CSC 4315: INTRODUCTION TO ARTIFICIAL INTELLIGENCE

# Week 4, Lecture 7
# Supervised Learning and Linear Regression from Scratch

## Lecture Overview

This lecture introduces the fundamental concepts of supervised learning and implements linear regression from scratch using NumPy. Understanding these core concepts provides the foundation for all machine learning algorithms we'll encounter in this course.

**Topics Covered:**

- What is machine learning? Types of learning
- Supervised learning: regression vs classification
- Linear regression: hypothesis, parameters, predictions
- Cost function: Mean Squared Error (MSE)
- Implementing linear regression from scratch
- Model evaluation and interpretation

# 1. Introduction to Machine Learning

## 1.1 What is Machine Learning?

Machine learning is a subset of artificial intelligence that enables computers to learn patterns from data without being explicitly programmed. Instead of writing rules manually, we provide data and let algorithms discover the underlying patterns.

```
import numpy as np
import matplotlib.pyplot as plt

print("Machine Learning Overview")
print("="*70)

# Traditional Programming vs Machine Learning
print("Traditional Programming:")
print("  Input: Data + Rules -> Output")
print("\nMachine Learning:")
print("  Input: Data + Output -> Rules (Model)")

# Types of Machine Learning
print("\nTypes of Machine Learning:")
print("1. Supervised Learning: Learn from labeled data")
print("   - Regression: Predict continuous values (e.g., house prices)")
print("   - Classification: Predict categories (e.g., spam/not spam)")
print("2. Unsupervised Learning: Find patterns in unlabeled data")
print("   - Clustering, dimensionality reduction")
print("3. Reinforcement Learning: Learn through trial and error")
print("   - Games, robotics, autonomous systems")
```

```
Machine Learning Overview
======================================================================
Traditional Programming:
  Input: Data + Rules -> Output

Machine Learning:
  Input: Data + Output -> Rules (Model)

Types of Machine Learning:
1. Supervised Learning: Learn from labeled data
   - Regression: Predict continuous values (e.g., house prices)
   - Classification: Predict categories (e.g., spam/not spam)
2. Unsupervised Learning: Find patterns in unlabeled data
   - Clustering, dimensionality reduction
3. Reinforcement Learning: Learn through trial and error
   - Games, robotics, autonomous systems
```

### 1.2 Supervised Learning Framework

In supervised learning, we have a dataset with input features (X) and corresponding target values (y). Our goal is to learn a function that maps inputs to outputs.

```
# Supervised Learning Framework
print("Supervised Learning Framework")
print("="*70)

# Key terminology
print("Key Terminology:")
print("  - Features (X): Input variables used for prediction")
print("  - Target (y): Output variable we want to predict")
print("  - Training data: Data used to train the model")
print("  - Test data: Data used to evaluate the model")
print("  - Model: The learned function that maps X -> y")
print("  - Parameters: Values learned during training")
print("  - Hyperparameters: Values set before training")

# Example: Predicting house prices
print("\nExample: Predicting House Prices")
print("-" * 40)

# Simple dataset: [size in sqft] -> price
np.random.seed(42)
sizes = np.array([1000, 1200, 1500, 1800, 2000, 2200, 2500])
prices = np.array([200, 250, 300, 350, 400, 430, 500])  # in thousands

print(f"House sizes (sqft): {sizes}")
print(f"Prices ($1000s):    {prices}")
print(f"\nNumber of samples: {len(sizes)}")
print(f"Number of features: 1 (house size)")
print(f"Target variable: price")
```

```
Supervised Learning Framework
======================================================================
Key Terminology:
  - Features (X): Input variables used for prediction
  - Target (y): Output variable we want to predict
  - Training data: Data used to train the model
  - Test data: Data used to evaluate the model
  - Model: The learned function that maps X -> y
  - Parameters: Values learned during training
  - Hyperparameters: Values set before training

Example: Predicting House Prices
----------------------------------------
House sizes (sqft): [1000 1200 1500 1800 2000 2200 2500]
Prices ($1000s):    [200 250 300 350 400 430 500]

Number of samples: 7
Number of features: 1 (house size)
Target variable: price
```

## 2. Linear Regression

### 2.1 The Linear Hypothesis

Linear regression assumes a linear relationship between input features and the target. For a single feature, the hypothesis is: h(x) = w*x + b, where w is the weight (slope) and b is the bias (y-intercept).

```
# Linear Regression Hypothesis
print("Linear Regression Hypothesis")
print("="*70)

# Mathematical formulation
print("Simple Linear Regression:")
print("  Hypothesis: h(x) = w * x + b")
print("  Where:")
print("    - w (weight): slope of the line")
print("    - b (bias): y-intercept")
print("    - x: input feature")
print("    - h(x): predicted output")

print("\nMultiple Linear Regression:")
print("  Hypothesis: h(x) = w1*x1 + w2*x2 + ... + wn*xn + b")
print("  Or in vector form: h(x) = w^T * x + b")

# Visualizing different hypotheses
x = np.linspace(0, 10, 100)

# Different parameter choices
params = [
    (2, 1, "w=2, b=1"),
    (1, 3, "w=1, b=3"),
    (0.5, 5, "w=0.5, b=5")
]

print("\nDifferent hypotheses produce different lines:")
for w, b, label in params:
    y_pred = w * x + b
    print(f"  {label}: h(0)={b}, h(10)={w*10+b}")
```

```
Linear Regression Hypothesis
======================================================================
Simple Linear Regression:
  Hypothesis: h(x) = w * x + b
  Where:
    - w (weight): slope of the line
    - b (bias): y-intercept
    - x: input feature
    - h(x): predicted output

Multiple Linear Regression:
  Hypothesis: h(x) = w1*x1 + w2*x2 + ... + wn*xn + b
  Or in vector form: h(x) = w^T * x + b

Different hypotheses produce different lines:
  w=2, b=1: h(0)=1, h(10)=21
  w=1, b=3: h(0)=3, h(10)=13
  w=0.5, b=5: h(0)=5, h(10)=10.0
```

## 2.2 Making Predictions

```
# Making predictions with linear regression
print("Making Predictions")
print("="*70)

# Our data
sizes = np.array([1000, 1200, 1500, 1800, 2000, 2200, 2500])
prices = np.array([200, 250, 300, 350, 400, 430, 500])

# Let's try some parameter values (we'll learn how to find optimal ones later)
w = 0.2    # price increases by $200 for every sqft
b = 0      # base price

# Predict prices
predictions = w * sizes + b

print(f"Parameters: w = {w}, b = {b}")
print(f"\nPredictions vs Actual:")
print("-" * 50)
print(f"{'Size':>8} | {'Actual':>10} | {'Predicted':>10} | {'Error':>10}")
print("-" * 50)
for size, actual, pred in zip(sizes, prices, predictions):
    error = actual - pred
    print(f"{size:>8} | ${actual:>9} | ${pred:>9.0f} | ${error:>9.0f}")

# Calculate total error
total_error = np.sum(prices - predictions)
print(f"\nTotal error: ${total_error:.0f}")
print(f"Average error: ${total_error/len(prices):.0f}")
```

```
Making Predictions
======================================================================
Parameters: w = 0.2, b = 0

Predictions vs Actual:
--------------------------------------------------
    Size |     Actual |  Predicted |      Error
--------------------------------------------------
    1000 |      $200 |      $200 |        $0
    1200 |      $250 |      $240 |       $10
    1500 |      $300 |      $300 |        $0
    1800 |      $350 |      $360 |      $-10
    2000 |      $400 |      $400 |        $0
    2200 |      $430 |      $440 |      $-10
    2500 |      $500 |      $500 |        $0

Total error: $-10
Average error: $-1
```

## 3. Cost Function: Measuring Error

### 3.1 Mean Squared Error (MSE)

The cost function measures how well our model fits the data. For linear regression, we use Mean Squared Error (MSE), which is the average of squared differences between predictions and actual values.

```
# Cost Function: Mean Squared Error
print("Cost Function: Mean Squared Error (MSE)")
print("="*70)

# Mathematical definition
print("MSE Formula:")
print("  J(w, b) = (1/m) * sum((h(x_i) - y_i)^2)")
print("  Where:")
print("    - m: number of training examples")
print("    - h(x_i): prediction for example i")
print("    - y_i: actual value for example i")

def compute_mse(y_true, y_pred):
    """Compute Mean Squared Error"""
    m = len(y_true)
    mse = np.sum((y_pred - y_true) ** 2) / m
    return mse

def compute_rmse(y_true, y_pred):
    """Compute Root Mean Squared Error"""
    return np.sqrt(compute_mse(y_true, y_pred))

# Test with our predictions
print("\nComputing MSE for our model:")
print(f"Actual prices:    {prices}")
print(f"Predicted prices: {predictions}")

errors = predictions - prices
squared_errors = errors ** 2

print(f"\nErrors:         {errors}")
print(f"Squared errors: {squared_errors}")

mse = compute_mse(prices, predictions)
rmse = compute_rmse(prices, predictions)

print(f"\nMSE:  {mse:.2f}")
print(f"RMSE: {rmse:.2f} (in same units as price)")
```

```
Cost Function: Mean Squared Error (MSE)
======================================================================
MSE Formula:
  J(w, b) = (1/m) * sum((h(x_i) - y_i)^2)
  Where:
    - m: number of training examples
    - h(x_i): prediction for example i
    - y_i: actual value for example i

Computing MSE for our model:
Actual prices:    [200 250 300 350 400 430 500]
Predicted prices: [200. 240. 300. 360. 400. 440. 500.]

Errors:         [  0. -10.   0.  10.   0.  10.   0.]
Squared errors: [  0. 100.   0. 100.   0. 100.   0.]
```

```
MSE:  42.86
RMSE: 6.55 (in same units as price)
```

## 3.2 Why Squared Error?

```
# Why we use squared error
print("Why Squared Error?")
print("="*70)

# Compare different error metrics
y_true = np.array([100, 200, 300])
y_pred = np.array([90, 210, 280])

errors = y_pred - y_true

print(f"True values:  {y_true}")
print(f"Predictions:  {y_pred}")
print(f"Errors:       {errors}")

# Different ways to aggregate errors
print("\nDifferent error metrics:")

# 1. Sum of errors (bad - can cancel out)
sum_errors = np.sum(errors)
print(f"  Sum of errors: {sum_errors} (cancels out!)")

# 2. Sum of absolute errors
sum_abs_errors = np.sum(np.abs(errors))
mae = np.mean(np.abs(errors))
print(f"  Mean Absolute Error (MAE): {mae:.2f}")

# 3. Sum of squared errors
sum_sq_errors = np.sum(errors ** 2)
mse = np.mean(errors ** 2)
print(f"  Mean Squared Error (MSE): {mse:.2f}")

print("\nWhy MSE is preferred:")
print("  1. Squared errors are always positive (no cancellation)")
print("  2. Penalizes large errors more than small errors")
print("  3. Differentiable everywhere (needed for gradient descent)")
print("  4. Has nice mathematical properties")
```

```
Why Squared Error?
======================================================================
True values:  [100 200 300]
Predictions:  [ 90 210 280]
Errors:       [-10  10 -20]

Different error metrics:
  Sum of errors: -20 (cancels out!)
  Mean Absolute Error (MAE): 13.33
  Mean Squared Error (MSE): 166.67

Why MSE is preferred:
  1. Squared errors are always positive (no cancellation)
  2. Penalizes large errors more than small errors
  3. Differentiable everywhere (needed for gradient descent)
  4. Has nice mathematical properties
```

## 4. Implementing Linear Regression from Scratch

### 4.1 The Closed-Form Solution (Normal Equation)

For linear regression, there's an analytical solution that directly computes the optimal parameters. This is called the Normal Equation.

```
# Normal Equation: Closed-form solution
print("Normal Equation (Closed-Form Solution)")
print("="*70)

# Mathematical derivation
print("For simple linear regression y = wx + b:")
print("  w = sum((x - x_mean)(y - y_mean)) / sum((x - x_mean)^2)")
print("  b = y_mean - w * x_mean")

def linear_regression_closed_form(X, y):
    """
    Compute optimal w and b using closed-form solution

    Args:
        X: Input features (1D array)
        y: Target values (1D array)

    Returns:
        w: Weight (slope)
        b: Bias (intercept)
    """
    # Compute means
    x_mean = np.mean(X)
    y_mean = np.mean(y)

    # Compute weight (slope)
    numerator = np.sum((X - x_mean) * (y - y_mean))
    denominator = np.sum((X - x_mean) ** 2)
    w = numerator / denominator

    # Compute bias (intercept)
    b = y_mean - w * x_mean

    return w, b

# Apply to our house price data
sizes = np.array([1000, 1200, 1500, 1800, 2000, 2200, 2500])
prices = np.array([200, 250, 300, 350, 400, 430, 500])

w_opt, b_opt = linear_regression_closed_form(sizes, prices)

print(f"\nOptimal parameters found:")
print(f"  w (slope): {w_opt:.6f}")
print(f"  b (intercept): {b_opt:.4f}")
print(f"\nInterpretation:")
print(f"  - Each additional sqft adds ${w_opt*1000:.2f} to price")
print(f"  - Base price (at 0 sqft): ${b_opt*1000:.2f}")
```

```
Normal Equation (Closed-Form Solution)
======================================================================
For simple linear regression y = wx + b:
  w = sum((x - x_mean)(y - y_mean)) / sum((x - x_mean)^2)
  b = y_mean - w * x_mean

Optimal parameters found:
  w (slope): 0.193548
```

```
  b (intercept): 23.0645

Interpretation:
  - Each additional sqft adds $193.55 to price
  - Base price (at 0 sqft): $23064.52
```

## 4.2 Complete Linear Regression Class

```
# Complete Linear Regression Implementation
print("Complete Linear Regression Implementation")
print("="*70)

class LinearRegression:
    """Simple Linear Regression from scratch"""

    def __init__(self):
        self.w = None  # Weight
        self.b = None  # Bias

    def fit(self, X, y):
        """
        Train the model using closed-form solution

        Args:
            X: Training features (n_samples,)
            y: Training targets (n_samples,)
        """
        # Compute means
        x_mean = np.mean(X)
        y_mean = np.mean(y)

        # Compute optimal parameters
        numerator = np.sum((X - x_mean) * (y - y_mean))
        denominator = np.sum((X - x_mean) ** 2)

        self.w = numerator / denominator
        self.b = y_mean - self.w * x_mean

        return self

    def predict(self, X):
        """Make predictions"""
        return self.w * X + self.b

    def score(self, X, y):
        """
        Compute R^2 score (coefficient of determination)
        """
        y_pred = self.predict(X)

        # Total sum of squares
        ss_tot = np.sum((y - np.mean(y)) ** 2)

        # Residual sum of squares
        ss_res = np.sum((y - y_pred) ** 2)

        # R^2 score
        r2 = 1 - (ss_res / ss_tot)
        return r2

# Train and evaluate
model = LinearRegression()
model.fit(sizes, prices)

print(f"Model parameters:")
print(f"  Weight (w): {model.w:.6f}")
print(f"  Bias (b): {model.b:.4f}")

# Make predictions
```

```
predictions = model.predict(sizes)
print(f"\nPredictions: {np.round(predictions, 2)}")
print(f"Actual:      {prices}")

# Evaluate
r2 = model.score(sizes, prices)
print(f"\nR^2 Score: {r2:.4f}")
print(f"(R^2 of 1.0 means perfect fit)")
```

```
Complete Linear Regression Implementation
======================================================================
Model parameters:
  Weight (w): 0.193548
  Bias (b): 23.0645

Predictions: [216.61 255.29 313.32 371.35 409.68 448.35 506.38]
Actual:      [200 250 300 350 400 430 500]

R^2 Score: 0.9915
(R^2 of 1.0 means perfect fit)
```

## 5. Model Evaluation

### 5.1 Evaluation Metrics

```
# Model Evaluation Metrics
print("Model Evaluation Metrics")
print("="*70)

def evaluate_regression(y_true, y_pred):
    """Compute common regression metrics"""

    n = len(y_true)

    # Mean Absolute Error (MAE)
    mae = np.mean(np.abs(y_true - y_pred))

    # Mean Squared Error (MSE)
    mse = np.mean((y_true - y_pred) ** 2)

    # Root Mean Squared Error (RMSE)
    rmse = np.sqrt(mse)

    # R^2 Score
    ss_res = np.sum((y_true - y_pred) ** 2)
    ss_tot = np.sum((y_true - np.mean(y_true)) ** 2)
    r2 = 1 - (ss_res / ss_tot)

    return {
        'MAE': mae,
        'MSE': mse,
        'RMSE': rmse,
        'R2': r2
    }

# Evaluate our model
metrics = evaluate_regression(prices, predictions)

print("Regression Metrics:")
print("-" * 40)
for name, value in metrics.items():
    print(f"  {name}: {value:.4f}")

print("\nInterpretation:")
print(f"  - MAE: On average, predictions are off by ${metrics['MAE']:.2f}k")
print(f"  - RMSE: Typical prediction error is ${metrics['RMSE']:.2f}k")
print(f"  - R^2: Model explains {metrics['R2']*100:.2f}% of variance")
```

```
Model Evaluation Metrics
======================================================================
Regression Metrics:
----------------------------------------
  MAE: 11.0968
  MSE: 168.7601
  RMSE: 12.9906
  R2: 0.9915

Interpretation:
  - MAE: On average, predictions are off by $11.10k
  - RMSE: Typical prediction error is $12.99k
  - R^2: Model explains 99.15% of variance
```

## 5.2 Making New Predictions

```
# Making predictions on new data
print("Making New Predictions")
print("="*70)

# Predict for new house sizes
new_sizes = np.array([1100, 1600, 1900, 2300, 3000])
new_predictions = model.predict(new_sizes)

print("Predictions for new houses:")
print("-" * 40)
for size, pred in zip(new_sizes, new_predictions):
    print(f"  {size} sqft -> ${pred:.2f}k (${pred*1000:.0f})")

# Confidence in predictions
print("\nNote on extrapolation:")
print(f"  Training data range: {sizes.min()} - {sizes.max()} sqft")
print(f"  - Predictions within this range are more reliable")
print(f"  - Predictions outside (e.g., 3000 sqft) may be less accurate")
```

```
Making New Predictions
======================================================================
Predictions for new houses:
----------------------------------------
  1100 sqft -> $235.97k ($235968)
  1600 sqft -> $332.74k ($332742)
  1900 sqft -> $390.81k ($390806)
  2300 sqft -> $468.23k ($468226)
  3000 sqft -> $603.71k ($603710)

Note on extrapolation:
  Training data range: 1000 - 2500 sqft
  - Predictions within this range are more reliable
  - Predictions outside (e.g., 3000 sqft) may be less accurate
```

## Summary

**Key Takeaways:**

- Machine learning learns patterns from data instead of explicit programming
- Supervised learning uses labeled data (input-output pairs)
- Linear regression models the relationship: y = wx + b
- MSE (Mean Squared Error) measures prediction quality
- The Normal Equation provides a closed-form solution
- R^2 score indicates how well the model explains variance
- Models should be evaluated on data they haven't seen

**Practice Exercises:**

1. Implement linear regression for multiple features (multivariate)
2. Create a dataset and fit a linear regression model
3. Plot the regression line against the data points
4. Compute all evaluation metrics for your model
5. Experiment with feature scaling and observe its effects