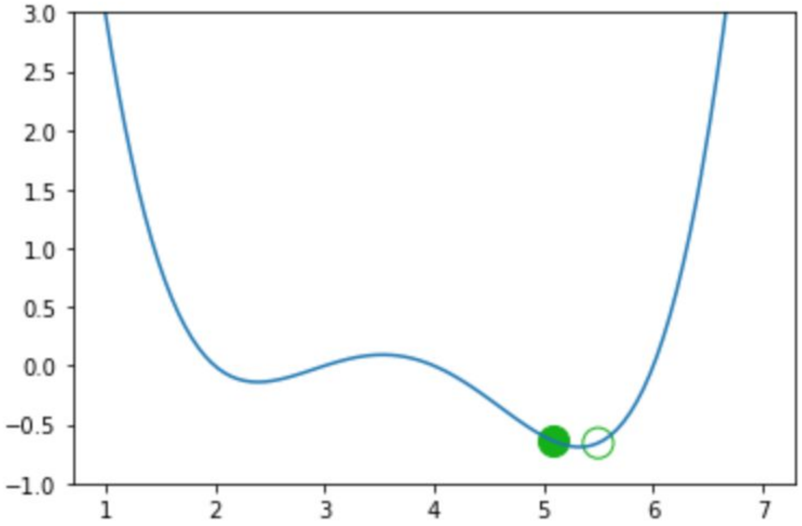


Manual Gradient Descent

```python
def plot_one_step(x):
    new_x = x - derivative_arbitrary(x)
    plot_arbitrary()
    plot_x_on_f(arbitrary, new_x)
    plot_x_on_f_empty(arbitrary, x)
    print(f'old x: {x}')
    print(f'new x: {new_x}')
```

```python
plot_one_step(5.4967)
```

```
old x: 5.4967
new x: 5.080917145374805
```

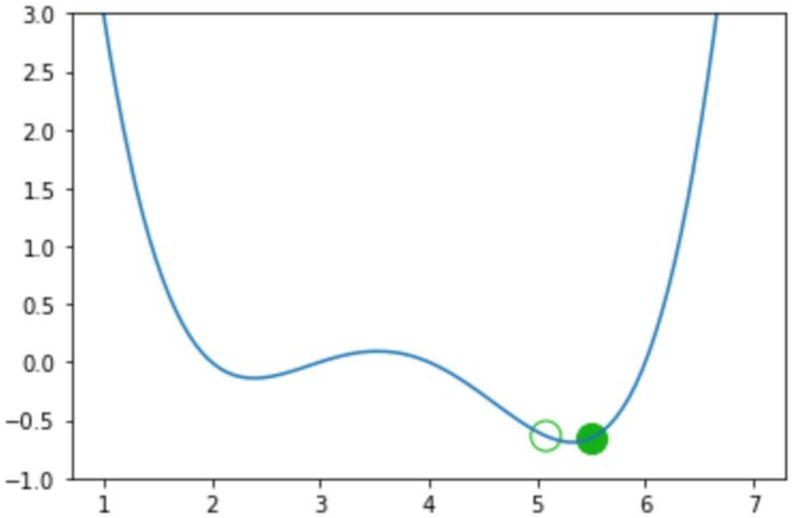


Manual Gradient Descent

```python
def plot_one_step(x):
    new_x = x - derivative_arbitrary(x)
    plot_arbitrary()
    plot_x_on_f(arbitrary, new_x)
    plot_x_on_f_empty(arbitrary, x)
    print(f'old x: {x}')
    print(f'new x: {new_x}')
```

```python
plot_one_step(5.080917145374805)
```

```
old x: 5.080917145374805
new x: 5.489966698640582
```

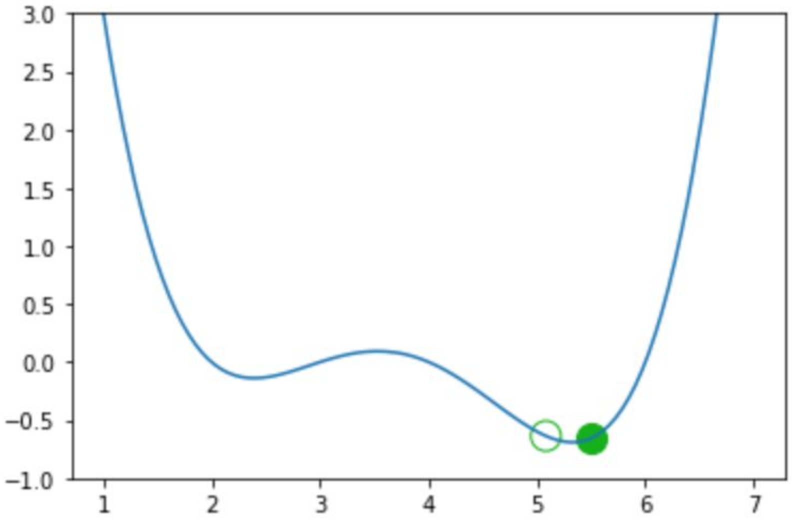


Manual Gradient Descent

```python
def plot_one_step(x):
    new_x = x - derivative_arbitrary(x)
    plot_arbitrary()
    plot_x_on_f(arbitrary, new_x)
    plot_x_on_f_empty(arbitrary, x)
    print(f'old x: {x}')
    print(f'new x: {new_x}')
```

```python
plot_one_step(5.080917145374805)
```

```
old x: 5.080917145374805
new x: 5.489966698640582
```



We appear to be bouncing back and forth. Turns out we are stuck!

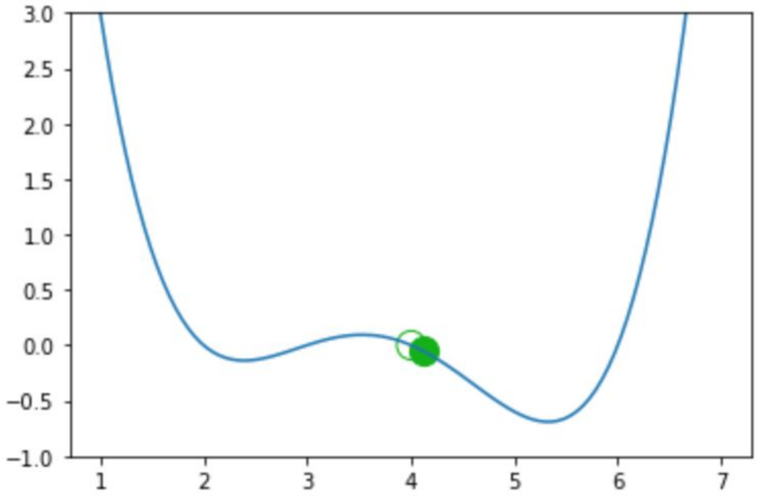
- Any suggestions for how we can avoid this issue?

Manual Gradient Descent with Slower “Learning Rate”

```python
def plot_one_step_better(x):
    new_x = x - 0.3 * derivative_arbitrary(x)
    plot_arbitrary()
    plot_x_on_f(arbitrary, new_x)
    plot_x_on_f_empty(arbitrary, x)
    print(f'old x: {x}')
    print(f'new x: {new_x}')
```

```python
plot_one_step_better(4)
```

```
old x: 4
new x: 4.12
```

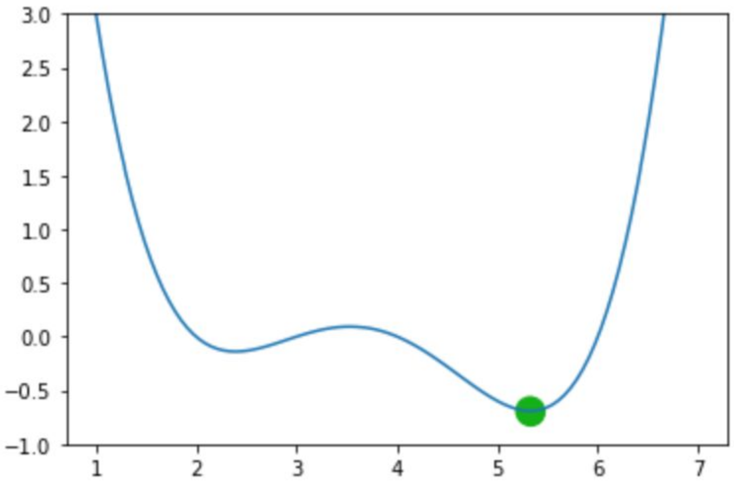


Manual Gradient Descent with Slower “Learning Rate” (many steps later)

```
def plot_one_step_better(x):
    new_x = x - 0.3 * derivative_arbitrary(x)
    plot_arbitrary()
    plot_x_on_f(arbitrary, new_x)
    plot_x_on_f_empty(arbitrary, x)
    print(f'old x: {x}')
    print(f'new x: {new_x}')
```

```
plot_one_step_better(5.323)
```

```
old x: 5.323
new x: 5.325108157959999
```



Gradient Descent Implementation

Lecture 12, Data 100 Spring 2022

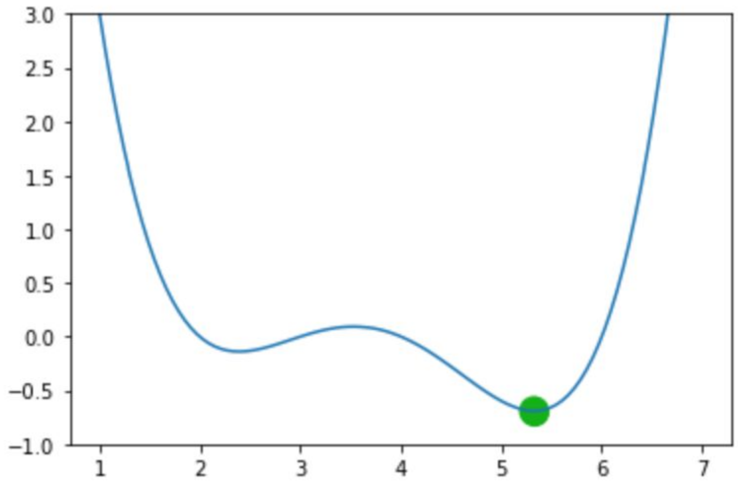
Gradient Descent as a Recurrence Relation

```python
def plot_one_step_better(x):
    new_x = x - 0.3 * derivative_arbitrary(x)
    plot_arbitrary()
    plot_x_on_f(arbitrary, new_x)
    plot_x_on_f_empty(arbitrary, x)
    print(f'old x: {x}')
    print(f'new x: {new_x}')
```

$$x^{(t+1)} = x^{(t)} - 0.3\frac{d}{dx}f(x)$$

```python
plot_one_step_better(5.323)
```

```
old x: 5.323
new x: 5.325108157959999
```



Our Recurrence Relation as Iterative Code

$$x^{(t+1)} = x^{(t)} - \alpha \frac{d}{dx} f(x)$$

```
def gradient_descent(df, initial_guess, alpha, n):
    """Performs n steps of gradient descent on df using learning rate alpha starting
       from initial_guess. Returns a numpy array of all guesses over time."""
    guesses = [initial_guess]
    current_guess = initial_guess
    while len(guesses) < n:
        current_guess = current_guess - alpha * df(current_guess)
        guesses.append(current_guess)

    return np.array(guesses)
```

```
trajectory = gradient_descent(derivative_arbitrary, 4, 0.3, 20)
trajectory
```

```
array([4.        , 4.12      , 4.26729664, 4.44272584, 4.64092624,
       4.8461837 , 5.03211854, 5.17201478, 5.25648449, 5.29791149,
       5.31542718, 5.3222606 , 5.32483298, 5.32578765, 5.32614004,
       5.32626985, 5.32631764, 5.32633523, 5.3263417 , 5.32634408])
```

Our Recurrence Relation as Iterative Code

$$x^{(t+1)} = x^{(t)} - \alpha \frac{d}{dx} f(x)$$

```python
def gradient_descent(df, initial_guess, alpha, n):
    """Performs n steps of gradient descent on df using learning rate alpha starting
       from initial_guess. Returns a numpy array of all guesses over time."""
    guesses = [initial_guess]
    current_guess = initial_guess
    while len(guesses) < n:
        current_guess = current_guess - alpha * df(current_guess)
        guesses.append(current_guess)

    return np.array(guesses)
```

```python
trajectory = gradient_descent(derivative_arbitrary, 4, 1, 20)
trajectory
```

```
array([4.        , 4.4       , 5.0464    , 5.4967306 , 5.08086249,
       5.48998039, 5.09282487, 5.48675539, 5.09847285, 5.48507269,
       5.10140255, 5.48415922, 5.10298805, 5.48365325, 5.10386474,
       5.48336998, 5.1043551 , 5.48321045, 5.10463112, 5.48312031])
```

Convergence of Gradient Descent

There is a rich literature exploring the convergence of many variants of gradient descent.

- Well beyond the scope of our course!
- For more, see a dedicated course in mathematical **optimization**.

Gradient Descent on a 1D Model

Lecture 12, Data 100 Spring 2022

Applying Gradient Descent to Our Tips Dataset

We've seen how to find the optimal parameters for a 1D linear model for the tips dataset:

- Using the derived equations from data 8.
- Using sklearn.
    - Uses gradient descent!

While in real practice in this course, you'll usually use sklearn, let's see how we can do the gradient descent ourselves.

We'll first fit a model that has no y-intercept, for maximum simplicity.

Let's try this out in our notebook.

Gradient Descent in Higher Dimensions

Lecture 12, Data 100 Spring 2022

A Two Parameter Model

Suppose we now try simple linear regression, which has two parameters:

$$\text{tip} = \theta_0 + \theta_1 \text{bill}$$

We'll use gradient descent to minimize the function below:

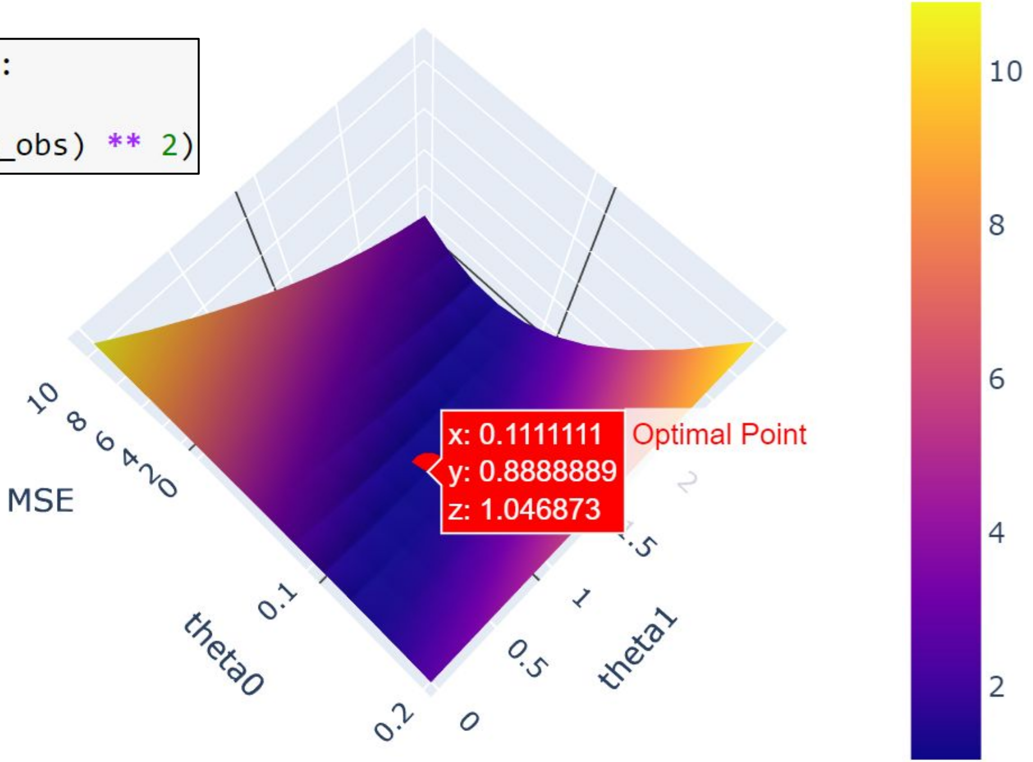
- Here, theta is a two dimensional vector!

```
def mse_loss(theta, X, y_obs):
    y_hat = X @ theta
    return np.mean((y_hat - y_obs) ** 2)
```

A 2D Loss Function

Here, we see the loss of our model as a function of our two parameters.

```python
def mse_loss(theta, X, y_obs):
    y_hat = X @ theta
    return np.mean((y_hat - y_obs) ** 2)
```



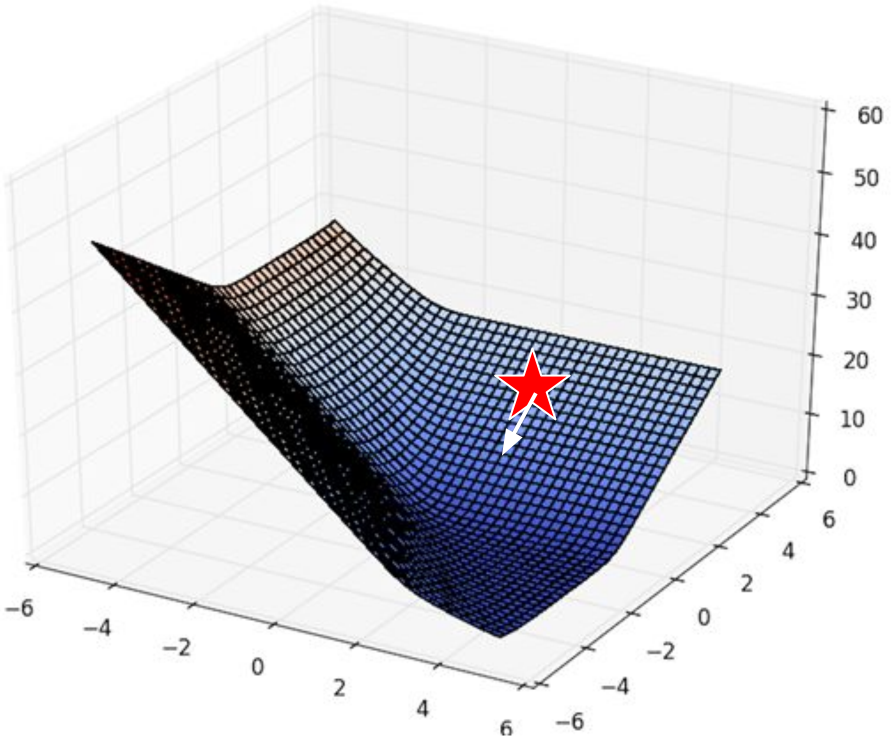
Gradient Descent

Just like earlier, we can follow the slope of our 2D function.

- On a 2D surface, the best way to go down is described by a 2D vector.

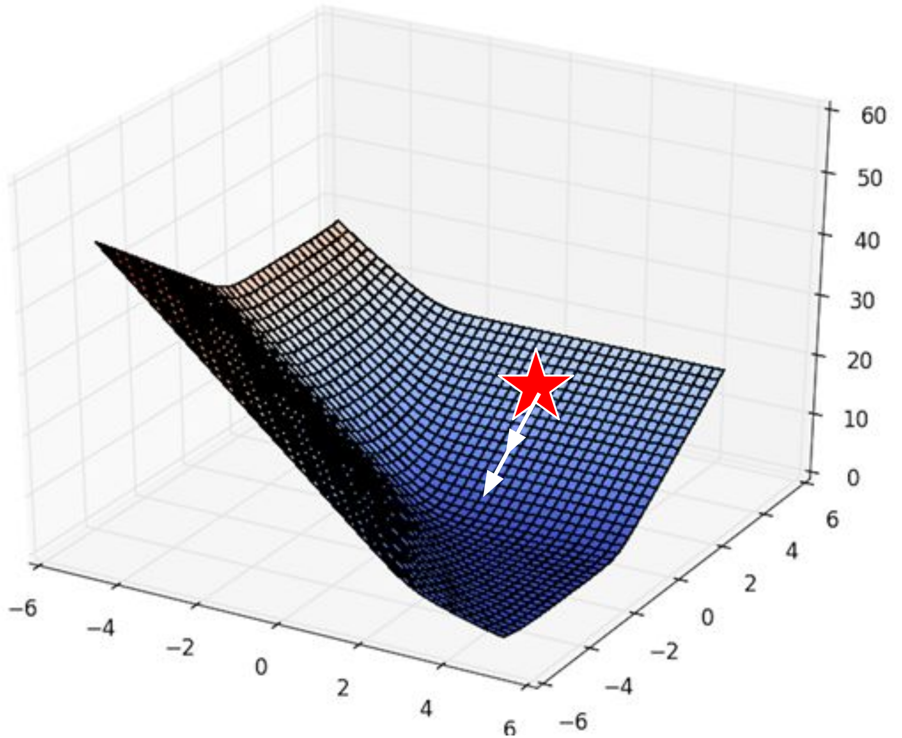
Approach #3: Gradient Descent

On a 2D surface, the best way to go down is described by a 2D vector.



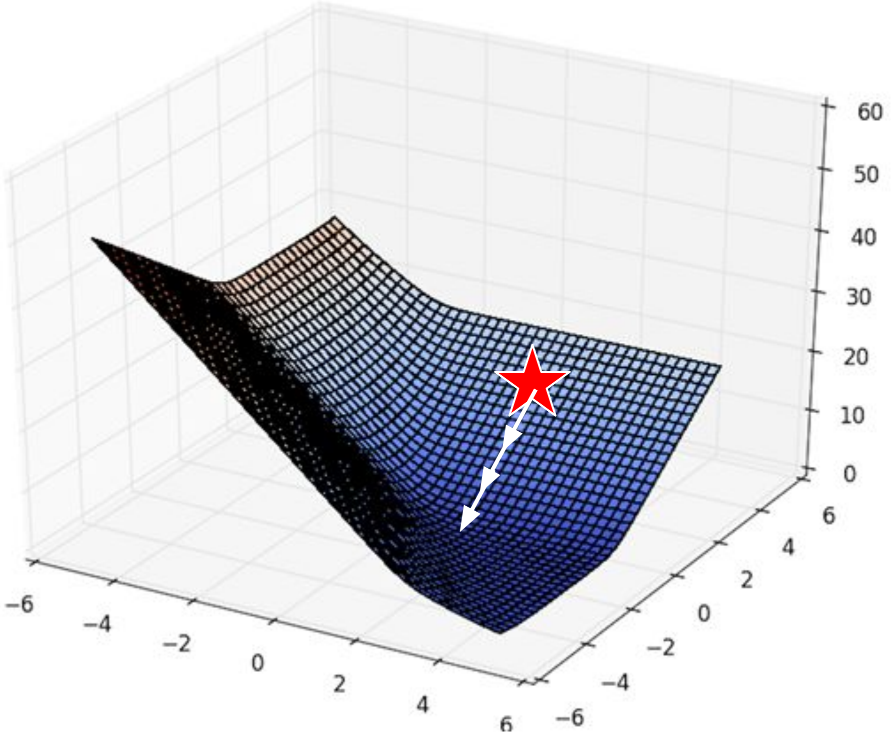
Approach #3: Gradient Descent

On a 2D surface, the best way to go down is described by a 2D vector.



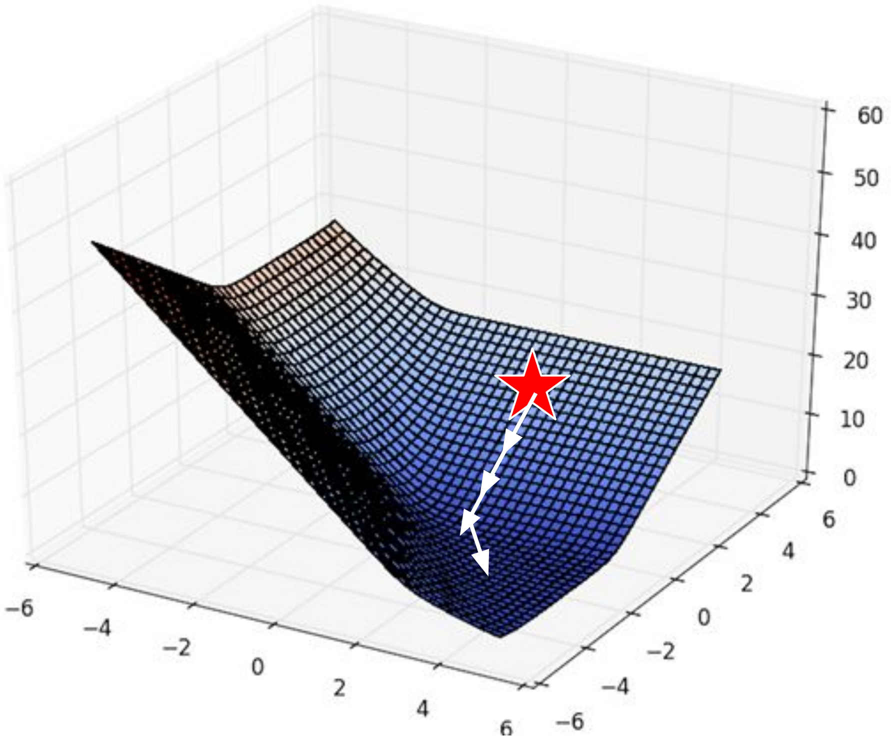
Approach #3: Gradient Descent

On a 2D surface, the best way to go down is described by a 2D vector.



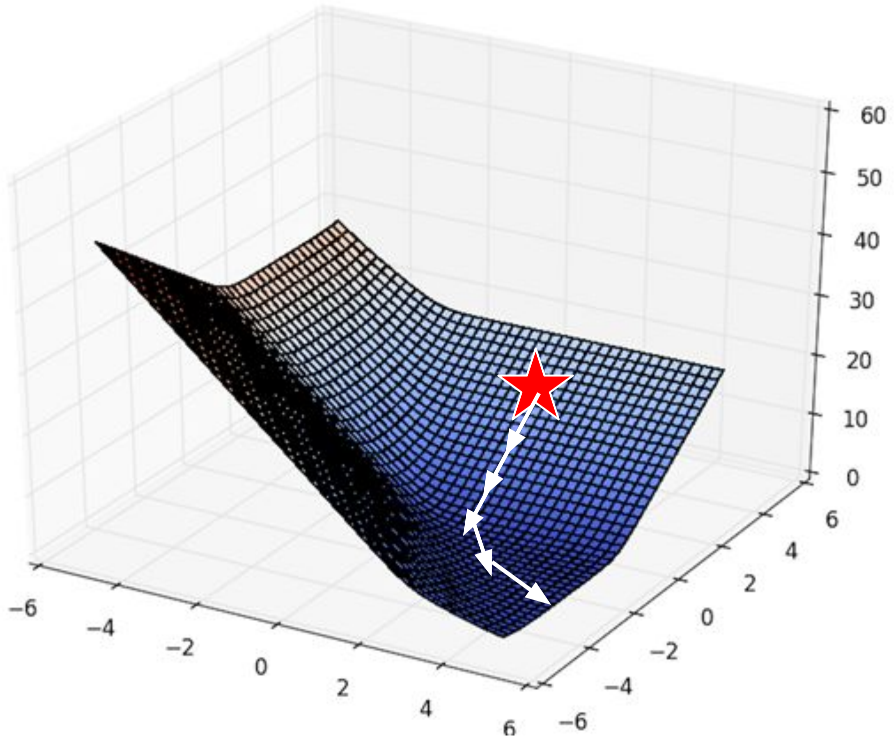
Approach #3: Gradient Descent

On a 2D surface, the best way to go down is described by a 2D vector.



Approach #3: Gradient Descent

On a 2D surface, the best way to go down is described by a 2D vector.



Example: Gradient of a 2D Function

Consider the 2D function: $$f(\theta_0, \theta_1) = 8\theta_0^2 + 3\theta_0\theta_1$$

For a function of 2 variables, $f(\theta_0, \theta_1)$ we define the gradient $\nabla_{\vec{\theta}} f = \frac{\partial f}{\partial \theta_0}\vec{i} + \frac{\partial f}{\partial \theta_1}\vec{j}$, where $\vec{i}$ and $\vec{j}$ are the unit vectors in the θ_0 and θ_1 directions.

$$\frac{\partial f}{\partial \theta_0} = 16\theta_0 + 3\theta_1$$

$$\frac{\partial f}{\partial \theta_1} = 3\theta_0$$

$$\nabla_{\vec{\theta}} f = (16\theta_0 + 3\theta_1)\vec{i} + 3\theta_0\vec{j}$$

Example: Gradient of a 2D Function in Column Vector Notation

Consider the 2D function: $$f(\theta_0, \theta_1) = 8\theta_0^2 + 3\theta_0\theta_1$$

Gradients are also often written in column vector notation.

$$\nabla_{\vec{\theta}} f(\vec{\theta}) = \begin{bmatrix} 16\theta_0 + 3\theta_1 \\ 3\theta_0 \end{bmatrix}$$

Example: Gradient of a Function in Column Vector Notation

For a generic function of p + 1 variables.

$$\nabla_{\vec{\theta}} f(\vec{\theta}) = \begin{bmatrix} \frac{\partial}{\partial \theta_0}(f) \\ \frac{\partial}{\partial \theta_1}(f) \\ \vdots \\ \frac{\partial}{\partial \theta_p}(f) \end{bmatrix}$$

How to Interpret Gradients

- You should read these gradients as:
  - If I nudge the 1st model weight, what happens to loss?
  - If I nudge the 2nd, what happens to loss?
  - Etc.

You Try:

Derive the gradient descent rule for a linear model with two model weights and MSE loss.

- Below we'll consider just one observation (i.e. one row of our data).

$$f_{\vec{\theta}}(\vec{x}) = \vec{x}^T\vec{\theta} = \theta_0 x_0 + \theta_1 x_1$$

$$\ell(\vec{\theta}, \vec{x}, y_i) = (y_i - \theta_0 x_0 - \theta_1 x_1)^2$$

Squared loss for a single prediction of our linear regression model.

$$\nabla_\theta \ell(\vec{\theta}, \vec{x}, y_i) = ?$$

You Try:

$$\ell(\vec{\theta}, \vec{x}, y_i) = (y_i - \theta_0 x_0 - \theta_1 x_1)^2$$

$$\frac{\partial}{\partial \theta_0} \ell(\vec{\theta}, \vec{x}, y_i) = 2(y_i - \theta_0 x_0 - \theta_1 x_1)(-x_0)$$

$$\frac{\partial}{\partial \theta_1} \ell(\vec{\theta}, \vec{x}, y_i) = 2(y_i - \theta_0 x_0 - \theta_1 x_1)(-x_1)$$

$$\nabla_\theta \ell(\vec{\theta}, \vec{x}, y_i) = \begin{bmatrix} -2(y_i - \theta_0 x_0 - \theta_1 x_1)(x_0) \\ -2(y_i - \theta_0 x_0 - \theta_1 x_1)(x_1) \end{bmatrix}$$

Summary

Gradient descent allows us to find the minima of functions.

- At each step, we compute the steepest direction of the function we're minimizing, yielding a p-dimensional vector.
- Our next guess for the optimal solution is our current solution minus this p-dimensional vector times a learning rate alpha.

(An earlier version of this slide mentioned convex functions. This concept will appear in the next lecture.)

LECTURE 12

Gradient Descent, sklearn

Content credit: Josh Hug, Joseph Gonzalez