Introduction to Python & Machine Learning

2025 Workshop in AI

Ramiro Ramirez, PhD

UT Health Science Center San Antonio

July 27, 2025

Overview

Overview

Why Fundamentals Matter

LLM *agentic* frameworks (e.g., Claude Code) can greatly accelerate software development. But they work best when you understand the underlying concepts. In this workshop we will pair tools with fundamentals:

- **Short intensive** on Python and machine learning basics (Which is what this part will consist of).
- **Hands-on Projects** where we build and run code, then use agentic tools to refactor/extend it.
- **Goal**: leave with durable mental models *and* a practical workflow with LLM tools.

Python — Learning Outcomes

By the end of the python section, you will be able to:

- Read and write idiomatic Python using core **data types**, **modules/imports**, **functions** (with docstrings and type hints), and **classes**.
- Apply **inheritance**, **polymorphism**, and **abstract base classes (ABC)** to model simple domains.
- Explain what *design patterns* are and sketch the **Backend-for-Frontend (BFF)** design pattern and build a tiny BFF service.

Machine Learning — Learning Outcomes

By the end of the machine learning section, you will be able to:

- Derive and implement **mean squared error (MSE)** as a loss/objective for regression.
- Implement **gradient descent** from scratch and reason about learning rate and convergence.
- Build a **linear regression** model end-to-end (data prep $\rightarrow$ train $\rightarrow$ evaluate) and plot a loss curve.

Overview

Primitive vs Derived

Python Data Types are the different data programmers can work with.

Definition (Informal)

Primitive types (informal). Data types that are built into the interpreter and don't (at the Python level) contain other user-visible objects.
Derived types (informal). Constructed from primitives data types.

Examples of Primitive Data Types:

- `int` – integers (e.g., `42`, `-17`)
- `float` – floating-point numbers (e.g., `3.14`, `-0.5`)
- `bool` – boolean values (`True`, `False`)
- `str` – strings (e.g., `"Hello, World!"`)
- `bytes` – byte sequences
- `list`, `tuple`, `dict`, `set` – built-in collections

Examples of Derived Data Types:

- Pandas DataFrames – tabular data structures
- NumPy arrays – multi-dimensional arrays for numerical computing

Jupyter Coding Lesson: Data Types

Lets go into the lesson

Flow Control

Key Constructs

Conditionals

Make decisions

- `if`: execute code when condition is true
- `elif`: check additional conditions
- `else`: fallback when all conditions fail

Loops

Repeat actions

- `for`: iterate over collections/sequences
- `while`: repeat until condition becomes false

Loop Control

Fine-tune execution

- `break`: exit loop immediately
- `continue`: skip to next iteration

Jupyter Coding Lesson: Data Types

Lets go into the lesson

- **What is a Class?** A blueprint or template that defines attributes (data) and methods (behavior).
- **What is an Object?** A concrete instance of a class with its own state.
- **Four Pillars of OOP:**
  - **Encapsulation** — bundle data methods, hide internals
  - **Abstraction** — expose only essential features
  - **Inheritance** — reuse and extend existing classes
  - **Polymorphism** — use a common interface for different types

OOP Pillar 1: Encapsulation

Bundle data and methods together, hide internal details

What is Encapsulation?

- Bundle related data and methods into a single unit (class)
- Hide internal implementation details from outside code
- Control access through public interfaces
- Protect data integrity with validation

Real-World Example: Bank Account

Public Interface:

- `deposit(amount)`
- `withdraw(amount)`
- `getBalance()`

Hidden Details:

- Private `balance` field
- Validation logic
- Transaction logging
- Security checks

OOP Pillar 2: Inheritance

Create new classes based on existing ones

What is Inheritance?

- Create new classes that extend existing classes
- Child classes automatically get parent's properties and methods
- Can override or extend parent behavior
- Establishes "IS-A" relationships

Real-World Example: Vehicle Hierarchy

Base Class: Vehicle

- Common properties: wheels, engine, fuel
- Common methods: `start()`, `stop()`, `refuel()`

Child Classes:

- **Car:** inherits Vehicle + adds doors, passengers
- **Truck:** inherits Vehicle + adds cargo capacity

OOP Pillar 3: Polymorphism

Same interface, different implementations

What is Polymorphism?

- Objects of different types respond to the same method calls
- Same interface, different behaviors underneath
- Runtime determination of which method to call
- Enables "one interface, many implementations"

Real-World Example: Drawing Shapes

Common Interface:

- All shapes have `draw()` method
- All shapes have `calculateArea()` method

Different Implementations:

- **Circle.draw():** draws a circle
- **Square.draw():** draws a square

OOP Pillar 4: Abstraction

Focus on essential features, hide complexity

What is Abstraction?

- Focus on what an object does, not how it does it
- Hide complex implementation details
- Provide simplified interfaces for complex systems
- Model real-world concepts at appropriate level

Real-World Example: Database Interface

Simple Interface:

- `save(user)`
- `findById(id)`
- `delete(user)`

Hidden Complexity:

- SQL query generation
- Connection management
- Error handling and retries
- Data validation and sanitization

How the Four Pillars Work Together

The foundation for design patterns

Synergy of OOP Principles

The four pillars work together to create robust, maintainable software:

- **Encapsulation + Abstraction:** Create clean, safe interfaces
- **Inheritance + Polymorphism:** Enable code reuse with flexibility
- **All Four Together:** Support complex design patterns and architectures

Next Steps

Now that we understand OOP foundations, we can explore how design patterns leverage these principles to solve common software design challenges.

Coding Lesson 3: OOP Principles

Lets go into the lesson

OOP Pillar 2: Inheritance

Create new classes based on existing ones

What is Inheritance?

- A subclass automatically gains the attributes and methods of its parent (super) class
- Establishes an *"IS-A"* relationship (e.g., `Car` **is a** `Vehicle`)
- Promotes code reuse and logical organization of related types

Key Benefits

- **Code reuse:** shared logic in one place
- **Consistency:** uniform interfaces across subclasses
- **Maintainability:** update base behavior without touching

Real-World Example: Vehicle Hierarchy

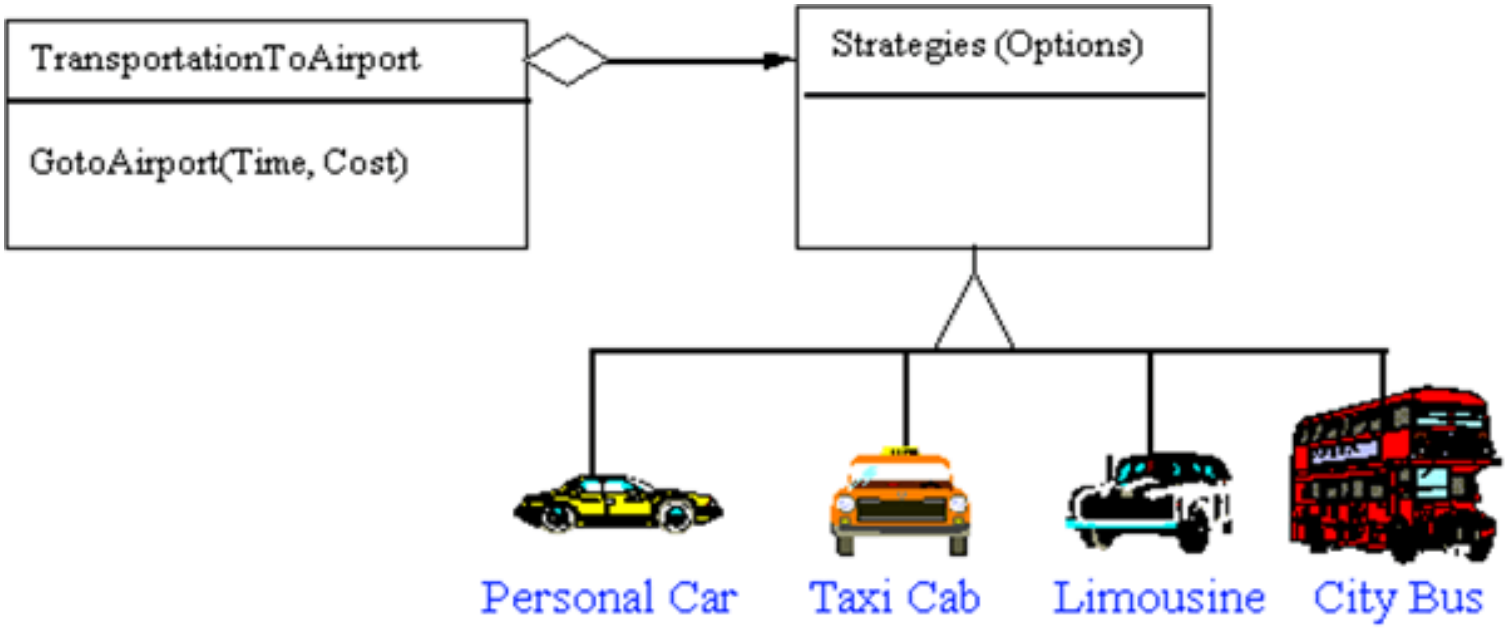
Base Class: `Vehicle`

- `wheels`, `engine`, `fuel`
- `start()`, `stop()`, `refuel()`

Subclasses: `Car`, `Truck`, `Motorcycle` $\Longrightarrow$ each adds its

Coding Lesson 3: Inheritence

Lets go into the lesson



Introduction to Architecture Design

Building maintainable and scalable software

Writing code isn't just about knowing syntax, actually the one of the most important parts is actually the design of the software/code. It's about creating **robust**, **maintainable**, and **scalable** systems.

- Good design helps you avoid duplication, manage complexity, and speed up future development.
- *Design patterns* capture proven, generalizable solutions to common problems in software architecture.

Definition (Design Pattern)

A design pattern is a *reusable* template for solving a specific kind of design problem in software. It describes the problem, the solution structure, and the consequences of applying that solution.

Observer Pattern: Real-World Origin

Making "publish–subscribe" simple

Emerged in Smalltalk-80's MVC at Xerox PARC (late 1970s) Needed a way for data models to "broadcast" changes to any number of views Formalized as the "Observer" pattern in the GoF book (1994)

Problem

- A teacher posts a new assignment on the online course portal
- Multiple systems student dashboard, parent email alerts, mobile app must update automatically

Introduction to Design Patterns

Common patterns by category with real-world origins

Creational Patterns

- **Factory Method:** create objects without specifying exact class
- **Abstract Factory:** families of related objects
- **Singleton:** single shared instance
- **Builder:** step-by-step object construction

Structural Patterns

- **Adapter:** make incompatible interfaces work together
- **Decorator:** add behavior dynamically
- **Facade:** simplify complex subsystem
- **Composite:** treat objects & compositions uniformly

Behavioral Patterns

- **Observer:** publish/subscribe event model
- **Strategy:** interchangeable algorithms
- **Command:** encapsulate requests as objects
- **Template Method:** define algorithm skeleton

Strategy Pattern: Intent

Open–Closed Principle (OCP) Alignment

Intent

Define a family of behaviors (algorithms), encapsulate each one, and make them interchangeable at runtime.

- Behaviors belong in their own classes, not in subclasses of the context.
- Context classes stay **open for extension** but **closed for modification**.

Why Not Just Inheritance?

The problems with traditional inheritance-based design

Scenario: A `Car` class with `brake()` and `accelerate()` methods

Inheritance Implementation:

- `FerrariModel` extends `Car`
  - `brake()` → "ABS sport brake"
  - `accelerate()` → "Race-tuned throttle"
- `FordModel` extends `Car`
  - `brake()` → "Standard hydraulic brake"
  - `accelerate()` → "Economy-mode throttle"

Problems with this approach:

- **Code duplication:** every new model reimplements methods
- **Rigid hierarchy:** can't mix sports brake with economy throttle
- **Poor maintainability:** updating brake logic requires editing all subclasses

The Solution

Strategy Pattern: Compose behaviors instead of inheriting them!

Composition Over Inheritance

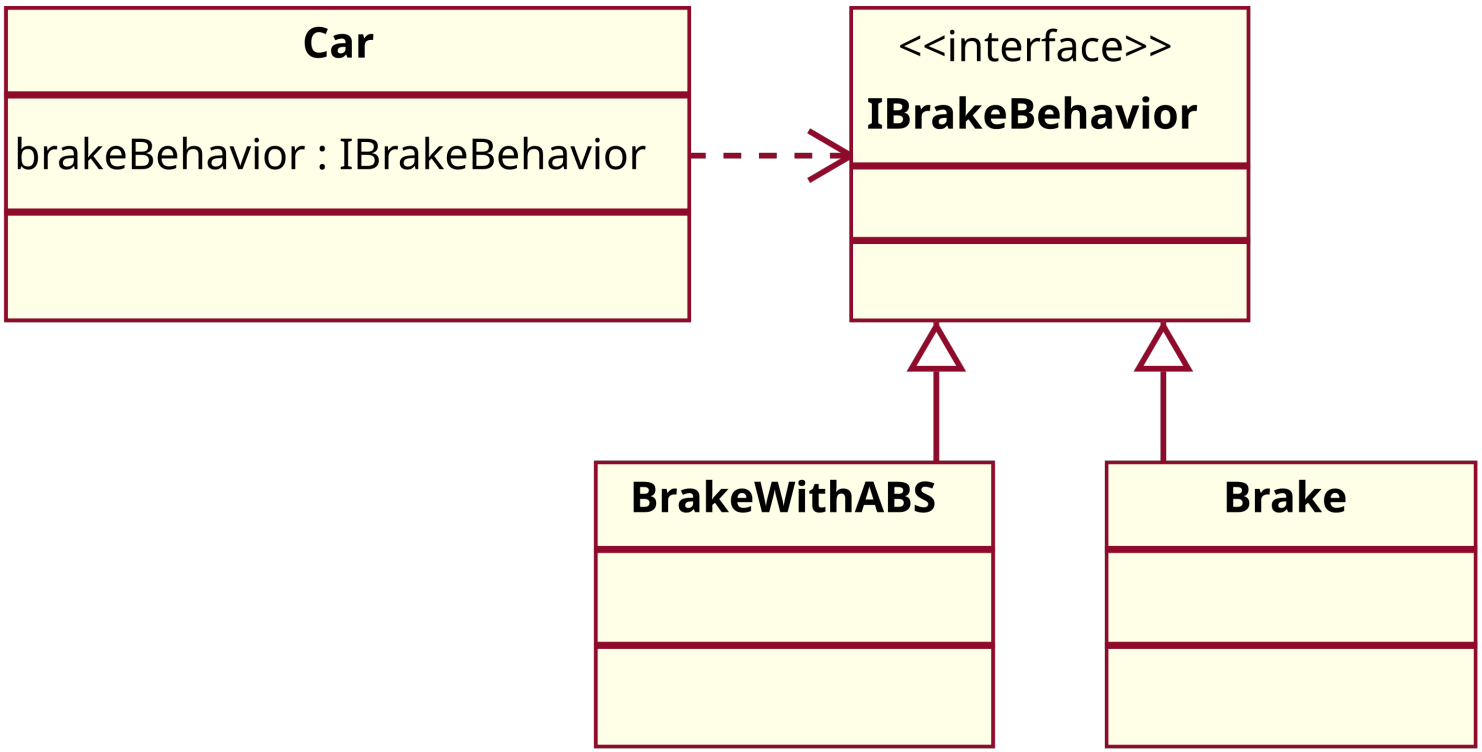
How Strategy Pattern solves the problem

What is Composition?

Inheritance: "A Ferrari **IS-A** Car" (Ferrari extends Car)
Composition: "A Car **HAS-A** brake behavior" (Car contains BrakeBehavior)

Strategy Pattern in Action

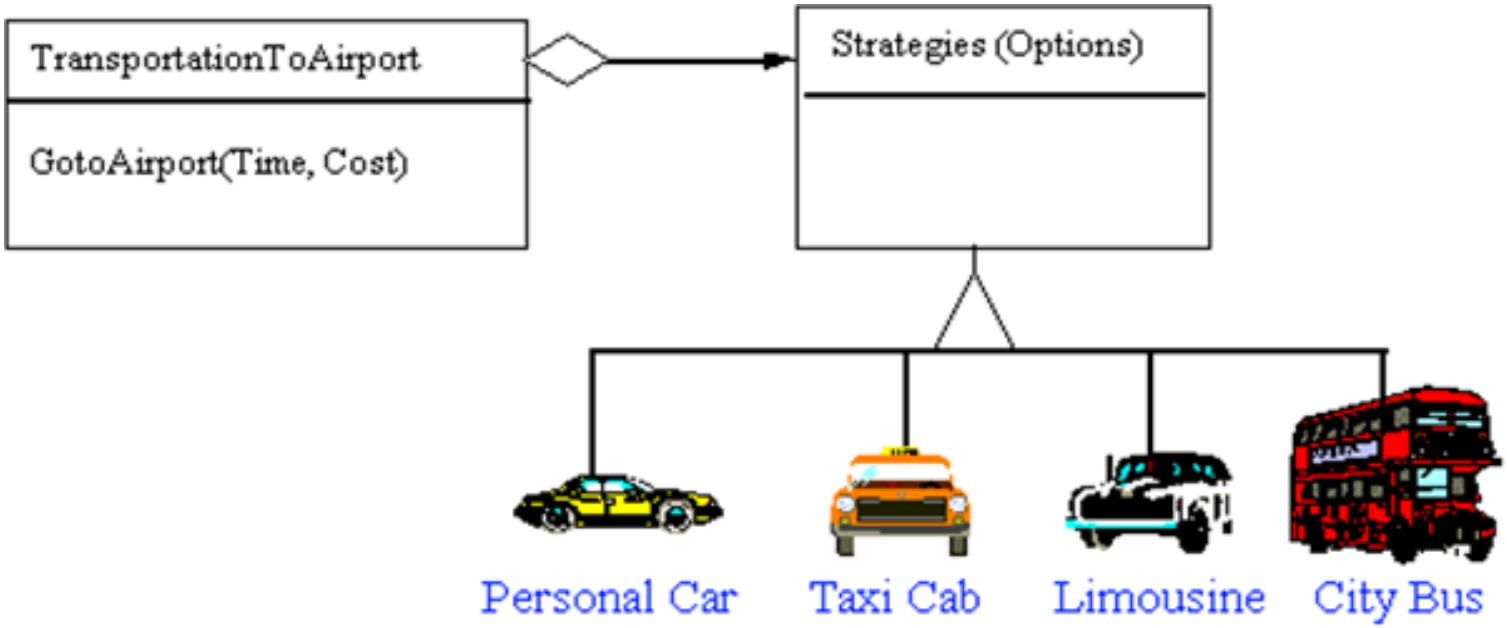
Car Behavior Composition



Note: The `Car` "has a" `BrakeBehavior` and "has a" `AccelerateBehavior` rather than "is a" subclass of each.

Coding Lesson 4: Design Patterns

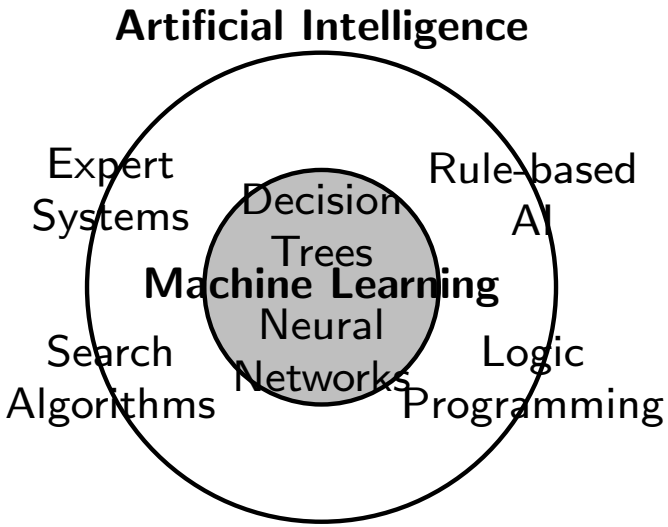
Lets go into the lesson



Overview

AI vs Machine Learning

- In modern times, *AI* is often used to mean **generative AI** or **LLMs** (Large Language Models). Historically, however, AI encompasses a broader set of methods for creating *intelligent* systems.
- **Machine Learning** is a subset of AI focused on algorithms and statistical models that let computers learn from data without being explicitly programmed for each task.
- **Traditional AI** includes *expert systems*, *rule-based reasoning*, and *search algorithms*



Machine Learning Paradigms: Overview

Machine learning tasks can be split into three main paradigms:

- **Supervised Learning:** Learn a mapping from inputs to known outputs using fully labeled data.
- **Semi-Supervised Learning:** Leverage a small labeled set plus a large unlabeled set to improve model accuracy.
- **Unsupervised Learning:** Discover hidden patterns or structures in entirely unlabeled data.

Question: *What paradigm do you think modern LLMs (e.g. GPT-4) are primarily based on, and why?*

Notational Conventions

Key symbols and their meanings

Symbol	Meaning
x or $\mathbf{x}$	input features (scalar or vector)
y	true label or target value
$\hat{y}$	model's prediction for y
X	feature matrix (stack of all $\mathbf{x}_i$ in dataset)
θ	parameter vector of the model
$\mathcal{L}(y, \hat{y})$ or $\mathcal{L}(\theta)$	loss function measuring error on a single example
α	learning rate (step size in gradient descent)
$\nabla_\theta J(\theta)$	gradient of the cost w.r.t. θ
$f_\theta : X \to \mathbb{R}^n$	Our machine learning model with learned parameters θ

Cost Function

$$J(\theta) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(y_i, \hat{y}_i) \tag{1}$$

Supervised Learning: Learning from Labeled Data

Many real-world tasks come with historical examples where each input has a correct label. We train models on these examples so they can predict labels for new, unseen data.

Definition

Supervised Learning is a paradigm where we learn a function $f_\theta : X \to \mathbb{R}^n$ from labeled examples $\{(x_i, y_i)\}$ to predict y for new inputs x.

Training Data: $\{(x_i, y_i)\}_{i=1}^n$

- x_i: input features (e.g., email text, house attributes, gene expression value)
- y_i: target label (e.g., spam/ham, sale price, tumor/no tumor)

Goal: Find f such that $f(x) \approx y$ on new data

Supervised Learning: Tasks Evaluation

Common Tasks:

- *Classification:* assign inputs to discrete categories (e.g., spam detection, image recognition)
- *Regression:* predict continuous values (e.g., house-price prediction, demand forecasting)

Evaluation Metrics:

- *Classification:* Accuracy, Precision, Recall, F1 score
- *Regression:* Mean Squared Error (MSE), R^2

Question: *Can you think of a real-world example where a regression model would be more appropriate than classification?*

Unsupervised Learning: Exploring Unlabeled Data

Finding Patterns Without Labels

Many datasets lack labels, yet contain rich structure waiting to be discovered.

Definition

Unsupervised Learning is a paradigm where we learn to identify patterns, structures, or representations solely from unlabeled data $\{x_i\}$.

Training Data: $\{x_i\}_{i=1}^n$ (no labels)

Goal: Reveal hidden groupings or compress data into informative representations.

Unsupervised Learning: Methods & Engagement

Some Techniques::

- **Clustering:** group similar points (e.g., k-means, DBSCAN)
- **Dimensionality Reduction:** project data into fewer dimensions while preserving structure (e.g., PCA, t-SNE)
- **Anomaly Detection:** identify outliers (e.g., One-Class SVM, Isolation Forest)

Challenges:

- *Choosing hyperparameters:* selecting number of clusters or embedding dimension
- *Scalability:* processing large datasets efficiently
- *Interpretability:* making sense of discovered patterns

Question:

What real-world problems would make labeling data impractical?

Semi-Supervised Learning: Bridging Labeled and Unlabeled Data

In many real-world domains, obtaining labels is costly or time-consuming, but unlabeled data is abundant. Semi-supervised learning sits between supervised and unsupervised paradigms to exploit both.

Definition

Semi-Supervised Learning is a paradigm where we learn from a small labeled dataset $\{(x_i, y_i)\}$ together with a large unlabeled dataset $\{x_j\}$ to improve predictive performance.

Training Data: $\underbrace{\{(x_i, y_i)\}_{i=1}^{m}}_{\text{few labels}} \bigcup \underbrace{\{x_j\}_{j=1}^{n}}_{\text{many unlabeled}}$

Goal: Use the structure in $\{x_j\}$ to guide learning of $f : X \to Y$ and boost accuracy over purely supervised approaches.

Semi-Supervised Learning: Methods & Engagement

Some Techniques:

- **Self-Training:** Train a model on labeled data, label the most confident unlabeled examples, then retrain.
- **Label Propagation:** Build a similarity graph on all data and spread label information across edges.
- **Consistency Regularization:** Encourage model predictions to be invariant under input perturbations on unlabeled data.

Challenges:

- *Confirmation Bias:* errors in pseudo-labels can reinforce themselves.
- *Graph Assumptions:* similarity metrics must reflect true label structure.

Gradient Descent: Practice and Error Reduction

When humans learn a new skill, what is our approach generally?

Gradient Descent: Practice and Error Reduction

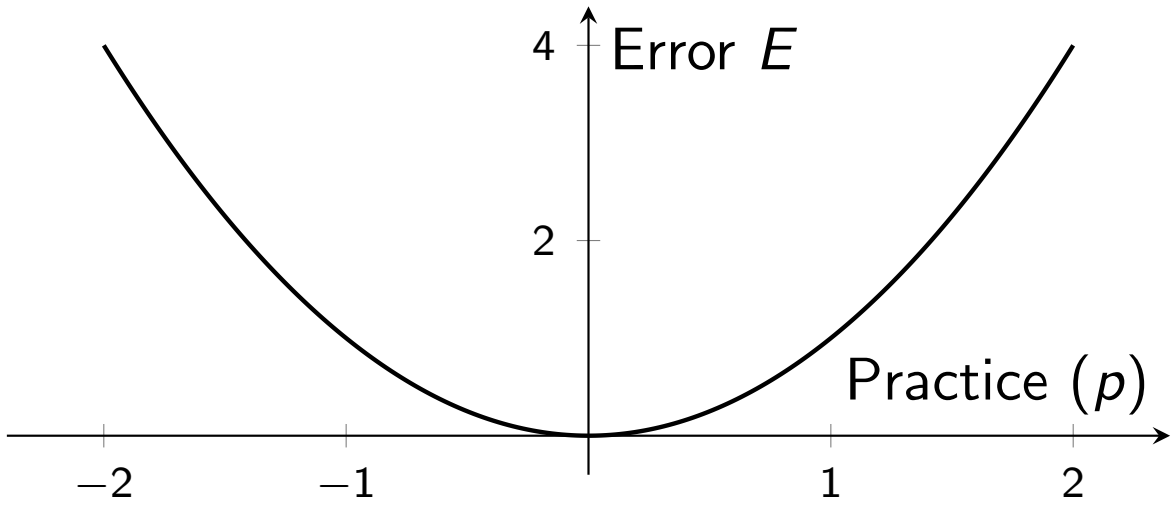
When humans learn a new skill, what is our approach generally?

1. Try an approach
2. Observe our mistakes
3. Adjust our strategy
4. Repeat until errors are minimized

This trial-and-error loop mirrors how gradient descent iteratively reduces model error.

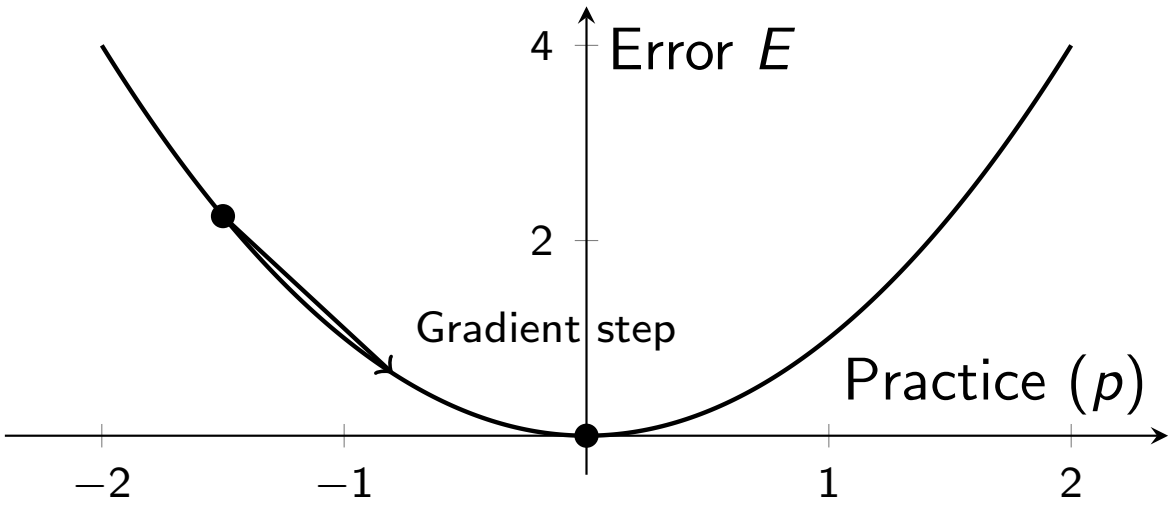
Gradient Descent: Visualize

Suppose we can represent the error as a function $E(p) = p^2$:



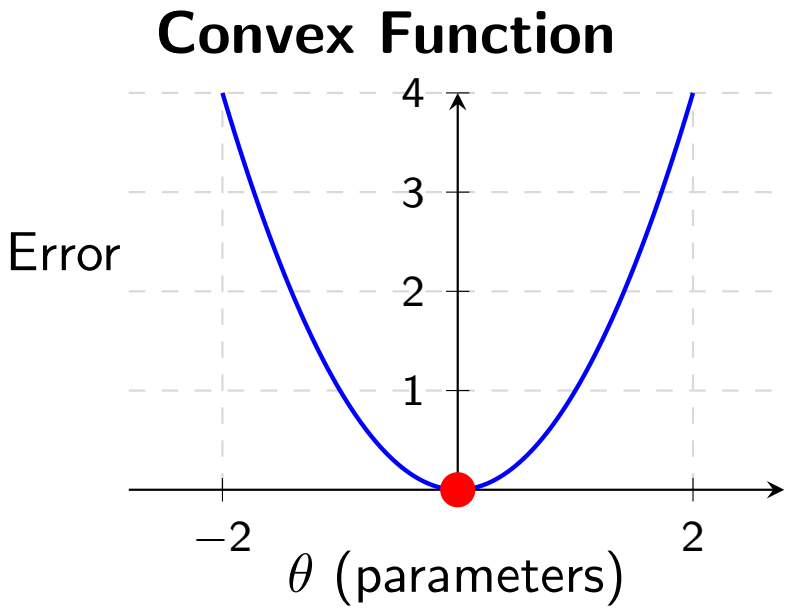
Gradient Descent: Visualize

Suppose we can represent the error as a function $E(p) = p^2$:



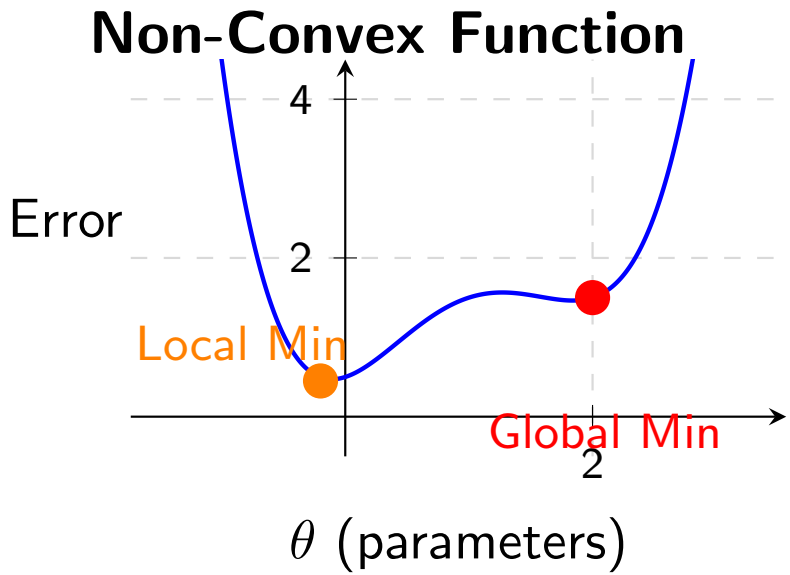
Gradient Descent: Global vs Local Minima

Understanding optimization landscapes in machine learning



Properties:

- Single minimum point
- Gradient descent always finds the optimum



Properties:

- Multiple local minima
- Gradient descent may get trapped

Gradient Descent: How to Find Minima?

Mathematical Formulation

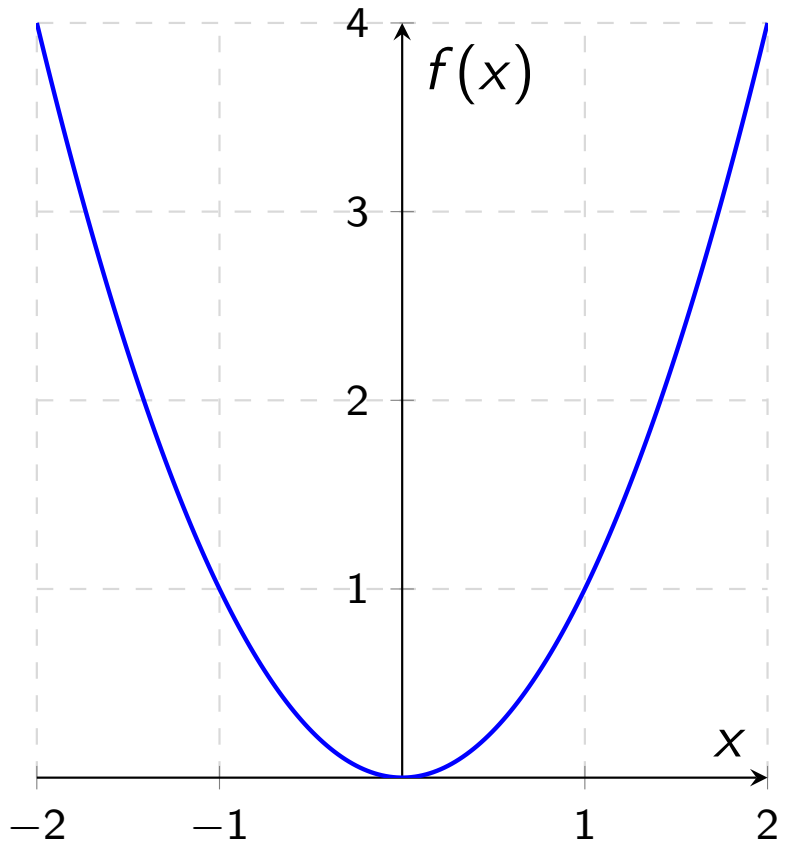
Rather than solving

$$f'(x) = 0$$

in closed form (which is often impossible or impractical in high-dimensional, nonconvex problems), gradient descent uses the iterative **update equation**:

$$x_{\text{new}} = x - f'(x),$$

Repeating this move "downhill" drives x toward a (local) minimum even when no analytic solution exists.



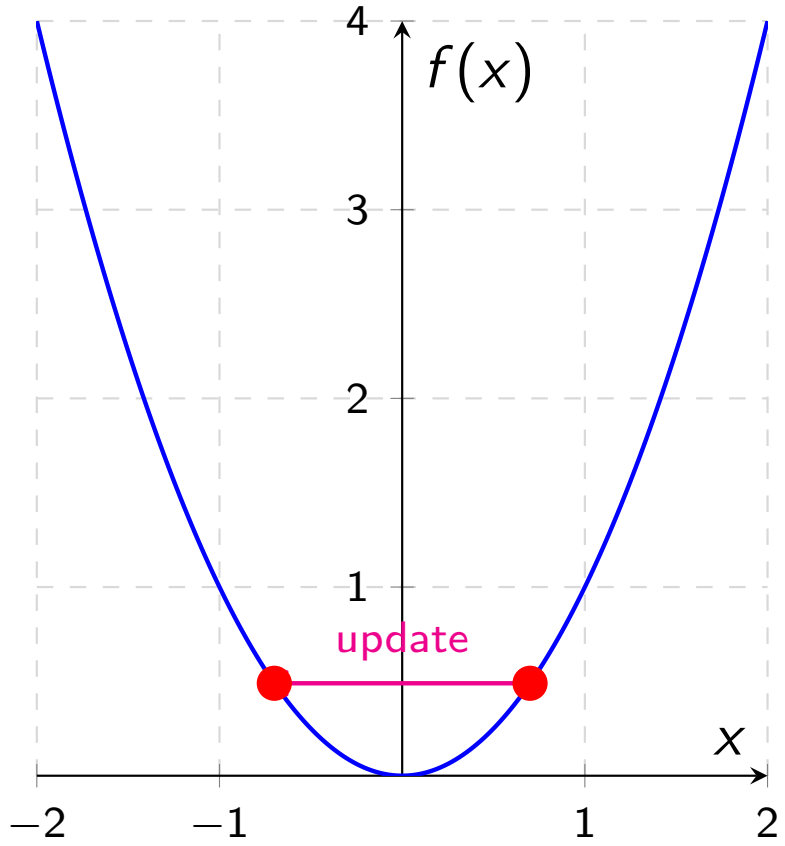
Gradient Descent: Numeric Example

One Update Step at $x = 0.7$

Firs let us find the derivative, $f'(x) = 2x$. Now let us pick a initial value say $x = 0.7$. The we have $f'(0.7) = 2 \cdot 0.7 = 1.4$. Therefore the new value is

$$x_{\text{new}} = 0.7 - 1.4 = -0.7$$

Loss drops from 0.49 at $x = 0.7$ to 0.49 at $x = -0.7$, but we've moved "downhill." But wait a minute, we overshot the minimum. And if we continue this we will keep over shooting the minimum.



Gradient Descent: Visualization

Moving Toward the Minimum

Question: How can we modify the **update equation**

$$x_{\text{new}} = x - f'(x)$$

such that we won't overshoot?

Gradient Descent: Visualization

Moving Toward the Minimum

Question: How can we modify the **update equation**

$$x_{\text{new}} = x - f'(x)$$

such that we won't overshoot?
Introduce a parameter α called the *learning rate*.

Definition (Gradient Descent Update Equation)

$$x_{\text{new}} = x - \alpha \cdot f'(x)$$

where $\alpha > 0$ is the learning rate that controls the step size.

Introducing the Learning Rate

Controlling the size of each update step

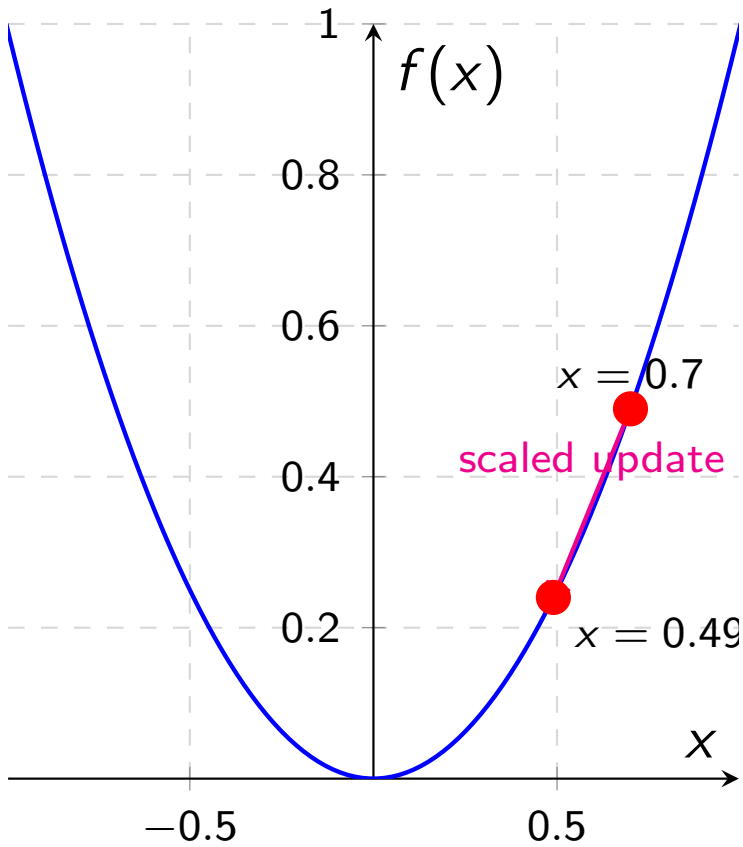
Repeating the steps start at $x = 0.7$:

$$f'(0.7) = 2 \cdot 0.7 = 1.4$$

Therefore

$$x_{\text{new}} = 0.7 - (0.15 \times 1.4) = 0.49.$$

A smaller α makes more conservative steps, ensuring stability.



Loss Functions and Cost Functions

Quantifying prediction error

When training a supervised model, we need a measure of how *bad* (Error) our predictions are. The function we choose to quantify and then minimize during learning is called a **loss function** $\mathcal{L}$.

Definition (Loss Function)

A loss function $\mathcal{L}(y, \hat{y}) = \mathcal{L}(\theta)$ measures the discrepancy between the true label y and the model's prediction $\hat{y} = f_\theta(x)$. It outputs a non-negative real number that quantifies the *cost* of making that prediction.

Common Examples:

- *Mean Squared Error (MSE):* $\mathcal{L}_{\text{MSE}}(y, \hat{y}) = y_i - \hat{y}_i^2$ Penalizes large deviations heavily; used in regression.

Cost Functions (Objective / Empirical Risk)

From per-example loss to what we actually minimize

Given a dataset with n samples and a model $\hat{y}_i = f_\theta(x_i)$:

Definition (Cost / Objective Function)

The (empirical-risk) cost function aggregates per-example losses into a single scalar to optimize:

$$J(\theta) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(y_i, \hat{y}_i).$$

This basically *adds* up the error across the entire training dataset.

The Role of Gradient Descent in Model Training

The Goal: Minimize the Loss Function

When training a model, we aim to make its predictions as accurate as possible. We quantify prediction error via a *loss function* $\mathcal{L}(\theta)$ on a single sample. The we compute the error across the dataset $J(\theta)$. The smaller $J(\theta)$, the better our model's fit to the data.

Therefore our optimization objective is

$$\min_{\theta} J(\theta)$$

Gradient descent provides an efficient, iterative way to approach this minimum.

Gradient Descent: A Tool for Optimization

Iterative Parameter Updates

Gradient descent adjusts the model parameters θ step by step to reduce the loss $\mathcal{L}(\theta)$.

The update equation (As previously done) is

$$\theta_{new} = \theta - \alpha \nabla_\theta \mathcal{J}(\theta)$$

where α is the *learning rate*.

Where:

- θ: Parameters of the model we wish to optimize.
- α: Learning rate, controlling the step size.
- $\nabla_\theta \mathcal{J}(\theta)$: Gradient of the loss w.r.t. θ.

Note: In the single-variable case we had $x_{new} = x - \alpha \frac{dy}{dx}$. In the multivariable case our loss depends on a parameter vector $\theta \in \mathbb{R}^d$:

In Summary

- **Objective:** Find parameters θ that minimize the loss function $\mathcal{L}(\theta)$.
- **Gradient Descent:** An iterative optimization algorithm that updates parameters using:

$$\theta \leftarrow \theta - \alpha \nabla_\theta J(\theta)$$

- **Learning Rate α:** Controls the step size at each iteration, must be carefully tuned to balance convergence speed and stability.

Remember

Too small $\alpha \rightarrow$ slow convergence; too large $\alpha \rightarrow$ overshooting and instability

Linear Regression

From Theory to Practice with Gradient Descent

Linear regression finds the best-fitting line (or hyperplane) through data by optimizing model parameters θ.

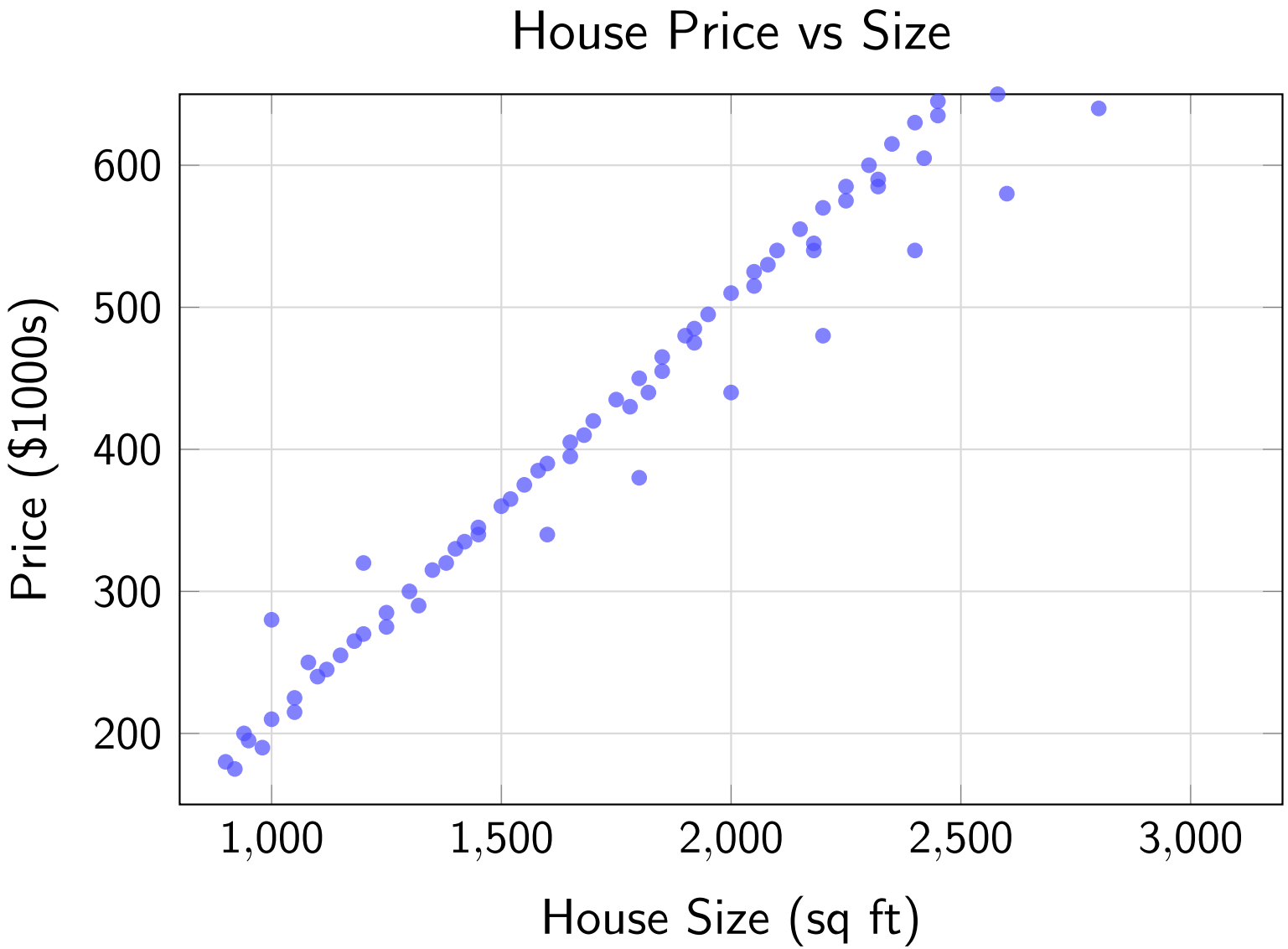
How to use Gradient Descent

While direct methods like the Normal Equation ($\theta = (X^T X)^{-1} X^T y$) exist, we will outline the typical workflow for using gradient descent:

- Step 1: Pick model
- Step 2: Identify loss function $\mathcal{L}(\theta)$
- Step 3: Compute gradients $\nabla_\theta J(\theta)$
- Step 4: Update parameters $\theta_{new} = \theta - \alpha \nabla_\theta J(\theta)$
- Step 5: Repeat until convergence

Note: Convergence occurs when successive updates change θ (or the loss) by only a negligible amount. In practice, you may also stop after a fixed number of iterations.

Example: House Price vs. House Size



Example: House Price vs. House Size

Step 1: Question - What is the model we will use?

Example: House Price vs. House Size

Step 1: Question - What is the model we will use?

$$y = mx + b$$

Example: House Price vs. House Size

Step 1: Question - What is the model we will use?

$$y = mx + b$$

What are the parameter θ we want to learn?

Example: House Price vs. House Size

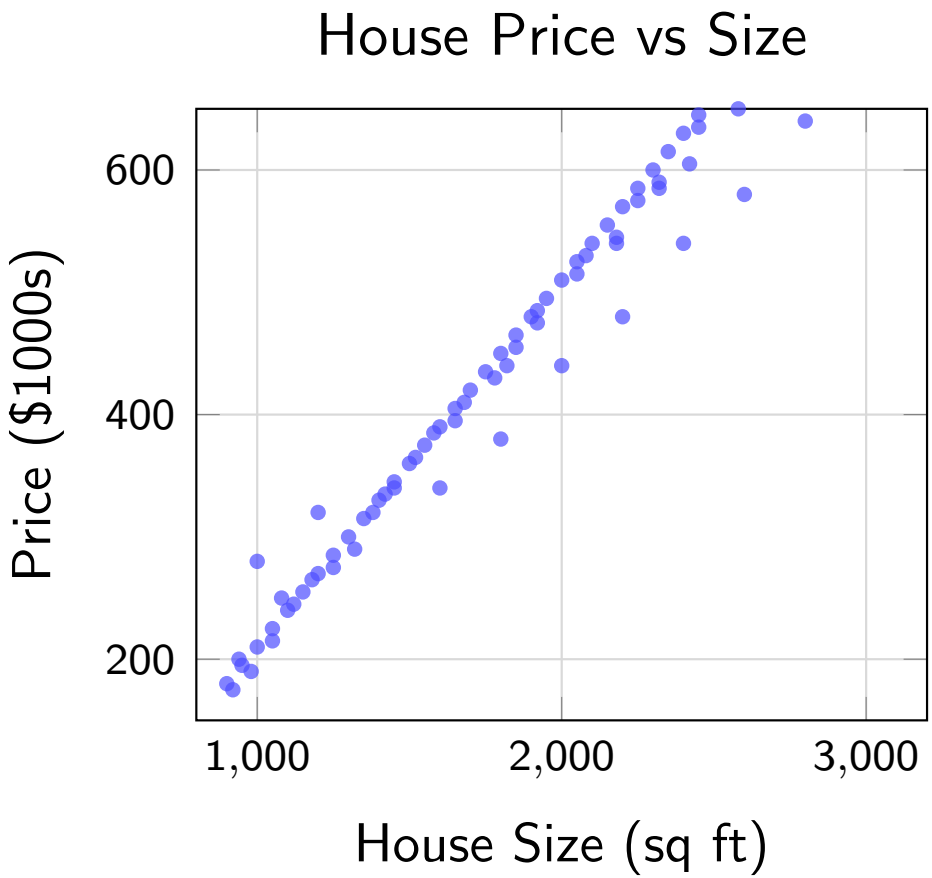
Step 1: Question - What is the model we will use?

$$y = mx + b$$

What are the parameter θ we want to learn?

- The parameters are m and b, so our parameter vector is $\theta = \begin{bmatrix} m \\ b \end{bmatrix}$
- We can also rewrite our equation to fit the machine learning syntax convention to

$$y = \theta_0 x + \theta_1$$



Example: House Price vs. House Size

Great now the we know what parameters we want to *learn*, we need a way to test the error.

Example: House Price vs. House Size

Great now the we know what parameters we want to *learn*, we need a way to test the error.

Step 2: Question - What would be a loss function $\mathcal{L}(\theta)$?

Example: House Price vs. House Size

Great now the we know what parameters we want to *learn*, we need a way to test the error.

Step 2: Question - What would be a loss function $\mathcal{L}(\theta)$? We can use the

$$\mathcal{L}(y, \hat{y}) = y_i - \hat{y}_i$$