

# STAT 4830: Numerical optimization for data science and ML

Lecture 0: Introduction

Professor Damek Davis

# Course overview

- **Focus:** Numerical optimization for data science and ML
- **Tools:** PyTorch, Python, occasional use of other libraries
- **LLM Policy:** Use them.
- **Deliverables:** HW, final project
  - **Homework:** 0-5 assignments.
  - **Final Project:** Incrementally developed throughout semester (more later)

# Do you have access to and experience with LLMs?

*Modern optimization and ML development fundamentally requires AI assistance for:*

- Debugging complex numerical code*
- Exploring implementation alternatives*
- Understanding mathematical concepts*
- Rapid prototyping of algorithms*

*Without these tools, students would:*

- 1. Struggle with industry-standard development practices*
- 2. Miss critical job-market skills*
- 3. Face unnecessary friction in learning core concepts*

*The course focuses on practical implementation - AI assistance isn't optional, its core to modern development workflows.*

# Prerequisites

- Basic calculus and linear algebra (Math 2400)
- Basic probability (Stat 4300)
- Python programming experience
- No advanced optimization/ML background needed

# Why PyTorch?

- Modern auto-differentiation frameworks drive deep learning success
- Enables rapid experimentation with:
  - New model architectures and
  - Novel optimization algorithms
- More flexible than traditional solver-based tools

# Optimization Approaches Compared

| CLASSICAL TOOLS  | VS | MODERN TOOLS   |
|--|----|--|
| WHEN TO USE: <ul style="list-style-type: none"><li>• Small/Medium Scale</li><li>• Structured Problems</li><li>• Need Formal Guarantees</li></ul> |    | WHEN TO USE: <ul style="list-style-type: none"><li>• Large-Scale Data</li><li>• Unstructured Problems</li><li>• Need Speed &amp; Scale</li></ul> |
| INPUT REQUIREMENTS: <ul style="list-style-type: none"><li>• DCP Rules</li><li>• Convex Functions</li><li>• Standard Form</li></ul>               |    | INPUT REQUIREMENTS: <ul style="list-style-type: none"><li>• Raw Python Code</li><li>• Any Function Type</li><li>• Black Box OK</li></ul>         |
| GUARANTEES: <ul style="list-style-type: none"><li>• Global Optimality</li><li>• Convergence Rates</li><li>• Optimality Certificates</li></ul>    |    | CAPABILITIES: <ul style="list-style-type: none"><li>• Handle Any Problem</li><li>• Fast Development</li><li>• Rapid Iteration</li></ul>          |
| WORKFLOW: <ol style="list-style-type: none"><li>1. Format Problem</li><li>2. Verify Convexity</li><li>3. Solve Exactly</li></ol>                 |    | WORKFLOW: <ol style="list-style-type: none"><li>1. Write Code</li><li>2. Train Model</li><li>3. Debug &amp; Iterate</li></ol>                    |
| TOOLS: CVX, MOSEK  |    | TOOLS: PyTorch, JAX  |

- HYBRID APPROACHES:
- Use Structure Where Possible
  - Add Flexibility Where Needed
  - Choose Based on Requirements

# Preview: spam classification

Let's start with a practical example:

- How do we automatically filter spam emails?
- Demonstrates core optimization concepts
- Shows PyTorch in action

# How computers read email

```
email1 = """  
Subject: URGENT! You've won $1,000,000!!!  
Dear Friend! Act NOW to claim your PRIZE money!!!  
"""  
  
email2 = """  
Subject: Team meeting tomorrow  
Hi everyone, Just a reminder about our 2pm sync.  
"""
```

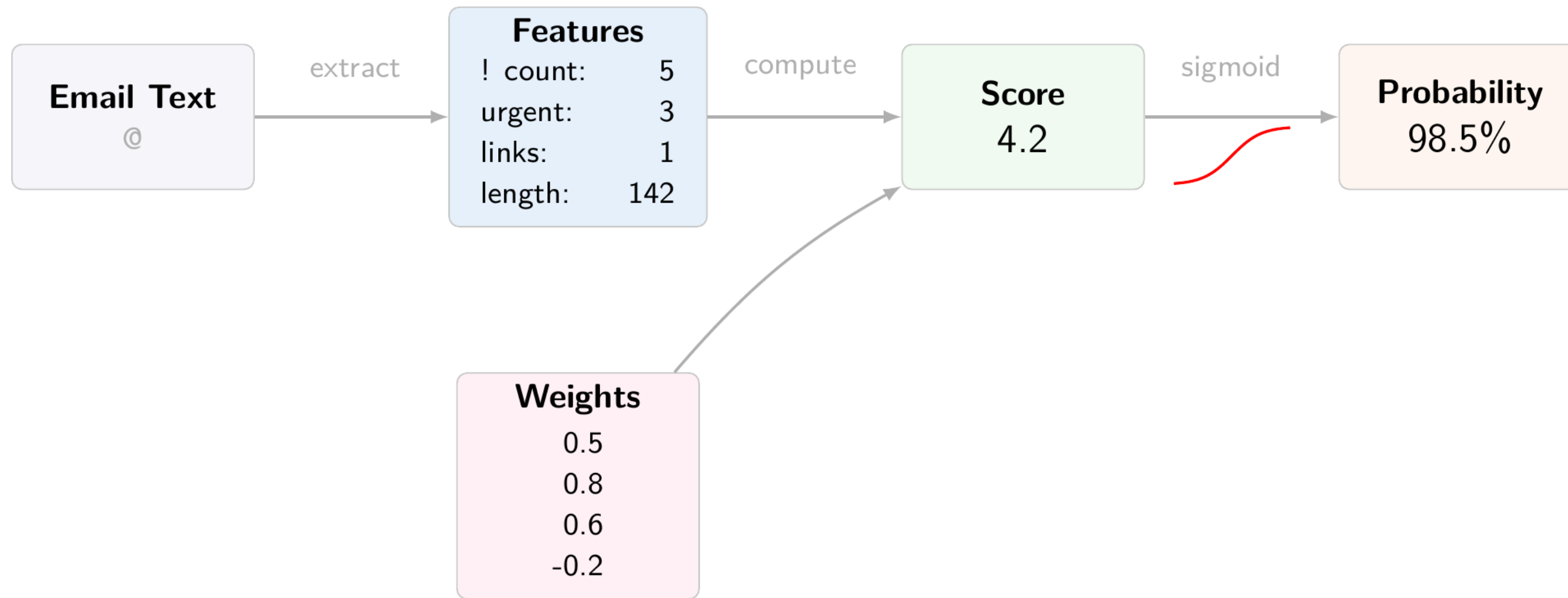


# Feature extraction

Convert text to numbers:

```
def extract_features(email):  
    features = {  
        'exclamation_count': email.count('!'),  
        'urgent_words': len(['urgent', 'act now', 'prize']  
                             & set(email.lower().split())),  
        'suspicious_links': len([link for link in email.split()  
                                 if 'www' in link]),  
        'time_sent': email.timestamp.hour,  
        'length': len(email)  
    }  
    return features
```

# Classification process

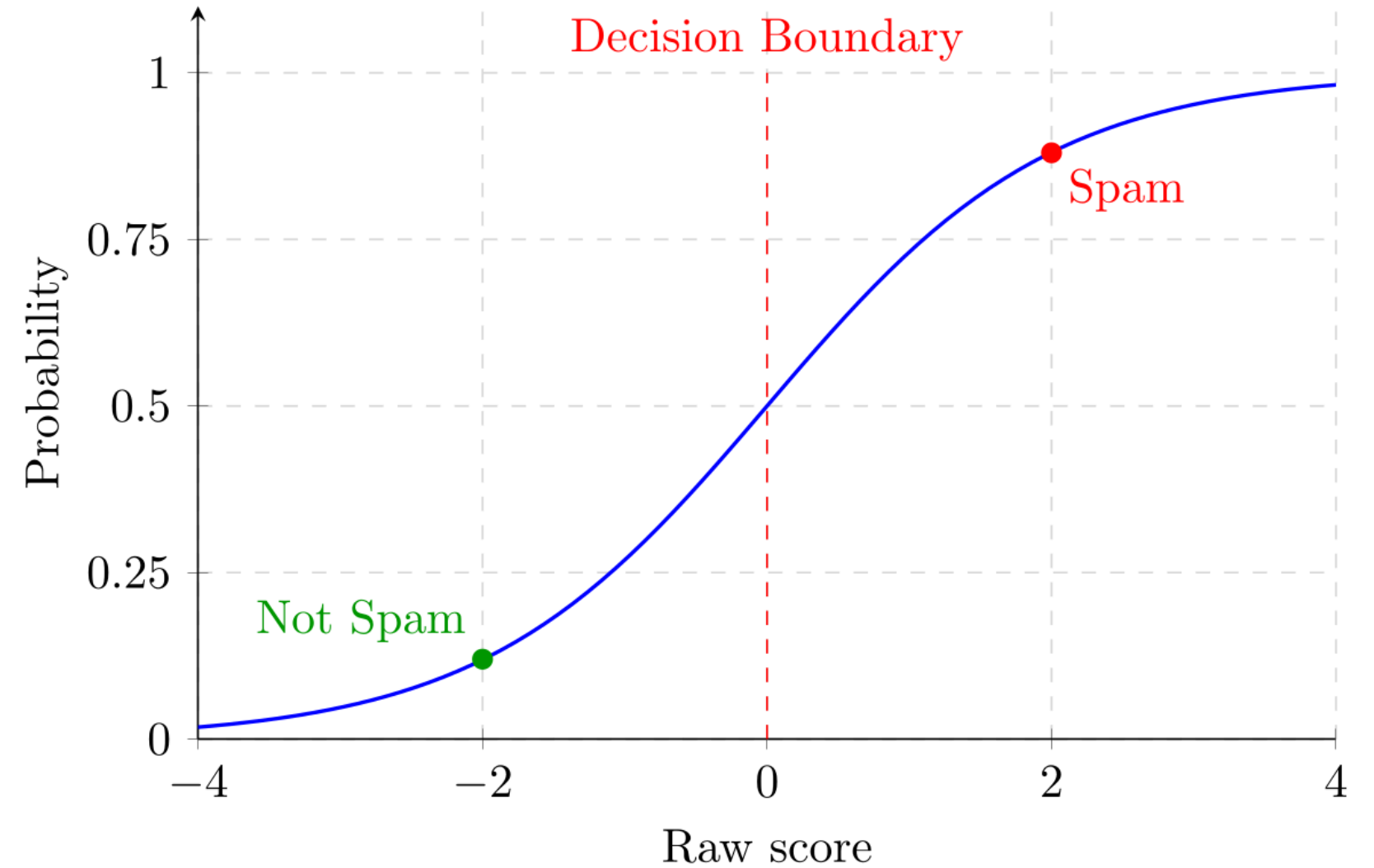


1. Extract numeric features
2. Multiply by weights
3. Sum weighted features
4. Convert to probability

# The sigmoid function

Converts any number into a probability (0-1):

```
def sigmoid(x):  
    return 1 / (1 + torch.exp(-x))
```



# Mathematical formulation

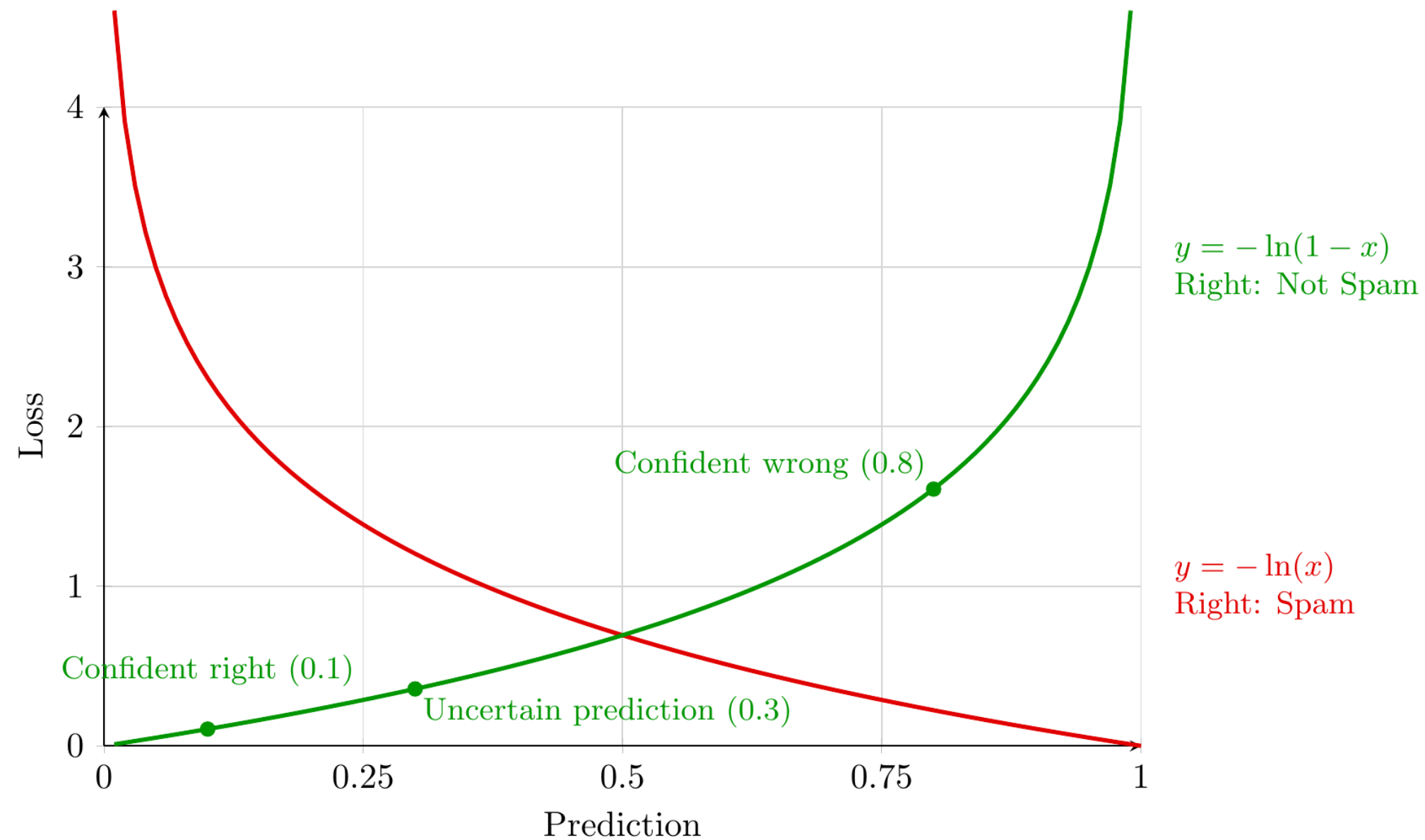
Our optimization problem:

$$\min_w \frac{1}{n} \sum_{i=1}^n \left[ -y_i \log(\sigma(x_i^\top w)) - (1 - y_i) \log(1 - \sigma(x_i^\top w)) \right]$$

Where:

- $w$  = weights vector
- $x_i$  = feature vector
- $y_i$  = true label (0/1)
- $\sigma$  = sigmoid function

# Cross-entropy loss

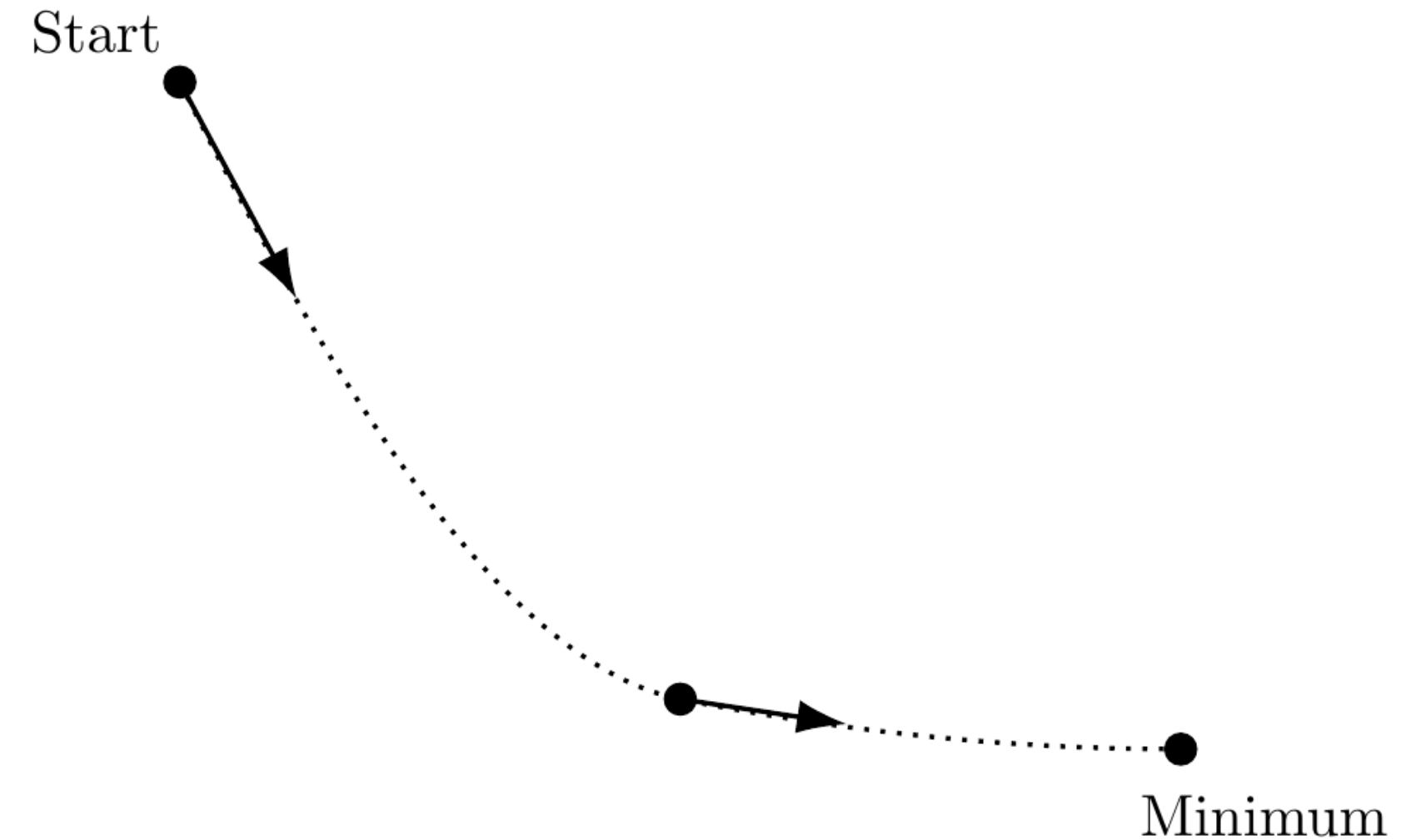


- Penalizes wrong predictions
- Rewards confident correct predictions
- Creates balanced learning

# How Gradient Descent Works

The optimization process works like hiking:

1. Look around you (measure gradient)
2. Take a step downhill
3. Repeat until you reach the bottom



# The optimization loop

Each iteration:

1. **Measure** how well current weights classify emails
2. **Calculate** gradient (direction of steepest error reduction)
3. **Update** weights by stepping in this direction
4. **Repeat** until convergence

The learning rate controls step size:

- Too small → slow progress
- Too large → overshooting

# PyTorch: What, how, and why

**What:** Modern framework for optimization and deep learning

**How:**

- Tracks operations in a computational graph
- Automatically computes gradients
- Enables parallel computation (CPU/GPU)

**Why:**

- Automates the hardest part (gradients)
- Makes experimentation fast
- Scales from simple to complex models



# Inside PyTorch: Tensors and autograd

```
# Tensors: The building blocks  
x = torch.tensor([1.0, 2.0], requires_grad=True)  
y = x * 2  
z = y.sum()  
  
# Automatic differentiation  
z.backward() # Computes gradients  
print(x.grad) # Shows  $\partial z / \partial x$ 
```

PyTorch builds a graph of operations, enabling automatic gradient computation.

# Implementation in PyTorch

```
# Initialize
weights = torch.randn(5, requires_grad=True)
learning_rate = 0.01

for _ in range(1000):
    # Forward pass
    predictions = spam_score(features, weights)
    loss = cross_entropy_loss(predictions, true_labels)

    # Backward pass
    loss.backward()

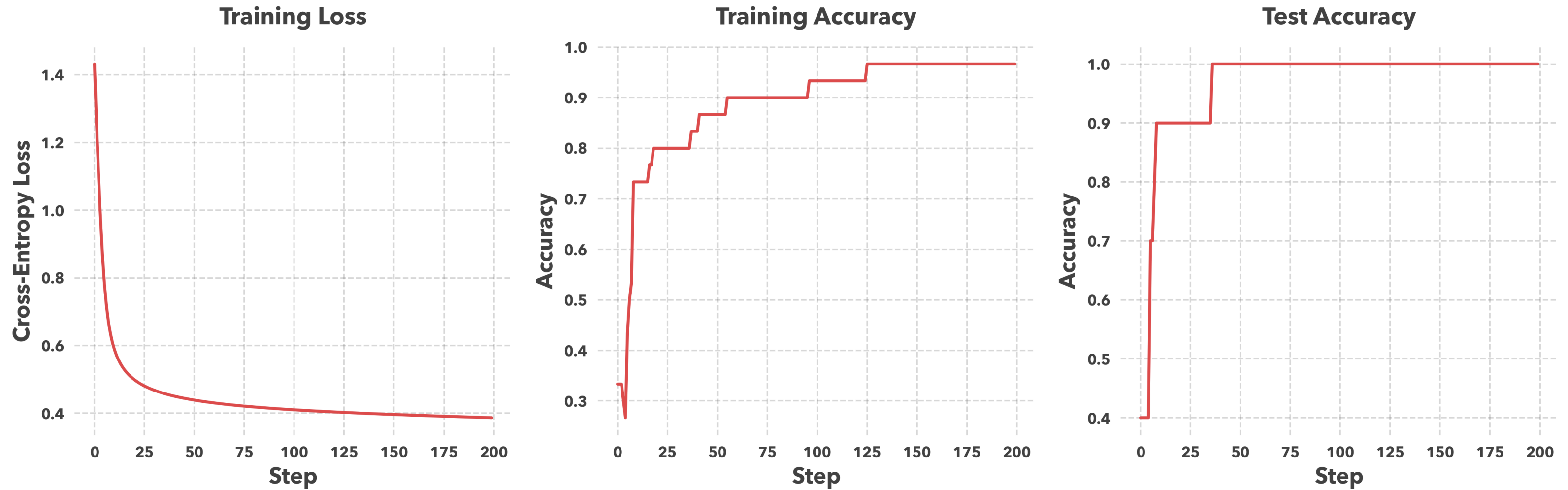
    # Update weights
    with torch.no_grad():
        weights -= learning_rate * weights.grad
        weights.grad.zero_()
```

# Try it yourself!

 Open in Colab

- Complete implementation in the notebook
- Experiment with different learning rates
- See how the loss changes during training
- Test the model on new emails

# Training results



Three key metrics:

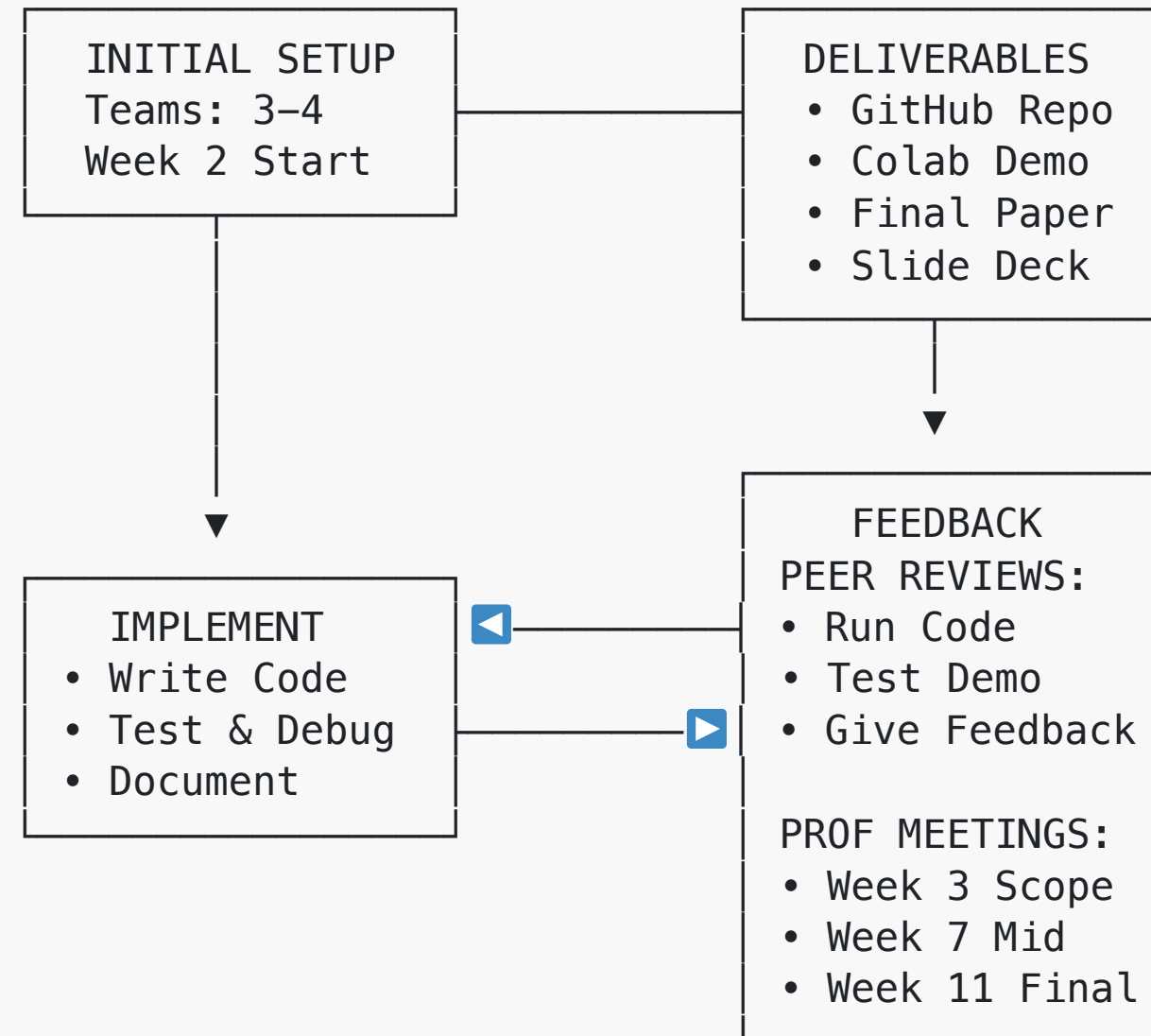
- **Loss** and **Training accuracy**: Performance on known data.
- **Test accuracy**: Performance on new emails

# Course structure

1. Linear algebra & direct methods
2. Problem formulations & classical software
3. Calculus for optimization
4. Automatic differentiation & PyTorch
5. First-order methods
6. Second-order methods
7. Advanced topics
8. Modern deep learning practice

# Final Project Structure

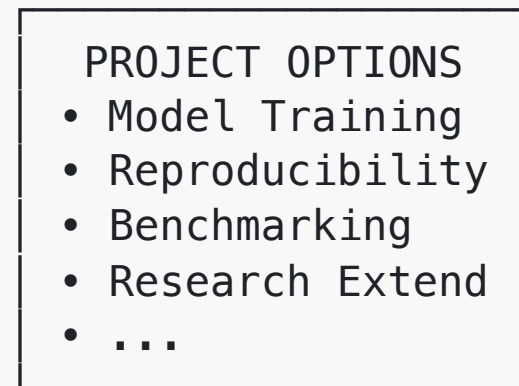
## ITERATIVE DEVELOPMENT PROCESS



## DEVELOPMENT WITH LLMs

- Write & review reports, documentation
- Develop & test code (verify outputs!)
- Regular commits with clear documentation

## PROJECT COMPONENTS



## BIWEEKLY SCHEDULE

Week 3: Report  
Week 4: Slides Draft  
Week 5: Report  
Week 6: Slides Draft  
Week 7: Report  
Week 8: ⚡LIGHTNING TALK⚡  
Week 9: Report  
Week 10: Slides Draft  
Week 11: Report  
Week 12: Slides Draft  
Week 13: Final Report  
Week 14: Final Present

# Learning outcomes

By course end, you'll be able to:

1. Model real problems as optimization problems
2. Select appropriate algorithms
3. Implement solutions in PyTorch
4. Apply optimization to practical problems
5. Conduct optimization research

# Getting started

- Review the syllabus
- Set up Python environment
- Try the [Colab notebook](#)
- Start thinking about project ideas



# Questions?

- Course website: <https://damek.github.io/STAT-4830/>
- Office hours: Listed on the course website
- Email: [damek@wharton.upenn.edu](mailto:damek@wharton.upenn.edu)
- Discord: Check email for invite.

