https://tinyurl.com/data100-lec13

LECTURE 13

sklearn, Gradient Descent

Optimization methods to analytically and numerically minimize loss functions.

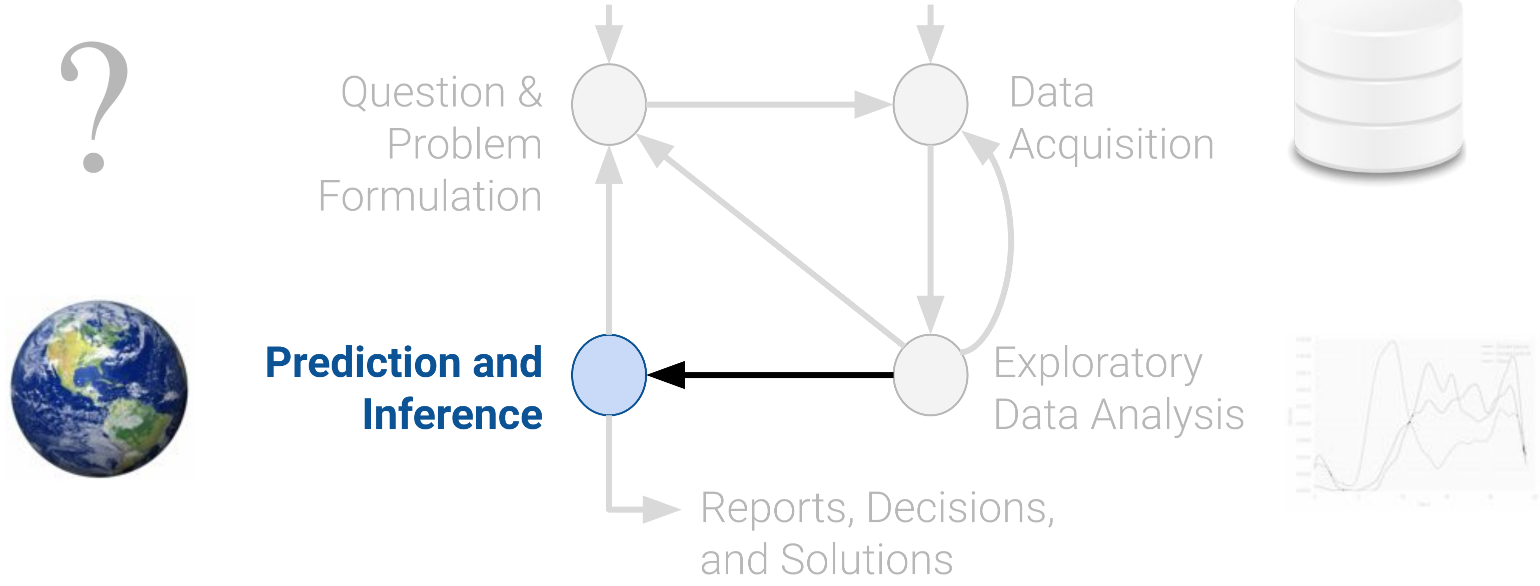
Data 100/Data 200, Fall 2022 @ UC Berkeley

Will Fithian and Fernando Pérez

Content credit: Josh Hug and Joseph Gonzalez

Plan for next two lectures: Model Implementation

3615537



(today)

Model Implementation I:
sklearn
Gradient Descent

Model Implementation II:
Gradient descent
Feature Engineering

Today's Goal: Ordinary Least Squares Numerically

3615537

1. Choose a model

Multiple Linear Regression

For each of our n datapoints:

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

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

2. Choose a loss function

L2 Loss

Mean Squared Error (MSE)

3. Fit the model

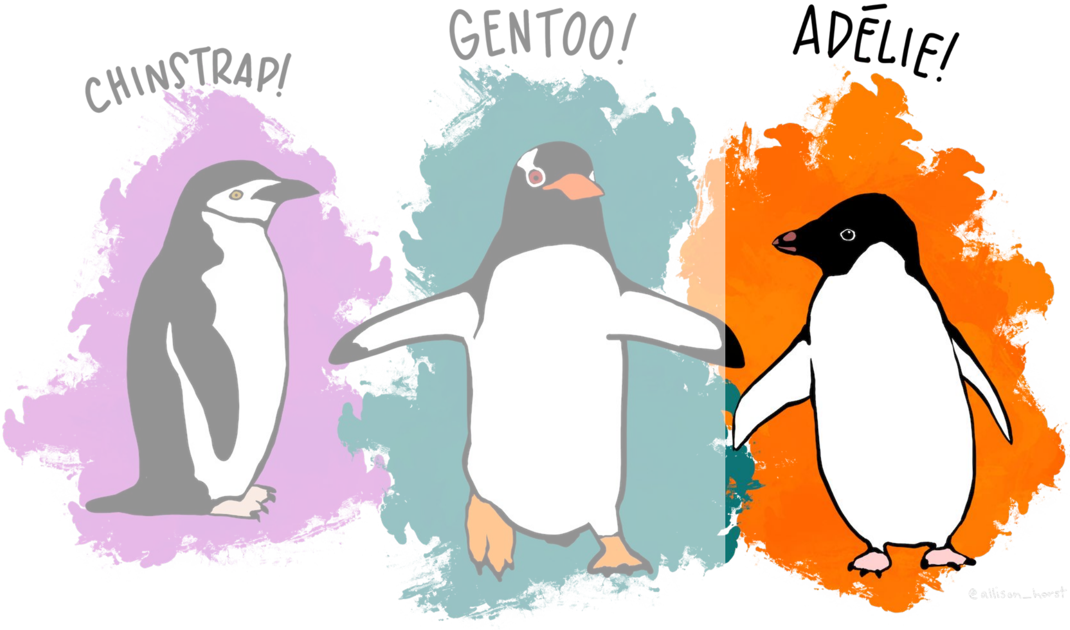
Minimize average loss with ~~calculus~~ ~~geometry~~ **numerical methods**

4. Evaluate model performance

MSE

Today: Penguins

3615537



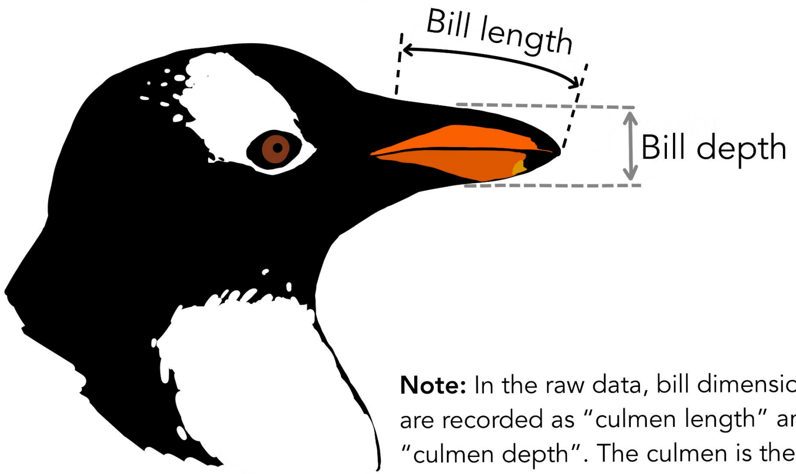

```
df = sns.load_dataset("penguins")
df = df[df["species"] == "Adelie"].dropna()
df
```

https://github.com/allisonhorst/palmerpenguins

Today's Penguin Task

3615537

	species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g	sex
0	Adelie	Torgersen	39.1	18.7	181.0	3750.0	Male
1	Adelie	Torgersen	39.5	17.4	186.0	3800.0	Female
2	Adelie	Torgersen	40.3	18.0	195.0	3250.0	Female
4	Adelie	Torgersen	36.7	19.3	193.0	3450.0	Female
5	Adelie	Torgersen	39.3	20.6	190.0	3650.0	Male
				...	...	...	...
				18.4	184.0	3475.0	Female
				17.8	195.0	3450.0	Female
				18.1	193.0	3750.0	Male
				17.1	187.0	3700.0	Female
				18.5	201.0	4000.0	Male



Note: In the raw data, bill dimensions are recorded as "culmen length" and "culmen depth". The culmen is the dorsal ridge atop the bill.

Suppose we could measure flippers and weight easily, but not bill dimensions.

How can we **predict** bill depth from flipper length and/or body mass?

Quick Note

Today's lecture is largely in a Jupyter notebook!

Follow along: https://ds100.org/sp23/lecture/lec13/

- Only formulas and notes are in this slide deck.

First, we'll review what you saw in Lab 06 this week.

3615537

Today's Roadmap

Lecture 12, Data 100 Fall 2022

- Simple Linear Regression in Code
- Ordinary Least Squares in Code
- 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 in Code

Lecture 12, Data 100 Fall 2022

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

3615537

slido

What should we pass in for X?

ⓘ Start presenting to display the poll results on this slide.

fit(*X*, *y*, *sample_weight=None*) [source]

Fit linear model.

Parameters:	**X : *{array-like, sparse matrix} of shape (n_samples, n_features)*** Training data. **y : *array-like of shape (n_samples,) or (n_samples, n_targets)*** Target values. Will be cast to X's dtype if necessary. **sample_weight : *array-like of shape (n_samples,), default=None*** Individual weights for each sample. *New in version 0.17:* parameter *sample_weight* support to LinearRegression.
Returns:	**self : *object*** Fitted Estimator.

Sklearn summary

3615537

1. Create an `sklearn` object.

```
from sklearn.linear_model import LinearRegression
model = LinearRegression()
```

2. `fit` the object to data.

   X is a **DataFrame** of (n_samples, n_features), hence the double brackets.

   y is a **Series** of (n_samples,), observations to predict.

```
X = df[["flipper_length_mm"]]
y = df["bill_depth_mm"]
model.fit(X, y)
```

3. Analyze fit or call `predict`.

```
df["sklearn_preds"] = model.predict(df[["flipper_length_mm"]])
```

```
from sklearn.metrics import mean_squared_error
mean_squared_error(df["bill_depth_mm"], df["sklearn_preds"])
```

```
model.intercept_        # why is this a scalar?
```

```
7.297305899612306
```

```
model.coef_             # why is this an array?
```

```
array([0.05812622])
```

https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html

3615537

Ordinary Least Squares in Code

Lecture 12, Data 100 Fall 2022

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

3615537

Multiple Linear Regression:

Linear model between ≥2 features and an observation y, with parameters θ.

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

Ordinary Least Squares

Choose optimal $\hat{\theta}$ that minimizes mean square error.

Normal Equation

The OLS solution $\hat{\theta}$ solves the normal equation.

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

If design matrix X is full column rank, then OLS solution is $\hat{\theta} = \left(\mathbb{X}^T\mathbb{X}\right)^{-1}\mathbb{X}^T\mathbb{Y}$

Bias Term / Intercept Review

To represent multiple linear regression **with an intercept** as a matrix multiplication:

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

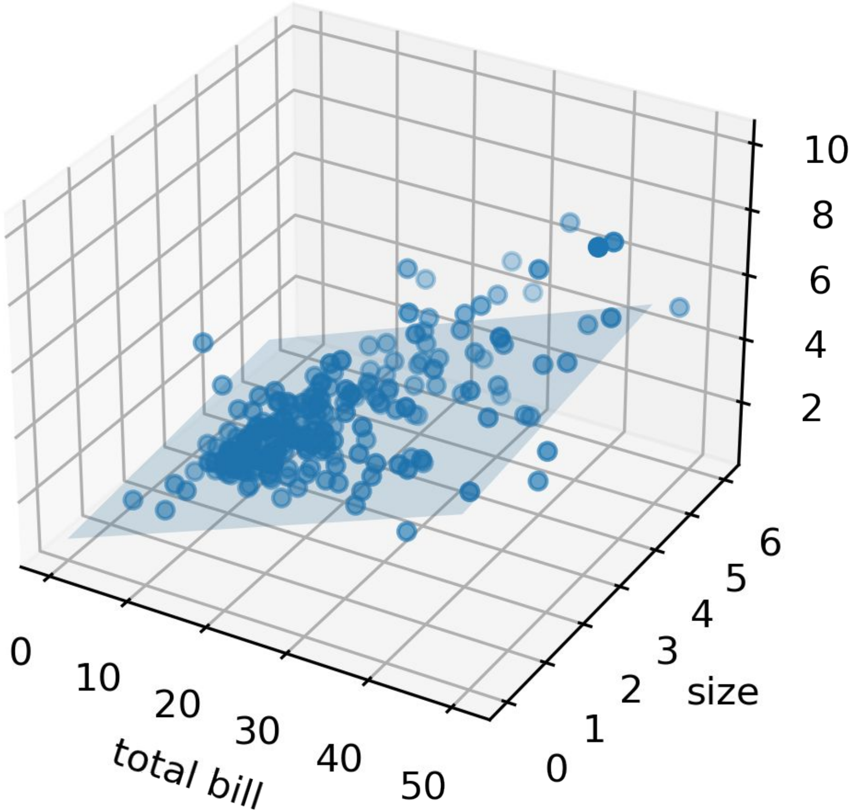
We need to introduce a **bias vector** into our design matrix.

$$\mathbb{X} = \begin{bmatrix} 1 & x_{11} & \dots & x_{1p} \\ 1 & x_{21} & \dots & x_{2p} \\ 1 & x_{31} & \dots & x_{3p} \\ \vdots & \vdots & \vdots & \vdots \\ 1 & x_{n1} & \dots & x_{np} \end{bmatrix}$$

This is because we are representing this dot product for each datapoint:

$$\hat{y} = \begin{bmatrix} 1 & x_1 & x_2 \end{bmatrix} \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \end{bmatrix}$$

Visualizing the Predictions of a 2D Model



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

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

Minimizing an Arbitrary 1D Function

Lecture 12, Data 100 Fall 2022

3615537

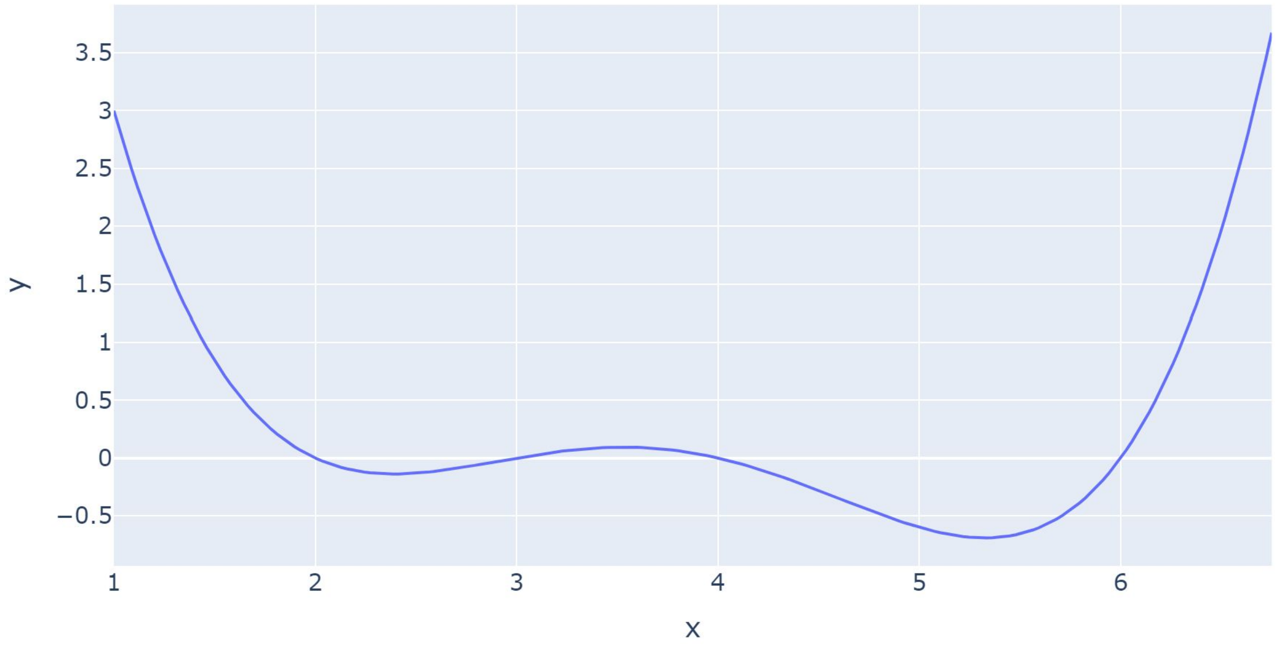
- Simple Linear Regression
- Multiple Linear Regression
- Minimizing an Arbitrary 1D Function
    - Gradient Descent Example
    - Gradient Descent Implementation
- Gradient Descent on a 1D Model
- Gradient Descent in Higher Dimensions

Arbitrary Function of Interest

3615537

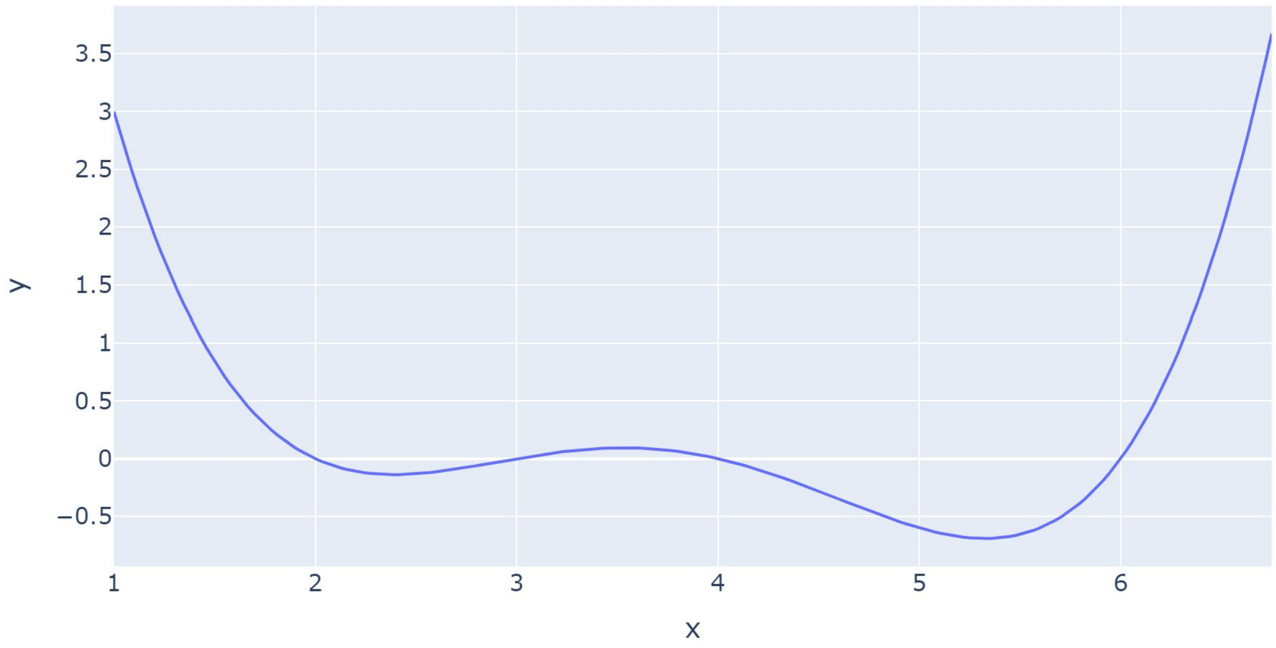
```
def arbitrary(x):
    return (x**4 - 15*x**3 + 80*x**2 - 180*x + 144)/10

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



Minimizing this Function using scipy.optimize.minimize

3615537




```python
from scipy.optimize import minimize

minimize(arbitrary, x0 = 6)
```

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

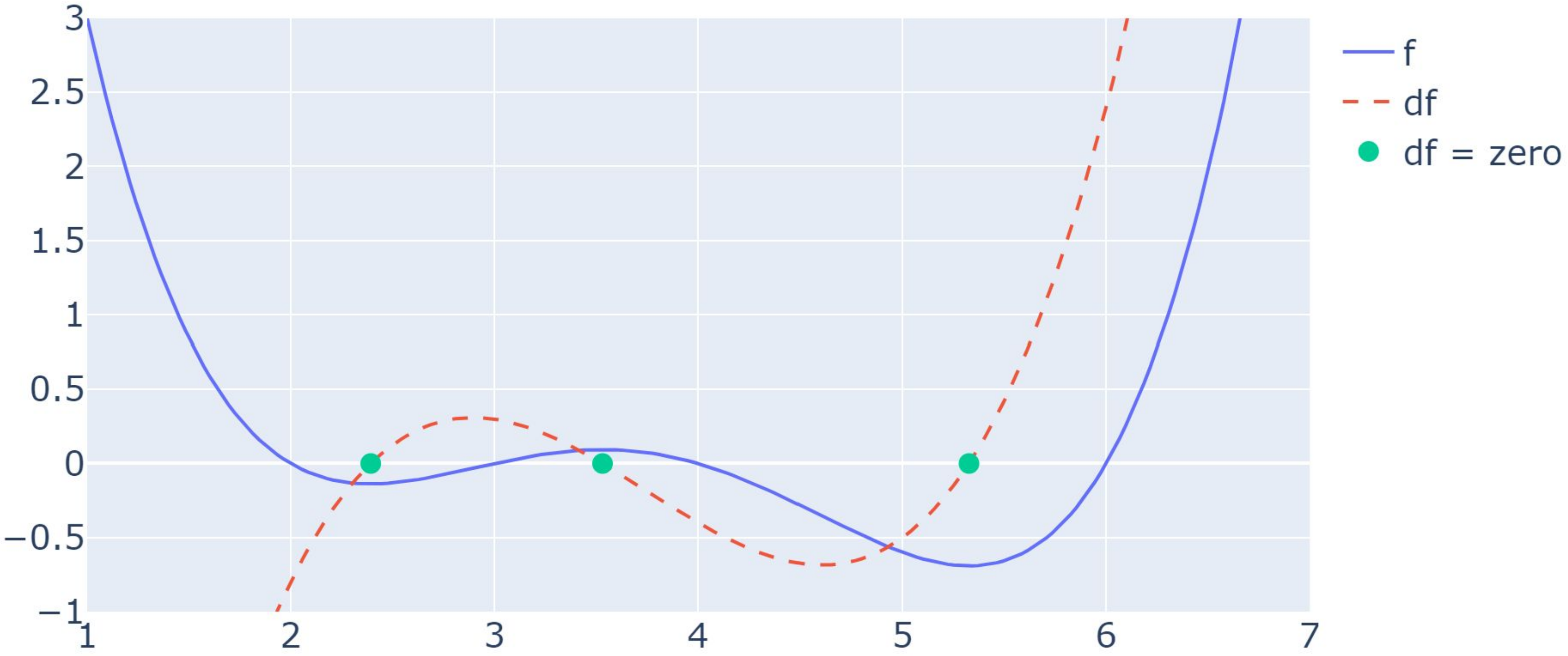
Minimizing an Arbitrary 1D Function - Gradient Descent Example

Lecture 12, Data 100 Fall 2022

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

3615537

Function, Roots, and Derivatives

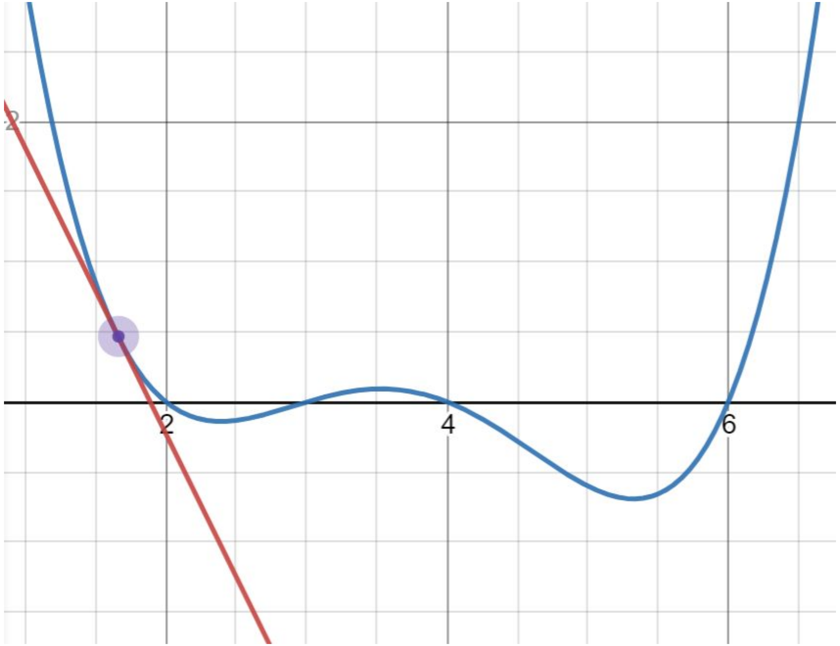


Derivative Tells Us Which Way to Go

3615537

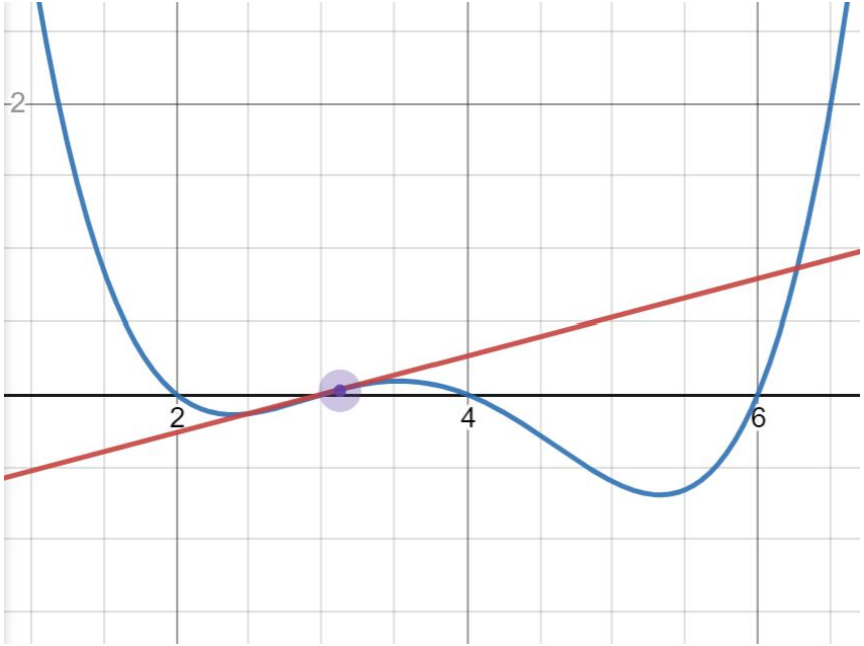
Derivative is negative, so go right.

- Follow the slope down.



Derivative is positive, so go left.

- Follow the slope down.



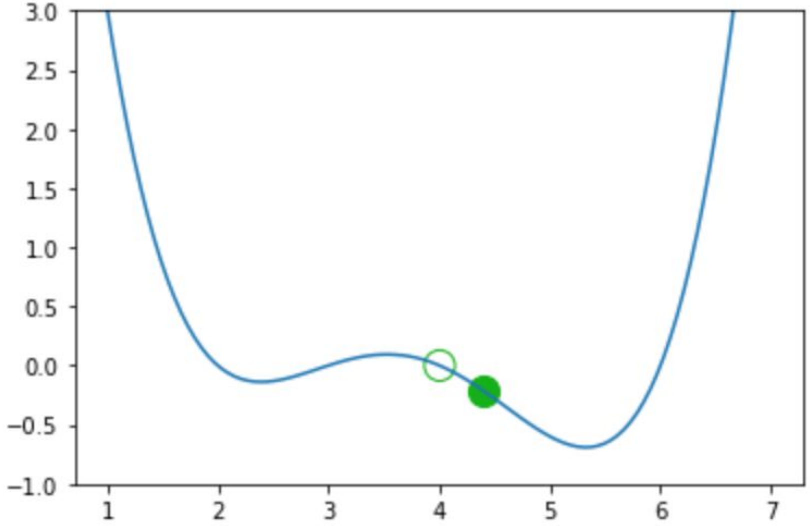
Link: https://www.desmos.com/calculator/twpnylu4lr

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
```

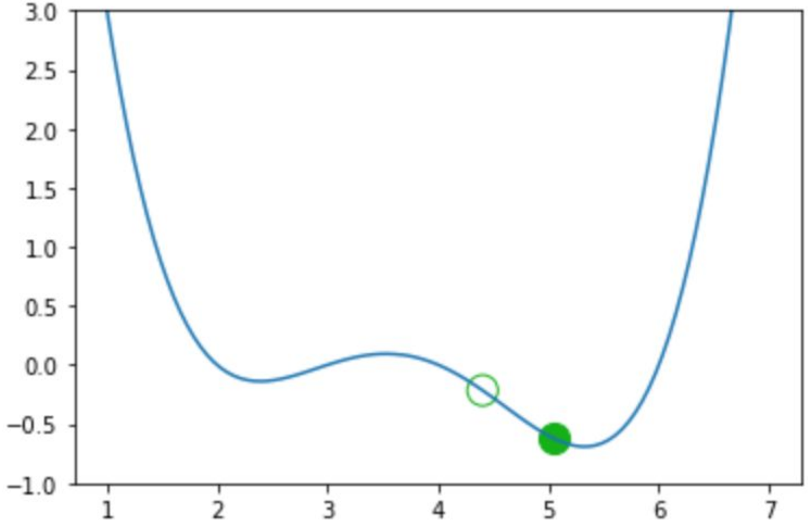


Manual Gradient Descent

```
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}')
```

```
plot_one_step(4.4)
```

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

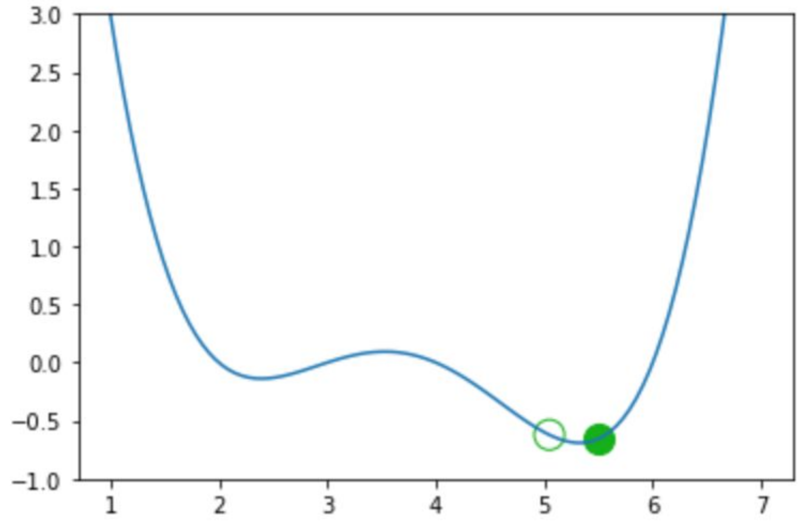


Manual Gradient Descent

```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
```

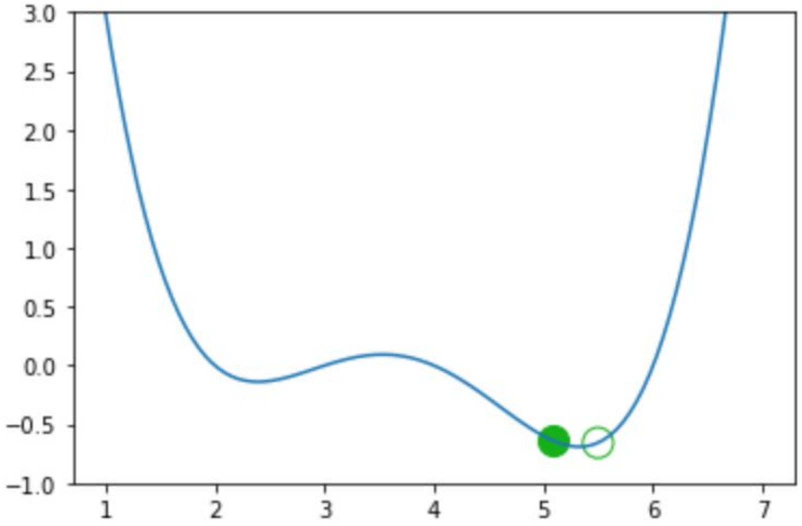


Manual Gradient Descent

```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
```

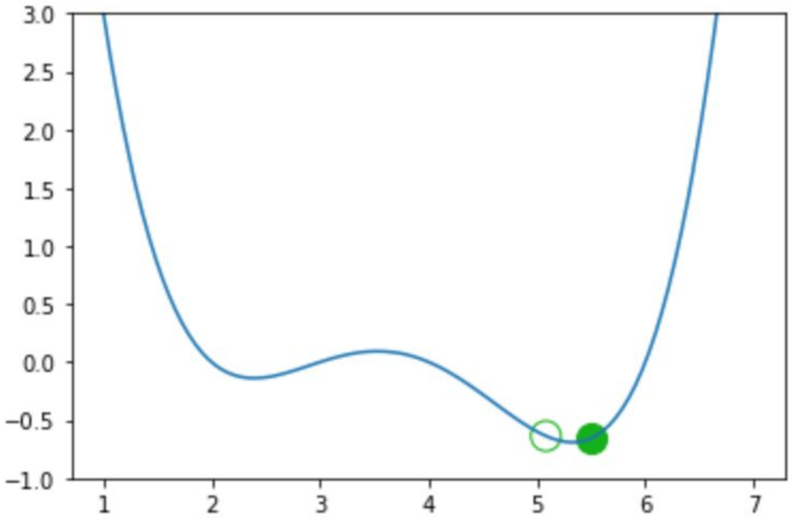


Manual Gradient Descent

```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
```

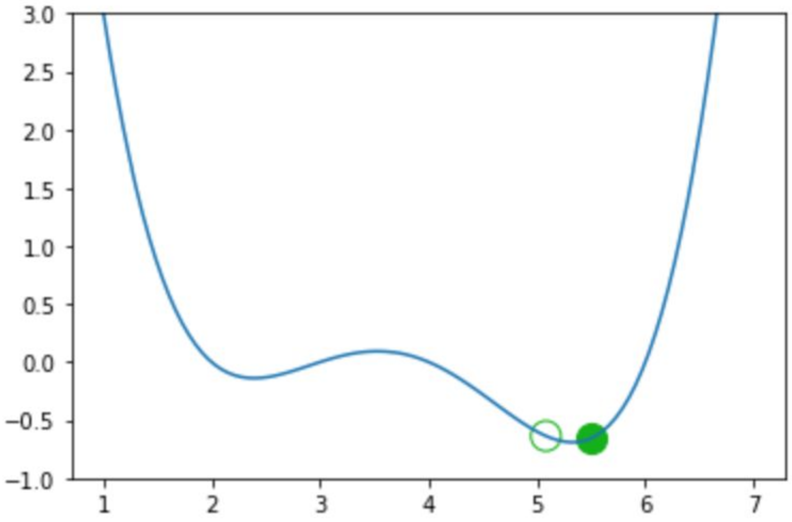


Manual Gradient Descent

```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}')
```

```
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!

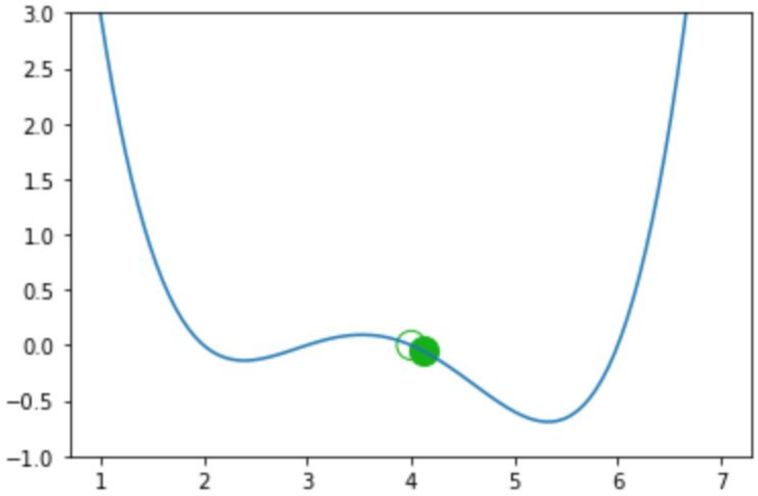
- Any suggestions for how we can avoid this issue?

Manual Gradient Descent with Slower “Learning Rate”

```
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(4)
```

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

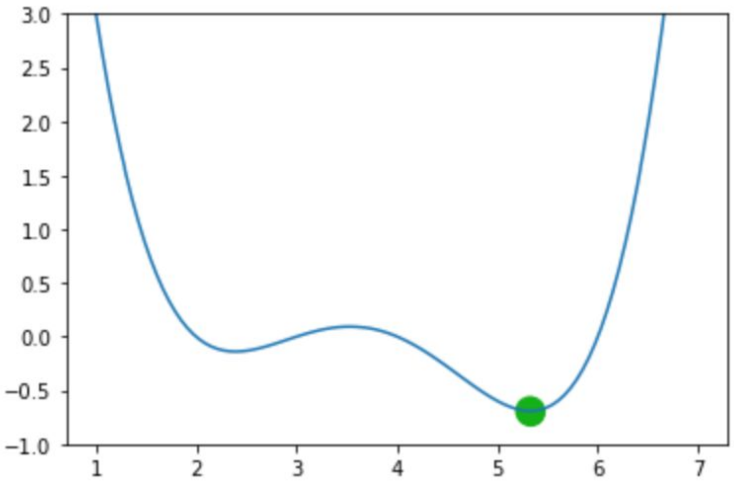


Manual Gradient Descent with Slower "Learning Rate" (many steps later)

```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(5.323)
```

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



3615537

Gradient Descent Implementation

Lecture 12, Data 100 Fall 2022

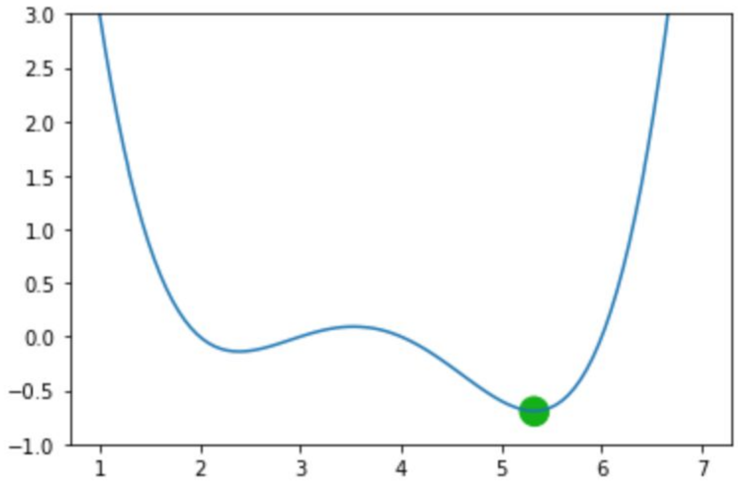
- Simple Linear Regression
- Multiple Linear Regression
- Minimizing an Arbitrary 1D Function
  - Gradient Descent Example
  - Gradient Descent Implementation
- Gradient Descent on a 1D Model
- Gradient Descent in Higher Dimensions

Gradient Descent as a Recurrence Relation

```
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
```



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

Learning rate "hyperparameter" that we choose

Our Recurrence Relation as Iterative Code

3615537

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

```
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

3615537

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

```
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, 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**.

slido

How many hours did you spend on Homework 05 (last week)?

ⓘ Start presenting to display the poll results on this slide.

Interlude

Announcements

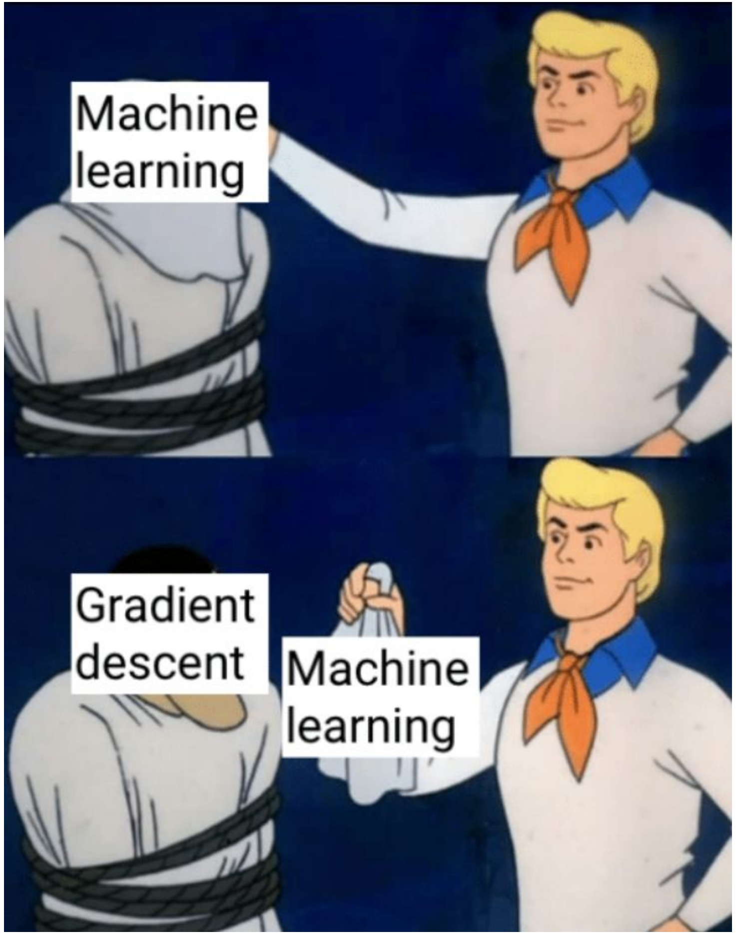
Midterm Review Session

- **Monday 3/6 2-4pm, VLSB 2060**
- in-person + recorded (no Zoom simulcast)
- Limited room capacity. Sign up here: EdStem

Midterm

- **Thursday 3/9 7-9pm, in-person**
- Rooms TBD
- More info here: EdStem
- Content: up through OLS
  (including Discussion 06, HW06, Lab06)
- All accommodations/alternates/left-hand desks:
  - Submit linked form by 3/1

Interlude



Gradient Descent on a 1D Model

Lecture 12, Data 100 Fall 2022

3615537

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

Applying Gradient Descent to Our Tips Dataset

3615537

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

- Using the derived equations from Data 8.
- Using **sklearn**.
    - **Uses gradient descent under the hood!**

In real practice in this course, we'll usually use sklearn. But for now, 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.

Terminology Clarification: Loss

3615537

We use the word "loss" in two different (but very related) contexts in this course.

- In general, loss is the cost function that measures how far off model's prediction(s) is(are) from the actual value(s).
    - **Per-datapoint loss** is a cost function that measures the cost of y vs $\hat{y}$ for a particular datapoint.
    - **Loss** (without any adjectives) is generally a cost function measured across all datapoints. Often times, **empirical risk** is average per-datapoint loss.
- **We prioritize using the latter term**, because we don't particularly look at a given datapoint's loss when optimizing a model.
    - In other words, the **dataset-level loss** is the **objective function** that we'd like to minimize using gradient descent.

Gradient Descent in Higher Dimensions

Lecture 12, Data 100 Fall 2022

3615537

A Two Parameter Model

3615537

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

$$\hat{y} = \theta_0 + \theta_1 \times \text{ tip}$$

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

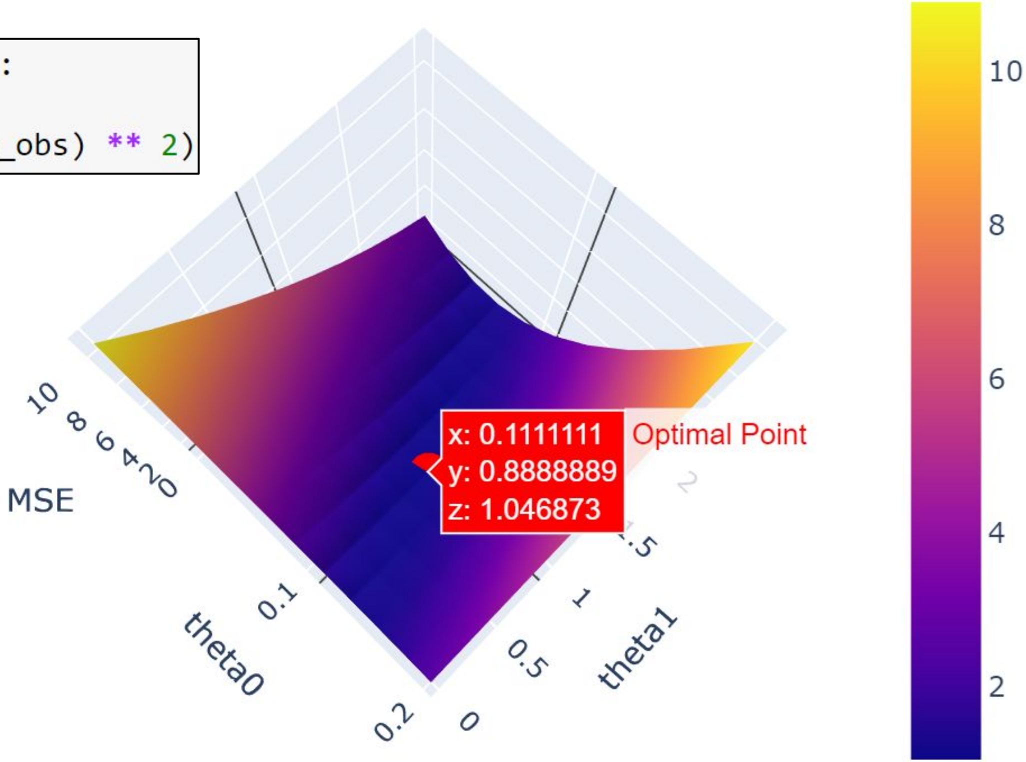
- Here, theta is a two dimensional vector!

```
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

3615537

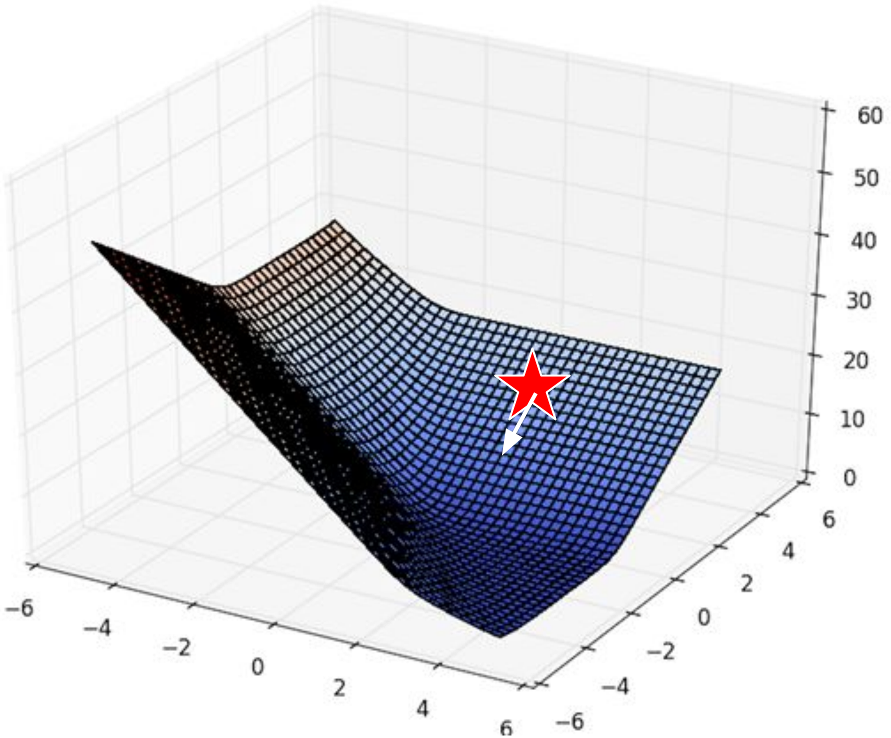
Just like earlier, we can follow the slope of our 2D function.

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

Approach #3: Gradient Descent

3615537

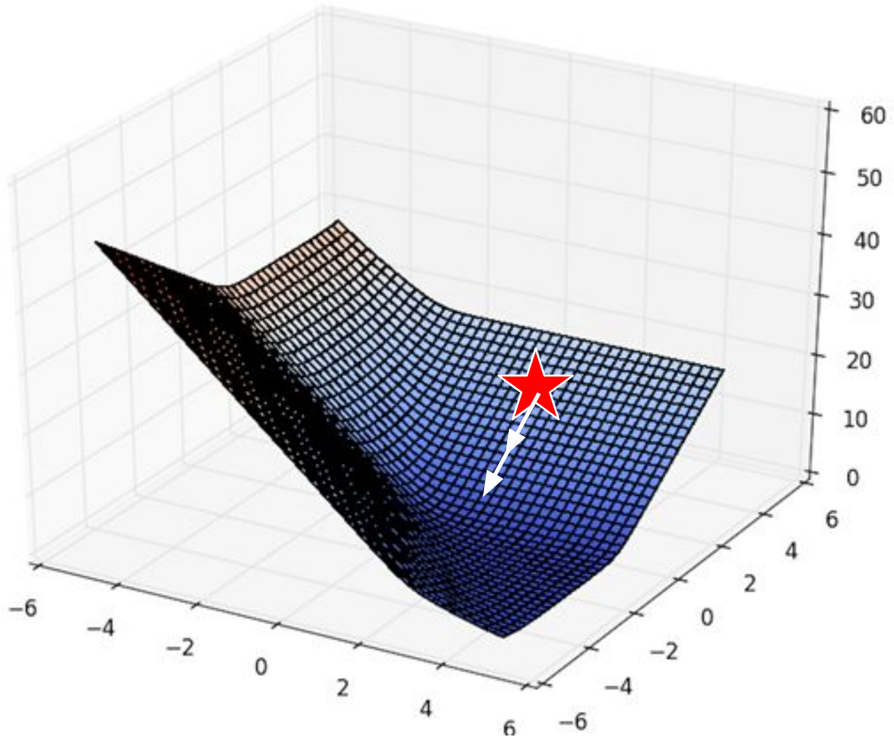
On a 2D surface, the best way to go down is described by a 2D vector.



Approach #3: Gradient Descent

3615537

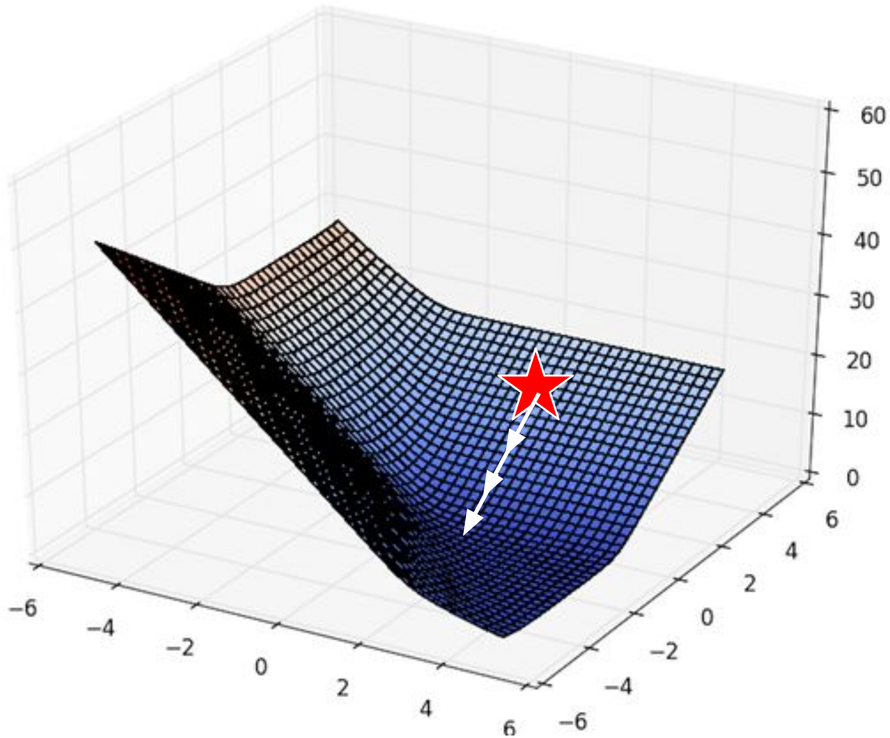
On a 2D surface, the best way to go down is described by a 2D vector.



Approach #3: Gradient Descent

3615537

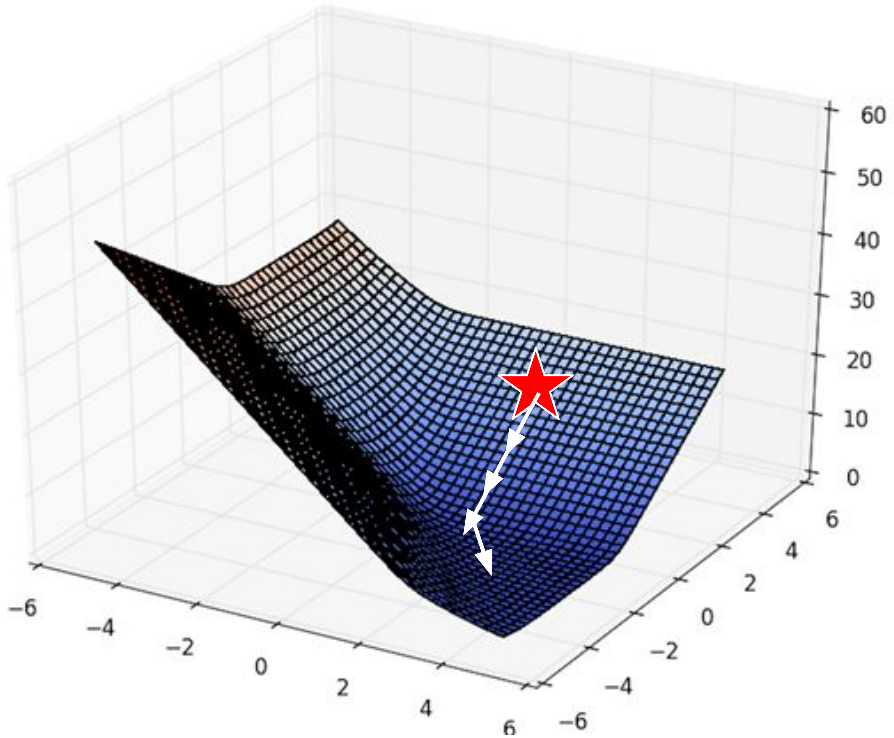
On a 2D surface, the best way to go down is described by a 2D vector.



Approach #3: Gradient Descent

3615537

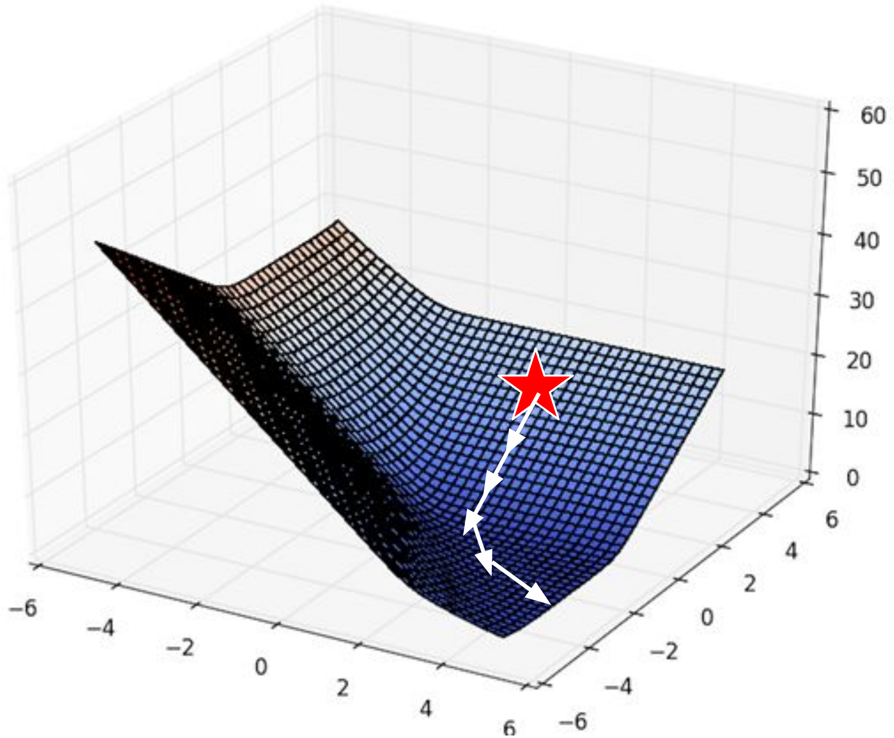
On a 2D surface, the best way to go down is described by a 2D vector.



Approach #3: Gradient Descent

3615537

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



Example: Gradient of a 2D Function

3615537

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

3615537

- 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:

3615537

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:

3615537

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

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

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

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

Summary

3615537

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 13

sklearn, Gradient Descent

Content credit: [Acknowledgments]