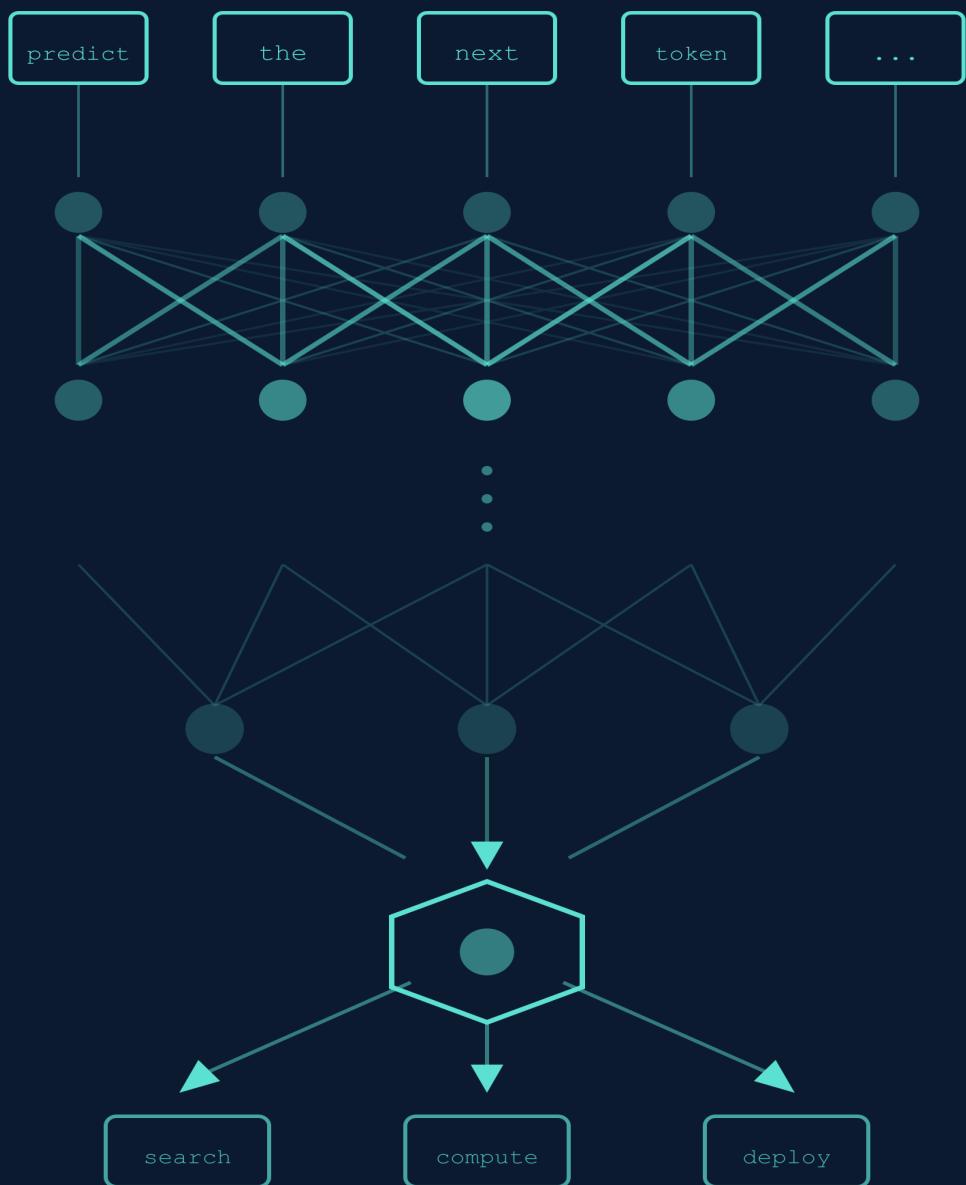


Practical Language Models II

From Intuition to Agents in Production
with Python and FastAPI



Igor Benav

Practical Language Models

From Intuition to Agents in Production

Igor Benav

Contents

| | |
|--|----------|
| Preface | 3 |
| About This Book | 3 |
| What You'll Build | 3 |
| How to Use This Book | 4 |
| Prerequisites | 4 |
| Book Structure | 5 |
| Contributors | 5 |
| 1 Chapter 1: The Learning Problem | 6 |
| 1.1 What Does It Mean to Learn? | 6 |
| 1.2 Supervised and Unsupervised Learning | 7 |
| 1.3 Two Kinds of Problems | 9 |
| 1.4 From Pixels to Predictions | 10 |
| 1.5 How Good Is Good Enough? | 14 |
| 1.6 Generalization | 18 |
| 1.7 Hands-On: Recognizing Handwritten Digits | 20 |
| 1.8 Exercises | 25 |

Preface

To be written.

About This Book

Most machine learning resources fall into two camps: API tutorials that have you calling `.fit()` without understanding the mechanics, or textbooks dense with notation that are targeted on researchers. This book tries to find a middle ground.

I'm not a researcher. I'm a developer who maintains open-source libraries and builds production AI pipelines for enterprise clients. In my experience, the biggest bottleneck in AI engineering isn't the model's complexity; it's building a system that is reliable, efficient, and maintainable. I wrote this book to give you the mental model I use to ship these systems.

We start with the mathematical definition of learning before touching any library. This isn't for academic rigor; it's for survival. When your model fails in production, you need to know if the problem is your data, your architecture, or your loss function. If you skip the math, you'll still know what's happening. If you engage with it, you'll be able to reason about new problems on your own.

We build neural networks from scratch in NumPy before using any framework. We use Python throughout, uv for package management, FastAPI for serving models, and PydanticAI for building LLM applications. We're not surveying every algorithm or teaching you the "best" approach. We're teaching you one coherent path from mathematical foundations to deployed applications, so you have solid ground to stand on when you explore other directions later.

The math is important, but it's not the gatekeeper. Every concept is introduced with intuition first: what we're trying to do and why. The equations formalize what you already understand. If you skip the math, you'll still know what's happening and why. If you engage with it, you'll know how and be able to reason about new problems on your own.

What You'll Build

The book follows a single thread: by the end, you'll have a good understanding of how language models work from the ground up and have built systems that people can actually use.

We start with the learning problem itself: what it means for a machine to learn, what can go wrong, and how you know it's working. Then we build neural networks from scratch in NumPy: perceptrons,

forward passes, backpropagation, gradient descent. You'll train a network by hand before any framework does it for you.

Then we tackle the core challenge of this book: how to represent language as numbers. We'll cover word embeddings, attention mechanisms, and the transformer architecture that powers modern language models. You'll understand why these models work, where they break, and what the math is actually doing.

Finally, we build. You'll create LLM-powered agents with tools, structured output, and retrieval-augmented generation (RAG). You'll design multi-step pipelines that combine what LLMs are good at with what code is good at. And you'll deploy the result with FastAPI so it's not just running on your machine.

How to Use This Book

This book assumes you know Python. You should be comfortable with classes, decorators, and `asyncio`, we'll also use NumPy. Some background in linear algebra and calculus is also necessary; specifically dot products and the chain rule. If you haven't seen these in a while, the book explains what you need when you need it. You don't need a math degree.

The chapters are meant to be read in order. Each one builds on the previous.

Each chapter should have these:

- **Intuition first:** What are we trying to do and why?
- **The math:** Formalizing the intuition with equations and concrete numbers
- **Implementation:** How to actually do it in code
- **Hands-on project:** A practical exercise that uses what you just learned
- **Exercises:** More practice on your own

Type the code yourself. Don't copy and paste. The errors you make and fix along the way are part of learning. Change things. Break things. See what happens.

When you get stuck, resist asking AI for the solution (yes, even though this is a book about AI). Ask for a hint if you need to, but do the thinking yourself. Struggling is normal. It's also how you actually learn, rather than just feeling like you're learning.

The complete code for each chapter's hands-on project is available at github.com/Applied-Computing-League/applied-ai. Consider it a last resort. If you look at the solution before genuinely struggling with the problem, you're only cheating yourself out of the learning.

Prerequisites

You'll need:

- Python 3.11 or newer installed

- A code editor (VS Code works well, I like Zed)
- A terminal you're comfortable using
- Comfortable Python knowledge (variables, functions, loops, lists, dictionaries, classes, decorators, `asyncio`)
- Basic linear algebra (vectors, dot products, matrix multiplication)
- Basic calculus (derivatives, chain rule)
- Willingness to read error messages instead of panicking

You don't need prior machine learning experience. You don't need to know NumPy, PyTorch, or any ML framework. We'll cover what you need as we go.

Book Structure

The book is organized in three parts.

Part I: Foundations covers the math and intuition you need before touching an API. What learning means, how neural networks work, how to represent text as numbers, how attention and transformers work, and how text generation actually happens. By the end of Part I, you'll understand what's going on inside a language model.

Part II: Building with LLMs takes you from understanding to building. You'll work with LLM APIs, create agents with tools, get structured data back from models, build retrieval-augmented generation systems, and design multi-step pipelines. By the end of Part II, you'll have built real AI applications.

Part III: Deploying to Production gets your applications off your laptop. You'll serve models with FastAPI, handle the reliability problems that come with depending on external AI services, and deploy to a real server.

Contributors

| Name | Role |
|------------|--------|
| Igor Benav | Author |

1 Chapter 1: The Learning Problem

When we talk about learning, we usually mean getting better at something through study or experience. A child learns to recognize dogs by seeing many dogs. A piano player gets better by playing thousands of songs. In both cases, the learning comes from data: examples, outcomes, patterns observed over time.

We'd love machines to do these kinds of tasks too: identify dogs, read handwriting, translate languages. The obvious approach is programming explicit rules: "if the image has pointy ears and a tail, it's probably a dog." But you'll see that this isn't a feasible approach. Ears come in all shapes. Some dogs don't have tails. Chihuahuas and rottweilers look nothing alike. The rules multiply, the exceptions multiply faster, and you never cover everything. The alternative is to skip the rules entirely and make machines go through the same process we go through: show the machine enough examples and let it figure out the patterns itself. That would be machine learning. The question is how to make it work.

This chapter sets up the problem. We'll define what it means for a machine to learn, look at the main types of learning, and understand what "good enough" means in mathematical terms. Everything else in this book (neural networks, transformers, language models) is a specific method for solving the problem we define here.

1.1 What Does It Mean to Learn?

Think about reading someone's handwriting. You see a weird digit and instantly know it's a 7. You've been doing this your whole life without thinking about it. But try writing down the rules for how you do it. "A 7 has a horizontal line at the top and a diagonal line going down." That works until someone crosses their 7, writes a 1 that looks like a 7, or writes a 7 that looks like a 1. The variation between people's handwriting is enormous, and no set of rules you write down will cover all of it.

We can imagine there's some platonic function that takes an image of a handwritten digit and correctly identifies it as 0 through 9. We as humans can sort of do it, but we can't explain exactly how our brain does it. We've just seen enough handwriting in your life that we learned the pattern. We want machines to do the same thing: look at enough examples and figure it out.

We have data: thousands of handwritten digits that humans have already labeled. We know this scribble is a 7, that one is a 1, this one is a 9. The hope is that a machine can look at these labeled examples and figure out a function that does a reasonable job of classifying new, unseen handwriting.

Let's make this precise. There exists some unknown function $f : X \rightarrow Y$ that we want to learn. X is the set of all possible inputs (all possible images of handwritten digits, for example). Y is the set of outputs (the digits 0 through 9 in this case).

We can't access f directly, we don't know it. What we have is a dataset $D = \{(x_1, y_1), (x_2, y_2), \dots, (x_N, y_N)\}$: a collection of input-output pairs where each $y_i = f(x_i)$. These are examples of this function's behavior.

Our job is to use D to find a function $g : X \rightarrow Y$ such that $g \approx f$. Not just on the examples we've seen, but on new inputs the machine has never encountered. That last part is what separates learning from memorization.

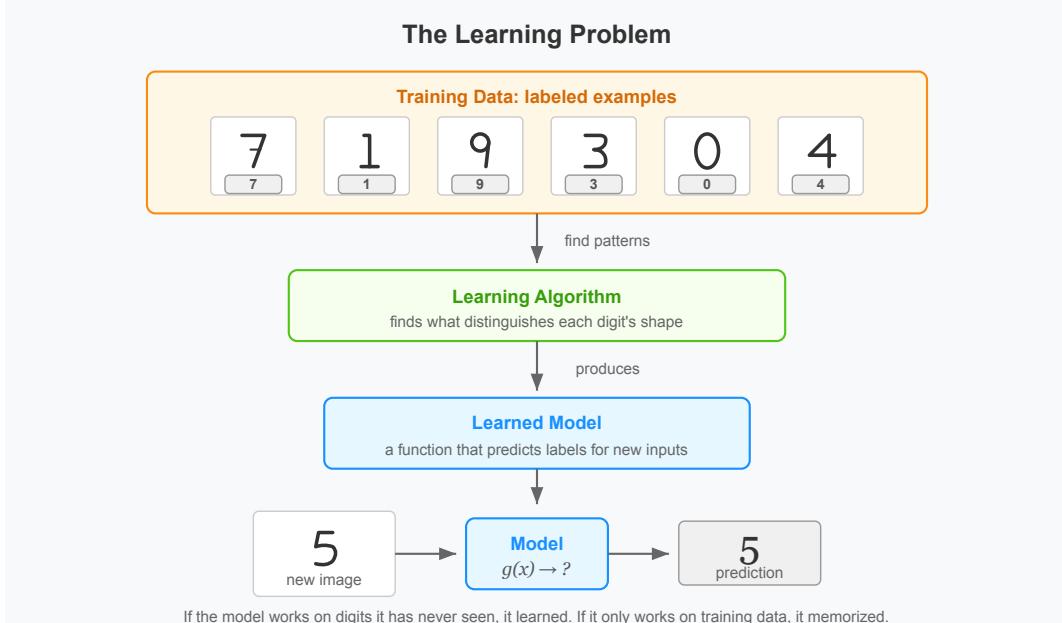


Figure 1.1: The Learning Problem

In plain language: f is the perfect classifier (that always gets it right). D is our training set (thousands of labeled digit images). g is what the algorithm produces after studying D . The algorithm's goal is to find a g that classifies new handwriting (digits not in D) correctly to a certain margin.

That's the abstract setup: an unknown function, a dataset, and a search for an approximation. But not all datasets look the same, and the kind of data you have determines what kind of learning is possible.

1.2 Supervised and Unsupervised Learning

The setup above assumes something specific about the data: every example has a label. Every digit image is tagged with the correct number. This is **supervised learning**, and it's the most common setting.

In supervised learning, the dataset looks like this:

$$D = \{(x_1, y_1), (x_2, y_2), \dots, (x_N, y_N)\}$$

Each example is a pair: an input x_i and its correct output y_i . The algorithm's goal is learning the relationship between inputs and outputs by studying these pairs. It's "supervised" because the labels act like a teacher telling the algorithm the right answer for each example.

But labels are expensive. Someone had to look at each of those thousands of images and write down the correct digit. For many problems, labeled data is scarce. What if you have millions of images but no labels?

Unsupervised learning works with unlabeled data:

$$D = \{x_1, x_2, \dots, x_N\}$$

No labels. The algorithm's task is to find structure on its own. It might discover that some images share similar shapes and group them together, but it doesn't know these groups are "0" and "7" (nobody told it); it simply found a meaningful separation in the data.

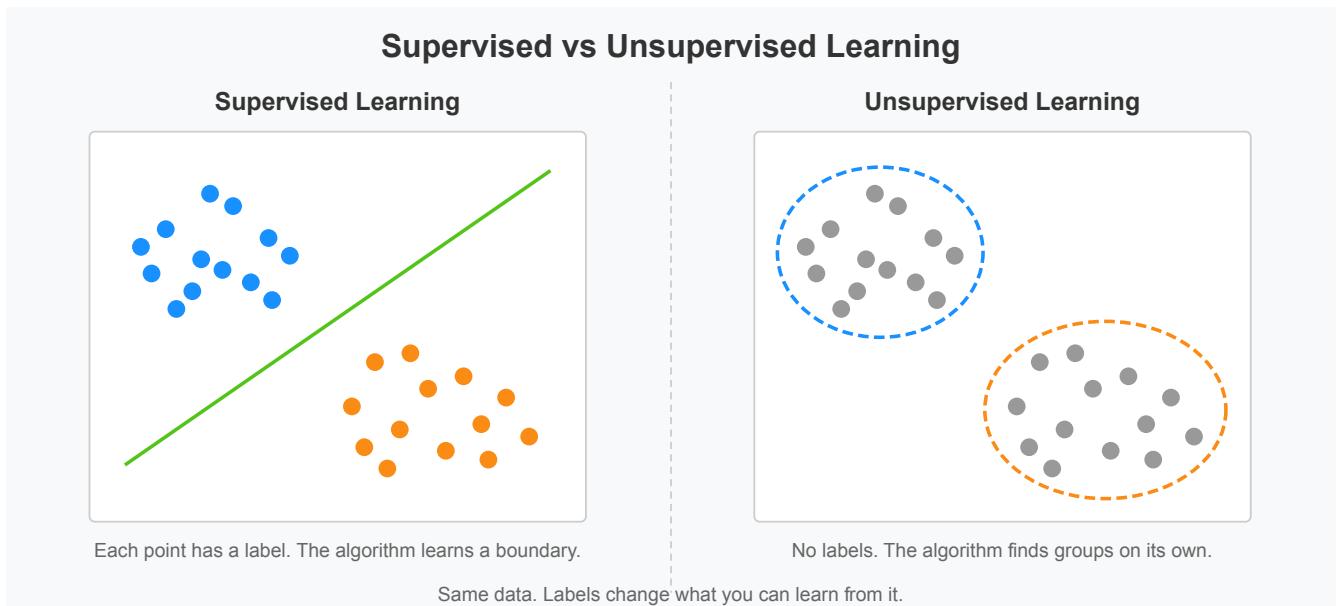


Figure 1.2: Supervised vs Unsupervised Learning

The digit recognizer we drafted lead us to supervised learning: the algorithm can't figure out that a scribble means "7" unless someone tells it. But imagine you have a million handwriting samples with no labels. An unsupervised algorithm could still group similar-looking characters together, separating the round shapes from the angular ones, the short ones from the tall ones. It wouldn't know the groups are "0" and "7" (nobody told it), but it found meaningful structure. Unsupervised learning shows up in places where labeling is expensive or impossible: grouping customers by purchasing behavior, detecting unusual network traffic, finding topics in a collection of documents.

There's a third setting worth mentioning briefly. **Reinforcement learning** is learning from interaction: an agent takes actions in an environment, receives rewards or penalties, and learns to maximize reward

over time. This is how AlphaGo learned to play Go: not from labeled examples of good moves, but from playing millions of games and learning what leads to winning. We won't cover reinforcement learning in this book, but you should know it exists.

We'll focus on supervised learning for the rest of this chapter and most of the book. The neural networks we build in Chapter 2 are supervised learners. The language models we study later were trained in a way that's technically self-supervised (the labels come from the text itself, as we'll see in Chapter 4), but the mechanics are closer to supervised learning than unsupervised.

Now we know we're doing supervised learning: we have inputs with labels, and we want to learn the relationship between them. But the nature of those labels is important. Predicting a category and predicting a number are fundamentally different problems.

1.3 Two Kinds of Problems

Within supervised learning, there are two types of problems based on what Y looks like.

Classification is when Y is a set of categories. The digit 0 through 9. Spam or not spam. Positive, negative, or neutral sentiment. The output is a label from a finite set. Our digit recognizer is a classification problem with $Y = \{0, 1, 2, \dots, 9\}$.

Regression is when Y is a number on a continuous scale: predicting the price of a house given its square footage; predicting tomorrow's temperature. The output is a value like \$425,000 or 23.5°C. There's no finite set of labels to choose from.

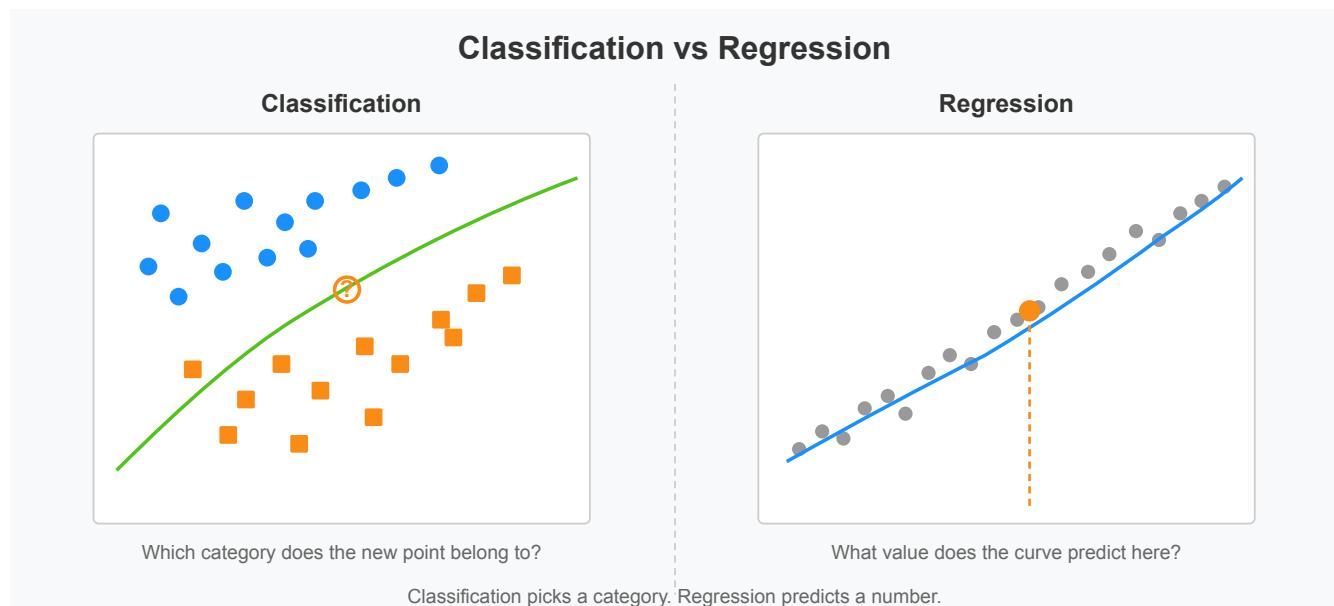


Figure 1.3: Classification vs Regression

The distinction is fundamental because it changes what “getting it wrong” means. In classification, you’re either right or wrong: the digit is a 7, and you said 7. In regression, you’re almost always a little off: the house sold for \$425,000 and you predicted \$419,000. How you measure that error is important, and we’ll get to it soon.

Most of what we build in this book involves classification. When a language model predicts the next word, it’s choosing from a vocabulary of possible words: that’s classification over a very large set. When a sentiment analyzer reads a review and outputs “positive,” that’s classification too.

We know the problem (find $g \approx f$), the data setting (supervised), and the type of output (classification). But we’ve been talking about all of this abstractly. What does x_i actually look like? What does the machine see when it looks at a handwritten digit? And how does it go from that raw input to a prediction?

1.4 From Pixels to Predictions

We’ve defined the problem abstractly: find g such that $g \approx f$. But what does g actually look like? How does a machine take an image of a handwritten digit and produce the number 7? Before we can measure how good a model is, we need to understand how it even makes a prediction at all.

Start with what the machine actually sees. You look at a handwritten digit and see a shape. The machine sees a grid (matrix) of numbers. Each pixel in a 28×28 image has a brightness value between 0 (black) and 255 (white). Flatten that grid into a row, and you have 784 numbers. That row of numbers (the one dimension vector we get by flattening the matrix) is x_i in our notation. That’s what the input looks like to the machine.

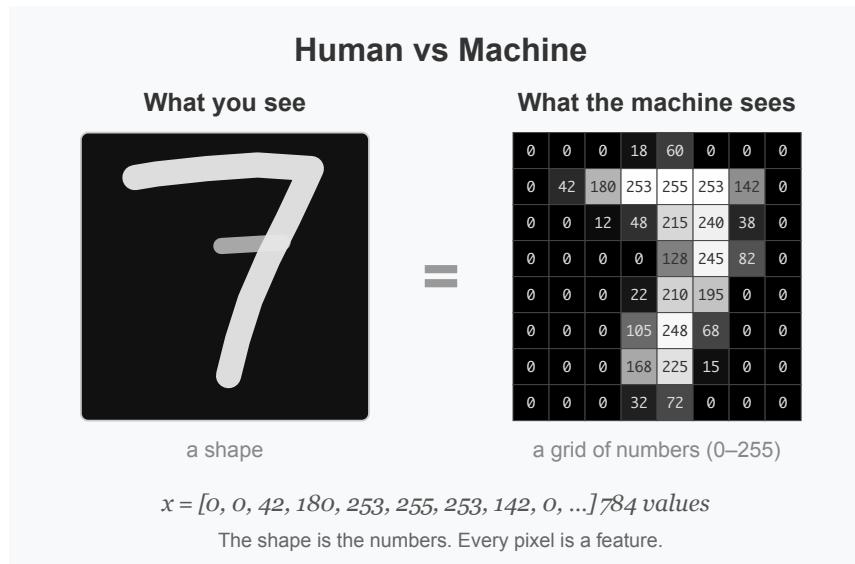


Figure 1.4: What the Machine Sees

$$x_i = [0, 0, 0, 12, 180, 255, 253, 142, 0, \dots] \quad (784 \text{ values})$$

The machine has no concept of “stroke” or “curve” or “loop.” It sees a list of numbers. Everything it learns, it learns from patterns in those numbers. This numerical representation is called the **features** of the input, and it’s one of the most important ideas in machine learning: what the model can learn is limited by what the features capture. Good features make learning easy; bad features make it impossible. We’ll see this concretely in the hands-on section, and it becomes the central question of Chapter 3 when we tackle text.

So we have inputs as lists of numbers. How do we get from there to a prediction? The simplest idea is **similarity**: if two images have similar pixel values, they’re probably the same digit. A new image that looks like the 7s we’ve seen before is probably a 7.

But “similar” needs a precise definition. We have two lists of 784 numbers. How different are they? One approach we could try: compare them position by position. If pixel 47 is bright in both images, the difference is small. If it’s bright in one and dark in the other, the difference is large. Square each difference (so negatives don’t cancel positives), add them all up, and take the square root. This is the **Euclidean distance**:

$$d(a, b) = \sqrt{\sum_{j=1}^{784} (a_j - b_j)^2}$$

Applying it:

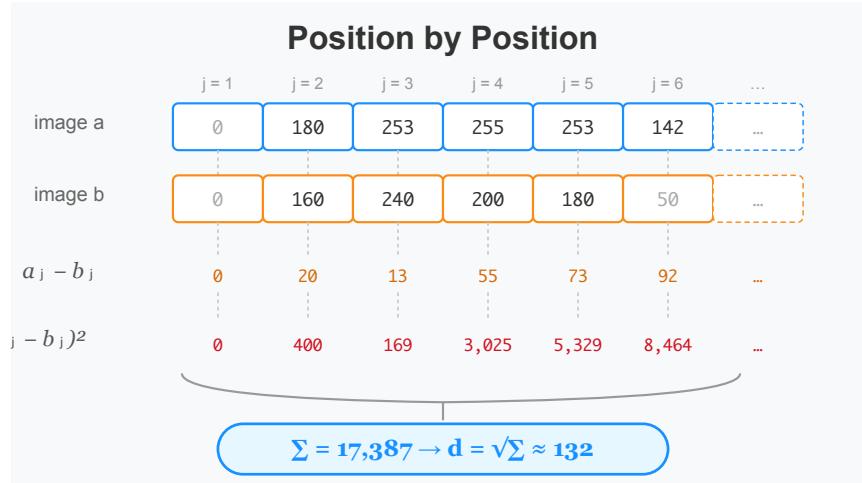
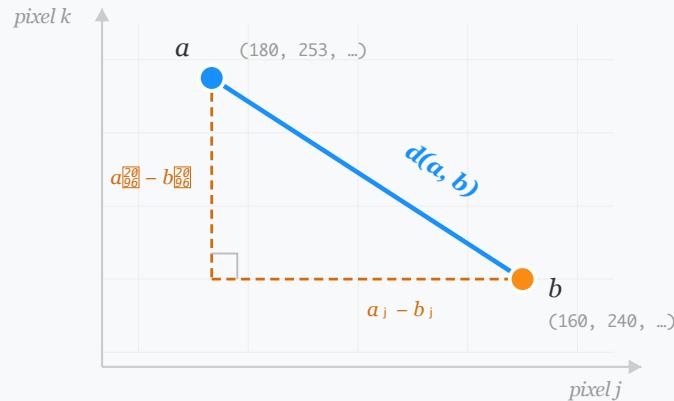


Figure 1.5: Position by Position

If two images are identical, every difference is 0 and the distance is 0. If they’re very different, the distance is large. Notice that we’re comparing each pixel to the pixel *at the same position* in the other image; this assumes the digit sits in roughly the same spot in both images. We’ll come back to that.

This is the same formula as the distance between two points in space, just extended to 784 dimensions. Each image is a point in a 784-dimensional space, and we're measuring how far apart two points are.

Images as Points in Space



Each image is a point. Distance = how far apart they are.
Same formula as Pythagoras, extended to 784 dimensions.

Figure 1.6: Images as Points in Space

Now we have a way to compare images. The simplest possible way to classify follows immediately: given a new image, compare it to every image in the training set. Find the most similar one. Use its label as the prediction. If the closest training image is a 7, predict 7. This is what we call **nearest neighbor**.

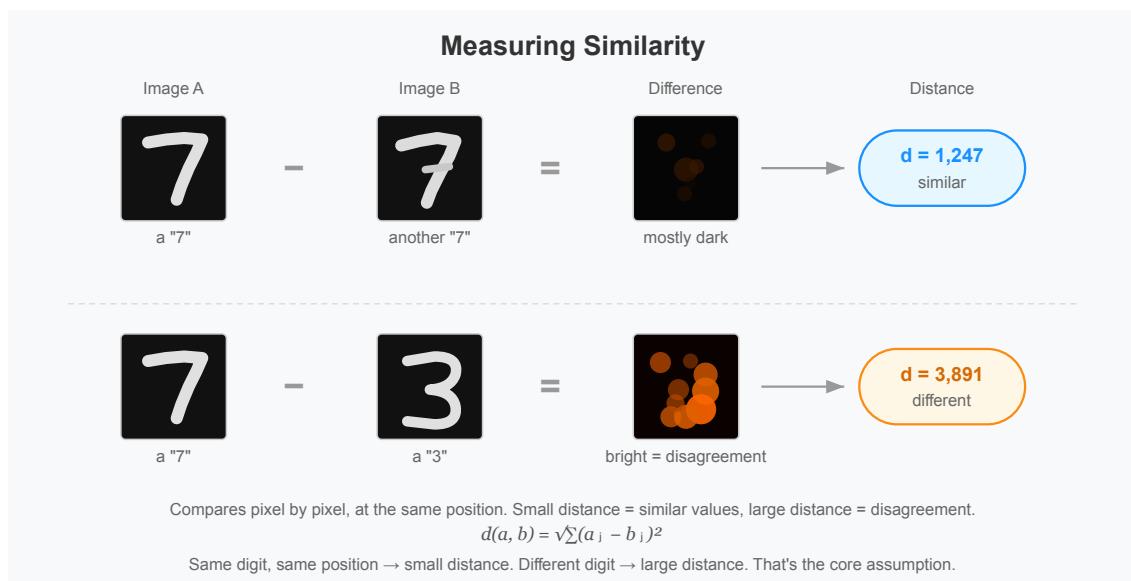


Figure 1.7: Measuring Similarity

This kind of works given all the assumptions hold. But think about the cost. To classify one new image,

you compare it against all 56,000 training images. To classify 14,000 test images, that's $56,000 \times 14,000 = 784$ million distance calculations. The model isn't a compact function; it's the entire training set. This is pure memorization: the "model" is the data itself.

Can we do better? What if instead of keeping every training image, we summarized each digit class as a single **template**? Average all the 7s to get a single blurry image of what a typical 7 looks like. Average all the 1s. Do this for each digit, and you have ten templates. To classify a new image, measure its distance to each of the ten templates and pick the closest.

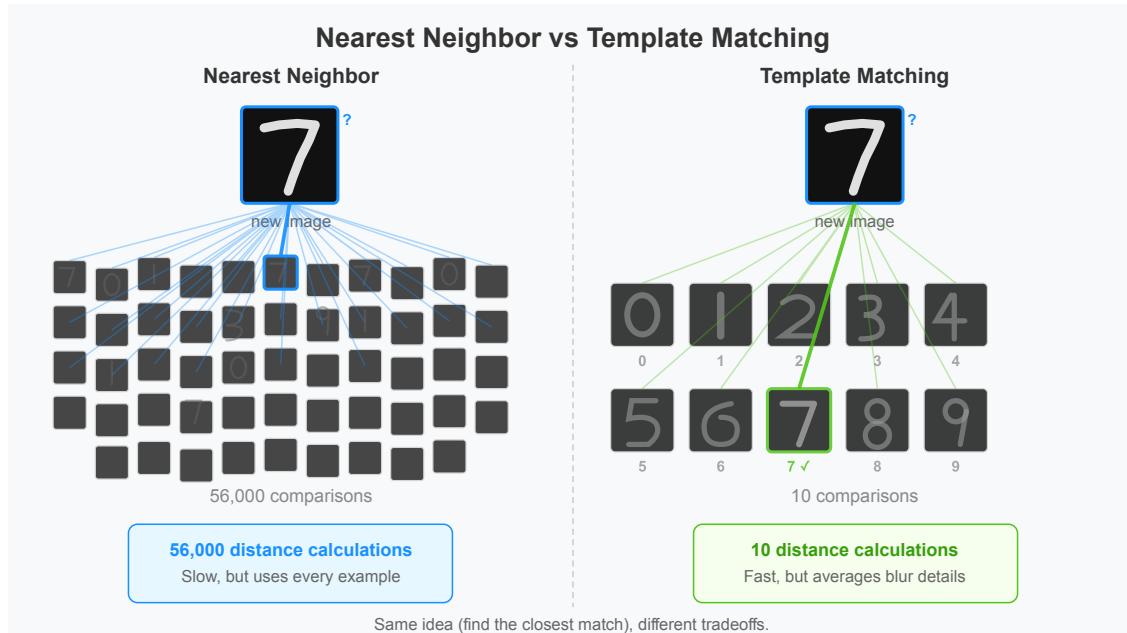


Figure 1.8: Nearest Neighbor vs Template Matching

This is called **template matching**, and it's our first real g : a function that takes an image and produces a digit label. It's faster (10 distance calculations instead of 56,000) and interpretable (you can literally look at the templates and see what the model "thinks" each digit looks like). The average 0 is a fuzzy oval. The average 1 is a narrow smear.

It's also obviously limited. Averaging thousands of different handwriting styles produces a blurry mess. When two digits have similar average shapes (4 and 9 both have a vertical stroke on the right), the templates can't really tell them apart. And some digits have multiple distinct styles: some people write 1 as a simple vertical stroke, others add serifs or a diagonal lean. One template can't capture both.

You could fix that last problem by keeping multiple templates per digit: instead of one average 7, keep three different "styles" of 7. This is the beginning of what is called **clustering**: automatically discovering distinct groups within a class. But even with more templates, there's a deeper problem we haven't dealt with.

Remember the assumption we flagged earlier: Euclidean distance compares each pixel to the pixel at the same position. Take a 7 and shift it three pixels to the right. Same digit, same handwriting, barely percep-

tible to a human. But the Euclidean distance to the original is enormous; on real MNIST digits, it can be larger than the distance to a completely different digit. Pixel-wise, a slightly displaced 7 looks less like itself than a 3 does. Why? Everywhere the digit has a sharp edge, shifting creates a large disagreement at two positions: where the edge was, and where it moved to. The distance doesn't measure how different two shapes are. It measures how much their edges have moved.

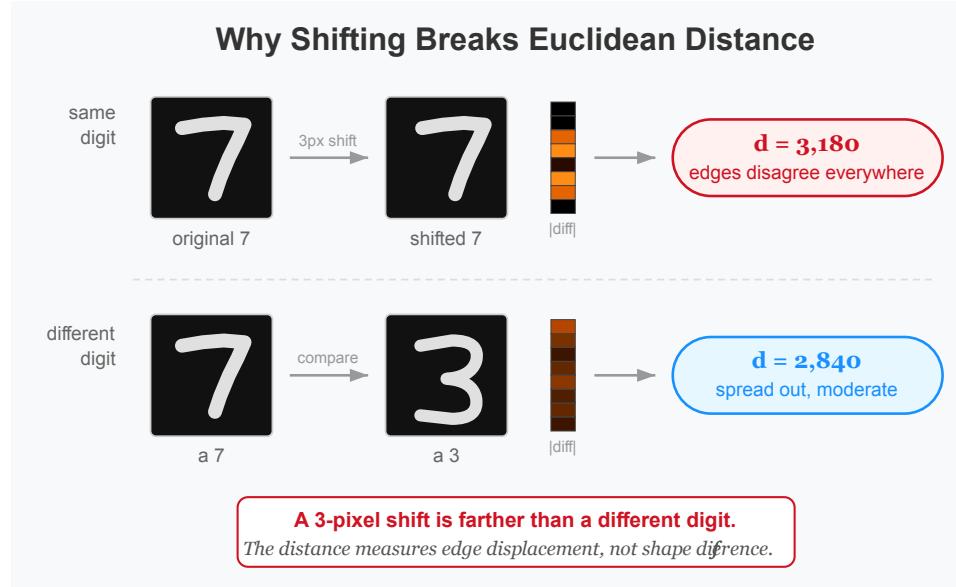


Figure 1.9: Why Shifting Breaks Euclidean Distance

No amount of templates or clusters will fix this. The features (raw pixels) don't capture position-invariance, and Euclidean distance is the wrong way to compare them.

This is why we need more powerful models. The template matcher can't learn that "a 7 shifted left three pixels is still a 7." We need to find something that can.

But the template matcher does something important: it gives us a concrete g that makes predictions we can evaluate. We need that, because the next question is: how do we measure whether g is any good?

1.5 How Good Is Good Enough?

We've said the goal is to find g such that $g \approx f$. But "approximately equal" is vague. We need a number that says how wrong g is, so we can make it less wrong.

Start with classification. Your model looks at a handwritten digit and predicts "7." The true label is "7." Good. Next one: true label is "3," prediction is "8." Bad. Over a set of N examples, you count:

$$\text{Accuracy} = \frac{\text{number correct}}{N}$$

If your model correctly classifies 9,200 out of 10,000 test digits, its accuracy is 92%. That's straightforward.

Regression is trickier. You're predicting numbers, so you'll essentially never get the exact value. You need to measure how far off you were. The simplest idea: take the difference between your prediction and the true value, and make it positive.

$$|g(x_i) - y_i|$$

If the house sold for \$425,000 and you predicted \$419,000, your error is \$6,000. Average this over all examples and you get the **Mean Absolute Error** (MAE):

$$\text{MAE} = \frac{1}{N} \sum_{i=1}^N |g(x_i) - y_i|$$

That works, but it treats all errors equally. Being off by \$1,000 ten times is the same total error as being off by \$10,000 once. In practice, big errors are usually much worse than small ones. A house price model that's usually close but occasionally wildly wrong is dangerous. One that's consistently off by a small amount is useful.

We need to penalize larger errors more than small errors, so instead of taking the absolute value, we can square the error:

$$(g(x_i) - y_i)^2$$

Small errors stay small when squared ($5^2 = 25$). Big errors explode ($100^2 = 10,000$). This punishes large mistakes disproportionately. Average these and you get the **Mean Squared Error** (MSE):

$$\text{MSE} = \frac{1}{N} \sum_{i=1}^N (g(x_i) - y_i)^2$$

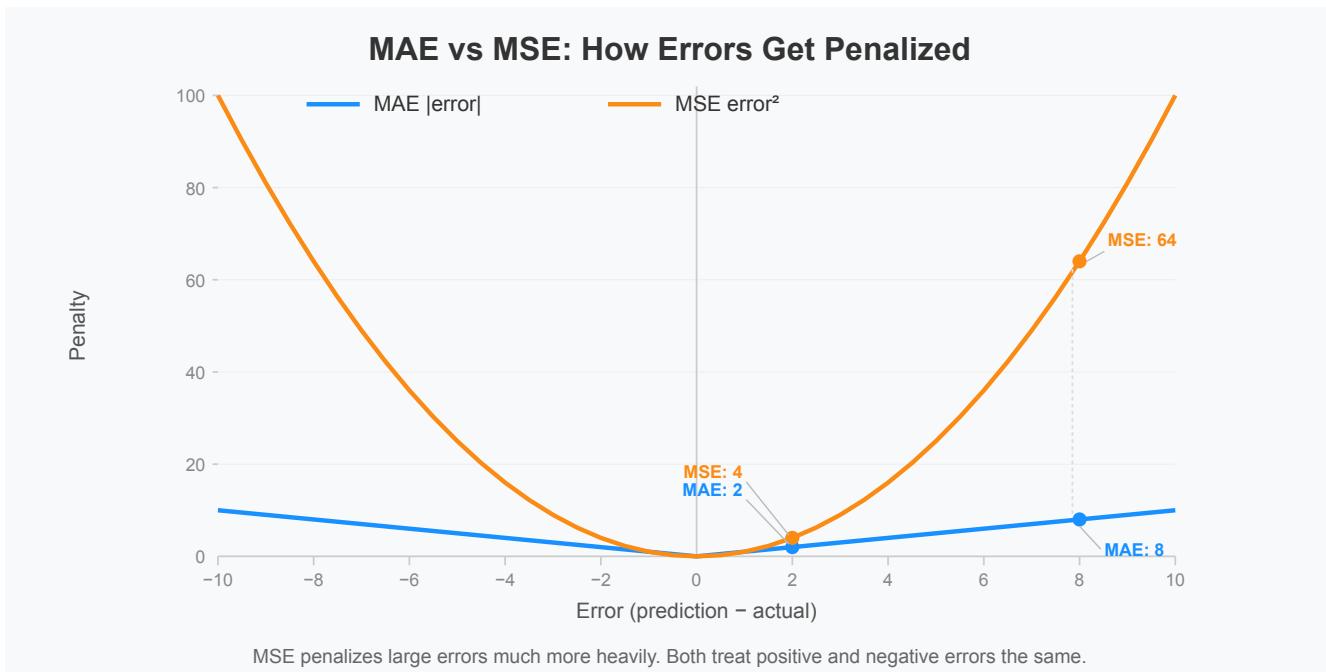


Figure 1.10: MAE vs MSE: How Errors Get Penalized

The diagram makes this concrete. For small errors, MAE and MSE are close. For large errors, MSE grows much faster. This is why MSE is the default for most regression problems: it strongly discourages the kind of catastrophic predictions that make a model useless in practice.

Notice anything familiar? MSE has the same structure as the Euclidean distance from earlier in the chapter. Your predictions form one list of numbers, the true labels form another, and MSE measures how far apart they are; position by position, squared, summed.

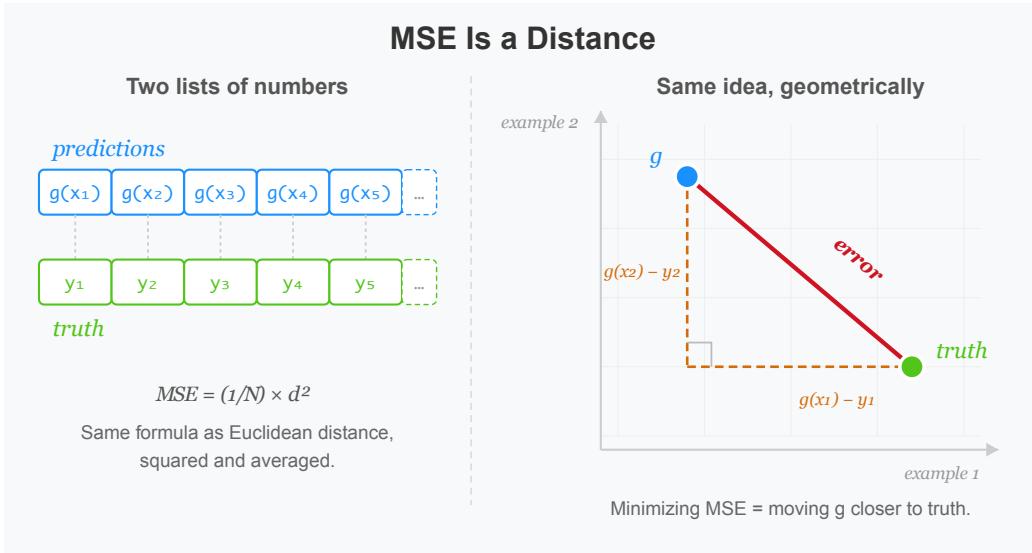


Figure 1.11: MSE Is a Distance

Minimizing MSE means moving your predictions as close as possible to the truth, in exactly the geometric sense we saw before.

These measures of error have a name: **loss functions** (sometimes called cost functions). A loss function takes your model's predictions and the true answers, and returns a single number that says how wrong you are. The entire learning process is about finding the function g that makes this number as small as possible, the closest one to the platonic truth we can get with our data.

The choice of loss function shapes what the model learns. MSE pushes the model to avoid big errors at all costs. MAE pushes the model to be right on average, tolerating occasional larger mistakes. The geometry shows why. MSE measures the straight-line distance between predictions and truth; and a single large error in any direction stretches that line. MAE walks the grid, one axis at a time; ten small errors add up the same as one error ten times larger.

Two Ways to Measure Distance

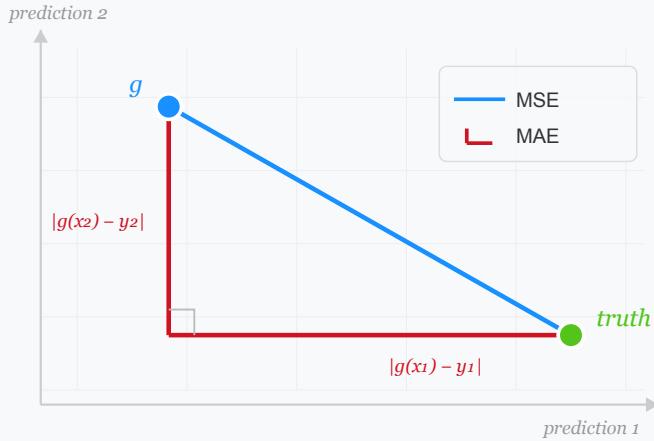


Figure 1.12: Two Ways to Measure Distance

A model that's off by \$1,000 on ten houses and perfect on the rest walks the same total distance as one that nails every house but misses one by \$10,000. MSE would strongly prefer the first model; MAE treats them equally.

So now we can measure how wrong a model is. But there's a trap. A model can score perfectly on the data it's seen and still be useless. Everything depends on what happens when the model encounters something new.

1.6 Generalization

A model that memorizes its training data can get 100% accuracy on that data. Every digit classified perfectly. MSE of exactly zero. By the loss function's measure, it's a perfect model.

But show it a new digit it hasn't seen, and it might fail completely. It didn't learn the pattern ("round shapes with a closed loop tend to be 0s"). It learned the specific examples ("pixel values matching image #4,721 map to the label 9"). This is **overfitting**: the model fits the training data too well and fails on anything new.

The opposite problem exists too. A model that's too simple might not capture the real patterns in the data. If you try to tell apart 0s and 1s with a single rule ("if the image is dark in the center, it's a 0"), you'll get some right but miss the complexity of real handwriting. This is **underfitting**: the model isn't powerful enough to learn what's actually there.



Figure 1.13: Underfitting, Good Fit, and Overfitting

Think about it in terms of studying for an exam. You have practice questions and answers from the past five years. You could memorize every answer word for word. On a test that reuses those exact questions, you'd score 100%. On a test with new questions covering the same material, you'd fail. You never learned the subject. You learned the specific answers.

A good student extracts the underlying principles and applies them to new questions. That's generalization. We want our models to do the same.

This is why we split data into **training data** and **test data**. Train the model on one set, evaluate it on another. The test data simulates "new, unseen examples." If the model does well on training data but poorly on test data, it's overfitting. If it does poorly on both, it's underfitting. If it does well on both, it learned something real.

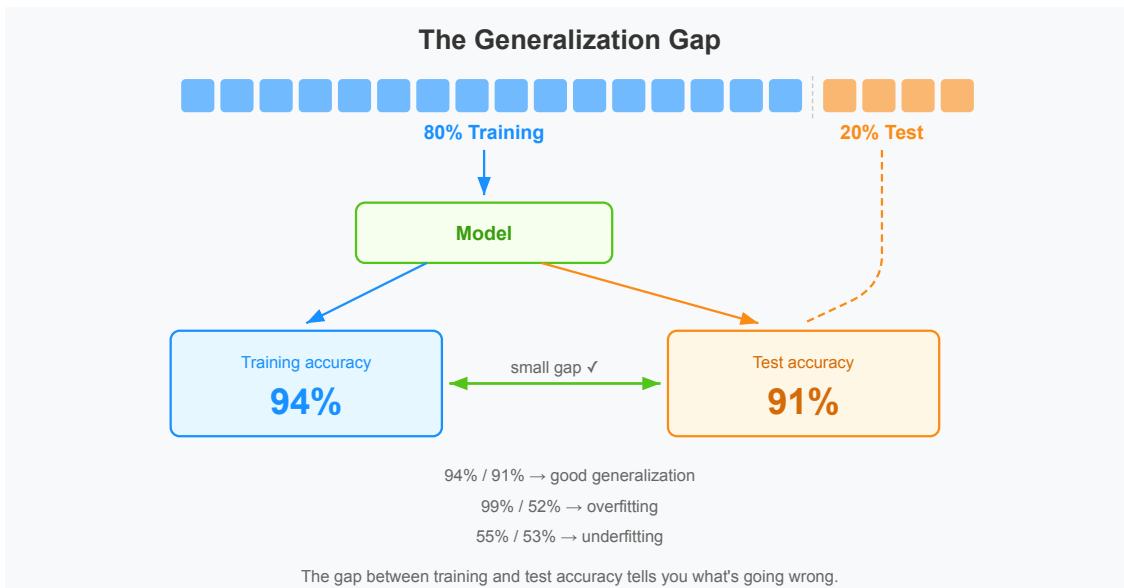


Figure 1.14: The Generalization Gap

A common split is 80% training, 20% testing. The exact ratio isn't sacred, but the principle is: never evaluate a model on data it trained on. That would be like grading students on the homework they copied. You'll see this in practice with companies claiming their language model can solve some benchmark, without mentioning that the benchmark questions were in the training data.

We talked about template matching and nearest neighbors earlier. Think about where they fall on the overfitting/underfitting spectrum. The nearest-neighbor classifier is the ultimate memorizer: it stores every training example and matches against them. On training data, it gets 100% (every image is its own nearest neighbor). But it might not generalize as well to new data. Template matching is the opposite: ten blurry averages can't memorize anything, so it's unlikely to overfit, but it might underfit by being too simple to capture real patterns. The right model is somewhere in between. That's enough theory. Let's build something.

1.7 Hands-On: Recognizing Handwritten Digits

We've been talking about handwritten digits the entire chapter. Now let's build the template matcher we described in the theory and see how it actually performs. We'll use MNIST: 70,000 handwritten digits, each a 28×28 pixel grayscale image labeled 0 through 9. We'll use scikit-learn to download the data, but everything else is pure NumPy. No magic function calls. We build every piece ourselves.

Set up your project:

```
mkdir chapter1
cd chapter1
uv init
```

```
uv add numpy matplotlib scikit-learn pandas
```

Create a file called `digits.py`. Let's start by loading the data and seeing what we're working with:

```
# digits.py
from sklearn.datasets import fetch_openml
import numpy as np
import matplotlib.pyplot as plt

mnist = fetch_openml("mnist_784", version=1, as_frame=False, parser="auto")
X = mnist.data
y = mnist.target.astype(int)

print(f"Number of images: {len(X)}")
print(f"Each image: {X.shape[1]} numbers (28 x 28 pixels)")
print(f"Labels: {np.unique(y)}")
```

Run it with `uv run python digits.py`. A few things to notice here. `X` isn't a regular Python list; it's a NumPy array, a grid of numbers that you can manipulate all at once. `X.shape` tells you its dimensions: 70,000 rows (one per image) by 784 columns (one per pixel). `np.unique(y)` returns the distinct values in the labels: the digits 0 through 9. You could do the same with `set(y)`, but NumPy's version returns them sorted.

Let's look at a few images. Each row of `X` is a flat list of 784 numbers. To display it as a picture, we need to fold it back into a 28×28 grid; that's what `.reshape(28, 28)` does. It doesn't change the data, just how it's arranged:

```
# digits.py
# ...existing code above

fig, axes = plt.subplots(2, 8, figsize=(12, 3))
for i, ax in enumerate(axes.flat):
    ax.imshow(X[i].reshape(28, 28), cmap="gray")
    ax.set_title(str(y[i]), fontsize=12)
    ax.axis("off")
plt.tight_layout()
plt.savefig("samples.png", dpi=150)
plt.show()
```

Some are clean. Some are messy. That variation is the whole reason rules don't work and learning does.

Now let's split the data. We need to shuffle first; the dataset comes sorted by digit, and we don't want the test set to be all 9s. NumPy's `rng.permutation` gives us a shuffled list of indices, and then `X[indices]` picks rows in that new order. This is called *fancy indexing*: instead of grabbing one row with `X[5]`, you hand NumPy a whole list of positions and get back all those rows at once:

```
# digits.py
# ...existing code above

rng = np.random.default_rng(42)
```

```

indices = rng.permutation(len(X))
X, y = X[indices], y[indices]

split = int(0.8 * len(X))
X_train, X_test = X[:split], X[split:]
y_train, y_test = y[:split], y[split:]

print(f"Training: {len(X_train)}, Test: {len(X_test)}")

```

56,000 to train on, 14,000 to test with. The model never sees the test set during training.

Now the template matcher. We described the idea earlier: compute the average image for each digit, then classify by nearest template. Let's wrap it in a class so the logic stays organized. First, the constructor and the fitting method:

```

# digits.py
# ...existing code above

class TemplateMatcher:
    def __init__(self):
        self.templates = None

    def fit(self, X, y):
        self.templates = np.zeros((10, X.shape[1]))
        for digit in range(10):
            mask = (y == digit)
            self.templates[digit] = X[mask].mean(axis=0)

```

`fit` builds the templates. The expression `y == digit` compares every label to `digit` at once; it returns an array of `True` and `False` values, one per image. When you use that array to index `X`, NumPy keeps only the rows where the value is `True`. This is called *boolean masking*, and it's the NumPy equivalent of a list comprehension like `[X[i] for i in range(len(X)) if y[i] == digit]`, just faster. Then `.mean(axis=0)` averages those rows column by column. The `axis=0` means “collapse the rows, keep the columns”: you go from, say, 5,000 images of the digit 7 down to one row of 784 numbers, the average 7.

Now the prediction method:

```

# digits.py, inside the TemplateMatcher class

def predict(self, X):
    predictions = np.zeros(len(X), dtype=int)
    for i, image in enumerate(X):
        distances = np.sqrt(((self.templates - image) ** 2).sum(axis=1))
        predictions[i] = distances.argmin()
    return predictions

```

This computes the Euclidean distance from one image to all ten templates at once. `self.templates - image` subtracts the image from every row of the templates. NumPy automatically lines up the shapes, an operation called *broadcasting*. We square each difference, sum across columns with

`.sum(axis=1)` (axis 1 = across each row), and take the square root. The result is ten distances, one per template. `.argmin()` returns the *index* of the smallest one, that's our prediction.

Let's fit the model:

```
# digits.py
# ...existing code above

model = TemplateMatcher()
model.fit(X_train, y_train)
```

That's our entire model. Ten rows of 784 numbers each. Let's see what these templates look like:

```
# digits.py
# ...existing code above

fig, axes = plt.subplots(1, 10, figsize=(15, 2))
for digit, ax in enumerate(axes):
    ax.imshow(model.templates[digit].reshape(28, 28), cmap="gray")
    ax.set_title(str(digit), fontsize=12)
    ax.axis("off")
plt.suptitle("Average image per digit", fontsize=14)
plt.tight_layout()
plt.savefig("templates.png", dpi=150)
plt.show()
```

There they are: the blurry ghosts we predicted. The 1 is a narrow smear down the center. The 0 is a fuzzy oval. The 8 looks like a blurry snowman. Look at the 4 and the 9; they look eerily similar: both have a vertical stroke on the right side. We can already predict those two will get confused.

Now let's classify the test set and check accuracy:

```
# digits.py
# ...existing code above

predictions = model.predict(X_test)

accuracy = (predictions == y_test).sum() / len(y_test)
print(f"Correct: {(predictions == y_test).sum()} / {len(y_test)}")
print(f"Accuracy: {accuracy:.1%}")
```

Around 81%. Not bad for ten averaged images and a distance calculation. No learning algorithm, no optimization, no gradients. Just averages and subtraction. But 81% means roughly 1 in 5 digits is wrong. Which ones?

Let's build a confusion matrix: a 10×10 grid where row i , column j counts how many times a true i was predicted as j :

```
# digits.py
# ...existing code above
```

```

def confusion_matrix(y_true, y_pred):
    matrix = np.zeros((10, 10), dtype=int)
    for true, pred in zip(y_true, y_pred):
        matrix[true][pred] += 1
    return matrix

confusion = confusion_matrix(y_test, predictions)

```

The diagonal is correct predictions. Everything off the diagonal is a mistake. Let's plot it:

```

# digits.py
# ...existing code above

def plot_confusion(matrix, filename="confusion.png"):
    fig, ax = plt.subplots(figsize=(8, 8))
    im = ax.imshow(matrix, cmap="Blues")
    ax.set_xlabel("Predicted", fontsize=12)
    ax.set_ylabel("Actual", fontsize=12)
    ax.set_title("What gets confused with what", fontsize=14)
    ax.set_xticks(range(10))
    ax.set_yticks(range(10))
    for i in range(10):
        for j in range(10):
            color = "white" if matrix[i, j] > matrix.max() / 2 else "black"
            ax.text(j, i, str(matrix[i, j]),
                    ha="center", va="center", color=color, fontsize=9)
    plt.colorbar(im, ax=ax, shrink=0.8)
    plt.tight_layout()
    plt.savefig(filename, dpi=150)
    plt.show()

plot_confusion(confusion)

```

Look at where the off-diagonal numbers are biggest. 4 gets confused with 9, just as we predicted from the templates. 3 gets confused with 5 and 8 (similar curves). These are exactly the confusions we'd expect from template matching: when two digits have similar average shapes, the model is worse telling them apart.

Let's compute per-digit accuracy to see which digits are easiest and hardest:

```

# digits.py
# ...existing code above

def per_digit_accuracy(y_true, y_pred):
    print("\nPer-digit accuracy:")
    for digit in range(10):
        mask = (y_true == digit)
        digit_acc = (y_pred[mask] == digit).sum() / mask.sum()
        print(f" {digit}: {digit_acc:.1%}")

per_digit_accuracy(y_test, predictions)

```

0 and 1 are easy (their shapes are distinctive). Digits like 5 and 8 are hard (their averages overlap with

other digits). This tells us something about the data itself: some categories are inherently easier to separate than others.

One last experiment. We said that features limit what the model can learn. Let's prove it by giving the model less to work with. We'll train two more template matchers: one that only sees the top half of each image, another that only sees the bottom half.

The notation `X_train[:, start:end]` takes a slice of columns from every row; all images, but only certain pixels. The colon before the comma means "all rows"; `start:end` after the comma selects the columns:

```
# digits.py
# ...existing code above

def test_partial(X_train, y_train, X_test, y_test, start, end, label):
    model = TemplateMatcher()
    model.fit(X_train[:, start:end], y_train)
    preds = model.predict(X_test[:, start:end])
    acc = (preds == y_test).sum() / len(y_test)
    print(f"{label}: {acc:.1%}")
    return acc
```

Now let's run all three:

```
# digits.py
# ...existing code above

print(f"\nFull image:      {accuracy:.1%}")
test_partial(X_train, y_train, X_test, y_test, 0, 392, "Top half only    ")
test_partial(X_train, y_train, X_test, y_test, 392, 784, "Bottom half only")
```

The full image wins. But the top half does slightly better than the bottom. The top portion of digits carries more distinguishing information than you might expect: the presence or absence of a horizontal bar, the shape of a curve's peak, whether strokes converge or diverge. Same model, same algorithm, different features, different results.

This is the entire learning problem in about 50 lines of NumPy. Our model was the simplest thing imaginable, and it got close to 81%. In Chapter 2, we'll build a neural network that learns its own features rather than relying on raw pixel averages, and you'll see exactly why it does better.

1.8 Exercises

1. Our classifier computes Euclidean distance (square root of sum of squared differences). Try **Manhattan distance** instead (sum of absolute differences). Does accuracy change? Why might one distance metric work better than the other for pixel data?
2. Instead of using the average image as each digit's template, try using the **median** image. The median is less sensitive to outliers than the mean. Does it make a difference? Visualize both templates

for a digit like 1, where some people write it with serifs and some without. Which template looks sharper?

3. Our model stores 10 templates. What if we stored more? We mentioned that some digits have multiple distinct writing styles. For each digit, try keeping $k = 3$ templates instead of one: pick 3 random images from the class as starting centers, assign each training image to its nearest center, recompute each center as the mean of the images assigned to it, and repeat that assign-and-recompute cycle 10 times. Now you have 30 templates. Classify by finding the nearest of all 30 and taking its digit label. Does accuracy improve? Which digits benefit most from having multiple templates?
4. Build the **nearest-neighbor classifier** we described: to classify a test image, find the single most similar training image and use its label. What accuracy does it get? It should be higher than template matching. But time how long it takes to classify the full test set versus our 10-template model. What's the tradeoff?
5. Implement a **train/test accuracy comparison**. Compute accuracy on the training set and the test set separately for both the template matcher and your nearest-neighbor classifier from exercise 4. For template matching, the gap should be small (the model is too simple to memorize). What does the gap look like for nearest neighbor? Which model overfits more, and why?
6. We showed that the top half of the image carries slightly more information than the bottom half. Go further: write code that evaluates accuracy using only a single row of pixels (28 features) at each of the 28 possible row positions. Plot accuracy vs row position. Which rows carry the most information? The result tells you where in the image the distinguishing features actually live.