Gradient Descent for Risk Minimization, Logistic Regression ([Ch 11], [Ch 17.1-17.3])

UC Berkeley Data 100 Fall 2019
Josh Hug

(Slides adapted from Sam Lau, Sandrine Dudoit and Joey Gonzalez)

Plan for Today

Part 1:

- Reviewing 5 different ways we've learned to optimize a 1D linear regression model, including gradient descent from last lecture.
  - Summarizes every technique from the last 2 and a half weeks of lecture.
- Introduce multi-dimensional gradient descent and optimize a multiple linear regression model.

Part 2:

- Introduce the idea of logistic regression.
  - Model produces a probability that observation belongs to a category.
  - Very closely related to linear regression. Same tools can be used to optimize.

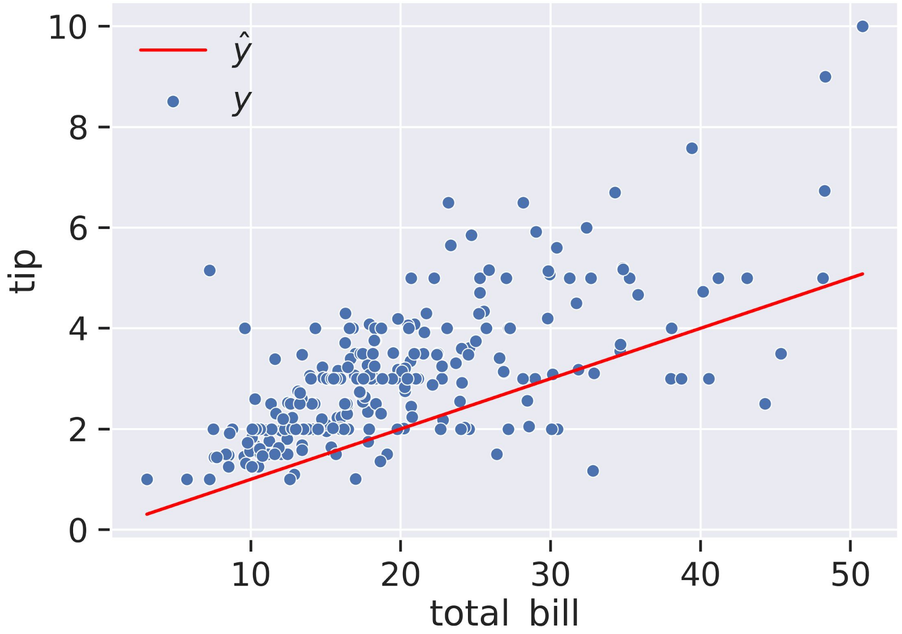
Optimizing Loss in 1D

Optimization Goal

Suppose we want to create a model that predicts the tip given the total bill for a table at a restaurant.

For this problem, we'll keep things simple and have only 1 parameter: gamma.

- $\hat{y} = f_{\hat{\gamma}(\vec{x})} = \hat{\gamma}\vec{x}$
- In other words, we are fitting a line with zero y-intercept.



See Notebook.

Optimization Goal

As discussed before, picking the best gamma is meaningless unless we pick:

- Loss function.
- Regularization term.

For this example, let's use the L2 loss and no regularization.

Solution Approach #1: Closed Form Solution

One approach is to use a closed form solution.

- On HW6 problem 7, you'll derive the closed form expression below:

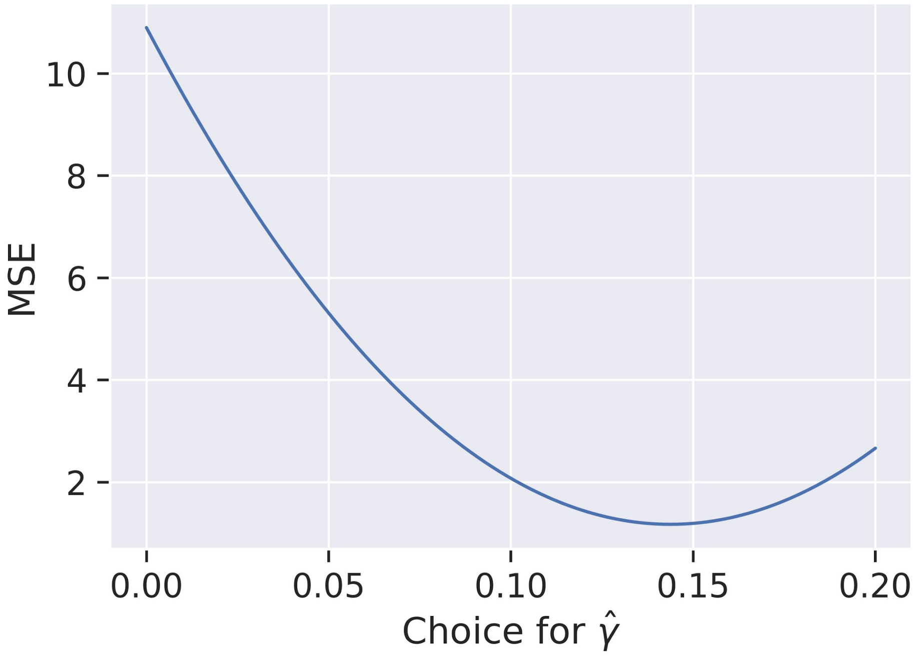
$$\hat{\gamma} = \frac{\sum x_i y_i}{\sum x_i^2}$$

Another closed form expression is just our standard normal equation:

$$\hat{\gamma} = (\mathbb{X}^T \mathbb{X})^{-1} \mathbb{X}^T \vec{y}$$

Solution Approach #2A: Brute Force Plotting

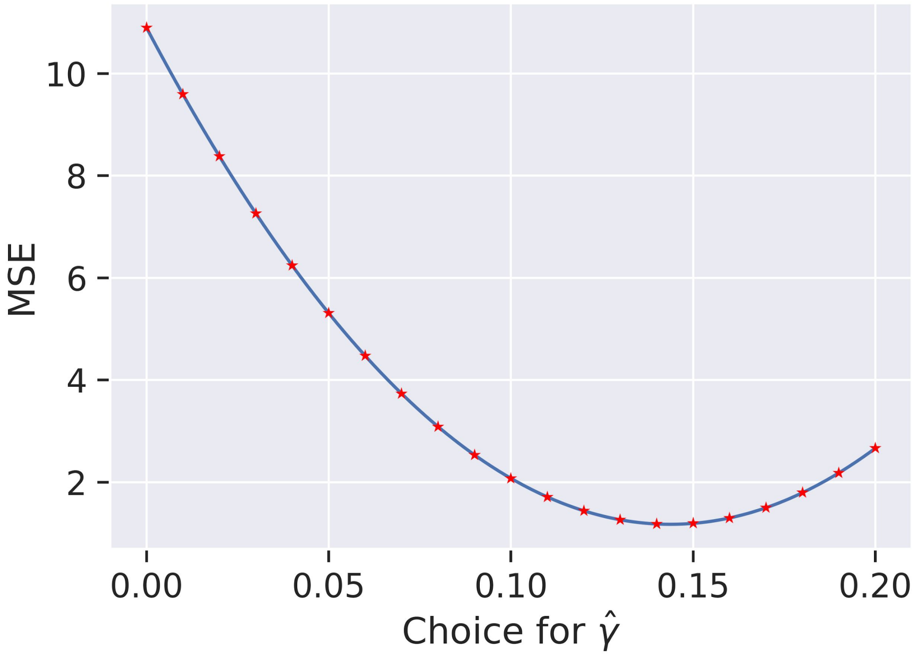
Another approach is to plot the loss and eyeball the minimum.




```
def mse_single_arg(gamma):
    """Returns the MSE on our data for the given gamma"""
    x = tips["total_bill"]
    y_obs = tips["tip"]
    y_hat = gamma * x
    return mse_loss(gamma, x, y_obs)
```

Solution Approach #2B: Brute Force

A related approach: Try a bunch of gammas and simply keep the best one.




```python
def mse_single_arg(gamma):
    """Returns the MSE on our data for the given gamma"""
    x = tips["total_bill"]
    y_obs = tips["tip"]
    y_hat = gamma * x
    return mse_loss(gamma, x, y_obs)
```

```python
def simple_minimize(f, xs):
    y = [f(x) for x in xs]
    return xs[np.argmin(y)]
```

```python
simple_minimize(mse_single_arg, np.linspace(0, 0.2, 21))
```

Solution Approach #3: Use Gradient Descent

We can use our gradient descent algorithm from before.

- To use this, we need to find the derivative of the function that we're trying to minimize.
- In the previous lecture, we minimized an arbitrary 4th degree polynomial.

$$\frac{d}{dx}$$

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

```
def df(x):
    return (4*x**3 - 45*x**2 + 160*x - 180)/10
```

Solution Approach #3: Use Gradient Descent

We can use our gradient descent algorithm from before.

- To use this, we need to find the derivative of the function that we're trying to minimize.
- In the previous lecture, we minimized an arbitrary 4th degree polynomial.

```
def df(x):
    return (4*x**3 - 45*x**2 + 160*x - 180)/10
```

```
def gradient_descent(df, initial_guess, alpha, n):
    guesses = [initial_guess]
    guess = initial_guess
    while len(guesses) < n:
        guess = guess - alpha * df(guess)
        guesses.append(guess)
    return np.array(guesses)
```

Solution Approach #3: Use Gradient Descent

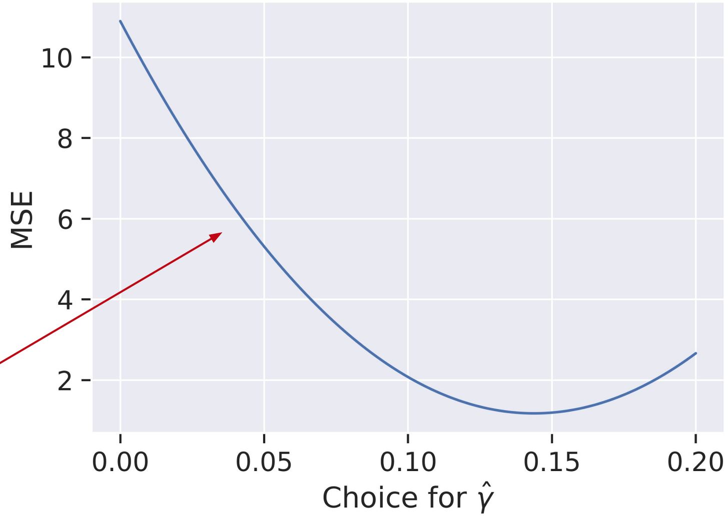
We can use our gradient descent algorithm from before.

- To use GD on our linear regression problem, we need to find the derivative of the function that we're trying to minimize, namely `mse_loss`.

$$\frac{d}{d\hat{\gamma}}$$

```
def mse_loss(gamma, x, y_obs):
    y_hat = gamma * x
    return np.mean((y_hat - y_obs) ** 2)
```

Need to compute the derivative of this curve.



Solution Approach #3: Use Gradient Descent

We can use our gradient descent algorithm from before.

- To use GD on our linear regression problem, we need to find the derivative of the function that we're trying to minimize, namely `mse_loss`.

x comes from the fact that $\hat{y} = x\hat{\gamma}$

$$\frac{d}{d\hat{\gamma}}$$

```python
def mse_loss(gamma, x, y_obs):
    y_hat = gamma * x
    return np.mean((y_hat - y_obs) ** 2)
```

```python
def mse_loss_derivative(gamma, x, y_obs):
    y_hat = gamma * x
    return np.mean(2 * (y_hat - y_obs) * x)
```

```python
def gradient_descent(df, initial_guess, alpha, n):
    guesses = [initial_guess]
    guess = initial_guess
    while len(guesses) < n:
        guess = guess - alpha * df(guess)
        guesses.append(guess)
    return np.array(guesses)
```

Solution Approach #4: scipy.optimize.minimize

As we've seen a few times in the class, we can use the scipy.optimize.minimize function to minimize our MSE function.

- No need to figure out the formula for the derivative.

```
import scipy.optimize
from scipy.optimize import minimize
minimize(mse_single_arg, x0 = 0)
```

```
def mse_single_arg(gamma):
    """Returns the MSE on our data for the given gamma"""
    x = tips["total_bill"]
    y_obs = tips["tip"]
    y_hat = gamma * x
    return mse_loss(gamma, x, y_obs)
```

Solution Approach #5: sklearn.linear_model.LinearRegression

We can also go above the level of abstraction of loss functions entirely and just use the LinearRegression model we saw in an earlier lab.

- Under the hood it's using the same MSE function and similar numerical techniques to compute gamma.

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

```
X = tips[["total_bill"]]
y = tips["tip"]
model.fit(X, y)
```

$\hat{\gamma}$

```
model.coef_
```

Why Use Gradient Descent (or other numerical solvers)?

- The beauty of GD is that it works for many types of models and loss function as long as you can take the gradient.
  - For example: If we add the L1 regularization term to our regression, gradient descent (or a similar algorithm) will be able to solve it.
  - Also, numerical methods often find solution faster than closed form analytic solution even if there is one (e.g. linear regression).
- Modeling recipe:
  - Pick model.
  - Pick loss function.
  - Pick regularization function.
  - Fit model by running gradient descent (or other solver).

Gradient Descent (in Multiple Dimensions)

Loss Minimization Game

From Fall 2018:

- https://tinyurl.com/3dloss18
- Try playing until you get the “You Win!” message.

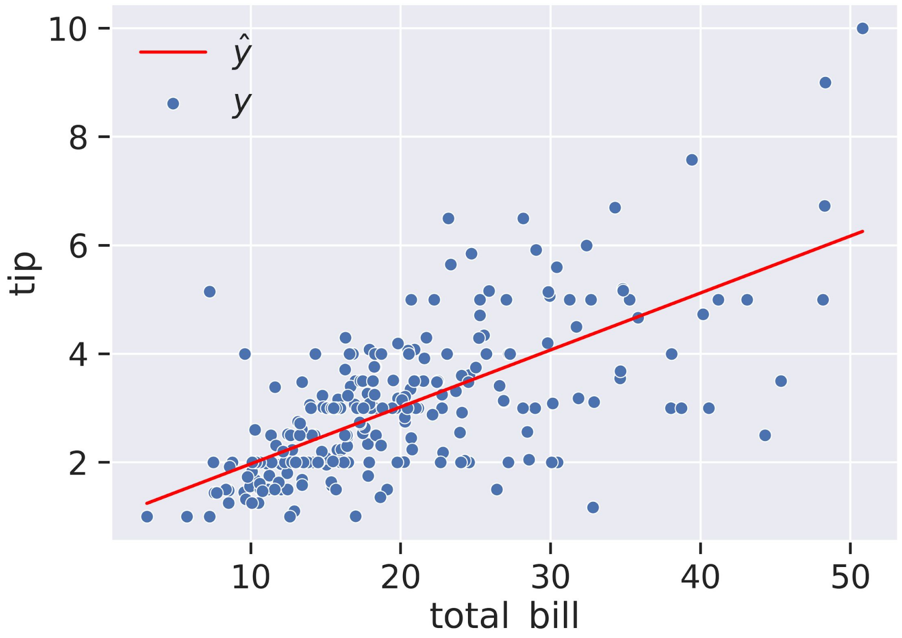
Optimization Goal

Now suppose we change our model so that it has two parameters θ_0 and θ_1.

- θ_0 is the y-intercept, and θ_1 is the slope.
- $$\text{tip} = \hat{\theta}_0 + \hat{\theta}_1 \text{bill}$$

$$\vec{\hat{y}} = f_{\vec{\hat{\theta}}}(\mathbb{X}) = \mathbb{X}\vec{\hat{\theta}}$$

$$\mathbb{X} = \begin{bmatrix} 1 & 16.99 \\ 1 & 10.34 \\ 1 & 21.01 \\ 1 & 23.68 \\ \vdots & \vdots \end{bmatrix}$$



Approach #1: Closed Form Solution

Since this is just a linear model, we can simply apply the normal equation.

$$\vec{\hat{y}} = f_{\vec{\hat{\theta}}}(\mathbb{X}) = \mathbb{X}\vec{\hat{\theta}} \qquad \vec{\hat{\theta}} = (\mathbb{X}^T\mathbb{X})^{-1}\mathbb{X}^T\vec{y}$$

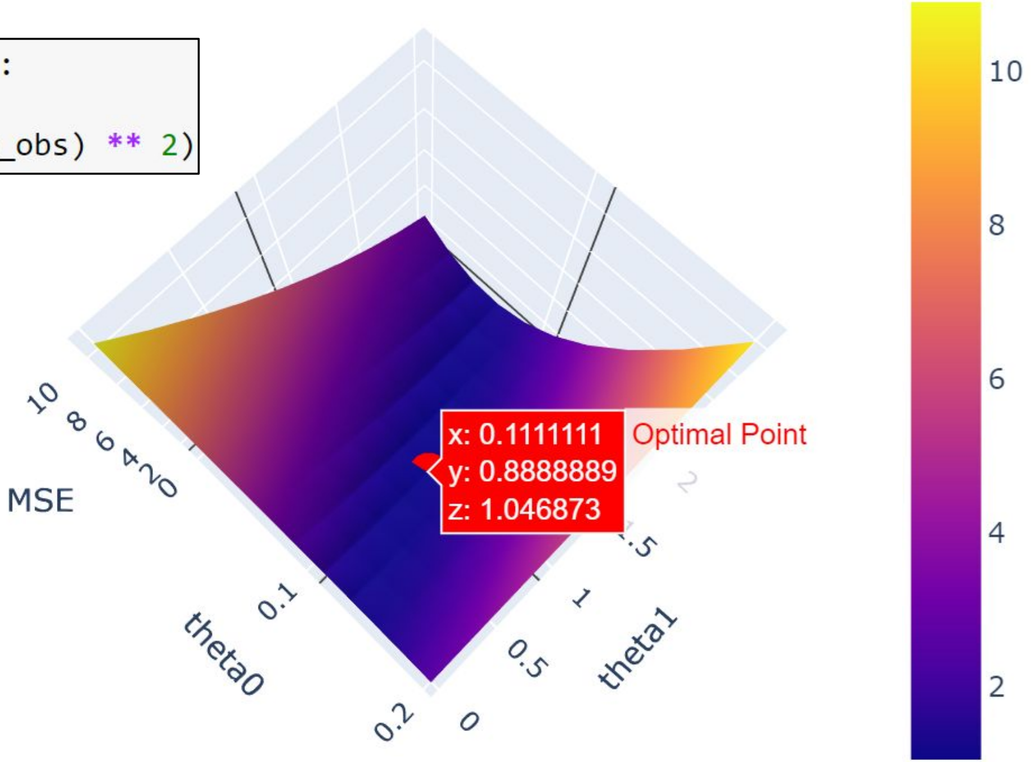
$$\mathbb{X} = \begin{bmatrix} 1 & 16.99 \\ 1 & 10.34 \\ 1 & 21.01 \\ 1 & 23.68 \\ \vdots & \vdots \end{bmatrix} \qquad \vec{y} = \begin{bmatrix} 1.01 \\ 1.66 \\ 3.50 \\ 3.31 \\ \vdots \end{bmatrix}$$

For reasons we won't discuss, when calculating the closed form equation above, it's generally better to use np.linalg.solve instead of np.linalg.inv.

Approach #2: Brute Force / Plotting

As before, we could just plot the 2D loss surface and find the minimum that way (plot is easy to understand in the notebook).

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



Solutions #4/#5: scipy.optimize.minimize / scipy.linear_model

As before, we can also use the scipy.optimize.minimize or scipy.linear_model libraries. Because it's exactly the same as before, we omit the exact details from this lecture.

Ultimately, both of these approaches use a numerical method similar to gradient descent.

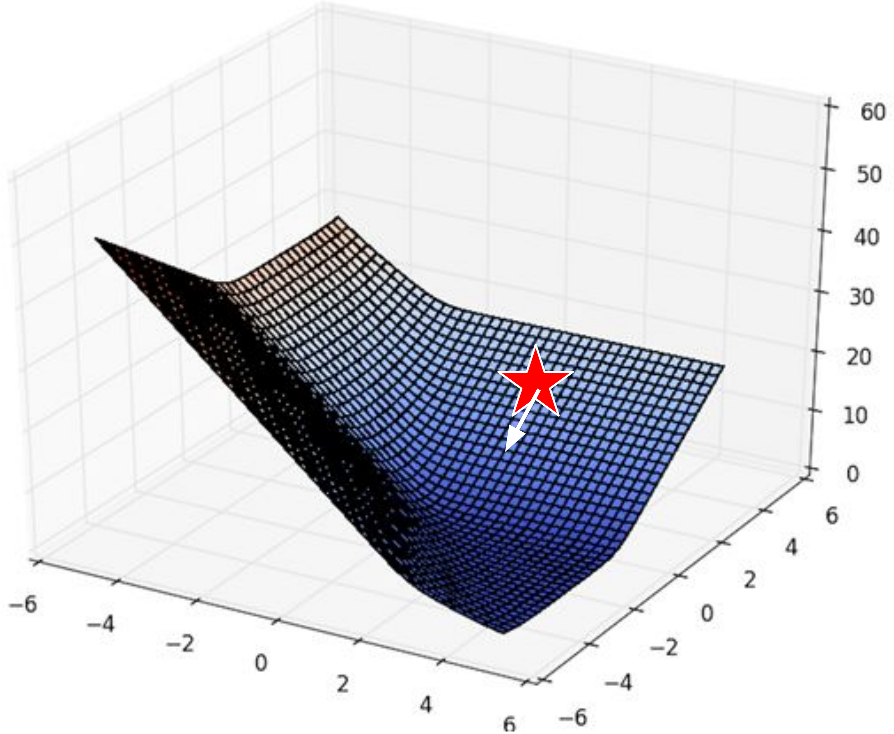
Approach #3: Gradient Descent

Another approach is to pick a starting point on our loss surface and follow the slope to the bottom.

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

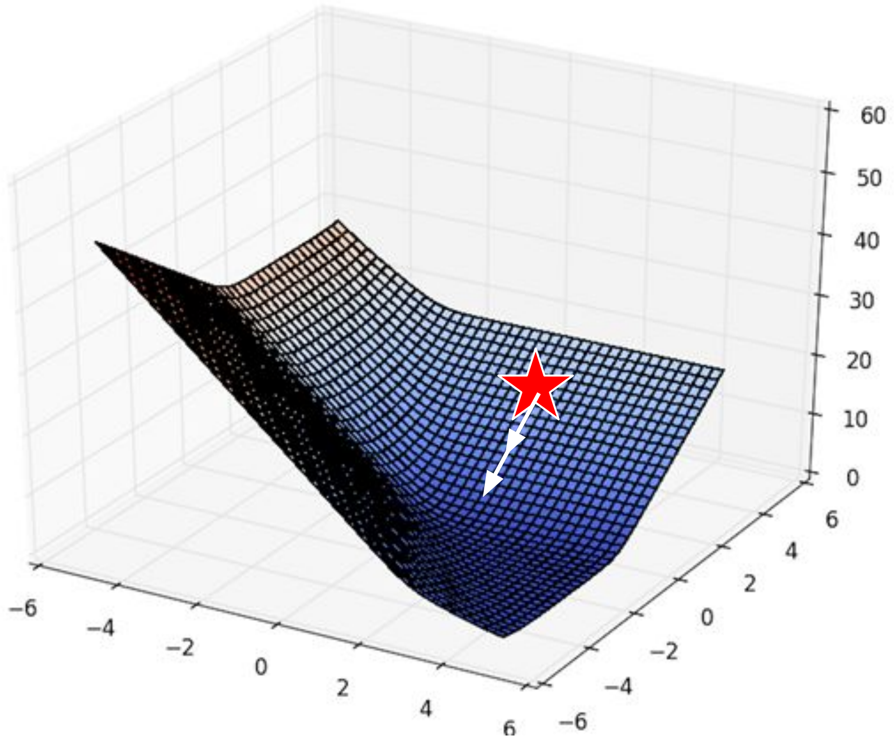
Approach #3: Gradient Descent

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



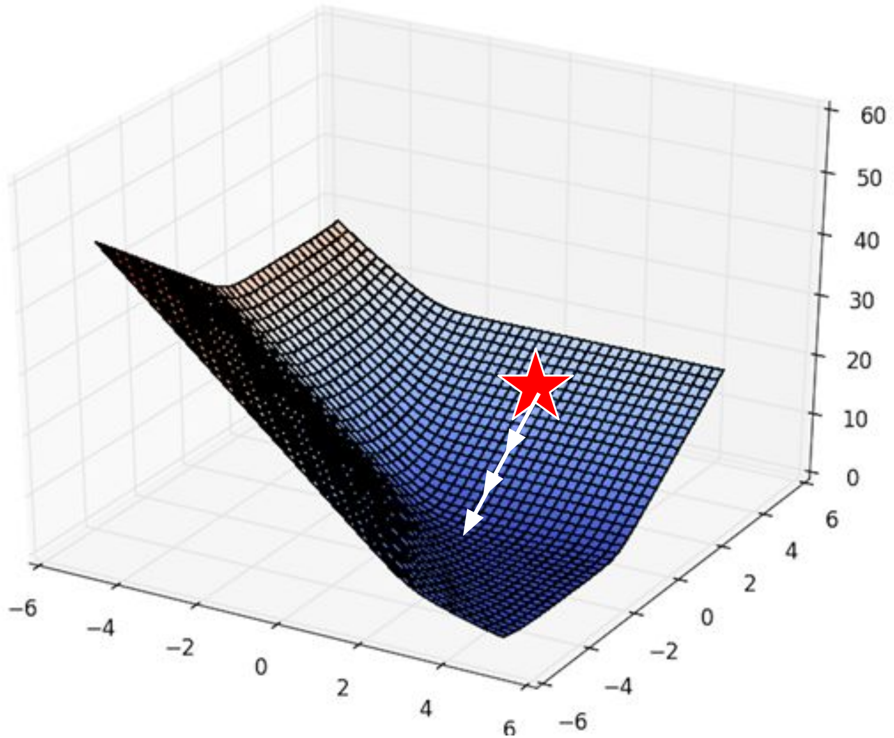
Approach #3: Gradient Descent

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



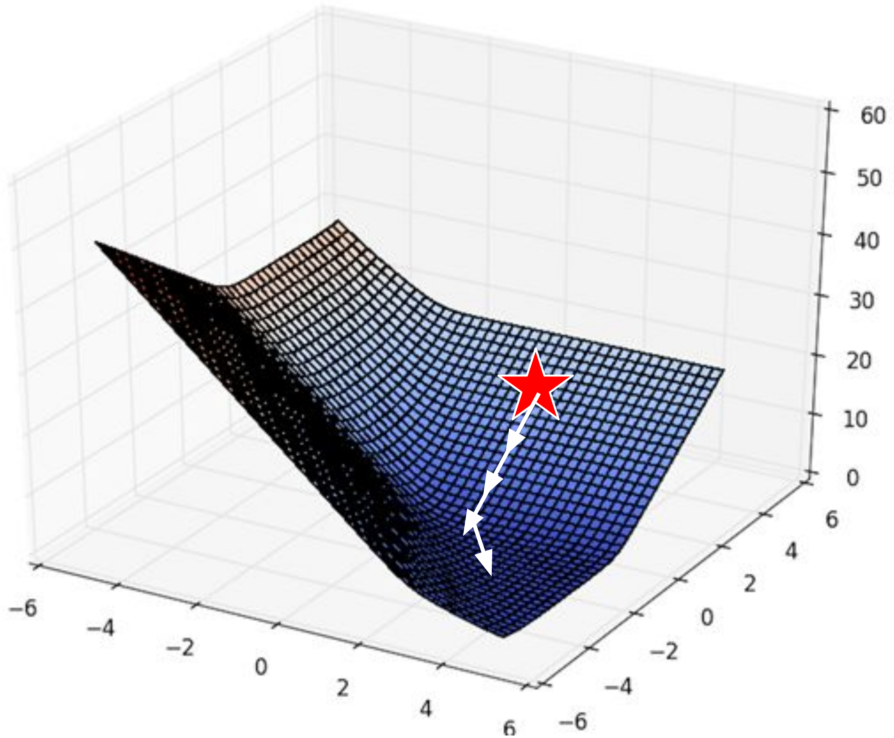
Approach #3: Gradient Descent

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



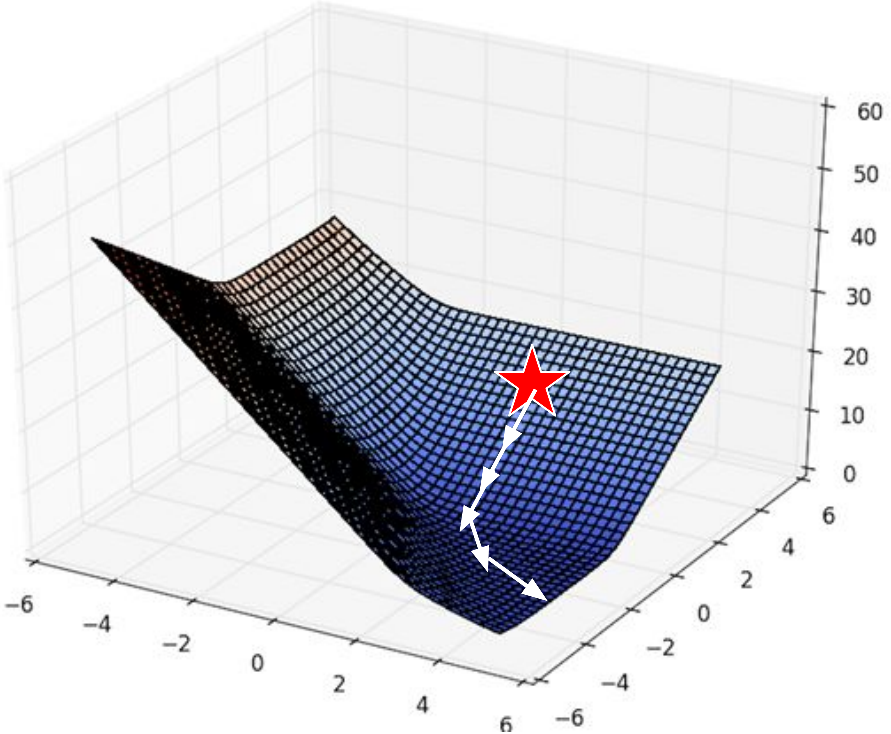
Approach #3: Gradient Descent

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_1 and θ_2 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.

This is similar to what you were doing when playing the loss game.

Batch Gradient Descent

- **Gradient descent** algorithm: nudge θ in negative gradient direction until θ converges.
- Batch gradient descent update rule:

Next value for θ

$$\vec{\theta}^{(t+1)} = \vec{\theta}^{(t)} - \alpha \nabla_{\vec{\theta}} L(\vec{\theta}, \mathbb{X}, \vec{y})$$

Gradient of loss wrt θ

Learning rate

θ: Model weights L: loss function
α: Learning rate, usually small constant
y: True values from training data

Gradient Descent Algorithm

- Initialize model weights to all zero
  - Also common: initialize using small random numbers
- Update model weights using update rule:

$$\vec{\theta}^{(t+1)} = \vec{\theta}^{(t)} - \alpha \nabla_{\vec{\theta}} L(\vec{\theta}, \mathbb{X}, \vec{y})$$

- Repeat until model weights don't change (convergence).
  - At this point, we have $\hat{\theta}$, our minimizing model weights

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 design matrix).

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

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

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

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

You Try:

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

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

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

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

The gradient for the entire dataset is the average of the gradients for each point, so we can run GD as-is.

You Try:

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

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

The gradient for the entire dataset is the average of the gradients for each point, so we use np.mean to compute that average.

```
def mse_gradient(theta, X, y_obs):
    """Returns the gradient of the MSE on our data for the given theta"""
    x0 = X.iloc[:, 0]
    x1 = X.iloc[:, 1]
    dth0 = np.mean(-2 * (y_obs - theta[0] * x0 - theta[1] * x1) * x0)
    dth1 = np.mean(-2 * (y_obs - theta[0] * x0 - theta[1] * x1) * x1)
    return np.array([dth0, dth1])
```

(demo)

You Try:

ℓ means loss for a single point;
L means average loss for dataset.

Logistic Regression

Logistic Regression

For this part of lecture, we'll primarily work from our notebook.

Our goal will be to provide an intuitive picture of how logistic regression can be used for classification.

https://tinyurl.com/badbluefish

Linear vs. Logistic Regression

In a **linear regression** model with 1 feature, our goal is to predict a **quantitative** variable (i.e., some real number) from that feature.

$$\hat{y} = f_{\hat{\beta}}(x) = x\hat{\beta}$$

- Our output can be **any real number**.

In a **logistic regression** model with 1 feature, our goal is to predict a **categorical** variable from that feature.

$$\hat{y} = f_{\hat{\beta}}(x) = P(Y = 1|x) = \sigma(x\hat{\beta})$$

- The output of **logistic regression** is always between 0 and 1, i.e. it is **quantitative**!
  - Gives probability under our model that the category is 1.
- Our goal is to perform **binary classification –** to predict either 0 or 1.
  - How do we actually classify, then? We will find out today!

Linear vs. Logistic Regression

In a **linear regression** model with p features, our goal is to predict a **quantitative** variable (i.e., some real number) from those features.

$$\hat{y} = f_{\hat{\vec{\beta}}}(\vec{x}) = \vec{x}^T \hat{\vec{\beta}}$$

- Our output can be **any real number**.

In a **logistic regression** model with p features, our goal is to predict a **categorical** variable from those features.

$$\hat{y} = f_{\hat{\vec{\beta}}}(\vec{x}) = P(Y = 1|\vec{x}) = \sigma(\vec{x}^T \hat{\vec{\beta}})$$

- The output of **logistic regression** is always between 0 and 1, i.e. it is **quantitative**!
  - Gives probability under our model that the category is 1.
- Our goal is to perform **binary classification –** to predict either 0 or 1.
  - How do we actually classify, then? We will find out today!

Remember, $\vec{x}^T \hat{\vec{\beta}} = x_1\hat{\beta}_1 + x_2\hat{\beta}_2 + ... + x_p\hat{\beta}_p$

The Logistic Regression and the Logistic Function

In logistic regression, we're modelling the **probability** that an observation belongs to class 1 (as opposed to class 0).

$$\hat{y} = P(Y = 1|\vec{x}) = \sigma(\vec{x}^T\vec{\hat{\beta}})$$

Our model looks like the linear regression model, except with a $\sigma(\cdot)$ around it.

$$\sigma(t) = \frac{1}{1 + e^{-t}}$$